Tietorakenteet ja algoritmit

(TKT 20001)

Kevät 2019

Jyrki Kivinen

Perustuu materiaaliin, jota ovat kehittäneet mm.
Antti Laaksonen, Matti Nykänen, Matti Luukkainen ja Patrik Floréen

Sisällys

1. Johdanto ja kertausta

Kurssin asema opetusohjelmassa

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

Kurssin suorittaminen

- Kurssin maksimipistemäärä on 60 pistettä:
  - 1. kurssikoe periodin III koeviikolla: 20 pistettä
  - 2. kurssikoe periodin IV koeviikolla: 20 pistettä
  - laskuharjoitukset: 20 pistettä.

- Korkeimman arvosanan 5/5 raja on (noin) 50 pistettä, arvosanan 1/5 raja 30 pistettä.

- Hyväksyttyyn suoritukseen vaaditaan lisäksi vähintään puolet koepisteistä (eli kahdesta kurssikokeesta yhteensä 20 pistettä).

- Laskuharjoituksista saatavat kurssipisteet määräytyvät ratkaistujen tehtävien lukumäärän perusteella lineaarisesti niin, että ratkaisemalla 90 prosenttia kaikista tehtävistä saa maksimimäärän 20 pistettä.

Toiminta kurssilla

- Laskuharjoitusten ratkaisut palautetaan sähköisesti käyttäen sekä Moodlea että TMC-järjestelmää.
- Valitettavasti Moodlea ja TMC:tä ei saa toimimaan yhdessä, joten opiskelijan pitää joka viikko ratkaista tehtäviä erikseen kummassakin. Samoin kirjanpito ratkaistuista tehtävistä on kummassakin järjestelmässä erikseen.
- Palautusten määräaika on kurssiviikosta 2 alkaen tiistaisin klo 10.00.
- Laskuharjoitusten tekemiseen annetaan ohjausta Algoritmipajassa, jonka ohjausajat on linkitetty kurssisivulle. Pajaan osallistuminen on erittäin suositeltavaa, mutta vapaaehtoista.
- Myös luennot ovat vapaaehtoisia.
- Kurssin Moodle-aluetta ja etenkin sen Uutiset-osiota pitää seurata.

DEFA (Digital Education for All)

- DEFA on kansallinen hanke, jonka tarkoituksena on mm. tarjota tkt:n ensimmäisen vuoden opinnot kaikille avoimina suomenkielisinä verkkokursseina.
- Tämän kevään Tietorakenteet ja algoritmit on osa DEFAn pilotointivaihetta.
- Kurssilla ei vielä ole täyttä verkkototeutusta, sillä kurssikokeisiin pitää osallistua paikan päällä.
- DEFA-opiskelijat saavat pisteitä tasan samoista suoritteista kuin HY:n tutkinto-opiskelijat.
- Sunnuntaihin 13.1. mennessä kurssille oli ilmoittautunut
  HY:n tutkinto-opiskelijoita: 233
  DEFA-opiskelijoita: 133.
- DEFA-opiskelijat voivat halutessaan osallistua luennoille ja pajoihin, paitsi jos tila loppuu tms.

Esitietovaatimuksisista

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

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

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

Lähdemateriaali

Kurssille on äskettäin valmistunut oppikirja

> Antti Laaksonen: *Tietorakenteet ja algoritmit*. Helsinki 2018.

Kirja on vapaasti saatavana PDF-muodossa; linkki on kurssin verkkosivulla. Kurssilla etenemme tämän kirjan mukaan noin luvun viikossa.

Lukuisista algoritmeja ja tietorakenteita käsittelevistä englanninkielisistä oppikirjoista lähimmin kurssin lähestymistapaa vastaa

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

Kirjan saa käyttöön sähköisesti yliopiston kirjaston kautta; linkki on kurssin verkkosivulla.

Cormen et al. sisältää huomattavasti enemmän materiaalia kuin tällä kurssilla käsitellään. Se sopiikin myös tämän aihepiirin jatkokursseille sekä hakuteokseksi myöhemmin.

Kurssilla käsiteltävistä asioista AVL-puut eivät löydy Cormenin kirjasta. Niihin hyvä lähde on esim.

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

Muita hyviä oppikirjoja ovat mm.

> A. Levitin: *Introduction to The Design and Analysis of Algorithms*, Addison-Wesley, 2003
>
> J. Kleinberg, E. Tardos: *Algorithm Design*, Pearson Addison-Wesley, 2006
>
> A. Aho, J. Hopcroft, J. Ullman: *Data Structures and Algorithms*, Addison-Wesley, 1983

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

Miksi pakollinen kurssi Tietorakenteet ja algoritmit?

- Monet tietorakenteet on helppo toteuttaa Javan valmiiden tietorakennetoteutusten avulla. Esim. seuraavassa luvussa esitettävän puhelinluettelon toteuttamiseen Javassa on valmiina luokka `TreeMap`. Miksi siis vaivautua opiskelemaan tietorakenteita 10 op:n verran?

- Vaikka tietorakennetoteutuksia on olemassa, ei niitä välttämättä osaa hyödyntää tehokkaasti ilman taustatietämystä niiden toimintaperiaatteista. Esim. kannattaako valita puhelinluettelon toteutukseen `Arraylist`, `HashMap`, `TreeMap` vai jokin muu Javan kokoelmakehyksestä löytyvä tietorakennetoteutus.

- Kaikiin ongelmiin ei löydy suoraan valmista tietorakennetta tai algoritmia ainakaan kielten standardikirjastojen joukosta. Esim. seuraavassa luvussa esitettävä kaupunkien välisten lyhimpien reittien tehokas etsiminen ei onnistu Javan valmiiden mekanismien avulla. Harjoittelemme tällä kurssilla algoritmien keksimistä myös hieman haastavammissa tilanteissa.

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

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

- Kuten lyhimpien polkujen esimerkistä sivulla 16–18 voidaan havaita, pienikin ongelma typerästi ratkaistuna on niin vaikea, ettei nopeinkaan kone siihen pysty.

- Vaikka koneet nopeutuvat, käsiteltävät datamäärät kasvavat suhteessa jopa enemmän.

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

- Paradoksaalisesti laitetehojen parantuminen saattaa jopa tehdä algoritmin tehokkuudesta tärkeämpää: algoritmien tehokkuuserot näkyvät yleensä selvästi vasta suurilla syötteillä, ja tehokkaat koneet mahdollistavat yhä suurempien syötteiden käsittelemisen.

- Muuttuva ympäristö tuo myös uusia tehokkuushaasteita:
  - moniytimisyyden yleistymisen myötä syntynyt kasvava tarve rinnakkaistaa laskentaa
  - hajautettu laskenta
  - reaaliaikainen laskenta
  - mobiililaitteiden rajoitettu kapasitetti
  - muistihierarkiat
  - jne.
- Tällä kurssilla keskitytään kuitenkin klassisiin perusasioihin: miten yhdellä perinteisellä prosessorilla käsitellään yhden koneen muistiin mahtuvaa dataa.

Tietorakenteiden opiskelu on monella tavalla "opettavaista"

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

Kurssin tavoitteet

Yleisemmin voidaan todeta, että kurssilla etsitään vastauksia seuraavanlaisiin kysymyksiin:

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

Osa kurssilla opittavista asioista (erityisesti algoritmien ja tietorakenteiden toteuttaminen Javalla) on sen luonteista, että niiden hallinta osoitetaan kokeiden sijasta vain harjoituksissa.

Johdantoesimerkki tietorakenteista

- Kaikki epätriviaalit ohjelmat joutuvat tallettamaan ja käsittelemään tietoa suoritusaikanaan

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

- Suoritusaikana tiedot tallennetaan tietorakenteeseen (engl. data structure)

Mikä on tietorakenne?

Tietorakenne tarkoittaa

- tapaa organisoida tieto koneen muistiin, ja
- operaatioita joiden avulla tietoa päästään käyttämään ja muokkaamaan.

Joskus lähes samasta asiasta käytetään nimitystä abstrakti tietotyyppi (engl. abstract data type, ADT)

- tietorakenteen sisäinen toteutus piilotetaan käyttäjältä
- abstrakti tietotyyppi näkyy käyttäjille ainoastaan operaatioina, joiden avulla tietoa käytetään
- abstrakti tietotyyppi ei siis ota kantaa siihen, miten tieto on koneen muistiin varastoitu.

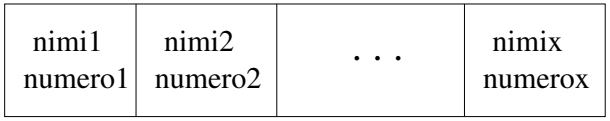
Mahdollinen puhelinluettelo-ohjelmiston tietorakenne

Tietorakenteen operaatioita ovat ainakin seuraavat:

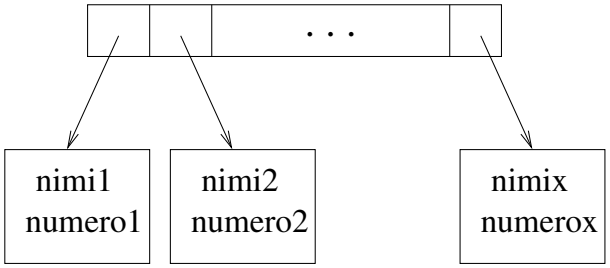
- **add**(n, p) lisää luetteloon nimen n ja sille numeron p
- **del**(n) poistaa nimen n luettelosta
- **findNumber**(n) palauttaa luettelosta henkilön n numeron
- **findName**(p) palauttaa luettelosta numeron p omistajan
- **update**(n, p) muuttaa n:lle numeroksi p:n
- **list**() listaa nimi-numeroparit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

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

- Tieto voisi olla talletettu suoraan taulukkoon



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



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

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

- Nämä suoraan taulukointiin perustuvat tietorakenteet eivät mahdollista esim. nimen perusteella tapahtuvan haun nopeaa toteutusta suurilla tietomäärillä. Opimme kurssin kuluessa useita parempia tietorakenteita (mm. hakupuut ja hajautusrakenteet) ongelman ratkaisemiseen.

Johdantoesimerkki algoritmeista

Tarkastellaan toisena esimerkkinä aivan toisentyyppistä ongelmaa

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

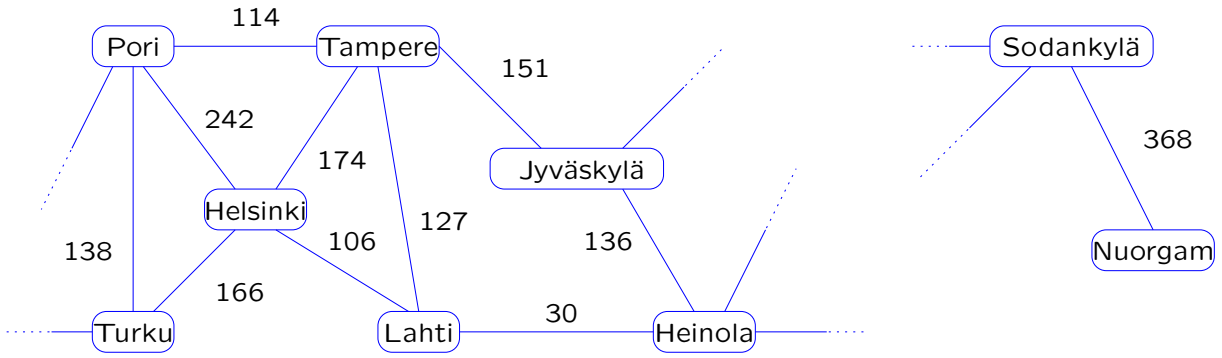
Helsinki–Lahti	106 km
Helsinki–Turku	166 km
Turku–Pori	138 km
Lahti–Tampere	127 km
...	...
Sodankylä–Nuorgam	368 km

$\Longrightarrow$

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

yht. 1328 km

- Tilanteeseen sopiva tietorakenne on painotettu suuntaamaton verkko



- Kysymyksessä on lyhimmän polun etsiminen verkosta.

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

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

Ongelma voitaisiin ratkaista raa'alla voimalla (brute force):

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

Raakaan voimaan perustuva ratkaisu on erittäin tehoton:

- oletetaan että on olemassa 32 paikkakuntaa
  $\Rightarrow$ karkeasti $30! \approx 2,7 \cdot 10^{32}$ mahdollista reittiä

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

- Raakaan voimaan perustuva ratkaisu on siis käytännössä käyttökelvoton.

- Kurssin loppupuolella esitetään useampikin tehokas algoritmi lyhimpien polkujen etsimiseen.
  Eräs näistä, Dijkstran algoritmi, käyttää itsessään suorituksen apuna keko-nimistä tietorakennetta.

Yhteenveto

- Tietorakenteita tarvitaan
  - suurten tietomäärien hallintaan (puhelinluettelo)
  - ongelmien mallintamiseen (kaupunkien väliset lyhimmät etäisyydet)
  - myös algoritmien laskennan välivaiheiden käsittelyyn (lyhimpien etäisyyksien etsimisessä Dijkstran algoritmin apuna keko).

- Tietorakenteet ja algoritmit liittyvät erottamattomasti toisiinsa:
  - tietorakenteiden toteuttamiseen tarvitaan algoritmeja
  - oikeat tietorakenteet tekevät algoritmista tehokkaan.

- Tietty ongelma voidaan usein ratkaista eri tavalla; on vaihtoehtoisia tietorakenteita tai tiettyyn operaatioon vaihtoehtoisia algoritmeja.

- Tilanteesta riippuu, mikä ratkaisuista on paras:
  - kuinka paljon dataa on
  - kuinka usein mitäkin operaatiota suoritetaan
  - mitkä ovat eri operaatioiden nopeusvaatimukset
  - onko muistitilasta pulaa jne.

Matematiikan kertausta: Logiikan merkinnät

Voimme muodostaa väittämistä uusia väittämiä yhdistämällä niitä loogisilla konnektiiveilla $\wedge$ (ja), $\vee$ (tai), $\Rightarrow$ (implikaatio) ja $\Leftrightarrow$ (ekvivalenssi). Loogisten lausekkeiden totuusarvot määräytyvät seuraavasti, kun T tarkoittaa tosi (true) ja F tarkoittaa epätosi (false):

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

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

Kvanttorien $\exists$ (on olemassa) ja $\forall$ (kaikilla) merkitys määritellään seuraavasti:

$\exists x \in A : \phi(x)$	jollain joukon A alkiolla pätee $\phi(x)$
$\forall x \in A : \phi(x)$	kaikilla joukon A alkioilla pätee $\phi(x)$.

Tässä A on jokin joukko ja ϕ jokin ominaisuus, joka joukon A alkioilla voi olla.

Jos esim. A on kaikkien paloautojen joukko ja $\phi(x)$ on väite "x on punainen", niin $\exists x \in A : \phi(x)$ tarkoittaa väitettä "on olemassa ainakin yksi punainen paloauto", ja $\forall x \in A : \phi(x)$ väitettä "kaikki paloautot ovat punaisia".

Jos perusjoukko A on asiayhteydestä selvä, se voidaan jättää merkitsemättä.

Loogisten symbolien käyttö on syytä rajata tilanteisiin, joissa ne oikeasti selventävät esitystä tai nimenomaan käsitellään formaalia logiikkaa. Luentokalvoilla jne. niitä toisinaan käytetään lyhennysmerkintöinä.

Matematiikan kertausta: Todistaminen

Tällä kurssilla pääpaino on algoritmien ja tietorakenteiden suunnittelemisessa ja soveltamisessa, ei matematiikassa ja todistamisessa. Nämä asiat kuitenkin kulkevat käsi kädessä:

- Vähän vaikeammalle ongelmalle ei välttämättä ole ilmeistä ratkaisualgoritmia, joka toimii oikein ja tehokkaasti.
- Algoritmia suunnitellessamme nimenomaan pyrimme pitämään huolen siitä, että syntyvällä algoritmilla on nämä halutut ominaisuudet.
- Kun meillä lopuksi on algoritmi ja selkeä käsitys siitä, miksi se toimii halutusti, niin voimme kirjata nämä perustelut todistukseksi asialle.

Tällä kurssilla oikeasti tarvitsemme hankalia todistuksia vain joidenkin monimutkaisimpien algoritmien yhteydessä.

Todistamista ja todistusten ymmärtämistä on kuitenkin hyvä harjoitella jo yksinkertaisemmissa tilanteissa.

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

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

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

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

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

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

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

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

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

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

- Miksi vastaväitetodistus toimii? Katsomme mitä tämä loogisesti tarkoittaa

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

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

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

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

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

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

Matematiikan kertausta: summakaavoja

- Aritmeettinen summa:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

  Summassa on n termiä, joiden keskiarvo on $(n+1)/2$ (sama kuin ensimmäisen ja viimeisen termin keskiarvo). Siis summa on $n \cdot (n+1)/2$.

- Geometrinen summa: kun $a \neq 1$, niin

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1}.$$

- Geometrinen sarja: kun $|a| < 1$, niin rajalla $n \to \infty$ saadaan

$$\sum_{i=0}^{\infty} a^i = \frac{1}{1 - a}.$$

- Geometrisen summan erikoistapaus $a = 2$:

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

- Yhtälölle

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

on hyödyllinen tulkinta lukujen binääriesitysten kautta.

- Muistetaan, että esim. binääriesityksen 101001 arvo saadaan laskemalla

$$101001_2 = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0.$$

- Erityisesti

$$\underbrace{111\ldots1_2}_{n+1 \text{ ykköstä}} = \sum_{i=0}^{n} 2^i.$$

- Toisaalta normaalilla binäärilukujen yhteenlaskulla saadaan

$$\underbrace{111\ldots1_2}_{n+1 \text{ ykköstä}} + 1 = \underbrace{10\ldots0_2}_{n+1 \text{ nollaa}} = 2^{n+1}.$$

- Siis

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

Matematiikan kertausta: Logaritmi

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

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

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

- Logaritmin laskusääntöjä:

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

- Nämä laskusäännöt voidaan johtaa suoraan logaritmin määritelmästä.

- Esim. olkoon $p = \log_a x$ ja $q = \log_a y$. Siis $a^p = x$ ja $a^q = y$. Täten

$$xy = a^p \cdot a^q = a^{p+q},$$

joten $\log_a(xy) = p + q = \log_a x + \log_a y$.

Kaksikantainen logaritmi $\log_2 n$ kertoo likimäärin luonnollisen luvun n binääriesityksen pituuden:

- Kun $b \geq 2$, niin pienin b-bittinen luku on

$$\underbrace{1000\ldots0}_{b-1 \text{ nollaa}} = 2^{b-1}.$$

- Suurin b-bittinen luku on

$$\underbrace{111\ldots1}_{b \text{ ykköstä}} = 2^b - 1$$

  (vertaa sivun 30 summalaskuun).

- Siis luonnollinen luku n on b-bittinen, jos ja vain jos

$$2^{b-1} \leq n < 2^b \quad \Leftrightarrow \quad b - 1 \leq \log_2 n < b \quad \Leftrightarrow \quad b = \lfloor \log_2 n \rfloor + 1,$$

  missä $\lfloor x \rfloor$ tarkoittaan luvun x pyöristämistä alaspäin lähimpään kokonaislukuun.

Siis tarkemmin luvun $n > 0$ pituus bitteinä on $\lfloor \log_2 n \rfloor + 1$.

Pseudokoodi algoritmien esityksessä

- Käytämme algoritmien kuvaukseen usein pseudokoodia emmekä mitään tiettyä ohjelmointikieltä.
- Tarkoituksena on esittää algoritmin perusperiaate takertumatta epäoleellisiin yksityiskohtiin.
  - Riippuu asiayhteydestä, mikä on ”oleellista”.
  - Algoritmien kuvauksessa käytetään varsinaisen pseudokoodin sijaan tai seassa myös luonnollista kieltä, matemaatisia merkintöjä jne.
- Pseudokoodin tarkoituksena on auttaa lukijaa, joka on ihminen, ymmärtämään algoritmin toiminta-ajatus.
- Ohjelmointikielillä taas pyritään kuvaamaan yksiselitteisesti, mitä tietokoneen halutaan tekevän.
- Tarkoitus on, että ainakin pienen harjoittelun jälkeen normaalilla ohjelmointitaidolla pystyy sujuvasti toteuttamaan pseudokoodina annetun algoritmin haluamallaan ohjelmointikielellä.

- Kirjallisuudessa on runsaasti erilaisia tapoja pseudokoodin kirjoittamiseen.
- Tällä kurssilla käytetty versio noudattaa jossain määrin Javan rakenteita:
  - kontrollirakenteina mm. tutut if, for ja while
  - taulukon alkioita indeksoidaan []-merkinnällä: A[i]
  - sijoituslausekkeessa käytetään = ja yhtäsuuruusvertailussa ==
  - kommentit alkavat //-merkinnällä
  - monimutkaiset asiat esitetään "oliona", jonka attribuutteihin viitataan pistenotaatiolla tyyliin A.length, henkilo.nimi, ...
  - metodien parametrien käyttäytyminen kuten Javassa, eli yksinkertaiset (int, double) arvona, monimutkaisista välitetään viite

- Käyttämämme pseudokoodin eroavuuksia Javaan:
  - loogiset konnektiivit kirjoitetaan and, or, not
  - pseudokoodin for muistuttaa enemmän Javan for eachia kuin normaalia for-lausetta
  - muuttujien, metodien parametrien tai paluuarvon tyyppejä ei määritellä eksplisiittisesti
  - lohkorakenne esitetään sisennyksen avulla aivan kuten Pythonissa, Javassahan lohkorakenne esitetään merkeillä { ja }
  - pseudokoodi on epäolio-orientoitunutta: olioihin liittyy vain attribuutteja, kaikki metodit ovat Javan mielessä staattisia eli saavat syötteet parametreina
  - return voi palauttaa monta arvoa
  - pseudokoodissa joitakin komentoja kirjoitetaan tarvittaessa englanniksi tai suomeksi, tyyliin: vaihda muuttujien x ja y arvot keskenään
- Pseudokoodi tulee nopeasti tutuksi kurssin edetessä.

Ohjelmointitaitoa: Taulukko

- Eräs kaikkein perustavanlaatuisimmista tietorakenteista useimmissa ohjelmointikielissä on taulukko (array).

- Taulukkoon voi tallentaa sen koon mukaisen määrän alkioita, ja yksittäisen alkion käsittely tapahtuu indeksoimalla.

- Koska tietokoneen muisti on oleellisesti yksi iso taulukko (tai useita sellaisia), taulukon toteutus konetasolla on suoraviivaista:
  - Jos taulukossa A yksittäinen alkio vie S muistipaikkaa ja alkio $A[0]$ on talletettu muistipaikasta B alkaen, niin alkio $A[i]$ löytyy muistipaikasta $B + i \cdot S$ alkaen.
  - Siis mikä tahansa yksittäinen alkio löytyy yhtä helposti.

- Java-ohjelmointikurssilla olet luultavasti oppinut myös kehittyneempiä Javan tukemia tietorakenteita vastaaviin tarkoituksiin (`ArrayList`, `TreeMap` jne.).

- Palaamme kurssilla näihin tietorakenteisiin sekä niiden toteuttamisen että soveltamisen näkökulmasta.

- Kurssin alkupään harjoitustehtävät on laadittu ratkaistavaksi käyttämättä mitään monimutkaisempia tietorakenteita kuin taulukko.
  - Kehittyneiden tietorakenteiden huolimaton käyttö voi aiheuttaa tehokkuus- tms. ongelmia.
  - Koska taulukko varmasti riittää, muiden tietorakenteiden miettiminen vain monimutkaistaa asiaa.

Ohjelmointitaitoa: Rekursio

- Metodi voi tunnetusti sisältää kutsuja muihin metodeihin.
- Jos metodi kutsuu itseään, se on rekursiivinen.
- Rekursion pitää kuitenkin päättyä joskus (eli ei saa esiintyä kehäpäättelyä): tämä perustapaus (eng. base case) ei kutsu metodia ja takaa sen, että rekursio päättyy.
- Rekursio on usein hyvä keino esittää monimutkaisen algoritmin toimintalogiikka, joskin sen sujuva ymmärtäminen ja käyttäminen vaatii jonkin verran harjoittelua.
- Esittelemme ensin rekursion perusidean yksinkertaisilla ”leikkikaluesimerkeillä” ja tarkastelemme sitten rekursiota hieman realistisemmissa sovelluksissa.

- Tarkastellaan seuraavaa yksinkertaista esimerkkiä: Olkoon x reaaliluku ja n luonnollinen luku. Tiedämme, että

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

- Tässä tapaus $n = 0$ on perustapaus, joka sisältää ratkaisun suoraan.

- Tämän voimme kirjoittaa ohjelmaksi:

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

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

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

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

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

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

- Koska n ei ole 0, suoritetaan rivi

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

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

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

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

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

- Pinon sisältö on siis seuraava:

2.0	0
2.0	1
2.0	2
2.0	3
2.0	4

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

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

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

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

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

- **Huomautus:** Potenssia ei oikeasti kannata toteuttaa rekursiolla. Esimerkki oli valittu, jotta se olisi helppo selittää. Parempi toteutus käyttää iteraatiota:

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

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

- Periaatteessa jokainen rekursio voidaan muuttaa iteraatioksi.
- Tulemme kurssin aikana näkemään algoritmeja, jotka on helppoa esittää rekursiivisesti mutta vaikeampaa koodata iteratiiviseksi ohjelmaksi.
- Rekursion edut: yleensä helpompi ymmärtää ja ohjelmoida, koodi on lyhyempää.
- Rekursion haitat: yleensä hitaampi ja vie enemmän muistia.

Muita matemaattisia esimerkkejä:

- Kertoman määritelmä on

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

- Tästä saamme heti rekursiivisen ohjelman:

```
public static long kertoma(int n) {
   if (n == 0)
      return 1;
   return n * kertoma(n-1);
}
```

- Tässäkin tapauksessa tehtävä on kuitenkin parempi ratkaista ilman rekursiota.

- Binomikertoimelle

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

tunnetaan palautuskaava

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

- Esitetään tästä saatava rekursiivinen algoritmi pseudokoodina:

```
binomikerroin(n, k)
    if k == 0 or k == n
        return 1
    else
        return binomikerroin(n − 1, k − 1) + binomikerroin(n − 1, k)
```

- Tämä on **erittäin tehoton** tapa binomikertoimen laskemiseksi eikä sitä pidä käyttää, elleivät luvut ole todella pieniä. Tehokkaampi tapa löytyy kurssikirjan luvusta 9.2.4.

Binäärihaku

- Hieman realistisempana rekursioesimerkkinä tarkastelemme binäärihakua, joka on myös muuten tärkeä algoritmi.

- Lähtökohtana oletetaan, että n-alkioisessa taulukossa $A[0 \ldots n-1]$ on kokonaislukuja suuruusjärjestyksessä; siis $A[i-1] \leq A[i]$ kaikilla i.

- Syötteenä annetaan kokonaisluku x, ja tehtävänä on löytää luvun x sijainti taulukossa A (ts. indeksi i jolla $A[i] = x$) tai todeta, että luku ei ole taulukossa.

- Binäärihaun perusidea on seuraava:
  - Verrataan haettavaa lukua x taulukon keskimmäiseen lukuun $A[n/2]$.
  - Jos x on pienempi, jatketaan rekursiivisesti taulukon alkupuoliskosta.
  - Jos x on suurempi, jatketaan rekursiivisesti taulukon loppupuoliskosta.
  - Kun taulukko on riittävän pieni, ratkaistaan ongelma suoraan ilman rekursiota.

- Yksityiskohdat vaativat vielä vähän työtä.

- Määritellään funktio binäärihaku(x, ala, ylä), joka etsii alkiota x osataulukosta A[ala. . . ylä]. Oletetaan, että kutsussa aina ala $\leq$ ylä.
- Alkuperäinen ongelma siis ratkeaa kutsulla binäärihaku(x, 0, A.size).
- Yksinkertaisuuden vuoksi taulukko A oletetaan määritellyksi globaalina muuttujana (ts. aliohjelman binäärihaku ulkopuolella niin, että se kuitenkin on käytettävissä k.o. aliohjelmassa).
- Sovitaan, että jos x ei ole taulukossa, niin palautusarvo on -1.
- Toteutuksessa vaatii hieman huolellisuutta pyöristysten ja rajatapausten hoitaminen niin, että hakualue aina lopulta pienenee yhden mittaiseksi.

- Saadaan rekursiivinen funktio

```
binäärihaku(x, ala, ylä)
    if ala == ylä
        if A[ala] == x
            return ala
        else
            return -1
    keski = ⌊(ala+ylä)/2⌋
    if x ≤ A[keski]
        return binäärihaku(x, ala, keski)
    else
        return binäärihaku(x, keski+1, ylä)
```

- Rekursiivinen esitystapa tuo selvästi esille, miten ongelman ratkaisu isolla taulukolla on palautettu saman ongelman ratkaisemiseen pienemmillä taulukoilla.

- Java-versio seuraavalla sivulla

```
public class Binaarihaku {

    static int[] A = new int[] {1, 1, 2, 3, 5, 8, 13, 21, 34};

    private static int binaarihaku(int x, int ala, int yla) {
        if (ala==yla) {
            if (A[ala]==x) return ala;
            else return -1;
        }
        int keski = (ala+yla)/2;
        if (x <= A[keski]) return binaarihaku(x, ala, keski);
        else return binaarihaku(x, keski+1, yla);
    }

    public static void main(String[] args) {
        System.out.println(binaarihaku(8, 0, A.length-1)); // 5
    }
}
```

- Tämäkin funktio on helppo muuttaa iteratiiviseksi:

```
binäärihaku(x)
    ala = 0
    ylä = A.length
    while ala < ylä
        keski = ⌊(ala+ylä)/2⌋
        if x ≤ A[keski]
            ylä = keski
        else
            ala = keski+1
    if A[ala] == x
        return ala
    else
        return -1
```

- Tämä on yksinkertainen ja tehokas toteutus, mutta ratkaisun taustalla oleva idea ei näy yhtä suoraan.

Rekursiivinen läpikäynti

- Tarkastellaan johdattelevana esimerkkinä luvuista 1, 2 ja 3 muodostettujen neljän pituisten jonojen tulostamista.
- Tulostettavia jonoja on siis $3^4 = 81$:

  (1, 1, 1, 1); (1, 1, 1, 2); (1, 1, 1, 3); (1, 1, 2, 1); . . . ;
  (3, 3, 2, 3); (3, 3, 3, 1); (3, 3, 3, 2); (3, 3, 3, 3).

- Tehtävä voidaan ratkaista helposti neljällä sisäkkäisellä silmukalla:

```
for a = 1 to 3
    for b = 1 to 3
        for c = 1 to 3
            for d = 1 to 3
                tulosta([a, b, c, d])
```

- Entä jos halutaan yleisemmin proseduuri tulostamaan kaikki n:n mittaiset jonot luvuista $1, \ldots, k$, missä n ja k annetaan parametreina?
- On helppo vaihtaa silmukoiden ylärajaksi k vakion 3 sijaan.
- Sen sijaan normaaleissa ohjelmointikielissä ei suoraan ole rakennetta, jolla saataisiin vaihteleva määrä sisäkkäisiä silmukoita.

- Oletetaan, että A[0. . . n-1] on globaali n-alkioinen taulukko. Rakennetaan siihen haluttuja lukujonoja yksi kerrallaan.

- Halutaan käydä läpi kaikki k^n mahdollista taulukon A[0. . . n-1] valintaa.

- Käytetään seuraavaa periaatetta:

```
// käy läpi kaikki tavat valita taulukko A[0...n-1]:
for i = 1 to k
    A[0] = i
    käy läpi kaikki tavat valita taulukko A[1...n-1]
```

- Tästä nähdään ongelmaan liittyvä rekursiivinen rakenne.

- Määritellään proseduuri listaa(n, k, m), joka käy läpi kaikki taulukon A[m. . . n-1] valinnat, kun A[0. . . m-1] on kiinnitetty.

- Tapauksessa m == n tulostetaan valmis taulukko.

- Saadaan seuraava rekursiivinen proseduuri:

```
listaa(n, k, m)
    if m == n
        print(A)
    else
        for i = 1 to k
            A[m] = i
            listaa(n, k, m+1)
```

- Alkuperäinen ongelma ratkaistaan kutsulla listaa(n, k, 0).

- Java-versio seuraavalla sivulla

```
import java.util.*;

public class Listaa {

    static int[] A;

    private static void listaa(int n, int k, int m) {
        if (n==m) {
            System.out.println(Arrays.toString(A));
        } else {
            for (int i=1; i<=k; i++) {
                A[m] = i;
                listaa(n, k, m+1);
            }
        }
    }

    public static void main(String[] args) {
        A = new int[4];
        listaa(4, 3, 0); // tulostaa 81 rivia [1, 1, 1, 1] ... [3, 3, 3, 3]
    }
}
```

- Samaa periaatetta voidaan käyttää muihinkin läpikäynteihin.
- Käydään esimerkiksi läpi kaikki joukot, joita voidaan muodostaa taulukon A[0. . . n-1] alkioista.
- (Tarkkaan sanoen käymme läpi ns. monijoukot (multiset), joissa sama alkio voi esiintyä useita kertoja.)
- Jokaisella alkiolla A[i] pitää päättää, otetaanko se mukaan. Käytetään boolean-taulukkoa mukana[0. . . n-1] pitämään kirjaa tehdyistä valinnoista.
- Saadaan rekursiivinen proseduuri

```
osajoukot(m)
    if m == n
        tulosta ne alkio A[i], joilla mukana[i] == true
    else
        mukana[m] = true
        osajoukot(m+1)
        mukana[m] = false
        osajoukot(m+1)
```

- Alkuperäinen ongelma ratkaistaan kutsulla osajoukot(0).
- Java-versio seuraavalla sivulla

```
public class Osajoukot {

    static int[] A;
    static boolean[] mukana;

    private static void osajoukot(int m) {
        if (m==A.length) {
            for (int i=0; i<A.length; i++)
                if (mukana[i]) System.out.print(A[i] + " ");
            System.out.println();
        } else {
            mukana[m] = true;
            osajoukot(m+1);
            mukana[m] = false;
            osajoukot(m+1);
        }
    }

    public static void main(String[] args) {
        A = new int[] {1, 2, 3, 4};
        mukana = new boolean[A.length];
        osajoukot(0); // tulostaa 16 rivia "1 2 3 4" ... <tyhja>
    }
}
```

- Osajoukoille voidaan tietysti tehdä muutakin kuin tulostaa.
- Tarkastellaan ongelmaa, voiko annetun taulukon A[0. . . n-1] alkiot jakaa kahteen osaan, joiden summa on sama.
- Seuraavan rekursiivisen funktion kutsu tasajako(0) palauttaa tosi, jos löytyy valinta, jolla mukana olevien summa on sama kuin pois jätettyjen:

```
tasajako(m)
    if m == n
        summa1 = 0
        summa2 = 0
        for i = 0 to n-1
            if mukana[i]
                summa1 = summa1 + A[i]
            else
                summa2 = summa2 + A[i]
        return (summa1 == summa2)
    else
        mukana[m] = true
        b1 = tasajako(m+1)
        mukana[m] = false
        b2 = tasajako(m+1)
        return (b1 or b2)
```

- Edellistä algoritmia voidaan yksinkertaistaa ja tehostaa huomaamalla, että tarvitsee pitää kirjaa ainoastaan mukaan ja pois valittujen summista.

- Otetaan mukaan ylimääräiset parametrit summa1 ja summa2, joihin summia kasvatetaan alkio kerrallaan. Taulukkoa mukana ei tarvita.

```
tasajako(summa1, summa2, m)
    if m == n
        return (summa1 == summa2)
    else
        b1 = tasajako(summa1+A[m], summa2, m+1)
        b2 = tasajako(summa1, summa2+A[m], m+1)
        return (b1 or b2)
```

- Ongelma ratkaistaan kutsulla tasajako(0, 0, 0).

- Java-versio seuraavalla sivulla

```
public class Tasajako {

    static int[] A;

    private static boolean tasajako(int summa1, int summa2, int m) {
        if (m==A.length)
            return (summa1==summa2);
        else {
            boolean b1 = tasajako(summa1+A[m], summa2, m+1);
            boolean b2 = tasajako(summa1, summa2+A[m], m+1);
            return (b1 || b2);
        }
    }

    public static void main(String[] args) {
        A = new int[] {3, 6, 3, 7, 5};
        System.out.println(tasajako(0, 0, 0)); // true
    }
}
```

- Entä jos tasajako ei ole mahdollinen?

- Edellinen ratkaisu on helppo yleistää löytämään pienin mahdollinen ero kahtiajaon osilla:

```
pieninEro(summa1, summa2, m)
    if m == n
        return |summa1 − summa2|
    else
        d1 = pieninEro(summa1+A[m], summa2, m+1)
        d2 = pieninEro(summa1, summa2+A[m], m+1)
        return min(d1, d2)
```

```
public class PieninEro {

    static int[] A;

    private static int pieninEro(int summa1, int summa2, int m) {
        if (m==A.length)
            return Math.abs(summa1-summa2);
        else {
            int d1 = pieninEro(summa1+A[m], summa2, m+1);
            int d2 = pieninEro(summa1, summa2+A[m], m+1);
            return Math.min(d1, d2);
        }
    }

    public static void main(String[] args) {
        A = new int[] {3, 6, 3, 7, 5};
        System.out.println(pieninEro(0, 0, 0)); // 0
        A = new int[] {3, 9, 4};
        System.out.println(pieninEro(0, 0, 0)); // 2
    }
}
```

- Entä jos lisäksi halutaan listata paras jako, ei pelkästään sen summien erotusta?
- Nyt meidän pitää taas pitää kirjaa esim. siitä, mitkä alkiot on valittu ensimmäiseen osajoukkoon.
- Teemme tämä mukana-taulukolla samaan tapaan kuin osajoukkojen läpikäynnissä.
- Lisäksi meidän täytyy pitää muistissa paras tähän mennessä löydetty ratkaisu ja päivittää sitä tarvittaessa.
- Java-esitys seuraavilla kahdella sivulla

```
import java.util.*;

public class ParasJako {

    static int[] A;

    static boolean[] mukana;
    static boolean[] parasMukana;
    static int parasArvo;

    private static void parasJako(int summa1, int summa2, int m) {
        if (m==A.length) {
            int arvo = Math.abs(summa1-summa2);
            if (arvo<parasArvo) {
                parasArvo = arvo;
                parasMukana = Arrays.copyOf(mukana, mukana.length);
            }
        } else {
            mukana[m] = true;
            parasJako(summa1+A[m], summa2, m+1);
            mukana[m] = false;
            parasJako(summa1, summa2+A[m], m+1);
        }
    }
```

```
    public static void main(String[] args) {
        A = new int[] {3, 7, 6, 3, 5};
        mukana = new boolean[A.length];
        parasArvo = 9999;
        parasJako(0, 0, 0);
        for (int i=0; i<mukana.length; i++)
            if (parasMukana[i]) System.out.print(A[i]+" ");
        System.out.println();
        // tulos oli "3 6 3"
        A = new int[] {3, 9, 4};
        mukana = new boolean[A.length];
        parasArvo = 9999;
        parasJako(0, 0, 0);
        for (int i=0; i<mukana.length; i++)
            if (parasMukana[i]) System.out.print(A[i]+" ");
        System.out.println();
        // tulos oli "3 4"
    }
}
```

2. Algoritmin tehokkuus

Tehokkuudella tarkoitamme tällä kurssilla (ja yleisemminkin) ennen kaikkea algoritmin ajankulutusta ja myös muistinkulutusta. Olemme erityisesti kiinnostuneet näiden skaalautumisesta suurilla syöteaineistoilla.

Tämän luvun jälkeen opiskelija

- tuntee algoritmin aika- ja tilavaativuuden peruskäsitteet
- tuntee tärkeimmät aikavaativuusluokat
- osaa arvioita iteratiivisten ja yksinkertaisten rekursiivisten algoritmien vaativuutta
- osaa käytää O-, Θ- ja Ω-merkintöjä.

Lue luku 2 Laaksosen kirjasta!

Johdantoesimerkki: liikaa informaatiota?

Tarkastellaan potenssin laskevaa iteratiivista algoritmia (Java-esitys sivulla 45):

		aika	suorituskertoja
potenssi(x, n)			
1	tulos = 1	c_1	1
2	for i = 1 to n	c_2	n
3	tulos = tulos · x	c_3	n
4	return tulos	c_4	1

- Symbolit c_1 jne. esittävät kunkin rivin yksittäisen suorituskerran viemää aikaa (esim. sekunteina), jonka arvo riippuu toteutuksesta ja laitteistosta.
- Kokonaisaika saadaan kertomalla rivien suoritusajat suorituskertojen lukumäärällä ja laskemalla kaikki yhteen.

Tarkastelemme algoritmin aikavaativuutta (time complexity) eli suoritusaikaa syötteen n funktiona. (Parametrin x arvolla ei ole tässä vaikutusta suoritusaikaan, jos oletamme käytettävän normaalia liukulukuaritmetiikkaa.)

Aikavaativuudeksi $T(n)$ saadaan

$$\begin{aligned} T(n) &= c_1 \cdot 1 + c_2 \cdot n + c_3 \cdot n + c_4 \cdot 1 \\ &= (c_2 + c_3)n + (c_1 + c_4). \end{aligned}$$

Yksinkertaistamme tämän sanomalla, että aikavaativuus on suuruusluokkaa (tai kertaluokkaa) $O(n)$, eli $T(n) = O(n)$.

Tarkkaan ottaen merkintä $T(n) = O(n)$ tarkoittaa, että sopivasti valituilla $c > 0$ ja $n_0 \in \mathbb{N}$ pätee $T(n) \leq cn$ kaikilla $n \geq n_0$. Palaamme tähän myöhemmin.

Käytännössä merkintä tarkoittaa, että

- otamme mukaan vain johtavan termin $(c_2 + c_3)n$, eli osan joka kasvaa nopeimmin parametrin n funktiona
- jätämme huomiotta vakiokertoimen $(c_2 + c_3)$.

Miksi emme tarkastele vakiokertoimia:

- Haluamme analysoida algoritmia toteutuksesta (laitteistosta, ohjelmointikielestä jne.) riippumattomalla tasolla.
- Joka tapauksessa niitä olisi käytännössä mahdoton saada selville.
- **Huom.** joissain tapauksissa näin abstrakti tarkastelu voi antaa harhaanjohtavan kuvan; esim. yksi levyhaku voi viedä paljon enemmän aikaa kuin algoritmin kaikki muu laskenta yhteensä.

Miksi emme tarkastele alemman kertaluvun termejä:

- niillä on vielä pienempi merkitys lopputulokseen kuin vakiokertoimilla.

Perussyy on kuitenkin tarve yksinkertaistaa asiat ymmärrettävään muotoon.

Pahimman tapauksen aikavaativuus

Aikavaativuus $T(n)$ ilmaisee algoritmin suoritusajan jonkin syötettä kuvaavan suureen n funktiona:

- Tällä kurssilla tyypillisesti n on jokin syötteen kokoa kuvaava luonnollinen suure: taulukon koko, merkkijonon pituus, puhelinluettelon numeroiden määrä tms.
- Toisinaan n voi olla jokin abstraktimpi syötteen vaikeutta kuvaava parametri.
- Hyvin teoreettisissa tarkasteluissa n on syötteen koko esim. bitteinä.

Usein kuitenkin suoritusaika voi vaihdella samankokoisilla syötteillä.

- Esim. jokin järjestämisalgoritmi voi toimia hyvin nopeasti, jos syötetaulukko sattuu olemaan jo valmiiksi järjestyksessä. (Ja ehkä yllättäen, järjestämisalgoritmi voi myös toimia *hitaasti* tässä tilanteessa.)

Yleensä tarkastelemme pahimman tapauksen (worst case) aikavaativuutta, jolloin $T(n)$ on pisin aika millään kokoa n olevalla syötteellä.

Esimerkki: Peräkkäishaku

Tarkastellaan alkion x etsimistä taulukosta $A[0 \ldots n-1]$ suoraviivaisesti peräkkäishaulla:

```
                                    aika    suorituskertoja
peräkkäishaku(x, A)
1   i = 0                           c1      1
2   while i < A.length              c2      m + 1
3       if A[i] = x                 c3      m + 1 tai m
4           return i                c4      1 tai 0
5       i = i + 1                   c5      m
6   return −1                       c6      0 tai 1
```

Tässä silmukan suorituskertojen lukumäärää on merkitty m:

- jos x on taulukossa, niin m on sen ensimmäinen esiintymisindeksi
- muuten $m = n$ (yksi yli taulukon viimeisen indeksin).

Suoritusaika on suurin, kun x ei ole taulukossa. Tällöin aikaa menee

$$\begin{aligned} & c_1 \cdot 1 + c_2 \cdot (n+1) + c_3 \cdot n + c_4 \cdot 0 + c_5 \cdot n + c_6 \cdot 1 \\ = \; & (c_2 + c_3 + c_5)n + (c_1 + c_2 + c_6). \end{aligned}$$

Siis pahimman tapauksen aikavaativuus on $T_{\text{pahin}}(n) = O(n)$.

Jos x on taulukossa indeksillä m, niin aikaa menee

$$\begin{aligned} & c_1 \cdot 1 + c_2 \cdot (m+1) + c_3 \cdot (m+1) + c_4 \cdot 1 + c_5 \cdot m + c_6 \cdot 0 \\ = \; & (c_2 + c_3 + c_5)m + (c_1 + c_2 + c_3 + c_4). \end{aligned}$$

Erityisesti paras tapaus (lyhin suoritusaika) on tilanteessa $m = 0$ eli kun haettu alkio on heti taulukon alussa. Algoritmin paras tapaus on vakioaikainen, eli parhaan tapauksen aikavaativuus pysyy samana vaikka n kasvaa.

