COMP.CS.300 Tietorakenteet ja algoritmit 1

Lähteet

Luentomoniste pohjautuu vahvasti prof. Antti Valmarin vanhaan luentomonisteeseen Tietorakenteet ja algoritmit.

Useimmat algoritmit ovat peräisin kirjasta Introduction to Algorithms; Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.

Lisäksi luentomonistetta koottaessa on käytetty seuraavia kirjoja:

- Introduction to The Design & Analysis of Algorithms; Anany Levitin
- Olioiden ohjelmointi C++:lla; Matti Rintala, Jyke Jokinen
- Tietorakenteet ja Algoritmit; Ilkka Kokkarinen, Kirsti Ala-Mutka
- The C++ Standard Library; Nicolai M. Josuttis

1 Johdanto

Mietitään ensin hiukan syitä tietorakenteiden ja algoritmien opiskelulle

Algorithms in the world

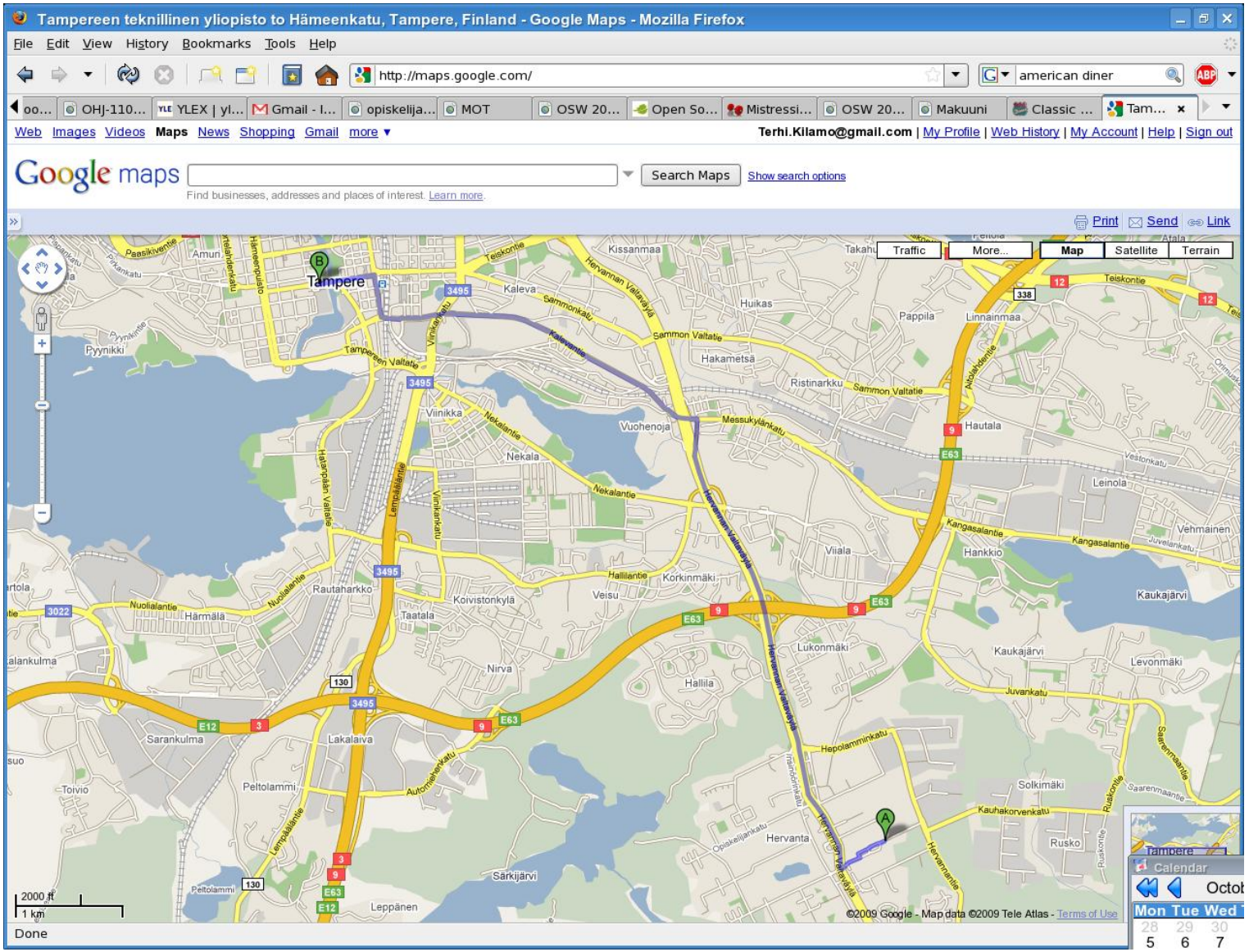
1.1 Miksi?

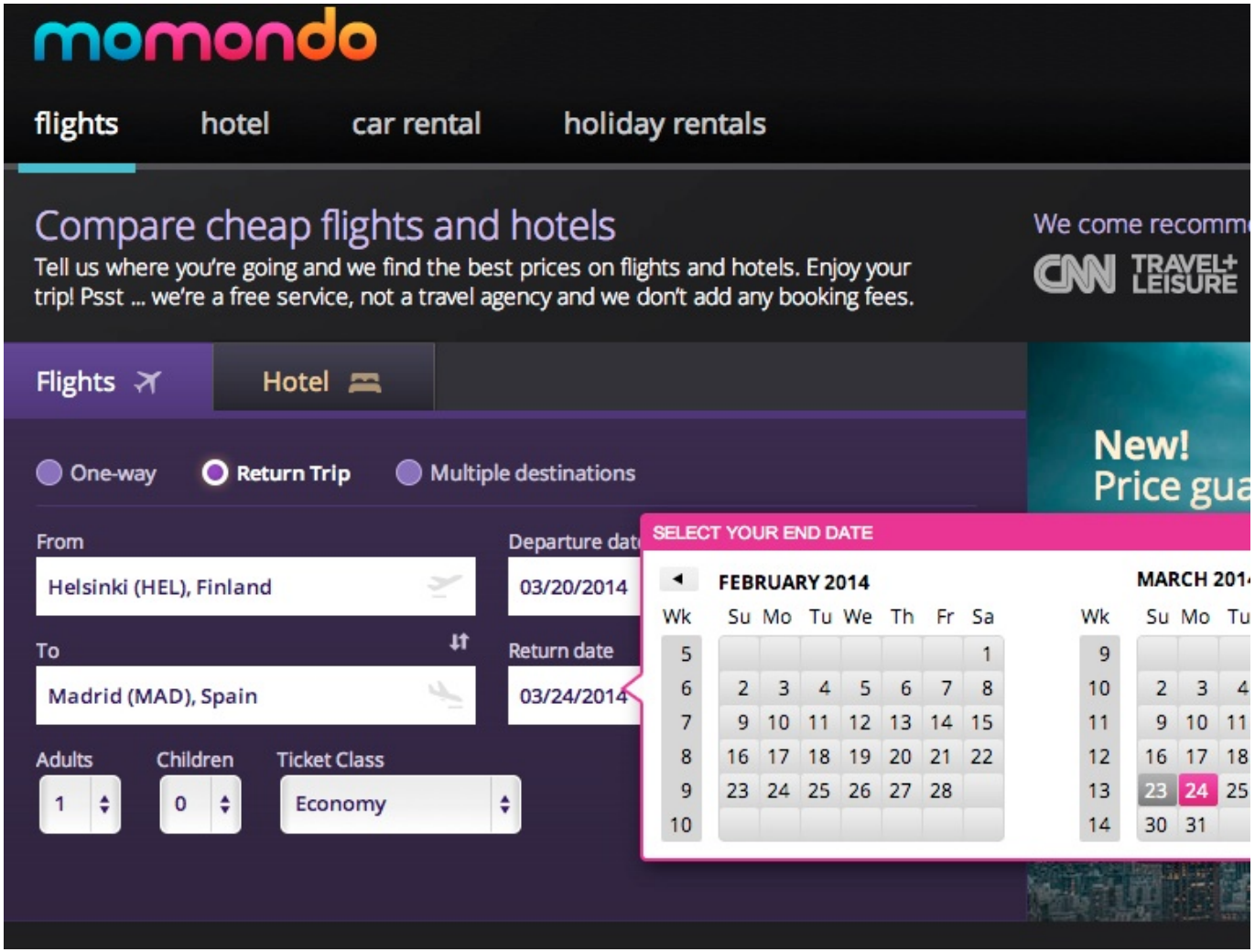
Mitkä ovat sinun elämääsi eniten vaikuttavat algoritmit?

Picture: Chris Watt

Tietokoneohjelmia ei ole olemassa ilman algoritmeja

- algoritmeihin törmäät esimerkiksi seuraavissa sovelluksissa:


Tampereen teknillinen yliopisto to Hämeenkatu, Tampere, Finland - Google Maps - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://maps.google.com/
american diner
Web Images Videos Maps News Shopping Gmail more
Terhi.Kilamo@gmail.com | My Profile | Web History | My Account | Help | Sign out
Google maps
Search Maps
Show search options
Find businesses, addresses and places of interest. Learn more.
Print Send Link
Traffic More... Map Satellite Terrain
Tampere
Pyynikki
Amuri
Kissanmaa
Pappila
Linnainmaa
Hakametsä
Ristinarkku
Viinikka
Vuohenoja
Nekala
Hautala
Leinola
Vehmainen
Hankkio
Kaukajärvi
Viiala
Veisu
Korkinmäki
Rautaharkko
Koivistonkylä
Taatala
Härmälä
Nirva
Hallila
Lukonmäki
Levonmäki
Sarankulma
Lakalaiva
Peltolammi
Toivio
Hervanta
Rusko
Solkimäki
Särkijärvi
Leppänen
2000 ft
1 km
©2009 Google - Map data ©2009 Tele Atlas - Terms of Use
Done
Calendar
Mon Tue Wed



aina, kun käytät tietokonetta, käytät myös algoritmeja.

1.2 Kaikki pennillä?

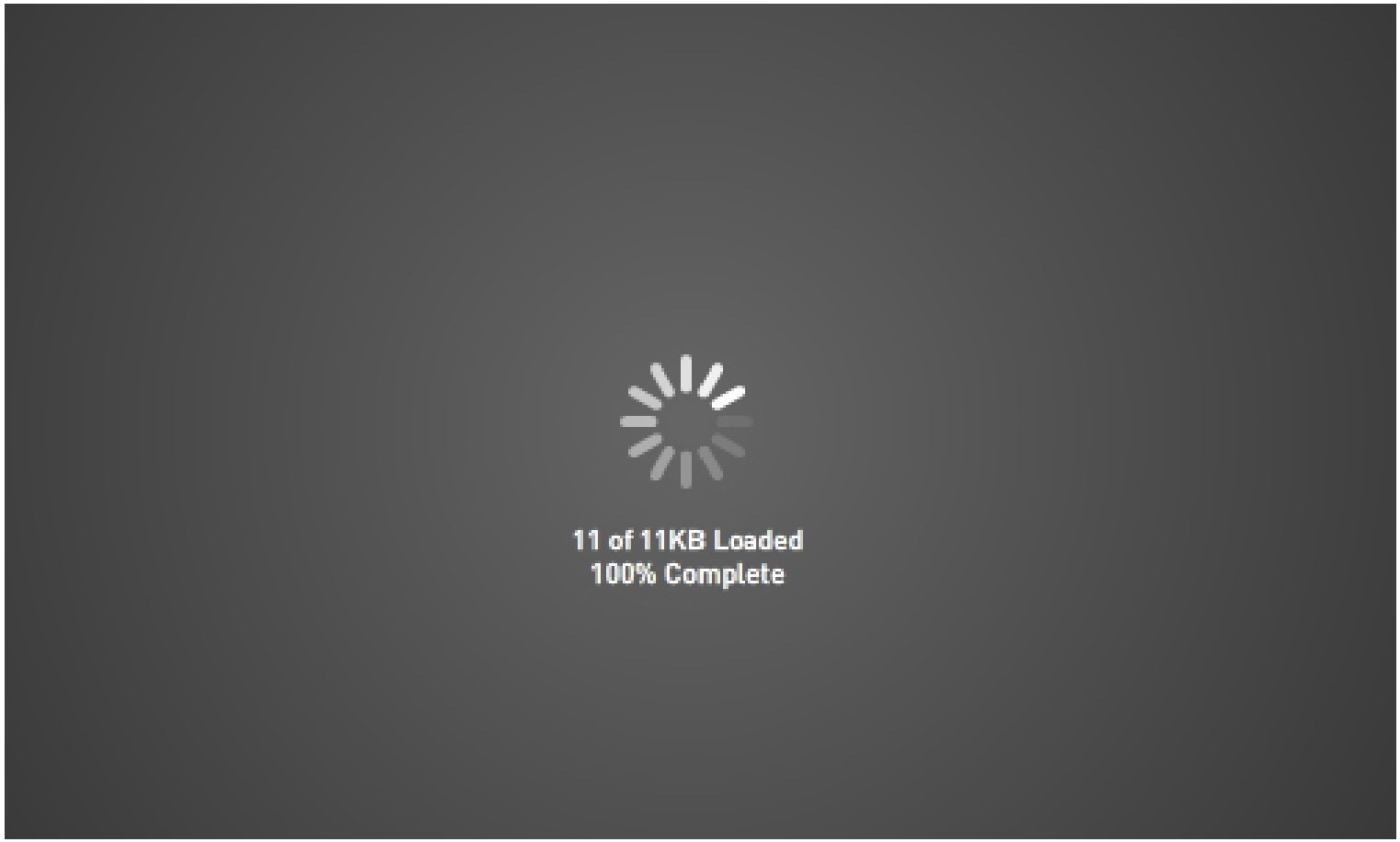
Due to an outside glitch, several British merchants on Amazon found their good were selling for just a penny last week. (Stephen Hilger/Bloomberg News)

Via: The Washington Post

Tietorakenteita tarvitaan ohjelmissa käsiteltävän tiedon tallettamiseen ja sen käsittelyn mahdollistamiseen ja helpottamiseen

- tietorakenteita on monia eivätkä ne kaikki sovi kaikkiin tilanteisiin
  $\Rightarrow$ ohjelmoijan pitää osata valita tilanteeseen sopivin
  $\Rightarrow$ vaihtoehtojen käyttäytyminen, vahvuudet ja heikkoudet on tunnettava

Modernit ohjelmointikielet tarjoavat valmiina helppokäyttöisiä tietorakenteita. Näiden ominaisuuksien sekä käyttöön vaikuttavien rajoitteiden tuntemiseksi tarvitaan perustietorakenneosaamista


11 of 11KB Loaded
100% Complete

Kuinka moni on turhautunut ohjelman tai esimerkiksi kännykän hitauteen?

- toiminnallisuus on toki ensisijaisen tärkeää kaikille ohjelmille, mutta tehokkuus ei ole merkityksetön sivuseikka
- on tärkeää huomioida ja miettiä ratkaisujaan myös ajan- ja muistinkäytön kannalta
- valmiin kirjaston käyttö näyttää usein suoraviivaisemmalta kuin onkaan

Näitä asioita käsitellään tällä kurssilla

2 Käsitteitä ja merkintöjä

Tässä luvussa esitellään kurssilla käytettävää käsitteistöä ja merkintätapoja.

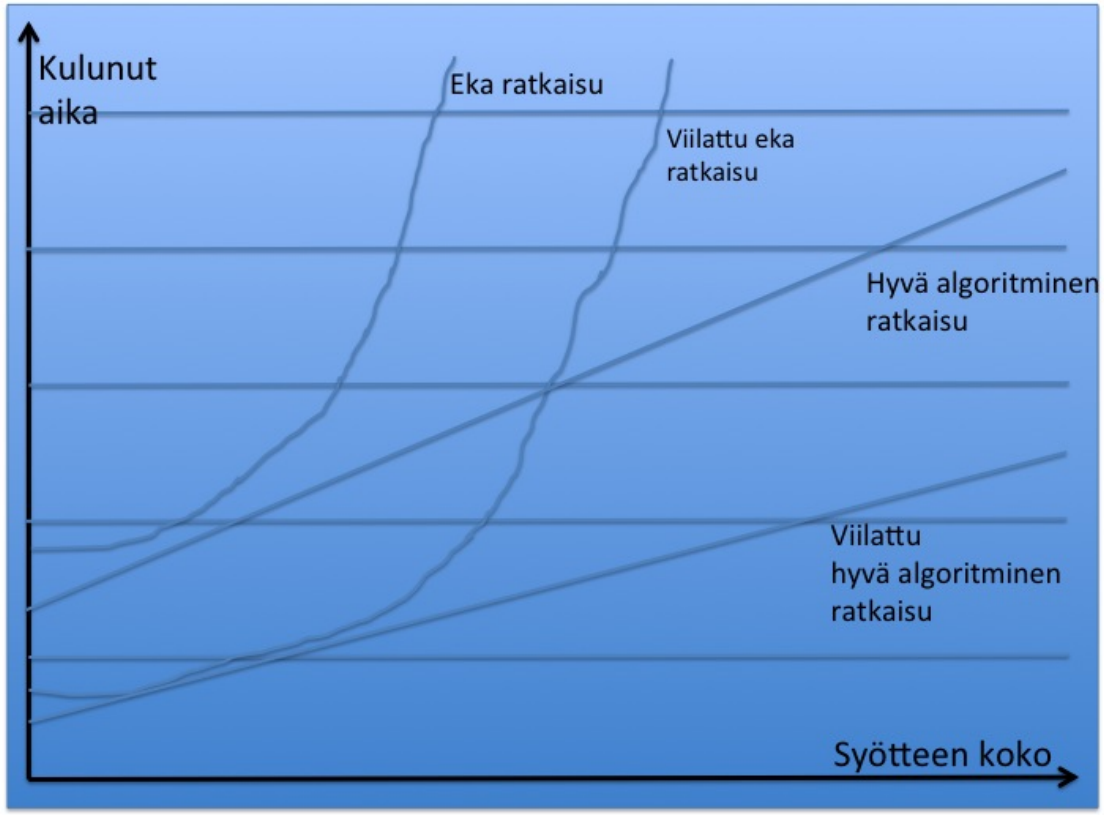
Luvussa käsitellään myös pseudokoodiesityksen ja ohjelmointikielisen koodin eroja. Esimerkkinä käytetään järjestämisalgoritmia INSERTION-SORT.

2.1 Tavoitteet

Kurssin keskeisenä tavoitteena on antaa opiskelijalle käyttöön peruskoneisto kuhunkin ohjelmointitehtävään sopivan ratkaisun valitsemiseen ja omien ratkaisujen tehokkuuden arvioimiseen karkealla tasolla.

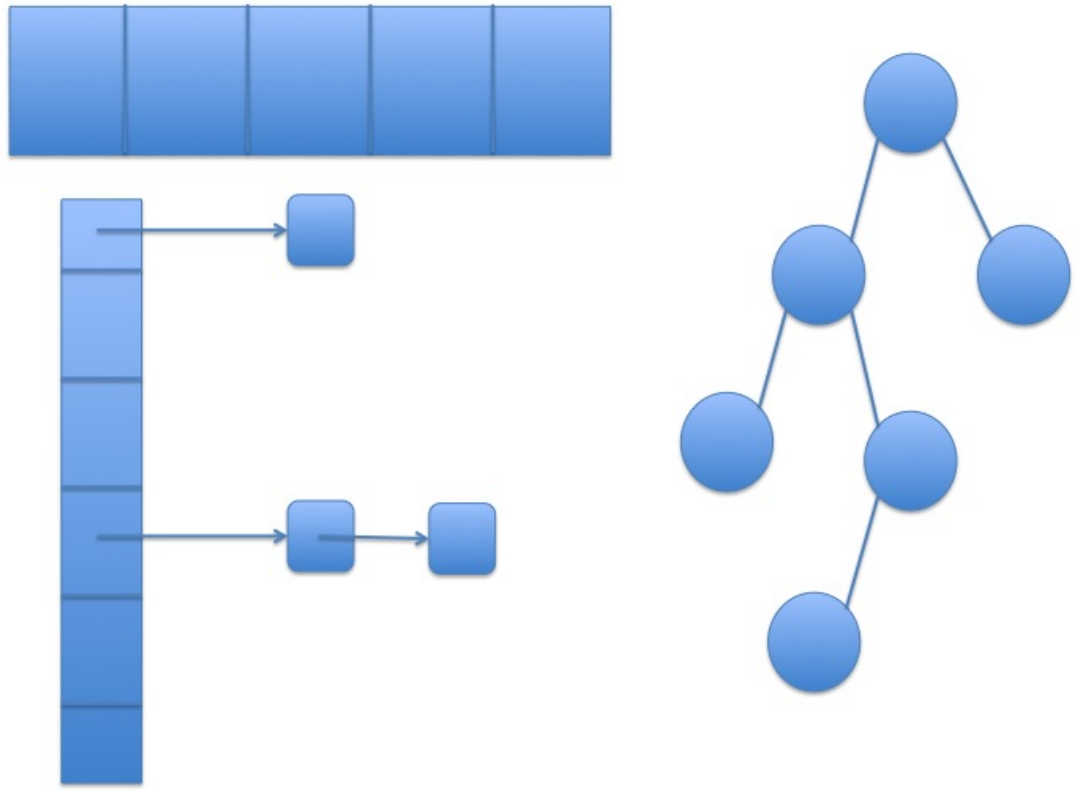
- Kurssilla keskitytään tilanteeseen sopivan tietorakenteen valintaan.
- Lisäksi käsitellään käytännön tilanteissa usein vastaan tulevia ongelmatyyppejä ja algoritmeja, joilla ne voi ratkaista.

- Kurssilla keskitytään lähinnä ns. hyviin algoritmeihin.
- Painotus on siis siinä, miten algoritmin ajankulutus kasvaa syötekoon kasvaessa, eikä niinkään yksityiskohtien optimoinnissa.



2.2 Peruskäsitteistöä

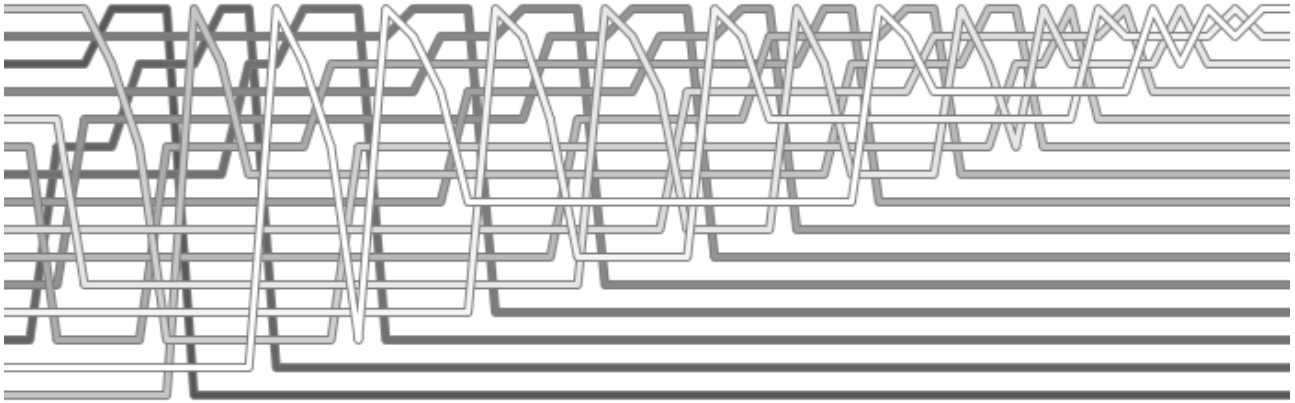
Tietorakenne



Tapa tallettaa ja järjestää tietoa:

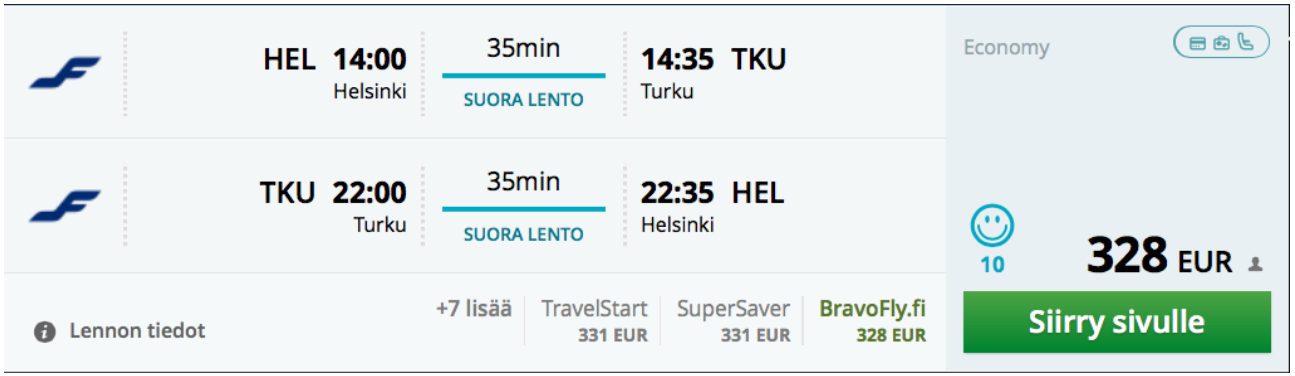
- tietoa pystytään lisäämään ja hakemaan algoritmien avulla.
- talletettua tietoa voidaan muokata
- tietorakenteita on useamman tasoisia: tietorakenne voi koostua toisista tietorakenteista

Algoritmi:



Kuva 1: kuva: Aldo Cortesi

- Joukko ohjeita tai askeleita jonkin ongelman ratkaisemiseksi
- Hyvin määritelty laskentamenetelmä, joka saa **syötteenään** alkion tai joukon alkioita ja ja tuottaa **tuloksenaan** alkion tai joukon alkioita
- Tapa tuottaa syötteestä tulos



- hyvin määritelty =
  - jokainen askel on kuvattu niin tarkasti, että lukija (ihminen tai kone) osaa suorittaa sen
  - jokainen askel on määritelty yksikäsitteisesti
  - samat vaatimukset pätevät askelten suoritusjärjestykselle
  - suorituksen tulee päättyä äärellisen askelmäärän jälkeen

Algoritmi ratkaisee jonkin hyvin määritellyn (laskenta)tehtävän.

- laskentatehtävä määrittelee, missä suhteessa tulosten tulee olla annettuihin syötteisiin
- esimerkiksi:
  - taulukon järjestäminen

    syötteet: jono lukuja $a_1, a_2, \ldots, a_n$

    tulokset: luvut $a_1, a_2, \ldots, a_n$ suuruusjärjestyksessä pienin ensin
  - lentoyhteyksien etsiminen

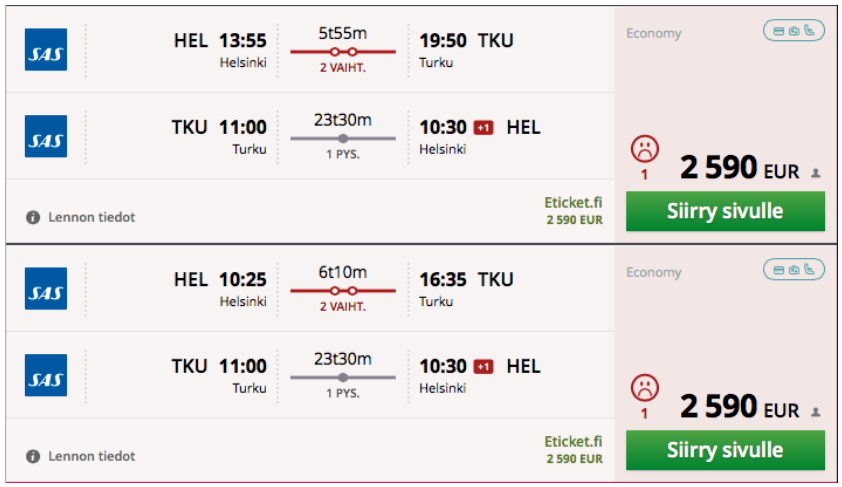
    syötteet: lentoreittiverkosto eli kaupunkeja joiden välillä lentoyhteyksiä

    tulokset: Lentojen numerot, yhteyden tiedot ja hinta.

- laskentatehtävän *esiintymä* eli *instanssi* saadaan antamalla tehtävän syötteille lailliset arvot
  - järjestämistehtävän instanssiesimerkki: 31, 41, 59, 26, 41, 58

Algoritmi on *oikea* (*correct*), jos se pysähtyy ja antaa oikeat tulokset aina kun sille on annettu laillinen syöte.

- algoritmin tai laskentatehtävän määritelmä saa kieltää osan muodollisesti mahdollisista syötteistä



algoritmi voi olla virheellinen kolmella tavalla

- antaa väärän lopputuloksen
- kaatuu kesken suorituksen
- ei koskaan lopeta

virheellinenkin algoritmi on joskus hyvin käyttökelpoinen, jos virhetiheys hallitaan!

- esim. luvun testaus alkuluvuksi

Periaatteessa mikä tahansa menetelmä kelpaa algoritmien esittämiseen, kunhan tulos on tarkka ja yksikäsitteinen.

- yleensä algoritmit toteutetaan tietokoneohjelmina tai laitteistoina
- käytännön toteutuksessa on otettava huomioon monia insinöörinäkökohtia
  - sopeuttaminen käyttötilanteeseen
  - syötteiden laillisuuden tarkistukset
  - virhetilanteiden käsittely
  - ohjelmointikielen rajoitukset
  - laitteiston ja kielen aiheuttamat nopeus- ja tarkoituksenmukaisuusnäkökohdat
  - ylläpidettävyys $\Rightarrow$ modulaarisuus jne.

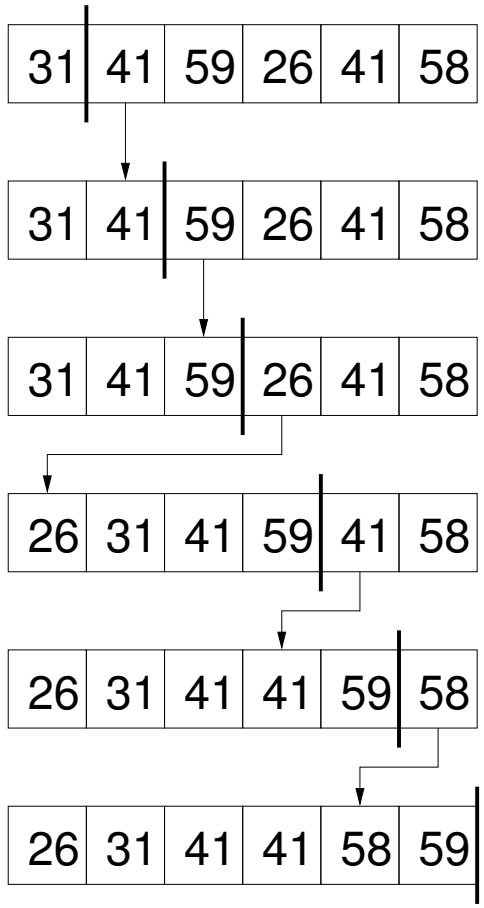
  $\Rightarrow$ algoritmin idea hukkuu helposti toteutusyksityiskohtien alle

Tällä kurssilla keskitytään algoritmien ideoihin ja algoritmit esitetään useimmiten pseudokoodina ilman laillisuustarkistuksia, virheiden käsittelyä yms.