Merkitsemme tätä $T_{\text{paras}}(n) = O(1)$.

Voimme myös tarkastella keskimääräistä aikavaativuutta.

Oletetaan, että

- haettu alkio löytyy aina taulukosta
- kaikki sijainnit $0, \ldots, n-1$ ovat yhtä todennäköisiä.

Keskimääräinen suoritusaika saadaan ottamalla odotusarvo edellisellä kalvolla lasketuista tapauksista $m = 0, \ldots, n-1$:

$$\begin{aligned}
T_{\text{keski}}(n) &= \frac{1}{n} \sum_{m=0}^{n-1} (c_2 + c_3 + c_5)m + (c_1 + c_2 + c_3 + c_4) \\
&= (c_2 + c_3 + c_5) \cdot \frac{1}{n} \sum_{m=0}^{n-1} m + (c_1 + c_2 + c_3 + c_4) \\
&= (c_2 + c_3 + c_5) \cdot \frac{1}{n} \cdot \frac{n(n-1)}{2} + (c_1 + c_2 + c_3 + c_4) \\
&= \tfrac{1}{2}(c_2 + c_3 + c_5)n + (c_1 + \tfrac{1}{2}c_2 + \tfrac{1}{2}c_3 + c_4 - \tfrac{1}{2}c_5).
\end{aligned}$$

Siis myös $T_{\text{keski}}(n) = O(n)$, mutta vakiokertoimet ovat pienemmät kuin pahimman tapauksen aikavaativuudella.

Tällä kurssilla, ja algoritmianalyysissa yleensäkin, keskitytään pahimman tapauksen aikavaativuuteen:

- Usein nimenomaan halutaan yläraja, jota enemmän aikaa ainakaan ei mene.
- Tyypillisesti pahin tapaus antaa oikean kuvan esim. algoritmien keskinäisistä suhteista myös ”tyypillisillä” syötteillä.
- **Kuitenkin** joissain tilanteissa pahin tapaus voi olla harhaanjohtavan pessimistinen.

Monesti keskimääräinen aikavaativuus olisi käytännössä kiinnostava, mutta

- analysoiminen on erittäin vaikeaa
- tarvitaan usein vaikeasti perusteltavissa olevia oletuksia syötteen jakaumasta.

Paras tapaus ei yleensä ole kiinnostava:

- algoritmeilla on usein epätyypillisiä helppoja tapauksia, jotka eivät kuvasta niiden normaalia toimintaa.

Aikavaativuuden arviointi käytännössä

Olemme pääsiassa kiinnostuneet arvioimaan algoritmin pahimman tapauksen aikavaativuutta suuruusluokan tarkkuudella (O-merkinnällä).

Edellisestä esimerkistä nähdään pari yleisperiaatetta.

Yksittäiset operaatiot: vakioaika $O(1)$

- tavalliset aritmeettiset operaatiot, vertailut, taulukon yhden alkion käsittely, . . .
- vakiomäärä tällaisia peräkkäin on edelleen $O(1)$
- mutta **ei** aliohjelmakutsu
- aritmeettiset operaatiot hyvin suurilla kokonaisluvuilla tai suurella tarkkuudella voivat vaatia tarkempaa analyysia.

Perussilmukka: lineaarinen aika $O(n)$

- Tämä on siis peräkkäishaun tyyppinen tapaus

```
for i = 1 to n
    // vakioaikaista laskentaa
```

- Yleisemmin jos silmukan yhden iteraation sisällä tehtävän laskennan aikavaativuus on $O(f(n))$ jollain f, niin silmukan

```
for i = 1 to n
    // laskentaa ajassa O(f(n))
```

  aikavaativuus on $O(nf(n))$.

- Erityisesti kahden sisäkkäisen silmukan

```
for i = 1 to n
    for j = 1 to n
        // laskentaa vakioajassa O(1)
```

  aikavaativuus on $O(n^2)$, kolmella sisäkkäisellä silmukalla $O(n^3)$ jne.

- Myös

```
for i = 1 to n/2
    // laskentaa vakioajassa O(1)
```

  jne. on $O(n)$; ylärajan muuttaminen vakiolla vaikuttaa vain vakiokertoimiin.

- Usein esiintyy tilanne, jossa sisemmän silmukan raja riippuu ulomman silmukan silmukkamuuttujasta:

```
for i = 1 to n
    for j = 1 to i
        // laskentaa vakioajassa O(1)
```

- Tämä on edelleen $O(n^2)$.
- Silmukan sisin osuus suoritetaan arvoilla $i = 1, \ldots, n$ ensin kerran, sitten 2 kertaa, 3 kertaa jne. aina i kertaan asti.
- Siis kaikkiaan sisimmän osuuden suorituskertoja tulee

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \tfrac{1}{2}n^2 + \tfrac{1}{2}n = O(n^2).$$

Peräkkäiset algoritminpätkät

- Kun vakiomäärä algoritminpätkiä suoritetaan peräkkäin, niin kokonaisaikavaativuudeksi tulee yksittäisten pätkien aikavaativuuksista suurin.
- Esim. silmukoiden yhdistelmällä

```
for i = 1 to n
    for j = 1 to n
        // laskentaa vakioajassa
for i = 1 to n
    // laskentaa vakioajassa
for i = 1 to n
    for j = 1 to n
        // laskentaa vakioajassa
```

  aikavaativuudeksi tulee $O(n^2)$.

- Tämä perustuu siihen, että jos esim. $f_1(n) = c_1 n^2$, $f_2(n) = c_2 n$ ja $f_3(n) = c_3 n^2$, niin

$$f_1(n) + f_2(n) + f_3(n) = (c_1 + c_3)n^2 + c_2 n = O(n^2).$$

Rekursion aikavaativuus

Tarkastellaan esimerkkinä potenssiinkorotuksen rekursiivista versiota:

```
                                        aika
potenssi(x, n)
1   if n == 0                           c1
2       return 0                        c2
3   return x·potenssi(x, n-1)           c3 + ???
```

- Ongelmana on rivin 3 käsittely.
- Kertolasku ja lopputuloksen palauttaminen ovat normaaleja vakioaikaisia operaatioita.
- Vakioon c_3 on laskettu nämä sekä aliohjelmakutsun valmistelutoimet (aktivaatiotietueen luominen).
- Mutta miten käsitellään rekursiivisen kutsun sisällä kuluva aika? Sitä on yllä alustavasti merkitty *???*.
- Algoritmin aikavaativuus $T(n)$ toteuttaa seuraavan "yhtälön":

$$T(n) = \begin{cases} c_1 + c_2 & \text{kun } n = 0 \\ c_1 + c_3 + ??? & \text{kun } n \geq 1. \end{cases}$$

- Koska "*???*" on kutsun potenssi(x, n-1) aikavaativuus, niin määritelmän mukaan itse asiassa se on $T(n-1)$.
- Vaikka emme tiedä, mitä $T(n-1)$ on, voimme sijoittaa sen edelliseen yhtälöön:

$$T(n) = \begin{cases} c_1 + c_2 & \text{kun } n = 0 \\ c_1 + c_3 + T(n-1) & \text{kun } n \geq 1. \end{cases}$$

- Yksinkertaisuuden vuoksi merkitsemme vielä $c_1 + c_2 = a$ ja $c_1 + c_3 = b$, ja saamme palautuskaavan eli rekursioyhtälön

$$T(n) = \begin{cases} a & \text{kun } n = 0 \\ T(n-1) + b & \text{kun } n \geq 1. \end{cases}$$

- Voimme ratkaista rekursioyhtälön soveltamalla rekursiivista osaa toistuvasti, kunnes päästään perustapaukseen:

$$\begin{aligned} T(n) &= T(n-1) + b \\ &= (T(n-1-1) + b) + b \\ &= T(n-2) + 2b \\ &= \ldots \\ &= T(n-i) + i \cdot b \\ &= \ldots \\ &= T(1) + (n-1)b \\ &= T(0) + nb \\ &= a + nb. \end{aligned}$$

- Siis $T(n) = O(n)$, eli tässä tapauksessa aikavaativuuden suuruusluokka on sama kuin iteratiivisella ratkaisulla.
- Yleisesti ottaen rekursiivisen algoritmin aikavaativuudesta voi aina kirjoittaa rekursioyhtälön, mutta sen ratkaiseminen voi olla vaikeaa tai mahdotonta.

Tarkastellaan toisena esimerkkinä seuraavaa rekursiorakennetta, jonka tyyppisiä esiintyi esim. osajoukkojen läpikäynnissä (s. 58):

```
rekursio(n)
1   if n == 0
2       return
3   rekursio(n-1)
4   rekursio(n-1)
```

- Samalla periaatteella kuin edellisessä esimerkissä saadaan aikavaativuudelle $T(n)$ rekursioyhtälö

$$T(n) = \begin{cases} a & \text{kun } n = 1 \\ 2 \cdot T(n-1) + b & \text{kun } n \geq 1. \end{cases}$$

- Oleellisena erona on siis kerroin 2 rekursiivissa osassa.
- **Huom.** tämä ei ole sellainen ”vakiokerroin”, jonka voi jättää huomiotta O-tarkastelussa, koska sen vaikutus kertautuu, kuten seuraavaksi nähdään.

Purkamalla taas rekursio saadaan

$$\begin{aligned}
T(n) &= 2T(n-1)+b \\
&= 2(2T(n-2)+b)+b \\
&= 2^2T(n-2)+2b+b \\
&= 2^2(2T(n-3)+b)+2b+b \\
&= 2^3T(n-3)+b\cdot(2^2+2+1) \\
&= \ldots = \quad 2^iT(n-i)+b\cdot\sum_{j=0}^{i-1}2^i \quad = \ldots \\
&= 2^nT(0)+b\cdot\sum_{j=0}^{n-1}2^i \\
&= a\cdot 2^n+b\cdot(2^n-1).
\end{aligned}$$

- Siis aikavaativuus on $O(2^n)$.
- Huomaa, että 2^n on n-alkioisen joukon osajoukkojen määrä. Tämä aikavaativuus vastaa tilannetta, jossa jokaiselle osajoukolle tehdään jotain vakioaikaista.
- Jos jokaiselle osajoukolle tehdään esim. ajan $O(n^2)$ vaativa operaatio, pitää ylläolevaan vaihtaa vakion a paikalle n^2.
- Tällöin aikavaativuudeksi tulee $O(n^2 2^n)$, jne.

Binäärihaku (s. 53) on esimerkki toisesta usein esiintyvästä rekursiotyypistä:

```
binäärihaku(x, ala, ylä)
    if ala == ylä
        if A[ala] == x
            return ala
        else
            return -1
    keski = ⌊(ala+ylä)/2⌋
    if x ≤ A[keski]
        return binäärihaku(x, ala, keski)
    else
        return binäärihaku(x, keski+1, ylä)
```

- Olkoon $T(n)$ binäärihaun aikavaativuus, kun n on läpikäytävän osataulukon koko ylä $-$ ala $+ 1$.
- Jos n on parillinen, niin rekursiivisiin kutsuihin menevissä osataulukoissa A[ala. . . keski] ja A[keski+1. . . ylä] on kummassakin tasan $n/2$ alkiota.

- Oletetaan yksinkertaisuuden vuoksi, että n on kakkosen potenssi eli $n = 2^k$ jollain $k \in \mathbb{N}$. Siis $k = \log_2 n$.
- Tällöin myös kaikissa rekursiivisissa kutsuissa osataulukon koko pysyy parillisena ja vältetään pyöristysongelmat.
- Nyt algoritmiin kirjoitetuista kahdesta rekursiivisesta kutsusta tasan yksi toteutuu.
- Saadaan rekursioyhtälö

$$T(n) = \begin{cases} a & \text{kun } n = 0 \\ T(n/2) + b & \text{muuten.} \end{cases}$$

Purkamalla tämä ja ottamalla huomioon oletus $n = 2^k$ saadaan

$$\begin{aligned} T(n) &= T(n/2) + b \\ &= (T((n/2)/2) + b) + b \\ &= T(n/2^2) + 2b \\ &= \ldots \\ &= T(n/2^i) + b \cdot i \\ &= \ldots \\ &= T(n/2^k) + b \cdot k \\ &= a + b \log_2 n. \end{aligned}$$

- Siis aikavaativuus on $O(\log n)$.
- Tämä voidaan melko suoraviivaisesti yleistää myös tapaukseen, jossa n ei ole kakkosen potenssi.
- Sivuutamme tämän matemaattisen harjoituksen, joka ei ole erityisen valaiseva algoritmiikan kannalta.

Verrataan tätä binäärihaun iteratiiviseen versioon (s. 53):

```
binäärihaku(x)
    ala = 0
    ylä = A.length
    while ala < ylä
        keski = ⌊(ala+ylä)/2⌋
        if x ≤ A[keski] then ylä = keski
        else ala = keski+1
    if A[ala] == x then return ala
    else return -1
```

- Koska while-silmukassa osataulukon koko ylä $-$ ala $+ 1$ aina puolittuu, silmukkaa suoritetaan $O(\log n)$ kertaa, missä n on taulukon koko.
- Koska muu on vakioaikaista, aikavaativuudeksi tulee taas $O(\log n)$.

Usein esiintyviä aikavaativuusluokkia

Vakioaikainen eli $O(1)$

- algoritmin suoritusaika pysyy samana mielivaltaisen suurilla syötteillä
- tyypillisesti koodinpätkä ilman toistoa tms.

Logaritminen eli $O(\log n)$

- syntyy esim. puolittamiseen perustuvista algoritmeista (binäärihaku)
- kasvaa erittäin hitaasti; esim.

$$\begin{aligned} \log_2 1000 &\approx 10 \\ \log_2 1000000 &\approx 20 \\ \log_2 1000000000 &\approx 30 \end{aligned}$$

- Tällainen tehokkuus edellytetään esim. suuria tietomääriä käsittelevien tietokantojen perusoperaatioilta.
- Ajassa $O(\log n)$ ei ehdi edes lukea n-alkioista taulukkoa, joten syötteen oletetaan yleensä olevan valmiiksi sopivassa tietorakenteessa (esim. binäärihaun järjestetty taulukko).

Lineaarinen eli $O(n)$

- tyypillinen perusalgoritmi, jossa esim. yksi for-silmukka
- paras mahdollinen aikavaativuus, jos koko syöte pitää lukea tai muuten käsitellä
- tyypillisesti sekunnissa ehtii käsitellä ainakin miljoona alkiota (vrt. TMC-tehtäviin)

Aikavaativuusluokka $O(n \log n)$

- Tähän asti nähdyillä arviointitekniikoilla ei ole ilmeistä, mistä tällainen aikavaativuus syntyy.
- Näemme viikolla 3, että tämä on esim. tehokkaiden järjestämisalgoritmien aikavaativuusluokka.
- Koska $\log n$ kasvaa hyvin hitaasti, tämä on melkein yhtä hyvä kuin lineaarinen.
- Tärkeä seuraus algoritmisuunnittelulle: jos olisi hyödyllistä olettaa syötteen olevan suuruusjärjestyksessä, niin sen voi aluksi järjestää suhteellisen pienellä lisätyöllä.

Neliöllinen eli $O(n^2)$

- esim. yksinkertaiset järjestämisalgoritmit (lisäysjärjestäminen jne.)
- toinen esimerkki: taulukon kaikkien alkioparien läpikäynti
- Voi olla hyvä pienillä syötteillä, mutta ei skaalaudu läheskään yhtä hyvin kuin $O(n \log n)$.

Kuutiollinen eli $O(n^3)$

- Tämä siis syntyy tyypillisesti kolmesta sisäkkäisestä silmukasta.
- esim. taulukon alkiokolmikoiden läpikäynti
- skaalautuvuus luonnollisesti vielä huonompi kuin neliöllisellä
- ja edelleen $O(n^4)$ jne. joita kuitenkaan ei kohdata tällä kurssilla (eikä käytännössä paljon muuallakaan)
- Yleisemmin aikavaativuuksia, jotka ovat $O(n^k)$ jollain vakiolla k, kutsutaan polynomisiksi.

Eksponentiaalinen eli $O(2^n)$ tai $O(3^n)$ jne.

- n-alkioisen joukon osajoukkoja on 2^n, mikä johtaa usein $O(2^n)$-tyyppisiin aikavaativuuksiin.
- Skaalautuvuus on todellinen ongelma, mutta sovellusten kannalta relevantin kokoisia syötteitä voi kenties käsitellä, etenkin jos vastausta ei tarvi silmänräpäyksessä.
- Eksponentiaaliset algoritmit ovat käytännössä tärkeitä mm. NP-täydellisten ongelmien ratkaisemisessa; tasajako-ongelma (s. 60) on esimerkki tällaisesta.
- Sanaa ”eksponentiaalinen” käytetään myös epämääräisemmin tarkoittamaan mitä tahansa aikavaativuutta, joka on suurempi kuin polynominen.

Permutaatioiden läpikäynti eli $O(n!)$

- vain hyvin pienille syötteille
- yritä parantaa algoritmi edes eksponentiaaliseksi (ei kuulu tämän kurssin alueeseen)

Kasvunopeuksien vertailua

- Jos syötteen koko n kasvaa yhdellä, niin esim. $O(n^3)$-algoritmin suoritusaika ei juuri kasva, mutta eksponentiaalisen $O(2^n)$-algoritmin ajankäyttö kaksinkertaistuu
- Jos käsiteltävien tietojen määrä kaksinkertaistuu:
  - vakioaikaisen algoritmin nopeus ei muutu
  - logaritminen algoritmi tarvitsee yhden lisäaskeleen
  - lineaarinen algoritmi tarvitsee kaksinkertaisen ajan
  - neliöllinen algoritmi tarvitsee nelinkertaisen ajan
  - kuutiollinen algoritmi tarvitsee kahdeksankertaisen ajan
  - $O(n^k)$-aikainen algoritmi tarvitsee 2^k-kertaisen ajan
  - eksponentiaalisen $O(2^n)$-algoritmin ajankäyttö korottuu toiseen potenssiin.
- Käytännössä eksponentiaalisten algoritmien suoritusaika alkaa jollain (kohtuullisen pienellä) n kasvaa räjähdysmäisesti.
- Eksponentiaalisilla algoritmeilla on silti oma tärkeä paikkansa myös käytännön algoritmiikassa.

Tila- eli muistivaativuus (space complexity)

Aikavaativuuden lisäksi arvioidaan usein algoritmin tilavaativuutta eli sen tarvitseman muistin määrää.

Tilavaativuudella tarkoitetaan algoritmin työtilaa eli paljonko muistia se tarvitsee syötteen tallentamisen lisäksi.

Tilavaativuus esitetään yleensä O-merkinnällä vastaavista syistä kuin aikavaativuuskin.

- esim. viikolla 3 esiteltävä lomitusjärjestäminen tarvitsee n-alkioisen taulukon järjestämisessä avuksi toisen n-alkioisen taulukon, johon alkuperäisen taulukon arvoja kopioidaan

  $\Rightarrow$ tilavaativuus on $O(n)$ eli lineaarinen

- myöhemmin esiteltävä kekojärjestäminen tarvitsee vain pari apumuuttujaa, joiden lukumäärä ei riipu järjestettävän taulukon koosta

  $\Rightarrow$ tilavaativuus on $O(1)$ eli vakio (englanniksi järjestäminen on "in place")

Rekursion tilavaativuus

- Rekursiivisessa aliohjelmassa rekursion jokainen kutsukerta varaa tilaa aliohjelman paikallisille muuttujille ja lisäksi vakiomäärän muistia paluuosoittimelle ja muuhun aliohjelmakutsun vaatimaan hallintoon.
- Tämä tila varataan aktivaatiotietueesta, joka viedään ajonaikaiseen pinoon.
- Jos paikallisia muuttujia on vakiomäärä, niin yhden aktivaatiotietueen koko on vakio.
- Tällöin koko pinon vaatima tila on verrannollinen pinon korkeuteen.
- Pinon korkeus kuitenkin vaihtelee suorituksen kestäessä.
- Koko algoritmin tilavaativuus on verrannollinen suurimpaan korkeuteen, jonka pino suorituksen aikana saavuttaa.

Esimerkki Tarkastellaan sivun 85 rekursiota

```
rekursio(n)
1   if n == 0
2       return
3   rekursio(n-1)
4   rekursio(n-1)
```

Aikavaativuudelle $T(n)$ saatiin palautuskaava

$$T(n) = \begin{cases} a & \text{kun } n = 0 \\ 2 \cdot T(n-1) + b & \text{kun } n \geq 1. \end{cases}$$

ja ratkaisuksi $T(n) = O(2^n)$.

Tilavaativuuden analysoinnissa on otettava huomioon, että rivin 4 rekursiivinen kutsu voi käyttää uudestaan saman muistitilan, jota rivin 3 kutsukin käytti.

Siis tilavaativuudelle $S(n)$ saadaan palautuskaava

$$S(n) = \begin{cases} c & \text{kun } n = 0 \\ S(n-1) + d & \text{kun } n \geq 1. \end{cases}$$

ja ratkaisuksi $S(n) = O(n)$.

Muut resurssit

Algoritmin tehokkuutta voidaan arvioida muillakin mittareilla

- jokin käytännössä tärkeä resurssi
  - tietokantaoperaation levyhakujen määrä
  - hajautetun toteutuksen tietoliikenteen määrä
  - jne.

- algoritmin toiminnan kannalta keskeinen perusoperaatio
  - järjestämisalgoritmin tekemien vertailujen määrä
  - jne.

Tilanteesta riippuu, mitkä mittarit ovat relevantteja.

Tällä kurssilla keskitytään tyypilliseen tilanteeseen, jossa suoritusaika on ensisijainen optimoitava resurssi.

Esimerkki: vaativuusluokat vs. vakiokertoimet

- Joltain aiemmalta kurssilta ehkä tuttu lisäysjärjestäminen järjestää taulukon ajassa $O(n^2)$.
- Seuraavalla viikolla tutustumme ajassa $O(n \log n)$ toimivaan lomitusjärjestämiseen.
- Vaativuusluokkien valossa lomitusjärjestäminen on selvästi parempi. Voiko vakiokerrointen tarkastelu muuttaa asetelmaa?
- Oletetaan, että lisäysjärjestäminen on toteutettu erittäin huolellisesti ja sen tarkkaa suoritusaikaa kuvaa funktio $T_{\text{lis}}(n) = 2n^2$, eli vakiokerroin olisi vain 2.
- Oletetaan toisaalta, että lomitusjärjestäminen on toteutettu huolimattomammin ja toteutuksen tarkkaa suoritusaikaa kuvaa funktio $T_{\text{lom}}(n) = 50n \log_2 n$.
- Jos syöte on pienehkö (suunnilleen $n < 200$), niin lisäysjärjestämisen aikavaativuus on itse asiassa pienempi.
- Koska nykyiset tietokoneet ovat erittäin nopeita, ei pienen syötteen ($n < 200$) suoritusajalla ole merkitystä edes huonosti toteutetulla algoritmilla.

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

Iso-O-notaatio tarkemmin

- Merkintätapaa $f(n) = O(g(n))$ kutsutaan yleisesti "iso-O-notaatioksi" tai -merkintätavaksi (big oh notation).
- Muodollisemmin käytetään termiä asymptoottinen aikavaativuus jne.
- Tarkastelemme tätä merkintätapaa täsmällisemmin. Olkoon f ja g funktioita $\mathbb{N} \to \mathbb{R}$. Merkintä $f = O(g)$ tarkoittaa

  on olemassa sellaiset vakiot $c > 0$ ja $n_0 \in \mathbb{N}$, että kaikilla $n \geq n_0$ pätee $0 \leq f(n) \leq cg(n)$.

- Siis riittävän suurilla n pitää päteä, että jollain vakiokertoimella skaalattu $g(n)$ on yläraja arvolle $f(n)$.
- Käytännössä merkitsemme tätä usein $f(n) = O(g(n))$; esim. $5n^2 + 23n + 12 = O(n^2)$.
- Etenkin aikavaativuuksien yhteydessä tarkasteltavat funktiot ovat yleensä positiivisia, joten emme jatkossa kiinnitä paljon huomiota ehtoon $0 \leq f(n)$.

- Päätellään täsmällistä määritelmää käyttäen, että tosiaan pätee $5n^2 + 23n + 12 = O(n^2)$. Siis määritellään $f(n) = 5n^2 + 23n + 12$ ja $g(n) = O(n^2)$ ja osoitetaan, että $f = O(g)$.
- Osoitamme määritelmän toteutumisen etsimällä sellaiset konkreettiset arvot c ja n_0, että $f(n) \leq cg(n)$ pätee kaikilla $n \geq n_0$. Tässä on yleensä paljon valinnanvaraa.
- Kokeillaan valita $n_0 = 10$.
- Funktion $f(n)$ johtava termi on valmiiksi haluttua muotoa *vakio* $\cdot\, g(n)$. Katsotaan muita termejä.
- Kun $n \geq 10$, pätee

$$23n = \frac{23}{n} \cdot n^2 \leq 2{,}3n^2 < 3n^2,$$

koska $23/n \leq 23/10$, ja selvästi

$$12 < 100 \leq n^2.$$

Siis arvoilla $n \geq n_0$ pätee

$$f(n) = 5n^2 + 23n + 12 < 5n^2 + 3n^2 + n^2 = 9n^2 = 9g(n),$$

joten esim. valinta $c = 9$ antaa halutun tuloksen.

Miten sitten osoitetaan, että joillain f ja g ei päde, että $f = O(g)$?

- Osoitetaan esimerkkinä, että $\frac{1}{100}n^3 \neq O(n^2)$, missä $f \neq O(g)$ tarkoittaa, että ei päde $f = O(g)$.
- Todistus on epäsuora. Tehdään vastaoletus, että $\frac{1}{100}n^3 = O(n^2)$.
- Siis joillain vakioilla c ja n_0 pätee, että $\frac{1}{100}n^3 \leq cn^2$ kun $n \geq n_0$.
- Jakamalla puolittain arvolla n^2 tästä seuraa $\frac{1}{100}n \leq c$ kun $n \geq n_0$.
- Tämä johtaa ristiriitaan millä tahansa $n \geq \max\{\, n_0, 100c + 1 \,\}$.

Esimerkkinä yleisistä periaatteista todistetaan seuraava:

Jos $f_1(n) = O(g_1(n))$ ja $f_2(n) = O(g_2(n))$, niin
$f_1(n) + f_2(n) = O(\max\{\, g_1(n), g_2(n) \,\})$.

- Käytimme tätä edellä siinä muodossa, että jos kahden peräkkäin suoritettavan algoritminpätkän aikavaativuudet ovat $O(g_1(n))$ ja $O(g_2(n))$, niin niiden yhteisaikavaativuus on $O(\max\{\, g_1(n), g_2(n) \,\})$.
- Oletuksen mukaan on olemassa sellaiset vakiot c_1, c_2, n_1 ja n_2, että

$$\begin{array}{rcll} f_1(n) & \leq & c_1 g_1(n) & \text{kun } n \geq n_1 \\ f_2(n) & \leq & c_2 g_2(n) & \text{kun } n \geq n_2. \end{array}$$

- Valitaan $n_0 = \max\{\, n_1, n_2 \,\}$ ja $c = c_1 + c_2$.
- Kun $n \geq n_0$, niin $n \geq n_1$ ja $n \geq n_2$. Tällöin

$$f_1(n) + f_2(n) \leq c_1 g_1(n) + c_2 g_2(n) \leq (c_1 + c_2) \max\{\, g_1(n), g_2(n) \,\},$$

kuten haluttiin.

- Merkinnässä "$f(n) = O(g(n))$" yhtäsuuruusmerkki ei tarkoita oikeaa matemaattista yhtäsuuruutta.
- Olisi täsmällisempää määritellä, että $O(g)$ on kaikkien niiden funktioiden f joukko, joilla on olemassa sellaiset vakiot c ja n_0, että pätee $f(n) \leq cg(n)$ kun $n \geq n_0$.
- Tämän jälkeen voisimme merkitä matemaattisesti korrektisti $f \in O(g)$.
- Tällä kurssilla kuitenkin väärinkäytämme yhtäsuuruusmerkkiä vakiintuneen käytännön mukaisesti.

Muut asymptoottiset merkinnät: Ω ja Θ

- Merkintä $f = O(g)$ tarkoittaa, että asymptoottisesti g on yläraja funktiolle f. Siis f kasvaa korkeintaan yhtä nopeasti kuin g
- Määritellään vastaavasti asymptoottinen alarajamerkintä $f = \Omega(g)$ tarkoittamaan, että f kasvaa ainakin yhtä nopeasti kuin g.
- Määrittelemme siis, että $f = \Omega(g)$, jos

  on olemassa sellaiset vakiot $c > 0$ ja $n_0 \in \mathbb{N}$, että kaikilla $n \geq n_0$ pätee $0 \leq cg(n) \leq f(n)$.

- Lisäksi määritellään, että $f = \Theta(g)$, jos pätee sekä $f = O(g)$ että $f = \Omega(g)$. Tällöin siis funktioilla f ja g on sama kasvunopeus.
- Kirjallisuudesta löytyy myös muita merkintätapoja, esim. $f = o(g)$ (siis pikku-o) ja $f = \omega(g)$. Määritelmien yksityiskohdissa voi olla pieniä eroja.

- Algoritmikirjallisuudessa usein merkitään $f = O(g)$, vaikka oikeastaan tarkoitetaan $f = \Theta(g)$.
- Esim. edellä totesimme peräkkäishaun (s. 74) pahimman tapauksen aikavaativuudelle suuruusluokan $T(n) = O(n)$.
- Koska myös $T(n) \geq cn$ sopivalla $c > 0$, pätee itse asiassa vahvemmin $T(n) = \Theta(n)$.
- Koska O antaa vain ylärajan, olisi periaatteessa matemaattisesti korrektia kirjoittaa myös esim. $T(n) = O(n^2)$; tämä on kuitenkin harhaanjohtavaa.
- Yleensä siis ilmoitamme algoritmin aikavaativuuden O-merkintää käyttäen sillä ymmärryksellä, että siihen ei ole jätetty turhaa väljyyttä.

Eksponentti- ja logaritmifunktioiden suuruusluokat

Algoritmien aikavaativuuksia verratessa on keskeistä, että kaikilla $a > 1$ ja $k \in \mathbb{N}$

- a^n kasvaa nopeammin kuin n^k
- $\log_a n$ kasvaa hitaammin kuin $n^{1/k}$.

Siis O-merkintää käyttäen

- $n^k = O(a^n)$, mutta $a^n \neq O(n^k)$
- $\log_a n = O(n^{1/k})$, mutta $n^{1/k} \neq O(\log n)$.

Logaritmin kantaluku jätetään yleensä pois O-merkinnän sisältä, koska $\log_a n = (1/\log_b a) \cdot \log_b n$ ja siis $\log_a n = \Theta(\log_b n)$ kaikilla $a, b > 1$.

Kokeilemalla tai laskemalla numeerisia esimerkkejä on helppo vakuuttua, että käytännössä erot esim. eksponentiaalisten ja polynomisten aikavaativuuksien välillä ovat hyvinkin suuria.

Tarkastelemme lyhyesti, miten O-merkinnän täsmällinen määritelmä toimii tässä yhteydessä.

Lähtökohtana on analyysista tutut raja-arvot:

$$\lim_{n\to\infty} \frac{n^k}{a^n} = 0$$

$$\lim_{n\to\infty} \frac{\log n}{n^{1/k}} = 0$$

jotka pätevät kaikilla $a > 1$ ja $k \in \mathbb{N}$.

Käymme läpi yhden esimerkkitapauksen, joka havainnollistaa eksponenttifunktion käyttäytymistä. Paremmin tällaisten asioiden todistamiseen pääsee tutustumaan analyysin peruskursseilla.

- Osoitetaan esimerkkinä, että $2^n \neq O(n^k)$ millä tahansa $k \in \mathbb{N}$.
- Siis pitää osoittaa, että valittiin mitkä tahansa $c > 0$ ja $n_0 \in \mathbb{N}$, niin on olemassa $n \geq n_0$, jolla $2^n > cn^k$.
- Osoitamme itse asiassa hieman vahvemmin, että millä tahansa $c > 0$ on olemassa sellainen $n_0 \in \mathbb{N}$, että $2^n > cn^k$ pätee kaikilla $n \geq n_0$.
- Tapauksessa $k = 0$ oikea puoli on vakio, joten voimme olettaa $k \geq 1$.
- Otamme lähtökohdaksi epäyhtälön

  $$2^x > x,$$

  joka pätee kaikilla $x \in \mathbb{R}$. Tapaus $x \in \mathbb{N}$ on helppo todistaa induktiolla. Yleisten reaalilukujen tapaus hoituu differentiaalilaskennan perustyökaluilla.
- Keskeinen idea näissä tarkasteluissa on, että eksponenttifunktio muuttaa yhteenlaskun kertolaskuksi: esim.

  $$2^n = 2^{n/3+n/3+n/3} = 2^{n/3} \cdot 2^{n/3} \cdot 2^{n/3} > \frac{n}{3} \cdot \frac{n}{3} \cdot \frac{n}{3} = \frac{n^3}{27} = \frac{n}{27} \cdot n^2,$$

  joten $2^n > 50n^2$ kun $n \geq 27 \cdot 50$.

Olkoon siis $c > 0$ ja $k \geq 1$. Osoitetaan, että riittävän suurilla n pätee

$$2^n > cn^k.$$

Korottamalla kumpikin puoli potenssiin $1/k$ tämä saadaan ekvivalenttiin muotoon

$$2^{n/k} > c^{1/k} n.$$

Kun kirjoitetaan tämä muodossa

$$2^{n/(2k)} \cdot 2^{n/(2k)} > 2kc^{1/k} \cdot \frac{n}{2k},$$

nähdään, että tämä pätee ainakin jos

$$2^{n/(2k)} > 2kc^{1/k} \qquad \text{ja} \qquad 2^{n/(2k)} > \frac{n}{2k}.$$

Ensimmäinen väite pätee, kun $n > 2k \log_2(2kc^{1/k})$.

Jälkimmäinen seuraa suoraan epäyhtälöstä $2^x > x$.

Voidaan siis valita $n_0 = \lceil 2k \log_2(2kc^{1/k}) \rceil$.

3. Järjestäminen

Järjestäminen eli taulukon alkioiden saattaminen suuruusjärjestykseen on usein tarvittava ja paljon tutkittu tehtävä, jossa syötteet voivat olla hyvin suuria.

Tämän luvun jälkeen opiskelija

- tuntee perusteellisesti lisäys-, lomitus- ja pikajärjestämisen
  - totettaminen, aikavaativuus, erityispiirteet sovellusten kannalta, . . .
- osaa soveltaa järjestämistä osana algoritmista ongelmanratkaisua (harjoitellaan TMC-tehtävissä)

Järjestämisalgoritmit ovat myös hyviä esimerkkejä hajota ja hallitse -menetelmästä algoritmisuunnittelussa. Tämä johtaa hieman aiempaa rikkaampiin esimerkkeihin rekursiosta sekä rekursioyhtälöistä aikavaativuusanalyysissa.

Lisäysjärjestäminen (insertion sort)

Ohjelmointikursseilla on (kenties) tutustuttu seuraavaan tapaan järjestää taulukko taulu[0 . . . $n-1$] kasvavaan järjestykseen:

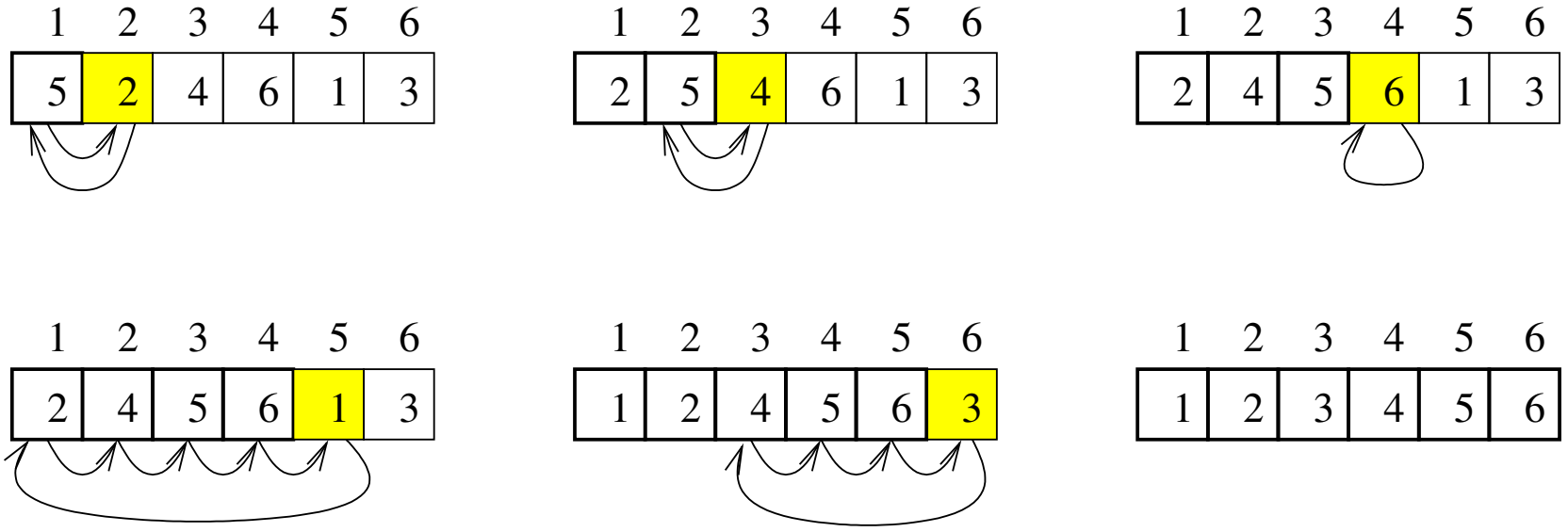
```
lisäysjärjestä(taulu[0...n-1])
    for i = 1 to n-1
        j = i-1
        while (j ≥ 0) and (taulu[j] > taulu[j+1])
            swap(taulu[j], taulu[j+1])
            j = j-1
```

Tiiviyden ja selkeyden vuoksi käytämme merkintää swap(x, y) järjestämisessä usein toistuvalle operaatiolle, jossa kahden alkion arvot vaihdetaan keskenään.

Lisäysjärjestämisen perusidea on pitää aina osataulukko taulu[0 . . . $i-1$] järjestyksessä.

- Aluksi $i = 1$, ja yksialkioinen taulukko on ilman muuta järjestyksessä.
- Jokainen silmukan iteraatio kasvattaa järjestettyä aluetta yhdellä sijoittamalla alkion taulu[i] oikealle paikalleen.
- Silmukasta poistuttaessa $i = n$, joten järjestetty alue kattaa koko taulukon.

Esimerkkikuva algoritmin toiminnasta:



Edellinen selitys lisäysjärjestämisen toiminnalle on esimerkki invariantin käytöstä.

- Lisäysjärjestämisessä for-silmukalla on silmukkainvarianttina väittämä

  Osataulukon $A[0 \ldots i-1]$ sisältämät taulukot ovat samat kuin alun perin, mutta suuruusjärjestyksessä.

- Huomaa, että tämä invariantti riippuu silmukkamuuttujasta i:
  - mitä pidemmälle silmukka etenee, sitä enemmän invariantti vaatii.

- Invariantti on algoritmin muuttujien ja tietorakenteiden tilaa koskeva väittämä, joka yksittäisellä suorituksen ajanhetkellä on joko tosi tai epätosi.

- Väittämä on silmukan invariantti, jos se on tosi silmukan jokaisen suorituskerran alkaessa.

- Tarkemmin jos ajatellaan silmukka kirjoitetuksi Java-tyyliin
  `for (i=1; i<=n-1; i++)`
  niin invariantin pitäisi olla tosi aina kun suoritetaan testi `i<=n-1`.

- Tämä sisältää myös viimeisen testikerran, jolloin testi on epätosi ja silmukka päättyy.

Sana ”invariantti” viittaa siihen, että kun ominaisuus on kerran voimassa, niin tiettyjen operaatioiden jälkeen se on taas voimassa.

Esim. lisäysjärjestämisessä silmukkamuuttujan i kasvattaminen laajentaa aluetta, jonka pitäisi olla järjestyksessä.

- Tämä voi tehdä invariantin epätodeksi.
- Silmukan rungossa tehtävät toimet kuitenkin palauttavat sen todeksi siihen mennessä, kun sitä seuraavan kerran tarkastellaan.
- Tässä käytetään hyväksi tietoa, että ennen silmukkamuuttujan kasvattamista invariantti oli voimassa, eli taulukko $A[0 \ldots i-2]$ on valmiiksi järjestyksessä.

Erilaiset invariantit ovat keskeisiä monimutkaisempien algoritmien ja tietorakenteiden suunnittelussa ja ymmärtämisessä.

Niitä tulee esiintymään silloin tällöin tällä kurssilla, mutta emme rupea systemaattisesti perehtymään niiden teoriaan.

Lisäysjärjestämisen aikavaativuus

Lisäysjärjestämisen pahin tapaus on taulukko, joka on vähenevässä järjestyksessä (kun siis halutaan kasvava):

- Tällöin while-silmukan pitää aina siirtää alkio taulu[i] paikkaan taulu[0] yksi askel kerrallaan.
- while-silmukan runkoa suoritetaan siis i kertaa.
- Yhteensä kaikilla i tästä tulee

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2},$$

  joten kokonaisaikavaativuus on $O(n^2)$.

Myös keskimääräinen aikavaativuus satunnaisessa järjestyksessä olevalla taulukolla on $O(n^2)$.

Parhaassa tapauksessa taulukko on valmiiksi kasvavassa järjestyksessä, jolloin while-silmukassa ei koskaan tehdä yhtään iteraatiota. Siis parhaan tapauksen aikavaativuus on $O(n)$.

Tutustumme seuraavaksi lomitus- ja pikajärjestämiseen, joilla aikavaativuudet ovat $O(n \log n)$. Myöhemmin tutustumme myös kekojärjestämiseen, jolla on tämä aikavaativuus.

Lisäysjärjestäminen on kuitenkin yksinkertainen, joten huolellisesti toteutettuna sen vakiokertoimet ovat pieniä.

Lisäysjärjestäminen onkin kelvollinen tai jopa suositeltava algoritmi, jos

- taulukko on pieni (esim. $n \leq 20$) tai
- tyypillisillä syötteillä taulukko on valmiiksi melkein oikeassa järjestyksessä.

Suurilla taulukoilla aikavaativuus $O(n^2)$ on täysin mahdoton ja pitää käyttää $O(n \log n)$-algoritmeja.

Lomitusjärjestäminen (merge sort)

Perusajatuksena on seuraava menetelmä järjestää (mahdollisesti vajaa) korttipakka:

1. Jos pakassa on vain yksi kortti, älä tee mitään.

2. Muuten
   - Jaa pakka kahteen suunnilleen yhtä suureen osaan A ja B.
   - Järjestä osa A soveltamalla tätä menetelmää rekursiivisesti.
   - Järjestä osa B soveltamalla tätä menetelmää rekursiivisesti.
   - Lomita nyt järjestyksessä olevat osapakat A ja B siten, että koko pakka tulee järjestykseen.

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

Lomitusjärjestäminen perustuu hajota ja hallitse (engl. divide-and-conquer) -tekniikkaan:

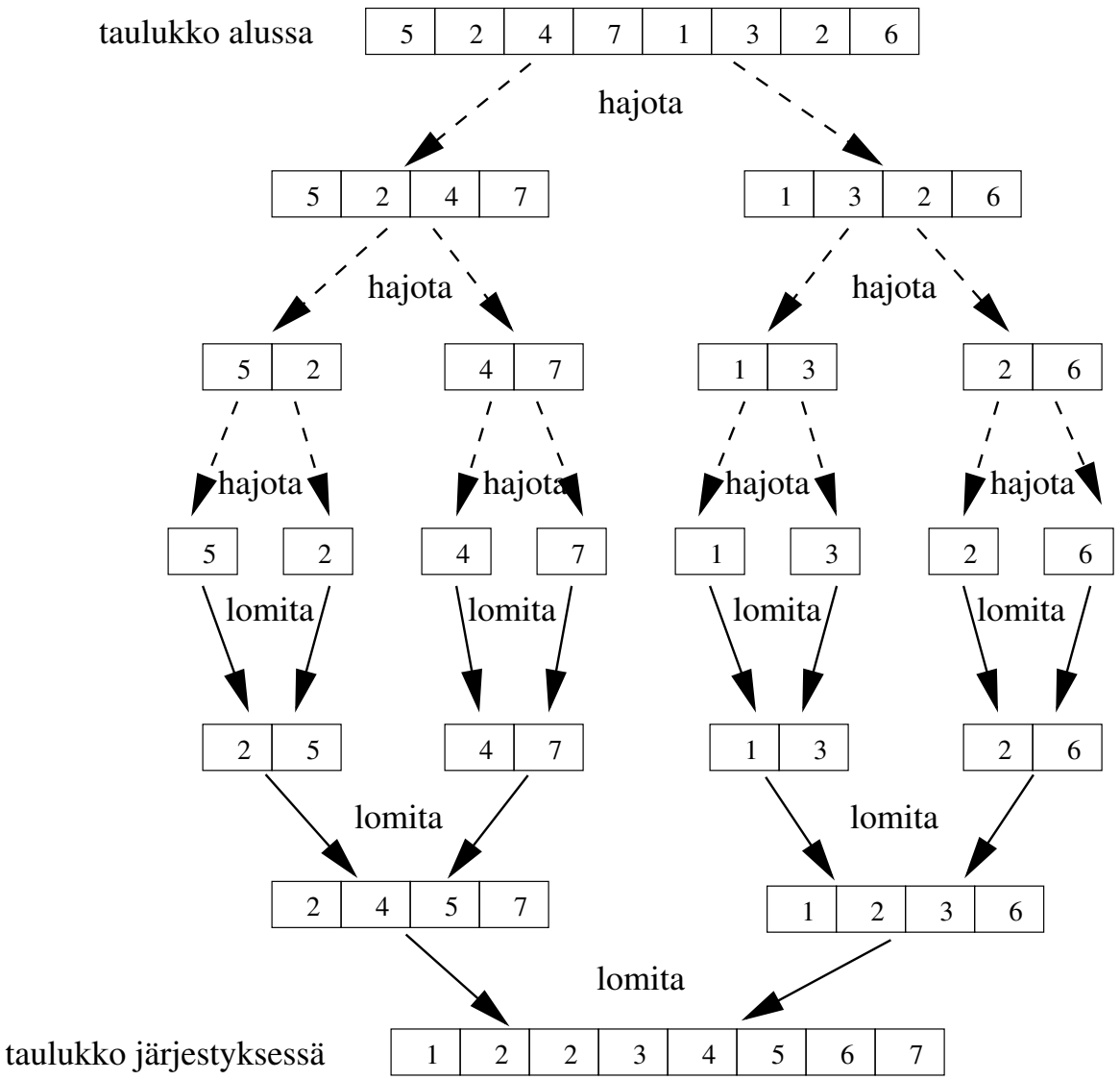
- hajotetaan ongelma pienempiin osaongelmiin
- ratkaistaan pienemmät osaongelmat rekursiivisesti
- hallitaan eli kootaan osaratkaisuista kokonaisratkaisu.

Allaoleva algoritmi järjestää osataulukon taulu[vasen. . . oikea]. Koko taulukko järjestetään kutsulla lomitusjärjestä(taulu, 0, n-1).

```
lomitusjärjestä(taulu, vasen, oikea)
    if vasen == oikea return
    keski = ⌊(vasen + oikea)/2⌋
    lomitusjärjestä(taulu, vasen, keski)
    lomitusjärjestä(taulu, keski+1, oikea)
    lomita(taulu, vasen, keski, keski+1, oikea)
```

Palaamme kohta proseduurin lomita toteuttamiseen.

Seuraavan sivun kuva antaa esimerkin rekursion toiminnasta.


taulukko alussa
5 2 4 7 1 3 2 6
hajota
5 2 4 7
1 3 2 6
hajota
hajota
5 2
4 7
1 3
2 6
hajota
hajota
hajota
hajota
5 2
4 7
1 3
2 6
lomita
lomita
lomita
lomita
2 5
4 7
1 3
2 6
lomita
lomita
2 4 5 7
1 2 3 6
lomita
taulukko järjestyksessä
1 2 2 3 4 5 6 7

Lomittamisessa on tarpeen käyttää aputaulukkoa.

Seuraavalla sivulla esitetty algoritmi lomittaa ensin osataulukot taulu[a1...b1] ja taulu[a2...b2] aputaulukkoon apu[a1...b2].

- Lomitusjärjestä-proseduurin merkintöjä käyttäen tässä a1=vasen, b1=keski, a2=keski+1 ja b2=oikea.
- Osataulukoiden taulu[a1...b1] ja taulu[a2...b2] oletetaan olevan järjestyksessä, ja tuloksena apu[a1...b2] sisältää samat alkiot järjestyksessä.
- Lopuksi aputaulukko kopioidaan vastaavalle paikalle alkuperäiseen taulukkoon.
- Lomitusalgoritmissa
  - a1 osoittaa vuorossa olevaa alkiota ensimmäisestä osataulukosta
  - a2 osoittaa vuorossa olevaa alkiota toisesta osataulukosta
  - pienempi alkio kopioidaan seuraavaan aputaulukon kohtaan ja kyseistä osoitinta kasvatetaan.

```
Iomita(taulu, a1, b1, a2, b2)
    a = a1
    b = b2
    for i = a to b
        if (a2 > b2) or ((a1 ≤ b1) and (taulu[a1] < taulu[a2]))
            // kopioidaan ensimmäisestä osataulukosta
            apu[i] = taulu[a1]
            a1 = a1+1
        else
            // kopioidaan toisesta osataulukosta
            apu[i] = taulu[a2]
            a2 = a2+1
    for i = a to b
        taulu[i] = apu[i]
```

- Ehto a2 > b2 testaa, onko toinen osataulukko kokonaan kopioitu. Jos on, niin loput alkiot tulevat ensimmäisestä osataulukosta.
- Ehto a1 ≤ b1 vastaavasti testaa, onko ensimmäisessä osataulukossa vielä kopioitavaa jäljellä.

Lomitusjärjestämisen aikavaativuus

Perustelemme, että lomitusjärjestämisen aikavaativuus on $O(n \log n)$.

- Yhden lomita-kutsun aikavaativuus on luokkaa $O(m)$, missä $m = b - a + 1$ on lomitettavien alkioiden lukumäärä. Proseduurin oleellinen osa nimittäin on yksi for-silmukka, jota toistetaan m kertaa.
- Koko prosessin aikavaativuuden havainnollistamiseksi yhdistetään eri vaiheiden ajankulutus sivun 122 kuvassa esiintyviin toimenpiteisiin.
- Yhden lomitusjärjestä-kutsun osalta
  - vakioajassa tapahtuvat ehtotesti jne. yhdistetään vastaavaan ”hajota”-kohtaan
  - lomita-kutsu yhdistetään vastaavaan ”lomita”-kohtaan
  - rekursion aikana tapahtuvat asiat yhdistetään (rekursiivisesti . . .) näiden välisiin kohtiin.

- Nyt kuvan jokaiseen ”lomita”-kohtaan on yhdistetty lomita-proseduurin kutsu ja sen aikavaativuus $O(m)$, missä m on lomitettavien alkioiden lukumäärä.
- Kun tarkastellaan kuvan yhtä kokonaista ”lomita”-rivillistä, havaitaan, että
  - jokainen taulukon alkio esiintyy tasan yhdessä lomituksessa
  - siis lomitettavien alkioiden kokonaislukumäärä on n
  - siis koko rivin aikavaativuus on $O(n)$.
- Lomitettavien osataulukoiden koko puolittuu aina siirryttäessä kuvassa yksi rivi ylöspäin. $\Rightarrow$ Rivejä on $O(\log n)$.
- Kun rivejä on $O(\log n)$ ja yksi rivi vie ajan $O(n)$, niin kokonaisaika on $O(n \log n)$.
- Kuvan ”hajota”-kohtia on yhtä monta kuin ”lomita”-kohtia ja kukin vie vain vakioajan, joten ne eivät kasvata aikavaativuuden suuruusluokkaa.

Siis aikavaativuus on $O(n \log n)$ riippumatta taulukon sisällöstä (pahin, paras ja keskimääräinen tapaus).

Aikavaativuutta voidaan tarkastella myös rekursioyhtälön kautta.

- Kun järjestettävän osataulukon koko on n, proseduurin lomitusjärjestä aikavaativuudelle $T(n)$ saadaan (hieman yksinkertaistaen) rekursio

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

- Tässä cn kuvaa lomita-kutsun aikavaativuutta.
- Oletetaan taas yksinkertaisuuden vuoksi $n = 2^k$ jollain $k \in \mathbb{N}$, jotta jaot menevät tasan.

Rekursiota voidaan nyt purkaa:

$$\begin{aligned}
T(n) &= 2T(n/2) + cn \\
&= 2(2T(n/4) + c(n/2)) + cn \\
&= 2^2 T(n/2^2) + cn + cn \\
&= 2^2(2T(n/2^3) + cn/2^2) + cn + cn \\
&= 2^3 T(n/2^3) + cn + cn + cn \\
&= \ldots \\
&= 2^k T(n/2^k) + \underbrace{cn + \ldots + cn}_{k \text{ kappaletta}} \\
&= n \cdot c + (\log_2 n) \cdot cn
\end{aligned}$$

kun $k = \log_2 n$. Siis ratkaisulle pätee $T(n) = O(n \log n)$.

(Tällaisten rekursioyhtälöiden ratkaisemista ei edellytetä opiskelijoilta tällä kurssilla.)

Lomitusjärjestämisen tilavaativuus

- Pahin yksittäinen muistitilaa kuluttava seikka on lomita-proseduurin aputaulukko, jonka koko on $O(n)$.
- Koska lomitusjärjestä on rekursiivinen, muistia menee lisäksi ajonaikaiseen pinoon.
- Rekursion syvyys on $O(\log n)$ ja yhden lomitusjärjestä-kutsun paikalliset muuttujat vievät vakiotilan, joten pinon tilavaativuus on $O(\log n)$.
- Siis aputaulukon tilavaativuus dominoi ja koko tilavaativuus on $O(n)$.
- Huomaa, että lomita ei ole rekursiivinen, joten aputaulukoita on käytössä vain yksi kerrallaan. (Käytännössä riittäisi luoda vain yksi aputaulukko ja käyttää aina sitä.)

Iteratiivinen lomitusjärjestäminen

- Tarkastelemalla taas sivun 122 kuvaa voidaan todeta, että kuvan yläpuoliskossa oikeastaan ei tapahdu mitään;
  erityisesti taulukon alkiot eivät liiku.

- Lomitusoperaatioiden kulku voidaan ennustaa täydellisesti katsomatta taulukon sisältöä:
  - ensin lomitetaan 1-alkoiset osataulukot
    taulu[2i] ja taulu[2i+1] kaikilla $i = 0, \ldots, n/2 - 1$
  - sitten lomitetaan 2-alkoiset osataulukot
    taulu[4i...4i+1] ja taulu[4i+2...4i+3] kaikilla $i = 0, \ldots, n/4 - 1$
  - sitten lomitetaan 4-alkoiset osataulukot
    taulu[8i...8i+3] ja taulu[8i+4...8i+7] kaikilla $i = 0, \ldots, n/8 - 1$
  - jne.

- Tässä on taas yksinkertaisuuden vuoksi oletettu, että jaot menevät aina tasan.

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

```
lomitusjärjestä2(taulu)
    pituus = 1          // lomitettavien lohkojen pituus
    while pituus < taulu.length
        alku = 0        // lomitettavan lohkoparin alkukohta
        while alku + pituus < taulu.length
            vasen = alku
            keski = alku + pituus - 1
            oikea = min{keski+pituus, taulu.length-1}
                        // koska viimeinen lohko voi olla vajaa
            lomita(taulu, vasen, keski, keski+1, oikea)
            alku = alku + 2 · pituus
        pituus = 2 · pituus
```

- Iteratiivinen lomitusjärjestäminen tekee tasan samat lomita-kutsut kuin rekursiivinen, joten senkin aikavaativuus on $O(n \log n)$.
- Myös aputaulukon käyttö ja siis tilavaativuus $O(n)$ on sama kuin rekursiivisessa versiossa.
- Iteratiivisessa versiossa kuitenkin vakiokertoimet voivat olla pienemmät.

Järjestämisalgoritmin vakaus

- Järjestämisalgoritmi on vakaa (stable), jos se säilyttää keskenään yhtä suurten alkioiden järjestyksen.
- Tämä voi olla relevanttia, jos järjestettäviin alkioihin liittyy muutakin dataa kuin se avain, jonka mukaan järjestetään.
- Oletetaan esim. että järjestettävän taulukon olioihin liittyy *etunimi* ja *sukunimi*. Halutaan järjestää ensisijaisesti sukunimen ja toissijaisesti etunimen mukaan.
- Menetellään seuraavasti:
  - Ensin järjestetään taulukko *etunimen* mukaan.
  - Sitten järjestetään taulukko *sukunimen* mukaan.
- Jos järjestämisalgoritmi on vakaa, saman *sukunimen* omaavat alkiot pysyvät keskenään samassa järjestyksessä kuin ensimmäisen järjestämisen jälkeen, eli *etunimen* mukaan.

- Tähän mennessä nähdyt lisäysjärjestäminen ja lomitusjärjestäminen ovat vakaita.
- Seuraavaksi nähtävä pikajärjestäminen ei ole vakaa.
- Javan valmiista järjestämismetodeista `Arrays.sort` ei ole (välttämättä) vakaa, mutta `Collections.sort` on.
- Edellä esitetyn kaltaiset tilanteet on kuitenkin Javassa parempi ratkaista määrittelemällä alkioluokalle `compareTo`-metodi, joka antaa suoraan oikean lopputuloksen; katso kurssikirjan lukua 3.4.

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

Pikajärjestäminen on toinen hajota ja hallitse -algoritmi. Korttipakka-analogiaa käyttäen se toimii seuraavasti:

1. Valitse pakasta yksi kortti jakoalkioksi (pivot).
2. Käy pakka läpi ja jaa se kahteen osaan:
   - Osaan 1 tulee jakoalkiota pienemmät kortit.
   - Osaan 2 tulee jakoalkiota suuremmat kortit.
3. Järjestä osa 1 soveltamalla tätä rekursiivisesti.
4. Järjestä osa 2 soveltamalla tätä rekursiivisesti.
5. Yhdistä osa 1, jakoalkiona ollut kortti ja osa 2.

Lomitusjärjestämiseen verrattuna ”hajota”-vaiheessa nähdään tässä enemmän vaivaa. Vastaavasti ”hallitse”-vaihe on hyvin helppo.

Jakoalkion valintaan ja yhtä suurten alkoiden käsittelyyn on erilaisia tapoja.

Seuraava algoritmi järjestää osataulukon taulu[vasen. . . oikea] tällä periaatteella:

pikajärjestä(taulu, vasen, oikea)

```
if vasen ≥ oikea return
k = jako(taulu, vasen, oikea)
pikajärjestä(taulu, vasen, k-1)
pikajärjestä(taulu, k+1, oikea)
```

Tässä siis aliohjelma jako

- palauttaa valitun jakokohdan
- siirtelee alkioita niin, että pienet ovat jakokohdan vasemmalla ja suuret oikealla puolella.

Katsomme seuraavaksi jako-aliohjelman toteutusta.

Eräs yksinkertainen toteutus jaolle on seuraava:

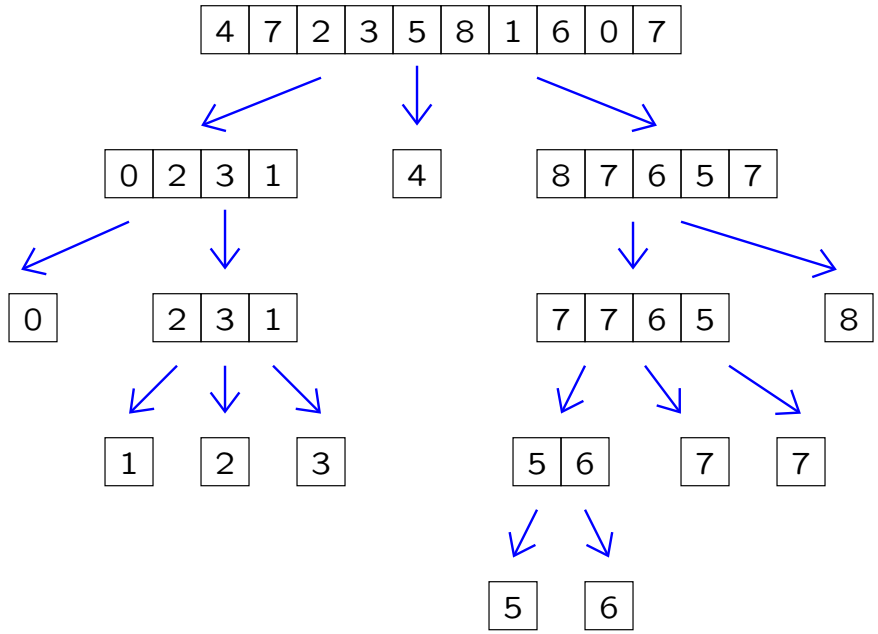
```
jako(A, vasen, oikea)
    k = vasen
    for i = vasen+1 to oikea
        if taulu[i] < taulu[vasen]
            k = k+1
            swap(taulu[i], taulu[k])
    swap(taulu[vasen], taulu[k])   // jakoalkio paikalleen
    return k
```

Tässä jakoalkioksi on valittu osataulukon ensimmäinen alkio.

Algoritmia kannattaa simuloida muutama kerta toiminnan ymmärtämiseksi. Perusidea on, että

- jakoalkio pysyy lähes loppuun asti paikassa taulu[vasen]
- taulu[vasen+1...k] sisältää jakoalkiota pienempiä lukuja
- taulu[k+1...i-1] sisältää jakoalkiota suurempia lukuja
- taulu[i...oikea] ovat vielä tutkimatta.

Kuvassa on havainnollistettu rekursiota taulukolla $[4, 7, 2, 3, 5, 8, 1, 6, 0, 7]$.
Nuolten sijainnit eivät ole merkitseviä.



Pikajärjestämisen aikavaativuus

- Aliohjelma jako koostuu oleellisesti yhdestä for-silmukasta. Aikavaativuus on $O(m)$, missä $m = b - a + 1$ on jaettavan osataulukon koko.
- Rekursion aikavaativuuden analysoiminen on paljon hankalampaa kuin lomitusjärjestämisellä, koska jakokohta vaihtelee taulukon sisällön mukaan.
- Jos olisi niin, että jako osuu aina tasan taulukon puoliväliin, niin aikavaativuudelle saataisiin oleellisesti sama rekursioyhtälö kuin lomitusjärjestämisessä:

$$T(n) = \begin{cases} c & \text{jos } n = 1 \\ 2T(n/2) + cn & \text{jos } n > 1. \end{cases}$$

  Aikavaativuudeksi tulisi $O(n \log n)$. Tämä on paras tapaus.
- Voidaan osoittaa (mutta ei tehdä tällä kurssilla), että myös keskimääräinen aikavaativuus on $O(n \log n)$, kun taulukon alkiot ovat kaikki eri suuruisia ja kaikki niiden järjestykset ovat yhtä todennäköisiä.

- Aikavaativuus $O(n \log n)$ pätee myös esim. jos jako menee aina niin, että pienempään osaan menee 10 % alkioista ja suurempaan 90 %.
- Rekursioyhtälöksi tulee tässä tapauksessa

$$T(n) = \begin{cases} c & \text{jos } n = 1 \\ T(\frac{9}{10} \cdot n) + T(\frac{1}{10} \cdot n) + cn & \text{jos } n > 1. \end{cases}$$

- Ratkaisu on edelleen $O(n \log n)$, mutta vakiokertoimet ovat suuremmat.
- Tämä seuraa siitä, että rekursion syvyydeksi tulee nyt

$$\log_{10/9} n = \frac{\log_2 n}{\log_2(10/9)} \approx 6{,}56 \cdot \log_2 n.$$

- Pikajärjestämisen pahin tapaus syntyy, jos jakoalkio on aina osataulukon pienin tai suurin alkio.
- Sivun 137 jako-aliohjelmassa jakoalkio on taulukon ensimmäinen, joten pahin tapaus syntyy erityisesti jos taulukko on jo valmiiksi on kasvavassa järjestyksessä!
- Rekursioyhtälöksi tulee

$$T(n) = \begin{cases} c & \text{jos } n = 1 \\ T(n-1) + T(1) + cn & \text{jos } n > 1, \end{cases}$$

mikä on suunnilleen sama kuin

$$T(n) = \begin{cases} c & \text{jos } n = 1 \\ T(n-1) + cn & \text{jos } n > 1. \end{cases}$$

Tästä seuraa

$$T(n) = cn + c(n-1) + c(n-2) + \ldots + c = c\sum_{i=1}^{n} i = c \cdot \frac{n(n+1)}{2},$$

eli pahimman tapauksen aikavaativuus on $O(n^2)$.

Jakoalkion valinta ja muut toteutuskysymykset

- Kuten olemme edellä todenneet, on suurilla syötteillä erittäin paha, jos järjestämisalgoritmi vie ajan $O(n^2)$.
- Pikajärjestäminen on kuitenkin erittäin hyvä algoritmi, jos pahimman tapauksen kaltaisia syötteitä ei esiinny usein.
- Tähän voidaan vaikuttaa valitsemalla hyvä menetelmä jako-aliohjelmaan.
- Pitää ottaa huomioon
  - miten jakoalkio valitaan
  - miten alkioita siirrellään.

Sivulla 137 esitetyssä yksinkertaisessa jako-toteutuksessa on (ainakin) kaksi ongelmaa:

- Kun jakoalkioksi valitaan aina osataulukon ensimmäinen alkio, niin valmiiksi järjestetty taulukko johtaa pahimpaan tapaukseen.
- Jos kaikki alkiot ovat yhtäsuuria, niin jakokohdaksi valikoituu taas taulukon alku (vaikka mikä tahansa muu kohta yhtäsuurien alkioiden keskellä olisi sallittu).

Nämä ovat ongelmallisia, koska käytännössä voi esiintyä ”melkein pahimpia” tapauksia:

- Taulukko voi olla melkein järjestyksessä, jos valmiiksi järjestettyyn taulukkoon on lisätty joitain alkioita.
- Taulukossa voi olla vain vähän eri arvoja (esim. halutaan lajitella jotain dataa sellaisen kentän mukaan, jolla on kolme mahdollista arvoa).

Tällöin parin jako-operaation jälkeen syntyy osataulukoita, jotka ovat vielä melko suuria ja edustavat pahinta tapausta.

- Selvästi parempi mutta vielä melko yksinkertainen jakoalgoritmi saadaan esim. tarkastelemalla taulukon ensimmäistä, viimeistä ja keskimmäistä kohtaa ja valitsemalla jakoalkioksi niistä keskimmäinen arvo.
- Jakoalkion valintaa ja muita pikajärjestämisen optimointeja on kehitetty ja tutkittu paljon.
- Javan `Arrays.sort` käyttää itse asiassa kahta jakoalkiota (dual-pivot).
- Pahin tapaus on aina $O(n^2)$, mutta hyvissä yleiskäyttöisissä pikajärjestystoteutuksissa pahimman tapauksen syötteet eivät vastaa mitään käytännön tilannetta.
- Yleinen optimointi on myös korvata rekursio lisäysjärjestämisellä, kun osataulukko on riittävän pieni (esim. $n < 20$).
- Hyvin optimoitu pikajärjestäminen on suosituin valinta yleiskäyttöiseksi järjestämisalgoritmiksi, koska sen vakiokertoimet ovat käytännössä pienet.

Pikajärjestämisen tilavaativuus

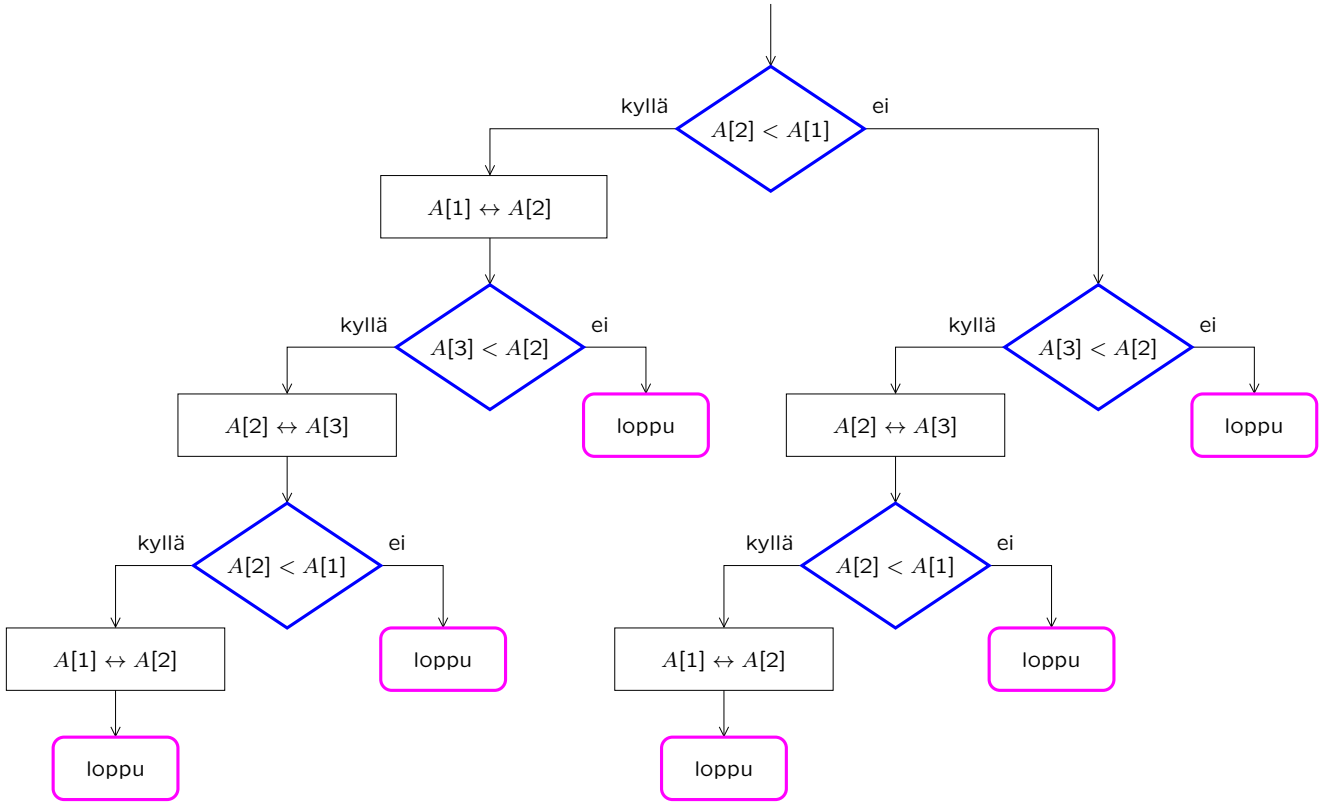
- Pikajärjestäminen ei tarvitse muuta apumuistia kuin aliohjelmakutsujen paikalliset muuttujat.
- Siis aktivaatiotietueen koko on vakio, joten muistitarve on O(rekursion syvyys).
- Tämä on
  - pahimmassa tapauksessa $O(n)$
  - keskimäärin $O(\log n)$.
- Toteutus on mahdollista optimoida niin, että muistivaativuus on $O(\log n)$ myös pahimmassa tapauksessa. (Sivuutamme yksityiskohdat.)
- Käytännössä jos muisti alkaa loppua, niin luultavasti suoritusaika on vielä pahempi ongelma.
- (Mutta Javassa voi olla tarpeen kasvattaa virtuaalikoneen pinoa oletusarvoa suuremmaksi.)

Alaraja vertailuihin perustuvalle järjestämiselle

- Edellä esitetyt järjestämisalgoritmit ovat kaikki vertailuihin perustuvia: ne käsittelevät järjestettäviä arvoja vain
  - testaamalla järjestysehtoja taulu$[i] <$ taulu$[j]$, taulu$[i] =$ taulu$[j]$, taulu$[i] >$ taulu$[j]$ jne.
  - vaihdoilla swap(taulu$[i]$, taulu$[j]$) ja yleisemmin sijoituksilla $x =$ taulu$[i]$ jne.
- Esimerkki muusta kuin vaihtoihin perustuvasta algoritmista voisi olla sellainen, joka olettaa alkioiden olevan kokonaislukuja ja esim. käyttää keskiarvoa (taulu$[1] +$ taulu$[n])/2$; tai testaa onko taulu$[i]$ parillinen; tai laskee taulukossa A esiintyvien arvojen lukumääriä.
- Osoitamme nyt, että mikä tahansa vertailuihin perustuva järjestämisalgoritmi suorittaa pahimmassa tapauksessa $\Omega(n \log n)$ vertailua.

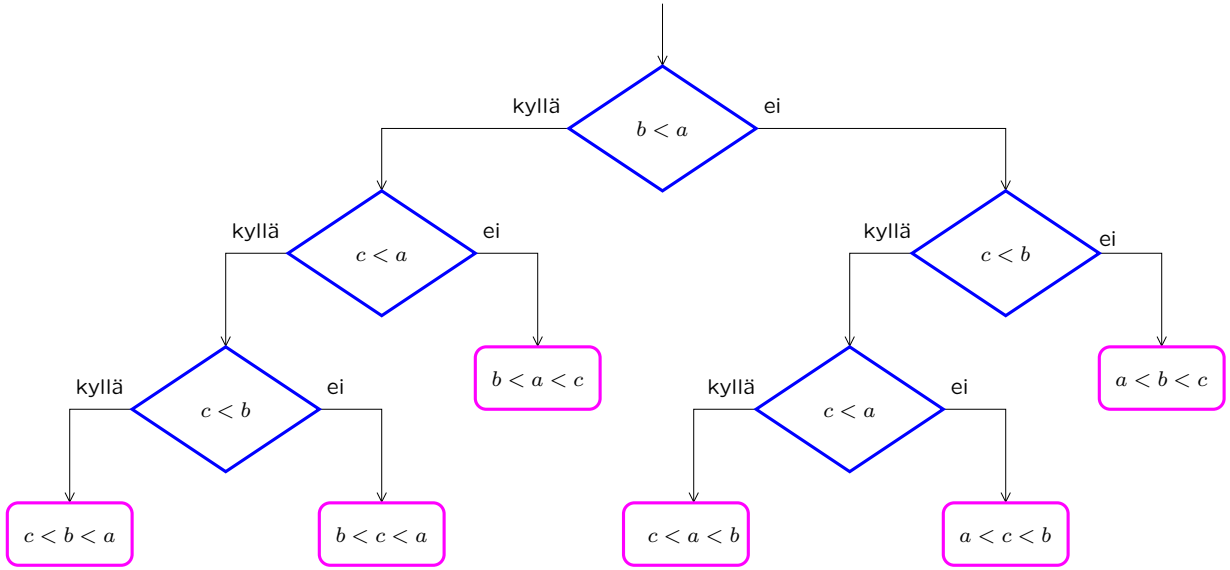
- Lähtökohtana on jokin vertailuihin perustuva järjestämisalgoritmi jollekin kiinteälle syötteen koolle n.

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



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

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



- Yksinkertaisuuden vuoksi oletetaan, että alkiot ovat erisuuria.

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

- Teemme nyt hyvin karkean arvion

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

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

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

Järjestäminen lineaarisessa ajassa

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

Sanomme tätä perusajatusta laskemisjärjestämiseksi.

Laskemisjärjestämisen perusversio pseudokoodina:

```
laskemisjärjestä1(A, k, n)
    for i = 0 to k
        C[i]= 0
    for j = 0 to n-1
        x = A[j]
        C[x] = C[x]+1
    y = 0
    for i = 0 to k
        for j = 1 to C[i]
            A[y] = i
            y = y+1
```

Tämä perusversio ei toimi, jos järjestettäviin alkioihin liittyy muutakin dataa kuin kokonaislukuarvo, jonka mukaan järjestetään.

- Algoritmi käy kerran läpi taulukon A, kahteen kertaan taulukon C ja sijoittaa luvun jokaiseen A:n paikkaan.
- Aikavaativuus siis $O(n + k)$.
- Jos $k = O(n)$ niin aikavaativuus on lineaarinen järjestettävän aineiston koon suhteen.
- Tilavaativuus on luonnollisesti $O(k)$.
- Yleistämme tämän tilanteeseen, jossa järjestettäviin kokonaislukuarvoihin voi liittyä muuta tietoa.

Pseudokoodi on seuraava:

```
laskemisjärjestä2(A, k, n)
1   for i = 0 to k
2       C[i] = 0
3   for j = 0 to n-1
4       x = A[j]
5       C[x] = C[x]+1
6   for i = 1 to k
7       C[i] = C[i]+C[i-1]
8   for j = n-1 downto 0
9       x = A[j]
10      B[C[x]-1]= x
11      C[x] = C[x]-1
12  for i = 0 to n-1
13      A[i] = B[i]
```

Toiminta-ajatus on selitetty seuraavalla kalvolla.

- Riveillä 1–5 lasketaan paikkaan C[x] alkion x esiintymiskerrat kuten edellisessä versiossa.
- Riveillä 6–7 taulukosta C muodostetaan kumulatiivinen summa, jolloin C[x] tulee sisältämään arvojen 0, 1, 2, . . . , x yhteismäärän.
- Siis jos B on järjestetty versio taulukosta A, niin arvon x viimeinen esiintymä tulee indeksille C[x]-1 (koska indeksointi on nollasta),
  toiseksi viimeinen indeksille C[x]-2 jne.
- Riveillä 8–11 kopioidaan arvot aputaulukkoon B tämän mukaisesti.
- Riveillä 12–13 kopioidaan arvot takaisin alkuperäiseen taulukkoon.

Reikäkorttijärjestäminen (radix sort)

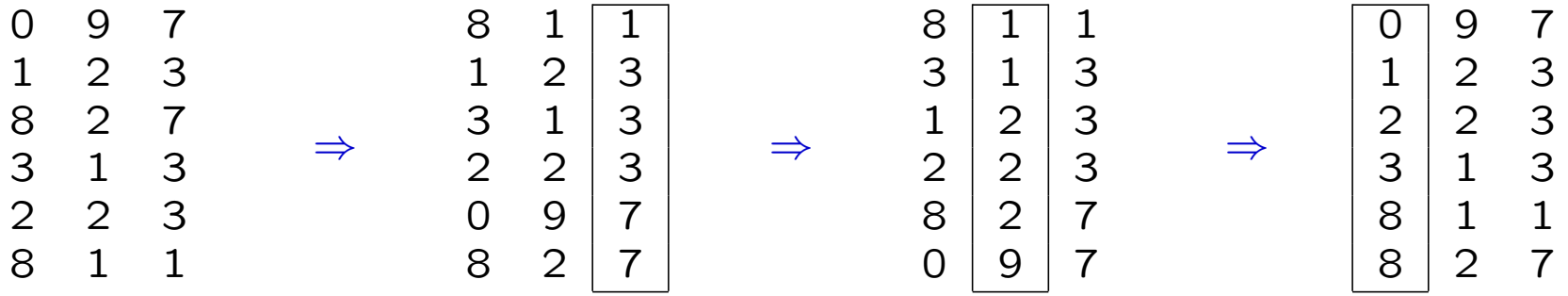
Tarkastellaan esimerkkinä kolmenumeroisten positiivisten kokonaislukujen järjestämistä. Tämä voidaan tehdä seuraavasti:

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

Kun kunkin vaiheen järjestäminen tehdään vakaasti, lopputuloksena luvut tulevat oikeaan järjestykseen (vrt. s. 133).

Tarvittaessa lisätään etunollia niin, että luvut ovat saman mittaisia.

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



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

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

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

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

4. Lista

Lista on yleisnimitys useille hieman erilaisille tietorakenteille, joissa ylläpidetään kokoelmaa tietoalkioita.

Tämän luvun jälkeen opiskelija

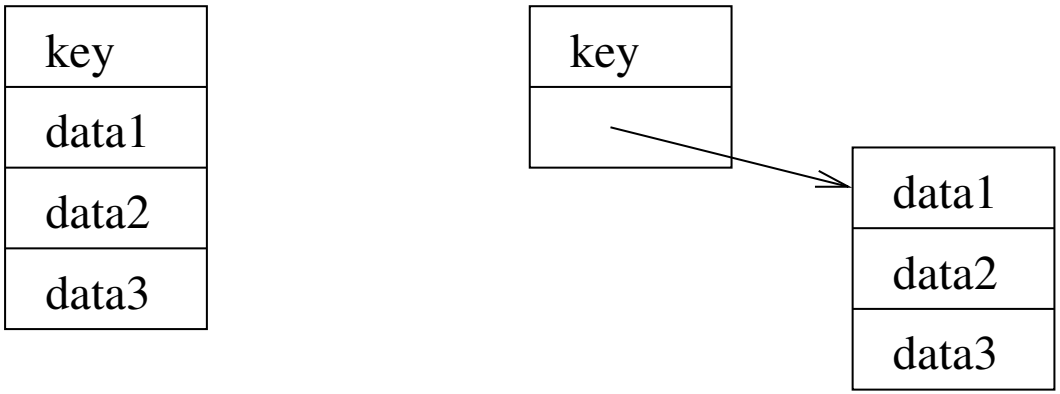
- osaa toteuttaa listan operaatioineen taulukkona ja linkitettynä
- tuntee listaoperaatioiden aikavaativuudet ja osaa soveltaa niitä
- osaa soveltaa listan erikoistapauksia pinoa ja jonoa
- tuntee tärkeimmät listarakenteisiin liittyvät Javan luokat ja niiden käytön.

Abstrakti tietotyyppi joukko

Abstraktina tietotyyppinä joukko on kokoelma tietoalkioita, joissa on

- avain (esim. opiskelijanumero), jonka avulla tietoalkiot identifioidaan
- muuta dataa (esim. etunimi, sukunimi, koulutusohjelma, . . .).

Avain ja data voivat olla osana samaa oliota, tai datakentät voidaan tallentaa omaksi oliokseen:



Jälkimmäinen tapa on järkevä esim. jos samaa dataa halutaan käsitellä useamman kuin yhden avaimen kautta

Tarkastelemme tässä dynaamista joukkoa, jonka alkioita voidaan muuttaa. Yleensä tarvitaan ainakin seuraavat operaatiot:

search(S, k): jos joukossa on avaimella k varustettu alkio, palauttaa viitteen tähän alkioon, muuten palauttaa viitteen NIL.

insert(S, x): lisää joukkoon alkion, johon x viittaa.

delete(S, x): poistaa tietueen, johon x viittaa.

Operaatioiden yksityiskohdat voivat vaihdella.

- Jatkossa oletamme yleensä, että insert-operaatiolla lisättävän avaimen alkiota ei ennestään ole joukossa. Jos halutaan sallia samalle avaimelle useita esiintymiä, pitää määritellä tarkemmin, miten niiden kanssa toimitaan.
- Tässä delete-operaatio saa valmiiksi parametrinaan poistettavan alkion x osoitteen joukon talletusrakenteessa. Jos poisto halutaan tehdä avaimen perusteella, se pitää ensin etsiä search-operaatiolla.

Muita usein esiintyviä operaatioita ovat esim.

size(S): palauttaa joukon S alkioiden lukumäärän.
isEmpty(S): palauttaa true, jos size(S) $= 0$.

Jos avaimille on määritelty suuruusjärjestys, myös seuraavat operaatiot tulevat kysymykseen:

min(S): palauttaa viitteen joukon pienimpään alkioon.
max(S): palauttaa viitteen joukon suurimpaan alkioon.
succ(S, x): palauttaa viitteen alkiota x seuraavaksi suurempaan alkioon; jos x on joukon suurin, palauttaa NIL.
pred(S, x): palauttaa viitteen alkiota x seuraavaksi pienempään alkioon; jos x on joukon pienin, palauttaa NIL.

Tässä ”pienin” jne. viittaavat avainten suuruusjärjestykseen.

Tässä luvussa käymmä yksityiskohtaisesti läpi joukon toteuttamisen taulukkolistana ja linkitettynä listana.

- Taulukkolista on yksinkertainen perustoteutus, joka sopii pienille tietomäärille ja tilanteisiin, joissa riittää rajoitettu operaatiovalikoima (esim. ei poistoja listan keskeltä)
- Linkitetty lista on perusesimerkki linkitetyistä rakenteista, joita tarvitaan myöhemmin monimutkaisemmissa tilanteissa.

Tehokkaampia yleiskäyttöisiä tietorakenteita suurille joukoille ovat

- hajaustustaulu (hash table) ja
- tasapainotettu hakupuu (balanced search tree),

joihin tutustumme seuraavissa luvuissa.

Taulukkolista (array list)

Taulukko on yksinkertainen useimpien ohjelmointikielten tarjoama tapa tallentaa kokoelma alkioita.

Tarkastellaan ensin yksinkertaista tapausta, jossa taulukkoon vain lisätään alkioita, eikä niiden keskinäisellä järjestyksellä ole väliä.

Joukon S esittämiseksi tarvitsemme kentät

elements: taulukko, johon alkiot talletetaan
next: kertoo seuraavan vapaan indeksin.

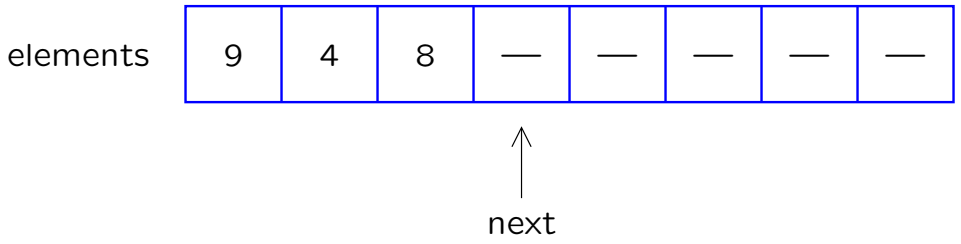
Aluksi next saa arvon 0.

Operaatio insert(S, x) sijoittaa alkion x positioon elements[next] ja kasvattaa next-arvoa yhdellä. Tämä toimii ajassa $O(1)$, kunhan taulukossa riittää tilaa.

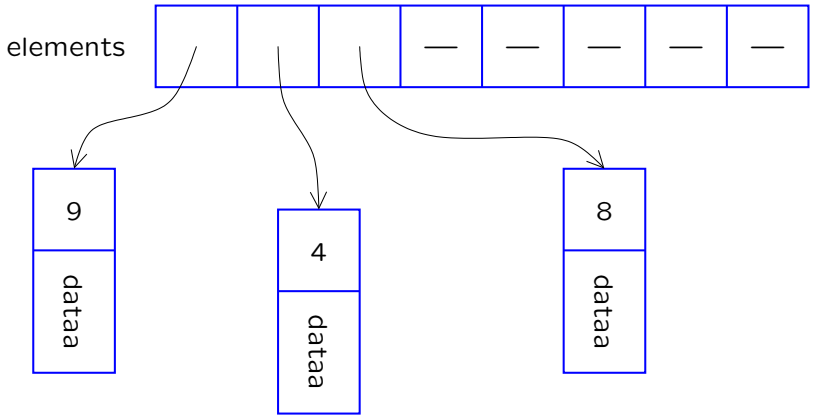
Taulukon positiot next ... elements.length $-$ 1 ovat käyttämättä.

Esimerkki

Lisätään aluksi tyhjään kokonaislukujoukkoon alkiot 9, 4 ja 8:



Jos avaimiin liittyy muuta dataa, taulukkoon tallennetaan tosiassa viitteet vastaaviin olioihin:



Search voidaan luonnollisesti toteuttaa peräkkäishaulla ajassa $O(n)$.

Taulukkolistaan on helppo toteuttaa myös viimeisen alkion poistaminen ajassa $O(1)$:

- edellisen sivun kuvassa deleteLast(S) palauttaisi osoittimen avaimen 8 sisältävään alkioon
- samalla next pienenee (ja asetetaan elements[next]=null).

Taulukkotototeutuksessa on ongelmana talletusalueen koon valinta, jos joukon suurinta kokoa ei tiedetä ennalta:

- jos elements-taulukko on turhan suuri, muistia menee hukkaan
- jos elements-taulukko on liian pieni, se tulee täyteen ja syntyy virhetilanne.

Taulukon kasvattaminen

Taulukkopohjaisissa listatoteutuksissa käytetään usein seuraavaa strategiaa:

- Varataan aluksi kohtuullisen kokoinen talletusalue.
- Kun talletusalue tulee täyteen, niin
  - varataan uusi talletusalue, joka on kaksi kertaa edellisen kokoinen
  - kopioidaan alkiot vanhalta talletusalueelta uudelle.

Näyttää, että kopioimisessa joudutaan tekemään paljon turhaa työtä.

- Kuitenkin kun joukon suurin koko on n alkiota, niin voidaan osoittaa, että kopioiminen n lisäysoperaation aikana yhteensä vie vain ajan $O(n)$.
- Siis kopioimisen aikavaativuus jaettuna tasan kaikille lisäysoperaatioille on $O(1)$ per operaatio. Sanomme, että lisäämisen tasoitettu (amortized) aikavaativuus on $O(1)$.
- Jotkin yksittäiset lisäysoperaatiot voivat kuitenkin viedä paljon aikaa, koska ne laukaisevat taulukon laajentamisen.

Tarkastellaan vielä taulukon kasvattamisen aikavaativuutta.

- Oletetaan, että taulukkoon lisätän kaikkiaan n alkiota.
- Siis talletusalueen koko m on enintään $2n$.
- Viimeisellä kasvatuksella taulukon kooksi tuli m, joten alkioita kopioitiin $m/2$ kappaletta.
- Toiseksi viimeisessä kasvatuksessa kopioitiin puolet tästä eli $m/4$ alkiota.
- Kaikkiaan alkioita kopiodaan korkeintaan $m/2 + m/4 + m/8 + \ldots = m$.
- Koska $m = O(n)$, kopioinnin aikavaativuudeksi kaikissa operaatioissa yhteensä tulee $O(n)$.

Tämä perustuu oleellisesti siihen, että jokaisella kasvatuskerralla taulukon koko kerrotaan jollain vakiolla. Vakion ei tietenkään tarvitse välttämättä olla juuri 2.

Haluttaessa voitaisiin vastaavasti poisto-operaatiossa puolittaa talletusalue, jos sen täyttöaste laskee esim. alle 25 %:n. (Usein kuitenkin nimenomaan taulukon suurin koko on tärkeintä, jolloin kerran varatun tilan vapauttaminen on vähemmän keskeistä).

Muutokset alussa ja lopussa

Voimme suoraviivaisesti yleistää taulukkotalletuksen sallimaan lisäys- ja poisto-operaatiot sekä alkuun että loppuun.

Toteutukseen tarvitaan

- taulukko alkioiden tallentamiseen, kuten edellä
- indeksi head, joka osoittaa listan ensimmäiseen alkioon
- indeksi tail, joka osoittaa talletusalueen ensimmäiseen tyhjään alkioon.