Otetaan esimerkiksi pienten taulukoiden järjestämiseen soveltuva algoritmi INSERTION-SORT:

Kuva 2: kuva: Wikipedia

- periaate:
  - toiminnan aikana taulukon alkuosa on järjestyksessä ja loppuosa ei
  - osien raja lähtee liikkeelle paikkojen 1 ja 2 välistä ja etenee askel kerrallaan taulukon loppuun
- kullakin siirtoaskeleella etsitään taulukon alkuosasta kohta, johon loppuosan ensimmäinen alkio kuuluu
  - uudelle alkiolle raivataan tilaa siirtämällä isompia alkioita askel eteenpäin
  - lopuksi alkio sijoitetaan paikalleen ja alkuosaa kasvatetaan pykälällä



Kurssilla käytetyllä pseudokoodiesityksellä INSERTION-SORT näyttää tältä:

```
INSERTION-SORT(A)                          (syöte saadaan taulukossa A)
1  for j := 2 to A.length do               (siirretään osien välistä rajaa)
2      key := A[j]                         (otetaan alkuosan uusi alkio käsittelyyn)
3      i := j − 1
4      while i > 0 and A[i] > key do       (etsitään uudelle alkiolle oikea paikka)
5          A[i + 1] := A[i]                (raivataan uudelle alkiolle tilaa)
6          i := i − 1
7      A[i + 1] := key                     (asetetaan uusi alkio oikealle paikalleen)
```

- **for**- yms. rakenteellisten lauseiden rajaus osoitetaan sisennyksillä
- *(kommentit)* kirjoitetaan sulkuihin kursiivilla
- sijoitusoperaattorina on ":=" ("=" on yhtäsuuruuden vertaaminen)
- ▷ -merkkillä varustettu rivi antaa ohjeet vapaamuotoisesti

- tietueen (tai olion) kenttiä osoitetaan pisteen avulla
  - esim. $opiskelija.nimi$, $opiskelija.numero$
- osoittimen x osoittaman tietueen kenttiä osoitetaan merkin $\rightarrow$ avulla
  - esim. $x \rightarrow nimi$, $x \rightarrow numero$
- ellei toisin sanota, kaikki muuttujat ovat paikallisia
- taulukoilla ja / tai osoittimilla kootun kokonaisuuden nimi tarkoittaa **viitettä** ko. kokonaisuuteen
  - tuollaiset isommat tietorakenteethan aina käytännössä kannattaa välittää viiteparametreina
- yksittäisten muuttujien osalta aliohjelmat käyttävät arvonvälitystä (kuten C++-ohjelmatkin oletuksena)
- osoitin tai viite voi kohdistua myös ei minnekään: NIL

2.3 Algoritmien toteutuksesta

Käytännön toteutuksissa teoriaa tulee osata soveltaa.

Esimerkki: järjestämisalgoritmin sopeuttaminen käyttötilanteeseen.

- harvoin järjestetään pelkkiä lukuja; yleensä järjestetään tietueita, joissa on
  - *avain* (*key*)
  - *oheisdataa* (*satellite data*)
- avain määrää järjestyksen
  $\Rightarrow$ sitä käytetään vertailuissa
- oheisdataa ei käytetä vertailuissa, mutta sitä on siirreltävä samalla kuin avaintakin

Edellisessä kappaleessa esitelty INSERTION-SORTissa muuttuisi seuraavalla tavalla, jos siihen lisättäisi oheisdata:

```
1   for j := 2 to A.length do
2       temp := A[j]
3       i := j − 1
4       while i > 0 and A[i].key > temp.key do
5           A[i + 1] := A[i]
6           i := i − 1
7       A[i + 1] := temp
```

- jos oheisdataa on paljon, kannattaa järjestää taulukollinen osoittimia tietueisiin ja siirtää lopuksi tietueet suoraan paikoilleen

Jotta tulokseksi saataisi ajokelpoinen ohjelma, joka toteuttaa INSERTION-SORT:n tarvitaan vielä paljon enemmän.

- täytyy ottaa käyttöön oikea ohjelmointikieli muuttujien määrittelyineen ja aliohjelmineen
- tarvitaan pääohjelma, joka hoitaa syötteenluvun ja sen laillisuuden tutkimisen ja vastauksen tulostamisen
  - on tavallista, että pääohjelma on selvästi algoritmia pidempi

Ohjelmointikieli määrää usein myös muita asioita, esim:

- Indeksointi alkaa 0:sta (pseudokoodissa usein 1:stä)
- Käytetäänkö edes indeksointia (tai taulukoita, tai...)
- (C++) Onko data oikeasti tietorakenteen sisässä, vai osoittimen päässä (jolloin dataa ei tarvitse siirtää ja sen jakaminen on helpompaa)
- Jos dataan viitataan epäsuorasti muualta, tapahtuuko se
  - Osoittimella
  - Älyosoittimella (esim. `shared_ptr`)
  - Iteraattorilla (jos data tietorakenteessa)
  - Indeksillä (jos data vektorissa tms.)
  - Hakuavaimella (jos data tietorakenteessa, josta haku nopeaa)
- Toteutetaanko rekursio iteroinnilla vai ei (riippuu myös ongelmasta)
- Ovatko algoritmin "parametrit"oikeasti parametreja, vai vain muuttujia tms.

Otetaan esimerkiksi edellä kuvatun ohjelman toteutus C++:lla:

```
#include <iostream>
#include <vector>
typedef std::vector<int> Taulukko;

void insertionSort( Taulukko & A ) {
  int key = 0; int i = 0;
  for( Taulukko::size_type j = 1; j < A.size(); ++j ) {
    key = A.at(j); i = j-1;
    while( i >= 0 && A.at(i) > key ) {
      A.at(i+1) = A.at(i); --i;
    }
    A.at(i+1) = key;
  }
}

int main() {
  unsigned int i;
  // haetaan järjestettävien määrä
  std::cout << "Anna taulukon koko 0...: "; std::cin >> i;
```

```
  Taulukko A(i); // luodaan taulukko
  // luetaan järjestettävät
  for( i = 0; i < A.size(); ++i ) {
    std::cout << "Anna A[" << i+1 << "]: ";
    std::cin >> A.at(i);
  }
  insertionSort( A );   // järjestetään

  // tulostetaan siististi
  for( i = 0; i < A.size(); ++i ) {
    if( i % 5 == 0 ) {
      std::cout << std::endl;
    }
    else {
      std::cout << " ";
    }
    std::cout << A.at(i);
  }
  std::cout << std::endl;
}
```

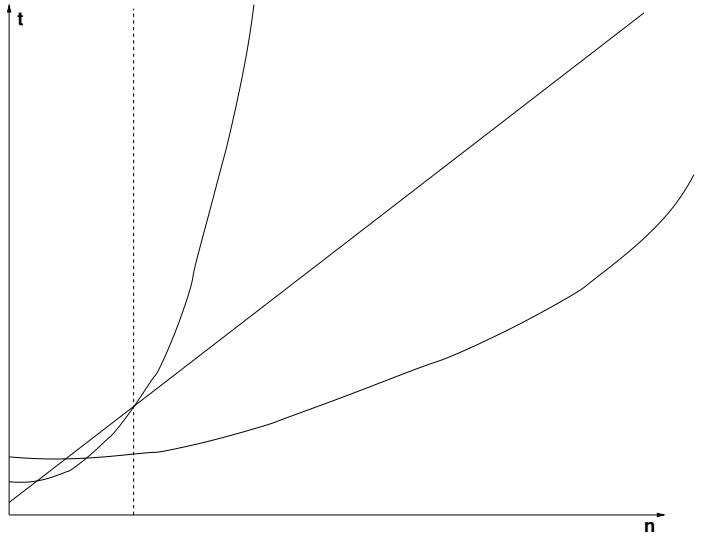
Ohjelmakoodi on huomattavasti pseudokoodia pidempi ja algoritmille ominaisten asioiden hahmottaminen on siitä paljon vaikeampaa.

Tämä kurssi keskittyy algoritmien ja tietorakenteiden periaatteisiin, joten ohjelmakoodi ei palvele tarkoituksiamme.

$\Rightarrow$ Tästä eteenpäin toteutuksia ohjelmointikielillä ei juurikaan esitetä.

4 Tehokkuus ja algoritmien suunnittelu

Tässä luvussa pohditaan tehokkuuden käsitettä ja esitellään kurssilla käytetty kertaluokkanotaatio, jolla kuvataan algoritmin *asymptoottista* käyttäytymistä eli tapaa, jolla algoritmin resurssien kulutus muuttuu syötekoon kasvaessa.



4.1 Kertaluokat

Algoritmin analysoinnilla tarkoitetaan sen kuluttamien resurssien määrän arvioimista

Tyypillisesti analysoidaan syötekoon kasvun vaikutusta algoritmin resurssien kulutukseen

Useimmiten meitä kiinnostaa algoritmin ajankäytön kasvu syötteen koon kasvaessa

- Voimme siis tarkastella ajankäyttöä irrallaan toteutusympäristöstä
- Itse asiassa voimme kuvata periaatteessa minkä tahansa peräkkäisiä operaatioita sisältävän toiminnan ajankulutusta

- **Algoritmin ajankäyttö**:
  Algoritmin suorittamien "askelten" suorituskertojen määrä
- **Askel**:
  syötekoosta riippumattoman operaation viemä aika.
- Emme välitä siitä, kuinka monta kertaa jokin operaatio suoritetaan kunhan se tehdään vain vakiomäärä kertoja.
- Tutkimme kuinka monta kertaa algoritmin suorituksen aikana kukin rivi suoritetaan ja laskemme nämä määrät yhteen.

- Yksinkertaistamme vielä tulosta poistamalla mahdolliset vakiokertoimet ja alemman asteen termit.

  $\Rightarrow$ Näin voidaan tehdä, koska syötekoon kasvaessa riittävästi alemman asteen termit käyvät merkityksettömiksi korkeimman asteen termin rinnalla.

  $\Rightarrow$ Menetelmä ei luonnollisestikaan anna luotettavia tuloksia pienillä syöteaineistoilla, mutta niillä ohjelmat ovat tyypillisesti riittävän tehokkaita joka tapauksessa.
- Kutsumme näin saatua tulosta algoritmin ajan kulutuksen kertaluokaksi, jota merkitään kreikkalaisella kirjaimella Θ (äännetään "theeta").

$$f(n) = 23n^2 + 2n + 15 \Rightarrow f \in \Theta(n^2)$$

$$f(n) = \tfrac{1}{2} n \lg n + n \Rightarrow f \in \Theta(n \lg n)$$

Esimerkki 1: taulukon alkioiden summaus

```
1   for i := 1 to A.length do
2         summa := summa + A[i]
```

- jos taulukon A pituus (syötekoko) on n, rivi 1 suoritetaan n + 1 kertaa
- rivi 2 suoritetaan n kertaa
- ajankulutus kasvaa siis n:n kasvaessa seuraavalla tavalla:

n	aika = $2n$ + 1
1	3
10	21
100	201
1000	2001
10000	20001

$\Rightarrow$ n:n arvo hallitsee ajankulutusta

- suoritamme edellä sovitut yksinkertaistukset: poistamme vakiokertoimen ja alemman asteen termin:

  $f(n) = 2n + 1 \Rightarrow n$

  $\Rightarrow$ saamme tulokseksi $\Theta(n)$

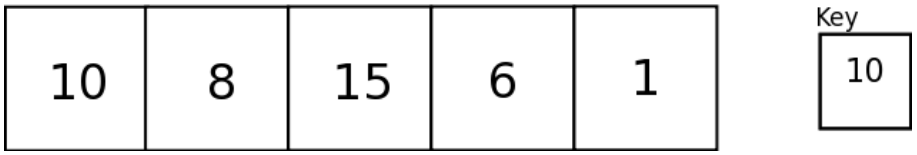
  $\Rightarrow$ kulutettu aika riippuu *lineaarisesti* syötteen koosta

Esimerkki 2: alkion etsintä järjestämättömästä taulukosta

```
1   for i := 1 to A.length do
2       if A[i] = key then
3           return i
```

- tässä tapauksessa suoritusaika riippuu syöteaineiston koon lisäksi sen koostumuksesta eli siitä, mistä kohtaa taulukkoa haluttu alkio löytyy
- On tutkittava erikseen:
  paras,
  huonoin ja
  keskimääräinen tapaus

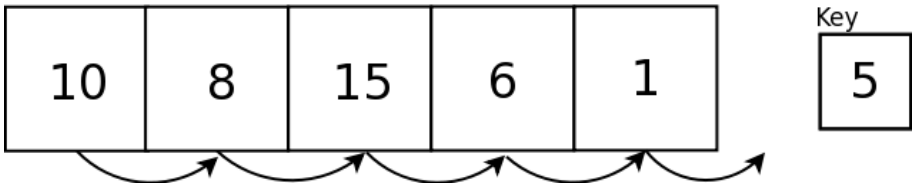
- paras tapaus:



Kuva 6: Etsintä: paras tapaus, löytyy ensimmäisestä alkiosta

$\Rightarrow$ alkio löytyy *vakioajassa* eli ajankulutus on $\Theta(1)$

– huonoin tapaus



Kuva 7: Etsintä: huonoin tapaus, löytyy viimeisestä tai ei ollenkaan

rivi 1 suoritetaan n + 1 kertaa ja rivi 2 n kertaa
$\Rightarrow$ suoritusaika on *lineaarinen* eli $\Theta(n)$.

- keskimääräinen tapaus:
  täytyy tehdä jonkinlainen oletus tyypillisestä eli keskimääräisestä aineistosta:
  * alkio on taulukossa todennäköisyydellä p ($0 \leq p \leq 1$)
  * ensimmäinen haettu alkio löytyy taulukon jokaisesta kohdasta samalla todennäköisyydellä
- voimme laskea suoraan todennäköisyyslaskennan avulla, kuinka monta vertailua keskimäärin joudutaan tekemään

- todennäköisyys sille, että alkio ei löydy taulukosta on 1 - p $\Rightarrow$ joudutaan tekemään n vertailua (huonoin tapaus)
- todennäköisyys sille, että alkio löytyy kohdasta i, on p/n $\Rightarrow$ joudutaan tekemään i vertailua
- odotusarvoinen tarvittavien vertailujen määrä saadaan siis seuraavasti:

$$[1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} \cdots + n \cdot \frac{p}{n}] + n \cdot (1 - p)$$

- oletamme, että alkio varmasti löytyy taulukosta eli p = 1, saamme tulokseksi (n+1)/2 eli $\Theta(n)$
  $\Rightarrow$ koska myös tapaus, jossa alkio ei löydy taulukosta, on ajankäytöltään lineaarinen, voimme olla varsin luottavaisia sen suhteen, että keskimääräinen ajankäyttö on kertaluokassa $\Theta(n)$
- kannattaa kuitenkin muistaa, että läheskään aina kaikki syötteet eivät ole yhtä todennäköisiä
  $\Rightarrow$ jokaista tapausta on syytä tutkia erikseen

Esimerkki 3: kahden taulukon yhteisen alkion etsintä

```
1   for i := 1 to A.length do
2       for j := 1 to B.length do
3           if A[i] = B[j] then
4               return A[i]
```

- rivi 1 suoritetaan 1 .. (n + 1) kertaa
- rivi 2 suoritetaan 1 .. ($n \cdot (n$ + 1)) kertaa
- rivi 3 suoritetaan 1 .. ($n \cdot n$) kertaa
- rivi 4 suoritetaan korkeintaan kerran

- nopeimmillaan algoritmi on siis silloin kun molempien taulukoiden ensimmäinen alkio on sama

  $\Rightarrow$ parhaan tapauksen ajoaika on $\Theta(1)$
- pahimmassa tapauksessa taulukoissa ei ole ainuttakaan yhteistä alkiota tai ainoastaan viimeiset alkiot ovat samat

  $\Rightarrow$ tällöin suoritusajaksi tulee *neliöllinen* eli $2n^2 + 2n + 1 = \Theta(n^2)$
- keskimäärin voidaan olettaa, että molempia taulukoita joudutaan käymään läpi noin puoleen väliin

  $\Rightarrow$ tällöin suoritusajaksi tulee $\Theta(n^2)$
  (tai $\Theta(nm)$ mikäli taulukot ovat eri mittaisia)

Palataan INSERTION-SORTiin. Sen ajankäyttö:

```
INSERTION-SORT( A )                          (syöte saadaan taulukossa A)
1   for j := 2 to A.length do                (siirretään osien välistä rajaa)
2       key := A[ j ]                        (otetaan alkuosan uusi alkio käsittelyyn)
3       i := j − 1
4       while i > 0 and A[ i ] > key do      (etsitään uudelle alkiolle oikea paikka)
5           A[ i + 1 ] := A[ i ]             (raivataan uudelle alkiolle tilaa)
6           i := i − 1
7       A[ i + 1 ] := key                    (asetetaan uusi alki o oikealle paikalleen)
```

- rivi 1 suoritetaan n kertaa
- rivit 2 ja 3 suoritetaan n - 1 kertaa
- rivi 4 suoritetaan vähintään n - 1, enintään (2 + 3 + 4 + $\cdots$ + n - 2) kertaa
- rivit 5 ja 6 suoritetaan vähintään 0, enintään (1 + 2 + 3 + 4 + $\cdots$ + n - 3) kertaa

- parhaassa tapauksessa, kun taulukko on valmiiksi järjestyksessä, koko algoritmi siis kuluttaa vähintään $\Theta(n)$ aikaa
- huonoimmassa tapauksessa, kun taulukko on käänteisessä järjestyksessä, aikaa taas kuluu $\Theta(n^2)$
- keskimääräisen tapauksen selvittäminen on jälleen vaikeampaa:
  * oletamme, että satunnaisessa järjestyksessä olevassa taulukossa olevista elementtipareista puolet ovat keskenään epäjärjestyksessä.

    $\Rightarrow$ vertailuja joudutaan tekemään puolet vähemmän kuin pahimmassa tapauksessa, jossa kaikki elementtiparit ovat keskenään väärässä järjestyksessä

    $\Rightarrow$ keskimääräinen ajankulutus on pahimman tapauksen ajankäyttö jaettuna kahdella: ((n - 1)n)/ 4 = $\Theta(n^2)$

3.3 Suunnitteluperiaate: Hajota ja hallitse

Suunnitteluperiaate *hajota ja hallitse* on todennäköisesti periaatteista kuuluisin.

Se toimii useiden tunnettujen tehokkaiden algoritmien periaatteena

Perusidea:

- ongelma jaetaan alkuperäisen kaltaisiksi, mutta pienemmiksi osaongelmiksi.
- pienet osaongelmat ratkaistaan suoraviivaisesti
- suuremmat osaongelmat jaetaan edelleen pienempiin osiin
- lopuksi osaongelmien ratkaisut kootaan alkuperäisen ongelman ratkaisuksi

Hajota ja hallitse on usein rekursiivinen rakenteeltaan: algoritmi kutsuu itseään osaongelmille

Pienten osaongelmien ratkaisemiseksi voidaan myös hyödyntää toista algoritmia

3.4 QUICKSORT

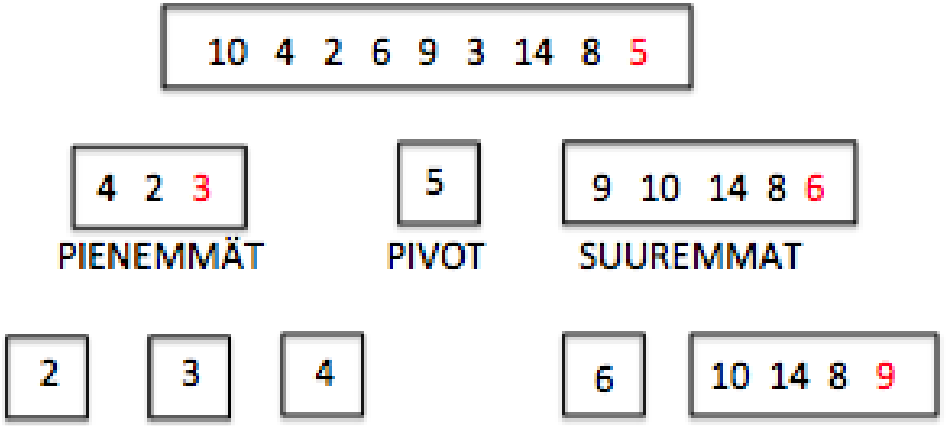
Ongelman jakaminen pienemmiksi osaongelmiksi

- Valitaan jokin taulukon alkioista jakoalkioksi eli *pivot-alkioksi*.
- Muutetaan taulukon alkioiden järjestystä siten, että kaikki jakoalkiota pienemmät tai yhtäsuuret alkiot ovat taulukossa ennen jakoalkiota ja suuremmat alkiot sijaitsevat jakoalkion jälkeen.
- Jatketaan alku ja loppuosien jakamista pienemmiksi, kunnes ollaan päästy 0:n tai 1:n kokoisiin osataulukoihin.

QUICKSORT-algoritmi

QUICKSORT($A, left, right$)

```
1   if left < right then                  (triviaalitapaukselle ei tehdä mitään)
2       p := PARTITION( A, left, right )  (muuten jaetaan alkuosaan ja loppuosaan)
3       QUICKSORT( A, left, p − 1 )       (järjestetään jakoalkiota pienemmät)
4       QUICKSORT( A, p + 1, right )      (järjestetään jakoalkiota suuremmat)
```



Kuva 3: Jako pienempiin ja suurempiin

Pienet osaongelmat:

- 0:n tai 1:n kokoiset osataulukot ovat valmiiksi järjestyksessä.

Järjestyksessä olevien osataulukoiden yhdistäminen:

- Kun alkuosa ja loppuosa on järjestetty on koko (osa)taulukko automaattisesti järjestyksessä.
  - kaikki alkuosan alkiothan ovat loppuosan alkioita pienempiä, kuten pitääkin

Ositus- eli *partitiointialgoritmi* jakaa taulukon paikallaan vaaditulla tavalla.

PARTITION($A, left, right$)

```
1  p := A[ right ]                    (otetaan pivotiksi viimeinen alkio)
2  i := left − 1                      (merkitään i:llä pienten puolen loppua)
3  for j := left to right − 1 do      (käydään läpi toiseksi viimeiseen alkioon asti)
4      if A[ j ] ≤ p                  (jos A[ j] kuuluu pienten puolelle...)
5          i := i + 1                 (... kasvatetaan pienten puolta...)
6          exchange A[ i ] ↔ A[ j ]   (... ja siirretään A[j] sinne)
7  exchange A[ i + 1 ] ↔ A[ right ]   (sijoitetaan pivot pienten ja isojen puolten väliin)
8  return i + 1                       (palautetaan pivot-alkion sijainti)
```

3.5 MERGESORT

Erinomainen esimerkki hajota ja hallitse -periaatteesta on MERGE-SORT järjestämisalgoritmi:

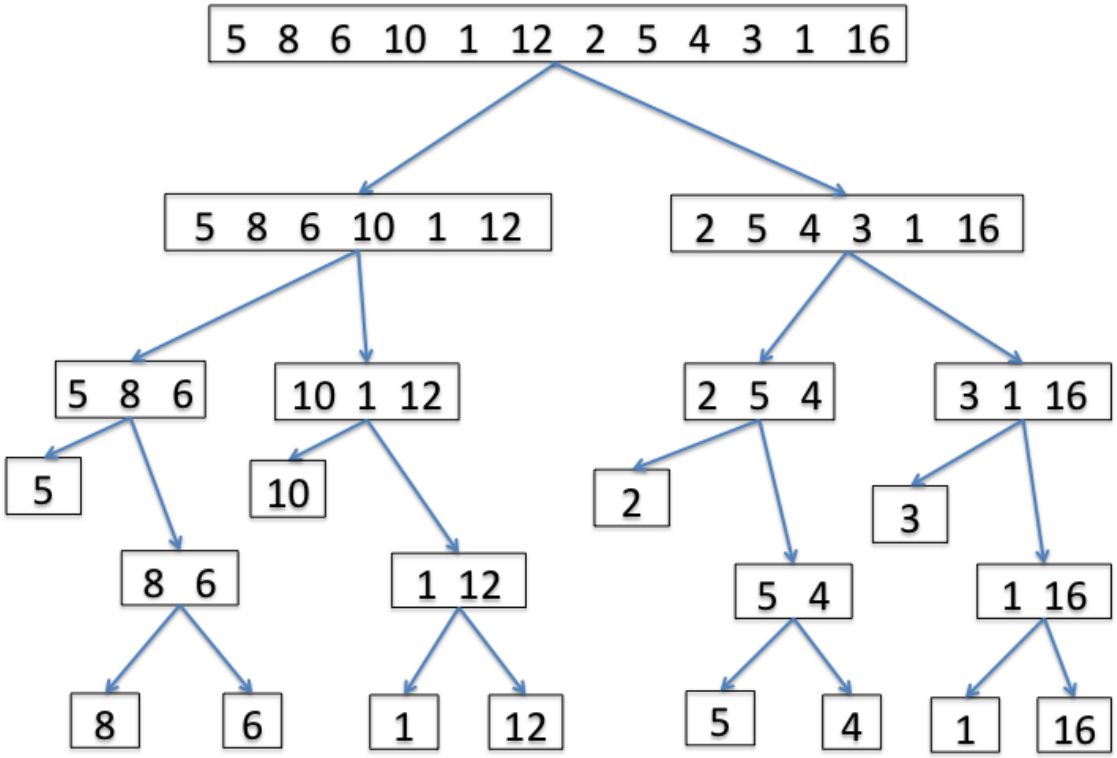
1. Taulukko jaetaan kahteen osaan $A[1..\lfloor n/2 \rfloor]$ ja $A[\lfloor n/2 \rfloor + 1..n]$.
2. Järjestetään puolikkaat rekursiivisesti
3. Limitetään järjestetyt puolikkaat järjestetyksi taulukoksi

- MERGE-SORT -ALGORITMI

MERGE-SORT($A, left, right$)

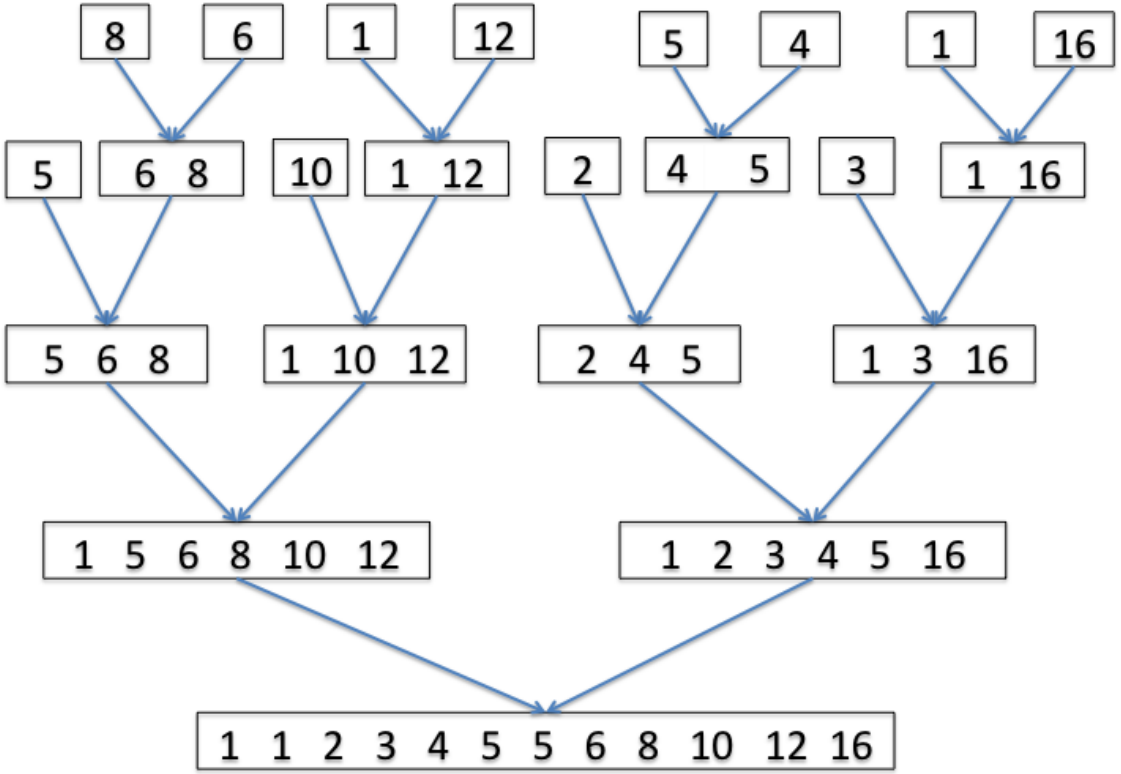
```
1   if left < right then                    (jos taulukossa on alkioita...)
2       mid := ⌊( left + right )/2⌋         (... jaetaan se kahtia)
3       MERGE-SORT( A, left, mid )          (järjestetään alkuosa...)
4       MERGE-SORT( A, mid + 1, right )     (... ja loppuosa)
5       MERGE( A, left, mid, right )        (limitetään osat siten, että järjestys säilyy)
```

Hajota:



Kuva 4: Jako osaongelmiin

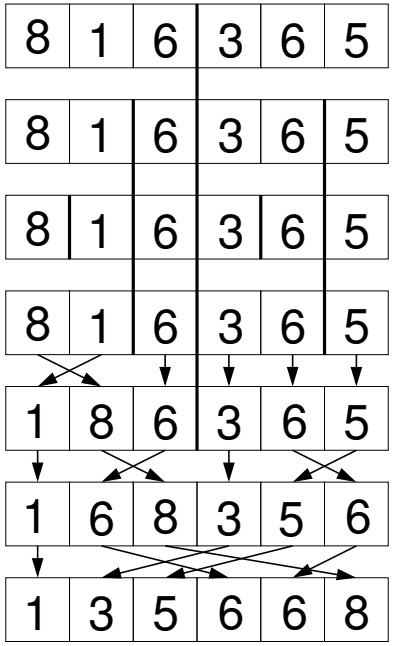
Hallitse:



Kuva 5: Osaongelmien ratkaisujen limitys

Eli:

- jaetaan järjestettävä taulukko kahteen osaan
- jatketaan edelleen osien jakamista kahtia, kunnes osataulukot ovat 0 tai 1 alkion kokoisia
- 0 ja 1 kokoiset taulukot ovat valmiiksi järjestyksessä eivätkä vaadi mitään toimenpiteitä
- lopuksi yhdistetään järjestyksessä olevat osataulukot limittämällä
- huomaa, että rekursiivinen algoritmi ei toimi kuvan tavalla molemmat puolet rinnakkain



– limityksen suorittava MERGE-algoritmi:

```
MERGE( A, left, mid, right )
1   for i := left to right do          (käydään koko alue läpi...)
2       B[ i ] := A[ i ]               (... ja kopioidaan se aputaulukkoon)
3   i := left                          (asetetaan i osoittamaan valmiin osan loppua)
4   j := left; k := mid + 1            (asetetaan j ja k osoittamaan osien alkuja)
5   while j ≤ mid and k ≤ right do     (käydään läpi, kunnes jompikumpi osa loppuu)
6       if B[ j ] ≤ B[ k ] then        (jos alkuosan ensimmäinen alkio on pienempi...)
7           A[ i ] := B[ j ]           (... sijoitetaan se tulostaulukkoon...)
8           j := j + 1                 (... ja siirretään alkuosan alkukohtaa)
9       else                           (muuten...)
10          A[ i ] := B[ k ]           (... sijoitetaan loppuosan alkio tulostaulukkoon...)
11          k := k + 1                 (... ja siirretään loppuosan alkukohtaa)
12      i := i + 1                     (siirretään myös valmiin osan alkukohtaa)
13  if j > mid then
14      k := 0
15  else
16      k := mid − right
17  for j := i to right do  (siirretään loput alkiot valmiin osan loppuun)
18      A[ j ] := B[ j + k ]
```

MERGE limittää taulukot käyttäen “pala kerrallaan” -menetelmää.

Tuottaako hajota ja hallitse tehokkaamman ratkaisun kuin pala kerrallaan?

Ei aina, mutta tarkastellaksemme tilannetta tarkemmin, meidän täytyy tutustua algoritmin analyysiin

5 Kertaluokkamerkinnät

Tässä luvussa käsitellään asymptoottisessa analyysissa käytettyjä matemaattisia merkintätapoja

Määritellään tarkemmin Θ, sekä kaksi muuta saman sukuista merkintää O ja Ω.