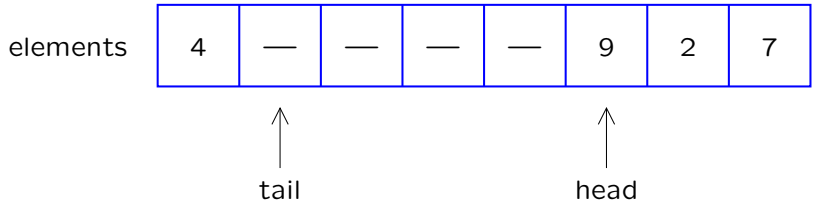
Indeksit head ja tail voivat ”kiertää ympäri”, jolloin tail $<$ head.

Toteutuksen yksityiskohdat voi tehdä muillakin tavoilla.

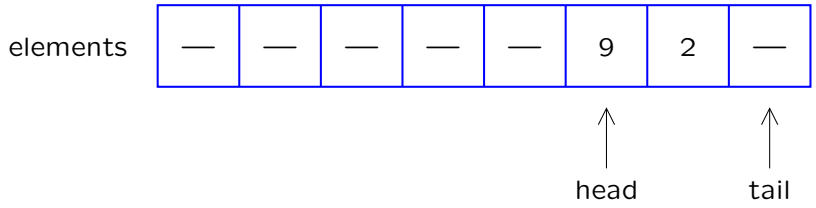
Taulukon kokoa voidaan hallita samalla kahdentamistekniikalla kuin edellä.

Esimerkki

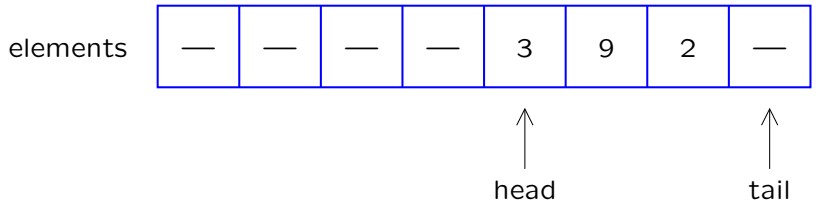
Suoritetaan addLast(4), addFirst(7), addFirst(2) ja addFirst(9):



Suoritetaan kaksi kertaa deleteLast:



Suoritetaan addFirst(3):



Linkitetty lista (linked list)

Linkitetyssä listassa emme varaa yhtenäistä aluetta alkioiden tallentamiseen, vaan varaamme jokaiselle alkiolle talletustilan erikseen. Tällaista talletustilaa sanotaan listan solmuksi (node).

Yksittäiset solmut ketjutetaan listaksi linkeillä.

Käytännössä solmu sisältää ainakin

- solmuun talletetun alkion avaimen
- linkin listassa seuraavaan solmuun.

Usein linkitys tehdään kahteen suuntaan, jolloin solmussa on lisäksi

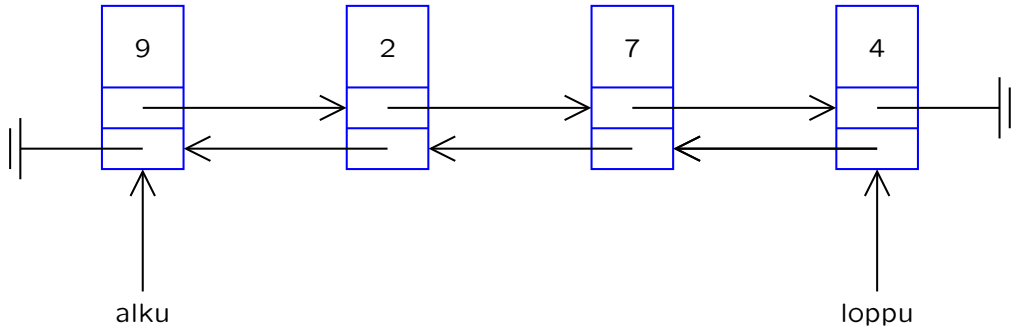
- linkki listassa edelliseen solmuun.

Lisäksi solmussa voi olla linkki

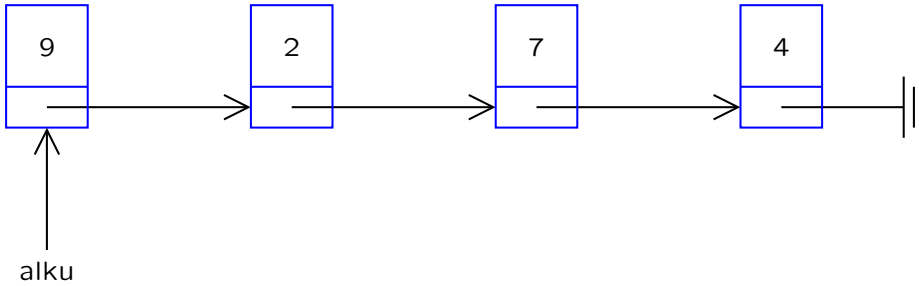
- muuhun dataan, joka avaimen lisäksi liittyy tietoalkioon;

yksinkertaisuuden vuoksi jätämme nämä datalinkit pois esimerkeistä.

Periaatekuva kahteen suuntaan linkitetystä listasta. ”Maadoitukset” listan päissä esittävät tyhjiä linkkejä (Javassa `null`, pseudokoodissa myös esim. NIL).



Yhteen suuntaan linkitetty lista:



(Linkit osoittavat kokonaisia solmuja; nuolen kärjen sijainti kuvassa on puhtaasti piirrostekninen seikka.)

Javassa linkit toteutetaan niin, että yhtä solmua esittävän `Solmu`-olion kenttinä on (kahteen suuntaan linkitetyssä listassa) kaksi muuta `Solmu`-oliota:

```
public class Solmu {

    public int avain;
    public Solmu seuraava;
    public Solmu edellinen;

    public Solmu(int avain, Solmu seur, Solmu edel) {
        this.avain = avain;
        this.seuraava = seur;
        this.edellinen = edel;
    }
}
```

Nyt listoja voidaan (jos välttämättä halutaan) luoda käsin tyyliin

```
Solmu t = new Solmu(9, null, null);
t.seuraava = new Solmu(2, null, t);
t.seuraava.seuraava = new Solmu(7, null, t.seuraava);
t.seuraava.seuraava.seuraava = new Solmu(4, null, t.seuraava.seuraava);
```

Järkevämpää tietysti on määritellä listalle luokka ja sille tarvittavat operaatiot:

```
public class LinkitettyLista {

    public Solmu alku;
    public Solmu loppu;

    public void LinkitettyLista() {
        alku = null;
        loppu = null;
    }

    public Solmu search(int x) {
        // ...
    }

    public void insert(Solmu s) {
        // ...
    }

    // ...
}
```

(Javassa kaikki tarvittava on toki valmiina; palaamme tähän.)

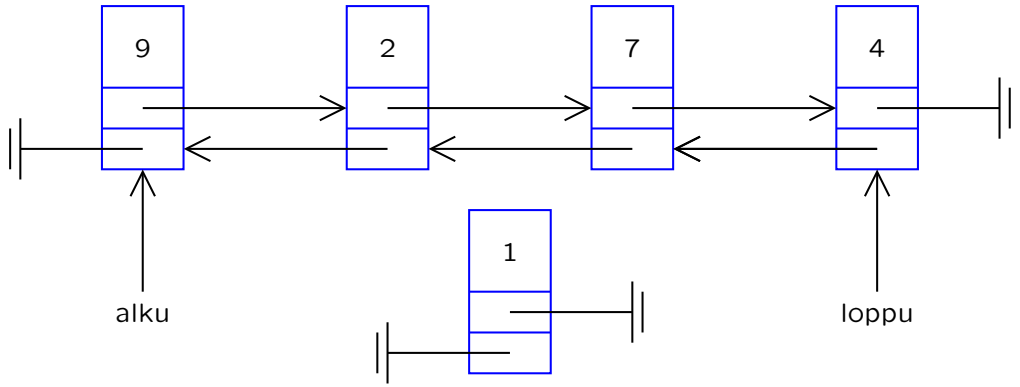
Hakuoperaatio toteutetaan käymällä listaa läpi alusta loppuun:

```
public Solmu search(int x) {
    Solmu s = alku;
    while (s != null) {
        if (s.avain == x) return s;
        s = s.seuraava;
    }
    return null;
}
```

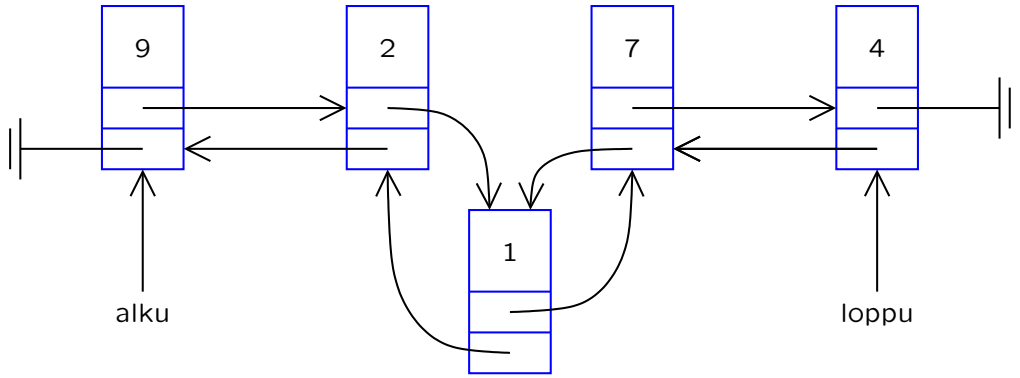
Tämä siis palauttaa osoittimen avaimen x sisältävään listan solmuun, tai `null` jos x ei ole listassa.

Myös lisäys- ja poisto-operaatioiden toteuttaminen on suoraviivaista, kun pidämme mielessä, miltä kuvan linkkeineen pitää näyttää ennen ja jälkeen operaation.

Esim. lisätään edelliseen kahteen suuntaan linkitettyyn listaan avainten 2 ja 7 väliin avain 1, joka on aluksi omana solmunaan:



Lopputulos:



Lisäyksen ja poiston toteutuksessa täytyy kuitenkin pitää mielessä useita erikoistapauksia:

- lista on tyhjä ennen operaatiota tai sen jälkeen
- listan alkuun tai loppuun tulee muutoksia.

Seuraava `insert`-toteutus lisää uuden solmun listan alkuun:

```
public void insert(Solmu s) {
    s.seuraava = alku;
    s.edellinen = null;
    if (alku != null) {
        alku.edellinen = s;
    } else {
        // lista oli tyhja
    loppu = s;
    }
    alku = s;
}
```

Voimme myös esim. tehdä lisäyksen parametrina annetun solmun jälkeen:

```
public void insertAfter(Solmu uusi, Solmu paikka) {
    // lisaa solmu "uusi" solmun "paikka" jalkeen
    uusi.seuraava = paikka.seuraava;
    uusi.edellinen = paikka;
    paikka.seuraava = uusi;
    if (uusi.seuraava != null) uusi.seuraava.edellinen = uusi;
    if (paikka == loppu) loppu = uusi;
}
```

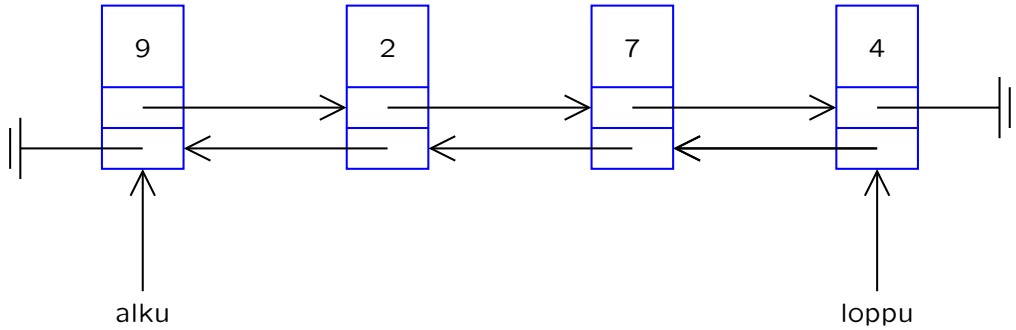
Nyt esimerkkilistamme voidaan luoda lisäämällä alkiot käänteisessä järjestyksessä aina listan alkuun:

```
LinkitettyLista lista = new LinkitettyLista();
lista.insert(new Solmu(4, null, null));
lista.insert(new Solmu(7, null, null));
lista.insert(new Solmu(2, null, null));
lista.insert(new Solmu(9, null, null));
```

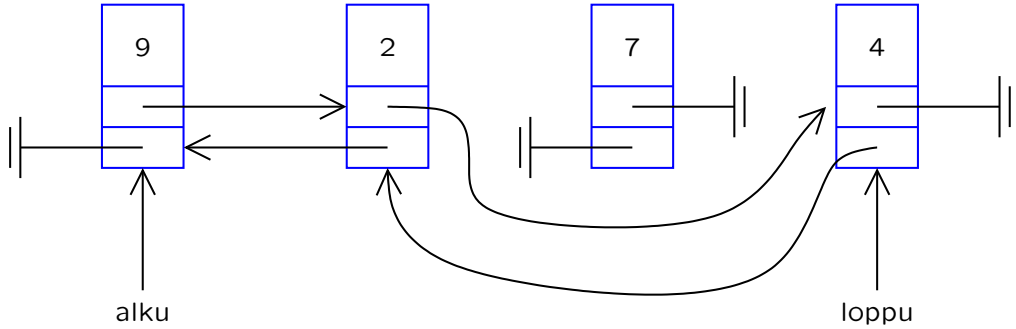
Lisätään avain 1 avaimen 2 perään:

```
Solmu s = lista.search(2);
lista.insertAfter(new Solmu(1, null, null), s);
```

Myös poisto kannattaa mieltää kuvan kautta:



Poistetaan avaimen 7 sisältävä alkio:



Huomaa, että yhteen suuntaan linkitetyn listan tapauksessa tämä olisi hieman konstikkaampaa, koska edelliseen alkioon ei suoraan pääse käsiksi.

Ottamalla taas huomioon erikoistapaukset saadaan

```
public void delete(Solmu pois) {
    if (pois.seuraava != null) {
        pois.seuraava.edellinen = pois.edellinen;
    } else {
        // poistettiin listan lopusta
        loppu = pois.edellinen;
    }
    if (pois.edellinen != null) {
        pois.edellinen.seuraava = pois.seuraava;
    } else {
        // poistettiin listan alusta
        alku = pois.seuraava;
    }
    pois.seuraava = null;    // siistimista
    pois.edellinen = null;   // roskienkeruuta varten
}
```

Edellisen sivun poisto tapahtuu kutsulla

```
lista.delete(lista.search(7));
```

Listarakenteiden vertailua

Listaoperaatioiden aikavaativuudet ovat seuraavat:

operaatio	taulukkolista	linkitetty lista
pääsy listan alkuun	$O(1)$	$O(1)$
pääsy listan loppuun	$O(1)$	$O(1)$
pääsy listan keskelle	$O(1)$	$O(n)$
lisäys/poisto listan alussa	$O(1)$	$O(1)$
lisäys/poisto listan lopussa	$O(1)$	$O(1)$
lisäys/poisto listan keskellä	$O(n)$	$O(1)$
haku avaimen perusteella	$O(n)$	$O(n)$

Edellä hahmoteltujen toteutusten valossa operaatiot toimivat vakioajassa, paitsi

- lisäys taulukkolistan keskelle vaatii muiden alkioiden siirtelyä tilan tekemiseksi; vastaavasti poiston jälkeen syntynyt tila pitää täyttää
- linkitetyssä listassa voi liikkua vain alkio kerrallaan alusta tai lopusta alkaen
- avaimen etsiminen vaatii aina listan läpikäyntiä alkio kerrallaan.

Jos taulukkolistan pitäisi avaimen mukaan järjestyksessä, niin hakuoperaation voisi toteuttaa ajassa $O(\log n)$ binäärihaulla.

- Tällöin aiheutuu kuitenkin paljon työtä lisäysten tekemisestä niin, että järjestys säilyy.
- Jos tällaiselle tuntuu olevan tarvetta, kannattaa harkita tasapainotettua hakupuuta (luku 6) tietorakenteeksi.

Linkitetyn listan keskellä tapahtuvien lisäys- ja poisto-operaatioiden vakioaikaisuus perustuu oletukseen, että osoitin oikeaan kohtaan saadaan parametrina.

- Jos esim. poistettava avain pitää ensin hakea Search-operaatiolla, niin kokonaisaikavaativuus on kuitenkin $O(n)$.

Verrattaessa taulukko- ja listatoteutuksen tehokkuutta käytännössä on tärkeää huomata, että tyypillisesti

linkitetyn listan vakiokertoimet ovat **paljon** suuremmat kuin taulukkolistan.

Tämä liittyy nykyaikaisten tietokoneiden toimintaperiaatteisiin:

- Tyypillinen listan käyttötapa on alkioiden käyminen läpi peräkkäin järjestyksessä.
- Tällöin taulukkototeutuksessa käydään keskusmuistissa vierekkäin sijaitsevia muistipaikkoja.
- Tämä tehostaa välimuistin käyttöä (ohjelmalla on hyvä muistiviitausten paikallisuus).
- Linkitetyssä listassa peräkkäiset listan alkiot eivät välttämättä ole tallennettuna lähekkäisiin muistipaikkoihin keskusmuistissa, jolloin tämä etu menetetään.

Vaikka linkitettyä listaa sellaisenaan ei usein olekaan syytä käyttää, sen periatteiden ymmärtäminen on välttämätöntä myöhemmin esiteltävien muiden linkitettyjen rakenteiden ymmärtämiseksi.

Pino ja jono

Pino (stack) ja jono (queue) ovat periaatteessa listan erikoistapauksia, joissa on käytössä hyvin rajatut operaatiot.

Niiden tunteminen ja tunnistaminen on tärkeää, koska ne esiintyvät monissa ilmiöissä tietojenkäsittelyssä ja ympäröivässä maailmassa.

Pino toimii periaatteella ''last in, first out'' (LIFO).

- UniCafessa puhtaat lautaset ovat pinossa.

Jono toimii periaatteella ''first in, first out'' (FIFO).

- UniCafessa opiskelijat ovat jonossa.

Pino on käytännössä lista, jossa lisäykset ja poistot kohdistuvat aina viimeiseen alkioon, eli pinotermeillä päällimmäiseen.

Pinolla S on käytössä on seuraavat operaatiot

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

push(S, x): vie alkion x pinon S päälle

isEmpty(S): palauttaa true, jos S on tyhjä eli ei sisällä yhtään alkiota.

Pino-operaatiot on helppo toteuttaa esim. taulukkolistan erikoistapauksena.

Perusesimerkki pinon soveltamisesta on sulkumerkkien tarkastaminen.

- Syötteenä on sulkumerkeistä `(){}[]` koostuva merkkijono.
- Halutaan tarkastaa, onko sulutus oikein muodostettu
  - jokaiselle alkusululle on loppusulku ja kääntäen
  - erityyppiset sulut eivät mene ristiin.
- Tehtävässä esiintyy LIFO-periaate: peräkkäisistä alkusuluista viimeinen suljetaan ensimmäisenä.

Ratkaisemme tehtävän pinoa käyttäen:

- Sulkumerkit käydään läpi yksi kerrallaan.
- Jos vastaan tulee alkusulku, viedään se pinoon.
- Jos vastaan tulee loppusulku, niin poimitaan pinosta alkusulku.
  - Jos nämä eivät vastaa toisiaan, sulutus menee ristiin.
  - Jos pino onkin tyhjä, lausekkeessa on liikaa loppusulkuja.
- Jos merkkijonon loputtua pinossa on alkioita, niin alkusulkuja oli liikaa.

Seuraavassa ratkaisussa oletetaan, että on määritelty luokka `Pino`, jossa pinon alkiot ovat tyyppiä `char`, ja sen metodit `push`, `pop` ja `isEmpty`.

Yksinkertaisuuden vuoksi mukana on vain merkit `(){}`.

```
public boolean tasapainoinen(String mj) {
  Pino pino = new Pino();
  for ( int i=0; i<mj.length(); i++){
     char c1 = mj.charAt(i);
     if ( c1 == '(' || c1 == '{' )
        pino.push(c1);
     else if ( c1 == ')' || c1 == '}' ) {
        if ( pino.isEmpty() ) // liian vähän ( tai {
           return false;
        char c2 = pino.pop();
        if ( c1 == ')' && c2 != '(' )      // väärä sulku
           return false;
        else if ( c1 == '}' && c2 != '{' ) // väärä sulku
           return false;
     }
  }
  if ( !pino.isEmpty() ) // liikaa ( tai {
     return false;
  else
    return true;
}
```

Jono on käytännössä lista, jossa lisäykset kohdistuvat listan loppuun ja poistot alkuun.

Jonolla Q on käytössä seuraavat operaatiot:

enqueue(Q, x): lisää alkion x jonon Q viimeiseksi
dequeue(Q): palauttaa arvonaan jonon Q ensimmäisen alkion ja poistaa sen jonosta
isEmpty(Q): palauttaa true, jos Q on tyhjä.

Myös jono-operaatiot on helppo toteuttaa taulukkolistan tai muun listatoteutuksen erikoistapauksena.

Koska erilaisia palveluja usein jaetaan asiakkaille FIFO-periaatteella, jonoilla on luonnostaan runsaasti sovelluksia

- käyttöjärjestelmissä jne.
- erilaisissa simulaatioissa.

Jos jonotuksessa halutaan suosia tiettyjä asiakkaita, tai simulaation tapahtuma-aikoja pitää voida säätää, tarkoituksenmukaisempi tietorakenne voi olla prioriteettijono, johon palataan myöhemmin.

Javan tarjoamat listarakenteet

Javassa on lukuisia valmiita luokkia erilaisille kokoelmille. Tämän luvun kannalta keskeisiä ovat

`ArrayList:` taulukkolista, jota voi ajatella vaihtuvankokoisena taulukkona
- muokkausoperaatioista vain lisäys loppuun toimii vakioajassa
- mielivaltaisen position lukeminen ja kirjoittaminen vakioajassa kuten taulukossa.

`ArrayDeque:` taulukkolista, jota voi käsitellä kummastakin päästä
- lisäykset ja poistot sekä alkuun että loppuun tehokkaita
- suositeltava toteutus esim. pinolle ja jonolle

`LinkedList:` kahteen suuntaan linkitetty lista
- lisäykset ja poistot myös keskellä tehokkaita, kun paikka annetaan iteraattorin kautta
- listan keskellä olevaan kohtaan pääseminen vie ajan $O(n)$.

ArrayList

Operaatiot ja niiden aikavaativuudet ovat niin kuin voisi olettaa lopusta kasvavalla taulukkolistalla.

- Konstruktorissa voidaan antaa talletusalueen koolle alkuarvo tai voidaan käyttää oletusarvoa (joka on 10).
- Taulukon kokoa kasvatetaan tarvittaessa sivujen 168–169 periaatteen mukaisesti.
- Perusoperaatiot
  - `add(arvo)`: lisäys loppuun
  - `get(indeksi)`: alkion lukeminen kohdasta `indeksi`
  - `set(indeksi, arvo)`: kohdan `indeksi` arvon vaihtaminen
  - `isEmpty()`, `size()`: kuten nimi sanoo.
- Vakiokertoimet pienempiä kuin `LinkedList`-toteutuksessa.
- Lue lisää kurssikirjasta ja Javan dokumentaatiosta.

ArrayDeque

"Kaksipäinen jono" (double-ended queue, mistä nimi); toiminta vastaa sivun 170 kuvaa

- taulukkolista, jonka koko tarvittaessa kasvaa kuten `ArrayList`illä
- perusoperaatiot
  - `addFirst` / `addLast`: lisäys alkuun / loppuun
  - `getFirst` / `getLast`: palauttaa arvonaan, mutta ei poista, listan ensimmmäisen / viimeisen alkion
  - `removeFirst` / `removeLast`: poistaa ensimmäisen / viimeisen alkion.
- ei lainkaan pääsyä listan keskelle

LinkedList

Perustoteutus kahteen suuntaan linkitetylle listalle

- `addFirst`, `addLast`, `removeFirst`, `removeLast` vakioajassa
- listan keskiosiin pääsee käsiksi iteraattorin avulla
- iteraattorin avulla myös lisäykset ja poistot keskelle vakioajassa
- mielivaltaisessa kohdassa olevan alkion hakeminen vie ajan $O(n)$.

Iteraattoria voi ajatella osoittimena, joka osoittaa kahden lista-alkion väliin.

- voi osoittaa myös kohtaan ennen ensimmäistä alkiota tai viimeisen alkion jälkeen, joten n-alkioisessa listassa on $n + 1$ mahdollista iteraattorin kohtaa
- metodi `next` siirtää iteraattorin seuraavaan väliin ja palauttaa arvonaan yli hypätyn alkion
- vastaavasti `previous` siirtyy taaksepäin
- `remove` poistaa viimeksi `next`- tai `previous`-operaatiolla palautetun alkion
- `add` lisää alkion `previous`- ja `next`-alkioiden väliin.

Huomaa, että edelliset ovat iteraattorin metodeja, ei listan.

Esimerkki Koodinpätkä

```
LinkedList<Integer> lista = new LinkedList<Integer>();
for (int i=0; i<n; i++) lista.add(i);
System.out.println(lista);
// Luodaan iteraattori ja alustetaan osoittamaan listan alkuun:
ListIterator<Integer> iteraattori = lista.listIterator(0);
iteraattori.next(); iteraattori.next();
System.out.println(iteraattori.next());
iteraattori.previous();
System.out.println(iteraattori.previous());
iteraattori.remove();
System.out.println(lista);
System.out.println(iteraattori.next());
iteraattori.remove();
System.out.println(lista);
```

tulostaa

```
[0, 1, 2, 3, 4]
2
1
[0, 2, 3, 4]
2
[0, 3, 4]
```

5. Hajautus

Hajautus (hashing) on tehokas ja käytännöllinen tapa toteuttaa dynaamisen joukon perusoperaatiot.

Tämän luvun jälkeen opiskelija

- tuntee hajautuksen peruskäsitteet
- tuntee tyypilliset hajautustaulun toteutusratkaisut
- osaa soveltaa hajautustauluja ohjelmointitehtävissä
- tuntee Javan hajautukseen liittyvät työkalut ja osaa käyttää niitä.

Esimerkki: shakkiasemien arvioiminen

- Shakkiohjelmat kokeilevat runsaasti erilaisia siirtovaihtoehtoja ja arvioivat syntyviä asemia (eli tilanteita).
- Arvio voi olla esim. ”+0.5”, jolloin koneen mielestä valkealla on noin puolen sotilaan arvoinen etu.
- Asemien arvioimiseen kuluvan ajan minimoiminen on olennaista laskentanopeudelle ja siis ohjelman pelivahvuudelle.
- Sama asema voi syntyä useilla eri tavoilla (pelaamalla samat siirrot eri järjestyksessä), mutta sitä ei haluta arvioida kuin kerran.
- Siis tarvitaan tietorakenne, joka tukee seuraavia:
  - Onko asema X jo arvioitu?
  - Jos on, niin mikä oli arvion lopputulos?
  - Merkitse muistiin asemalle X arvio v.

Havainto: ainakin periaatteessa tähän kelpaa abstrakti tietotyyppi joukko:

- avaimena asema shakkipelissä
- datana arvio asemasta.

Verrattuna aiemmin esitettyihin esimerkkeihin

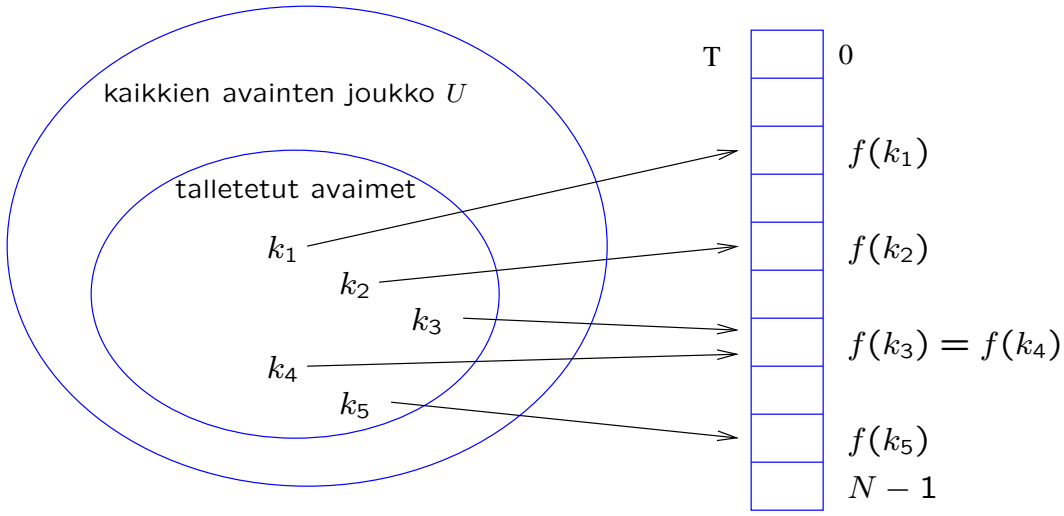
- potentiaalisten avainten joukko todella suuri, avaimet monimutkaisia
- search-operaation tehokkuusvaatimus tiukka
- myös insert ja ehkä delete tarvitaan.

Hajautuksen perusajatus

- Mahdollisten avainten joukko U on hyvin suuri.
- Mikä tahansa yksittäinen sovellus käyttää vain häviävän pientä osaa joukon U alkioista.
- Tallennetaan pareja $\langle avain, data \rangle$, missä $avain \in U$ ja $data$ voi olla mitä tahansa.
- Avaimet ja data talletetaan hajautustauluun (hash table), jonka indeksit ovat väliltä $0 \ldots N-1$.
- Hajautustaulun koko N on paljon pienempi kuin avainten lukumäärä $|U|$.
- Hajautusfunktio $f : U \to \{0, \ldots, N-1\}$ kertoo, mihin kohtaan hajautustaulua kukin avain pitäisi sijoittaa.

Hajautusfunktio f siis liittää jokaiseen avaimeen $x \in U$ taulukon indeksin $f(x) \in \{\, 0, \ldots, N-1 \,\}$.

- Voidaan ajatella, että hajautustaulussa on N lokeroa (bin, bucket), joihin varsinainen tieto talletetaan.
- Jos $f(x) = j$, se tarkoittaa, että avain x kuuluu lokeroon j.
- Koska lokeroita on N, mahdollisia avaimia on $|\,U\,|$, ja $N \ll U$, tulee pakostakin yhteentörmäyksiä (collision) eli tilanteita, joissa kaksi eri avainta kuuluu samaan lokeroon.
- Muodollisemmin yhteentörmäys on tilanne, jossa $x \neq y$ mutta $f(x) = f(y)$
- **Kysymys 1:** miten valitaan f niin, että yhteentörmäyksiä ei tule liikaa?
- **Kysymys 2:** kun yhteentörmäyksiä kuitenkin tulee, niin miten ne käsitellään?



Periaatekuva hajautuksesta.
Avaimilla k_3 ja k_4 on yhteentörmäys.

Yleisesti hyväksi havaittu ja myös Javan `HashSet`- ja `HashMap`-toteutusten pohjana oleva menetelmä on seuraava:

- Oletetaan ensin, että avaimet ovat luonnollisia lukuja: $U = \{0, \ldots, |U| - 1\}$. Palaamme hetken päästä yleisempään tapaukseen.
- Hajautusfunktiona jakojäännösmenetelmä (eli jakolaskumenetelmä): $f(x) = x \bmod N$ missä $x \bmod N$ tarkoittaa jakojäännöstä, kun kokonaisluku x jaetaan kokonaisluvulla N.
- Siis erityisesti $x \bmod N$ on joukossa $\{0, \ldots, N-1\}$ kaikilla x.
- Lokeroon j tulee näin ollen avaimet j, $j + N$, $j + 2N$, ...
- yhteentörmäykset käsitellään ketjuttamalla: lokeroon k tulevat avaimet (ja niihin liittyvä data) talletetaan linkitettyyn listaan
- varsinainen hajautustaulu $T[0 \ldots N-1]$ sisältää N osoitinta
- $T[j]$ osoittaa lokeroa j vastaavan linkitetyn listan alkuun.

Merkkijonojen hajauttaminen

Jos avaimet eivät ole kokonaislukuja, niin avaimesta x pitää ensin laskea kokonaislukuarvo $g(x)$ jollain sopivalla menetelmällä.
Tärkeä erikoistapaus on merkkijonojen hajauttaminen.

- Kun hajautettavana on k-merkkinen merkkijono $x = a_0 \ldots a_{k-1}$, niin ensin tulkitsemme sen kokonaislukujonoksi $c_0, \ldots, c_{k-1}$, missä c_i on merkin a_i numeroarvo (yleensä ASCII-koodi; esim. jos $a_i =$ 'a' niin $c_i = 97$).
- Eräs mahdollinen tapa saada tästä yksi kokonaisluku on laskea

$$g(x) = c_0 + c_1 + \ldots + c_{k-1}.$$

- Tämä on yksinkertainen ja ottaa huomioon merkkijonon kaikki merkit, mikä on hyvä.
- Ongelmana on, että se antaa saman arvon kaikille merkkijonoille, joissa on samat merkit eri järjestyksessä.
- Tällaisia voisi esiintyä esim. käsiteltässä jotain koodeja, joissa käytetään pientä määrää eri merkkejä.
- Tällöin tulee paljon yhteentörmäyksiä, mitä halutaan välttää.

Parempi tapa on polynominen hajautus:

$$g(x) = c_0 A^{k-1} + c_1 A^{k-2} + \ldots + A^2 c_{k-3} + A c_{k-2} + c_{k-1},$$

missä A on sopivasti valittu vakio. Tämä on kokeissa ja käytännössä havaittu hyväksi.

Koska

$$\begin{aligned} g(a_0 \ldots a_{k-1}) &= c_0 A^{k-1} + c_1 A^{k-2} + \ldots + A^2 c_{k-3} + A c_{k-2} + c_{k-1} \\ &= A \cdot (c_0 A^{k-2} + c_1 A^{k-3} + \ldots + A c_{k-2} + c_{k-2}) + c_{k-1} \\ &= A \cdot g(a_0 \ldots a_{k-2}) + c_{k-1}, \end{aligned}$$

saadaan tämä tehokkaasti lasketuksi seuraavasti:

$z = c_0$
for $i = 1$ to $k - 1$
    $z = A \cdot z + c_i$.

Toteutuksessa pitää ottaa huomioon, että laskettavat arvot kasvavat helposti kokonaistyypin arvoalueen ulkopuolelle eli tapahtuu aritmeettinen ylivuoto.

- Jos tiedetään, että tulosta aiotaan käyttää jakolaskumenetelmässä jakajana N, niin voidaan välttää suuret luvut ottamalla jakojäännös jokaisessa välivaiheessa:

$$z = (A \cdot z + c_i) \bmod N.$$

- Tämä antaa saman lopputuloksen kuin jos käytettäisiin mielivaltaisen tarkkuuden aritmetiikkaa ja otettaisiin jakojäännös vasta lopputuloksesta.
- Tällä kurssilla (esim. harjoitustehtävissä) käytämme yksinkertaisuuden vuoksi tätä menetelmää ylivuotojen välttämiseen.

Hajautuksen aikavaativuus

Ketjuttamalla toteutetun hajautustaulun kaikki operaatiot (insert, search, delete) noudattavat seuraavaa kaavaa:

1. Etsi annettua avainta k vastaava lista (siis käytännössä osoitin listan alkuun) $L = T[f(k)]$.

2. Suorita haluttu operaatio listalle L.

Aikavaativuus on siis $O(1 + |\,L\,|)$, missä $|\,L\,|$ on listan pituus ja lisävakio tulee hajautusfunktion laskemisesta ym.

Miten hajautustaulu eroaa siitä, että pareja $\langle avain, data \rangle$ talletettaisiin tavalliseen taulukkoon:

Tavallisen taulukon etuja:

- tavallinen taulukko käyttää muistia mahdollisimman tehokkaasti (jos koko on suunnilleen tiedossa etukäteen)
- tavallinen taulukko voidaan järjestää jne.

Hajautustaulun keskeinen etu:

- avaimesta nähdään vakioajassa sen oikea paikka taulukossa
- tosin avainta voidaan vielä joutua etsimään listasta.

Hajautustaulun tehokkuuden kannalta keskeinen suure on täyttösuhde (load factor):

- Oletetaan lokeroiden lukumääräksi N.
- Oletetaan taulukkoon talletetuksi n avainta.
- Täyttösuhde on nyt $\alpha = n/N$.

Tästä seuraa, että

- α on myös ylivuotoketjujen pituuksien keskiarvo
- jos nyt tehdään **oletus**, että $f(x)$ saa kunkin arvoista $0, \ldots, N-1$ samalla todennäköisyydellä, niin operaatioiden aikavaativuudet ovat odotusarvoisesti $O(1+\alpha)$
- erityisesti jos α on vakio (ts. taulukon koko N valitaan niin että se on verrannollinen alkioiden määrään n), niin operaatiot *search*, *insert* ja *delete* keskimäärin vakioajassa
- em. oletus on yleensä käytännössä riittävän lähellä totuutta hyvillä hajautusfunktioilla.

Yleensä pyritään pitämään täyttösuhde melko pienenä vakiona.

- Esim. Javan `HashMap` pitää oletusarvoisesti $\alpha \leq 0{,}75$.
- Täyttösuhteen pienentäminen nopeuttaa hakuja mutta lisää muistinkulutusta.
- Sopivan täyttösuhteen löytäminen voi vaatia kokeilua.

Ei ole realistista yrittää valita niin pieni täyttösuhde, että yhteentörmäyksiltä kokonaan vältyttäisiin

- Syntymäpäiväparadoksi: jos $n \geq \sqrt{2N}$ ja avaimet sijoitetaan tauluun tasaisen satunnaisjakauman mukaan, niin ainakin todennäköisyydellä $1/2$ tulee ainakin yksi yhteentörmäys.
- Siis yhteentörmäyksiä todennäköisesti alkaa tulla jo kun $\alpha = \sqrt{2/N} \ll 1$.

Jos alkioiden lukumäärä ei ollut etukäteen tiedossa ja täyttösuhde näyttää kasvavan liian suureksi, voidaan tehdä uudelleenhajauttaminen (rehash):

- varataan uusi kaksi kertaa suurempi taulu
- siirretään alkiot vanhasta taulusta uuteen yksi kerrallaan.

Tämä ei ole niin tehotonta kuin ensi silmäyksellä voisi luulla:

- voidaan osoittaa, että operaatiot vievät edelleen keskimäärin $O(1 + \alpha)$ (samoin oletuksin kuin edellä)
- kuitenkin jotkin yksittäiset operaatiot (ne jotka laukaisevat uudelleenhajauttamisen) voivat olla hyvin hitaita
- siirtelyyn kuluva aika voidaan analysoida oleellisesti samaan tapaan kuin ArrayListin tapauksessa.

Pahimmassa tapauksessa kaikilla tallennettavilla avaimilla on sama hajautusfunktion arvo.

- Tällöin kaikki avaimet päätyvät samaan lokeroon eli yhteen linkitettyyn listaan.
- Tällöin aikavaativuus on sama kuin linkitetyllä listalla, eli erityisesti search vie ajan $O(n)$.
- Käytännössä yleensä voidaan olettaa, että hajautusfunktio jakaa alkiot riittävän tasaisesti ja keskimääräisen tapauksen aikavaativuus $O(1)$ kaikille operaatioille pätee.
- On kuitenkin syytä tiedostaa, että on olemassa huonoja hajautusfunktioita ja vaikeita aineistoja.
- Erityisesti jos hajautusfunktion on julkisesti tiedossa, niin pahantahtoinen vastustaja voi aina laatia aineiston, jossa on paljon yhteentörmäyksiä.

Javan hajautusratkaisuja

Javassa on valmiina luokat `HashSet` ja `HashMap`.

HashSet on joukko, johon tietty avain joko kuuluu tai ei kuulu; avaimeen ei liity muuta dataa

HashMap liittää jokaiseen siihen tallennettuun avaimeen myös muuta dataa.

Javassa jokaisella oliolla on metodi `hashCode()`, joka palauttaa oliolle lasketun kokonaislukuarvon.

Hajautusfunktio lasketaan soveltamalla jakojäännösmenetelmää tähän `hashCode`-arvoon.

Yhteentörmäykset käsitellään ketjuttamalla. (Java 8:ssa jos lista kasvaa tietyn pituusrajan yli, siitä tehdään tasapainotettu hakupuu.)

Javan `String`-olioiden `hashCode` lasketaan polynomisella hajautuksella käyttäen arvoa $A = 31$.

- Javassa aritmeettinen ylivuoto ei kuitenkaan aiheuta poikkeusta, vaan lopputulos vain pyörähtää ympäri.
- Kun lasketaan `hashCode`, tämän annetaan rauhassa tapahtua.
- Esim. merkkijonon "ZZZZZ" `hashCode` on 85887450, mutta lisäämällä yksi 'Z' saadaan merkkijono "ZZZZZZ", jonka `hashCode` on -1632456256.

Jos määritellään omia luokkia, niin Javan luokasta `Object` peritty hashCode ei yleensä ole sopiva hajautukseen, koska se perustuu olion identiteettiin eikä sisältöön. Esim.

```
public class Pari {

    public int x;
    public int y;

    public Pari(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
    ...

Pari p1 = new Pari(1, 2); Pari p2 = new Pari(1, 2);
System.out.println(p1.hashCode() + " " + p2.hashCode());
System.out.println(p1.equals(p2));
```

tulostaa

```
705927765 366712642
false
```

Jos itse määriteltyjä luokkia haluaa käyttää hajautusavaimina, niin yleensä pitää siis uudelleenmääritellä (override) niiden `hashCode`-metodi.

Samalla myös `equal`-metodi pitää tällöin määritellä uudelleen.

Joillekin Javan luokille, kuten String, on valmiiksi määritelty sopiva `hashCode`.

Myös listatyypeille (ArrayList, LinkedList) on määritelty listan sisällöstä määräytyvä `hashCode`:

```
ArrayList<Integer> al = new ArrayList<Integer>(3);
LinkedList<Integer> ll = new LinkedList<Integer>();
al.add(0); al.add(1); al.add(2);
ll.add(0); ll.add(1); ll.add(2);
System.out.println(al.hashCode());
System.out.println(ll.hashCode());
```

tulostaa

```
29824
29824
```

Jakolaskumenetelmä hajautusfunktiona

Jakolaskumenetelmä on hyvä yleiskäyttöinen hajautusfunktio.

Koska hajautusfunktio on $k \bmod N$, talletusalueen koon N valinta voi vaikuttaa hajautusfunktion käyttäytymiseen.

Mille tahansa hajautusfunktiolle voidaan löytää esimerkkejä, joilla se toimii huonosti. Pyrkimys on kuitenkin valita sellainen hajautusfunktio, että käytännössä helposti esiintyvät tapaukset eivät aiheuta ongelmia.

Emme tässä tarkastele hajautusfunktioiden teoriaa syvällisemmin, mutta toteamme kaksi kirjallisuudessa usein esitettävää suositusta: valitaan N niin, että se on

- alkuluku
- ei lähellä mitään kakkosen kokonaislukupotenssia 2^p.

Esimerkki: mitä voi tapahtua, jos N ei ole alkuluku.

- Oletetaan, että hajautetaan merkkijonoja, joissa kaikissa viimeisenä merkkinä on välilyönti (ASCII-koodi 32).
- Javan hashCode antaa näille kaikille kokonaislukuarvoja muotoa $g(x) = 31 \cdot p + 32 = 31 \cdot (p + 1) + 1$.
- Oletetaan nyt, että talletusalueen koko N on jaollinen 31:llä: $N = 31q$ jollain $q \in \mathbb{N}$.
- Jos nyt $p + 1 = rq + s$, missä $0 \leq s \leq q$, niin

$$31 \cdot (p + 1) + 1 = 31 \cdot (rq + s) + 1 = r \cdot 31q + 31s + 1 = rN + 31s + 1.$$

- Siis hajautusfunktio $g(x) \bmod N$ voi saada vain arvoja $31s + 1$, missä $s \in \mathbb{N}$; kaikki alkiot sijoitetaan lokeroihin

$$1, 32, 63, 94, \ldots$$

ja suurin osa talletusalueesta jää käyttämättä.

Esimerkki: mitä voi tapahtua, jos $N = 2^p - 1$.

- Olkoon hajautettavana luonnollinen luku k.
- Jaetaan luvun k binääriesitys p bitin lohkoihin:
  - c_0 on luku, jota esittää p vähiten merkitsevää bittiä (bitit $0, \ldots, p-1$)
  - c_1 on luku, jota esittää bitit $p, \ldots, 2p-1$
  - c_2 on luku, jota esittää bitit $2p, \ldots, 3p-1$
  - jne., viimeisenä c_m.
- Suoraviivainen lasku (seuraavalla kalvolla) osoittaa, että nyt

$$k \bmod N = \left(\sum_{i=0}^{m} c_i \right) \bmod N.$$

- Siis hajautusfunktion $f(k)$ arvo riippuu vain k:sta muodostettujen p bitin mittaisten lohkojen summasta.
- Käytännössä voi hyvin tulla vastaan syötteitä, jotka on saatu esim. tällaisten lohkojen järjestystä vaihtelemalla.

Tarkastellaan nyt hajautusfunktion $k \bmod N$ laskemista tarkemmin.

Nähdään, että

$$\begin{aligned} k &= \sum_{i=0}^{m} (2^p)^i c_i \\ &= \sum_{i=0}^{m} (N+1)^i c_i \\ &= \sum_{i=0}^{m} (N^i + i \cdot N^{i-1} + \ldots + i \cdot N + 1) c_i \\ &= N \cdot \sum_{i=0}^{m} (N^{i-1} + i \cdot N^{i-2} + \ldots + i \cdot 1) + \sum_{i=0}^{m} c_i \\ &= qN + \sum_{i=0}^{m} c_i \end{aligned}$$

missä q on kokonaisluku.

Siis $k \bmod N$ riippuu vain lohkoista saatujen lukujen summasta $c_0 + \ldots + c_m$.

Hajautus: yhteenveto

Hajautus on yleensä käytännössä erittäin hyvin toimiva ratkaisu:

- Operaatioiden search, insert ja delete keskimääräinen aikavaativuus on $O(1)$ per operaatio.
- Tämä edellyttää hieman oletuksia hajautusfunktion toimivuudesta suhteessa hajautettaviin avaimiin.
- Normaalitilanteissa on helppo valita hajautusfunktio, jolla pahin tapaus $O(n)$ vältetään.

Hajautuksella on paljon muitakin sovelluksia kuin joukko-tietotyypin toteuttaminen

- sormenjälkitekniikat
- tietoturva (joskin hajautusfunktiolle asetettavat vaatimukset ovat erilaiset).

Hajautuksen haittana joidenkin sovellusten kannalta on, että se ei tue avainten järjestykseen perustuvia operaatioita min, max, succ ja pred.

- Jos näitä tarvitaan, niin seuraavassa luvussa esiteltävät tasapainoiset hakupuu ovat parempi vaihtoehto
- tasapainoiset hakupuut myös takaavat pahimman tapauksen aikavaativuudeksi $O(\log n)$ per operaatio.

6. Binäärihakupuu

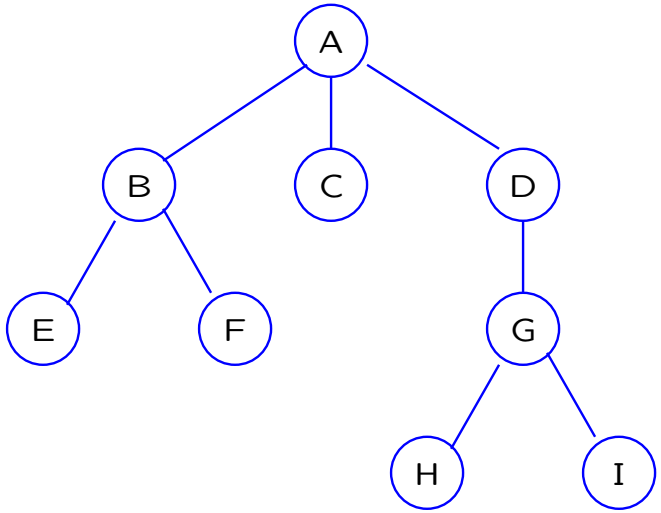
Binäärihakupuu on avainten järjestykseen perustuva tietorakenne, joka tarjoaa kaikki dynaamisen joukon operaatiot ajassa $O(\log n)$. Yleisemmin puut on tärkeitä paitsi tiedon tallentamisessa myös käsitteellisemmin esim. erilaisten hierarkioiden mallintamisessa.

Tämän luvun jälkeen opiskelija

- tuntee puihin liittyvät peruskäsitteet
- osaa toteuttaa binääripuita käsitteleviä algoritmeja
- tuntee binäärihakupuun operaatiot ja osaa soveltaa niitä
- osaa kuvailla binäärihakupuun tasapainottamisen eri operaatioiden yhteydessä.

Yleisten puiden peruskäsitteet

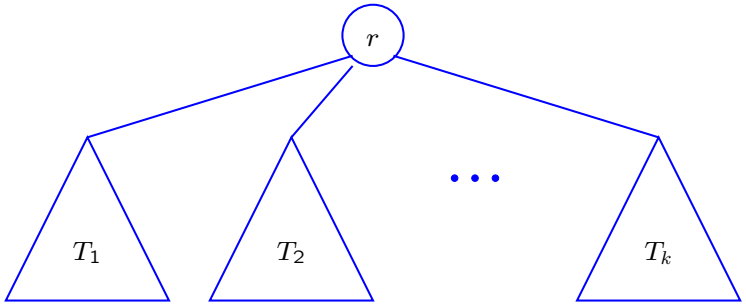
Alla on esimerkki puusta, jossa on 9 solmua A, B, C, . . . , I ja niitä yhdistämässä 8 viivaa eli kaarta. Solmu A on puun juuri.



Puulle voidaan esittää rekursiivinen määritelmä:

(Juurellinen) puu on sellainen kokoelma solmuja (node, vertex) ja niitä yhdistäviä kaaria (edge), että

- joko kokoelma on tyhjä tai
- yksi solmuista, r, on juuri (root), johon kaaret liittävät nolla tai useampia alipuita (subtree) $T_1, \ldots T_k$, jotka itsekin ovat puita.



Rekursiivinen määritelmä voi tuntua tässä vaiheessa melko abstraktilta, mutta se liittyy luontevasti myöhemmin nähtäviin puita käsitteleviin rekursiivisiin algoritmeihin.

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

- Tietojenkäsittelytieteessä puu piirretään juuri ylöspäin.

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

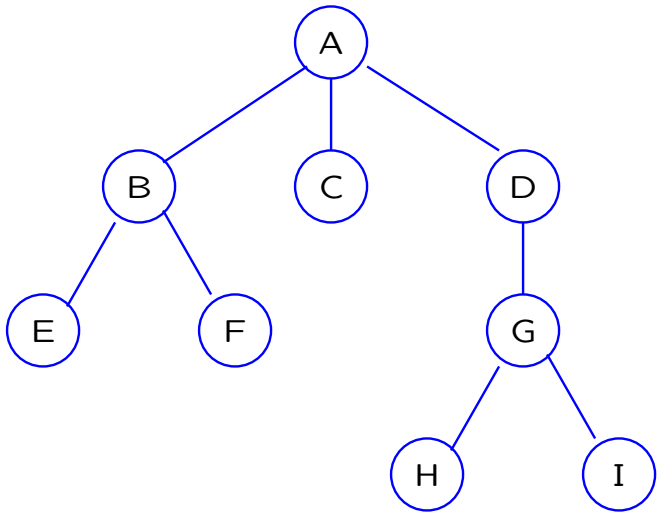
Määritellään muutamia puihin liittyviä käsitteitä

- Alipuiden $T_1, \ldots, T_k$ juuret ovat solmun r lapsia (child) ja r on lastensa vanhempi (parent); tässä r ja T_i viittaavat edellisen sivun määritelmään ja kuvaan
- Havainto: Juurta lukuunottamatta jokaisella solmulla on tasan yksi vanhempi. Siksi erityisesti kaarten lkm = solmujen lkm $-$ 1.
- Solmu, jolla ei ole lapsia, on lehti (leaf). Muut kuin lehtisolmut ovat sisäsolmuja.
- Solmut, joilla on yhteinen vanhempi, ovat sisaruksia (sibling).
- Isovanhempi ja lapsenlapsi määritellään luonnollisella tavalla.

- Solmun jälkeläisiä eli seuraajia (descendant) ovat
  - solmu itse ja
  - sen lasten jälkeläiset.

  Erityisesti lehtisolmu on itse ainoa jälkeläisensä.
- Solmun aitoja jälkeläisiä (proper descendant) ovat sen muut jälkeläiset kuin se itse.
- Vastaavasti määritellään esi-isä eli edeltäjä (ancestor) ja aito esi-isä.
- Solmu ja sen jälkeläiset muodostavat alipuun, jonka juuri kyseinen solmu on.

Esimerkki



- puun juuri A, jolla lapset B, C ja D
- puun lehtiä ovat solmut E, F, C, H ja I
- E, F ja G ovat solmun A lapsenlapsia ja A on solmujen E, F ja G isovanhempi
- B, C ja D ovat keskenään sisaruksia; samoin E ja F
- solmut D, G, H ja I muodostavat alipuun, jonka juurena on D
- solmun H aidot esi-isät ovat G, D ja A

Lisää määritelmiä:

- Polku (path) solmusta x_1 solmuun x_k on sellainen jono solmuja $x_1, x_2, \ldots, x_k$, että solmu x_{i+1} on aina solmun x_i lapsi tai vanhempi ja mikään solmu ei esiinny polulla useammin kuin kerran.

  Polun pituus on sen kaarien lukumäärä.

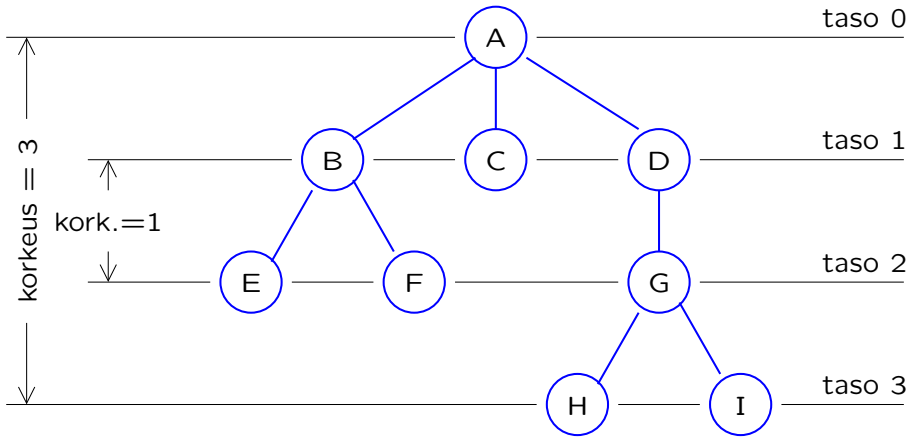
  Esim. edellisen sivun kuvassa (B, A, D, G) on polku solmusta B solmuun G; polun pituus on 3.

  Erikoistapaus: jokaisesta solmusta on nollan pituinen polku itseensä.

- Solmun x syvyys eli taso (depth, level) on polun pituus puun juuresta solmuun x.

- Solmun x korkeus (height) on pisimmän solmusta x johonkin lehteen vievän polun pituus eli solmusta x alkavassa alipuussa syvimmän lehden syvyys.

- Puun korkeus on sen juuren korkeus.



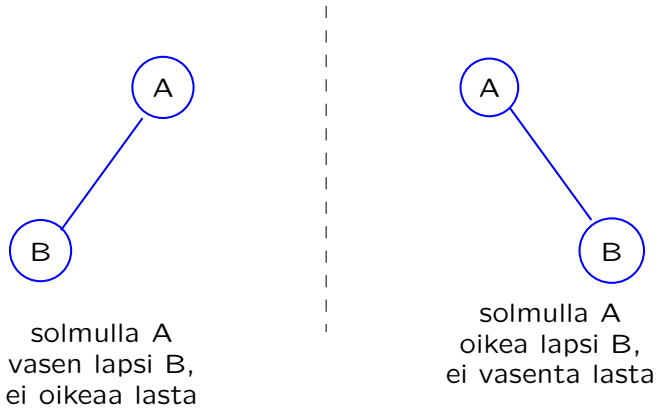
- Kuvaan on merkitty solmujen tasot ja solmujen A ja B korkeudet.
- Tason 1 solmujen B, C ja D korkeudet ovat 1, 0 ja 2.
- Koko puun korkeus on sama kuin juuren korkeus eli 3.

Määritelmät tulevat pikku hiljaa tutuksi, kun niitä ruvetaan käyttämään.

Binääripuu

Jos puun jokaisella solmulla on korkeintaan kaksi lasta, kysymyksessä on binääripuu (binary tree).

Binääripuussa erotetaan solmun vasen ja oikea lapsi, vaikka lapsia olisikin vain yksi. Siis seuraavat kaksi kuvaa esittävät eri binääripuita:



Vastaavasti puhutaan vasemmasta ja oikeasta alipuusta.

Myös tyhjä puu, jossa ei ole yhtään solmua, lasketaan binääripuuksi. Osoittautuu käteväksi sopia, että tyhjän puun korkeus on -1 (eli yksi vähemmän kuin yksisolmuisen puun korkeus).

Binääripuu voidaan toteuttaa samanlaisena solmuista koostuvana linkitettynä rakenteena kuin linkitetty lista.

Haluamme jatkossa tallentaa binääripuuhun (*avain*, *data*)-arvoja joukko-operaatioiden toteuttamiseksi. Tällöin jos olio *solmu* on tällainen linkitetyn puun solmu, niin sillä on kentät

solmu.avain: solmuun talletetun alkion avain
solmu.data: alkioon liittyvä muu tieto
solmu.vasen: linkki vasempaan lapseen
solmu.oikea: linkki oikeaan lapseen
solmu.vanhempi: linkki vanhempaan.

Osoitin vanhempaan ei ole välttämätön mutta yksinkertaistaa monia operaatioita.

Kuten aiempien tietorakenteiden yhteydessä, unohdamme yleensä pois data-kentät esityksen yksinkertaistamiseksi.

Solmujen ja tasojen määrä

Havainto 1: Binääripuun tasolla i on korkeintaan 2^i solmua.

- Tämä voidaan todeta induktiolla.
- Tasolla 0 on vain yksi solmu, nimittäin juuri, ja $2^0 = 1$.
- Oletetaan, että tasolla i on korkeintaan 2^i solmua.
- Tasolla $i+1$ voi olla korkeintaan kaksi lasta jokaiselle tason i solmulle eli korkeintaan $2 \cdot 2^i = 2^{i+1}$.

Havainto 2: Jos binääripuun korkeus on h, siinä on korkeintaan $2^{h+1} - 1$ solmua.

- Puussa nimittäin on korkeintaan 2^i solmua kullakin tasoista $i = 0, \ldots, h$.
- Solmujen lukumäärä on korkeintaan $\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$.

Lemma 1 Jos binääripuussa on n solmua ja sen korkeus on h, niin

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

Todistus: Yläraja n:lle tulee havainnosta 2.

Alaraja tulee siitä, että tasoilla $0, \ldots, h$ pitää jokaisella olla ainakin yksi solmu. □

Lemma 2 Jos binääripuussa on n solmua ja sen korkeus on h, niin

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

Todistus: Ratkaisemalla h lemman 1 epäyhtälöstä saadaan $h \leq n - 1$ ja $h \geq \log_2(n + 1) - 1$. □

Nämä arviot liittyvät binääripuilla toteutettujen joukko-operaatioiden aikavaativuuteen seuraavasti:

- Näemme, että binäärihakupuuna toteutettuun joukkoon kohdistuvat operaatiot voidaan toteuttaa ajassa $O(h)$, missä h on puun korkeus.
- Jos puun rakennetta ei rajoiteta, tästä tulee aikavaativuudeksi epätyydyttävä $O(n)$.
- On kuitenkin mahdollista pitää binäärihakupuu tasapainoisena niin, että n alkiota on pakattu melkein minimaaliseen määrään tasoja, eli $h = O(\log n)$.
- Siis tasapainottamalla saadaan operaatioiden aikavaativuudeksi $O(\log n)$.

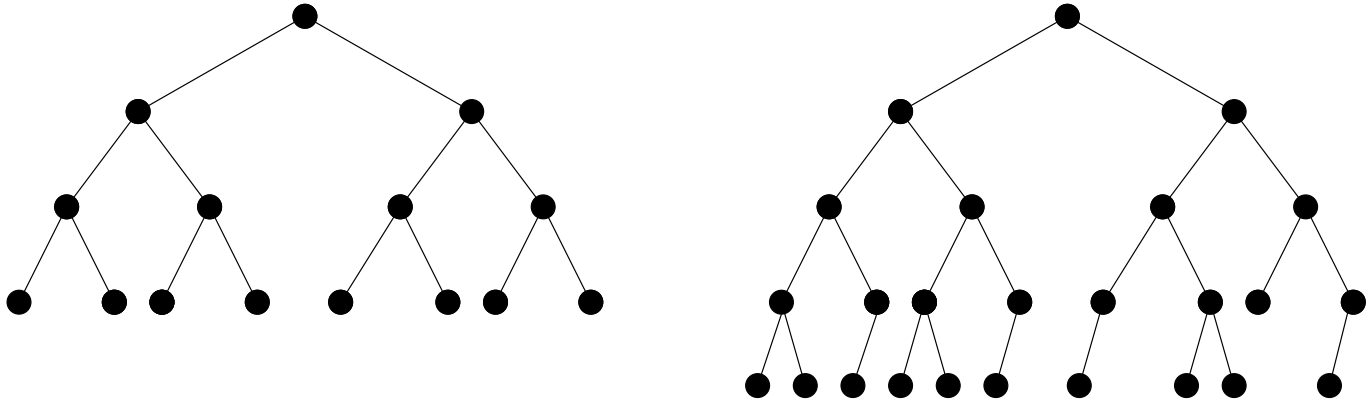
Käymme kuitenkin vielä läpi hiukan lisää binääripuiden perusteita, ennen kuin pääsemme joukko-operaatioiden toteutukseen.

Binääripuu on

- täysi (full), jos jokaisella solmulla on joko 0 tai 2 lasta
- täydellinen (complete), jos se on täysi ja kaikki lehdet ovat samalla tasolla.

Kuvassa vasemmalla täydellinen binääripuu, jossa siis annettuun korkeuteen on pakattu suurin mahdollinen määrä solmuja.

Jos puu on viimeistä tasoa lukuunnottamatta täydellisen binääripuun kaltainen, voidaan puuta nimittää melkein täydelliseksi (kuva oikealla).



(Nimitykset voivat hieman vaihdella eri lähteissä.)

Binääripuun läpikäynti

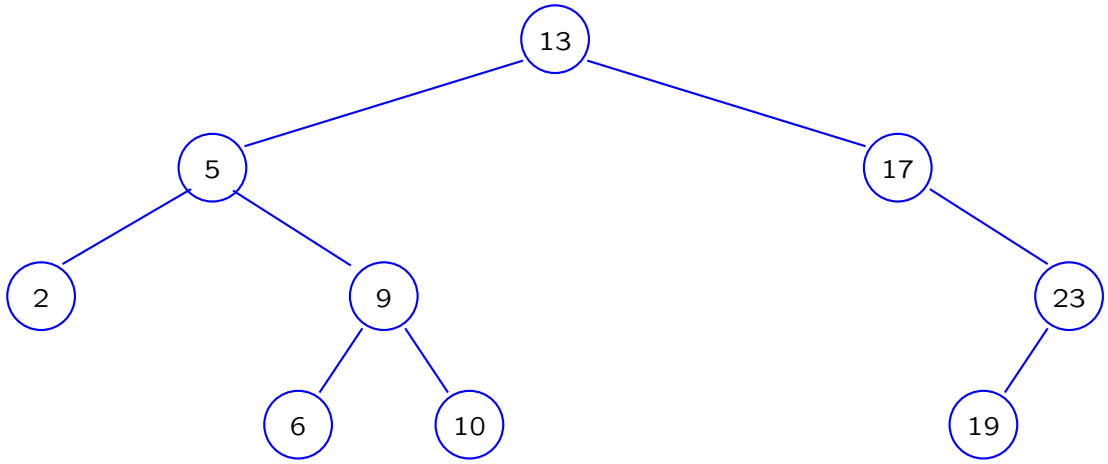
Binääripuita käsitellään usein rekursiivisilla algoritmeilla.

Seuraava proseduuri tulostaa binääripuun avaimet järjestyksessä vasemmalta oikealle, kun sille annetaan parametrina osoitin puun juureen:

```
tulosta(solmu)
    if solmu == null return
    tulosta(solmu.vasen)
    print(solmu.avain)
    tulosta(solmu.oikea)
```

Tätä solmujen käsittelyjärjestystä sanotaan sisäjärjestykseksi, koska ”print” on kahden rekursiivisen kutsun välissä.

Esimerkki



Proseduuri tulostaa

2 5 6 9 10 13 17 19 23.

Vaihtamalla avaimen tulostus toiseen paikkaan saadaan esijärjestys

```
tulosta(solmu)
    if solmu == null return
    print(solmu.avain)
    tulosta(solmu.vasen)
    tulosta(solmu.oikea)
```

ja jälkijärjestys

```
tulosta(solmu)
    if solmu == null return
    tulosta(solmu.vasen)
    tulosta(solmu.oikea)
    print(solmu.avain)
```

Tulostamalla edellinen puu esijärjestyksessä saadaan

13 5 2 9 6 10 17 23 19

ja jälkijärjestyksessä

2 6 10 9 5 19 23 17 13.

Näitä rekursiivisia läpikäyntejä voidaan soveltaa paljon muuhunkin kuin vain tulostamiseen.

Yleisperiaate siis on:

esijärjestys: käsitellään ensin solmussa oleva arvo ja sitten rekursiivisesti vasen ja oikea alipuu

sisäjärjestys: käsitellään rekursiivisesti vasen alipuu, sitten solmussa oleva arvo ja lopuksi rekursiivisesti oikea alipuu

jälkijärjestys: käsitellään ensin rekursiivisesti vasen ja oikea alipuu ja lopuksi solmussa oleva arvo.

Eri läpikäyntijärjestykset sopivat eri tilanteisiin.

Jälkijärjestys sopii algoritmeihin, joissa tietoa kerätään lehdistä alkaen.

Esimerkkinä on puun korkeuden laskeminen. Muistetaan, että tyhjän puun korkeudeksi sovittiin -1.

```
korkeus(solmu)
    if  solmu == null
        return -1
    return 1 + max{ korkeus(solmu.vasen), korkeus(solmu.oikea) }
```

Koko puun korkeus saadaan antamalla parametrina osoitin puun juureen.

Esijärjestys sopii tiedon levittämiseen juuresta kohti lehtiä.

Esimerkkinä oletetaan, että solmuissa on kokonaislukuarvoinen kenttä *data*. Muutetaan jokaisen solmun data-arvoksi sen esi-isien data-arvojen summa.

```
muuta(solmu, summa)
    if  solmu == null
        return
    summa = summa + solmu.data
    solmu.data = summa
    muuta(solmu.vasen, summa)
    muuta(solmu.oikea, summa)
```

Aluksi kutsutaan ”muuta(juuri, 0)”, missä juuri on osoitin puun juureen.

Binäärihakupuu (binary search tree)

Binäärihakupuu on eräs tapa toteuttaa abstrakti tietotyyppi joukko.

Kuten pian näemme, operaatiot toimivat ajassa $O(h)$, missä h on avaimista muodostetun binääripuun korkeus.

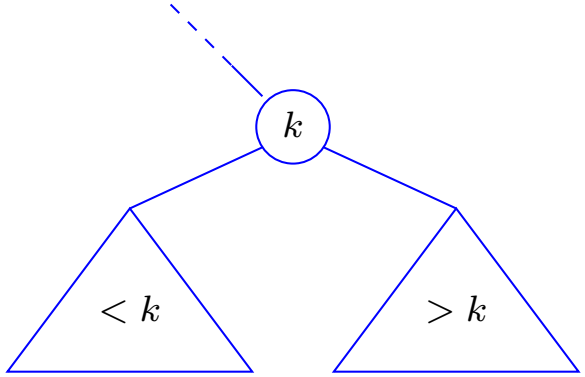
Myöhemmin esitettävällä tasapainottamisella pääsemme aikavaativuuteen $O(\log n)$, missä n on avainten lukumäärä.

Avaimilla pitää olla jokin hyvin määritelty suuruusjärjestys. Binäärihakupuu tukee myös suuruusjärjestykseen perustuvia operaatiota min, pred jne.

Binäärihakupuu on binääripuu, jonka jokaisessa solmussa on avain.

Avainten pitää toteuttaa binäärihakupuuehto: jokaisessa puun solmussa

- kaikki solmun vasemman alipuun avaimet ovat pienempiä kuin solmun avain ja
- kaikki solmun oikean alipuun avaimet ovat suurempia kuin solmun avain.



Oletamme tässä, että avain voi esiintyä joukossa vain kerran. Jos haluamme sallia useampikertaiset esiintymät, meidän pitää miettiä erikseen, mitä esim. search-operaation halutaan palauttavan.

Sanomme jatkossa binäärihakupuita lyhyesti hakupuiksi.

Merkintöjä

Oletamme hakupuualgoritmien pseudokoodissa, että jos *puu* on hakupuutyyppinen olio, niin sillä on kenttä *puu.juuri*, joka on osoitin puun juurisolmuun.

Tyhjällä puulla puu.juuri saa arvon null.

Juurisolmu ja muutkin solmut sisältävät kentät

solmu.avain: solmuun talletetun alkion avain
solmu.vasen: linkki vasempaan lapseen
solmu.oikea: linkki oikeaan lapseen
solmu.vanhempi: linkki vanhempaan

ja mahdollisesti *solmu.data*, johon liittyvät toimet sivuutamme jatkossa.

Uusia solmuja luodaan pseudokoodissa tyyliin

```
solmu = new puusolmu
solmu.avain = ...
```

Alkion etsiminen hakupuusta

Hakupuuehdon perusajatus on siinä, että annettua avainta etsiessä tiedetään, mihin suuntaan linkkejä pitää seurata.

Kutsu search(haettuAvain, puu.juuri) etsii puusta solmun, jossa on haettu avain.

Jos avain löytyy, palautetaan osoitin sen solmuun. Muuten palautusarvo on null.

```
search(haettuAvain, solmu)
    if (solmu == null) or (solmu.avain == haettuAvain) return solmu
    if haettuAvain < solmu.avain return search(haettuAvain, solmu.vasen)
    else return search(haettuAvain, solmu.oikea)
```

Haku binäärihakupuusta muistuttaa binäärihakua järjestetystä taulukosta.
Puun etuna on lisäysten ja poistojen tehokkuus, kuten pian näemme.
Siis hakupuu yhdistää järjestetyn taulukon ja linkitetyn listan hyvät puolet.

Haku on helppo toteuttaa myös ilman rekursiota:

```
search(haettuAvain, puu)
    solmu = puu.juuri
    while (solmu ≠ null) and (solmu.avain ≠ haettuAvain)
        if haettuAvain < solmu.avain
            solmu = solmu.vasen
        else
            solmu = solmu.oikea
    return solmu
```

Sekä rekursiivisessa että iteratiivisessa tapauksessa kuljetaan pahimmassa tapauksessa polku puun juuresta lehteen.

Jokaisessa polun solmussa tehdään vakiomäärä laskentaa.

Siis aikavaativuus on $O(h)$, missä h on puun korkeus. Pahimmassa tapauksessa (jos puu on hyvin epätasapainoinen) tämä on sama kuin $O(n)$, missä n on avainten lukumäärä.

Alkion lisääminen

Operaatio insert(avain, puu) lisää annetun avaimen puuhun.

Sovimme tässä, että jos avain on jo puussa, operaatio palauttaa null ja jättää puun ennalleen. Muuten palautusarvo on osoitin lisättyyn solmuun, jossa avain nyt on tallennetuna.

Lisäysoperaatio etenee aluksi aivan kuten avaimen haku. Jos hakupolku päättyy null-linkkiin, tiedetään, että tämä on paikka, josta avaimen pitäisi löytyä. Siis lisätään se siihen.

Algoritmissa muuttuja edellinenSolmu pitää kirjaa siitä, mikä solmu viimeksi ohitettiin hakupolulla. Kun polku loppuu, tiedämme, että avain tulee tämän solmun lapseksi.

Koska tyhjä puu käsitellään erikoistapauksena, edellinen solmu on aina olemassa.

Aikavaativuus on sama kuin search-operaatiolla eli $O(h)$, missä h on puun korkeus.

```
insert(lisättäväAvain, puu)
    if puu.juuri == null        // erikoistapaus: tyhjä puu
        puu.juuri = new puusolmu
        puu.juuri.avain = lisättäväAvain
        puu.juuri.vasen = puu.juuri.oikea = puu.juuri.vanhempi = null
    else                        // puu ei ole tyhjä
        solmu = puu.juuri
        while (solmu ≠ null) and (solmu.avain ≠ lisättäväAvain)
            edellinenSolmu = solmu
            if lisättäväAvain < solmu.avain
                solmu = solmu.vasen
            else
                solmu = solmu.oikea
        if solmu == null        // lisättävä avain ei ole puussa
            solmu = new puusolmu
            solmu.avain = lisättäväAvain
            solmu.vanhempi = edellinenSolmu
            solmu.vasen = solmu.oikea = null
            if lisättäväAvain < edellinenSolmu.avain
                edellinenSolmu.vasen = solmu
            else
                edellinenSolmu.oikea = solmu
            return solmu
        else return null        // lisättävä avain oli jo puussa
```

Pienin ja suurin alkio

Hakupuun pienin alkio löydetään seuraamalla linkkejä vasemmalle niin pitkälle kuin pääsee. Palautusarvo on osoitin pienimmän avaimen solmuun, tai tyhjällä puulla null. Tämäkin algoritmi toimii ajassa $O(h)$.

```
min(puu)
    if puu.juuri == null return null
    solmu = puu.juuri
    while (solmu.vasen ≠ null)
        solmu = solmu.vasen
    return solmu
```

Suurin alkio löytyy vastaavasti oikealta:

```
max(puu)
    if puu.juuri == null return null
    solmu = puu.juuri
    while (solmu.oikea ≠ null)
        solmu = solmu.oikea
    return solmu
```

Emme tämän jälkeen esitä yksityiskohtaisia pseudokoodeja, koska niistä tulee lukuisten käsiteltävien tapausten johdosta monimutkaisia.

Selitämme algoritmien toimintaperiaatteet kuvallisesti, minkä perusteella pseudokoodin voi halutessaan kirjoittaa edellisten mallien mukaisesti.

Seuraaja ja edeltäjä

Kun x on sallittu avaimen arvo, niin sen seuraaja-arvo on pienin sellainen puussa oleva avain k, että $k > x$. Emme kuitenkaan vaadi, että avaimen x pitää todella esiintyä puussa.

Sovimme, että succ(x, puu) palauttaa osoittimen siihen puun solmuun, jossa arvon x seuraaja-arvo on. Jos puussa ei ole suurempia arvoja kuin x, succ palauttaa null.

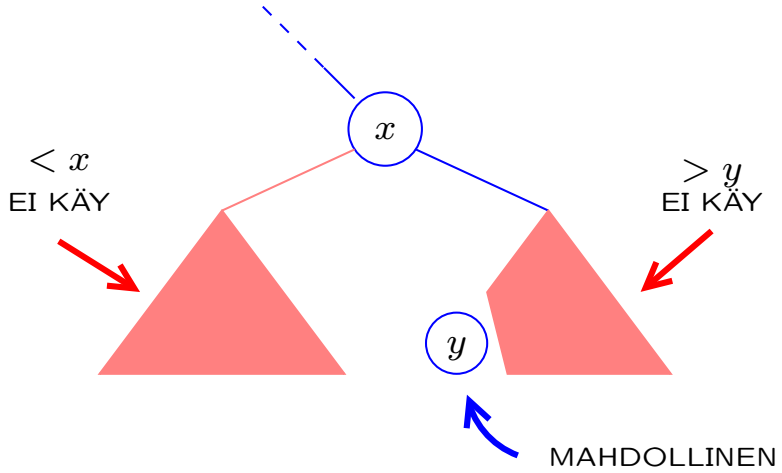
Vastaavasti arvon x edeltäjäarvo on suurin puussa oleva avain k, jolla $k < x$. Kutsu pred(x, puu) palauttaa osoittimen edeltäjäsolmuun, tai null jos x:ää pienempiä arvoja ei ole.

Käymme läpi seuraajan etsimisen. Edeltäjää etsittäessä vaihdetaan vasen ja oikea keskenään, samoin pienempi ja suurempi kuin.

Seuraajan succ(x) etsimisessä käydään läpi sama juuresta alkava hakupolku kuin avainta x etsittäessä. Perustelemme succ-algoritmin toiminnan eliminointimenettelyllä: selvitämme ensin, missä osissa puuta seuraaja ainakaan ei voi olla.

Siis etsitään arvon x seuraajaa. Tarkastellaan ensin tapausta, että x löytyy puusta: $x = s$.avain jollain solmulla s.

- Koska solmun s vasemman alipuun avaimet ovat pienempiä kuin x, ne eivät voi olla seuraajia.
- Oikean alipuun avaimista pienin (merkitty alla y) voi olla seuraaja. Muut eivät kelpaa, koska y on niiden ja x:n välissä.

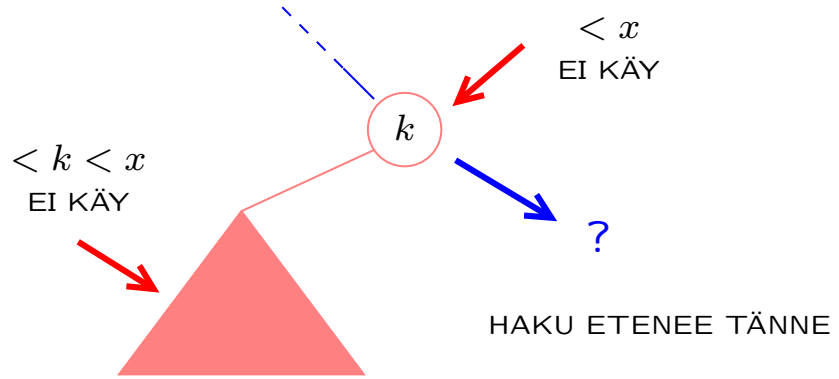


Tarkastellaan nyt solmua s juuresta x:ään vievällä hakupolulla. Tilanteeseen ei vaikuta, löytyykö x lopulta puusta.

Merkitään $k = s$.avain. Jos haku etenee solmusta s oikealle, niin

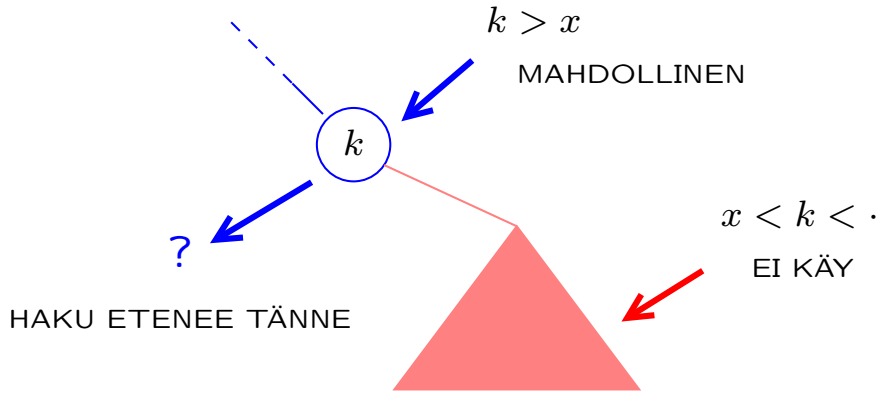
- $x > k$, joten k ei voi olla seuraaja
- alipuussa s.vasen avaimet ovat pienempiä kuin k ja siis nekin pienempiä kuin x; eivät voi olla seuraajia.

Siis mahdollisia seuraajia on ainoastaan oikeassa alipuussa, jonne haku etenee; ohitetut solmut eivät tule kysymykseen seuraajina.



Tarkastellaan nyt tilannetta, kun haku etenee solmusta s vasemmalle. Merkitään taas $k = s$.avain.

- Koska $k > x$, niin k voisi olla seuraaja.
- Solmun s oikean alipuun avaimet eivät tule kysymykseen, koska k on niiden ja x:n välissä.



Yhteenvetona voidaan todeta, että jäljellä on enää

- ne x:ään johtavan hakupolun solmut, joista haku jatkui vasemmalle; olkoot näiden avaimet $k_1, \ldots, k_m$ siinä järjestyksessä kun ne on kohdattu
- x:n oikean alipuun minimi y (jos kyseinen alipuu on olemassa).

Kaikki muut puun solmut ovat alipuissa, jotka on todettu mahdottomiksi.

Koska avaimesta k_i on aina menty vasemmalle, pätee $k_1 > k_2 > \ldots > k_m > x$.

Koska y on x:n alla ja siis k_m:stä vasemmalle, mutta x:stä oikealle, pätee $x < y < k_m$.

Siis x:ää seuraava avain on

- x:n oikean alipuun minimi y, jos sellainen on olemassa
- muuten viimeinen hakupolun avain k_m, josta mentiin vasemmalle, jos sellainen on olemassa
- muuten ei määritelty (koska puussa ei ole x:ää suurempia arvoja).

Alkion poistaminen

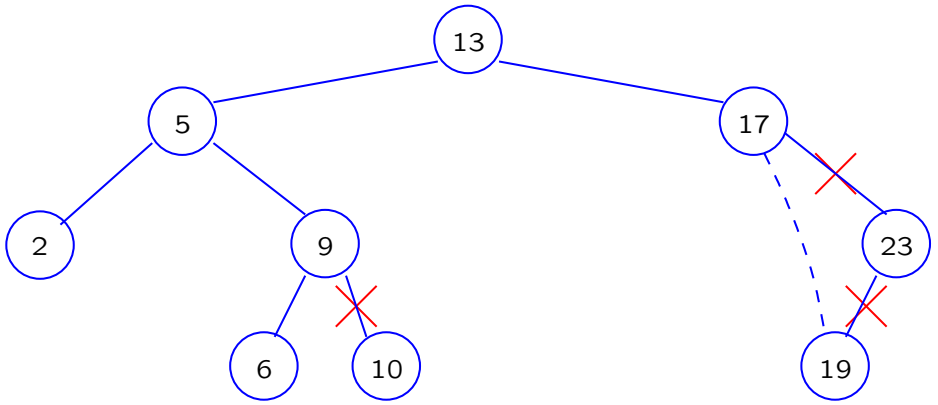
Oletetaan annetuksi osoitin solmuun, joka halutaan poistaa. (Tämä pitää tarvittaessa ensin etsiä search-operaatiolla.)

Poisto-operaation delete(solmu, puu) jakautuu kolmeen eri tapaukseen sen mukaan, kuinka monta lasta poistettavalla solmulla on.

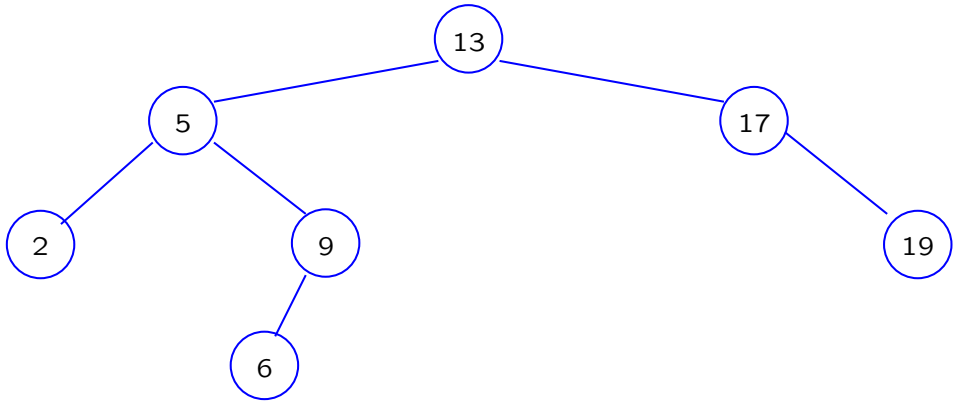
Jos lapsia on 0 eli poistettavana on lehti, sen voi yksinkertaisesti poistaa vaihtamalla siihen osoittavan linkin arvoksi null.

Jos lapsia on 1, niin olkoon p poistettavan solmun vanhempi ja q poistettavan solmun ainoa lapsi. Linkitetään solmu q suoraan solmun p lapseksi, ja poisto on tehty. Jos poistettavana on juuri, jolloin vanhempaa p ei ole, tehdään lapsesta q uusi juuri.

Esimerkki: Poistetaan solmut, joissa on avaimet 10 ja 23



Lopputulos



Jos poistettavalla solmulla on 2 lasta, solmua ei suoraan voi poistaa puurakenteesta: siltä jäisi 2 orpoa lasta, mutta sen vanhemmasta vapautuisi tilaa vain yhdelle.

Siksi etsimme ensin poistettavan solmun seuraajan.

- Koska poistettavalla solmulla on kaksi lasta, edellä esitetyn mukaan seuraaja on vasemman alipuun minimi.

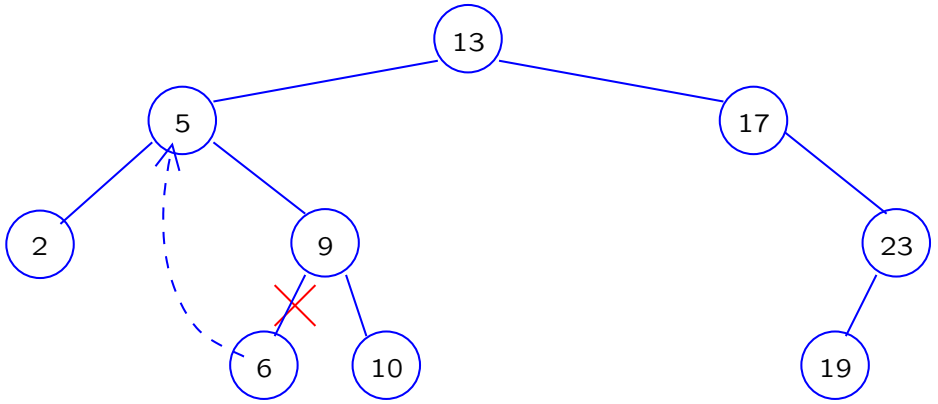
Sitten siirrämme avaimen seuraajasolmusta poistettavaan solmuun.

- Koska avaimen ja sen seuraajan välissä ei ole avaimia, tämä ei riko hakupuuominaisuutta.

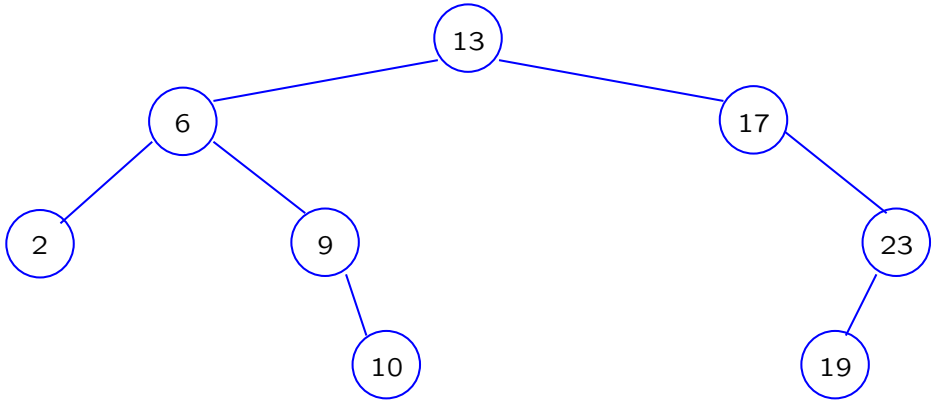
Lopuksi poistamme seuraajasolmun puurakenteesta.

- Koska seuraajasolmu oli alipuun minimi, sillä ei ole vasenta alipuuta, joten lapsia on korkeintaan 1.

Esimerkki: Poistetaan 5, jolla kaksi lasta; tilalle tulee seuraaja 6, jonka alkuperäinen solmu poistuu



Lopputulos



Tasapainoiset puut

Edellä esitettyjen operaatioiden aikavaativuudet ovat kaikki $O(h)$, missä h on puun korkeus. Pahimmassa tapauksessa tämä on $O(n)$, missä n on avainten lukumäärä.

Jos pitäisimme puun melkein täydellisenä, korkeus olisi vain $O(\log n)$. Melkein täydellisen puurakenteen ylläpitäminen edellyttäisi kuitenkin lisäysten ja poistojen yhteydessä niin paljon lisätyötä, että kokonaisuutena ajankulutus luultavasti kasvaisi.

Tasapainottamisen ajatuksena on asettaa puulle jokin melkein täydellisyyttä lievempi tasapainoehto, joka

- on sen verran löysä, että sen pitäminen voimassa ei vaadi kohtuuttomasti lisätyötä, mutta
- takaa kuitenkin, että puun korkeus on $O(\log n)$.

Tärkeitä tasapainotettuja hakupuurakenteita ovat mm.

AVL-puut: historiallisesti ensimmäinen (Adelson-Velsky ja Landis, 1962) ja helppo ymmärtää

punamustat puut: joitain tehokkuusetuja AVL-puihin verrattuna; Javan `TreeSet`- ja `TreeMap`-luokan toteutustapa

B-puut: toimivat erityisen hyvin levymuistin yhteydessä.

Tällä kurssilla tutustumisen kohteeksi on valittu AVL-puut niiden suhteellisen yksinkertaisten ja intuitiivisten algoritmien takia.

AVL-puu on silti monimutkaisin yksittäinen tietorakenne, johon tällä kurssilla tutustutaan.