5.1 Asymptoottinen aika-analyysi

Edellisessä luvussa yksinkertaistimme suoritusajan lauseketta melkoisesti:

- jätimme jäljelle ainoastaan eniten merkitsevän termin
- poistimme sen edestä vakiokertoimen

$\Rightarrow$ kuvastavat algoritmin käyttäytymistä, kun syötekoko kasvaa kohti ääretöntä

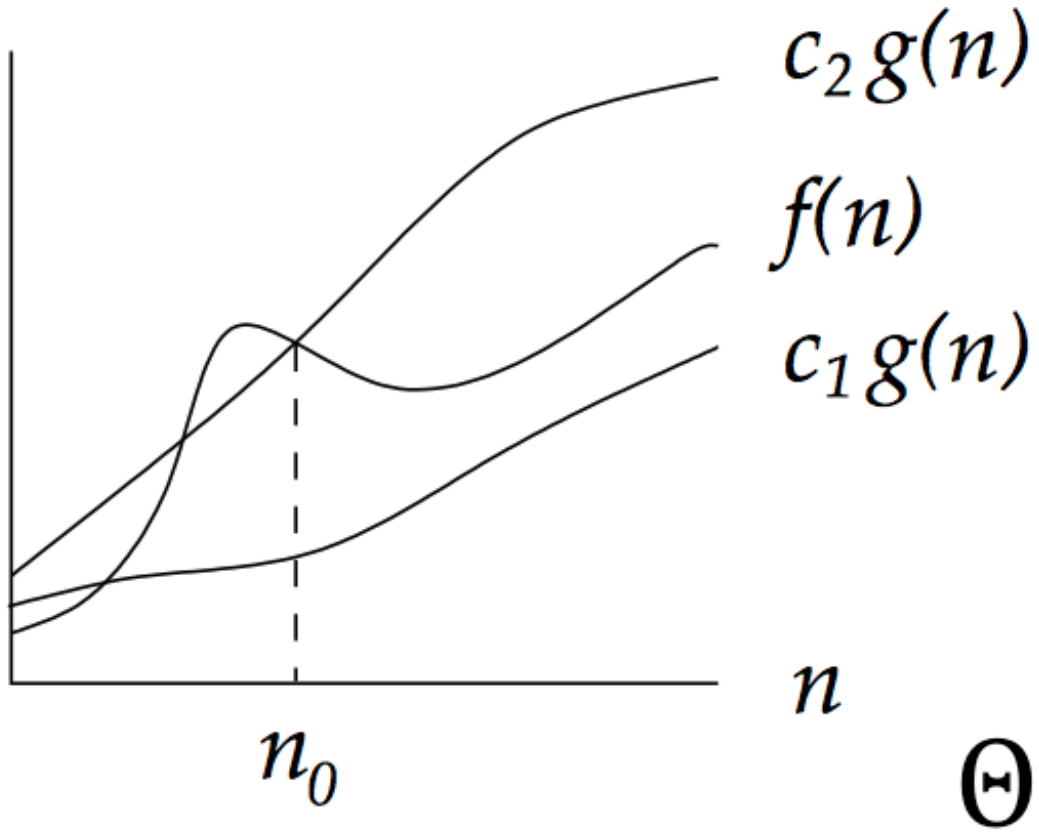
- siis kuvaavat *asymptoottista* suorituskykyä

$\Rightarrow$ antavat käyttökelpoista tietoa **vain jotain rajaa suuremmilla syötteillä**

- todettiin, että usein raja on varsin alhaalla

  $\Rightarrow$ Θ- yms. merkintöjen mukaan nopein on myös käytännössä nopein, paitsi aivan pienillä syötteillä

Θ-merkintä



Kuva 8: Θ-merkintä

eli matemaattisesti

- olkoon $g(n)$ funktio luvuilta luvuille

  $\Theta(g(n))$ *on niiden funktioiden* $f(n)$ *joukko, joille on olemassa positiiviset vakiot* c_1, c_2 *ja* n_0 *siten, että aina kun* $n \geq n_0$, *niin*

$$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

  - $\Theta(g(n))$ on *joukko* funktioita
    $\Rightarrow$ tulisi kirjoittaa esim. $f(n) \in \Theta(g(n)) \Rightarrow$ ohjelmistotieteessä vakiintunut käytäntö kuitenkin on käyttää = -merkintää

Funktion $f(n)$ kuuluvuuden kertaluokkaan $\Theta(g(n))$, voi siis todistaa etsimällä jotkin arvot vakioille c_1, c_2 ja n_0 ja osoittamalla, että funktion arvo pysyy n:n arvoilla n_0:sta alkaen arvojen $c_1 g(n)$ ja $c_2 g(n)$ välillä (eli suurempana tai yhtäsuurena kuin $c_1 g(n)$ ja pienempänä tai yhtäsuurena, kuin $c_2 g(n)$).

Esimerkki: $3n^2 + 5n - 20 = \Theta(n^2)$

- valitaan $c_1 = 3$, $c_2 = 4$ ja $n_0 = 4$
- $0 \leq 3n^2 \leq 3n^2 + 5n - 20 \leq 4n^2$ kun $n \geq 4$, koska silloin $0 \leq 5n - 20 \leq n^2$
- yhtä hyvin olisi voitu valita $c_1 = 2$, $c_2 = 6$ ja $n_0 = 7$ tai $c_1 = 0{,}000\,1$, $c_2 = 1\,000$ ja $n_0 = 1\,000$
- tärkeää on vain, että voidaan valita *jotkut* positiiviset, ehdot täyttävät c_1, c_2 ja n_0

Tärkeä tulos: jos $a_k > 0$, niin

$$a_k n^k + a_{k-1} n^{k-1} + \cdots + a_2 n^2 + a_1 n + a_0 = \Theta(n^k)$$

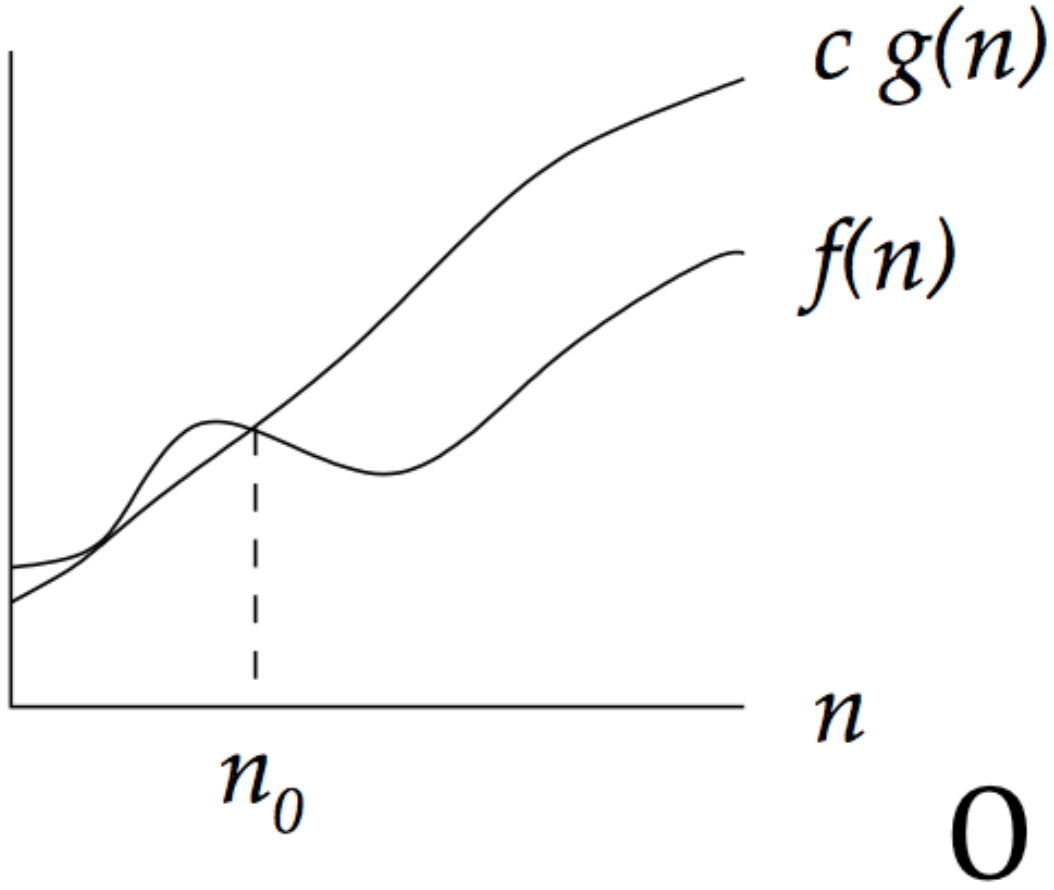
- toisin sanoen, jos polynomin eniten merkitsevän termin kerroin on positiivinen, Θ-merkintä sallii kaikkien muiden termien sekä e.m. kertoimen abstrahoinnin pois

Vakiofunktiolle pätee $c = \Theta(n^0) = \Theta(1)$

- $\Theta(1)$ ei kerro, minkä muuttujan suhteen funktioita tarkastellaan

  $\Rightarrow$ sitä saa käyttää vain kun muuttuja on asiayhteyden vuoksi selvä $\Rightarrow$ yleensä algoritmien tapauksessa on

O-merkintä



Kuva 9: O-merkintä

O-merkintä on muuten samanlainen kuin Θ-merkintä, mutta se rajaa funktion ainoastaan ylhäältä.

$\Rightarrow$ *asymptoottinen yläraja*

Määritelmä:

> *$O(g(n))$ on niiden funktioiden $f(n)$ joukko, joille on olemassa positiiviset vakiot c ja n_0 siten, että aina kun $n \geq n_0$, niin*
>
> $$0 \leq f(n) \leq c \cdot g(n)$$

- pätee: jos $f(n) = \Theta(g(n))$, niin $f(n) = O(g(n))$
- päinvastainen ei aina päde: $n^2 = O(n^3)$, mutta $n^2 \neq \Theta(n^3)$
- tärkeä tulos: jos $k \leq m$, niin $n^k = O(n^m)$
- jos **hitaimman** tapauksen suoritusaika on $O(g(n))$, niin **jokaisen** tapauksen suoritusaika on $O(g(n))$

Usein algoritmin hitaimman (ja samalla jokaisen) tapauksen suoritusajalle saadaan pelkällä vilkaisulla jokin O-merkinnällä ilmoitettava yläraja.

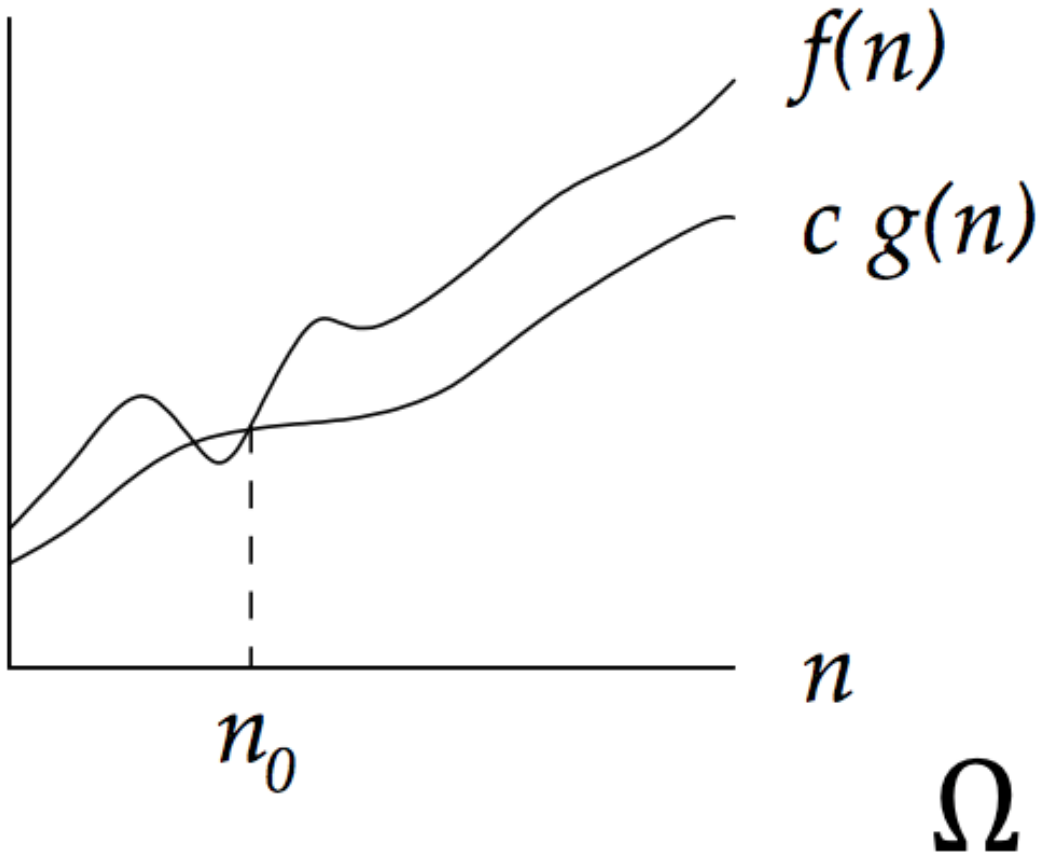
Usein vain yläraja onkin kiinnostava
$\Rightarrow$ O-merkinnällä on suuri käytännön merkitys

Esimerkki: INSERTION-SORT

rivi	kerta-aika
for j := 2 **to** $A.length$ **do**	$O(n)$
key := $A[j]$	$\cdot \quad O(1)$
i := $j-1$	$\cdot \quad O(1)$
while $i > 0$ and $A[i] > key$ **do**	$\cdot \quad O(n)$
$A[i+1]$:= $A[i]$	$\cdot \quad \cdot \quad O(1)$
i := $i-1$	$\cdot \quad \cdot \quad O(1)$
$A[i+1]$:= key	$\cdot \quad O(1)$

Jolloin pahimmalle tapaukselle saadaan suoritusaika
$O(n) \cdot O(n) \cdot O(1) = O(n^2)$

Ω-merkintä (äännetään “iso oomega”)



Kuva 10: Ω-merkintä

Ω-merkintä on muuten täysin samanlainen kuin Θ-merkintä, mutta se rajaa funktion vain alhaalta.

$\Rightarrow$ *asymptoottinen alaraja*

määritelmä:

> $\Omega(g(n))$ *on niiden funktioiden* $f(n)$ *joukko, joille on olemassa positiiviset vakiot* c *ja* n_0 *siten, että aina kun* $n \geq n_0$*, niin*
>
> $$0 \leq c \cdot g(n) \leq f(n)$$

- määritelmistä seuraa tärkeä tulos:
  $f(n) = \Theta(g(n))$ *jos ja vain jos* $f(n) = O(g(n))$ *ja* $f(n) = \Omega(g(n))$.
- jos **nopeimman** tapauksen suoritusaika on $\Omega(g(n))$, niin **jokaisen** tapauksen suoritusaika on $\Omega(g(n))$

Käytännön hyötyä Ω-merkinnästä on lähinnä tilanteissa, joissa jonkin ratkaisuvaihtoehdon parhaan tapauksenkin tehokkuus on epätyydyttävä, jolloin ratkaisu voidaan hylätä välittömästi.

Merkintöjen keskinäiset suhteet

$f(n) = \Omega(g(n))$ ja $f(n) = O(g(n)) \iff f(n) = \Theta(g(n))$

Kertaluokkamerkinnöillä on samankaltaisia ominaisuuksia kuin lukujen vertailuilla:

$$\begin{aligned} f(n) &= O(g(n)) \quad a \leq b \\ f(n) &= \Theta(g(n)) \quad a = b \\ f(n) &= \Omega(g(n)) \quad a \geq b \end{aligned}$$

Eli, jos $f(n)$:n korkeimman asteen termi, josta on poistettu vakiokerroin $\leq g(n)$: n vastaava, $f(n) = O(g(n))$ jne.

Yksi merkittävä ero: reaaliluvuille pätee aina tasan yksi kaavoista $a < b$, $a = b$ ja $a > b$, mutta vastaava ei päde kertaluokkamerkinnöille.

$\Rightarrow$ Kaikkia funktioita ei pysty mielekkäällä tavalla vertaamaan toisiinsa kertaluokkamerkintöjen avulla (esim. n ja $n^{1+\sin n}$).

Hieman yksinkertaisten:

- Jos algoritmi on $\Omega(g(n))$, sen resurssien kulutus on ainakin kertaluokassa $g(n)$.
  - vrt. kirja maksaa ainakin noin kympin.
- Jos algoritmi on $O(g(n))$, sen resurssien kulutus on korkeintaan kertaluokassa $g(n)$.
  - vrt. kirja maksaa korkeintaan noin kympin.
- Jos algoritmi on $\Theta(g(n))$, sen resurssien kulutus on aina kertaluokassa $g(n)$.
  - vrt. kirja maksaa suunnilleen kympin.

Kaikkien algoritmien kaikkien tapausten suoritusajalle ei välttämättä voida antaa mitään ratkaisua Θ-notaatiolla.

Esimerkkinä Insertion-Sort:

- paras tapaus on $\Omega(n)$, mutta ei $\Omega(n^2)$
- pahin tapaus on $O(n^2)$, mutta ei $O(n)$
  $\Rightarrow$ kaikille tapauksille yhteistä Θ-arvoa ei voida määrittää

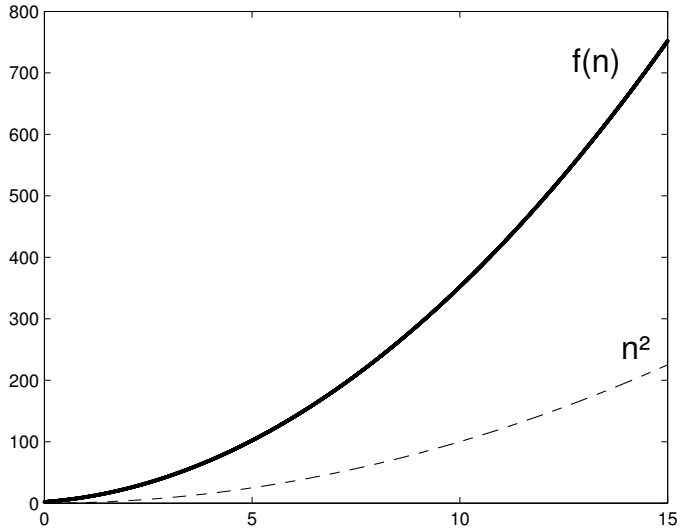
Esimerkki

Otetaan funktio $f(n) = 3n^2 + 5n + 2$.

Suoritetaan sille aiemmin sovitut yksinkertaistukset:

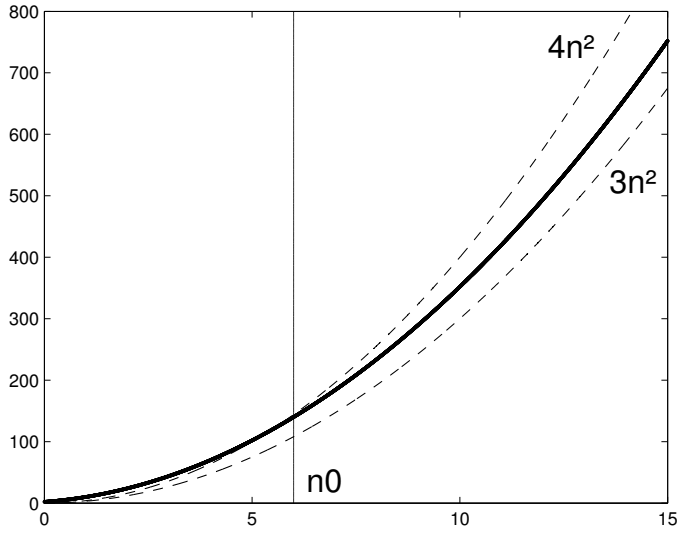
- alemman asteen termit pois
- vakiokertoimet pois

$\Rightarrow f(n) = \Theta(n^2)$



Vakuuttuaksemme asiasta etsimme kertoimet c_1 ja c_2:

$3n^2 \leq 3n^2 + 5n + 2 \leq 4n^2$, kun $n \geq 6$
$\Rightarrow c_1 = 3$, $c_2 = 4$ ja $n_0 = 6$ toimivat
$\Rightarrow f(n) = O(n^2)$ ja $\Omega(n^2)$
$\Rightarrow f(n) = \Theta(n^2)$

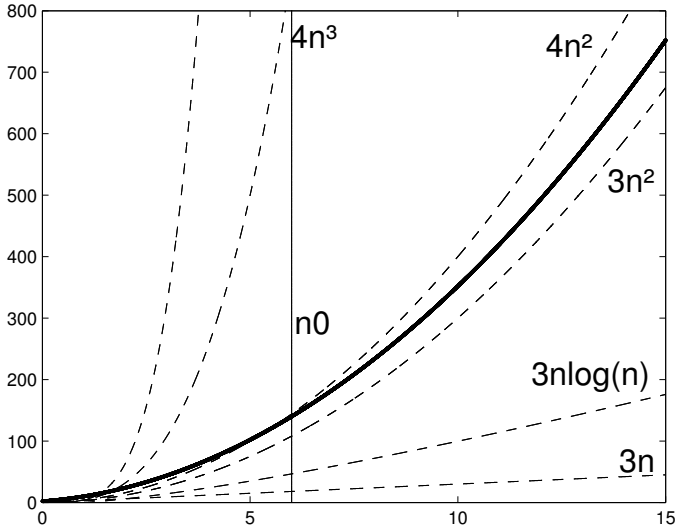


Selvästi kerroin $c_2 = 4$ toimii myös kun $g(n) = n^3$, sillä kun $n \geq 6, n^3 > n^2$
$\Rightarrow f(n) = O(n^3)$

- sama pätee kun $g(n) = n^4$...

Ja alapuolella kerroin $c_1 = 3$ toimii myös kun $g(n) = n \lg n$, sillä kun $n \geq 6, n^2 > n \lg n$
$\Rightarrow f(n) = \Omega(n \lg n)$

- sama pätee kun $g(n) = n$ tai $g(n) = \lg n$



5.2 Suorituskykykäsitteitä

Tähän mennessä algoritmien suorituskykyä on arvioitu lähinnä suoritusajan näkökulmasta. Muitakin vaihtoehtoja kuitenkin on:

- Voidaan mitata myös esimerkiksi muistinkulutusta tai kaistanleveyttä.

Lisäksi käytännössä tulee ottaa huomioon ainakin seuraavat seikat:

- Millä mittayksiköllä resurssien kulutusta mitataan?
- Miten syötekoko määritellään?
- Mitataanko huonoimman, parhaan vai keskimääräisen tapauksen resurssien kulutusta?
- Millaiset syötekoot tulevat kysymykseen?
- Riittääkö kertaluokkamerkintöjen tarkkuus vai tarvitaanko tarkempaa tietoa?

Ajoajan mittayksiköt

Valittaessa ajoajan mittayksikköä "askelta" pyritään yleensä mahdollisimman koneriippumattomaan ratkaisuun:

- Todelliset aikayksiköt kuten sekunti eivät siis kelpaa.
- Vakiokertoimet käyvät merkityksettömiksi.
  $\Rightarrow$ Jäljelle jää kertaluokkamerkintöjen tarkkuustaso.

$\Rightarrow$ askeleeksi voidaan katsoa mikä tahansa enintään vakioajan vievä operaatio.

- Vakioaikaiseksi tulkitaan mikä tahansa operaatio, jonka ajankulutus on riippumaton syötekoosta.
- Tällöin on olemassa jokin syötteen koosta riippumaton aikamäärä, jota operaation kesto ei milloinkaan ylitä.
- Yksittäisiä askeleita ovat esimerkiksi yksittäisen muuttujan sijoitus, **if**-lauseen ehdon testaus etc.
- Askeleen rajauksen kanssa ei tarvitse olla kovin tarkka, koska $\Theta(1) + \Theta(1) = \Theta(1)$.

Muistin käytön mittayksiköt

Tarkkoja yksiköitä ovat lähes aina bitti, tavu (8 bittiä) ja sana (jos sen pituus tunnetaan).

Eri tyyppien muistin käyttö on usein tunnettu, joskin se vaihtelee vähän eri tietokoneiden ja kielten välillä.

- kokonaisluku on yleensä 16 tai 32 bittiä
- merkki on yleensä 1 tavu = 8 bittiä
- osoitin on yleensä 4 tavua = 32 bittiä
- taulukko $A[1 \ldots n]$ on usein $n \cdot$ <alkion koko>

$\Rightarrow$ Tarkka muistin käytön arvioiminen on usein mahdollista, joskin huolellisuutta vaativaa.

Kertaluokkamerkinnät ovat käteviä silloin, kun tarkka tavujen laskeminen ei ole vaivan arvoista.

Jos algoritmi säilyttää yhtäaikaa koko syöteaineiston, niin arvioinnissa kannattaa erottaa syöteaineiston kuluttama muisti muusta muistin tarpeesta.

- $\Theta(1)$ lisämuistin tarve vs. $\Theta(n)$ lisämuistin tarve
- Kannattaa kuitenkin huomata, että esimerkiksi merkkijonon etsintä syötetiedostosta ei talleta yhtäaikaa koko syöteaineistoa, vaan selaa sen läpi.

6 Pikalajittelu ja satunnaistaminen

MERGE-SORTilla osaongelmiin jako oli helppoa ja ratkaisujen yhdistämisessä nähtiin paljon työtä.

Tutustutaan seuraavaksi keskimäärin erittäin nopeaan järjestämisalgoritmiin QUICKSORT, jossa työ tehdään jakovaiheessa.

QUICKSORTin kautta kohdataan uusi suunnitteluperiaate: *satunnaistaminen*.

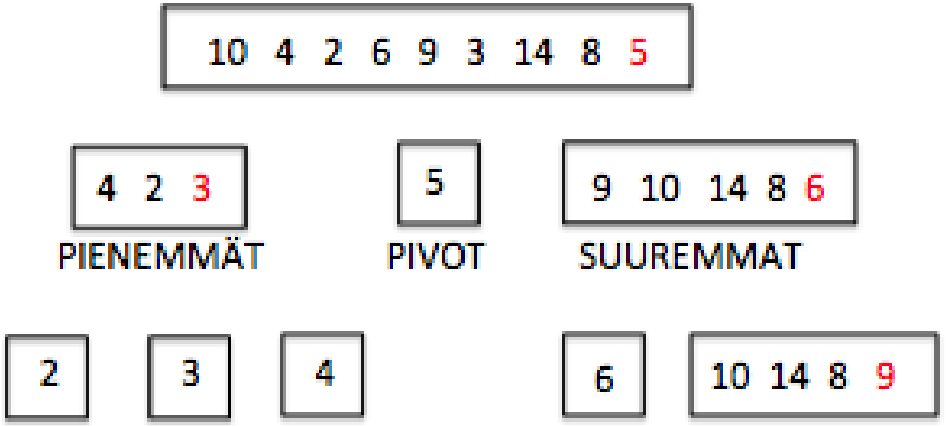
6.1 QUICKSORT

Ongelman jakaminen pienemmiksi osaongelmiksi

- Valitaan jokin taulukon alkioista jakoalkioksi eli *pivot-alkioksi*.
- Muutetaan taulukon alkioiden järjestystä siten, että kaikki jakoalkiota pienemmät tai yhtäsuuret alkiot ovat taulukossa ennen jakoalkiota ja suuremmat alkiot sijaitsevat jakoalkion jälkeen.
- Jatketaan alku ja loppuosien jakamista pienemmiksi, kunnes ollaan päästy 0:n tai 1:n kokoisiin osataulukoihin.

Kertaus: QUICKSORT-algoritmi

QUICKSORT($A, left, right$)
1 **if** $left < right$ **then** *(triviaalitapaukselle ei tehdä mitään)*
2 p := PARTITION($A, left, right$) *(muuten jaetaan alkuosaan ja loppuosaan)*
3 QUICKSORT($A, left, p-1$) *(järjestetään jakoalkiota pienemmät)*
4 QUICKSORT($A, p+1, right$) *(järjestetään jakoalkiota suuremmat)*



Kuva 11: Jako pienempiin ja suurempiin

Pienet osaongelmat:

- 0:n tai 1:n kokoiset osataulukot ovat valmiiksi järjestyksessä.

Järjestyksessä olevien osataulukoiden yhdistäminen:

- Kun alkuosa ja loppuosa on järjestetty on koko (osa)taulukko automaattisesti järjestyksessä.
  - kaikki alkuosan alkiothan ovat loppuosan alkioita pienempiä, kuten pitääkin

Ositus- eli *partitiointialgoritmi* jakaa taulukon paikallaan vaaditulla tavalla.

PARTITION($A, left, right$)

```
1  p := A[ right ]                    (otetaan pivotiksi viimeinen alkio)
2  i := left − 1                      (merkitään i:llä pienten puolen loppua)
3  for j := left to right − 1 do      (käydään läpi toiseksi viimeiseen alkioon asti)
4      if A[ j ] ≤ p                  (jos A[ j] kuuluu pienten puolelle...)
5          i := i + 1                 (... kasvatetaan pienten puolta...)
6          exchange A[ i ] ↔ A[ j ]   (... ja siirretään A[j] sinne)
7  exchange A[ i + 1 ] ↔ A[ right ]   (sijoitetaan pivot pienten ja isojen puolten väliin)
8  return i + 1                       (palautetaan pivot-alkion sijainti)
```

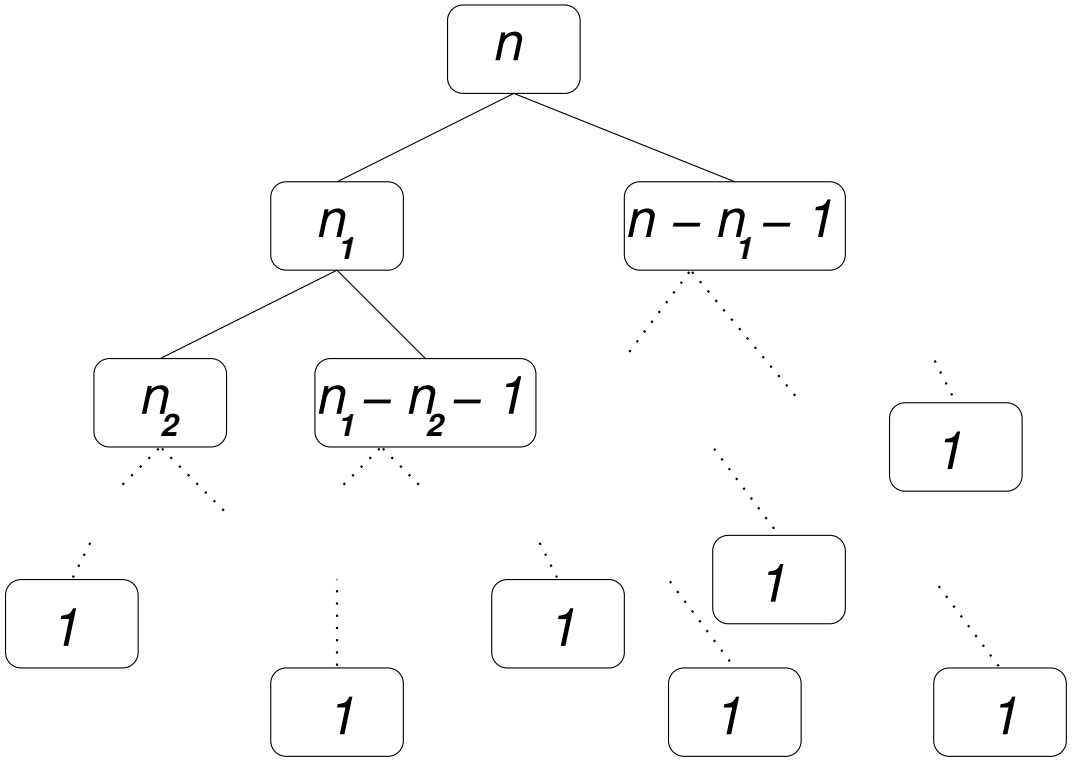
Kuinka nopeasti PARTITION toimii?

- **For**-silmukka tekee n - 1 kierrosta, kun n on $right$ - $left$
- Kaikki muut operaatiot ovat vakioaikaisia.