AVL-puut

AVL-puussa vaaditaan, että puu toteuttaa tasapainoehdon:

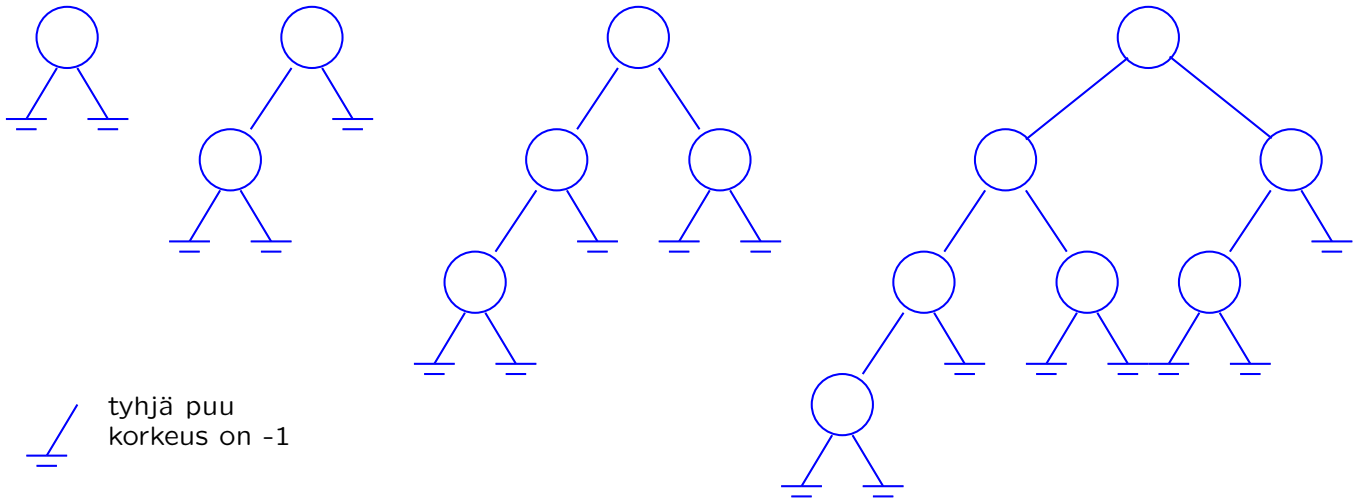
> minkä tahansa solmun vasemman ja oikean alipuun korkeuksien erotus on -1, 0 tai 1.

(Muistetaan, että tyhjän puun korkeudeksi sovittiin -1.)

AVL-tasapainoehdon ylläpitäminen lisäysten ja poistojen yhdeydessä edellyttää, että solmuihin tallennetaan eksplisiittisesti niiden senhetkinen korkeus ja sitä päivitetään tarvittaessa.

Tasapainoehdon takia esim. vasempaan alipuuhun ei voi lisätä mielivaltaisen paljon avaimia ilman, että niitä pitää lisätä myös oikeaan alipuuhun. Vaatii kuitenkin hieman tarkempaa analyysia nähdä, että AVL-puun korkeus tosiaan on $O(\log n)$.

Esimerkki: pienimpiä AVL-puita, joiden korkeudet ovat 0, 1, 2 ja 3



Havainnollisuuden vuoksi kuvaan on merkitty myös tyhjä alipuut, joiden korkeus on -1.

Olkoon $f(h)$ pienin määrä solmuja, joista voidaan rakentaa AVL-puu, jonka korkeus on h. Osoitamme, että $f(h) \geq ca^h$ joillain $a > 1$ ja $c > 0$.

Jos puun korkeus on h, niin ainakin yhden alipuun korkeus on $h-1$. AVL-puussa toisen alipuun korkeus on vähintään $h-2$.

Siis jos halutaan rakentaa mahdollisimman vähillä solmuilla AVL-puu, jonka korkeus on h, niin otetaan

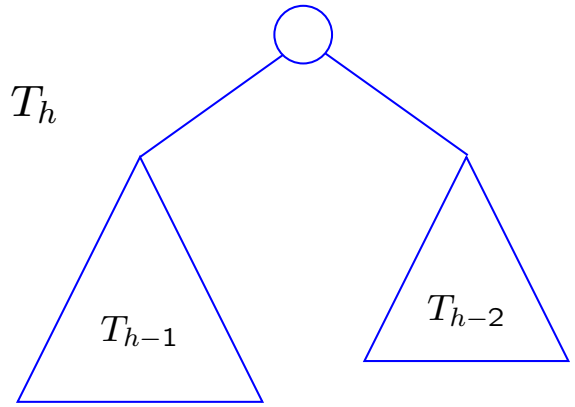
- yhdeksi alipuuksi AVL-puu, jonka korkeus on $h-1$ ja jossa on minimimäärä $f(h-1)$ solmuja
- toiseksi alipuuksi AVL-puu, jonka korkeus on $h-2$ ja jossa on minimimäärä $f(h-2)$ solmuja.

Kun lasketaan mukaan juurisolmu, saadaan palautuskaava

$$f(h) = f(h-1) + f(h-2) + 1.$$

(Kuva seuraavalla sivulla.)

Pienin korkeutta h oleva AVL-puu T_h rakentuu rekursiivisesti:



Kun $|\,T\,|$ tarkoittaa puun solmujen lukumäärää, niin

$$|\,T_h\,| = |\,T_{h-1}\,| + |\,T_{h-2}\,| + 1.$$

Yksinkertaistetaan palautuskaavaa määrittelemällä $g(h) = f(h) + 1$, jolloin se tulee muotoon

$$g(h) = g(h-1) + g(h-2).$$

Tehdään alustava arvaus, että $g(h) \geq ca^h$ joillain a ja c. Katsotaan, mitä tapahtuisi, jos yritettäisiin todistaa tämä induktiolla.

Siis tehdään ”induktio-oletus”, että $g(h-1) \geq ca^{h-1}$ ja $g(h-2) \geq ca^{h-2}$. Sijoitetaan tämä palautuskaavaan:

$$\begin{aligned} g(h) &= g(h-1) + g(h-2) \\ &\geq ca^{h-1} + ca^{h-2} \\ &= c(a+1)a^{h-2}. \end{aligned}$$

Nyt jos $a + 1 \geq a^2$, saadaan haluttu tulos

$$g(h) \geq ca^2 \cdot a^{h-2} = ca^h.$$

Halutaan siis ratkaista epäyhtälö $a^2 - a - 1 \leq 0$.

Vastaavan yhtälön $a^2 - a - 1 = 0$ juuret ovat

$$a = \frac{1}{2} \pm \frac{\sqrt{5}}{2}.$$

Ei ole sattumaa, että positiivinen juuri on itse asiassa kultainen leikkaus

$$\phi = \frac{1 + \sqrt{5}}{2} \approx 1{,}61803.$$

Siis esim. $a = 1{,}618$ toimii palautuskaavassa.

Meidän pitää vielä valita vakio c niin, että ”induktion perustapaus” tulee kuntoon.

Halutaan siis löytää sellainen c, että

$$g(h) \geq c \cdot 1{,}618^h$$

pätee pienillä h. Muistetaan, että $g(h) = f(h) + 1$, missä $f(h)$ on pienin määrä solmuja AVL-puussa, jonka korkeus on h. Koska $f(0) = 1$ ja $f(1) = 2$, saadaan

$$g(0) = 2 = 2 \cdot 1{,}618^0$$
$$g(1) = 3 = \frac{3}{1{,}618} \cdot 1{,}618^1 \approx 1{,}854 \cdot 1{,}618^1.$$

Siis esim. valinnalla $c = 1{,}85$ pätee

$$g(h) \geq c \cdot 1{,}618^h$$

kun $h = 0$ ja $h = 1$. Tekemällä nyt edellä hahmoteltuun tapaan induktio palautuskaavan $g(h) = g(h-1) + g(h-2)$ avulla nähdään, että tämä pätee kaikille h.

Muistetaan, että $g(h) = f(h) + 1$, missä $f(h)$ on pienin määrä solmuja AVL-puussa, jonka korkeus on h.

Siis jos AVL-puussa on n solmua ja sen korkeus on h, niin pätee

$$n \geq 1{,}85 \cdot 1{,}618^h - 1.$$

Ratkaisemalla tästä h ja approksimoimalla saadaan

$$\begin{aligned} \log_2(n+1) &\geq \log_2 1{,}85 + h \cdot \log_2 1{,}618 \\ &\geq 0{,}887 + h \cdot 0{,}694 \end{aligned}$$

eli

$$\begin{aligned} h &\leq \frac{\log_2(n+1) - 0{,}887}{0{,}694} \\ &\approx 1{,}44 \cdot \log_2(n+1) - 1{,}28. \end{aligned}$$

AVL-puut siis häviävät korkeudessa korkeintaan kertoimella 1,44 verrattaessa melkein täydellisillä puilla saavutettavaan parhaaseen mahdolliseen tilanteeseen

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

Alkion lisääminen AVL-puuhun

Alkion lisääminen AVL-puuhun tapahtuu periaatteellisella tasolla seuraavasti:

1. Lisää alkio aivan kuin tasapainottamattomassa hakupuussa.
2. Korjaa muuttuneet *korkeus*-arvot.
3. Tasapainota solmut, joissa lisäys on aiheuttanut tasapainoehdon rikkoutumisen.

Tasapainottaminen tapahtuu suorittamalla kiertoja, joihin palaamme kohta.

Koska solmun lisääminen voi vaikuttaa vain niiden solmujen korkeuteen, joiden jälkeläiseksi solmu lisättiin, niin

- *korkeus*-arvot voivat muuttua vain polulla lisätystä solmusta juureen
- samoin tasapainoehto voi rikkoutua vain tällä polulla.

Hakupuun kierrot

Kierto (rotation) on hakupuuhun kohdistuva operaatio, jossa muutetaan muutaman toisiaan lähellä sijaitsevan solmun sijaintia suhteessa toisiinsa. Solmujen korkeudet muuttuvat hieman, mutta hakupuuominaisuus säilyy.

Kierto vasemmalle ja kierto oikealle ovat toistensa käänteisoperaatioita (kuva seuraavalla sivulla):

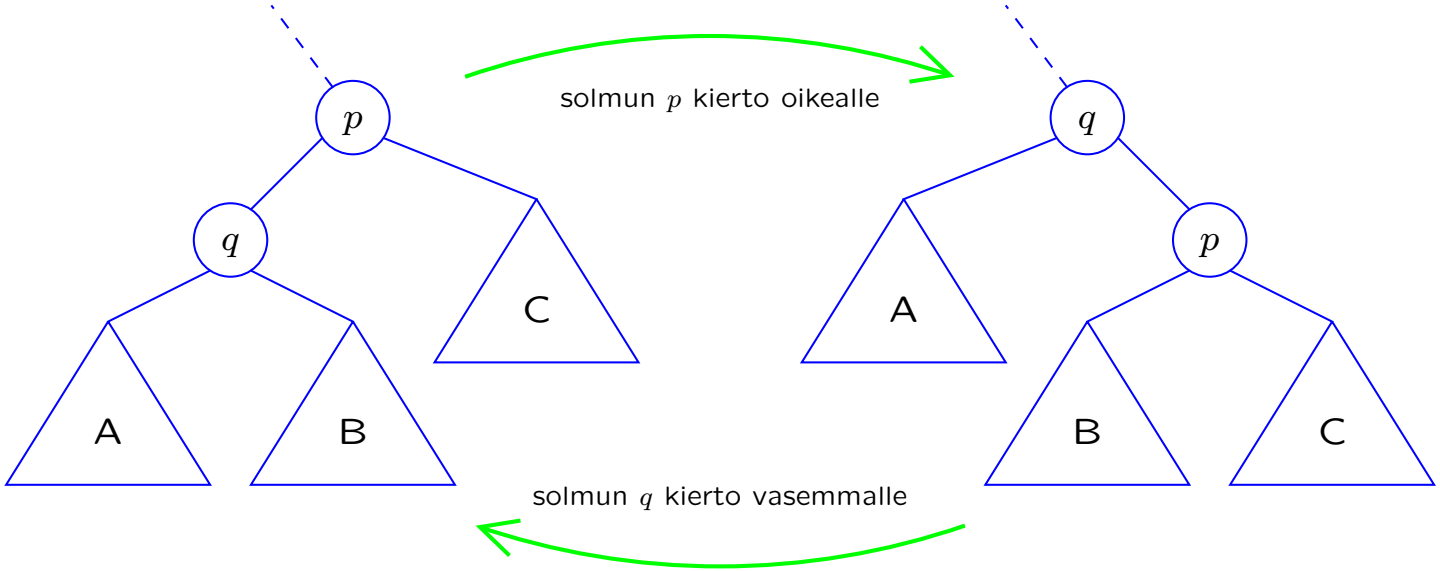
- kierto oikealle solmussa p nostaa solmun p vasenta alipuuta ylöspäin
- kierto vasemmalle solmussa p nostaa solmun p oikeaa alipuuta ylöspäin.

Osa AVL-puun lisäysoperaatiossa syntyvistä epätasapainoista voidaan korjata yhdellä kierrolla.

Jotkin vaativat hieman enemmän työtä.

Aloitamme helpoista tapauksista.

Kierto vasemmalle ja oikealle:



Oleellista on solmujen p ja q sijainnit; alipuiden A, B ja C paikat määräytyvät hakupuuominaisuuden perusteella.

Lisäysoperatio hieman täsmennettynä tapahtuu seuraavasti:

1. Lisää solmu kuten tasapainottamattomassa puussa.
2. Kulje lisätystä solmusta polkua takaisin kohti juurta päivittäen samalla *korkeus*-arvoja.
3. Kun kohtaat solmun, jossa tasapainoehto meni rikki, niin pysähdy tasapainottamaan se.

Oleellinen kysymys tietysti on, miten tasapainottaminen tapahtuu.

Sen ymmärtämiseksi pitää tarkastella erilaisia tilanteita, jotka voivat johtaa tasapainon rikkoutumiseen.

Tarkastellaan tilannetta, kun

- solmu x on lisätty,
- solmusta x juurta kohti kulkiessa solmu p on ensimmäinen, jossa tasapainoehto on rikkoutunut.
- oletetaan, että solmu x on lisätty solmun p vasempaan alipuuhun; lisäys oikeaan käsitellään symmetrisesti.

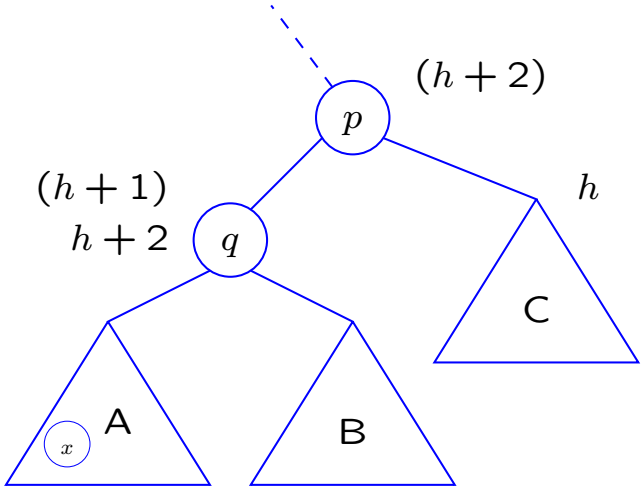
Koska

- ennen lisäystä solmu p oli tasapainossa
- lisäys vasempaan alipuuhun rikkoi tasapainoehdon

niin jollain h pätee (kuva seuraavalla sivulla), että

- oikean alipuun korkeus on h ennen ja jälkeen lisäyksen
- vasemman alipuun korkeus oli $h + 1$ ennen lisäystä mutta $h + 2$ lisäyksen jälkeen.

Lisäys vasempaan alipuuhun on rikkonut solmun p tasapainon



- joidenkin solmujen korkeudet on merkitty; korkeudet suluissa viittaavat tilanteeseen ennen lisäystä
- vasen alipuu on jatkoa ajatellen esitetty osiin jaettuna.

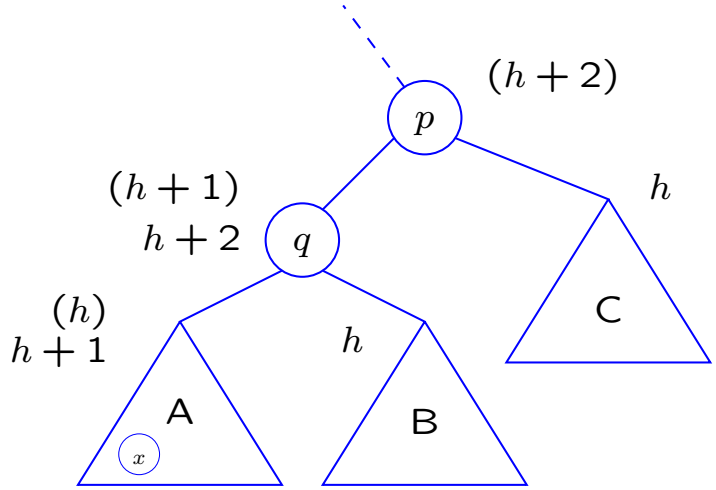
Oletetaan nyt edelleen, että lisäys on tullut solmun p vasemman alipuun vasempaan alipuuhun, kuvassa A.

(Tapaus, jossa lisäys on tullut vasemman alipuun oikeaan alipuuhun B esitetään hetken kuluttua.)

Nyt voidaan päätellä seuraavaa:

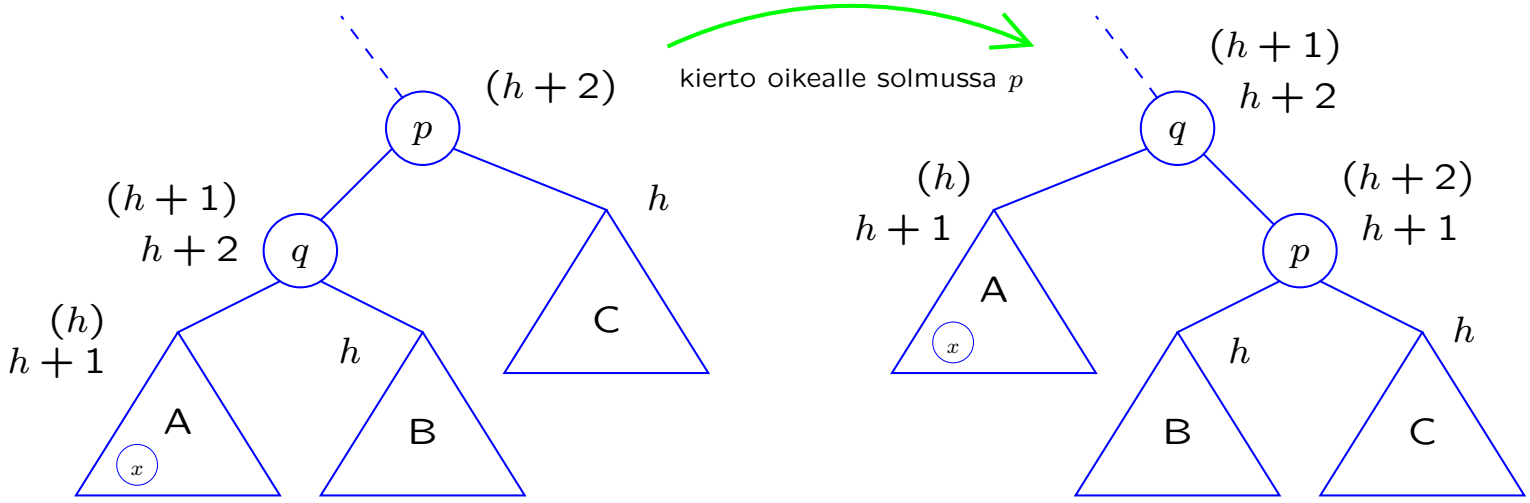
- Lisäys kasvatti solmun q korkeuden arvosta $h + 1$ arvoon $h + 2$.
- Solmu p on ensimmäinen kohdattu epätasapainoinen solmu, joten q oli tasapainossa vielä lisäksen jälkeen.
- Siis solmun q alipuiden korkeudet olivat kumpikin h ennen lisäystä, ja h ja $h + 1$ lisäyksen jälkeen.
- Koska lisäys kohdistui puuhun A, sen korkeus kasvoi ja B:n korkeus pysyi ennallaan.

Lisäys on tullut solmun p vasemman alipuun vasempaan alipuuhun:



- epätasapaino aiheutuu puun A ulottumisesta liian syvälle suhteessa puuhun C
- korjataan tilanne nostamalla puuta A kiertämällä p oikealle.

Lisäys on tullut solmun p vasemman alipuun vasempaan alipuuhun:



- solmun p tasapaino on korjattu kiertämällä sitä oikealle
- havaitaan, että muutkin kuvan solmut ovat tasapainossa
- lisäksi solmun q uusi korkeus on sama kuin solmun p vanha korkeus, joten tästä ylöspäin korkeuksiin ei tule muutoksia eikä siis uusia epätasapainoja voi syntyä.

Siis jos solmun p epätasapaino johtui siitä, että lisäys oli tullut sen vasemman alipuun vasempaan alipuuhun, tilanne korjautui suorittamalla yksi kierto oikeaan.

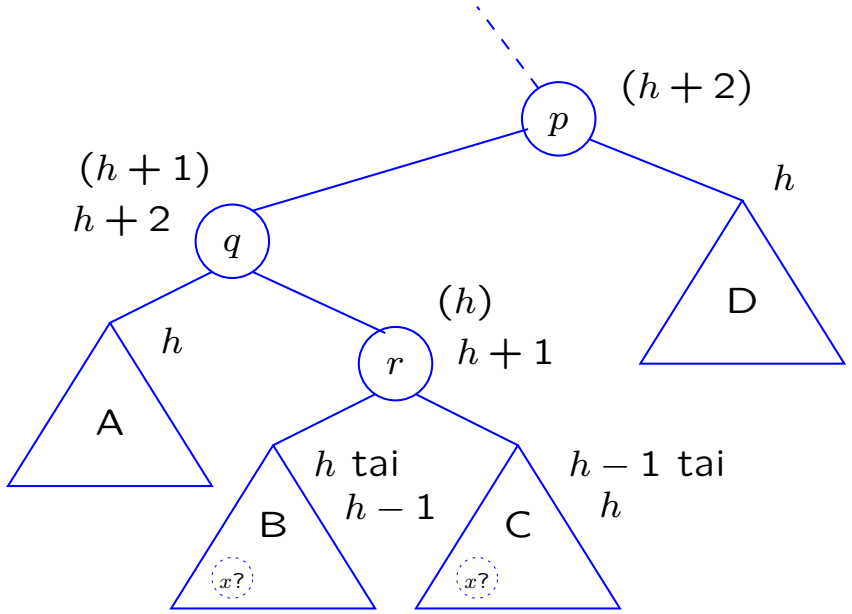
Tarkastellaan nyt tilannetta, kun lisäys on tullut vasemman alipuun oikeaan alipuuhun (kuva seuraavalla sivulla).

Jatkoa varten oikea alipuu on jaettu osiin. Solmun r ja alipuun A korkeudet on päätelty samalla logiikalla kuin edellisessä tapauksessa.

Edelleen samalla logiikalla

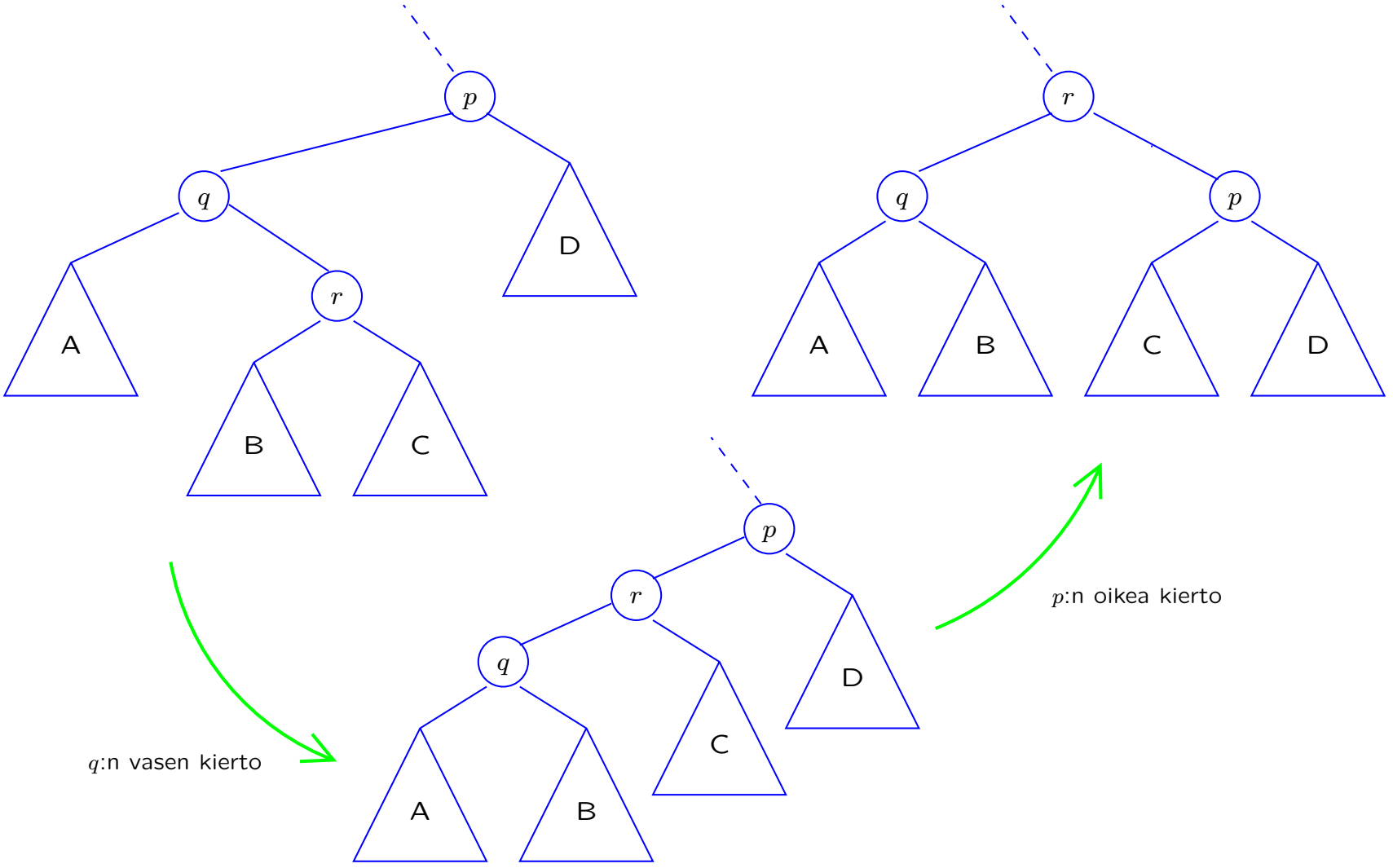
- joko lisäys on tullut alipuuhun B, jolloin lisäyksen jälkeen B:n korkeus on h ja C:n korkeus $h-1$
- tai lisäys on tullut alipuuhun C, jolloin lisäyksen jälkeen C:n korkeus on h ja B:n korkeus $h-1$

Lisäys on tullut solmun p vasemman alipuun oikeaan alipuuhun:

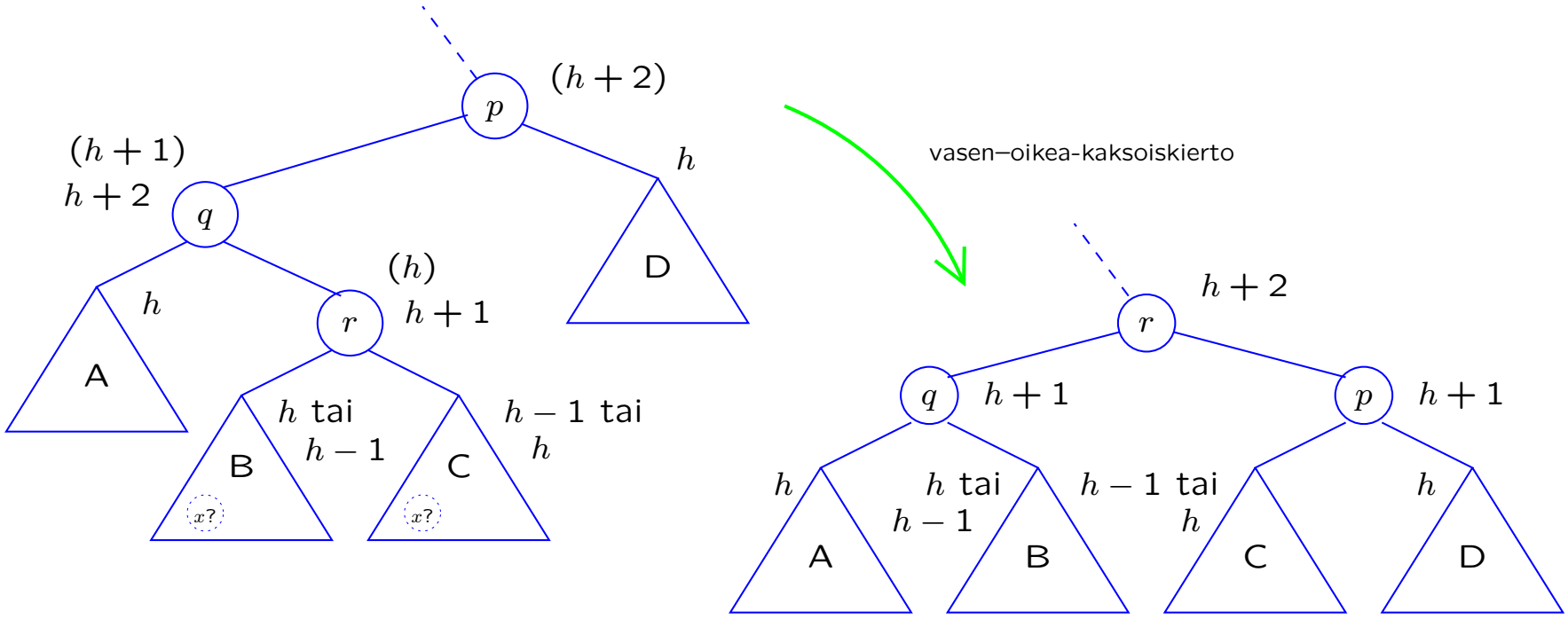


- jompi kumpi puista B ja C on liian syvällä
- mikään yksittäinen kierto ei riitä varmasti korjaamaan tilannetta
- ratkaisu on vasen–oikea-kaksoiskierto: kierretään ensin q vasemmalle ja sitten p oikealle.

Periaatekuva vasen–oikea-kaksoiskierrosta:



Kaksoiskierto sovellettuna edelliseen tilanteeseen:



Taas huomataan, että solmun r uusi korkeus on sama kuin solmun p vanha korkeus, joten tästä ylöspäin ei enää tule muutoksia.

Yhteenveto AVL-puun lisäysoperaatiosta:

1. Tee lisäys kuten tasapainottamattomassa puussa.
2. Palaa lisätystä solmusta kohti juurta päivittäen *korkeus*-arvoja ja tarkastaen tasapainoehtoja.
3. Kun löytyy ensimmäinen solmu p, joka ei enää ole tasapainossa, niin
   - jos lisäys oli p:n vasemman alipuun vasempaan alipuuhun, tee kierto oikeaan
   - jos lisäys oli p:n vasemman alipuun oikeaan alipuuhun, tee vasen–oikea-kaksoiskierto
   - jos lisäys oli p:n oikean alipuun vasempaan alipuuhun, tee oikea–vasen-kaksoiskierto
   - jos lisäys oli p:n oikean alipuun oikeaan alipuuhun, tee kierto vasempaan.
4. Nyt muutoksia ei enää tule, joten polun läpikäyminen voidaan lopettaa.

Alkion poistaminen AVL-puusta

Alkion poistaminen soveltaa samaa ajatusta kuin lisäys, mutta on hieman monimutkaisempi:

1. Poista alkio kuten tasapainottamattoman puun tapauksessa. Olkoon x solmu, joka poistui puurakenteesta. (Huomaa, että poistettaessa alkio, jolla on kaksi lasta, x ei ole tämän alkion alkuperäinen solmu, vaan sen seuraajasolmu; ks. sivut 259–260.)

2. Kulje polku solmusta x puun juureen päivittäen *korkeus*-arvoja ja tarkastaen tasapainoehtoja.

   Kun löytyy solmu, joka ei ole tasapainossa, tasapainota se sopivilla kierroilla.

Toisin kuin lisäyksessä, tasapainotusta voi joutua tekemään useassa solmussa.

Tämä selvenee, kun katsomme, millaisilla kierroilla tasapainotus tapahtuu.

Oikeiden kiertojen löytämiseksi tarkastellaan taas, mikä aiheuttaa epätasapainon.

Poistonkin yhteydessä on helpompi jäsentää tilanne sen mukaan, mikä alipuu on liian korkea. Tämä siis nimenomaan ei ole se alipuu, johon poisto on kohdistunut.

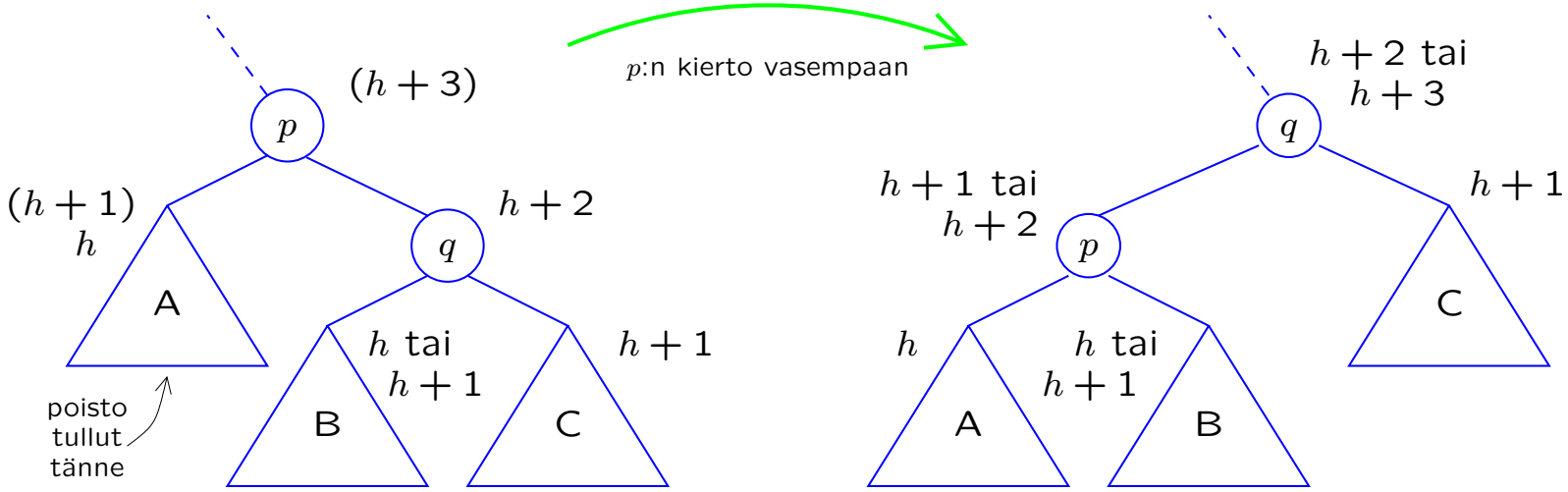
Tarkastellaan tapausta, jossa solmu p on epätasapainossa ja poisto on kohdistunut sen vasempaan alipuuhun.

Nyt on kaksi mahdollisuutta: liian korkea alipuu on

- oikean alipuun oikea alipuu: tämä ratkeaa kierrolla vasempaan
- oikean alipuun vasen alipuu: tämä ratkeaa oikea–vasen-kaksoiskierrolla.

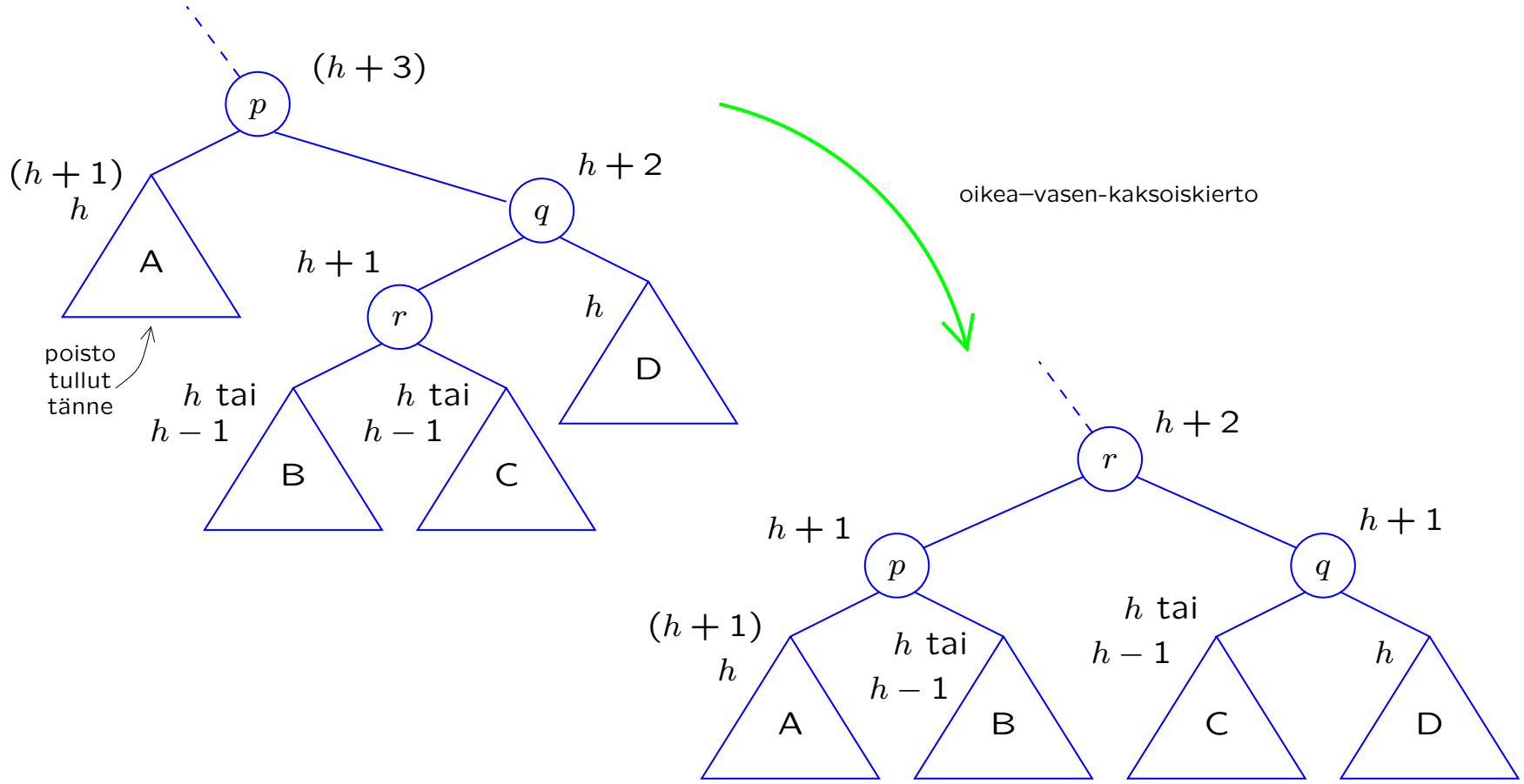
Siis tapaukset ja niiden ratkaisut ovat samankaltaiset kuin lisäyksessä. Perustelemme ratkaisut vain lyhyesti, koska päättely on samanlaista kuin aiemminkin.

Poiston jälkeen oikean alipuun oikea alipuu liian korkea:



Toisin kuin lisäyksessä, solmun q uusi korkeus voi olla pienempi kuin solmun p vanha korkeus. Tarkistuksia pitää siis vielä jatkaa puussa ylöspäin.

Poiston jälkeen oikean alipuun vasen alipuu liian korkea:



Solmun r uusi korkeus on pienempi kuin solmun p vanha korkeus, joten tarkistuksia pitää jatkaa puussa ylöspäin.

Tasapainotettujen hakupuiden aikavaativuus

Olemme edellä todenneet, että ilman tasapainotusta kaikki operaatiot toimivat ajassa $O(h)$, missä h on hakupuun korkeus.

Tasapainottaminen vie hieman lisäaikaa lisäyksen ja poiston yhteydessä. Pahimmassakin tapauksessa lisälaskentaa tehdään vakiomäärä juureen johtavan polun kussakin solmussa. Siis aikavaativuuden luokka $O(h)$ pätee edelleen.

AVL-ehdon takia $h = O(\log n)$, missä n on puun solmujen lukumäärä.

Siis AVL-puilla kaikki joukko-operaatiot toimivat pahimmassakin tapauksessa ajassa $O(\log n)$.

Tasapainotetut hakupuut Javassa

Javassa on valmiina tasapainotettuihin hakupuihin perustuvat luokat

TreeSet: joukko avaimia, joihin ei liity muuta dataa
TreeMap: sisältää (avain, data) -pareja.

Toteutus perustuu punamustiin puihin.

Jos avaimena käyttään itsemääriteltyä luokkaa, sille pitää määritellä `compareTo` ja `equals` niin, että ne vertaavat asianmukaisella tavalla avainten kenttiin talletettuja arvoja.

Yhteenveto hakupuista

Tasapainotetulla hakupuulla operaatiot search, insert, delete, min, max, succ ja pred toimivat kaikki pahimmassa tapauksessa ajassa $O(\log n)$.

Hajautustaulun operaatiot toimivat keskimäärin ajassa $O(1)$, jos hajautusfunktio toimii odotetulla tavalla. Hajautustaulussa myös vakiokertoimet ovat kohtuullisen pieniä, joten sen voi yleensä olettaa toimivan hieman tehokkaammin kuin tasapainotettu hakupuu.

Tasapainotettu hakupuu on parempi kuin hajautustaulu, jos

- tarvitaan avainten järjestykseen perustuvia operaatioita tai
- pahimman tapauksen aikavaativuus on tärkeä.

Hakupuita (ja hajautustauluja) käytetään nimenomaan silloin, kun halutaan dynaaminen joukko eli lisäyksiä tai poistoja tulee pitkin algoritmin suoritusta.

Jos kaikki alkiot lisätään kerralla eikä niitä sen jälkeen muuteta, on tehokkaampaa tallentaa ne taulukkoon ja järjestää se. Kokonaisaikavaativuus on sama $O(n \log n)$, mutta järjestämisen vakiokertoimet ovat paljon paremmat.

7. Keko

Prioriteetijono (priority queue) on abstrakti tietotyyppi, jota voidaan ajatella jonon ja pinon yleistyksenä. Lisättäessä alkio prioriteettijonoon sille annetaan kiireellisyysarvo eli prioriteetti, ja poistot tapahtuvat prioriteetin mukaisessa järjestyksessä.

Keko (heap) on tietorakenne, jolla prioriteettijono voidaan toteuttaa hyvin tehokkaasti.

Tämän luvun jälkeen opiskelija

- osaa soveltaa prioriteettijonoa ongelmanratkaisussa
- tuntee yksityiskohtaisesti prioriteettijonon toteutuksen kekona ja tästä johtuvat operaatioiden aikavaativuudet
- tuntee kekojärjestämisen toiminnan yksityiskohdat ja aikavaativuuden.

Prioriteettijono (priority queue)

Prioriteettijono on muunnelma abstraktista tietotyypistä joukko.

Prioriteettijonon alkioilla oletetaan määritellyksi kenttä prioriteetti. Useimmiten prioriteetiksi laitetaan kokonaislukuja, mutta riittää, että niille on määritelty suuruusjärjestys. Prioriteettijonossa saa olla useita alkioita samalla prioriteetilla.

Prioriteettijonon operaatiot ovat

isEmpty: palauttaa *true* jos joukossa ei ole yhtään alkiota

insert(x): lisää alkion x prioriteettijonoon

min: palauttaa arvonaan prioriteettijärjestyksessä ensimmäisen alkion; ei muuta prioriteettijonon sisältöä

deleteMin: palauttaa arvonaan prioriteettijärjestyksessä ensimmäisen alkion ja poistaa sen prioriteetijonosta. Jos usealla alkiolla on sama ensimmäinen prioriteetti, valitaan niistä jokin mielivaltainen.

Tässä on kuvattu minimiprioriteettijono, jossa esim. kokonaislukuarvoisista prioriteeteista ensimmäisenä valitaan lukuarvoltaan pienin. Yhtä hyvin voitaisiin tietysti toteuttaa käänteinen järjestys, jolloin operaatioista käytetään merkintöjä deleteMax jne.

Esimerkki Järjestelmään tulee palvelupyyntöjä, jotka halutaan toteuttaa tärkeysjärjestyksessä.

Muodostetaan jokaisesta palvelupyynnöstä tietue, jossa on

- palvelupyynnölle annettu tärkeys väliltä 1–10, missä 1 on tärkein mahdollinen
- palvelupyynnön saapumisaika
- palvelupyynnön tunnuskoodi, jonka avulla esim. erillisestä hajautustaulusta saadaan tarkemmat tiedot pyydetystä palvelusta.

Laitetaan tietueet prioriteettijonoon niin, että prioriteettina toimii pari (tärkeys, saapumisaika):

- tärkeämmät tehtävät tehdään ennen vähemmän tärkeitä
- yhtä tärkeät tehtävät tehdään saapumisjärjestyksessä.

Valitaan aina prioriteettijonosta seuraava tehtävä ja haetaan sen tunnuskoodin perusteella tarkemmat tiedot, miten se pitäisi suorittaa.

Esimerkki Halutaan simuloida järjestelmää, jossa on asiakkaita, palvelupisteitä jne.

Pidetään yllä prioriteettijonoa, jossa on tapahtumia ja prioriteettina toimii tapahtuman (ennakoitu) tapahtumisaika. Tapahtumia poimitaan jonosta yksi kerrallaan.

Ensin luodaan vaikkapa joukko tapahtumia ”asiakas saapuu”, joiden ajat valitaan sopivasta satunnaisjakaumasta.

Kun ”asiakas saapuun” -tapahtuma tulee suoritusvuoroon simulaation ajanhetkellä t, luodaan tapahtuma ”asiakas pääsee palveltavaksi” ajanhetkelle $t + s$, missä s on asiakkaan haluaman palvelupisteen jononpituus hetkellä t.

Kun ”asiakas pääsee palveltavaksi” ajanhetkellä t, luodaan tapahtuma ”asiakas poistuu” ajanhetkelle $t + p$, missä p on asiakkaan vaatiman palvelun viemä aika.

Prioriteettijonon operaatiot voisi suoraan toteuttaa ajassa $O(\log n)$ käyttäen tasapainotettua hakupuuta. Esitämme kuitenkin seuraavassa prioriteettijonon toteutuksen kekoa käyttäen. Kekototeutuksella aikavaativuudet ovat

isEmpty:	$O(1)$
insert:	$O(\log n)$
min:	$O(1)$
deleteMin:	$O(\log n)$.

Lisäksi keon muodostaminen annetuista n alkiosta, kun keko on aluksi tyhjä, onnistuu ajassa $O(n)$.

Keko on tasapainotettua hakupuuta helpompi toteuttaa ja sen vakiokertoimet ovat pienempiä.

Keko tarvitsee tallennettavien alkioiden lisäksi vain vakiomäärän muistia (jos alkioiden lukumäärä on ennalta tiedossa). Näemme myöhemmin, että tämä mahdollistaa mm. kekojärjestämisen:

- aikavaativuus $O(n \log n)$
- tilavaativuus $O(1)$.

Binäärikeko (binary heap)

Binäärikeot ovat tietynlaisia binääripuita. Keon käsite yleistyy myös muille kuin binääripuille, mutta emme käsittele niitä tällä kurssilla. Jatkossa siis keko tarkoittaa aina binäärikekoa.

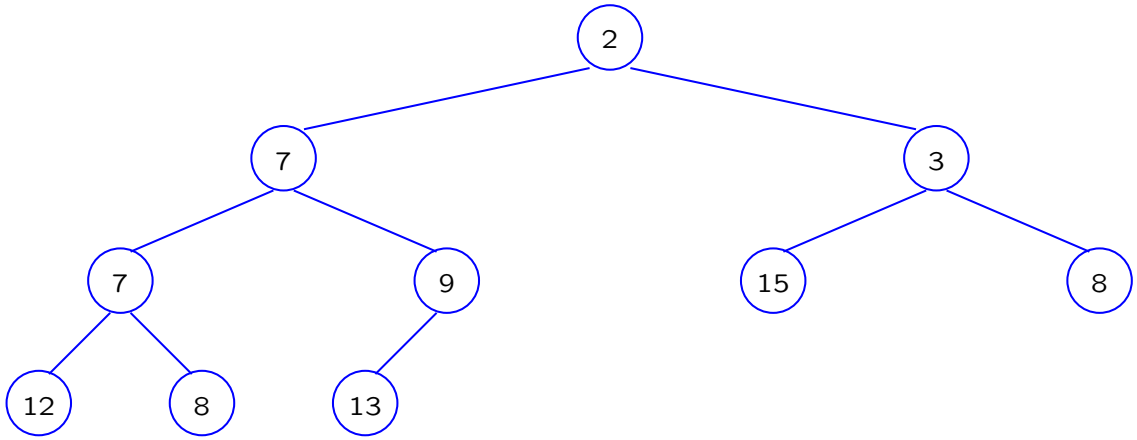
Keko voi olla minimi- tai maksimikeko. Minimikeossa jokaisen solmun tulee toteuttaa kekoehto

> solmun avain on pienempi tai yhtäsuuri kuin sen lasten avaimet.

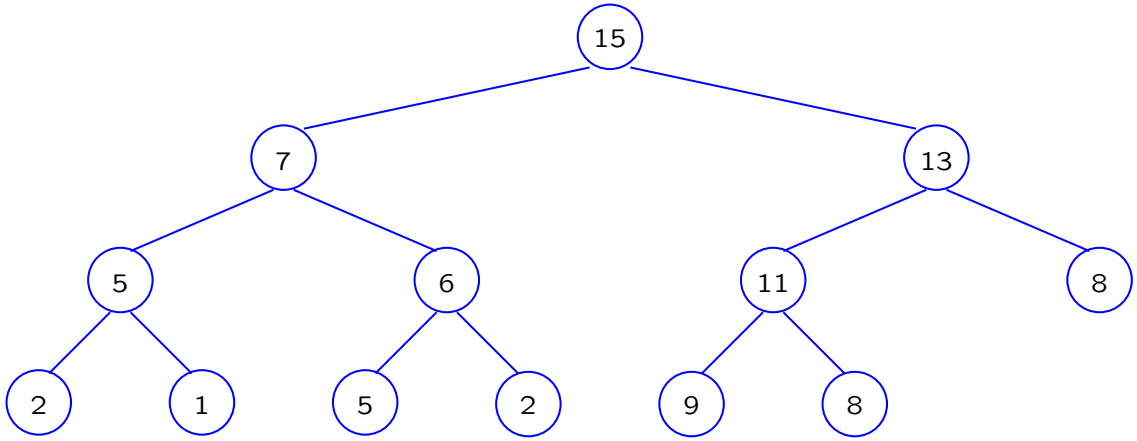
Siis erityisesti koko keon pienin avain on juuressa. Maksimikeossa vastaavasti solmun avain on suurempi tai yhtäsuuri kuin lasten avaimet.

Lisäksi keolta vaaditaan, että sen pitää olla melkein täydellinen:

- mahdollisesti viimeistä tasoa lukuunottamatta kaikki puun tasot ovat täynnä, ts. tasolla i on 2^i solmua
- viimeisellä tasolla solmut ovat niin vasemmalla kuin mahdollista.



Esimerkki minimikeosta



Esimerkki maksimikeosta

Jos keon korkeus on h, eli siinä on tasot $0, 1, 2, \ldots, h$, niin siinä on solmuja ainakin

$$n \geq \sum_{i=0}^{h-1} 2^i + 1 = (2^{(h-1)+1} - 1) + 1 = 2^h$$

kappaletta. Siis kääntäen jos solmuja on n, niin korkeus h on korkeintaan

$$h \leq \log_2 n.$$

Yksi syy keon ''tasapainoehdolle'' on kuten hakupuissa, että korkeus halutaan pitää logaritmisena.

Kun keolta vaaditaan esim. AVL-ehtoa paljon vahvemmin, että sen pitää olla melkein täydellinen, niin lisäksi mahdollistetaan keon tehokas tallentaminen taulukkona.

Keon tallentaminen taulukkoon

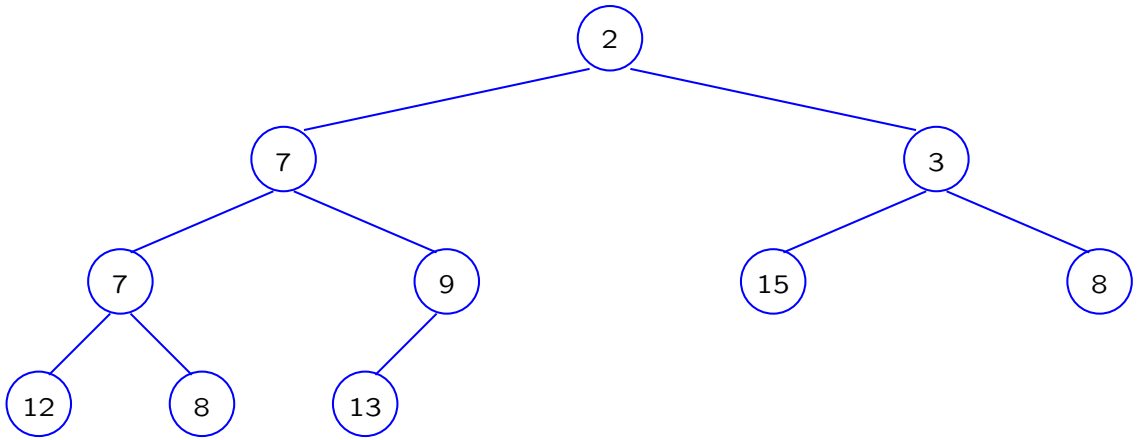
Ideana on yksinkertaisesti tallentaa keossa olevat solmut taulukkoon

- juuresta alkaen
- taso kerrallaan
- kukin taso vasemmalta oikealle.

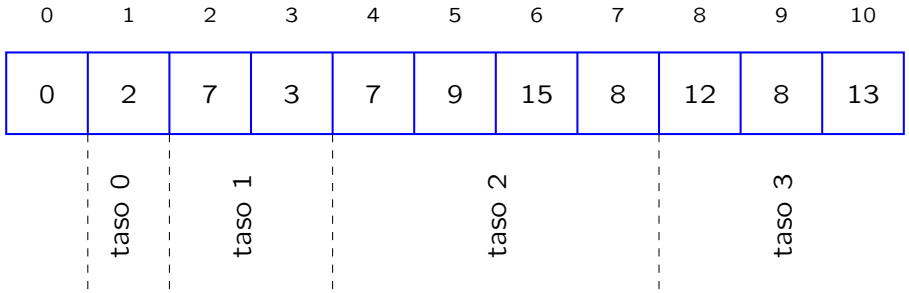
Laskujen yksinkertaistamiseksi jätetään taulukon paikka 0 tyhjäksi. Tällöin

- taso 0 (eli juuri) tulee taulukon kohtaan 1
- taso 1 (eli juuren lapset) tulee taulukon kohtaan $[2 \ldots 3]$
- taso 2 tulee taulukon kohtaan $[4 \ldots 7]$
- taso 3 tulee taulukon kohtaan $[8 \ldots 15]$
- yleisesti taso i tulee kohtaan $[2^i \ldots 2^{i+1} - 1]$

Esimerkki Minimikeko



tallennetaan taulukkoon muodossa



Huomataan, että

- tason i vasemmanpuoleisin solmu on tallennettu kohtaan 2^i ja sen lapset ovat tason $i+1$ vasemmanpuoleisimmat solmut $2^{i+1} = 2 \cdot (2^i)$ ja $2^{i+1} + 1 = 2 \cdot 2^i$
- tason i oikeanpuoleisin solmu on tallennettu kohtaan $2^{i+1} - 1$ja sen lapset ovat tason $i+1$ oikeanpuoleisimmat solmut $2^{i+2} - 2 = 2 \cdot (2^{i+1} - 1)$ ja $2^{i+2} - 1 = 2 \cdot (2^{i+1} - 1) + 1$
- yleisesti kohtaan p talletetun solmun lapset ovat kohdissa $2p$ ja $2p+1$
- vastaavasti kohtaan p tallennetun solmun vanhempi on kohdassa $\lfloor p/2 \rfloor$.

Emme siis tarvitse puurakenteen tallentamiseen linkkejä, vaan voimme korvata ne aritmetiikalla:

- vasen$(p) = 2p$
- oikea$(p) = 2p + 1$
- vanhempi$(p) = \lfloor p/2 \rfloor$.

Keon koko ei yleensä ole etukäteen tiedossa ja muuttuu usein. Ei siis ole käytännöllistä pitää talletusalueena toimivaa taulukkoa jatkuvasti tasan samankokoisena kuin varsinainen keko:

- varataan aluksi jokin kohtuullisen kokoinen taulukko
- pidetään muuttujassa viimeinen kirjaa siitä, mikä on taulukon viimeinen keon talletukseen käytetty kohta
- jos alkioiden lisäyksen myötä talletusalue tulee täyteen, niin kasvatetaan taulukkoa samaan tapaan kuin taulukkolistan ja hajautustaulun tapauksessa.

Puussa siirtymistä esittävät funktiot saadaan nyt tarkemmin seuraavaan muotoon, missä palautusarvo 0 tarkoittaa, että kyseistä solmua ei ole puussa:

```
vasen(p)
    if 2p > viimeinen return 0
    else return 2p

oikea(p)
    if 2p + 1 > viimeinen return 0
    else return 2p + 1

vanhempi(p)
    return ⌊p/2⌋
```

Prioriteettijonon operaatiot taulukkomuotoisessa keossa

Tarkastelemme minimikekoa; maksimikeon operaatioissa vaihdetaan luonnollisesti alkioiden erisuuruusvertailujen suunnat.

Prioriteettijonon operaatioista min on minimikeossa helppo toteuttaa vakioajassa, koska pienin arvo on valmiiksi juuressa:

```
min()
  return taulukko[1]
```

Poistossa ja lisäyksessä keon rakenteen säilyttäminen vaatii hieman työtä.

Operaatioiden toiminta-ajatus on helpompi ymmärtää keon puuesitystä tarkastelemalla, vaikka toteutus viime kädessä käyttääkin taulukkoa.

Oletamme tässä, että varattu taulukko on riittävän suuri kaikille lisäyksille.

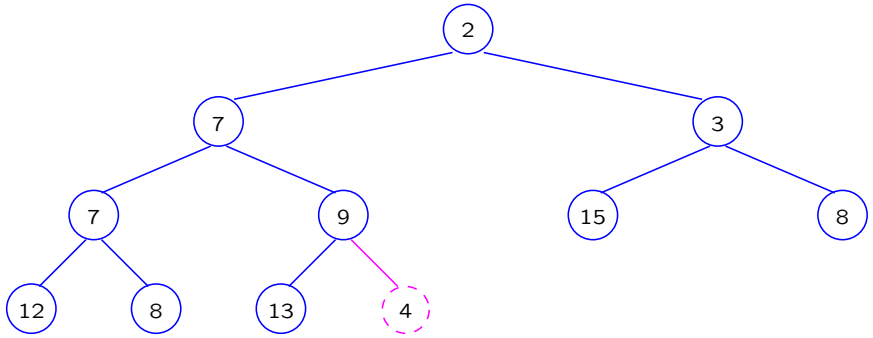
Lisäys kekoon

Lisäysoperaation ideana on

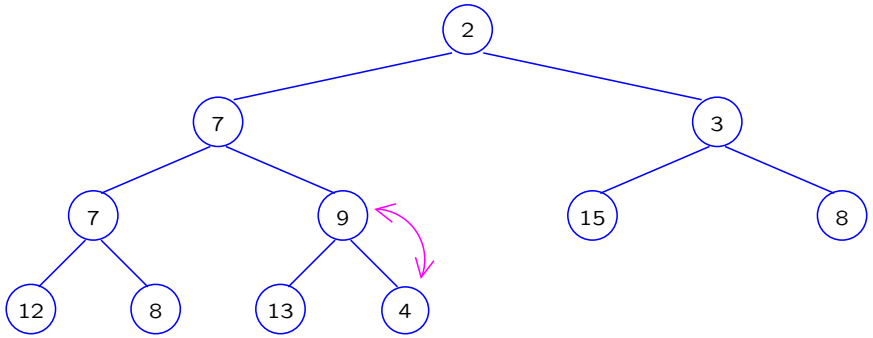
- lisätä uusi alkio keon seuraavaan vapaaseen paikkaan
- kuljettaa sitä puussa ylöspäin, kunnes kekoehto pätee.

Seuraavassa pseudokoodissa oletetaan, että alkioita vertailtaessa "taulukko[p] $<$ taulukko[q]" palauttaa tosi, jos alkio taulukko[p] on prioriteettijärjestyksessä ennen alkiota taulukko[q].

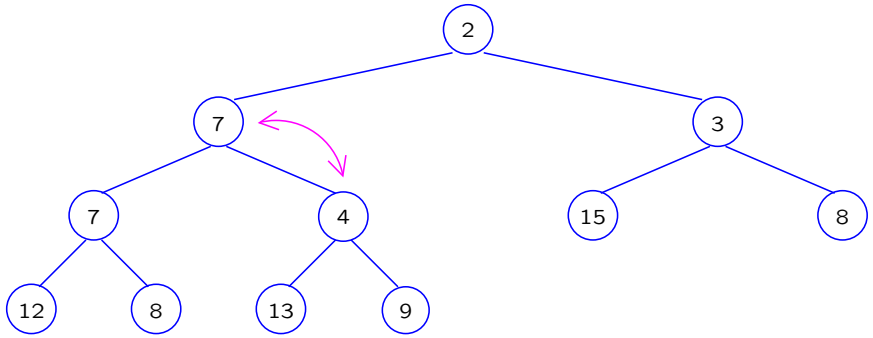
```
insert(x)
    viimeinen = viimeinen + 1
    taulukko[viimeinen] = x
    p = viimeinen
    while (p > 1) and (taulukko[p] < taulukko[vanhempi(p)])
        vaihda taulukko[p] ja taulukko[vanhempi(p)] keskenään
        p = vanhempi(p)
```



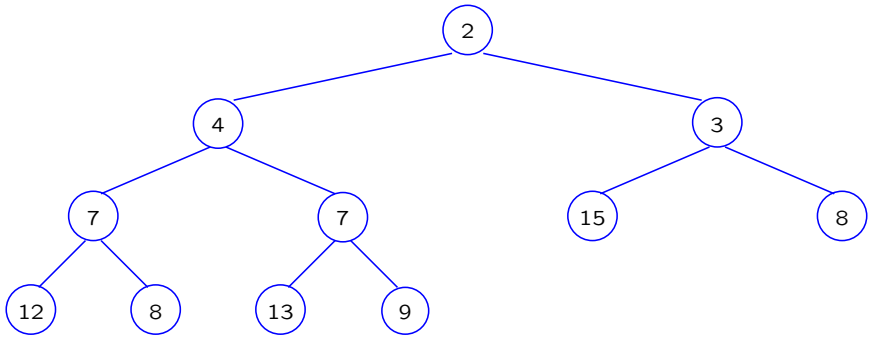
vaihe 1: lisää solmu 4



vaihe 2: vaihda 4 ja 9



vaihe 3: vaihda 4 ja 7



vaihe 4: valmis

Minimikekoon lisätään avain 4.

Huomaa, että lisättyä arvoa siirretään ylöspäin vain, kun se on pienempi kuin vanhemman arvo.
(Edellisen esimerkin vaiheessa 2 pätee $4 < 9$.)

Koska alun perin kekoehto oli voimassa, vanhemman arvo puolestaan on pienempi tai yhtäsuuri kuin mahdollisen sisaren arvo.
(Vaiheessa 2 pätee $9 \leq 13$.)

Siis lisätty arvo on pienempi kuin sisaren arvo.
(Vaiheessa 2 pätee $4 < 13$.)

Lisätyn arvon siirtäminen ylöspäin ei riko kekoehtoa kummankaan lapsen suuntaan.
(Vaiheen 2 jälkeen $4 < 13$ ja $4 < 9$.)

Huom. Edellä esitetty pseudokoodi tekee paljon turhaa työtä vaihdoissa.

Lisättävää alkiota ei ole syytä kirjoittaa taulukkoon, ennen kuin sen lopullinen sijainti on tiedossa:

```
insert(x)
    viimeinen = viimeinen + 1
    p = viimeinen
    while (p > 1) and (x < taulukko[vanhempi(p)])
        taulukko[p] = taulukko[vanhempi(p)]
        p = vanhempi(p)
    taulukko[p] = x
```

Esitämme kuitenkin havainnollisuuden vuoksi algoritmit muodossa, jossa on vaihtoja.

Poisto keosta

Keosta poistettava alkio siis on aina puun juuri.

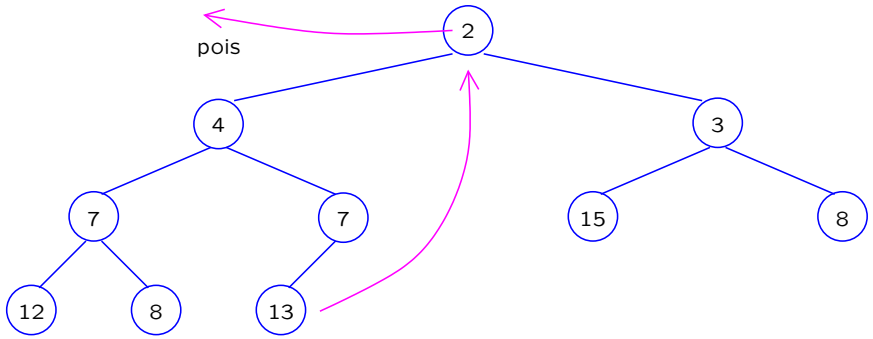
Perusajatuksena on

- siirtää taulukon viimeinen alkio uudeksi juureksi
- siirtää tätä alkiota puussa alaspäin, kunnes kekoehto tulee taas voimaan.

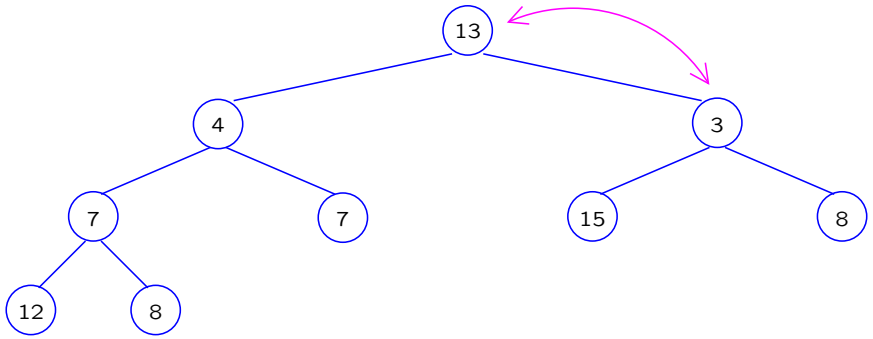
Alkiota siirretään alaspäin, jos se on suurempi kuin ainakin toinen lapsi.

Siirrettäessä se vaihdetaan pienemmän lapsen kanssa.

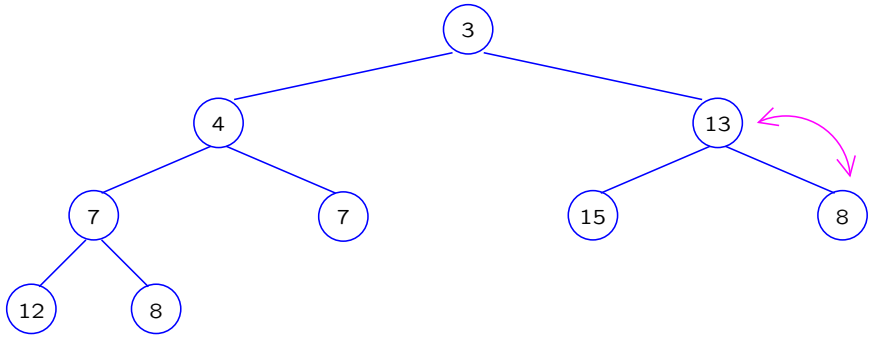
Jos se vaihdettaisiin suuremman lapsen kanssa, niin suuremmasta lapsesta tulisi pienemmän vanhempi vastoin kekoehtoa.



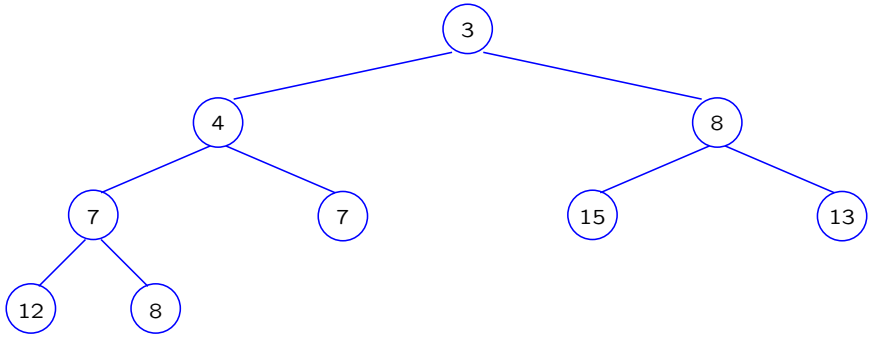
vaihe 1: siirretään viimeinen avain juureksi



vaihe 2: vaihdetaan 13 ja 3



vaihe 3: vaihdetaan 13 ja 8



vaihe 4: valmis

Minimikeosta poistetaan pienin alkio.

Poisto-operaation pseudokoodi, jossa keko-ominaisuuden palauttava painaAlas on erikseen seuraavalla sivulla:

```
deleteMin()
    arvo = taulu[1]
    taulu[1] = taulu[viimeinen]
    viimeinen = viimeinen - 1
    painaAlas(1)
    return arvo
```

Proseduuri painaAlas(p) (seuraava sivu)

- olettaa, että solmusta p alkavan alipuun muut solmut paitsi p toteuttavat kekoehdon
- lopputuloksena kekoehto pätee koko alipuussa, myös solmussa p.

Esitetään keko-ominaisuuden palauttava proseduuri yksinkertaisuuden vuoksi rekursiivisena:

```
painaAlas(solmu)
    if vasen(solmu) > viimeinen          // solmulla ei lapsia
        return
    else if vasen(solmu) == viimeinen    // vain vasen lapsi
        pienempiLapsi = vasen(solmu)
    else                                 // kaksi lasta
        if taulu[vasen(solmu)] < taulu[oikea(solmu)]
            pienempiLapsi = vasen(solmu)
        else
            pienempiLapsi = oikea(solmu)
        if taulu[solmu] > taulu[pienempiLapsi]
            vaihda taulu[solmu] ja taulu[pienempiLapsi]
            painaAlas(pienempiLapsi)
```

Lisäyksen ja poiston aikavaativuus

Pahimmassa tapauksessa sekä lisäys että poisto

- käyvät läpi polun puun kaikkien tasojen läpi
- tekevät polun jokaisessa solmussa vakiomäärän laskentaa.

Koska tasoja on $O(\log n)$, myös aikavaativuus on $O(\log n)$, missä n on alkioiden määrä keossa.

Edellä oletetaan, että lisäyksessä taulukon koko ei lopu kesken. Jos halutaan varautua taulukon koon kasvattamiseen, käytetään samanlaista kaksinkertaistusideaa kuin taulukkolistalla (s. 167–168).

Tällöin edelleen lisäyksen tasoitettu aikavaativuus on $O(\log n)$ per operaatio, vaikka jotkin yksittäiset operaatiot voivat viedä enemmän aikaa.

Alkion prioriteetin muuttaminen

Toisinaan prioriteettijonossa halutaan sallia myös operaatiot

increasePriority: lisää keossa olevan alkion prioriteettia
decreasePriority: vähentää keossa olevan alkion prioriteettia.

Sinänsä keossa olevan alkion prioriteetin muuttaminen on helppoa: muutetaan se, ja

- jos prioriteetti kasvoi, niin siirretään alkiota kohti juurta kuten lisäyksessä
- jos prioriteeti väheni, niin siirretään alkiota alaspäin kuten poistossa.

Ongelmana on, että ensin pitää löytää alkio keosta.

Esimerkki Tarkastellaan prioriteettijonoa, jossa on järjestelmälle tulleita palvelupyyntöjä kiireellisyyden mukaisesti (s. 295).

Jos halutaan muuttaa jonkin pyynnön kiireellisyyttä, pitää ensin kyseisen pyynnön tunnuskoodin perusteella löytää sen sijainti keossa. Käytännössä tämä vaatisi esim. hajautustaulun, jossa jokaiselle tunnuskoodille on tallennettu osoite keossa.

Nyt aina kun keossa tehdään muutoksia, niin siirtyneiden alkioiden osoitteet hajautustaulussa pitää päivittää.

Tämä on täysin mahdollista, mutta aiheuttaa ylimääräistä työtä myös kun prioriteetin muutosoperaatioita ei tarvita. Siksi esim. Javan prioriteettijonototeutus ei tarjoa prioriteetin muutosoperaatioita.

Taulukon organisoiminen keoksi

Oletetaan, että taulukossa taulu[$1 \ldots n$] on n alkiota. (Huomaa, että indeksointi alkaa ykkösestä kekomallin mukaisesti.)

Halutaan järjestellä alkioita niin, että taulukko keoksi tulkittuna toteuttaa kekoehdon.

Ongelman voi ratkaista järjestämällä taulukon. Osoittautuu kuitenkin, että myös ajassa $O(n)$ toimiva ratkaisu on mahdollinen.

Taulukosta tehdään keko seuraavasti:

```
teeKeko(taulu[1...n])
    viimeinen = n
    viimeinenLapsellinen = vanhempi(viimeinen)
    for i=viimeinenLapsellinen to 1 step -1
        // nyt taulu[i+1...viimeinen] toteuttaa keko-ominaisuuden
        painaAlas(i)
```

Periaate:

- saatetaan lopusta alkaen solmu kerrallaan keko-ominaisuus voimaan
- solmut käsitellään lopusta alkaen, koska silloin aina kutsussa painaAlas(i) pätee oletus, että solmusta i alkava alipuu solmua i lukuunottamatta toteuttaa kekoehdon
- alussa voidaan hypätä lapsettomien solmujen yli, koska niissä kekoehto pätee automaattisesti.

Proseduurin teeKeko aikavaativuuden analysoimiseksi todetaan ensin, että kutsun painaAlas(i) aikavaativuus on $O(1 + \text{vaihdot}(i))$, missä vaihdot(i) on kutsun painaAlas(i) aiheuttamien vaihtojen lukumäärä

Siis proseduurin teeKeko aikavaativuus on

$$O\left(\sum_{i=1}^{n}(1 + \text{vaihdot}(i)\right) = O(n + \text{vaihtojen kokonaismäärä}),$$

missä vaihtojen kokonaismäärä tarkoittaa kaikkien painaAlas-kutsujen aikana tehtyjen vaihtojen yhteismäärää.

Koska yksi kutsu aiheuttaa enintään $O(\log n)$ vaihtoa ja kutsuja on $O(n)$, aikavaativuudelle saadaan suoraan yläraja $O(n \log n)$. Tarkemmalla analyysilla saadaan kuitenkin tarkemmin, että aikavaativuus on $O(n)$.

Analysoidaan siis vaihtojen kokonaismäärää. Ryhmitellään vaihdot sen mukaan, miltä tasolta alkio siirtyy alaspäin:

- tasolla 0 on vain yksi alkio, joka voi siirtyä alaspäin
- tasolta 1 alaspäin voi pahimmassa tapauksessa siirtyä tason 1 alkuperäiset 2 alkio ja lisäksi yksi tasolta 0 tasolle 1 siirtynyt alkio
- tasolta 2 alaspäin voivat pahimmassa tapauksessa siirtyä kaikki tasojen 0, 1 ja 2 alkuperäiset arvot, joita on $1 + 2 + 4$ kappaletta
- yleisesti tasolta k alaspäin voi pahimmassa tapauksessa siirtyä $2^0 + 2^1 + 2^2 + \ldots + 2^k = 2^{k+1} - 1$ alkiota.

Olkoon nyt keon korkeus h ja solmujen lukumäärä n. Kuten sivulla 299 on todettu, $n \geq 2^h$.

Kun lasketaan vaihdot tasoittain, saadaan niiden määrälle yläraja

$$\sum_{k=0}^{h-1}(2^{k+1} - 1) = 2 \cdot \sum_{k=0}^{h-1} 2^k - h = 2 \cdot 2^h - h \leq 2n.$$

Siis koko proseduurin teeKeko aikavaativuus on $O(n)$.

Kekojärjestäminen

Kekojärjestämisen perusajatus on seuraava:

- muodosta taulukosta maksimikeko
- poista keosta alkioita yksi kerrallaan (suurimmasta pienimpään) ja täytä taulukko niillä lopusta alkaen.

Koska keko pienenee samaa tahtia kuin lopullinen taulukko täyttyy, ei tarvita mitään aputaulukkoa.

Siis samaa taulukkoa taulu$[1 \ldots n]$ käytetään suorituksen aikana kolmeen eri tarkoitukseen:

- aluksi se sisältää alkiot täysin mielivaltaisessa järjestyksessä
- sitten se organisoidaan keoksi
- lopuksi sinne muodostetaan järjestetty taulukko alkio kerrallaan viimeisestä alkaen.

Sama pseudokoodina:

```
kekojärjestäminen(taulu[1 ... n])
    teeMaksimiKeko(taulu)    // nyt viimeinen == n
    for i=1 to n − 1
        x = deleteMax()      // viimeinen pienenee
        taulu[viimeinen+1] = x
```

Koska maksimikeon muodostaminen tapahtuu ajassa $O(n)$ ja silmukassa on $O(n)$ prioriteettijono-operaatiota, jotka vievät ajan $O(\log n)$

- aikavaativuus on $O(n \log n)$
- tilavaativuus on $O(1)$.

Tämä ei muuta sitä aiemmin esitettyä havaintoa, että käytännössä hyvin toteutettu pikajärjestäminen on yleensä paras.

Kekojärjestäminen on kuitenkin ainoa vaihtoehto, jos halutaan pahimman tapauksen aikavaativuus $O(n \log n)$ ja vakiotila.

Prioriteettijono Javassa

Javassa on valmiina luokka `PriorityQueue`, joka toteuttaa (minimi)prioriteettijonon.

Tässä prioriteettijonon alkioilla ei ole erikseen määriteltyä prioriteettikenttää, vaan prioriteettijono organisoidaan alkioluokan luonnollisen järjestyksen suhteen.

Metodit ovat

- `add`: kuten insert edellä; aikavaativuus $O(\log n)$
- `peek`: kuten min edellä; aikavaativuus $O(1)$
- `poll`: kuten deleteMin edellä; aikavaativuus $O(\log n)$.

Metodit `contains` ja `remove` mahdollistavat mielivaltaisen alkion etsimisen ja poistamisen, mutta niiden aikavaativuus on $O(n)$.

Konstruktorissa voidaan antaa talletusalueen koolle alkuarvo tai jättää se oletusarvoon (joka on 11).

Maksimiprioriteettijonon voi määritellä antamalla konstruktorille parametrina käänteisen järjestyksen:

```
PriorityQueue<Integer> jono =
    new PriorityQueue<Integer>(100, Collections.reverseOrder());
```

Jos alkioluokkana haluaa käyttää itse määriteltyä luokkaa, sille pitää toteuttaa metodi `compareTo` ja merkitä, että se toteuttaa `Comparable`-rajapinnan.

Yhteenveto

Keko toteuttaa prioriteettijonon operaatiot seuraavilla aikavaativuuksilla:

insert: $O(\log n)$
min: $O(1)$
deleteMin: $O(\log n)$.

Insert-operaation yhteydessä tämä on pahimman tapauksen tasoitettu aikavaativuus, jos halutaan varautua talletusalueen koon dynaamiseen kasvattamiseen. Muuten kaikki aikavaativuudet pätevät pahimmassa tapauksessa.

Taulukon organisoiminen keoksi onnistuu ajassa $O(n)$.

Kekojärjestämisellä pahimman tapauksen aikavaativuus on $O(n \log n)$ ja tilavaativuus $O(1)$.

8. Algoritmien suunnittelutekniikoita

Tarkastelemme tekniikoita erityisesti kombinatorisiin etsintä- ja optimointiongelmiin, joissa annetuista peruskomponenteista tulee koota tietyt ehdot täyttävä ratkaisu. Esimerkkinä tästä on viikolla 1 esitetty tasajako-ongelma: jaa lukujoukko kahteen osaan, joiden summat ovat yhtä suuret.

Tämän luvun jälkeen opiskelija

- tuntee peruuttamiseen perustuvan läpikäynnin perusajatuksen ja osaa toteuttaa sitä käyttävän algoritmin.
- osaa tehostaa ratkaisujen läpikäyntiä branch and bound -menetelmällä
- tuntee dynaamisen ohjelmoinnin ajatuksen ja osaa soveltaa sitä perustilanteissa.

Tämä luku syventää luvussa 1 rekursion yhteydessä nähtyjä esimerkkejä ja valmistaa myöhemmin verkkojen yhteydessä nähtäviin vaikeampiin algoritmeihin.

Näiden luentojen suhde kurssikirjaan

Luentoviikolla 8 käsitellään tämä luentomateriaalin luku 8.

Laaksosen oppikirjasta tämä vastaa lukuja 8 ja 9.

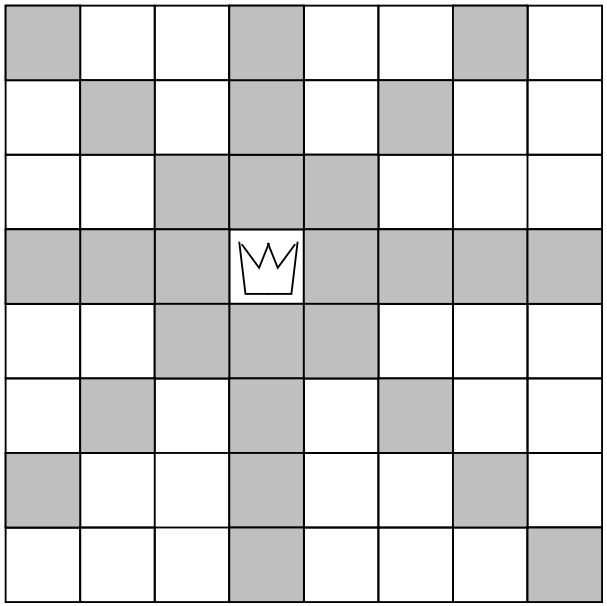
- Oppikirjan lukua 8 ei juurikaan käsitellä, vaan sen sijaan esitellään toisentyyppisiä algoritmitekniikoita.
- Oppikirjan luvun 9 asia eli dynaaminen ohjelmointi käsitellään suppeammin kuin kirjassa.
- Ratkaisun taustalla on kokemukset syksyltä 2018, jolloin oppikirjaa käytettiin ensimmäistä kertaa.
- Tässä kohdassa voi olla hyödyllistä kerrata myös oppikirjan luku 1.3 rekursiosta.

Tämän jälkeen luentoviikkojen numerointi on yhden jäljessä oppikirjan lukujen numeroinnista.

Esimerkkiongelma: kahdeksan kuningatarta

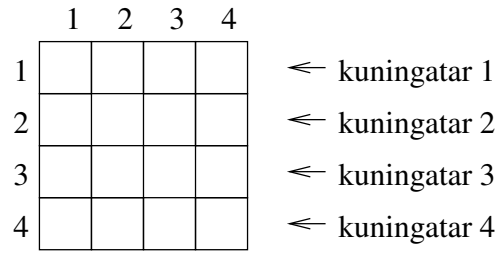
Kahdeksan kuningattaren ongelma on perinteinen ajanvieteongelma: miten sijoitetaan shakkilaudalle 8 kuningatarta siten, että ne eivät uhkaa toisiaan?

Shakissa kuningatar uhkaa samalla rivillä, sarakkeella (eli shakkiterminologialla linjalla) sekä diagonaalilla olevia ruutuja



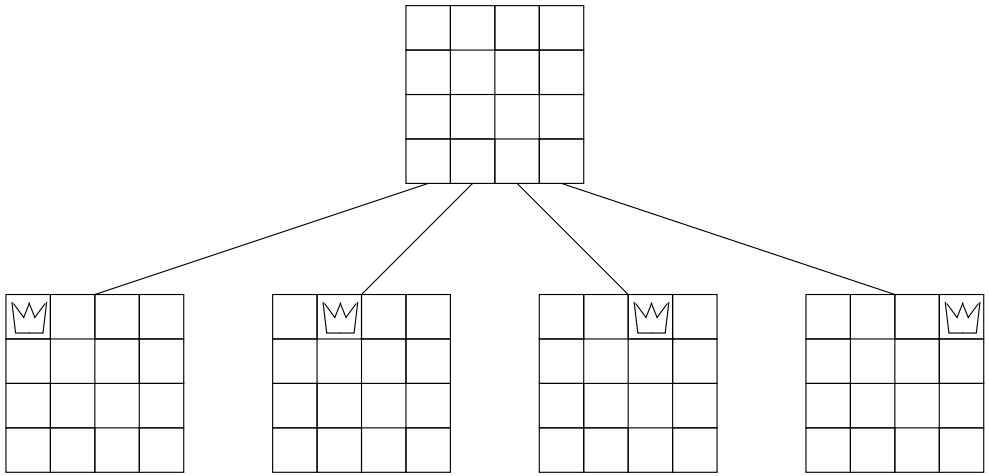
Ongelman yleistetyssä versiossa pitää sijoittaa n kuningatarta $n \times n$ ruudun laudalle. Ongelma on ratkaistavissa kaikilla $n \geq 4$.

Tarkastellaan ensin tapausta $n = 4$. Selvästi jokaisella rivillä täytyy olla tasan 1 kuningatar:



Etsitään oikea kuningatarasetelma systemaattisesti:

- aloitetaan tyhjältä laudalta
- tämän jälkeen asetetaan kuningatar riville 1
- neljä eri mahdollisuutta:



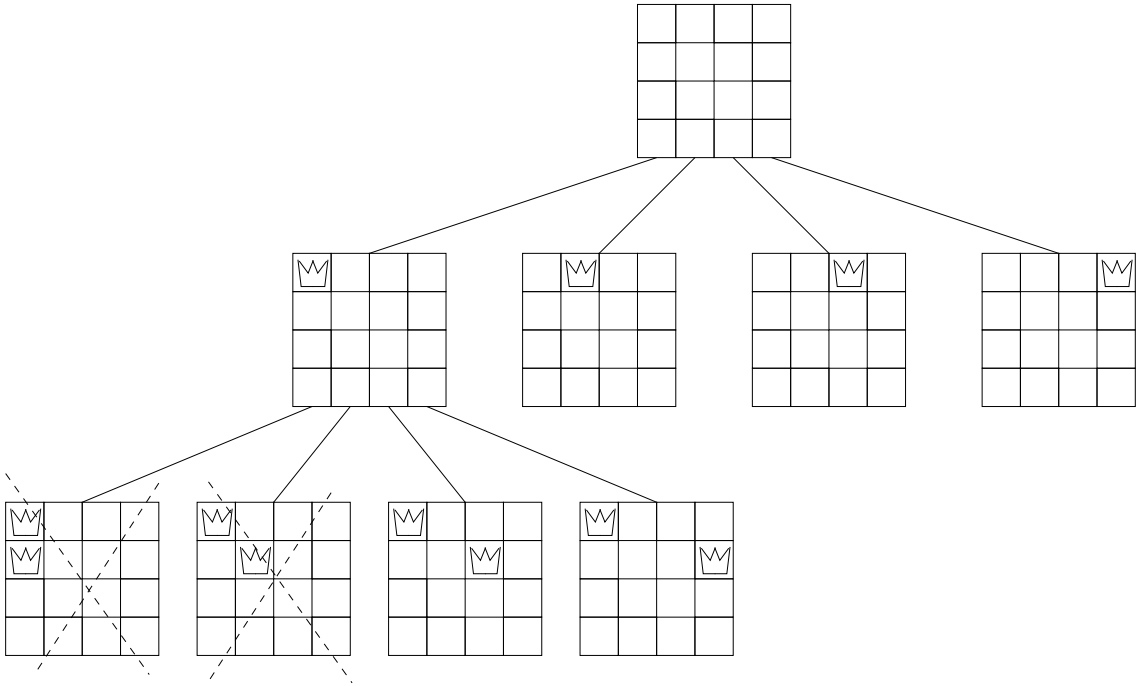
Ongelman ratkaisuprosessi voidaan mallintaa puuna:

- Juurena on tyhjä lauta.
- Solmuilla on lapsia, jotka saadaan laittamalla laudalle yksi kuningatar lisää. Tarkastelemme tässä ratkaisutapaa, jossa lautaa täytetään rivi kerrallaan.
- Lehtiä ovat asemat, joissa jokaisella rivillä on yksi kuningatar.
- Tavoitteena on löytää yksi lehti, jossa uhkaamattomuusehto pätee.

Käytännössä karsimme puusta heti pois haarat, joissa on lisätty kaksi toisiaan uhkaavaa kuningatarta.

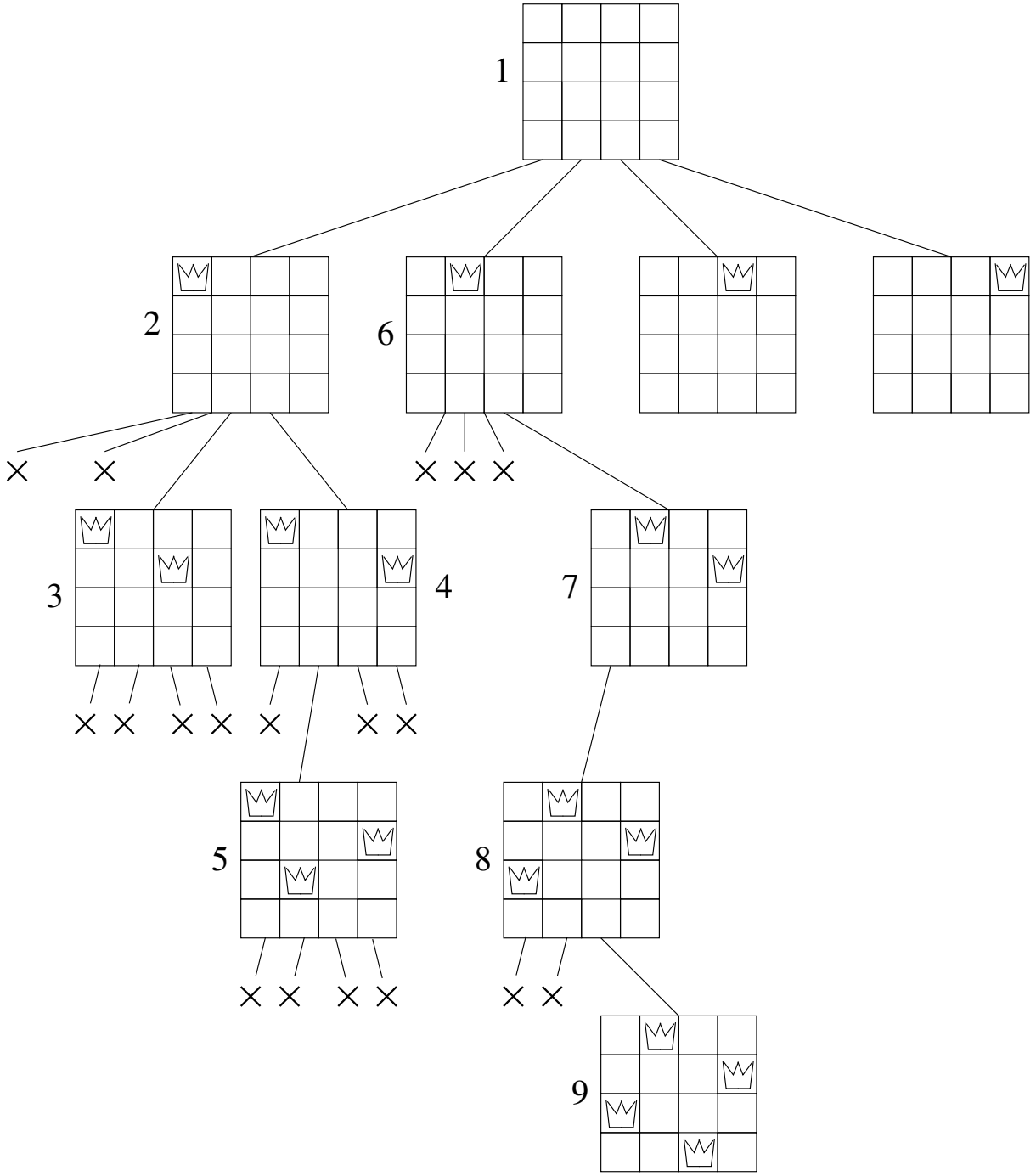
Jatketaan siis edellä aloitetun puun täyttämistä.