$\Rightarrow$ Sen suoritusaika on $\Theta(n)$.

Kuten MERGE-SORTilla QUICKSORTin analyysi ei ole yhtä suoraviivainen rekursion vuoksi

- Koska PARTITIONIA ja rekursiivista kutsua lukuunottamatta kaikki QUICKSORTIN operaatiot ovat vakioaikaisia, keskitymme tarkastelemaan PARTITIONIN instanssien käyttämää aikaa.



- Kokonaisaika on edellisen kuvan esittämän puun solmujen aikojen summa.
- 1 kokoiselle taulukolle suoritus on vakioaikaista.
- Muille suoritus on lineaarista osataulukon kokoon nähden.
  $\Rightarrow$ Kokonaisaika on siis Θ(solmujen numeroiden summa).

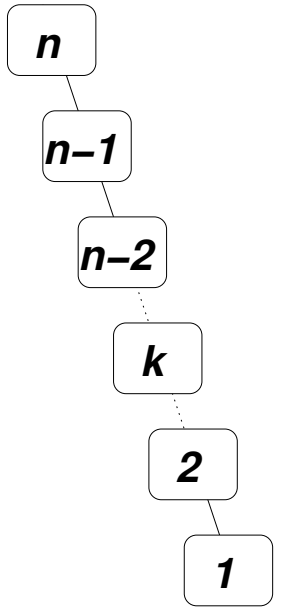
Pahimman tapauksen suoritusaika

- Solmun numero on aina pienempi kuin isäsolmun numero, koska jakoalkio on jo oikealla paikallaan, eikä se kuulu kumpaankaan järjestettävään osataulukkoon

  $\Rightarrow$ puussa voi siis olla kerroksia enintään n kappaletta

- pahin tapaus realisoituu, kun jakoalkioksi valitaan aina pienin tai suurin alkio
  - näin tapahtuu esimerkiksi valmiiksi järjestetyllä taulukolla
- solmujen numeroiden summa on n + n - 1 + $\cdots$ + 2 + 1

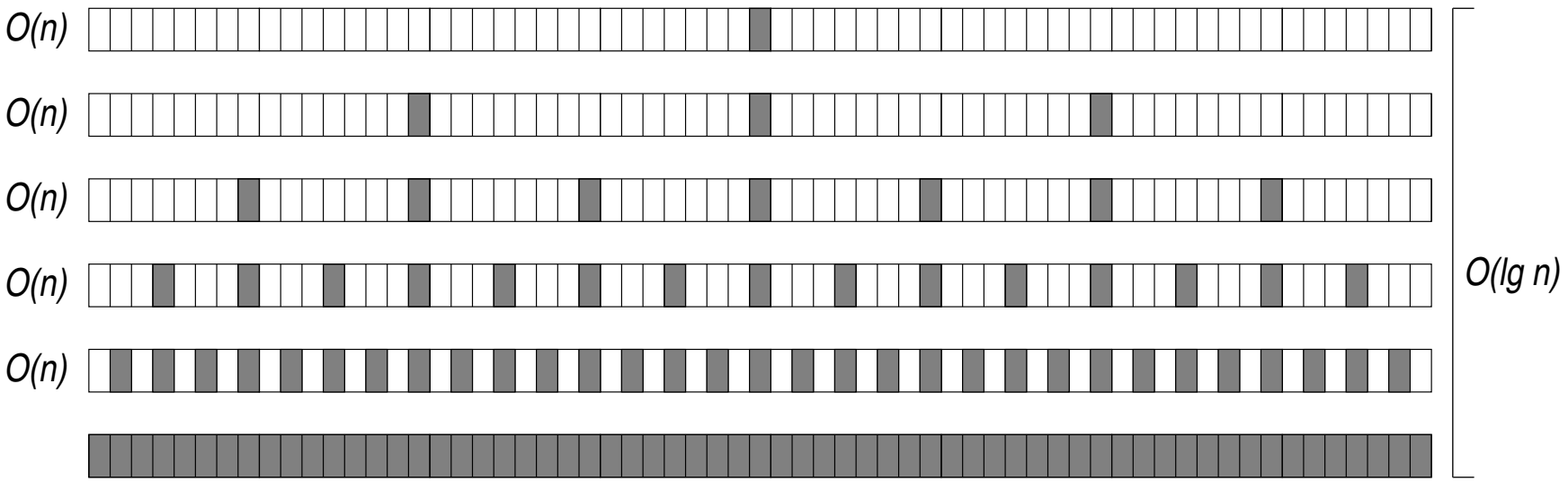
$\Rightarrow$ QUICKSORTIN suoritusaika on $O(n^2)$



Paras tapaus realisoituu kun taulukko jakautuu aina tasan.

- Alla oleva kuva näyttää kuinka osataulukoiden koot pienenevät.
  - harmaat ruudut ovat jo oikealla paikallaan olevia alkioita
- Jokaisella tasolla tehtävä työ on kertaluokassa $\Theta(n)$.
  - tästä saamme pessimistisen arvion suorituspuun korkeudelle parhaassa tapauksessa $\Rightarrow O(\lg n)$

  $\Rightarrow$ Parhaan tapauksen suoritusajan yläraja on $O(n \lg n)$.



QUICKSORTIN kohdalla parhaan ja huonoimman tapauksen suoritusajat eroavat toisistaan selvästi.

- Olisi mielenkiintoista tietää keskimääräinen suoritusaika.
- Sen analysoiminen menee tämän kurssin tavoitteiden ulkopuolelle, mutta on osoitettu, että jos aineisto on tasajakautunutta, keskimääräinen suoritusaika on $\Theta(n \lg n)$.
- Keskimääräinen suoritusaika on siis varsin hyvä.

QUICKSORTIIN liittyy kuitenkin se kiusallinen seikka, että sen pahimman tapauksen suoritus on hidasta ja sen esiintyminen on käytännössä varsin todennäköistä.

- On helppoa kuvitella tilanteita, joissa aineisto on jo järjestyksessä tai melkein järjestyksessä.

$\Rightarrow$ Tarvitaan keino, jolla pahimman tapauksen systemaattisen esiintymisen riskiä saadaan pienennettyä.

Satunnaistaminen on osoittautunut tässä varsin tehokkaaksi.

QUICKSORTIN etuja ja haittoja

Etuja:

- järjestää taulukon keskimäärin erittäin tehokkaasti
  - ajoaika on keskimäärin $\Theta(n \lg n)$
  - vakiokerroin on pieni
- tarvitsee vain vakiomäärän lisämuistia
- sopii hyvin virtuaalimuistiympäristöön
- käyttää tehokkaasti välimuistia

Haittoja:

- ajoaika on hitaimmillaan $\Theta(n^2)$
- hitaimman tapauksen tuottava syöte on kiusallisen tavallinen ilman satunnaistamista
- rekursiivisuus
  $\Rightarrow$ tarvitsee lisämuistia pinolle
- epävakaus

7 C++:n standardikirjasto

Tässä luvussa käsitellään C++:n standardikirjaston tietorakenteita ja algoritmeja.

Tarkoituksena on käsitellä sellaisia asioita, joihin tulee kiinnittää huomiota, jotta kirjastoa tulisi käyttäneeksi tarkoituksenmukaisesti ja tehokkaasti.

7.1 Yleistä C++:n standardikirjastosta

Standardikirjasto standardoitiin C++-kielen mukana syksyllä 1998, ja sitä on jonkin verran laajennettu myöhemmissä versioissa. Uusin standardiversio on C++17 vuodelta 2017.

Kirjasto sisältää tärkeimmät perustietorakenteet ja algoritmit.

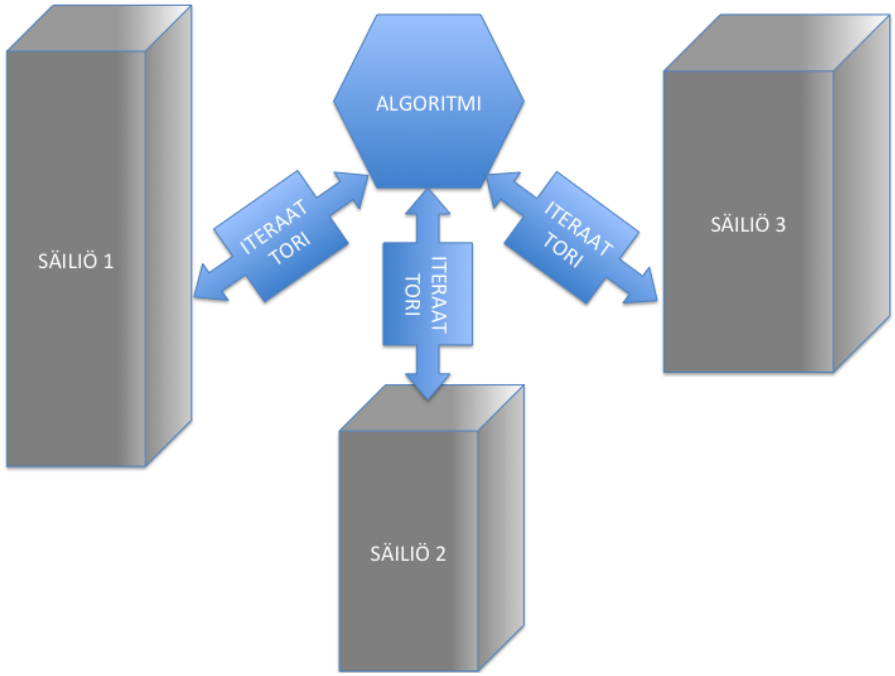
- useimmat tämän aineiston alkupuolen tietorakenteet ja algoritmit mukana muodossa tai toisessa

Rajapinnat ovat harkittuja, joustavia, geneerisiä ja tyyppiturvallisia.

Rajapintojen tarjoamien operaatioiden tehokkuudet on ilmaistu $\mathcal{O}$-merkinnällä.

Kirjaston geneerisyys on toteutettu käännösaikaisella mekanismilla: C++:n *malli* (template)

Tietorakennekurssin kannalta kiinnostavin standardikirjaston elementti on ns. STL (Standard Template Library): *säiliöt* eli kirjaston tarjoamat tietorakenteet, *geneeriset algoritmit* sekä *iteraattorit*, joiden avulla säiliöiden alkioita käsitellään.



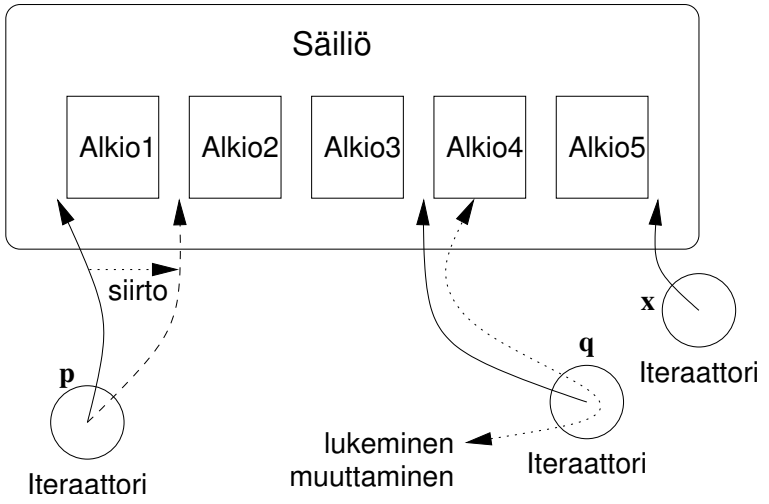
Kuva 12: STL:n osaset

C++11:n mukana tulleet *lambdat* ovat myös keskeisiä

7.2 Iteraattorit

Kaikki standardikirjaston tietorakenteet näyttäytyvät meille varsin samanlaisina mustina laatikoina, joista oikeastaan tiedämme ainoastaan, että ne sisältävät tallettamiamme alkioita ja että ne toteuttavat tietyt rajapintafunktiot.

Pystymme käsittelemään säiliöiden sisältöä ainoastaan noiden rajapintafunktioiden sekä iteraattorien avulla.



Iteraattorit ovat kahvoja tai "kirjanmerkkejä" tietorakenteen alkioihin.

- kukin iteraattori osoittaa joko tietorakenteen alkuun, loppuun tai kahden alkion väliin.
- säiliöiden rajapinnassa on yleensä funktiot **begin()** ja **end()**, jotka palauttavat säiliön alkuun ja loppuun osoittavat iteraattorit
- iteraattorin läpi pääsee käsiksi sen oikealla puolella olevaan alkioon, paitsi jos kysymyksessä on *käänteisiteraattori* (reverse iterator), joilloin sen läpi käsitellään vasemmanpuoleista alkiota
- käänteisiteraattorille myös siirto-operaatiot toimivan käänteisesti, esimerkiksi ++ siirtää iteraattoria pykälän vasemmalle
- **begin()**:ä ja **end()**:ä vastaavat käänteisiteraattorit saa **rbegin()**:llä ja **rend()**:llä

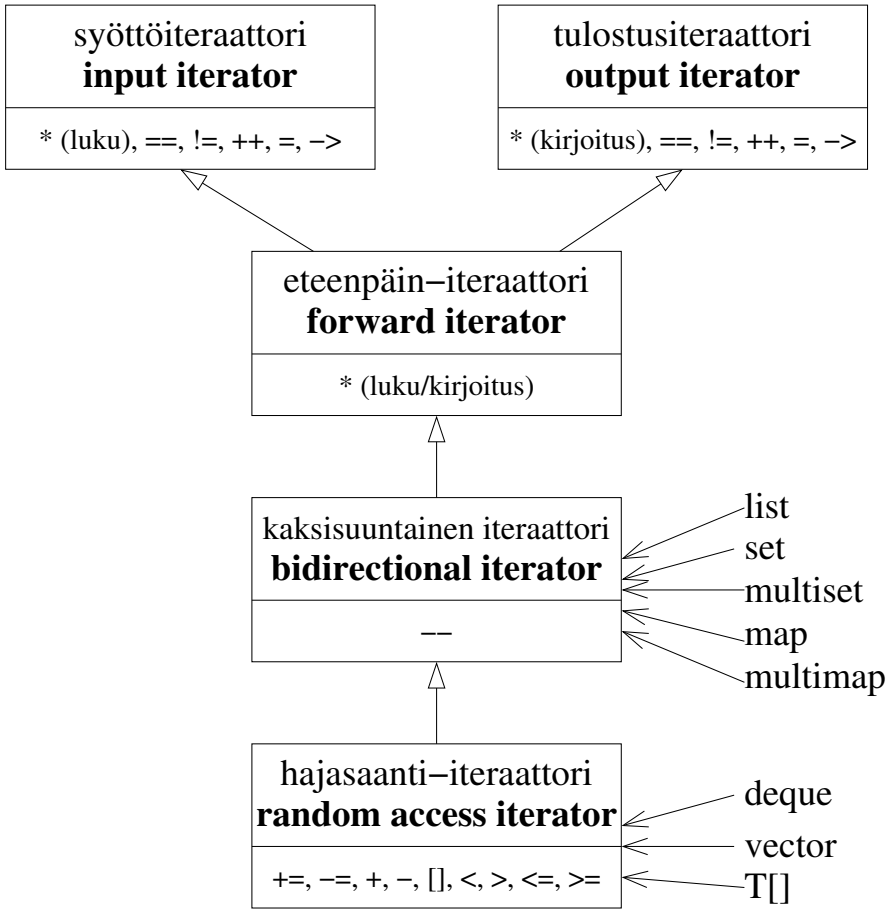
- nimensä mukaisesti iteraattoria voi siirtää säiliön sisällä, ja sen avulla säiliön voi käydä läpi
- iteraattorin avulla voi lukea ja kirjoittaa
- säiliöön lisättävien ja siitä poistettavien alkioiden sijainti yleensä ilmaistaan iteraattoreiden avulla

Kullakin säiliöllä on oma iteraattorityyppinsä.

- eri säiliöt tarjoavat erilaisia mahdollisuuksia siirtää iteraattoria nopeasti paikasta toiseen (vrt. taulukon/listan mielivaltaisen alkion lukeminen)
- suunnitteluperiaatteena on ollut, että kaikkien iteraattoreille tehtävien operaatioiden tulee onnistua vakioajassa, jotta geneeriset algoritmit toimisivat luvatulla tehokkuudella riippumatta siitä mikä iteraattori niille annetaan
- iteraattorit voidaan jakaa kategorioihin sen mukaan, millaisia vakioaikaisia operaatioita ne pystyvät tarjoamaan

Syöttöiteraattorin (input iterator) avulla voi vain lukea alkioita, mutta ei muuttaa niitä

- iteraattorin osoittaman alkion arvon voi lukea (*p)
- iteraattorin osoittaman alkion kentän voi lukea tai kutsua sen jäsenfunktiota (p->)
- iteraattoria voi siirtää askeleen eteenpäin (++p tai p++)
- iteraattoreita voi sijoittaa ja vertailla toisiinsa (p=q, p==q, p!=)



Tulostusiteraattori (output iterator) on kuten syöttöiteraattori, mutta sen avulla voi vain muuttaa alkioita. (*p=x)

Eteenpäin-iteraattori (forward iterator) on yhdistelmä syöttö- ja tulostusiteraattorien rajapinnoista.

Kaksisuuntainen iteraattori (bidirectional iterator) osaa lisäksi siirtyä yhden askeleen kerrallaan taaksepäin. (--p tai p--)

Hajasaanti-iteraattori (random access iterator) on kuin kaksisuuntainen iteraattori, mutta sitä voi lisäksi siirtää mielivaltaisen määrän eteen- tai taaksepäin.

- iteraattoria voi siirtää n askelta eteen- tai taaksepäin (p+=n, p-=n, q=p+n, q=p-n)
- iteraattorista n:n alkion päässä olevan alkion pystyy lukemaan ja sitä pystyy muokkaamaan (p(n))
- kahden iteraattorin välisen etäisyyden pystyy selvittämään (p-q)
- iteraattoreiden keskinäistä suuruusjärjestystä voi vertailla, iteraattori on toista “pienempi”, jos sen paikka on säiliössä ennen tätä (p<q, p<=q, p>q, p>=q)

Iteraattorit ovat osoitinabstraktio $\rightarrow$ Iteraattorien operaatioiden syntaksi muistuttaa selvästi C++:n osoitinsyntaksia.

Iteraattorit saadaan otettua käyttöön komennolla
`#include <iterator>`

Oikean tyyppinen iteraattori saadaan luotua esimerkiksi seuraavalla syntaksilla.

```
säiliö<talletettava tyyppi>::iterator p;
```

Iteraattoreiden kanssa avainsana `auto` on käyttökelpoinen:

```
auto p = begin( säiliö );
// → std::vector<std::string>::iterator
```

Säiliöihin tehtävät poistot ja lisäykset saattavat *mitätöidä* säiliöön jo osoittavia iteraattoreita.

- ominaisuus on säiliö-kohtainen, joten sen yksityiskohdat käsitellään säiliöiden yhteydessä

Tavallisten iteraattorien lisäksi STL tarjoaa joukon iteraattorisovittimia.

- niiden avulla voidaan muunnella geneeristen algoritmien toiminnallisuutta
- edellä mainitut *käänteisiteraattorit* (reverse iterator) ovat iteraattorisovittimia
- *lisäysiteraattorit* (insert iterator/inserter) ovat tärkeitä iteraattorisovittimia.
  - ne ovat tulostusiteraattoreita, jotka lisäävät alkioita halutulle paikalle säiliöön kopioimisen sijasta
  - säiliön alkuun lisäävän iteraattorin saa funktiokutsulla `front_inserter`(säiliö)
  - säiliön loppuun lisäävän iteraattorin saa funktiokutsulla `back_inserter`(säiliö)
  - annetun iteraattorin kohdalle lisäävän iteraattorin saa funktiokutsulla `inserter`(säiliö, paikka)

- *virtaiteraattorit* (stream iterator) ovat syöttö- ja tulostusiteraattoreita, jotka käyttävät säiliöiden sijaista C++:n tiedostovirtoja
  - `cin` virrasta haluttua tyyppiä lukevan syöttöiteraattorin saa syntaksilla `istream_iterator`<tyyppi> (`cin`)
  - `cout` virtaan haluttua tyyppiä pilkuilla erotettuna tulostavan tulostusiteraattorin saa syntaksilla `ostream_iterator`<tyyppi> (`cout, ','`)
- *siirtoiteraattorit* (move iterator) muuttavat alkion kopioinnin iteraattorin avulla alkion siirtämiseksi.

7.3 Säiliöt

Standardikirjaston säiliöt kuuluvat pääsääntöisesti kahteen kategoriaan rajapintojensa puolesta:

- *sarjat* (sequence)
  - alkioita pystyy hakemaan niiden järjestysnumeron perusteella
  - alkioita pystyy lisäämään ja poistamaan halutusta kohdasta
  - alkioita pystyy selaamaan järjestyksessä
- *assosiatiiviset säiliöt* (associative container)
  - alkiot sijoitetaan säiliöön *avaimen* määräämään kohtaan
  - talletettavien alkioiden avainten arvoja pitää pystyä vertaamaan toisiinsa oletusarvoisesti operaattorilla < järjestetyissä säiliöissä
- Rajapintojen tarjoamista jäsenfunktiosta näkee, mikä säiliön mielekäs käyttötarkoitus on

Kirjaston säiliöt:

Säiliötyyppi	Kirjasto

Sarjat	`array`
	`vector`
	`deque`
	`list`
	(`forward_list`)

Assosiatiiviset	`map` `set`
Järjestämättömät assosiatiiviset	`unordered_map` `unordered_set`
Säiliösovittimet	`queue` `stack`

Säiliöt noudattavat arvon välitystä.

- säiliö ottaa talletettavasta datasta kopion
- säiliö palauttaa kopioita sisältämästään tiedosta

  $\Rightarrow$ säiliön ulkopuolella tehtävät muutokset eivät vaikuta säiliön sisältämään dataan
- kaikilla säiliöihin talletettavilla alkioilla tulee olla kopiorakentaja ja sijoitusoperaattori.
  - perustyypeillä sellaiset ovat valmiina
- itse määriteltyä tyyppiä olevat alkiot kannattaa tallettaa osoittimen päähän
  - näinhän kannattaisi tehdä tehokkuussyistä joka tapauksessa

- C++11 tarjoaa kätevän työkalun muistinhallinnan helpottamiseksi `shared_pointer` tilanteissa, joissa useampi taho tarvitsee resurssia
  - sisältää sisäänrakennetun viitelaskurin ja tuhoaa alkion kun viitelaskuri nollautuu
  - deleteä ei tarvitse eikä saakaan kutsua
  - luominen: `auto pi = std::make_shared<Olio>(params);`
  - näppärä erityisesti jos halutaan tehdä kahden avaimen mukaan järjestetty tietorakenne:
    * sijoitetaan oheisdata `shared_pointer`in päähän
    * sijoitetaan osoittimet kahteen eri säiliöön kahden eri avaimen mukaan

Sarjat:

Taulukko `array<tyyppi>` on vakiokokoinen taulukko.

- Luodaan `std::array<tyyppi, koko> a = {arvo, arvo,...};`
- Indeksointi jäsenfunktiolla `.at()` tai `[]`-operaatiolla. Funktioilla `front()` ja `back()` voidaan käsitellä ensimmäistä ja viimeistä alkiota.
- Tarjoaa iteraattorit ja käänteisiteraattorit
- `empty()`, `size()` ja `max_size()`
- Funktion `data()` avulla päästään suoraan käsiksi sisällä olevaan taulukkoon

Taulukon operaatiot ovat vakioaikaisia, mutta `fill()` ja `swap()` ovat $O(n)$

Vektori `vector<tyyppi>` on lopustaan joustavarajainen taulukko

- Luodaan `vector<int> v {arvo, arvo, ...};`
- Vakioaikainen indeksointi `.at()`, `[]` sekä (tasatusti) vakioaikainen lisäys `push_back()` ja poisto `pop_back()` vektorin lopussa
- alkion lisääminen muualle `insert()`:illä ja poistaminen `erase`:lla on lineaarista, $O(n)$
- `emplace_back(args);` rakentaa alkion suoraan vektoriin
- Vektorin kokoa voi kasvattaa funktiolla `.resize(koko, alkuarvo);`
  - alkuarvo on vapaaehtoinen
  - tarvittaessa vektori varaa lisää muistia automaattisesti
  - muistia voi varata myös ennakkoon: `.reserve(koko)`, `.capacity()`
- iteraattorien mitätöitymistä tapahtuu seuraavissa tilanteissa
  - mikäli vectorille ei ole etukäteen varattu riittävää tilaa, voi mikä tahansa lisäys aiheuttaa kaikkien iteraattoreiden mitätöitymisen

  - poistot aiheuttavat mitätöitymisen ainoastaan poistopaikan jälkeen tuleville iteraattoreille
  - lisäykset keskelle aiheuttavat aina lisäyspaikan jälkeen tulevien iteraattoreiden mitätöitymisen
- `vector<bool>`:ille on määritelty erikoistoteutus, joka poikkeaa siitä mitä yleinen toteutus tekisi muistinkäytön tehostamiseksi
  - tavoite: mahdollistaa 1 bitti / alkio, kun tavallinen toteutus luultavasti veisi 1 tavu / alkio eli 8 bittiä / alkio

Pakka `deque<tyyppi>` on molemmista päistään avoin taulukko

- luodaan `deque<tyyppi> d {arvo, arvo, arvo...};`
- Rajapinta vektorin kanssa yhtenevä, mutta tarjoaa tehokkaan ($O(1)$ tasattu suoritusaika) lisäyksen ja poiston *molemmissa* päissä: `.push_front(alkio)`, `.emplace_front(args)`, `.pop_front()`
- iteraattorien mitätöitymistä tapahtuu seuraavissa tilanteissa
  - kaikki lisäykset voivat mitätöidä iteraattorit
  - poistot keskelle mitätöivät kaikki iteraattorit
  - kaikki paitsi päihin kohdistuvat lisäys- ja poisto-operaatiot voivat mitätöidä viitteet ja osoittimet

Lista on säiliö, joka tukee kaksisuuntaista iterointia

- luodaan `list<tyyppi> l {arvo, arvo, arvo };`
- lisäys ja poisto kaikkialla vakioaikaista, indeksointioperaatiota ei ole
- lisäys ja poisto eivät mitätöi iteraattoreita ja viitteitä (paitsi tietysti poistettuihin alkioihin)
- listalla on monenlaisia erikoispalveluja
  - `.splice(paikka, toinen_lista)` siirtää toisen listan nykyisen osaksi paikan eteen, $O(1)$.
  - `.splice(paikka, lista, alkio)` siirtää alkion toisesta tai samasta listasta paikan eteen, $O(1)$.
  - `.splice(paikka, lista, alku, loppu)` siirtää välin alkiot paikan eteen, $O(1)$ tai lineaarinen
  - `.merge(toinen_lista)` ja `.sort()`, vakaa, keskimäärin $O(nlogn)$
  - `.reverse()`, lineaarinen

Assosiatiiviset säiliöt:

Joukko `set<tyyppi>` ja **monijoukko** `multiset<tyyppi>` on dynaaminen joukko, josta voi

- etsiä, lisätä ja poistaa logaritmisessa ajassa
- selata suuruusjärjestyksessä tasatussa vakioajassa siten että läpikäynti alusta loppuun on aina lineaarinen operaatio
- alkioilla on oltava suuruusjärjestys "<"
- voi määritellä erikseen joko osana tyyppiä tai rakentajan parametrina
- tutkii yhtäsuuruuden kaavalla $\neg(x<y \vee y<x) \Rightarrow$ "<" määriteltävä järkevästi ja tehokkaaksi

Monijoukossa sama alkio voi olla moneen kertaan, joukossa ei

Luodaan `std::set<tyyppi> s {arvo, arvo, arvo...};`

- alkion arvon muuttaminen on pyritty estämään
  - sen sijaan pitää poistaa vanha alkio ja lisätä uusi
- mielenkiintoisia operaatioita:
  - `.find(alkio)` etsii alkion (monijoukolle ensimmäisen monesta), tai palauttaa `.end()` jollei löydä
  - `.lower_bound(alkio)` etsii ensimmäisen, joka on $\geq$ alkio
  - `.upper_bound(alkio)` etsii ensimmäisen, joka on $>$ alkio
  - `.equal_range(alkio)` palauttaa `make_pair( .lower_bound(alkio), .upper_bound(alkio) )`, mutta selviää yhdellä etsinnällä (joukolle ko. välin leveys on 0 tai 1)
  - joukoille `insert` palauttaa parin (paikka, lisättiin), koska alkiota ei saa lisätä, jos se on jo joukossa
- standardi lupaa, että iteraattorit eivät vanhene lisäyksessä ja poistossa (paitsi tietysti poistettuihin alkioihin kohdistuneet)

Kuvaus `map<avaimen_tyyppi, alkion_tyyppi>` ja **monikuvaus** `multimap<avaimen_tyyppi, alkion_tyyppi>`

- alkiot avain-oheisdatapareja
  - parin tyyppi on `pair<tyyppi1, tyyppi2>`
  - parin voi tehdä funktiolla `make_pair`:illä
  - parin kentät saa operaatioilla `.first()`, `.second()`
- luodaan

  ```
  std::map m<avain_tyyppi, alkio_tyyppi> m {{avain1, arvo1},
   {avain2, arvo2}, avain3, arvo3},...};
  ```

  esim.

  ```
  std::map<std::string,int> anim { {"bear",4}, {"giraffe",2},
   {"tiger",7} };
  ```

- `map`:ia voi poikkeuksellisesti indeksoida avaimen avulla $O(logn)$
  - Jos avainta ei löydy, lisää arvoparin avain-arvo rakenteeseen
- nytkään iteraattorit eivät vanhene lisäyksessä ja poistossa

Hajautustaulu, Unordered set/multiset, joka sisältää joukon alkioita ja unordered map/multimap, joka sisältää joukon alkioita, jotka assosioidaan avainarvojoukolle.

- unordered map/set muistuttavat rajapinnaltaan mapia ja setia
- tärkeimmät erot:
  - alkiot eivät ole järjestyksessä (unordered)
  - lisäys, poisto ja etsintä ovat keskimäärin vakioaikaisia ja pahimmassa tapauksessa lineaarisia
  - tarjoavat hajautuksen kannalta olennaisia funktioita, kuten `rehash(koko)`, `load_factor()`, `hash_function()` ja `bucket_size()`.

- hajautustalun kokoa kasvatetaan automaattisesti, jotta lokeroiden keskimääräinen täyttöaste saadaan pidettyä sovitun rajan alapuolella
  - hajautustaulun koon muuttaminen *(rehashing)* on keskimäärin lineaarinen pahimmillaan neliöllinen operaatio
  - koon muuttaminen mitätöi kaikki iteraattorit, muttei osoittimia eikä viitteitä

Lisäksi Standardikirjastosta löytyy joitakin muita säiliöitä:

Bittivektori `bitset<bittien_määrä>`

- `#include<bitset>`
- tarkoitettu kiinteän kokoisten binääristen bittisarjojen käsittelyyn
- tarjoaa tyypillisiä binäärisiä operaatioita (AND, OR, XOR, NOT)

Merkkijonot `string`

- `#include<string>`
- vaikka C++:n merkkijonot on optimoitu muuhun tarkoitukseen eikä niitä yleensä ajatella säiliöinä, ne ovat muun lisäksi säiliöitäkin
- säilövät merkkejä, mutta saadaan säilömään muutakin
- niillä on mm. iteraattorit, `[...]`, `.at(...)`, `.size()`, `.capacity()` ja `swap`
- merkkijonot voivat kasvaa hyvin suuriksi ja varaavat tarvittaessa automaattisesti lisää muistia

- merkkijonojen muokkausoperaatioiden (katenointi, poisto) kanssa kannattaa olla varovainen, koska niissä suoritetaan muistinvarausta ja kopiointia, minkä vuoksi ne ovat pitkille merkkijonoille varsin raskaita
- usein on muutenkin järkevää sijoittaa merkkijonot esimerkiksi osoittimen päähän sijoitettaessa niitä säiliöihin, jottei niitä turhaan kopioitaisi
- samasta syystä merkkijonot tulee välittää viiteparametreina

Säiliöiden lisäksi STL tarjoaa joukon säiliösovittimia, jotka eivät itsessään ole säiliöitä, mutta joiden avulla säiliön rajapinnan saa “sovitettua toiseen muottiin”:

Pino `stack<alkion_tyyppi, säiliön_tyyppi>`

- tarjoaa normaalien luokka-operaatioiden lisäksi vain
  - pino-operaatiot, `.push(...)`, `.top()`, `.pop()`
  - koon kyselyt `.size()` ja `.empty()`
  - vertailut "==", "<" jne.
- `.pop()` ei palauta mitään, ylimmän alkion saa tarkasteltavaksi `.top()`:illa
- pinon ylintä alkiota voi muuttaa paikallaan:
  ```
  pino.top() = 35;
  ```
- kurssin kannalta kiinnostavaa on, että käyttäjä voi valita taulukkoon tai listaan perustuvan toteutuksen
  - mikä tahansa säiliö, joka tarjoaa `back()`, `push_back()` ja `pop_back()` käy, erityisesti `vector`, `list` ja `deque`.
  - `stack<tyyppi> perus_pino;` (`deque`)
  - `stack<tyyppi, list<tyyppi> > lista_pino;`

Jono `queue<alkion_tyyppi, säiliön_tyyppi>`

- jono-operaatiot `.push(...)`, `.pop()`, `.front()`, `.back()`(!)
- mikä tahansa säiliö, joka tarjoaa `front()`, `back()`, `push_back()` ja `pop_front` käy
- muuten kutakuinkin samanlainen kuin pino

Prioriteettijono `priority_queue<alkion_tyyppi,säiliön_tyyppi>`

- lähes täysin samanlainen rajapinta kuin jonolla
- toteutus kekona
- mikä tahansa säiliö, jolla `front()`, `push_back()` ja `pop_back()` ja hajasaanti-iteraattoreita tukeva käy, erityisesti `vector` (oletus) ja `deque`
- alkioilla eri järjestys: `.top()` palauttaa suurimman
- yhtäsuurista palauttaa minkä vain
- ylintä alkiota ei voi muuttaa `top`:in avulla paikallaan
- kuten assosiatiivisilla säiliöillä, järjestämisperiaatteen voi antaa <>-parametrina tai rakentajan parametrina (strict weak ordering)

tieto-rakenne	lisäys loppuun	lisäys muualle	1. poisto	alkion poisto	n:s alkio (indeks.)	tietyn etsintä	suurimman poisto
array					$O(1)$	$O(n)_{[2]}$	
vector	$O(1)$	$O(n)$	$O(n)$	$O(n)_{[1]}$	$O(1)$	$O(n)_{[2]}$	$O(n)_{[3]}$
list	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)_{[3]}$
deque	$O(1)$	$O(n)_{[4]}$	$O(1)$	$O(n)_{[1]}$	$O(1)$	$O(n)_{[2]}$	$O(n)_{[3]}$
stack$_{[9]}$	$O(1)$			$O(1)_{[5]}$			
queue$_{[9]}$		$O(1)_{[6]}$		$O(1)_{[7]}$			
priority queue$_{[9]}$		$O(\log n)$ $_{[10]}$					$O(\log n)_{[8]}$
set (multiset)		$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(\log n)$	$O(\log n)$
map (multimap)		$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(\log n)$	$O(\log n)$
unordered_ (multi)set		$O(n)$ $\approx \Theta(1)$		$O(n)$ $\approx \Theta(1)$		$O(n)$ $\approx \Theta(1)$	$O(n)$ $O(n)$
unordered_ (multi)map		$O(n)$ $\approx \Theta(1)$		$O(n)$ $\approx \Theta(1)$		$O(n)$ $\approx \Theta(1)$	$O(n)$ $O(n)$

[1] vakioaikainen viimeiselle alkiolle, muuten lineaarinen
[2] logaritminen jos tietorakenne on järjestetty, muuten lineaarinen
[3] vakioaikainen jos tietorakenne on järjestetty, muuten lineaarinen
[4] vakioaikainen ensimmäiselle alkiolle, muuten lineaarinen
[5] mahdollinen vain viimeiselle alkiolle
[6] vain alkuun lisääminen on mahdollista
[7] vain lopusta poistaminen on mahdollista
[8] kysyminen vakioajassa, poistaminen logaritmisessa ajassa
[9] säiliösovitin
[10] lisäys tapahtuu automaattisesti kekojärjestyksen mukaiselle paikalle

7.4 Geneeriset algoritmit

Standardikirjasto tarjoaa useimmat tähän mennessä käsitellyistä algoritmeista.

Algoritmit on kaikki toteutettu funktiomalleina, jotka saavat kaikki tarvitsemansa tiedon käsiteltävistä säiliöistä parametrien avulla.

Algoritmeille ei kuitenkaan koskaan anneta parametrina kokonaisia säiliöitä vaan ainoastaan iteraattoreita niihin.

- algoritmeilla voidaan käsitellä myös säiliön osia kokonaisten säiliöiden sijasta
- algoritmi voi saada parametrinaan iteraattoreita erityyppisiin säiliöihin, jolloin yhdellä funktiokutsulla voidaan yhdistää esimerkiksi vectorin ja listan sisällöt ja tallettaa tulos joukkoon
- algoritmien toimintaa voidaan muuttaa iteraattorisovittimien avulla
- ohjelmoija voi toteuttaa omiin tietorakenteisiinsa iteraattorit, jonka jälkeen algoritmit toimivat myös niille

Kaikkia algoritmeja ei kuitenkaan pystytä suorittamaan kaikille tietorakenteille tehokkaasti.
$\Rightarrow$ osa algoritmeista hyväksyy parametreikseen vain tietyn iteraattorikategorian iteraattoreita.

- tämä takaa algoritmien tehokkuuden, koska kaikki iteraattorin tarjoamat operaatiot ovat vakioaikaisia
- jos iteraattori on väärää tyyppiä, annetaan käännösaikainen virhe-ilmoitus

  $\Rightarrow$ jos algoritmille annetaan tietorakenne, jolle sitä ei voida toteuttaa tehokkaasti, se ei edes käänny

Standardikirjaston algoritmit ovat kirjastossa `algorithm`. Lisäksi standardi määrittelee C-kielen algoritmikirjaston `cstdlib`.

jakaa algoritmit kolmee pääryhmään: muuttamattomat sarjalliset operaatiot, muuttavat sarjalliset operaatiot ja järjestäminen sekä siihen liittyvät operaatiot.

Seuraavaksi lyhyt kuvaus joistakin kurssin kannalta kiinnostavimmista algoritmeista (näiden lisäksi on vielä

runsaasti suoraviivaisia selaamiseen yms. perustuvia algoritmeja):

Puolitushaku

- `binary_search(`eka`,` $loppu$`,` $arvo$`)` kertoo onko $arvo$ järjestetyssä jononpätkässä
  - eka ja $loppu$ ovat iteraattoreita, jotka osoittavat etsittävän alueen alkuun ja loppuun, muttei välttämättä säiliön alkuun ja loppuun
- samaa arvoa voi olla monta peräkkäin

  $\Rightarrow$ `lower_bound` ja `upper_bound` palauttavat sen alueen rajat, jolla on $arvo$a
  - alaraja on, yläraja ei ole mukana alueessa
- rajat saa myös pariksi yhdistettynä yhdellä etsinnällä: `equal_range`
- vertaa BIN-SEARCH sivu 74

Järjestämisalgoritmit

- `sort`($alku$, $loppu$) ja `stable_sort`($alku$, $loppu$)
- `sort`in suoritusaika $O(nlogn)$ ja `stable_sort`in $O(nlogn)$ jos tarpeeksi lisämuistia on saatavilla, muuten $O(nlog^2n)$
- järjestelyalgoritmit vaativat parametreikseen hajasaanti-iteraattorit

  $\Rightarrow$ eivät toimi listoille, mutta niissä on oma `sort` (ja ei-kopioiva `merge`) jäsenfunktiona
- löytyy myös järjestäminen, joka lopettaa, kun halutun mittainen alkuosa on järjestyksessä: `partial_sort`($alku$, $keski$, $loppu$)
- lisäksi `is_sorted`($alku$, $loppu$) ja `is_sorted_until`($alku$, $loppu$)

`nth_element(` eka, $\ddot{a}nn\ddot{a}s$, $loppu$ `)`

- etsii alkion, joka järjestetyssä säiliössä olisi kohdalla $\ddot{a}nn\ddot{a}s$
- muistuttaa algoritmia RANDOMIZED-SELECT
- iteraattoreiden tulee olla hajasaanti-iteraattoreita

Ositus (partitiointi)

- `partition(`eka`,` $loppu$`,` $ehtofunktio$`)` epävakaa, erikseen `stable_partition`.
- `stable_partition(`eka`,` $loppu$`,` $ehtofunktio$`)` vakaa, mutta hitaampi ja/tai varaa enemmän muistia
- järjestää välillä eka - $loppu$ olevat alkiot siten, että ensin tulevat alkiot, joille $ehtofunktio$ palauttaa true ja sitten, ne joille se palauttaa false.
- vrt. QUICK-SORTn yhteudessä esitelty PARTITION
- partition on tehokkuudeltaan lineaarinen
- lisäksi `is_partitioned` ja `partition_point`

`merge(` $alku1$ `,` $loppu1$ `,` $alku2$ `,` $loppu2$ `,` $maali$`)`

- Algoritmi limittää välien $alku1$ - $loppu1$ ja $alku2$ - $loppu2$ alkiot ja kopioi ne suuruusjärjestyksessä iteraattorin $maali$ päähän
- algoritmi edellyttää, että alkiot yhdistettävillä väleillä ovat järjestyksessä
- vertaa sivun 45 MERGE
- algoritmi on lineaarinen
- $alku$- ja $loppu$-iteraattorit ovat syöttöiteraattoreita ja $maali$ on tulostusiteraattori

Keot

- STL:stä löytyy myös vastineet luvun 3.1 kekoalgoritmeille
- `push_heap(` eka `,` $loppu$`)` HEAP-INSERT
- `pop_heap(` eka `,` $loppu$ `)` vaihtaa huippualkion viimeiseksi (eli paikkaan $loppu - 1$) ja ajaa HEAPIFY:n osalle $eka \ldots loppu - 1$
  - vrt. HEAP-EXTRACT-MAX
- `make_heap(` eka `,` $loppu$`)` BUILD-HEAP

- `sort_heap(` eka , $loppu$ `)` HEAPSORT
- lisäksi `is_heap` ja `is_heap_until`
- iteraattoreiden tulee olla hajasaanti-iteraattoreita

Joukko-operaatiot

- C++:n standardikirjasto sisältää tätä tukevia funktioita
- `includes(` $eka1$, $loppu1$, $eka2$, $loppu2$ `)` osajoukko $\subseteq$
- `set_union(` $eka1$, $loppu1$, $eka2$, $loppu2$, $tulos$ `)` unioni $\cup$
- `set_intersection(...)` leikkaus $\cap$
- `set_difference(...)` erotus -
- `set_symmetric_difference(...)`
- $alku$- ja $loppu$-iteraattorit ovat syöttöiteraattoreita ja $tulos$ on tulostusiteraattori

`find_first_of(` $eka1$, $loppu1$, $eka2$, $loppu2$ `)`

- lopussa voi lisäksi olla tutkittavia alkioita rajaava ehto

- etsii ensimmäisestä jonosta ensimmäisen alkion, joka on myös toisessa jonossa
- jono voi olla taulukko, lista, joukko, . . .
  - yksinkertainen toteutus on hitaimmillaan $\Theta(nm)$, missä n ja m ovat jonojen pituudet
- toinen jono selataan jokaiselle ensimmäisen jonon alkiolle
  – hitain tapaus kun ei löydy
  $\Rightarrow$ hidasta, jos molemmat jonot pitkiä
- toteutus saataisiin yksinkertaiseksi, nopeaksi ja muistia säästäväksi vaatimalla, että jonot ovat järjestyksessä

HUOM! Mikään STL:n algoritmi ei automaattisesti tee säiliöihin lisäyksiä eikä poistoja, vaan ainoastaan muokkaa olemassa olevia alkioita.

- esimerkiksi `merge` ei toimi, jos sille annetaan tulostusiteraattoriksi iteraattori tyhjän säiliön alkuun
- jos tulostusiteraattorin halutaan tekevän lisäyksiä kopioinnin sijasta, tulee käyttää iteraattorisovitinta lisäysiteraattori

7.5 Lambdat: ()(){}

Algoritmikirjaston yhteydessä on paljon tilanteita, joissa on tarve välittää funktiolle toiminnallisuutta

- esim. `find_if`, `for_each`, `sort`

Lambdat ovat nimettömiä, määrittelemättömän tyyppisiä funktion kaltaisia. Ne ottavat parametreja, palauttavat paluuarvon ja pystyvät viittaamaan luontiympäristönstä muuttujiin sekä muuttamaan niitä.

Syntaksi: `[ympäristö](parametrit)->paluutyyppi {runko}`

- Jos lambda ei viittaa ympäristöönsä `ympäristö` on tyhjä
- `parametrit` voi puuttua
- jos `->paluutyyppi`ä ei ole annettu, se on void. Yksittäisestä return-lauseesta se voidaan päätellä
- esim. `[](int x, int y){ return x+y;}`

```
for_each( v.begin(), v.end(),[] (int val) {cout<<val<<endl;});
std::cin >> raja; //paikallinen muuttuja
std:find_if(v.begin(), v.end(),[raja](int a){return a<raja;});
```

STL:n algoritmeja voi ajatella nimettyinä erityissilmukoina, joiden runko lambda on

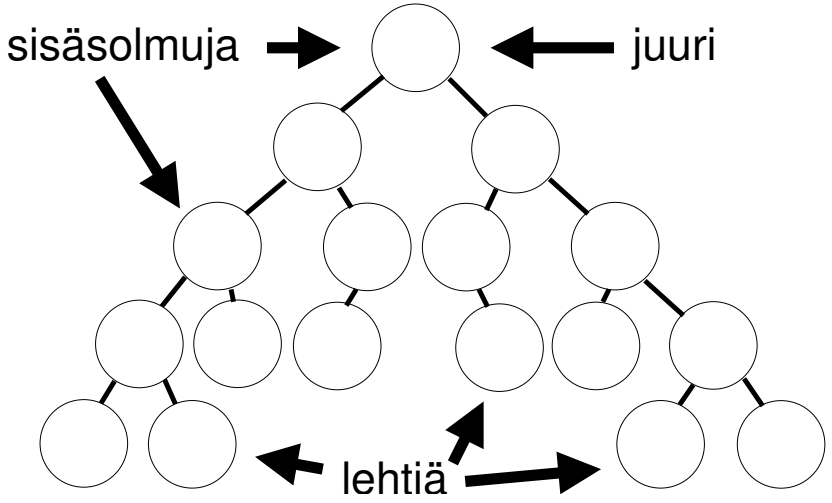
```
bool kaikki = true;
for (auto i : v)
{
    if (i%10 != 0) {
        kaikki = false;
        break;
    }
}
if (kaikki) {...}

if (std::all_of(v.begin(), v.end(), [](int i){return i%10==0;}){...}
```

8.1 Puu

Ennen kuin käydään käsiksi kekoon, määritellään sen tueksi käsite *puu*.

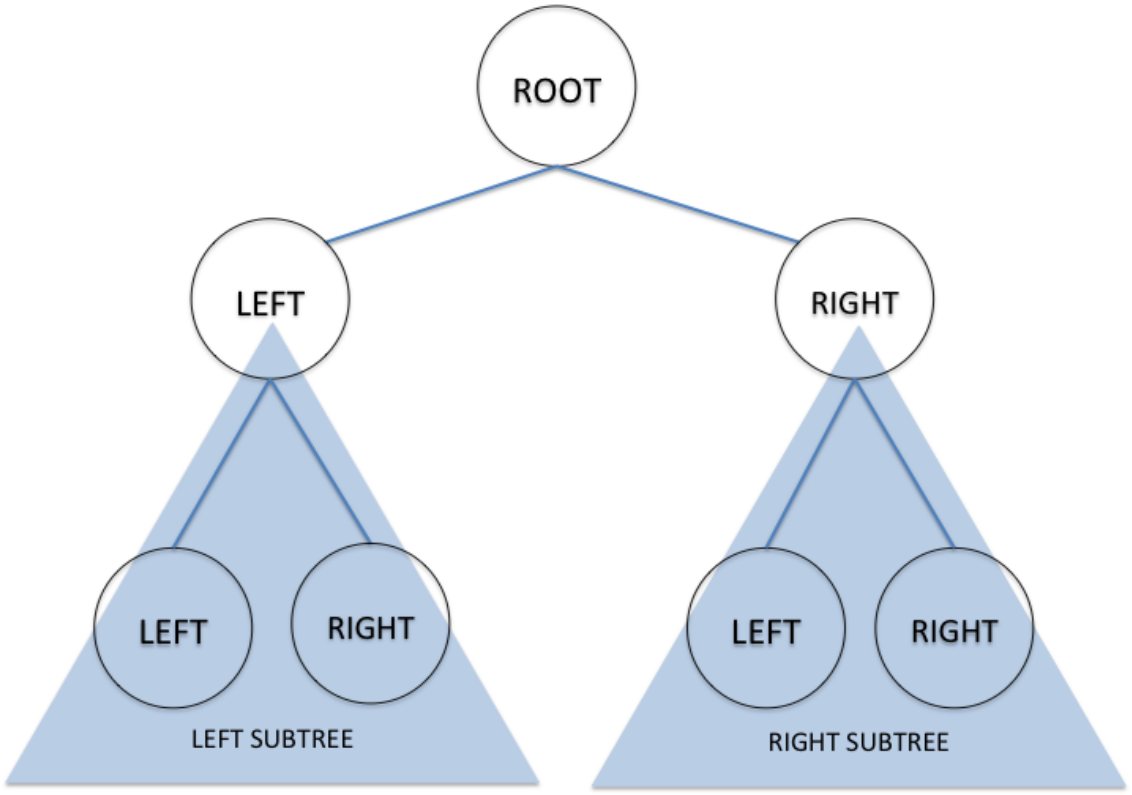
Puu on:



- rakenne, joka koostuu solmuista, joilla on mielivaltainen määrä lapsia.

- *Binääripuussa* lasten määrä on rajoitettu välille 0–2. Tällöin lapset nimetään *vasen (left)* ja *oikea (right)*
- solmu on lapsiensa *isä (parent)*
- lapseton solmu on *lehti (leaf)*, ja muut solmut ovat *sisäsolmuja (internal node)*
- puussa on korkeintaan yksi solmu, jolla ei ole isää. Isätön solmu on puun *juuri (root)*.
  - kaikki muut solmut ovat juuren lapsia, lastenlapsia jne.

- puun rakenne on rekursiivinen: kunkin solmun jälkeläiset muodostavat puun *alipuun*, jonka juuri kyseinen solmu on



Kuva 13: Binääripuun rekursiivisuus

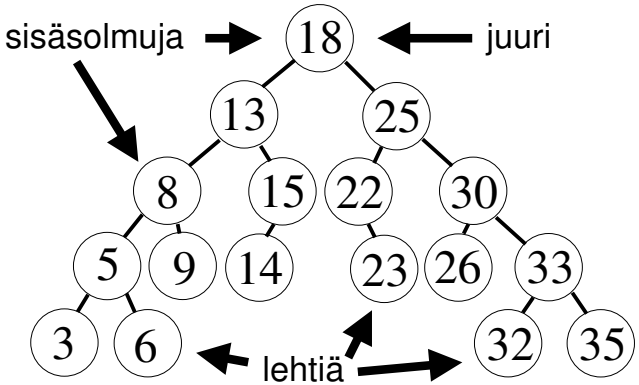
- puun solmun *korkeus (height)* on pisimmän solmusta

suoraan alas lehteen vievän polun pituus
  - pituus lasketaan kaarien mukaan, jolloin lehden korkeus on 0
- puun korkeus on sen juuren korkeus
- puu on *täydellisesti tasapainotettu (completely balanced)*, jos sen juuren lasten määräämien alipuiden korkeudet eroavat toisistaan enintään yhdellä, ja alipuut on täydellisesti tasapainotettu
- n-solmuisen puun korkeus on vähintään $\lfloor \lg n \rfloor$ ja korkeintaan n - 1 (logaritmin kantaluku riippuu lasten maksimimäärästä)
  $\Rightarrow O(n)$ ja $\Omega(\lg n)$

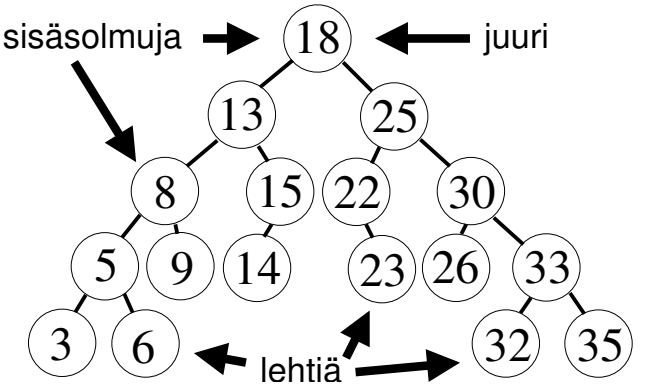
Puun solmut voidaan käsitellä monessa eri järjestyksessä.

- *esijärjestys* (*preorder*) eli ensin käsitellään juuri, sitten rekursiivisesti lapset.
  - kutsu: PREORDER-TREE-WALK$(T.root)$
  - esimerkin käsittelyjärjestys on 18, 13, 8, 5, 3, 6, 9, 15, 14, 25, 22, 23, 30, 26, 33, 32, 35

```
PREORDER-TREE-WALK(x)
1   if x ≠ NIL then
2       käsittele alkio x
3       for child in x→children do
4           PREORDER-TREE-WALK(child)
```



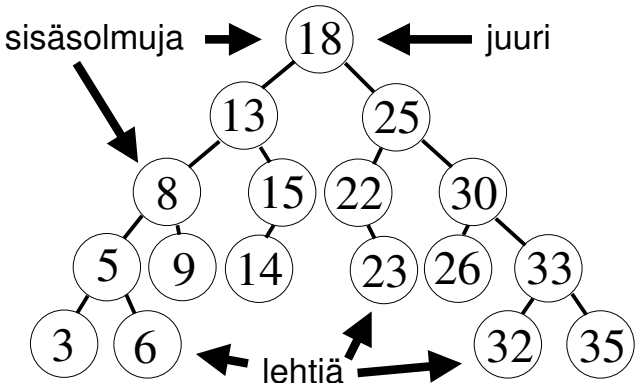
- *välijärjestys* (*inorder*)
  - välijärjestys koskee lähinnä *binääripuuta*, siinä käsitellään ensin rekursiivisesti vasen lapsi, sitten juuri ja lopuksi rekursiivisesti oikea lapsi
  - esimerkissä 3, 5, 6, 8, 9, 13, 14, 15, 18, 22, 23, 25, 26, 30, 32, 33, 35



INORDER-TREE-WALK(x)
1 **if** $x \neq$ NIL **then**
2 INORDER-TREE-WALK($x \rightarrow left$)
3 käsittele alkio x
4 INORDER-TREE-WALK($x \rightarrow right$)

- *jälkijärjestys* (*postorder*), eli ensin käsitellään rekursiivisesti lapset, lopuksi vasta juuri
  - esimerkissä 3, 6, 5, 9, 8, 14, 15, 13, 23, 22, 26, 32, 35, 33, 30, 25, 18

```
POSTORDER-TREE-WALK(x)
1  if x ≠ NIL then
2      for child in x→children do
3          POSTORDER-TREE-WALK(child)
4      käsittele alkio x
```



Puun läpikäynnin ajankäyttö:

- ajoaika $\Theta(n)$, algoritmit kutsuvat itseään kahdesti joka solmussa: kerran vasemmalle ja kerran oikealle lapselle
- lisämuistin tarve = Θ(rekursion maksimisyvyys) = $\Theta(h+1)$ = $\Theta(h)$

Tasattu eli amortisoitu ajoaika

Vektori on joustavarajainen taulukko eli sen kokoa kasvatetaan tarvittaessa.

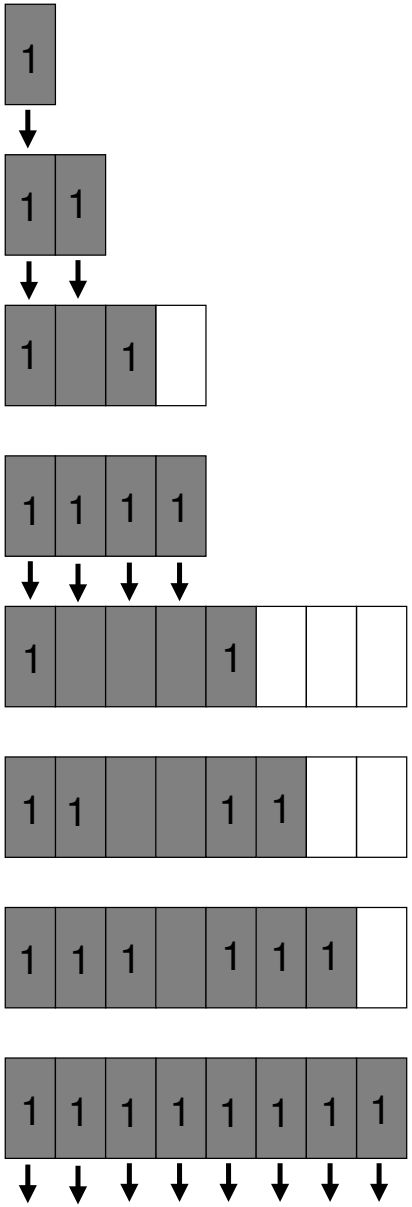
- kun uusi alkio ei enää mahdu taulukkoon, varataan uusi suurempi ja siirretään kaikki alkiot sinne
- taulukko ei koskaan kutistu

  $\Rightarrow$ muistin varaus ei vähene muuten kuin kopioimalla vektoriin kokonaan uusi sisältö

$\Rightarrow$ Alkion lisäämisen vectorin loppuun sanottiin olevan *tasatusti (amortisoidusti)* vakioaikaista.

- Amortisoidusti lähestyttäessä suoritusaikaa tarkastellaan kokonaisuutena, tutkitaan operaatiosarjojen suoritusaikaa yksittäisten operaatioiden sijaan
  - jokaista kallista muistinvarausta vaativaa lisäysoperaatiota edeltää kalliin operaation hintaan suoraan verrannollinen määrä halpoja lisäysoperaatioita
  - kalliin operaation kustannus voidaan jakaa tasan halvoille operaatioille
  - tällöin halvat operaatiot ovat edelleen vakioaikaisia, tosin vakiokertoimen verran hitaampia kuin oikeasti
  - kallis operaatio voidaan maksaa säästöillä

$\Rightarrow$ kaikki lisäysoperaatiot vektorin loppuun ovat tasatusti vakioaikaisia



Tämä voidaan todentaa vaikkapa kirjanpitomenetelmällä:

- laskutetaan jokaisesta lisäyksestä kolme rahaa
- yksi raha käytetään lisäyksen todellisiin kustannuksiin
- yksi raha laitetaan säästöön lisätyn alkion i kohdalle
- yksi raha laitetaan säästöön alkion $i - \frac{1}{2} \cdot vector.capacity()$ kohdalle
- kun tulee tarve laajentaa taulukkoa, jokaisella alkiolla on yksi raha säästössä, ja kallis kopiointi voidaan maksaa niillä

6.2 Suunnitteluperiaate: Satunnaistaminen

Satunnaista on eräs algoritmien suunnitteluperiaatteista.

- Sen avulla voidaan usein estää huonoimpien tapausten patologinen ilmeneminen.
- Parhaan ja huonoimman tapauksen suoritusajat eivät useinkaan muutu, mutta niiden esiintymistodennäköisuus käytännössä laskee.
- Huonot syötteet ovat täsmälleen yhtä todennäköisiä kuin mitkä tahansa muut syötteet riippumatta alkuperäisestä syötteiden jakaumasta.
- Satunnaistaminen voidaan suorittaa joko ennen algoritmin suoritusta satunnaistamalla sen saama syöteaineisto tai upottamalla satunnaistaminen algoritmin sisälle.
  - jälkimmäisellä tavalla päästään usein parempaan tulokseen
  - usein se on myös helpompaa kuin syötteen esikäsittely

- Satunnaistaminen on hyvä ratkaisu yleensä silloin, kun
  - algoritmi voi jatkaa suoritustaan monella tavalla
  - on vaikea arvata etukäteen, mikä tapa on hyvä
  - suuri osa tavoista on hyviä
  - muutama huono arvaus hyvien joukossa ei haittaa paljoa
- Esimerkiksi QUICKSORT voi valita jakoarvoksi minkä tahansa taulukon alkion
  - hyviä valintoja ovat kaikki muut, paitsi lähes pienimmät ja lähes suurimmat taulukossa olevat arvot
  - on vaikea arvata valintaa tehdessä, onko ko. arvo lähes pienin / suurin
  - muutama huono arvaus silloin tällöin ei turmele QUICKSORTin suorituskykyä

  $\Rightarrow$ satunnaistaminen sopii QUICKSORTille

Satunnaistamisen avulla voidaan tuottaa algoritmi RANDOMIZED-QUICKSORT, joka käyttää satunnaistettua PARTITIONIA.

- Ei valita jakoarvoksi aina $A[\, right\,]$:tä, vaan valitaan jakoarvo satunnaisesti koko osataulukosta.
- Jotta PARTITION ei menisi rikki, sijoitetaan jakoarvo silti kohtaan $right$ taulukkoa

  $\Rightarrow$ Nyt jako on todennäköisesti melko tasainen riippumatta siitä, mikä syöte saatiin ja mitä taulukolle on jo ehditty tehdä.

```
RANDOMIZED-PARTITION( A, left, right )
1   p := RANDOM(left, right)          (valitaan satunnainen alkio pivotiksi)
2   exchange A[ right ] ↔ A[ p ]      (asetetaan se taulukon viimeiseksi)
3   return PARTITION( A, left, right )  (kutsutaan tavallista partitiointia)
```

```
RANDOMIZED-QUICKSORT( A, left, right )
1   if left < right then
2       p := RANDOMIZED-PARTITION( A, left, right )
3       RANDOMIZED-QUICKSORT( A, left, p − 1 )
4       RANDOMIZED-QUICKSORT( A, p + 1, right )
```

RANDOMIZED-QUICKSORTIN ajoaika on keskimäärin $\Theta(n \lg n)$
samoin kuin tavallisenkin QUICKSORTIN.

- RANDOMIZED-QUICKSORTILLE kuitenkin varmasti pätee keskimääräisen ajankäytön analyysin yhteydessä tehtävä oletus, jonka mukaan pivot-alkio on osataulukon pienin, toiseksi pienin jne. aina samalla todennäköisyydellä.
- Tavalliselle QUICKSORTILLE tämä pätee ainoastaan, jos aineisto on tasaisesti jakautunutta.

$\Rightarrow$ RANDOMIZED-QUICKSORT on yleisessä tapauksessa tavallista QUICKSORTIA parempi.

QUICKSORTIA voidaan tehostaa myös muilla keinoilla:

- Voidaan järjestää pienet osataulukot pienille taulukoille tehokkaalla algoritmilla (esim. INSERTIONSORT) avulla.
  - voidaan myös jättää ne vain järjestämättä ja järjestää taulukko lopuksi INSERTIONSORTIN avulla
- Jakoarvo voidaan valita esimerkiksi kolmen satunnaisesti valitun alkion mediaanina.
- On jopa mahdollista käyttää aina mediaanialkiota jakoalkiona.

Mediaani on mahdollista etsiä nopeasti niin sanotun laiskan QUICKSORTIN avulla.