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



Kaksi vasemmanpuoleisinta yritystä karsitaan, koska rivien 3 ja 4 sijoituksista riippumatta niistä ei voi tulla kelvollista ratkaisua.

Seuraavalla sivulla on ratkaisun löytymiseen asti piirretty ratkaisupuu.


1
2
6
3
4
7
5
8
9

Edellisen sivun kuvassa ratkaisun aikana tarkastellut solmut on numeroitu niiden läpikäyntijärjestyksessä, joka on samalla puun esijärjestys.

(Määrittelimme luvussa 6 esijärjestyksen vain binääripuille, mutta esi- ja jälkijärjestys yleistyvät luonnollisella tavalla muillekin puille.)

Huomaa, että tätä puuta ei missään vaiheessa ole tarkoitus muodostaa tietorakenteena tietokoneen muistiin. Se on vain käsitteellinen apuväline.

Käytännössä puun läpikäynti tehdään rekursiolla, kuten kohta tarkemmin näemme. Tällöin muistissa on kerrallaan yhtä juuresta lähtevää polkua vastaava osa puusta.

Tämä ratkaisumenetelmä on esimerkki raakaan voimaan (brute force) perustuvasta algoritmista:

- Mahdollisten ratkaisujen joukkoa eli ratkaisuavaruutta käydään läpi systemaattisesti, kunnes haluttu ratkaisu löytyy.
- Emme yritäkään nopeuttaa ratkaisun löytymistä ongelman luonteeseen perustuvilla päätelmillä tms.

Tällaista ratkaisuavaruuden läpikäyntiä sanotaan peruuttavaksi etsinnäksi (backtracking):

- Jos solmu ei ole lehti, mutta kaikki sen lapset on käsitelty tai karsittu, niin kyseisestä solmusta ei pääse eteenpäin.
- Ratkaiseminen jatkuu peruuttamalla eli poistamalla viimeksi lisättyjä kuningattaria kunnes päädytään solmuun, jossa on vielä kokeilemattomia vaihtoehtoja.

Tällaisten ratkaisualgoritmien aikavaativuus on yleensä hyvin suuri; niin tässäkin.

Esitämme n kuningattaren ongelman ratkaisun täsmällisemmin pseudokoodina. Tämä johtaa samantyyppiseen rekursiiviseen algoritmiin, joita tarkasteltiin luvussa 1 esimerkkinä rekursiosta.

Parametrilistojen lyhentämiseksi käytämme paria globaalia muuttujaa:

- n on ongelman koko (laudan rivien ja sarakkeiden lukumäärä)
- paikat[0...n-1] on taulukko, johon ratkaisua rakennetaan rivi kerrallaan. Jos rivin i kuningatar on asetettu sarakkeelle j, niin paikat[i]==j.

Aloitamme siis rivien ja sarakkeiden numeroinnin nollasta.

Algoritmin runkona toimii funktio kuningattaret.

Kutsu kuningattaret(k)

- olettaa, että osataulukko paikat[0. . . k-1] sisältää kelvollisen tavan sijoittaa k ensimmäistä kuningatarta k ensimmäiselle riville
- palauttaa true jos taulukon loppuosa paikat[k. . . n-1] on mahdollista täyttää niin, että saadaan kelvollinen ratkaisu; lisäksi tulostaa tämän ratkaisun
- palauttaa false muuten.

Funktion toiminta-ajatus on kokeilla sijaintiin paikat[k] kaikkia mahdollisia arvoja ja tarkastaa ne rekursiivisesti.

Ratkaiseminen aloitetaan kutsulla kuningattaret(0).

Käytämme kahta apumetodia:

tulosta() tulostaa taulukon paikat[0...n-1] mukaisen ratkaisun sopivaksi katsotulla tavalla

sallittu(r, s) palauttaa *true*, jos aiemmat kuningattaret osataulukossa paikat[0...r-1] eivät estä uuden kuningatteren sijoittamista rivin r sarakkeelle s:

```
sallittu(r, s)
    for i=0 to r-1
        if (paikat[i]==s)                    // sarake varattu
            return false
        if (|r − i| == |s − paikat[i]|)      // diagonaali varattu
            return false
    return true
```

Ratkaisu tulee nyt muotoon

```
kuningattaret(seuraavaRivi)
    if (seuraavaRivi == n)    //päästiin ratkaisuun asti
        tulosta()
        return true
    for sarake=0 to n-1
        if (sallittu(seuraavaRivi, sarake))
            paikat[seuraavaRivi] = sarake
            if (kuningattaret(seuraavaRivi+1))
                // lopetetaan ensimmäiseen ratkaisuun
                return true
    return false
```

- Kun yksikin ratkaisu löytyy, funktio palauttaa true ja rekursio purkautuu.
- Kun jossain kutsussa kaikki sarake-valinnat on käyty läpi tuloksetta, palautetaan false , jolloin peruutetaan edelliseen valintakohtaan.

Kun ei lasketa rekursiivisissa kutsuissa kuluvaa aikaa, niin yksi kuningattaret-kutsu vie ajan $O(n^2)$.

Rekursiivisten kutsujen lukumäärää on vaikea arvioida.

- Jonkinlaisen ylärajan saa siitä, että jokaiselle sarakkeelle pitää tulla tasan yksi kuningatar, joten taulukko paikat[0...n-1] käy läpi lukujen $0 \ldots n-1$ permutaatioita.
- Siis puussa on enintään $n!$ lehteä.
- Siis solmuja kaikkiaan on enintään $n \cdot n!$.

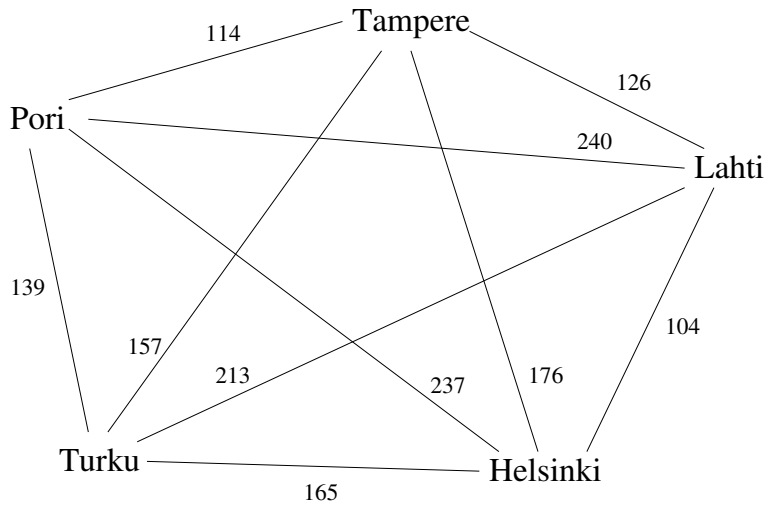
Tästä saadaan kokonaisaikavaativuudelle karkea yläraja $O(n^3 n!)$.

Käytännössä

- yhden ratkaisun löytyminen on melko nopeaa suurillakin n
- jos halutaan käydä kaikki ratkaisut läpi (esim. laskea ratkaisujen lukumäärä), aikavaativuus kasvaa nopeasti
- suurin n, jolla ratkaisujen lukumäärä tunnetaan, on 27.

Kauppamatkustajan ongelma (Travelling Salesman Problem, TSP)

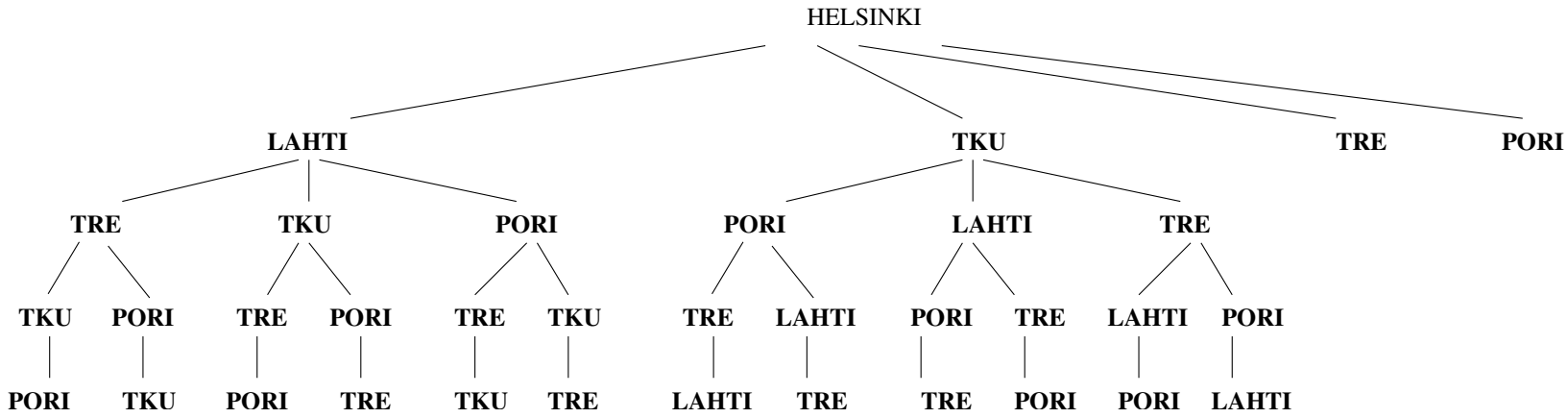
Helsingissä asuvan kauppamatkustajan täytyy vierailla Lahdessa, Turussa, Porissa ja Tampereella.



Mikä on lyhin reitti joka alkaa Helsingistä ja päättyy Helsinkiin ja sisältää yhden vierailun kussakin kaupungissa?

Ongelman eri variaatioilla on runsaasti käytännön sovelluksia, ja ratkaisualgoritmeja on tutkittu paljon. Tarkastelemme tässä vain ratkaisun yleisperiaateita.

Ratkaisuavaruus voidaan esittää samantyyppisenä puuna kuin kuningatarongelmassa:



- Juuri esittää tilannetta, jossa ei vielä ole lähdetty alkupisteestä.
- Lehtinä on valmiit reitit. (Paluuta lopuksi lähtökaupunkiin ei ole enää merkitty.)
- Solmulla on lapsina kaikki vielä läpikäymättömät kaupungit.
- Solmu edustaa osareittiä, jossa käydään kaupunkeja läpi samassa järjestyksessä kuin juuresta kyseiseen solmuun johtavalla polulla.

Puu voidaan käydä läpi samaan tapaan kuin kuningatarongelmassa.

Nyt kuitenkaan läpikäyntiä ei voi lopettaa ensimmäiseen lehteen, vaan ne pitää kaikki käydä läpi, jotta varmasti löydetään lyhin reitti.

Oletamme, että globaaleina muuttujina on määritelty

- kaupunkien lukumäärä n
- etäisyystaulukko etäisyys[0...n-1, 0...n-1], missä etäisyys[i,j] on etäisyys kaupungista i kaupunkiin j
  - reitti alkaa kaupungista 0
  - etäisyydet ovat symmetriset: etäisyys[i,j]=etäisyys[j,i].

Laskennan runkona on rekursiivinen funktio tsp, joka

- saa parametrina osittaisen reitin
- palauttaa arvonaan lyhimmän reitinpituuden, joka saadaan jatkamalla tätä osittaista reittiä.

Osittainen reitti esitetään antamalla

- luonnollinen luku k, joka kertoo, kuinka monta kaupunkia reitille on jo valittu
  - jos k=1, niin reitillä on vasta alkukohta 0
  - jos k=n, niin reitti on valmis.
- kokonaislukutaulukko reitti, jonka osuus reitti[0...k-1] kertoo jo lisätyt kaupungit
  - alustuksena merkitään alku- ja loppukohta reitti[0]=reitti[n]=0
  - parametri k kertoo, että seuraavaksi ollaan valitsemassa kaupunki reitti[k].

Laskennan helpottamiseksi lisäämme vielä kaksi parametria, jotka voitaisiin haluttaessa laskea parametrien k ja reitti perusteella:

- mukana[0. . . n-1] on boolean-taulukko, jossa
  mukana[i]==true jos ja vain jos kaupunki i on valittu reitille,
  ts. reitti[j]==i jollain $0 \leq j \leq$ n-1
- kokonaisluku pituus kertoo osareitin 0 $\rightarrow$ reitti[1] $\rightarrow$ reitti[2] $\rightarrow \ldots \rightarrow$ reitti[k-1] pituuden.

Laskenta käynnistyy kutsulla tsp(reitti, mukana, 1, 0), missä

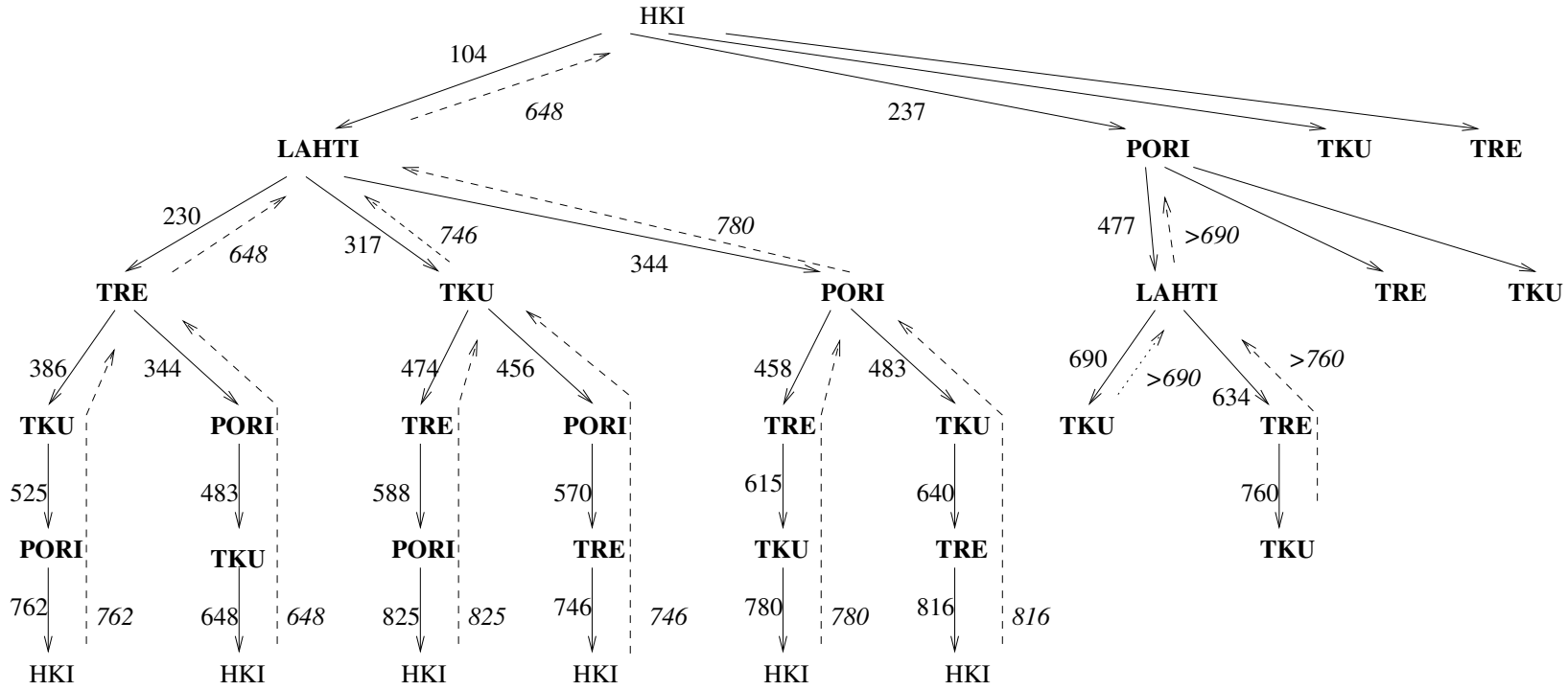
- reitti[0] ja reitti[n] on alustettu 0:aan (tarkoittaen kaupunkia 0)
- mukana[0]==true ja mukana[i]==false muuten
- parametri 1 tarkoittaa, että seuraavaksi valitaan reitti[1]
- parametri 0 tarkoittaa, että tähänastisen reitin pituus on nolla.

Aliohjelma tulee muotoon

```
tsp(reitti, mukana, k, pituus)
    if (k==n)
        // reitistä puuttuu enää paluu alkukohtaan
        return pituus + etäisyys[reitti[n-1],0]
    paras = SUURI    // suurempi kuin mikään mahdollinen reitin pituus
    for i=1 to n-1
        // kokeillaan kaupunkia i reitin kohtaan k
        if (mukana[i]==false)
            reitti[k] = i
            mukana[i] = true
            arvo = tsp(reitti, mukana, k+1, pituus+etäisyys[reitti[k-1],i])
            paras = min(paras, arvo)
            mukana[i] = false
    return paras
```

Puun läpikäyntiä voidaan tehostaa

- pitämällä kirjaa parhaasta tähän mennessä löytyneestä reitistä
- lopettamalla haaran tutkiminen, kun nähdään, että se ei voi parantaa tulosta.



Esim. osareittiä Helsinki → Pori → Lahti → Turku → ?
ei kannata tutkia pidemmälle, koska on jo matkustettu 690 km, kun reitillä
Helsinki → Lahti → Tampere → Pori → Turku → Helsinki
koko reitti on vain 648 km.

Liitetään mukaan parametri paras, jossa on paras toistaiseksi löydetty reitinpituus:

```
tsp(reitti, mukana, k, pituus, paras)
    if (k==n)
        return pituus + etäisyys[reitti[n-1],0]
    for i=1 to n-1
        if (mukana[i]==false)
            reitti[k] = i
            mukana[i] = true
            uusiPituus = pituus+etäisyys[reitti[k-1],i]
            if (uusiPituus<paras)
                arvo = tsp(reitti, mukana, k+1, uusiPituus, paras)
                paras = min(paras, arvo)
            mukana[i] = false
    return paras
```

Tästä yleisestä ratkaisuperiaatteesta käytetään nimeä branch and bound.

Jokaiselle puun haaralle (branch) lasketaan *alaraja* (bound) joka kertoo, kuinka suuri kustannus kaikilla haarasta löytyvillä ratkaisuilla ainakin on.

Jos aiemmin on jo löydetty ratkaisu, jonka kustannus on pienempi kuin käsillä olevan haaran alaraja, niin haara voidaan karsia.

- Algoritmi antaa oikean lopputuloksen, kunhan on taattua, että alaraja on varmasti pienempi tai yhtäsuuri kuin mikään haarasta löytyvä ratkaisu.
- Mitä suurempi alaraja on, sitä tehokkaammin se karsii puuta ja nopeuttaa laskentaa.

Edellä alarajaksi valittiin uusiPituus, joka laskee tähän asti kuljetun matkan.

Koska etäisyydet eivät voi olla negatiivisia, tämä on turvallinen alaraja.

Tiukempia alarajoja voi johtaa tarkastelemalla myös niitä kaupunkeja, jotka vielä ovat reitin ulkopuolella.

Eräs mahdollisuus on lisätä arvoon uusiPituus jokaiselle kaupungille k, joka ei vielä ole reitillä, arvo

$$\frac{1}{2}\left(\text{etäisyys[k,n1]} + \text{etäisyys[k,n2]}\right),$$

missä n1 ja n2 ovat kaupungin k kaksi lähintä naapuria.

Tämän pienemmällä kustannuksella nimittäin kaupunkia k ei voi liittää osaksi reittiä.

Branch and bound -tekniikan hyödyllisyys on tapauskohtaista ja vaikeaa arvioida.

Eräällä 17 kaupungin testiaineistolla suoraviivaisella toteutuksella ja kannettavalla peruskoneella

- pelkästään arvoa uusiPituus alarajana käyttävä algoritmi (s. 347) käytti noin 15 minuuttia ja tutki noin 18 miljardia solmua (eli teki $18 \cdot 10^9$ tsp-kutsua)
- sivulla 349 hahmotellulla alarajalla kului noin 20 sekuntia ja tutkittiin noin 7 miljoonaa solmua
- täysin optimoimaton algoritmi veisi arviolta useita päiviä.

Toisella, 26 kaupunkia sisältävällä, testiaineistolla tehokkaampikin branch and bound -algoritmi vei 21 minuuttia.

Tämä kokeilu **ei missään nimessä** edusta viimeisintä sanaa kauppamatkustajan ongelman ratkaisemisessa.

Kauppamatkustajan ongelmaa on tutkittu paljon, koska se on sekä sovellusten että teorian kannalta kiinnostava.

Laskennan teorian näkökulmasta kauppamatkustajan ongelma on NP-kova ongelma: se kuuluu suureen joukkoon ongelmia, joista ei tiedetä, onko niille olemassa polynomisessa ajassa toimiva ratkaisualgoritmi. Tähän liittyvistä kysymyksistä saa hieman lisätietoa kurssilla *Laskennan mallit*.

Näihin ongelmiin liittyvien algoritmien teoriaa ja käytäntöä tarkastellaan enemmänkin algoritmiikan maisterikursseilla, kuten *Design and Analysis of Algorithms* ja erityisesti *Combinatorial Optimization*.

Tasajako-ongelma (partition)

Tarkastellaan toisena esimerkkinä branch and bound -tekniikasta sivulla 60 esitettyä tasajako-ongelmaa:

- annettu taulukko luvut[0...n-1] positiivisia kokonaislukuja
- kysytään, voiko taulukon luvut jakaa kahteen osaan, joiden summa on sama.

Seuraavalla sivulla on sivun 60 algoritmi, jonka merkinnät on muutettu vastaamaan edellisiä esimerkkejä:

- taulukko mukana kirjaa tehdyt valinnat niin, että
  mukana[i]==true jos luvut[i] on sijoitettu osaan 1 ja
  mukana[i]==false jos luvut[i] on sijoitettu osaan 2
- k on seuraavavaksi sijoitettavan luvun indeksi.

```
tasajako(mukana, k)
    if (k==n)
        summa1 = 0
        summa2 = 0
        for i = 0 to n-1
            if mukana[i]
                summa1 = summa1 + luvut[i]
            else
                summa2 = summa2 + luvut[i]
        return (summa1 == summa2)
    mukana[k] = true
    b1 = tasajako(k+1)
    mukana[k] = false
    b2 = tasajako(k+1)
    return (b1 or b2)
```

Tehostetaan ensin koodia toteamalla, että

- riittää pitää kirjaa joukkojen 1 ja 2 summien erotuksesta
- jos ensimmäinen rekursiivinen kutsu palauttaa true , toinen on turha:

```
tasajako(erotus, k)
    if (k==n)
        return (erotus==0)
    if tasajako(erotus+luvut[k], k+1)
        return true
    else
        return tasajako(erotus-luvut[k], k+1)
```

Branch and bound -karsinnan tekemiseksi todetaan, että

> jos $|\text{erotus}| > \sum_{i=k}^{n-1} \text{luvut[i]}$, niin mikään tapa sijoitella loput luvut ei tuota ratkaisua.

Oletetaan, että summat $\sum_{i=k}^{n-1}$ luvut[i] on tallennettu taulukkoon loppusumma[k].

Saadaan seuraava algoritmi:

```
tasajako(erotus, k)
    if (k==n)
        return (erotus==0)
    if |erotus| > loppusumma[k]
        return false
    if tasajako(erotus+luvut[k], k+1)
        return true
    else
        return tasajako(erotus-luvut[k], k+1)
```

Tätä voidaan vielä tehostaa järjestämällä aluksi taulukko luvut[0...n-1] laskevaan järjestykseen, jolloin loppusummat ovat mahdollisimman pieniä ja karsintaehto laukeaa nopeasti.

Dynaaminen ohjelmointi (dynamic programming)

Dynaaminen ohjelmointi (suomeksi myös *taulukointi*) on yleiskäyttöinen algoritmitekniikka, jossa suoritusta nopeutetaan tallentamalla sopivia osatuloksia muistiin.

(Sana ”ohjelmointi” esiintyy tässä vanhassa merkityksessään ”optimointi”.)

Tarkastelemme johdattelevana esimerkkinä binomikertoimia.

Matematiikasta tiedetään, että n-alkioisesta joukosta voidaan valita k-alkioinen osajoukko

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

eri tavalla.

Miten voimme laskea tämän tekemättä laskutoimituksia hyvin suurilla kertomafunktion arvoilla?

Esimerkki: binomikertoimet

Jos n-alkioisesta joukosta $A = \{ a_1, \ldots, a_n \}$ pitää valita k-alkioinen osajoukko B, niin

- jos $k = 0$, niin ainoa mahdollisuus on $B = \emptyset$ (tyhjä joukko)
- jos $k = n$, niin ainoa mahdollisuus on $B = A$
- jos $0 < k < n$, niin on kaksi mahdollisuutta:
  - valitaan $a_n \in B$ ja sen jälkeen $k - 1$ alkiota joukosta $\{ a_1, \ldots, a_{n-1} \}$
  - valitaan $a_n \notin B$, jolloin kaikki k alkiota valitaan joukosta $\{ a_1, \ldots, a_{n-1} \}$.

Olkoon $B(n,k)$ erilaisten valintojan lukumäärä.

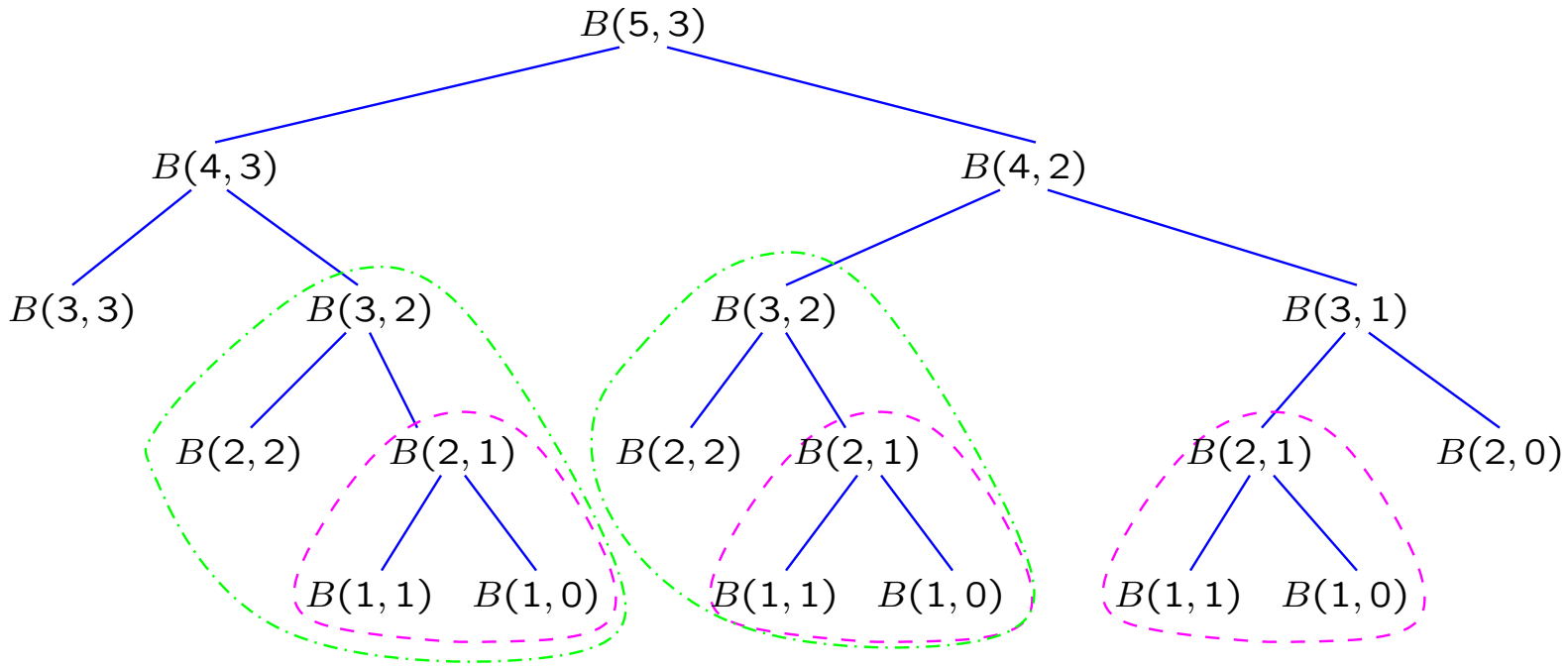
Edellisen perusteella

$$\begin{aligned} B(n,0) &= 1 \\ B(n,n) &= 1 \\ B(n,k) &= B(n-1,k) + B(n-1,k-1) \qquad \text{kun } 0 < k < n. \end{aligned}$$

Tästä palautuskaavasta saadaan suoraan rekursiivinen algoritmi binomikertoimen laskemiseksi (s. 48).

Tämä algoritmi on kuitenkin **erittäin tehoton**, koska samoja arvoja lasketaan monta kertaa.

Laskettaessa $B(5,3)$ arvot $B(3,2)$ ja $B(2,1)$ lasketaan useaan kertaan:



Suuremmilla n ja k tilanne muuttuu nopeasti vielä paljon pahemmaksi.

Suoraviivainen tapa välttää laskujen toistamista on muistiinmerkitseminen (memoization):

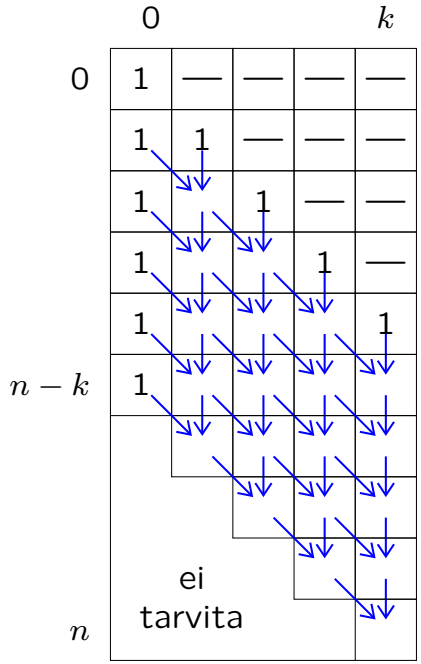
- Kun arvo $B(n,k)$ on saatu lasketuksi, merkitään se muistiin taulukkoon arvo.
- Merkitään myös totuusarvotaulukkoon onLaskettu, että tapaus (n,k) on käsitelty; taulukko alustetaan arvoon false.
- Ennen rekursion aloittamista katsotaan, löytyykö haluttu arvo valmiina.

```
binomi(n,k)
    if ((k==0) or (k==n)) return 1
    if (not onLaskettu[n,k])
        arvo[n,k] = binomi(n-1,k)+binomi(n-1,k-1)
        onLaskettu[n,k] = true
    return arvo[n,k]
```

Aikavaativuus on $O(nk)$, koska jokaista taulukon onLaskettu alkiota kohden tehdään korkeintaan kaksi binomi-kutsua.

Taulukointiratkaisua voidaan yksinkertaistaa ja tehostaa toteamalla suunnittelemalla laskenta etukäteen.

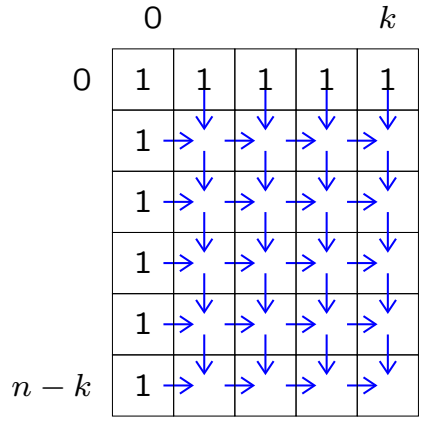
Katsotaan, mitä arvoja $B(i, j)$ tarvitaan arvon $B(n, k)$ laskemiseen:



Täyttämällä taulukko ylhäältä alas ja vasemmalta oikealle saadaan seuraava algoritmi:

```
binomi(n,k)
    for i = 0 to n
        for j = max(0, i-(n-k)) to min(k, i)
            if ((j==0) or (j==i))
                arvo[i,j] = 1
            else
                arvo[i,j] = arvo[i-1,j-1]+arvo[i-1,j]
    return arvo[n,k]
```

Algoritmia voidaan vielä vähän siistiä kääntämällä taulukon diagonaalit riveiksi:



Nyt taulukon kohtaan (i, j) tulee arvo $B(i + j, j)$.
Siis $B(n, k)$ on kohdassa $(n - k, k)$.

Pseudokoodina:

```
binomi(n,k)
    for j = 0 to k
        taulu[0,j] = 1
    for i = 1 to n-k
        for j = 0 to k
            if (j==0)
                taulu[i,j] = 1
            else
                taulu[i,j] = taulu[i,j-1]+taulu[i-1,j]
    return taulu[n-k,k]
```

Algoritmin aikavaativuus on selvästi $O(nk)$.
Myös tilavaativuus on $O(nk)$.

Ratkaisua voidaan vielä tehostaa toteamalla, että taulukon vanhoja rivejä ei tarvitse pitää tallessa:

```
binomi(n,k)
    for j = 0 to k
        rivi[j] = 1
    for i = 1 to n-k
        for j = 0 to k
            rivi[j] = rivi[j-1]+rivi[j]
    return rivi[k]
```

Aikavaativuus on edelleen $O(nk)$, mutta tilavaativuudeksi tulee $O(k)$.

Ongelman ratkaisemisessa dynaamisella ohjelmoinnissa on keskeistä, että

- ongelman ratkaisu saadaan kootuksi pienempien osaongelmien ratkaisuista
- osaongelmat voidaan ratkaista toisistaan riippumatta
- keskenään identtiset osaongelmat toistuvat.

Tällaisen algoritmin laatimisessa oleellista on keksiä, miten jako osaongelmiin kannattaa tehdä, ja laatia tämän perusteella rekursio.

Sopivan jaon keksiminen voi olla erittäin vaikeaa. Tässä auttaa, että tuntee esimerkkejä tekniikan soveltamisesta.

Sellaisissakin tilanteissa, joissa rekursio itse asiassa ei ole erityisen monimutkainen, dynaamiseen ohjelmointiin perustuva ratkaisu voi tuntua epäintuitiiviselta:

- ongelmaa tarkastellaan kokoavasti (bottom-up), kun tietojenkäsittelyssä tyypillisemmin suositaan osittavia (top-down) ratkaisuja
- algoritmi tekee paljon näennäisen turhaa laskentaa, ennen kuin nähdään, miten se johtaa haluttuun lopputulokseen.

Osajoukkosumma (subset sum)

Hieman monipuolisempana esimerkkinä tarkastelemme osajoukkosumman nimellä tunnettua algoritmia:

> Annettu: taulukko $A[0 \ldots n]$ positiivisia kokonaislukuja, positiivinen kokonaisluku M
>
> Kysymys: Voidaanko taulukon A luvuista valita sellainen osajoukko, että valittujen lukujen summa on tasan M.

Siis matemaattisesti, kun on annettu $A = \{ a_0, \ldots, a_{n-1} \} \subseteq \mathbb{N}_+$ ja $M \in \mathbb{N}_+$, onko olemassa $I \subseteq \{ 0, \ldots, n-1 \}$, jolla

$$\sum_{i \in I} a_i = M.$$

Erikoistapaus $M = \frac{1}{2} \sum_{i=0}^{n-1} a_i$ on sama kuin aiemmin esitetty tasajako-ongelma.

Toisaalta osajoukkosummaongelma itse on erikoistapaus repunpakkausongelmasta (knapsack problem).

Olkoon $S(A, M) =$ true, jos joukosta A voidaan muodostaa summa M, ja $S(A, M) =$ false muuten.

Dynaamisen ohjelmoinnin soveltamiseksi meidän pitää löytää ongelmasta $S(A, M)$ sopiva osaongelmarakenne.

Olkoon $A = \{ a_0, \ldots, a_{n-1} \}$ ja $A' = \{ a_0, \ldots, a_{n-2} \} = A - \{ a_{n-1} \}$. Miten voimme ratkaista ongelman $S(A, M)$, jos oletetaan, että $S(A', M)$ on ratkaistu?

- Selvästi jos $S(A', M) =$ true, niin myös $S(A, M) =$ true; alkiota a_{n-1} ei tarvita summan muodostamisessa.
- Mutta jos $S(A', M) =$ false, voi silti olla $S(A, M) =$ true.
- Tällöin alkiota a_{n-1} tarvitaan summan muodostamisessa; summa on muotoa

$$M = a_{n-1} + \sum_{i \in I} a_i,$$

  missä $I \subseteq \{ 0, \ldots, n-2 \}$.
- Tässä tapauksessa siis $S(A', M - a_{n-1}) =$ true.

Siistitään nyt edellisen sivun idea rekursioyhtälön muotoon.

Merkitään $S(k, x) =$ true, jos alkioista $\{ a_0, \ldots, a_{k-1} \}$ voidaan saada summa x.

Ottamalla huomioon reunaehdot saadaan rekursio

$$\begin{aligned} S(0,0) &= \text{true} \\ S(0,x) &= \text{false, jos } x > 0 \\ S(k+1,x) &= S(k,x) \textbf{ or } S(k,x-a_k). \end{aligned}$$

Alkuperäisen ongelman vastaus on $S(n, M)$.

Siis kannattaa tallentaa arvoja $S(k, x)$ taulukkoon arvoilla $k = 0, 1, 2, \ldots, n$.
Suurin kiinnostava summan arvo x on $x = M$.

```
osajoukkosumma(A[0...n-1],M)
    taulu[0, 0] = true
    for x=1 to M
        taulu[0, x] = false
    for k=1 to n
        for x=0 to M
            if (x ≥ A[k-1])
                taulu[k, x] = taulu[k-1, x] or taulu[k-1, x-A[k-1]]
            else
                taulu[k, x] = taulu[k-1, x]
    return taulu[n, M]
```

Kuten binomikerrointen tapauksessa, voimme parantaa algoritmin tilavaativuutta, koska vanhoja rivejä ei tarvitse pitää muistissa.

Tässä meidän pitää kuitenkin päivittää uudet arvot riville lopusta alkaen, ettei synny ketjuja, joissa arvo $A[k-1]$ lasketaan moneen kertaan.

```
osajoukkosumma(A[0...n-1],M)
    rivi[0] = true
    for x=1 to M
        rivi[x] = false
    for k=1 to n
        for x=M to A[k-1] step -1
            rivi[x] = rivi[x] or rivi[x-A[k-1]]
    return rivi[M]
```

Edellinen dynaamiseen ohjelmointiin perustuva algoritmi osajoukkosummalle vie ajan $O(nM)$ ja tilan $O(M)$.

On syytä huomata, että tämä aika- ja tilavaativuus eivät ole polynomisia syötteen koon suhteen.

Jos syötteen koko on b bittiä, niin M voi olla 2^{cb} jollain vakiolla $0 < c < 1$.

Jos syötteessä esiintyvät luvut ovat riittävän pieniä, dynaaminen ohjelmointi on käyttökelpoinen.

Jos ei voida olettaa, että luvut ovat pieniä, ongelman voi ratkaista samantyyppisellä peruuttavalla etsinnällä kuin tasajako-ongelman. Tällaisen algoritmin aikavaativuus on kuitenkin eksponentiaalinen lukujen määrän n suhteen.

Myös osajoukkosumma on NP-täydellinen ongelma; syötteen koon suhteen polynomisen algoritmin olemassaolo on avoin ongelma.

Repunpakkausongelma (knapsack problem)

Esitämme vielä repunpakkausongelmasta yleisemmän version, jossa on useita sallittuja ratkaisuja, joista halutaan valita jollain mittarilla paras. Tällaisia ongelmia sanotaan optimointiongelmiksi erotukseksi päätösongelmista, joissa vastaus on ''kyllä'' tai ''ei''.

Repunpakkausongelmassa on annettu joukko X esineitä ja jokaiselle esineelle $x \in X$ paino $w(x) \in \mathbb{N}_+$ ja arvo $v(x) \in \mathbb{N}_+$. Lisäksi on annettu painoraja $W \in \mathbb{N}_+$ (''repun koko''). Tehtävänä on valita sellainen $I \subseteq X$, että

- $\sum_{x \in I} w(x) \leq W$ ja
- $\sum_{x \in I} v(x)$ on mahdollisimman suuri.

Siis halutaan valita mahdollisimman arvokas joukko esineitä ylittämättä kuitenkaan painorajaa.

Todetaan ensin, että ahne heuristiikka, jossa valitaan esineitä kilohinnan mukaan pienenevässä järjestyksessä, ei välttämättä johda optimaaliseen ratkaisuun.

Esimerkki: kaksi esinettä a ja b, missä

- $w(a) = 2$ ja $v(a) = 3$
- $w(b) = 99$ ja $v(b) = 148$
- $W = 100$.

Nyt vain toinen esineistä a ja b mahtuu reppuun, koska $2 + 99 > 100$.

Esineellä a hinta/painosuhde on parempi: $3/2 > 148/99$.

Kuitenkin jos valitaan a, niin ratkaisun arvo on vain 3, kun valitsemalla b saataisiin 148 eli paljon suurempi arvo.

Edellisen sivun ahnetta heuristiikkaa voi yrittää parannella, mutta mitään helppoa aina oikein toimivaa ratkaisua ei löydy.

Tällekään ongelmalle ei tunneta syötteen koon suhteen polynomisessa ajassa toimivaa ratkaisua. Kuten osajoukkosummaongelmassa, voimme kuitenkin ratkaista ongelman dynaamisella ohjelmoinnilla, jos syötteessä esiintyvät luvut ovat pieniä.

Oletetaan, että esinejoukko on $X = \{ 0, \ldots, n-1 \}$.

Kokeillaan samanlaista lähestymistapaa kuin osajoukkosummassa. Olkoon $R(k, p)$ suurin mahdollinen arvo, kun käytössä on esineet $\{ 0, \ldots, k-1 \}$ ja painoraja on p.

Miten $R(k+1,p)$ voidaan ratkaista tarkastelemalla arvoja $R(k,p')$, missä mahdollisesti $p' \neq p$?

- Jos paras ratkaisu painorajalla p ja esineillä $\{0,\ldots,k\}$ ei käytä esinettä k, niin $R(k+1,p) = R(k,p)$.
- Mikä sitten on paras ratkaisu, jos esinettä k käytetään?
- Esineen k lisäksi pitää valita loput esineet $I \subseteq \{0,\ldots,k-1\}$ niin, että

$$w(k) + \sum_{x \in I} w(x) \leq p \qquad \text{eli} \qquad \sum_{x \in I} w(x) \leq p - w(k).$$

  Esineet I kannattaa valita niin, että painorajan $p - w(k)$ puitteissa niiden arvo on suurin mahdollinen.
- Tässä tapauksessa siis $R(k+1,p) = R(k,p-w(k)) + v(k)$.
- Oikea arvo $R(k+1,p)$ on parempi näistä kahdesta vaihtoehdosta.

Saadaan algoritmi

```
repunpakkaus(w, v, W)
    for p=0 to W
        taulu[0,p] = 0
    for k=0 to n-1
        for p=0 to W
            if p-w[k] ≥ 0
                taulu[k+1, p] = max(taulu[k, p], taulu[k, p-w[k]] + v[k])
            else
                taulu[k+1, p] = taulu[k, p]
    return taulu[n, W]
```

Algoritmin aika- ja tilavaativuus ovat $O(nW)$. Kuten osajoukkosummassa, tilavaativuudeksi saadaan $O(W)$ säilyttämällä vain taulukon viimeisin rivi.

Yhteenveto

- Peruuttava etsintä on tapa käydä systemaattisesti läpi kaikki erilaiset ratkaisukombinaatiot.
- Branch and bound on menetelmä karsia etsinnästä pois ratkaisuavaruuden osia niin, että paras ratkaisu silti taatusti löytyy. Tämä perustuu alarajoihin annetun osittaisen ratkaisun täydennyskustannuksille.
- Dynaamisessa ohjelmoinnissa ratkaistaan ensin pienimmät osaongelmat, taulukoidaan ratkaisut ja rakennetaan niistä pikku hiljaa isompia.
- Toimivan osaongelmajaon löytäminen dynaamisessa ohjelmoinnissa voi vaatia huomattavaa kekseliäisyyttä.