- Jaetaan taulukko "pienten alkioiden" alaosaan ja "suurten alkioiden" yläosaan kuten QUICKSORTissa.
- Lasketaan, kumpaan osaan i:s alkio kuuluu, ja jatketaan rekursiivisesti sieltä.
- Toiselle osalle ei tarvitse tehdä enää mitään.

```
RANDOMIZED-SELECT( A, left, right, goal )
1   if left = right then                          (jos osataulukko on yhden kokoinen...)
2       return A[ left ]                          (... palautetaan ainoa alkio)
3   p := RANDOMIZED-PARTITION( A, left, right )   (jaetaan taulukko pieniin ja isoihin)
4   k := p − left + 1                             (lasketaan monesko jakoalkio on)
5   if i = k then                                 (jos jakoalkio on taulukon i:s alkio...)
6       return A[ p ]                             (...palautetaan se)
7   else if i < k then                            (jatketaan etsintää pienten puolelta)
8       return RANDOMIZED-SELECT( A, left, p − 1, goal )
9   else                                          (jatketaan etsintää suurten puolelta)
10      return RANDOMIZED-SELECT( A, p + 1, right, goal − k )
```

RANDOMIZED-SELECTIN suoritusajan alaraja:

- Jälleen kaikki muu on vakioaikaista paitsi RANDOMIZED-PARTITION ja rekursiivinen kutsu.
- Parhaassa tapauksessa RANDOMIZED-PARTITIONIN valitsema jakoalkio on taulukon i:s alkio, ja ohjelman suoritus loppuu.
- RANDOMIZED-PARTITION ajetaan kerran koko taulukolle.

$\Rightarrow$ Algoritmin suoritusaika on $\Omega(n)$.

RANDOMIZED-SELECTIN suoritusajan yläraja:

- RANDOMIZED-PARTITION sattuu aina valitsemaan pienimmän tai suurimman alkion, ja i:s alkio jää suuremmalle puoliskolle
- työmäärä pienenee vain yhdellä askeleella joka rekursiotasolla.

$\Rightarrow$ Algoritmin suoritusaika on $O(n^2)$.

Keskimääräisen tapauksen ajoaika on kuitenkin $O(n)$.

Algoritmi löytyy esimerkiksi STL:stä nimellä `nth_element`.

Algoritmi on mahdollista muuttaa myös toimimaan aina lineaarisessa ajassa.

8.2 Suunnitteluperiaate: Muunna ja hallitse

Muunna ja hallitse on suunnitteluperiaate, joka

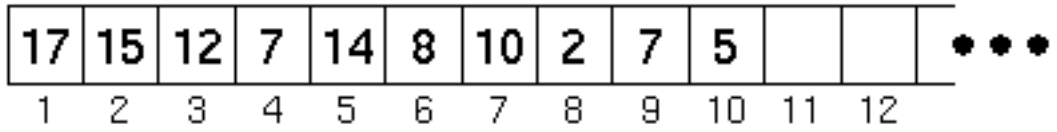
- Ensin muokkaa ongelman instanssia muotoon, joka on helpompi ratkaista – muunnosvaihe
- Sitten ongelma voidaan ratkaista – hallintavaihe

Ongelman instanssi voidaan muuntaa kolmella eri tavalla:

- Yksinkertaistaminen (*Instance simplification*): yksinkertaisempi tai kätevämpi instanssi samasta ongelmasta
- Esitystavan muutos (*Representation change*): saman instanssin toinen esitystapa
- Ongelman muunnos (*Problem reduction*): ratkaistaan sellaisen ongelman, jolle algoritmi on jo valmiina, instanssi

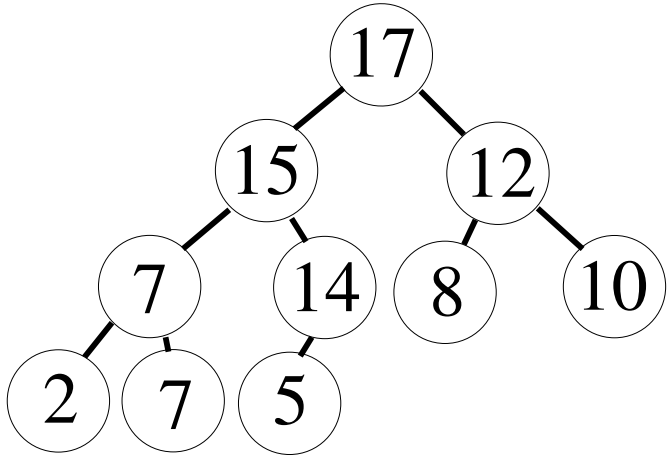
Keko

Taulukko $A[1 \ldots n]$ on *keko*, jos $A[i] \geq A[2i]$ ja $A[i] \geq A[2i+1]$ aina kun $1 \leq i \leq \lfloor \frac{n}{2} \rfloor$ (ja $2i+1 \leq n$).



Tämä on helpompi ymmärtää, kun tulkitaan keko täydellisesti tasapainotetuksi binääripuuksi, jonka

- juuri on talletettu taulukon paikkaan 1
- paikkaan i talletetun solmun lapset (jos olemassa) on talletettu paikkoihin $2i$ ja $2i+1$
- paikkaan i talletetun solmun isä on talletettu paikkaan $\lfloor \frac{i}{2} \rfloor$



Tällöin jokaisen solmun arvo on suurempi tai yhtä suuri kuin sen lasten arvot.

Kekopuun jokainen kerros on täysi, paitsi ehkä alin, joka on täytetty vasemmasta reunasta alkaen.

Jotta kekoa olisi helpompi ajatella puuna, määrittelemme isä- ja lapsisolmut löytävät aliohjelmat.

- ne ovat toteutettavissa hyvin tehokkaasti bittisiirtoina
- kunkin suoritusaika on aina $\Theta(1)$

PARENT(i)
    return $\lfloor i/2 \rfloor$

LEFT(i)
    return $2i$

RIGHT(i)
    return $2i + 1$

$\Rightarrow$ Nyt keko-ominaisuus voidaan lausua seuraavasti:

$A[\text{PARENT}(i)] \geq A[i]$ *aina kun* $2 \leq i \leq A.heapsize$

- $A.heapsize$ kertoo keon koon (myöhemmin nähdään, ettei se aina ole välttämättä sama kuin taulukon koko)

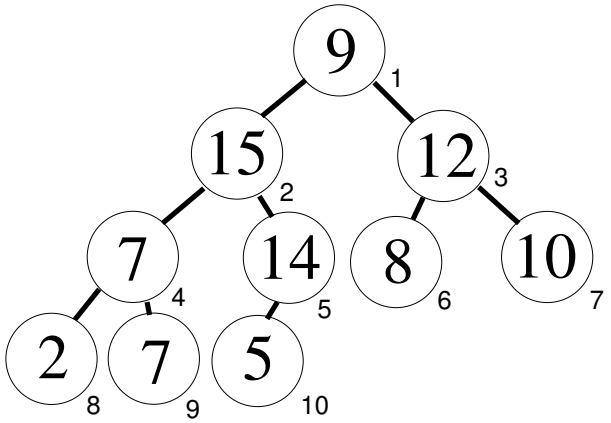
Keko-ominaisuudesta seuraa, että keon suurin alkio on aina keon juuressa, siis taulukon ensimmäisessa lokerossa.

Jos keon korkeus on h, sen solmujen määrä on välillä $2^h \ldots 2^{h+1} - 1$.

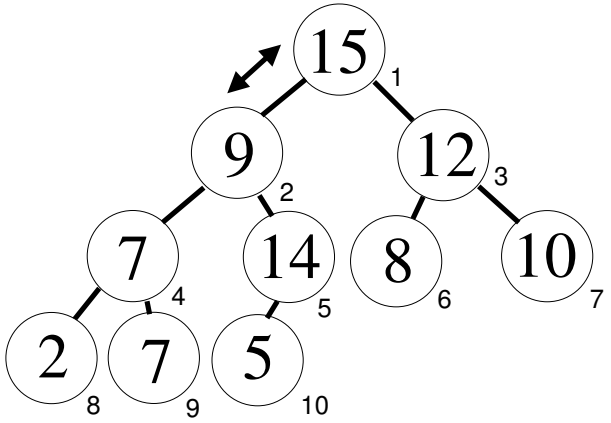
$\Rightarrow$ Jos keossa n solmua, sen korkeus on $\Theta(\lg n)$.

Alkion lisääminen kekoon ylhäältä:

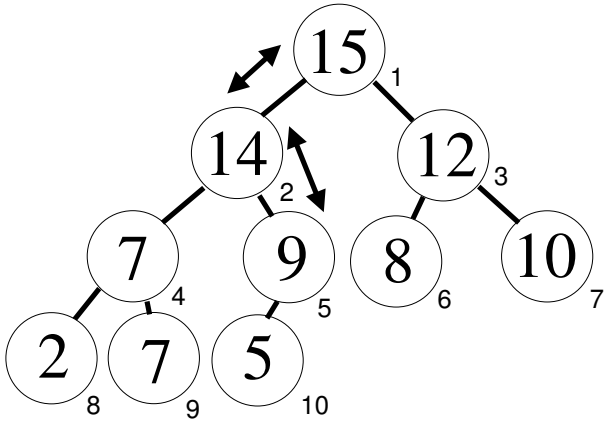
- oletetaan, että $A[1 \ldots n]$ on muuten keko, mutta keko-ominaisuus ei päde kekopuun juurelle
  - toisin sanoen $A[1] < A[2]$ tai $A[1] < A[3]$



- ongelma saadaan siirrettyä alemmas puussa valitsemalla juuren lapsista suurempi, ja vaihtamalla se juuren kanssa
  - jotta keko-ominaisuus ei hajoaisi, pitää valita lapsista suurempi - siitähän tulee toisen lapsen uusi isä



- sama voidaan tehdä alipuulle, jonka juureen ongelma siirtyi, ja sen alipuulle jne. kunnes ongelma katoaa
  - ongelma katoaa viimeistään kun saavutetaan lehti
    $\Rightarrow$ puu muuttuu keoksi



Sama pseudokoodina:

```
HEAPIFY( A, i )                    (i kertoo paikan, jonka alkio saattaa olla liian pieni)
1  repeat                                                  (toistetaan, kunnes keko on ehjä)
2      old_i := i                                          (otetaan i:n arvo talteen)
3      l := LEFT( i )
4      r := RIGHT( i )
5      if l ≤ A.heapsize and A[ l ] > A[ i ] then          (vasen lapsi on suurempi kuin i)
6          i := l
7      if r ≤ A.heapsize and A[ r ] > A[ i ] then          (oikea lapsi on vielä suurempi)
8          i := r
9      if i ≠ old_i then                                   (jos suurempi lapsi löytyi...)
10         exchange A[ old_i ] ↔ A[ i ]                    (...siirretään rike alaspäin)
11 until i = old_i                                         (jos keko oli jo ehjä, lopetetaan)
```

- Suoritus on vakioaikaista kun rivin 11 ehto toteutuu heti ensimmäisellä kerralla kun sinne päädytään: $\Omega(1)$.
- Pahimmassa tapauksessa uusi alkio joudutaan siirtämään lehteen asti koko korkeuden verran.
  $\Rightarrow$ Suoritusaika on $O(h) = O(\lg n)$.

Keon rakentaminen

- seuraava algoritmi järjestää taulukon uudelleen niin, että siitä tulee keko:

```
BUILD-HEAP(A)
1  A.heapsize := A.length                  (koko taulukosta tehdään keko)
2  for i := ⌊A.length/2⌋ downto 1 do       (käydään taulukon alkupuolisko läpi)
3      HEAPIFY(A, i)                       (kutsutaan Heapifyta)
```

- Lähdetään käymään taulukkoa läpi lopusta käsin ja kutsutaan HEAPIFYTA kaikille alkioille.
  - ennen HEAPIFY-funktion kutsua keko-ominaisuus pätee aina i:n määräämälle alipuulle, paitsi että paikan i alkio on mahdollisesti liian pieni
  - yhden kokoisia alipuita ei tarvitse korjata, koska niissä keko-ominaisuus pätee triviaalisti
  - HEAPIFY(A, i):n jälkeen i:n määräämä alipuu on keko

  $\Rightarrow$ HEAPIFY($A, 1$):n jälkeen koko taulukko on keko

- BUILD-HEAP ajaa **for**-silmukan $\lfloor \frac{n}{2} \rfloor$ kertaa ja HEAPIFY on $\Omega(1)$ ja $O(\lg n)$, joten
  - nopein suoritusaika on $\lfloor \frac{n}{2} \rfloor \cdot \Omega(1) + \Theta(n) = \Omega(n)$
  - ohjelma ei voi koskaan käyttää enempää aikaa kuin $\lfloor \frac{n}{2} \rfloor \cdot O(\lg n) + \Theta(n) = O(n \lg n)$
- Näin saamamme hitaimman tapauksen suoritusaika on kuitenkin liian pessimistinen:

- HEAPIFY on $O(h)$, missä h on kekopuun korkeus
- i:n muuttuessa myös puun korkeus vaihtelee

kerros	h	HEAPIFY-suoritusten määrä
alin	0	0
toinen	1	$\lfloor \frac{n}{4} \rfloor$
kolmas	2	$\lfloor \frac{n}{8} \rfloor$
...	...	...
ylin	$\lfloor \lg n \rfloor$	1

- siis pahimman tapauksen suoritusaika onkin
  $\frac{n}{4} \cdot 1 + \frac{n}{8} \cdot 2 + \frac{n}{16} \cdot 3 + \cdots = \frac{n}{2} \cdot \sum_{i=1}^{\infty} \frac{i}{2^i} = \frac{n}{2} \cdot 2 = n \Rightarrow O(n)$

$\Rightarrow$ BUILD-HEAPIN suoritusaika on aina $\Theta(n)$

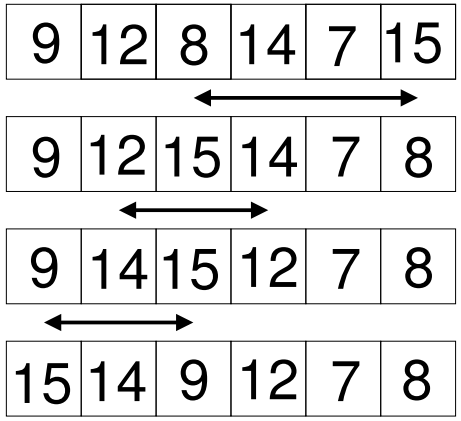
8.3 Järjestäminen keon avulla

Taulukon alkioiden ärjestäminen voidaan toteuttaa tehokkaasti kekoa hyödyntäen:

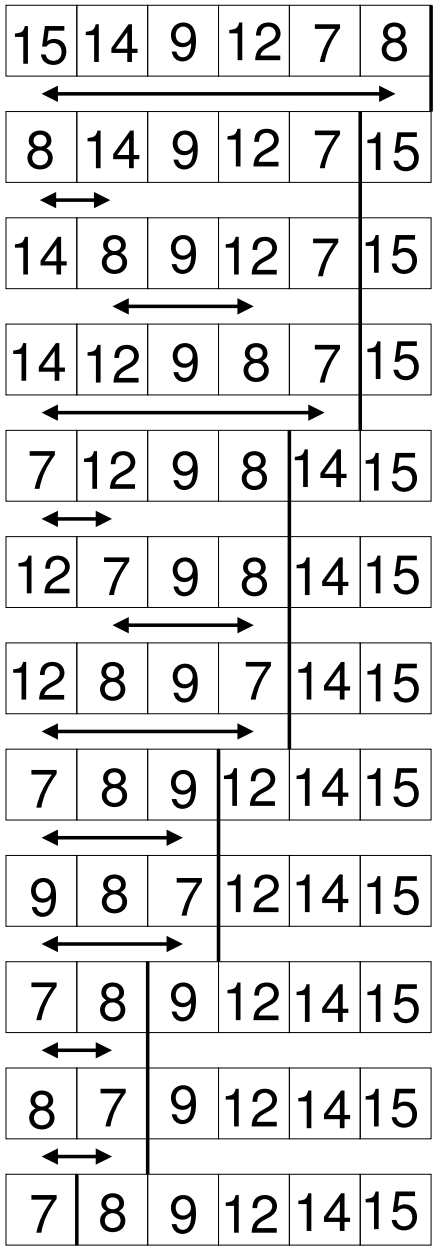
```
HEAPSORT( A )
1  BUILD-HEAP( A )                    (muutetaan taulukko keoksi)
2  for i := A.length downto 2 do      (käydään taulukko läpi lopusta alkuun)
3      exchange A[ 1 ] ↔ A[ i ]       (siirretään keon suurin alkio keon viimeiseksi)
4      A.heapsize := A.heapsize − 1   (siirretään suurin alkio keon ulkopuolelle)
5      HEAPIFY( A, 1 )                (korjataan keko, joka on muuten kunnossa, mutta...)
                                      (... sen ensimmäinen alkio saattaa olla liian pieni)
```

Esitetäänpä sama kuvien avulla:

- ensin taulukko muutetaan keoksi
- esimerkistä on helppo havaita, ettei operaatio ole kovin raskas
  - keko-ominaisuus on selvästi järjestystä heikompi



- kuvassa voi nähdä kuinka järjestetyn loppuosan koko kasvaa, kunnes koko taulukko on järjestyksessä
- kasvatusten välillä keko-osuus korjataan
- korjaus näyttää tällaisessa pienessä esimerkissä tarpeettoman monimutkaiselta
  - korjaukseen ei suurillakaan taulukoilla kulu kovin montaa askelta, ainoastaan logaritminen määrä



HEAPSORTIN suoritusaika koostuu seuraavista osista.

- BUILD-HEAP rivillä 1 suoritetaan kerran: $\Theta(n)$
- **for**-silmukan sisältö suoritetaan n - 1 kertaa
  - rivien 3 ja 4 operaatiot ovat vakioaikaisia
  - HEAPIFY käyttää aikaa $\Omega(1)$ ja $O(\lg n)$

  $\Rightarrow$ Saadaan yhteensä $\Omega(n)$ ja $O(n \lg n)$
- alaraja on tarkka
  - jos kaikki alkiot ovat samanarvoisia, keon korjaustoimenpiteitä ei tarvita koskaan ja HEAPIFY on aina vakioaikainen
- myös yläraja on tarkka
  - tämän osoittaminen on hieman hankalampaa ja tyydymmekin myöhemmin saatavaan tulokseen vertailuun perustuvan järjestämisen nopeudesta

Huom! Edelliset suoritusaikalaskelmat olettavat, että keon pohjana käytettävällä tietorakenteella on vakioaikainen indeksointi.

- Kekoa kannattaa käyttää ainoastaan silloin!

HEAPSORTIN etuja ja haittoja

Etuja:

- järjestää taulukon paikallaan
- ei koskaan käytä enempää kuin $\Theta(n \lg n)$ aikaa

Haittoja:

- suoritusajan vakiokerroin on suurehko
- epävakaus
  - samanarvoisten alkioiden keskinäinen järjestys ei säily

8.4 Prioriteettijono

Prioriteettijono (*priority queue*) on tietorakenne, joka pitää yllä joukkoa S alkioita, joista jokaiseen liittyy *avain* (*key*), ja sallii seuraavat operaatiot:

- INSERT(S, x) lisää alkion x joukkoon S
- MAXIMUM(S) palauttaa sen alkion, jonka avain on suurin
  - jos monella eri alkiolla on sama, suurin avain, valitsee vapaasti minkä tahansa niistä
- EXTRACT-MAX(S) poistaa ja palauttaa sen alkion, jonka avain on suurin
- vaihtoehtoisesti voidaan toteuttaa operaatiot MINIMUM(S) ja EXTRACT-MIN(S)
  - samassa jonossa on joko **vain maksimi-** tai **vain minimioperaatiot**!

Prioriteettijonoilla on monia käyttökohteita

- tehtävien ajoitus käyttöjärjestelmässä
  - uusia tehtäviä lisätään komennolla INSERT
  - kun edellinen tehtävä valmistuu tai keskeytetään, seuraava valitaan komennolla EXTRACT-MAX
- tapahtumapohjainen simulointi
  - jono tallettaa tulevia (= vielä simuloimattomia) tapahtumia
  - avain on tapahtuman tapahtumisaika
  - tapahtuma voi aiheuttaa uusia tapahtumia
    $\Rightarrow$ lisätään jonoon operaatiolla INSERT
  - EXTRACT-MIN antaa seuraavan simuloitavan tapahtuman
- lyhimmän reitin etsintä kartalta
  - simuloidaan vakionopeudella ajavia, eri reitit valitsevia autoja, kunnes ensimmäinen perillä
  - prioriteettijonoa tarvitaan käytännössä myöhemmin esiteltävässä lyhimpien polkujen etsintäalgoritmissa

Prioriteettijonon voisi käytännössä toteuttaa järjestämättömänä tai järjestettynä taulukkona, mutta se olisi tehotonta.

- järjestämättömässä taulukossa MAXIMUM ja EXTRACT-MAX ovat hitaita
- järjestetyssä taulukossa INSERT on hidas

Sen sijaan keon avulla prioriteettijonon voi toteuttaa tehokkaasti.

- Joukon S alkiot talletetaan kekoon A.
- MAXIMUM$(\ S\ )$ on hyvin helppo, ja toimii ajassa $\Theta(1)$.

```
HEAP-MAXIMUM( A )
1  if A.heapsize < 1 then          (tyhjästä keosta ei löydy maksimia)
2      error "heap underflow"
3  return A[ 1 ]                   (muuten palautetaan taulukon ensimmäinen alkio)
```

- EXTRACT-MAX(S) voidaan toteuttaa korjaamalla keko poiston jälkeen HEAPIFYN avulla.
- HEAPIFY dominoi algoritmin ajoaikaa: $O(\lg n)$.

```
HEAP-EXTRACT-MAX( A )
1  if A.heapsize < 1 then          (tyhjästä keosta ei löydy maksimia)
2      error "heap underflow"
3  max := A[ 1 ]                   (suurin alkio löytyy taulukon alusta)
4  A[ 1 ] := A[ A.heapsize ]       (siirretään viimeinen alkio juureen)
5  A.heapsize := A.heapsize - 1    (pienennetään keon kokoa)
6  HEAPIFY( A, 1 )                 (korjataan keko)
7  return max
```

- INSERT(S, x) lisää uuden alkion kekoon asettamalla sen uudeksi lehdeksi, ja nostamalla sen suuruutensa mukaiselle paikalle.
  - se toimii kuten HEAPIFY, mutta alhaalta ylöspäin
  - lehti joudutaan nostamaan pahimmassa tapauksessa juureen asti: ajoaika $O(\lg n)$

```
HEAP-INSERT( A, key )
1   A.heapsize := A.heapsize + 1          (kasvatetaan keon kokoa)
2   i := A.heapsize                       (lähdetään liikkeelle taulukon lopusta)
3   while i > 1 and A[ PARENT(i) ] < key do (edetään kunnes ollaan juuressa tai ...)
                                          (...kohdassa jonka isä on avainta suurempi)
4       A[ i ] := A[ PARENT(i) ]          (siirretään isää alas päin)
5       i := PARENT(i)                    (siirrytään ylöspäin)
6   A[ i ] := key                         (asetetaan avain oikealle paikalleen)
```

$\Rightarrow$ Keon avulla saadaan jokainen prioriteettijonon operaatio toimimaan ajassa $O(\lg n)$.

Prioriteettijonoa voidaan ajatella abstraktina tietotyyppinä, johon kuuluu talletettu data (joukko S) ja operaatiot (INSERT, MAXIMUM,EXTRACT-MAX.

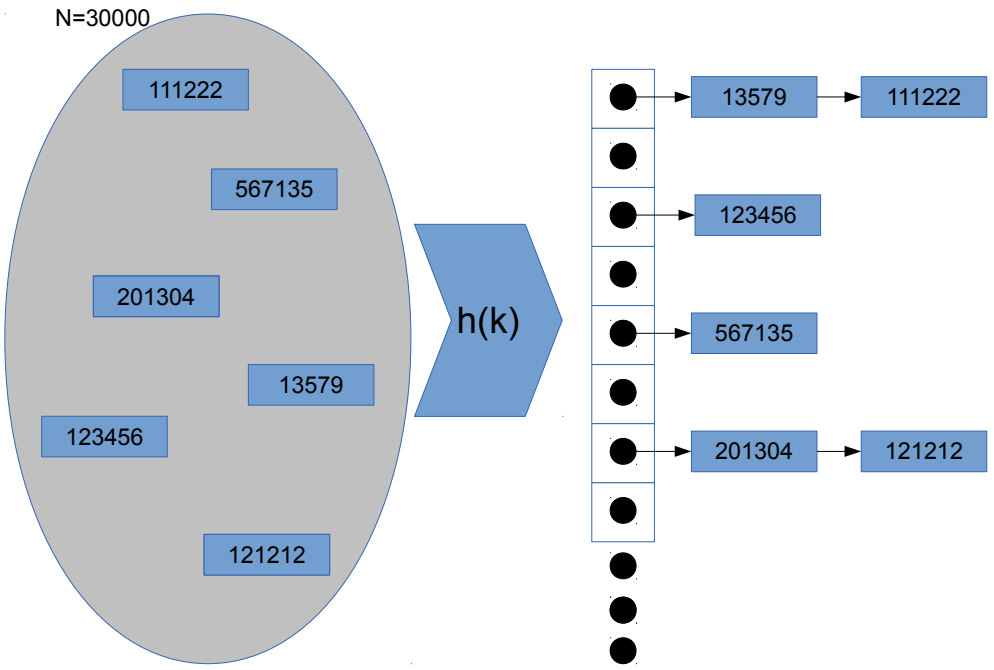
- käyttäjälle kerrotaan ainoastaan operaatioiden nimet ja merkitykset, muttei toteutusta
- toteutus kapseloidaan esimerkiksi pakkaukseksi (Ada), luokaksi (C++) tai itsenäiseksi tiedostoksi (C)

$\Rightarrow$ Toteutusta on helppo ylläpitää, korjata ja tarvittaessa vaihtaa toiseen, ilman että käyttäjien koodiin tarvitsee koskea.

11.1 Hajautustaulu

Hajautustaulun ideana on tiivistää dynaamisen joukon avainten arvoalue pienemmäksi *hajautusfunktion* (*hash function*) h avulla, siten että ne voidaan tallettaa taulukkoon.

- taulukon etuna on sen tarjoama tehokas vakioaikainen indeksointi



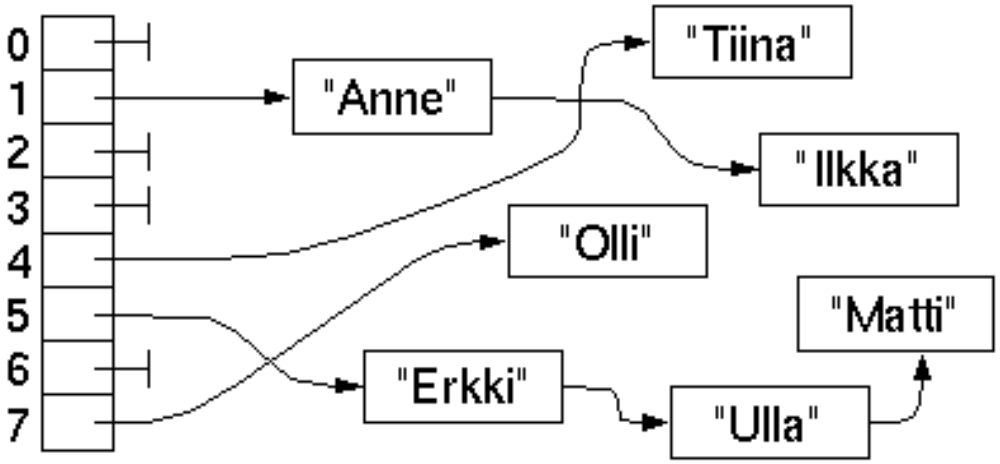
Avainten arvoalueen tiivistämisestä seuraa kuitenkin ongelma: *törmäykset*.

- useampi kuin yksi alkio voi hajautua samaan hajautustaulun lokeroon

Tavallisin tapa ratkaista ongelma on *ketjuttaminen (chaining)*.

- samaan lokeroon hajautuvat alkiot talletetaan listoihin
- muitakin ratkaisutapoja on
  - *avoimen osoituksen* käsittelytavalla alkio laitetaan sekundääriseen lokeroon, mikäli primäärinen ei ole vapaana
  - joissakin tapauksissa avainten arvoalue on niin pieni, että arvoalueen tiivistämistä ei tarvita, eikä siis synny törmäyksiäkään
    - tällainen *suoraosoitustaulu (direct-access table)* on hyvin yksinkertainen ja tehokas
  - tällä kurssilla kuitenkin käsitellään ainoastaan ketjutettuja hajautustauluja

Alla oleva kuva esittää ketjutettua hajautustaulua, jonka avaimet on hajautettu etukirjaimen mukaan viereisen taulukon avulla.



$h(k)$	alkukirjain			
0		H	P	X
1	A	I	Q	Y
2	B	J	R	Z
3	C	K	S	Ä
4	D	L	T	Ö
5	E	M	U	Å
6	F	N	V	
7	G	O	W	

Onko tämä hyvä hajautus?

- Ei. Katsotaan seuraavaksi, miksei.

Ketjutettu hajautustaulu tarjoaa ainoastaan *sanakirjan* operaatiot, mutta ne ovat hyvin yksinkertaisia:

CHAINED-HASH-SEARCH(T, k)
▷ etsi listasta $T[h(k)]$ alkio, jonka avain on k

CHAINED-HASH-INSERT(T, x)
▷ lisää x listan $T[h(x{\rightarrow}key)]$ alkuun

CHAINED-HASH-DELETE(T, x)
▷ poista x listasta $T[h(x{\rightarrow}key)]$

Suoritusajat:

- lisäys: $\Theta(1)$
- etsintä: hitaimmassa tapauksessa $\Theta(n)$
- poisto: jos lista kaksisuuntainen, niin $\Theta(1)$; yksisuuntaisella hitaimmillaan $\Theta(n)$, koska poistettavan edeltäjä on ensin etsittävä listasta
  - käytännössä ero ei kuitenkaan ole kovin merkittävä, koska yleensä poistettava alkio joudutaan joka tapauksessa etsimään listasta

Ketjutetun hajautustaulun operaatioiden *keskimääräiset* suoritusajat riippuvat listojen pituuksista.

- huonoimmassa tapauksessa kaikki alkiot joutuvat samaan listaan jolloin suoritusajat ovat $\Theta(n)$
- keskimääräisen tapauksen selville saamiseksi käytämme seuraavia merkintöjä:
  - m = hajautustaulun koko
  - n = alkioiden määrä taulussa
  - $\alpha = \frac{n}{m}$ = *täyttöaste* (*load factor*) eli listan keskimääräinen pituus
- lisäksi keskimääräisen suoritusajan arvioimiseksi on tehtävä oletus siitä, miten hyvin h hajauttaa alkiot
  - jos esim. $h(k)$ = nimen alkukirjaimen 3 ylintä bittiä, niin kaikki osuvat samaan listaan
  - usein oletetaan että, jokaisella alkiolla on yhtä suuri todennäköisyys osua mihin tahansa lokeroon
  - *tasainen hajautus* (*simple uniform hashing*)
  - oletetaan myös, että $h(k)$:n laskenta kuluttaa $\Theta(1)$ aikaa

- jos etsitään alkiota, jota ei ole taulussa, niin joudutaan selaamaan koko lista läpi
  $\Rightarrow$ joudutaan tutkimaan keskimäärin α alkiota
  $\Rightarrow$ suoritusaika keskimäärin $\Theta(1+\alpha)$
- jos oletetaan, että listassa oleva avain on mikä tahansa listan alkio samalla todennäköisyydellä, joudutaan listaa selamaan etsinnän yhteydessä keskimäärin puoleen väliin siinäkin tapauksessa, että avain löytyy listasta
  $\Rightarrow$ suoritusaika keskimäärin $\Theta(1+\frac{\alpha}{2}) = \Theta(1+\alpha)$
- jos täyttöaste pidetään alle jonkin kiinteän rajan (esim. $\alpha <$ 50 %), niin $\Theta(1+\alpha) = \Theta(1)$
  $\Rightarrow$ ketjutetun hajautustaulun kaikki operaatiot voi toteuttaa keskimäärin ajassa $\Theta(1)$
  - tämä edellyttää, että hajautustaulun koko on samaa luokkaa kuin sinne talletettavien alkioiden määrä

Laskiessamme keskimääräistä suoritusaikaa oletimme, että hajautusfunktio hajauttaa täydellisesti. Ei kuitenkaan ole mitenkään itsestään selvää, että näin tapahtuu.

Hajautusfunktion laatu on kriittisin tekijä hajautustaulun suorituskyvyn muodostumisessa.

Hyvän hajautusfunktion ominaisuuksia:

- hajautusfunktion on oltava deterministinen
  - muutoin kerran tauluun pantua ei välttämättä enää koskaan löydetä!
- tästä huolimatta olisi hyvä, että hajautusfunktion arvo olisi mahdollisimman "satunnainen"
  - kuhunkin lokeroon tulisi osua mahdollisimman tarkasti $\frac{1}{m}$ avaimista

- valitettavasti täysin tasaisesti hajottavan hajautusfunktion teko on useinmiten mahdotonta
  - eri arvojen esiintymistodennäköisyydet aineistossa ovat yleensä tuntemattomia
  - aineisto ei yleensä ole tasaisesti jakautunut
    - lähes mikä tahansa järkevä hajautusfunktio jakaa tasaisesti jakautuneen aineiston täydellisesti
- yleensä hajautusfunktio pyritään muodostamaan siten, että se sotkee tehokkaasti kaikki syöteaineistossa luultavasti esiintyvät säännönmukaisuudet
  - esimerkiksi nimien tapauksessa ei katsota yksittäisiä kirjaimia, vaan otetaan jotenkin huomioon nimen kaikki bitit

- esittelemme kaksi yleistä usein hyvin toimivaa hajautusfunktion luontimenetelmää
- oletamme, että avaimet ovat luonnollisia lukuja 0, 1, 2, . . .
  - jollei näin ole, avain voidaan usein tulkita luonnolliseksi luvuksi
  - esim. nimen saa luvuksi muuttamalla kirjaimet numeroiksi ASCII-koodiarvon mukaan, ja laskemalla ne sopivasti painottaen yhteen

Hajautusfunktion luonti *jakomenetelmällä* on yksinkertaista ja nopeaa.

- $h(k) = k \mod m$
- sitä kannattaa kuitenkin käyttää vain, jos m:n arvo on sopiva
- esim. jos $m = 2^b$ jollekin $b \in N = \{0, 1, 2, \ldots\}$, niin

$$h(k) = k\text{:n } b \text{ alinta bittiä}$$

$\Rightarrow$ funktio ei edes katso kaikkia k:n bittejä
$\Rightarrow$ funktio todennäköisesti hajauttaa huonosti, jos avaimet ovat peräisin binäärijärjestelmästä

- samasta syystä tulee välttää m:n arvoja muotoa $m = 10^b$, jos avaimet ovat peräisin kymmenjärjestelmän luvuista
- jos avaimet ovat muodostetut tulkitsemalla merkkijono 128-järjestelmän luvuksi, niin m = 127 on huono valinta, koska silloin saman merkkijonon kaikki permutaatiot osuvat samaan lokeroon
- hyviä m:n arvoja ovat yleensä alkuluvut, jotka eivät ole lähellä 2:n potensseja
  - esim. halutaan $\approx$ 700 listaa $\Rightarrow$ 701 kelpaa
- kannattaa tarkistaa kokeilemalla pienellä "oikealla" aineistolla, hajauttaako funktio avaimet tehokkaasti

Hajautusfunktion luonti *kertomenetelmällä* ei aseta suuria vaatimuksia m:n arvolle.

- valitaan vakio A siten, että $0 < A < 1$
- $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$
- jos $m = 2^b$, koneen sanapituus on w, ja k ja $2^w \cdot A$ mahtuvat yhteen sanaan, niin $h(k)$ voidaan laskea helposti seuraavasti:

$$h(k) = \lfloor \frac{(((2^w \cdot A) \cdot k) \mod 2^w)}{2^{w-b}} \rfloor$$

- mikä arvo A:lle tulisi valita?
  - kaikki A:n arvot toimivat ainakin jollain lailla
  - kuulemma $A \approx \frac{\sqrt{5}-1}{2}$ toimii usein aika hyvin

12 Binäärihakupuu

http://imgur.com/L77FY5X

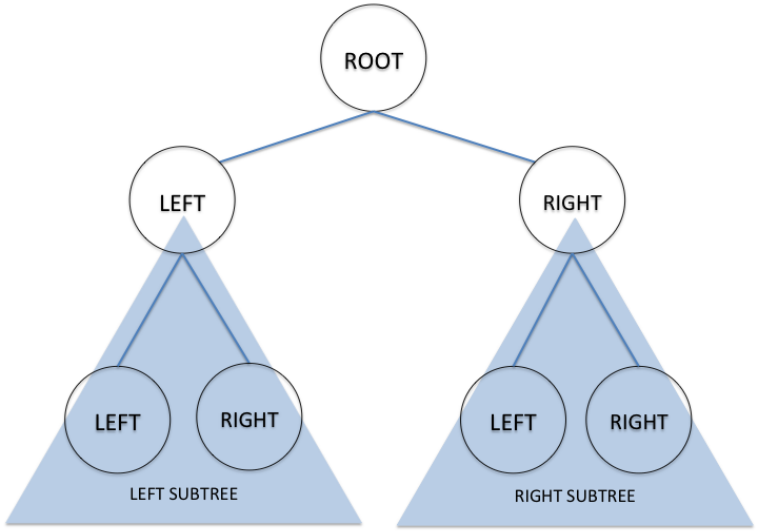
Tässä luvussa käsitellään erilaisia yleisiä puurakenteita.

- Ensin opitaan, millainen rakenne on binäärihakupuu,
- ja tasapainotetaan binäärihakupuu muuttamalla se puna-mustaksi puuksi.
- Sitten tutustutaan monihaaraisiin puihin: merkkijonopuu Trie ja B-puu.
- Lopuksi vilkaistaan splay- ja AVL-puita.

12.1 Tavallinen binäärihakupuu

Kertauksena: *Binääripuu* (*binary tree*) on äärellinen solmuista (*node*) koostuva rakenne, joka on joko

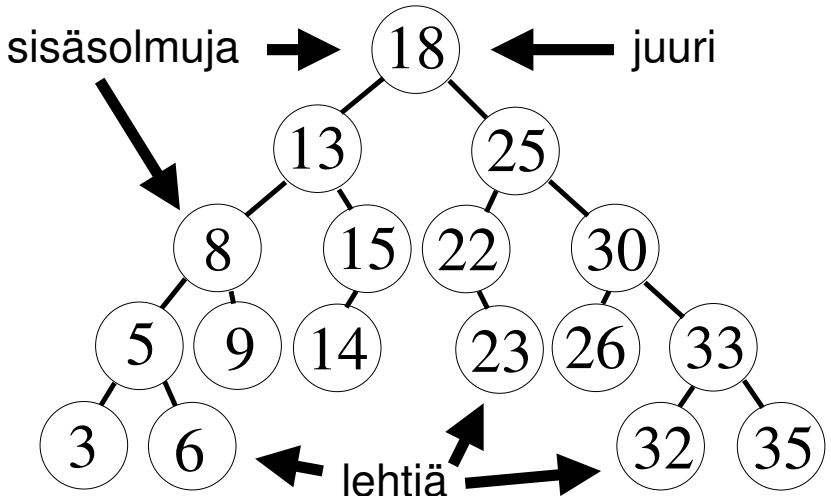
- tyhjä, tai
- sisältää yhden solmun nimeltä *juuri* (*root*), sekä kaksi binääripuuta nimeltä *vasen alipuu* (*left subtree*) ja *oikea alipuu* (*right subtree*).



Kuva 14: Kertaus: Binääripuu

Lisäksi määritellään:

- Lapseton solmu: *lehti* (*leaf*).
- Muut solmut *sisäsolmuja*.
- Solmu on lastensa *isä* (*parent*) ja solmun *esi-isiä* (*ancestor*) ovat solmu itse, solmun isä, tämän isä jne.
- *Jälkeläinen* (*descendant*) vastaavasti.

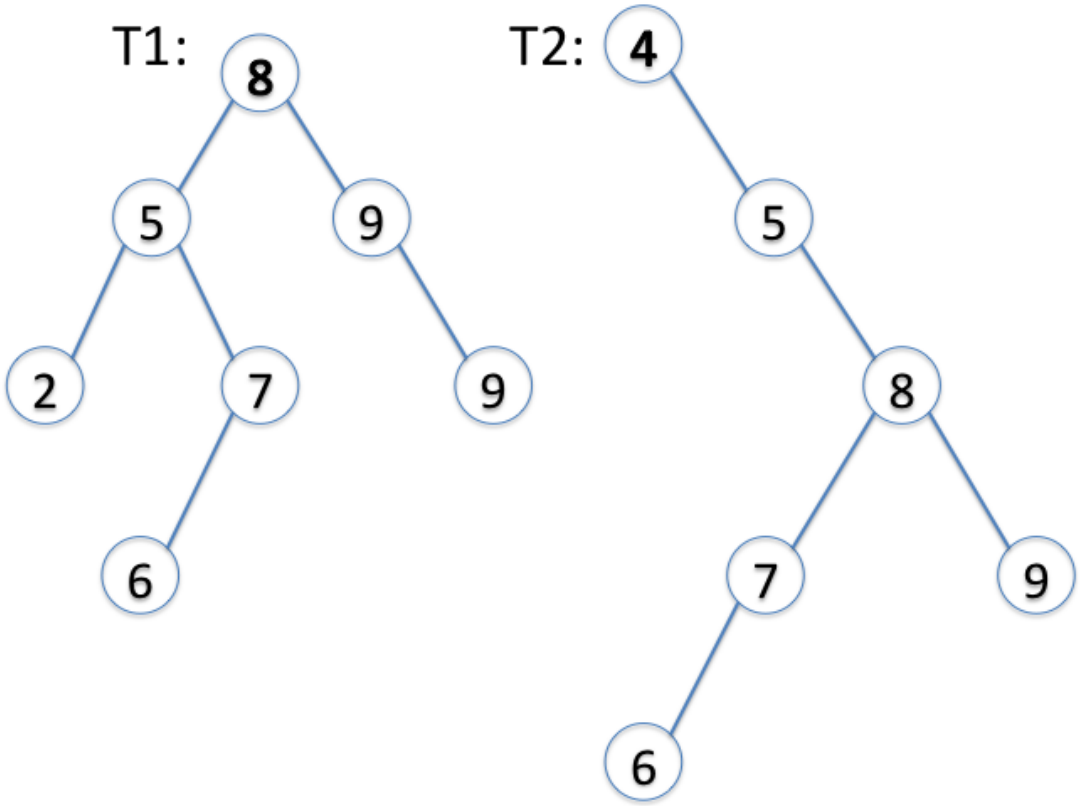


Binäärihakupuu (*binary search tree*) on binääripuu, jonka kaikille solmuille x pätee:

> *Jos l on mikä tahansa solmu x:n vasemmassa alipuussa ja r mikä tahansa solmu x:n oikeassa alipuussa, niin*
>
> $$l.key \leq x.key \leq r.key$$

- Edellisen sivun binääripuu on binäärihakupuu
- Luvussa 8.2 esitelty kekorakenne on binääripuu muttei binääri**haku**puu

Useinmiten binäärihakupuu esitetään linkitettynä rakenteena, jossa jokaisessa alkiossa on kentät avain (key), vasen lapsi ($left$), oikea lapsi ($right$) ja vanhempi (p (parent)). Lisäksi alkiolla on oheisdataa.



Kuva 15: Hakupuita

Avaimen haku binäärihakupuusta :

- koko puusta haku R-TREE-SEARCH$(T.root, k)$
- palauttaa osoittimen x solmuun, jolle $x{\rightarrow}key = k$, tai NIL, jos tällaista solmua ei ole

```
R-TREE-SEARCH(x, k)
1  if x = NIL or k = x→key then
2       return x                          (etsitty avain löytyi)
3  if k < x→key then                      (jos etsitty on pienempi kuin avain...)
4       return R-TREE-SEARCH(x→left, k)   (...etsitään vasemmasta alipuusta)
5  else                                   (muuten...)
6       return R-TREE-SEARCH(x→right, k)  (...etsitään oikeasta alipuusta)
```

Algoritmi suunnistaa juuresta alaspäin huonoimmassa tapauksessa pisimmän polun päässä olevaan lehteen asti.

- suoritusaika $O(h)$, missä h on puun korkeus
- lisämuistin tarve $O(h)$, rekursion vuoksi

Saman voi tehdä myös ilman rekursiota, mikä on suositeltavaa.

- tällöin lisämuistin tarve on vain $\Theta(1)$
- ajoaika on yhä $O(h)$

```
TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ x→key do   (kunnes avain on löytynyt tai ollaan lehdessä)
2      if k < x→key then            (jos etsitty on pienempi kuin avain...)
3          x := x→left              (...siirrytään vasemmalle)
4      else                         (muuten...)
5          x := x→right             (...siirrytään oikealle)
6  return x                         (palautetaan tulos)
```

Minimi ja *maksimi*:

- minimi löydetään menemällä vasemmalle niin kauan kun se on mahdollista

```
TREE-MINIMUM(x)
1  while x→left ≠ NIL do
2        x := x→left
3  return x
```

- maksimi löydetään vastaavasti menemällä oikealle niin kauan kun se on mahdollista

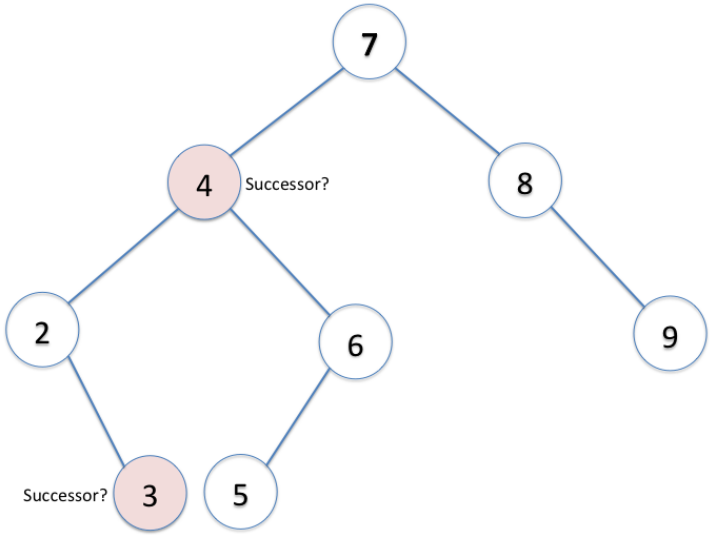
```
TREE-MAXIMUM(x)
1  while x→right ≠ NIL do
2        x := x→right
3  return x
```

- molempien ajoaika on $O(h)$ ja lisämuistin tarve $\Theta(1)$

Solmun *seuraajaa* ja *edeltäjää* kannattaa etsiä binäärihakupuusta puun rakenteen avulla mieluummin kuin avainten arvojen perusteella.

- tällöin kaikki alkiot saadaan käytyä niiden avulla läpi, vaikka puussa olisi yhtä suuria avaimia

  $\Rightarrow$ tarvitaan siis algoritmi, joka etsii annettua solmua välijärjestyksessä seuraavan solmun
- sellainen voidaan rakentaa algoritmin TREE-MINIMUM avulla

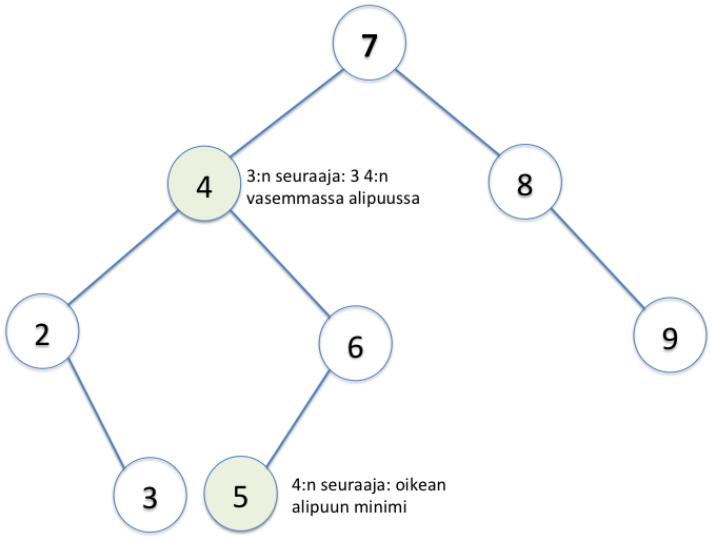


Kuva 16: Solmun seuraaja?

Binäärihakupuun solmun seuraaja on joko:

- oikean alipuun pienin alkio
- tai solmusta juureen vievällä polulla ensimmäinen kohdattu solmu, jonka vasempaan alipuuhun solmu kuuluu

jos edellä mainittuja solmuja ei löydy, on kysymyksessä puun suurin solmu



Kuva 17: Seuraajat

```
TREE-SUCCESSOR(x)
1  if x→right ≠ NIL then                (jos oikea alipuu löytyy...)
2      return TREE-MINIMUM(x→right)     (...etsitään sen minimi)
3  y := x→p                             (muuten lähdetään kulkemaan kohti juurta)
4  while y ≠ NIL and x = y→right do     (kunnes ollaan tultu vasemmasta lapsesta)
5      x := y
6      y := y→p
7  return y                             (palautetaan löydetty solmu)
```

- huomaa, että avainten arvoja ei edes katsota!
- vrt. seuraajan löytäminen järjestetystä listasta
- ajoaika $O(h)$, lisämuistin tarve $\Theta(1)$
- TREE-PREDECESSOR voidaan toteuttaa vastaavalla tavalla

TREE-SUCCESSORIN ja TREE-MINIMUMIN avulla voidaan rakentaa toinen tapa selata puu läpi välijärjestyksessä.

```
TREE-SCAN-ALL(T)
1   if T.root ≠ NIL then
2       x := TREE-MINIMUM(T.root)        (aloitetaan selaus puun minimistä)
3   else
4       x := NIL
5   while x ≠ NIL do                     (selataan niin kauan kun seuraajia löytyy)
6       käsittele alkio x
7       x := TREE-SUCCESSOR(x)
```

- jokainen kaari kuljetaan kerran molempiin suuntiin
  $\Rightarrow$ TREE-SCAN-ALL selviää ajassa $\Theta(n)$, vaikka kutsuukin TREE-SUCCESSORia n kertaa

- lisämuistin tarve $\Theta(1)$

  $\Rightarrow$ TREE-SCAN-ALL on asymptoottisesti yhtä nopea, ja muistinkulutukseltaan asymptoottisesti parempi kuin INORDER-TREE-WALK

  - vakiokertoimissa ei suurta eroa

  $\Rightarrow$ kannattaa valita TREE-SCAN-ALL, jos tietueissa on p-kentät

- TREE-SCAN-ALL sallii useat yhtäaikaiset selaukset, INORDER-TREE-WALK ei

Lisäys binäärihakupuuhun:

```
TREE-INSERT(T, z)                          (z osoittaa käyttäjän varaamaa alustettua tietuetta)
1   y := NIL; x := T.root                  (aloitetaan juuresta)
2   while x ≠ NIL do                       (laskeudutaan kunnes kohdataan tyhjä paikka)
3       y := x                             (otetaan potentiaalinen isä-solmu talteen)
4       if z→key < x→key then              (siirrytään oikealle tai vasemmalle)
5           x := x→left
6       else
7           x := x→right
8   z→p := y                               (sijoitetaan löydetty solmu uuden solmun isäksi)
9   if y = NIL then
10      T.root := z                        (puun ainoa solmu on juuri)
11  else if z→key < y→key then             (sijoitetaan uusi solmu isänsä vasemmaksi . . . )
12      y→left := z
13  else                                   (. . . tai oikeaksi lapseksi)
14      y→right := z
15  z→left := NIL; z→right := NIL
```

Algoritmi suunnistaa juuresta lehteen; uusi solmu sijoitetaan aina lehdeksi.
$\Rightarrow$ ajoaika $O(h)$, lisämuistin tarve $\Theta(1)$

Poisto on monimutkaisempaa, koska se voi kohdistua sisäsolmuun:

```
TREE-DELETE(T, z)                                  (z osoittaa poistettavaa solmua)
1  if z→left = NIL or z→right = NIL then           (jos z:lla on vain yksi lapsi . . . )
2      y := z                                      (. . . asetetaan z poistettavaksi tietueeksi)
3  else
4      y := TREE-SUCCESSOR(z)                      (muuten poistetaan z:n seuraaja)
5  if y→left ≠ NIL then                            (otetaan talteen poistettavan ainoa lapsi)
6      x := y→left
7  else
8      x := y→right
9  if x ≠ NIL then                                 (jos lapsi on olemassa . . . )
10     x→p := y→p                                  (. . . linkitetään se poistettavan tilalle)
11 if y→p = NIL then                               (jos poistettava oli juuri . . . )
12     T.root := x                                 (. . . merkitään x uudeksi juureksi)
13 else if y = y→p→left then                       (sijoitetaan x poistettavan tilalle . . . )
14     y→p→left := x                               (. . . sen isän vasemmaksi . . . )
15 else
16     y→p→right := x                              (. . . tai oikeaksi lapseksi)
17 if y ≠ z then                                   (jos poistettiin joku muu kuin z . . . )
18     z→key := y→key                              (. . . vaihdetaan poistetun ja z:n datat)
19     z→satellitedata := y→satellitedata
20 return y                                        (palautetaan osoitin poistettuun solmuun)
```

Huom! Rivillä 5 todellakin tiedetään, että y:llä on korkeintaan yksi lapsi.

- jos z:lla on vain yksi lapsi, y on z
- jos rivillä 4 kutsutaan TREE-SUCCESSORIA tiedetään, että z:lla on oikea alipuu, jonka minimi on y
  - minimillä ei voi olla vasenta lasta

Algoritmi näyttää monimutkaiselta, mutta rivin 4 TREE-SUCCESSORIA lukuunottamatta kaikki operaatiot ovat vakioaikaisia.
$\Rightarrow$ ajoaika on siis $O(h)$ ja lisämuistin tarve Θ(1)

Siis kaikki dynaamisen joukon perusoperaatiot saadaan binäärihakupuulla toimimaan ajassa $O(h)$ ja lisämuistilla $\Theta(1)$:

SEARCH, INSERT, DELETE, MINIMUM, MAXIMUM, SUCCESSOR ja PREDECESSOR

Binäärihakupuita - kuten muitakin tietorakenteita - voi sovittaa uusiin tehtäviin lisäämällä uusia tehtävän kannalta olennaista tietoa sisältäviä kenttiä.

- tällöin perusoperaatiot tulee muokata ylläpitämään myös uusien kenttien sisältöä
- esimerkiksi lisäämällä solmuihin kenttä, joka kertoo solmun virittämän alipuun koon
  - saadaan toteutettua algoritmi, joka palauttaa puun korkeuteen nähden lineaarisessa ajassa i:nnen alkion
  - saadaan toteutettua algoritmi, joka puun korkeuteen nähden lineaarisessa ajassa kertoo, monesko kysytty alkio on suuruusjärjestyksessä
  - ilman ylimääräistä kenttää algoritmit täytyisi toteuttaa huomattavasti tehottomammin puun alkioiden määrään nähden lineaarisessa ajassa

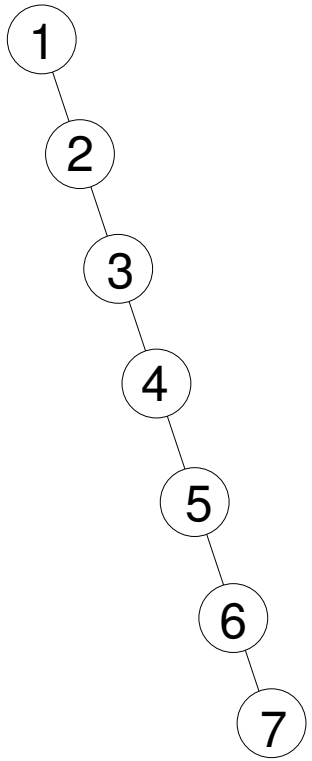
12.2 Kuinka korkeita binäärihakupuut yleensä ovat?

Kaikki dynaamisen joukon perusoperaatiot saatiin binäärihakupuulla toimimaan ajassa $O(h)$. $\Rightarrow$ puun korkeus on tehokkuuden kannalta keskeinen tekijä.

Jos oletetaan, että alkiot on syötetty satunnaisessa järjestyksessä, ja jokainen järjestys on yhtä todennäköinen, suoraan INSERTillä rakennetun binäärihakupuun korkeus on keskimäärin $\Theta(\lg n)$.
$\Rightarrow$ kaikki operaatiot keskimäärin $\Theta(\lg n)$

Valitettavasti lopputulos on surkeampi, jos avaimet syötetään (lähes) suuruusjärjestyksessä, kuten viereisestä kuvasta voi havaita.

- korkeus on n -1, surkeaa!



Ongelmaa ei pystytä ratkaisemaan järkevästi esimerkiksi satunnaistamalla, jos halutaan säilyttää kaikki dynaamisen joukon operaatiot.

Puu pitää siis tasapainottaa. Siihen palataan myöhemmin.

15 Tasapainotetut puurakenteet

Binäärihakupuu toteuttaa kaikki dynaamisen joukon operaatiot $O(h)$ ajassa

Kääntöpuolena on, että puu voi joskus litistyä listaksi, jolloin tehokkuus menetetään ($O(n)$)

Tässä luvussa käsitellään tapoja pitää huolta siitä, ettei litistymistä käy

Ensin opitaan tasapainoitus puna-mustan puun invarianttia ylläpitämällä

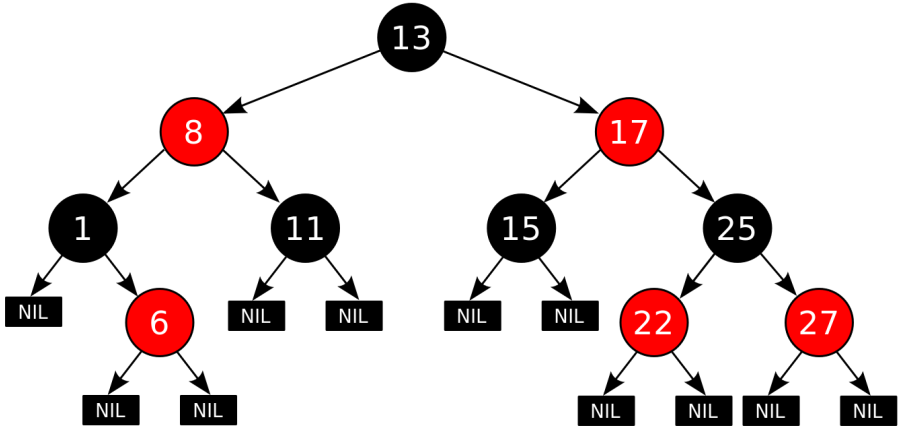
Lopuksi vilkaistaan muista tasapainotetuista binäärihakupuista Splay- ja AVL-puita

15.1 Puna-musta binäärihakupuu

Puna-mustat puut ovat tasapainotettuja binäärihakupuita.

Ne tekevät lisäysten ja poistojen yhteydessä tasapainotustoimenpiteitä, jotka takaavat, ettei haku ole koskaan tehoton vaikka alkiot olisikin lisätty puuhun epäsuotuisassa järjestyksessä.

- puna-musta puu ei voi koskaan litistyä listaksi, kuten perusbinäärihakupuu



Kuva 23: Punamustapuu (via Wikipedia, ©Colin M.L. Burnett (CC BY-SA 3.0))

Puna-mustien puiden perusidea:

- jokaisessa solmussa on yksi lisäbitti: *väri* (*colour*)
  - arvot *punainen* ja *musta*
- muut kentät ovat vanhat tutut key, $left$, $right$ ja p
  - jätämme oheisdatan näyttämättä, jotta pääideat eivät hukkuisi yksityiskohtien taakse
- värikenttien avulla ylläpidetään *puna-mustan puun invarianttia*, joka takaa, että puun korkeus on aina kertaluokassa $\Theta(\lg n)$

Puna-mustien puiden **invariantti**:

1. Jos solmu on punainen, niin sillä joko
   - ei ole lapsia, tai
   - on kaksi lasta, ja ne molemmat ovat mustia.
2. Jokaiselle solmulle pätee: jokainen solmusta alas 1- tai 0-lapsiseen solmuun vievä polku sisältää saman määrän mustia solmuja.
3. Juuri on musta.

Solmun x *musta-korkeus* (*black-height*) $bh(x)$ on siitä alas 1- tai 0-lapsiseen solmuun vievällä polulla olevien mustien solmujen määrä.

- invariantin osan 3 mukaisesti jokaisen solmun mustakorkeus on yksikäsitteinen
- jokaisella vaihtoehtoisella polulla on sama määrä mustia solmuja
- koko puun mustakorkeus on sen juuren mustakorkeus

Puna-mustan puun maksimikorkeus

- merkitään korkeus = h ja solmujen määrä = n
- kunkin juuresta lehteen vievän polun solmuista vähintään puolet ($\lfloor \frac{h}{2} \rfloor + 1$) ovat mustia (invariantin osat 1 ja 3)
- jokaisella juuresta lehteen vievällä polulla on saman verran mustia solmuja (invariantin osa 2)

  $\Rightarrow$ ainakin $\lfloor \frac{h}{2} \rfloor + 1$ ylintä tasoa täysiä

  $\Rightarrow n \geq 2^{\frac{h}{2}}$

  $\Rightarrow h \leq 2 \lg n$

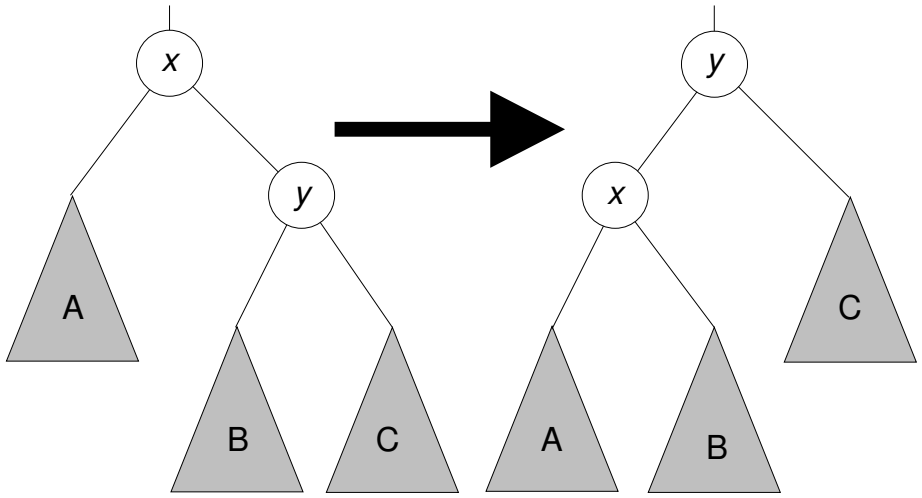
Invariantti siis todellakin takaa puun korkeuden pysymisen logaritmisena puun alkioiden määrään nähden.
$\Rightarrow$ Dynaamisen joukon operaatiot SEARCH, MINIMUM, MAXIMUM, SUCCESSOR ja PREDECESSOR saadaan toimimaan puna-mustille puille ajassa $O(\lg n)$.

- binäärihakupuulle operaatiot toimivat ajassa $O(h)$, ja puna-musta puu on binäärihakupuu, jolle $h = \Theta(\lg n)$

Puna-mustien puiden ylläpitämiseen ei kuitenkaan voida käyttää samoja lisäys- ja poistoalgoritmeja kuin tavallisilla binäärihakupuilla, koska ne saattavat rikkoa invariantin.

Niiden sijaan käytetään algoritmeja RB-INSERT ja RB-DELETE.

- operaatiot RB-INSERT ja RB-DELETE perustuvat *kiertoihin* (*rotation*)
- kiertoja on kaksi: vasemmalle ja oikealle
- ne muuttavat puun rakennetta, mutta säilyttävät binäärihakupuiden perusominaisuuden kaikille solmuille



- kierto vasemmalle
  - olettaa, että solmut x ja y ovat olemassa
- kierto oikealle vastaavasti
  - $left$ ja $right$ vaihtaneet paikkaa

```
LEFT-ROTATE(T, x)
1   y := x→right; x→right := y→left
2   if y→left ≠ NIL then
3       y→left→p := x
4   y→p := x→p
5   if x→p = NIL then
6       T.root := y
7   else if x = x→p→left then
8       x→p→left := y
9   else
10      x→p→right := y
11  y→left := x; x→p := y
```

- molempien kiertojen ajoaika on $\Theta(1)$
- ainoastaan osoittimia muutetaan

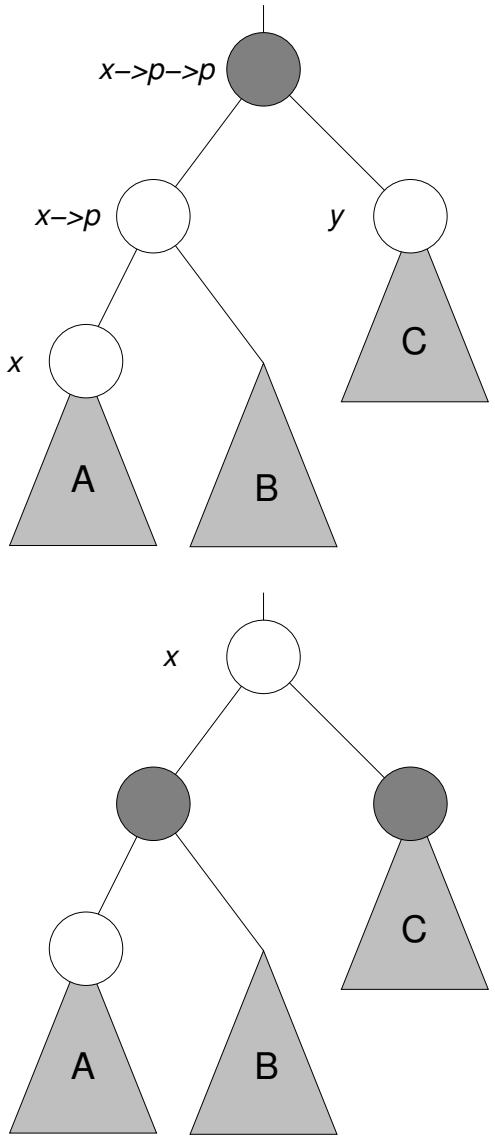
Lisäyksen perusidea

- ensin uusi solmu lisätään kuten tavalliseen binäärihakupuuhun
- sitten lisätty väritetään punaiseksi
- mitä puna-mustien puiden perusominaisuuksia näin tehty lisäys voi rikkoa?

- Invariantin osa
  - 1 rikkoutuu lisätyn solmun osalta, jos sen isä on punainen; muuten se ei voi rikkoutua.
  - 2 ei rikkoudu, koska minkään solmun alla olevien mustien solmujen määrät ja sijainnit eivät muutu, ja lisätyn alla ei ole solmuja.
  - 3 rikkoutuu, jos puu oli alun perin tyhjä.

- korjataan puu seuraavasti:
  - ominaisuutta 2 pilaamatta siirretään 1:n rike ylöspäin kunnes se katoaa
  - lopuksi 3 korjataan värittämällä juuri mustaksi (ei voi pilata ominaisuuksia 1 ja 2)
- 1:n rike = sekä solmu että sen isä ovat punaisia
- siirto tapahtuu värittämällä solmuja ja tekemällä kiertoja

```
 RB-INSERT(T, x)
1   TREE-INSERT(T, x)
2   x→colour := RED
    (suoritetaan silmukkaa kunnes rike on hävinnyt tai ollaan saavutettu juuri)
3   while x ≠ T.root and x→p→colour = RED do
4       if x→p = x→p→p→left then
5           y := x→p→p→right
6           if y ≠ NIL and y→colour = RED then  (siirretään rikettä ylöspäin)
7               x→p→colour := BLACK
8               y→colour := BLACK
9               x→p→p→colour := RED
10              x := x→p→p
11          else                                (siirto ei onnistu → korjataan rike)
12              if x = x→p→right then
13                  x := x→p; LEFT-ROTATE(T, x)
14              x→p→colour := BLACK
15              x→p→p→colour := RED
16              RIGHT-ROTATE(T, x→p→p)
17      else
...     ▷ sama kuin rivit 5...16 paitsi "left" ja "right" vaihtaneet paikkaa
30  T.root→colour := BLACK                      (väritetään juuri mustaksi)
```



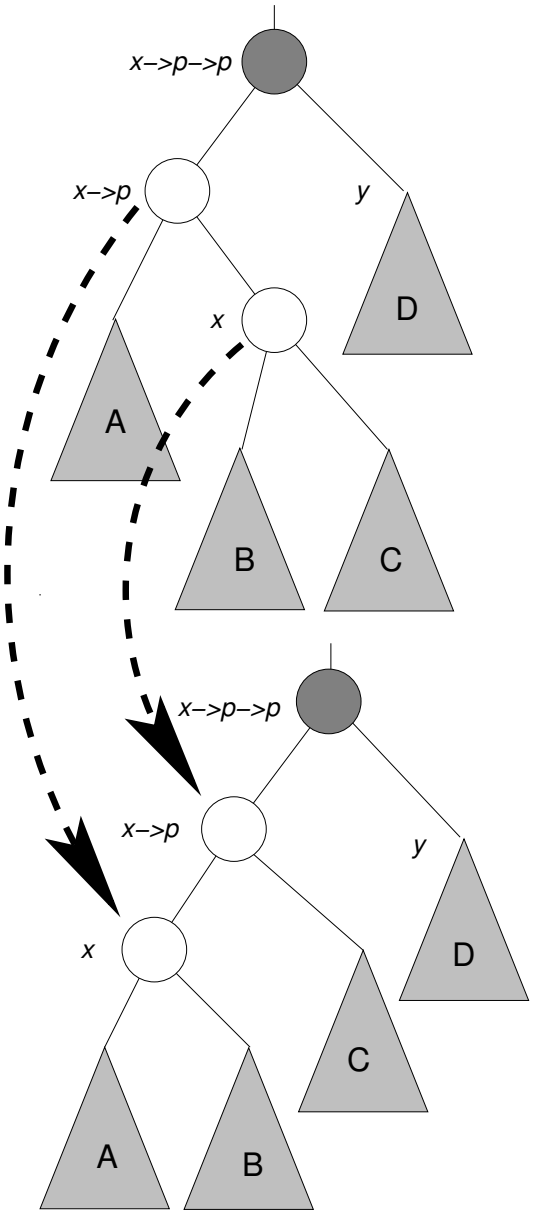
Ominaisuuden 1 rikkeen siirto ylöspäin:

- solmu x ja sen isä ovat molemmat punaisia.
- myös solmun x setä on punainen ja isoisä musta.

$\Rightarrow$ rike siirretään ylöspäin värittämällä sekä x:n setä että isä mustiksi ja isoisä punaiseksi.

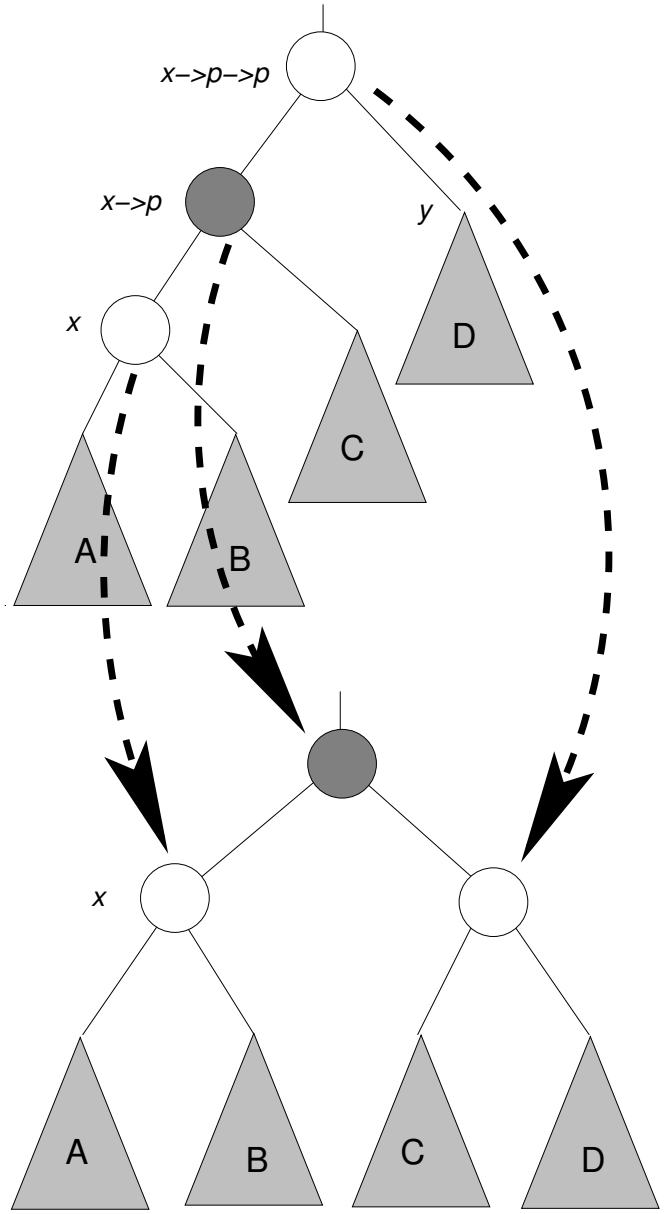
Korjauksen jälkeen:

- ominaisuus 1 saattaa olla edelleen rikki
  - solmu x ja sen isä saattavat molemmat olla punaisia
- ominaisuus 2 ei rikkoudu
  - kaikkien polkujen mustien solmujen määrä pysyy samana
- ominaisuus 3 saattaa rikkoutua
  - jos ollaan noustu juureen asti, se on saatettu värittää punaiseksi



Mikäli punaista setää ei ole olemassa, rikettä ei voi siirtää ylöspäin vaan se täytyy poistaa käyttäen monimutkaisempaa menetelmää:

- Varmistetaan ensin, että x on isänsä vasen lapsi tekemällä tarvittaessa kierto vasemmalle.



- tämän jälkeen väritetään x:n isä mustaksi ja isoisä punaiseksi, ja suoritetaan kierto oikealle
  - isoisä on varmasti musta, koska muuten puussa olisi ollut kaksi punaista solmua päällekkäin jo ennen lisäystä

Korjauksen jälkeen:

- puussa ei enää ole päällekkäisiä punaisia solmuja
- korjausoperaatiot yhdessä eivät riko 2. ominaisuutta

  $\Rightarrow$ puu on ehjä ja korjausalgoritmin suorittaminen voidaan lopettaa

Poistoalgoritmin yleispiirteet

- ensin solmu poistetaan kuten tavallisesta binäärihakupuusta
  - w osoittaa poistettua solmua
- jos w oli punainen tai puu tyhjeni kokonaan, puna-musta-ominaisuudet säilyvät voimassa
  $\Rightarrow$ ei tarvitse tehdä muuta
- muussa tapauksessa korjataan puu RB-DELETE-FIXUPin avulla aloittaen w:n (mahdollisesta) lapsesta x ja sen isästä $w{\rightarrow}p$
  - TREE-DELETE takaa, että w:llä oli enintään yksi lapsi

```
RB-DELETE(T, z)
1   w := TREE-DELETE(T, z)
2   if w→colour = BLACK and T.root ≠ NIL then
3       if w→left ≠ NIL then
4           x := w→left
5       else
6           x := w→right
7       RB-DELETE-FIXUP(T, x, w→p)
8   return w
```

```
RB-DELETE-FIXUP(T, x, y)
1   while x ≠ T.root and (x = NIL or x→colour = BLACK) do
2       if x = y→left then
3           w := y→right
4           if w→colour = RED then
5               w→colour := BLACK; y→colour := RED
6               LEFT-ROTATE(T, y); w := y→right
7           if (w→left = NIL or w→left→colour = BLACK) and
            (w→right = NIL or w→right→colour = BLACK)
            then
8               w→colour := RED; x := y
9           else
10              if w→right = NIL or w→right→colour = BLACK then
11                  w→left→colour := BLACK
12                  w→colour := RED
13                  RIGHT-ROTATE(T, w); w := y→right
14              w→colour := y→colour; y→colour := BLACK
15              w→right→colour := BLACK; LEFT-ROTATE(T, y)
16              x := T.root
17      else
...             ▷ sama kuin rivit 3...16 paitsi "left" ja "right" vaihtaneet paikkaa
32      y := y→p
33  x→colour := BLACK
```

15.2 AVL-puut ja Splay-puut

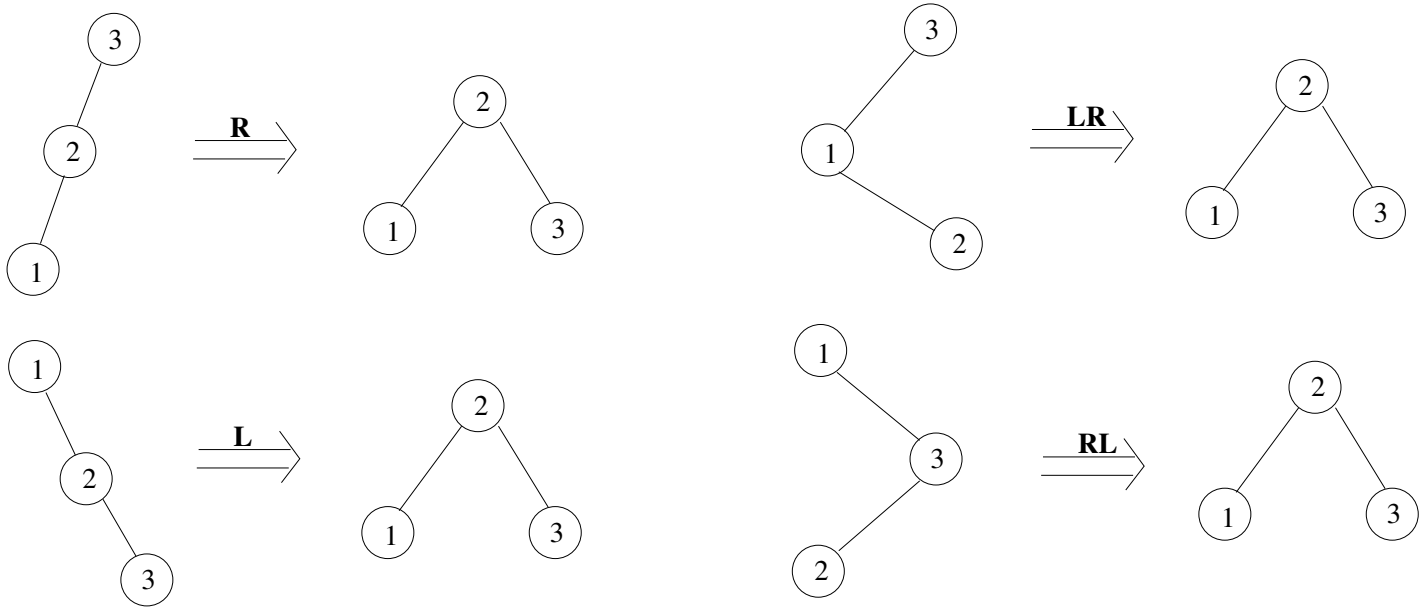
AVL puu (Adelson-Velsky, Landis mukaan) on binäärihakupuu, jossa jokaisella solmulla on **tasapainokerroin**: 0, +1, tai -1, kun tasapainossa.

- kerroin määräytyy solmun oikean ja vasemman alipuun korkeuksien erotuksesta.

Kun uuden solmun lisäys tekee AVL-puusta epätasapainoisen, puu palautetaan tasapainoiseksi tekemällä rotaatioita.

Mahdollisia rotaatioita on neljä:

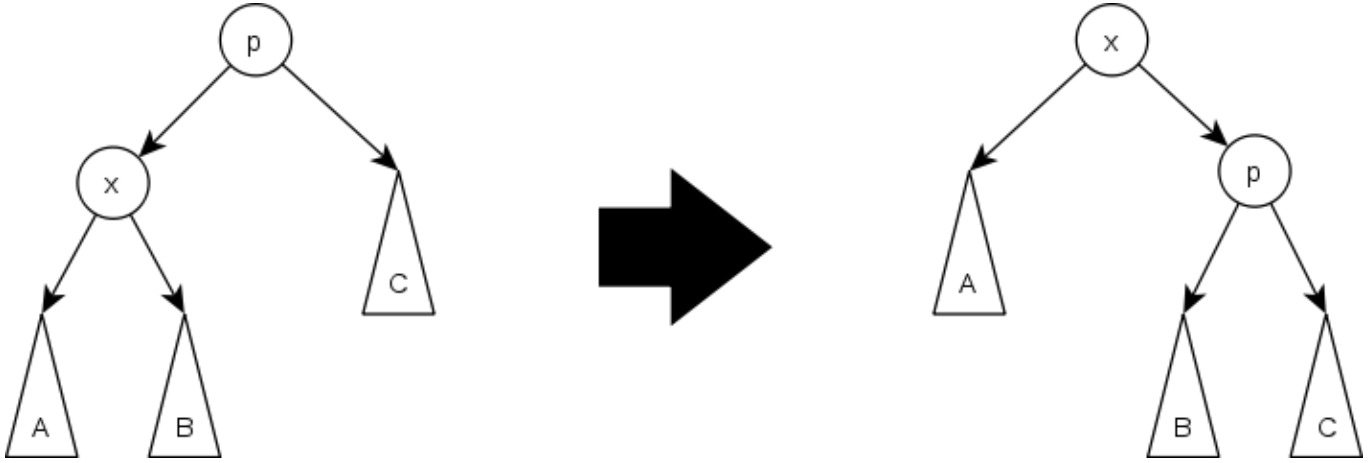
- Oikealle
- Vasemmalle
- Kaksois-rotaatio vasen-oikea
- Kaksois-rotaatio oikea-vasen



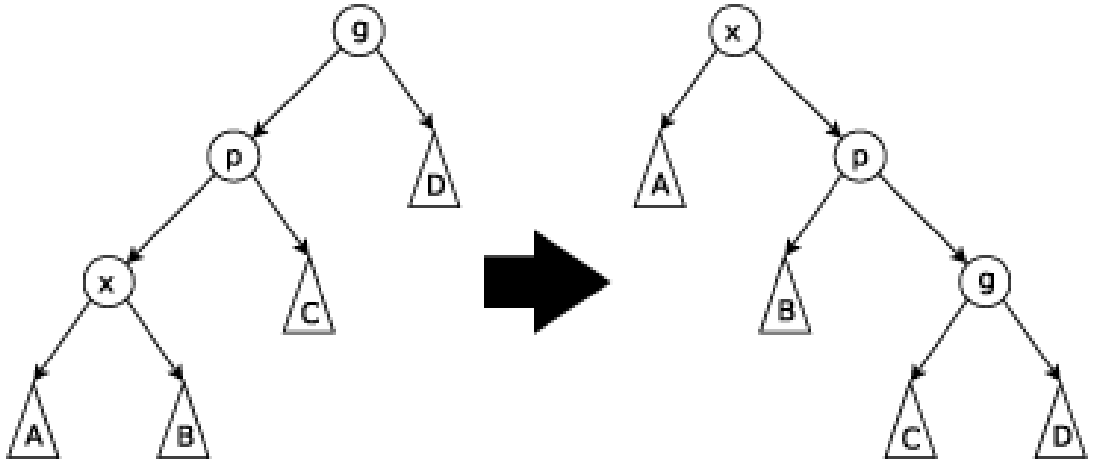
Splay puu on binäärihakupuu, jossa lisäominaisuutena viimeksi haetut alkiot ovat nopeita hakea uudelleen.

Splay-operaatio suoritetaan solmulle haun yhteydessä. Tämä ns. splay-askelien sekvenssi siirtää solmun askel askeleelta lähemmäksi juurta ja lopulta juureksi.

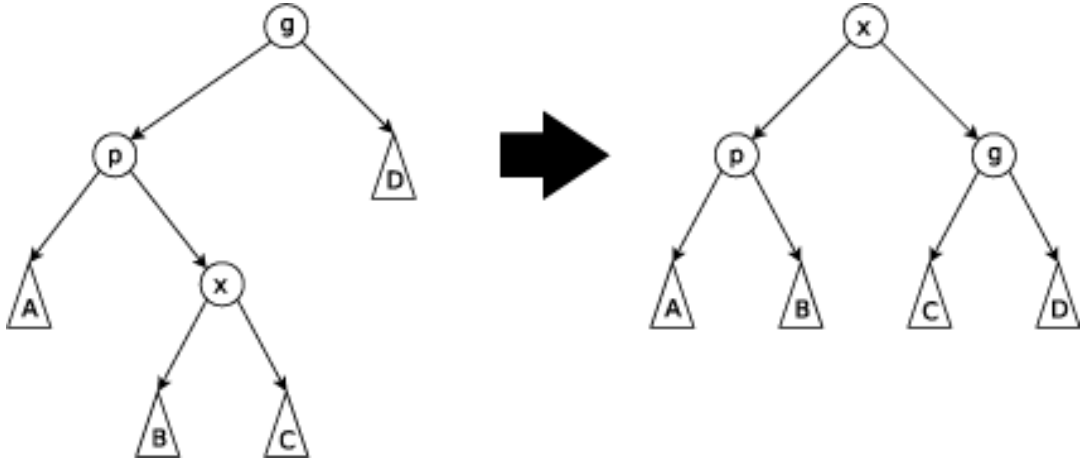
- Zig-askel:



- Zig-Zig-askel:



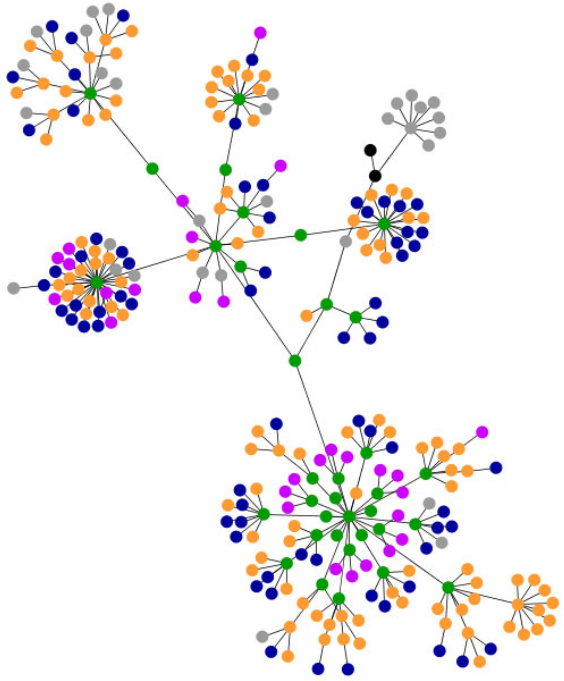
- Zig-Zag-askel:



13 Graafit

Seuraavaksi tutustutaan tietorakenteeseen, jonka muodostavat pisteet ja niiden välille muodostetut yhteydet – graafiin.

Keskitymme myös tyypillisimpiin tapoihin etsiä tietoa graafista eli graafialgoritmeihin.

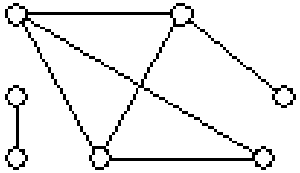


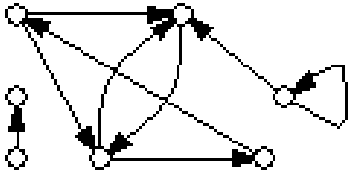
Kuva: Flickr, Rachel D

13.1 Graafien esittäminen tietokoneessa

Graafi on ohjelmistotekniikassa keskeinen rakenne, joka koostuu solmuista (*vertex, node*) ja niitä yhdistävistä *kaarista* (*edge, arc*).

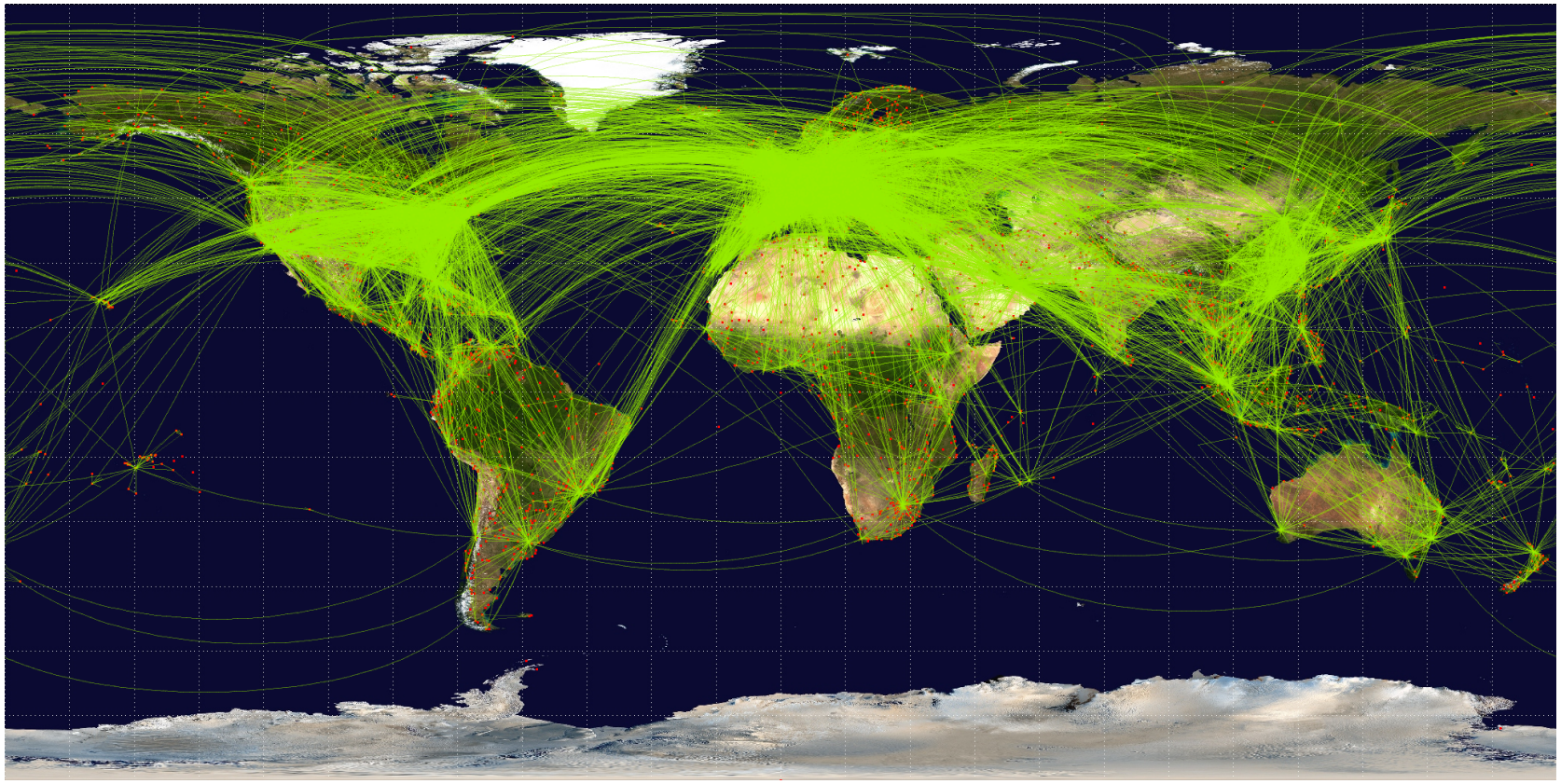
Graafi voi olla *suuntaamaton* (*undirected*) tai *suunnattu* (*directed*).



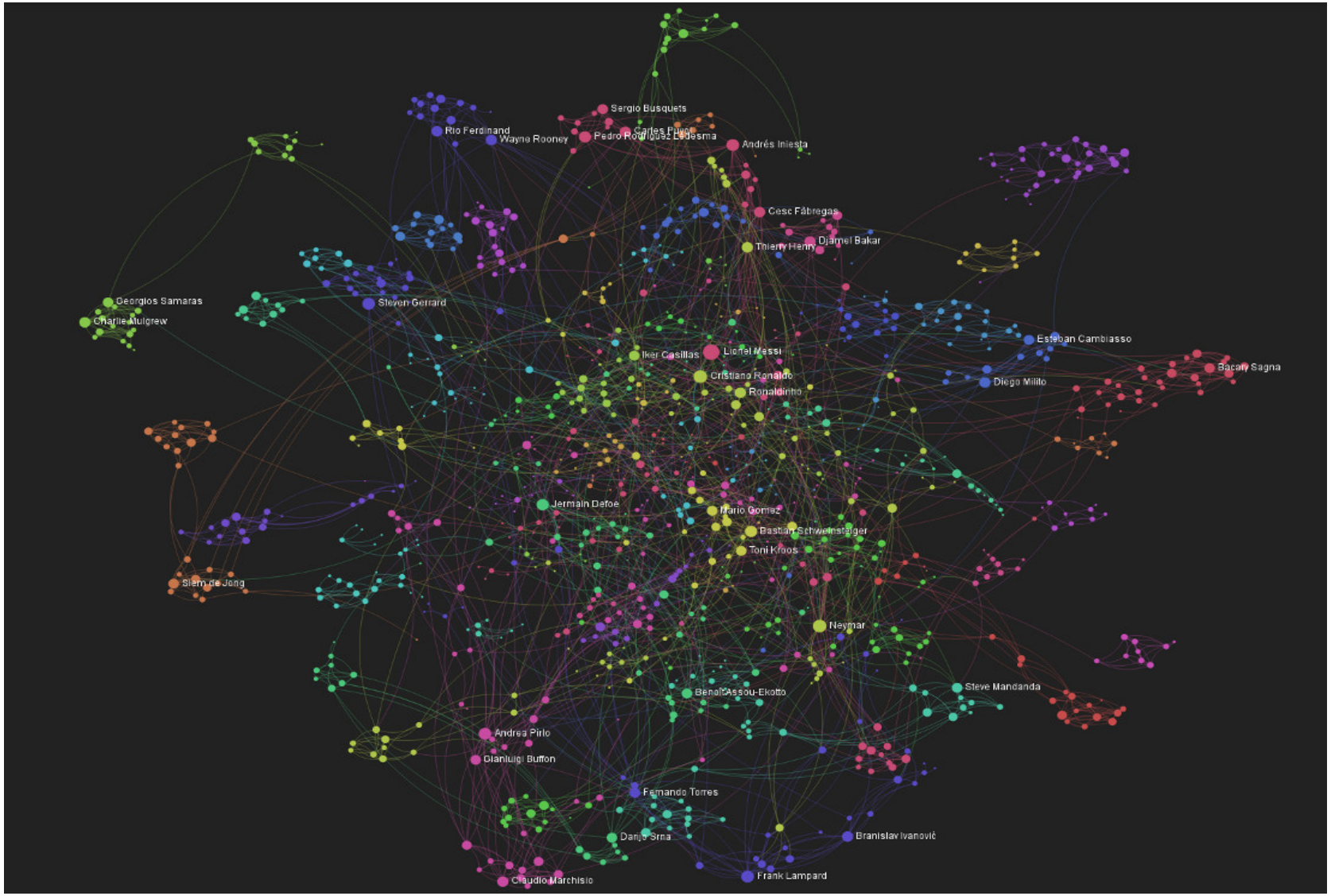


Kuva 18: Kuva: Getty Images

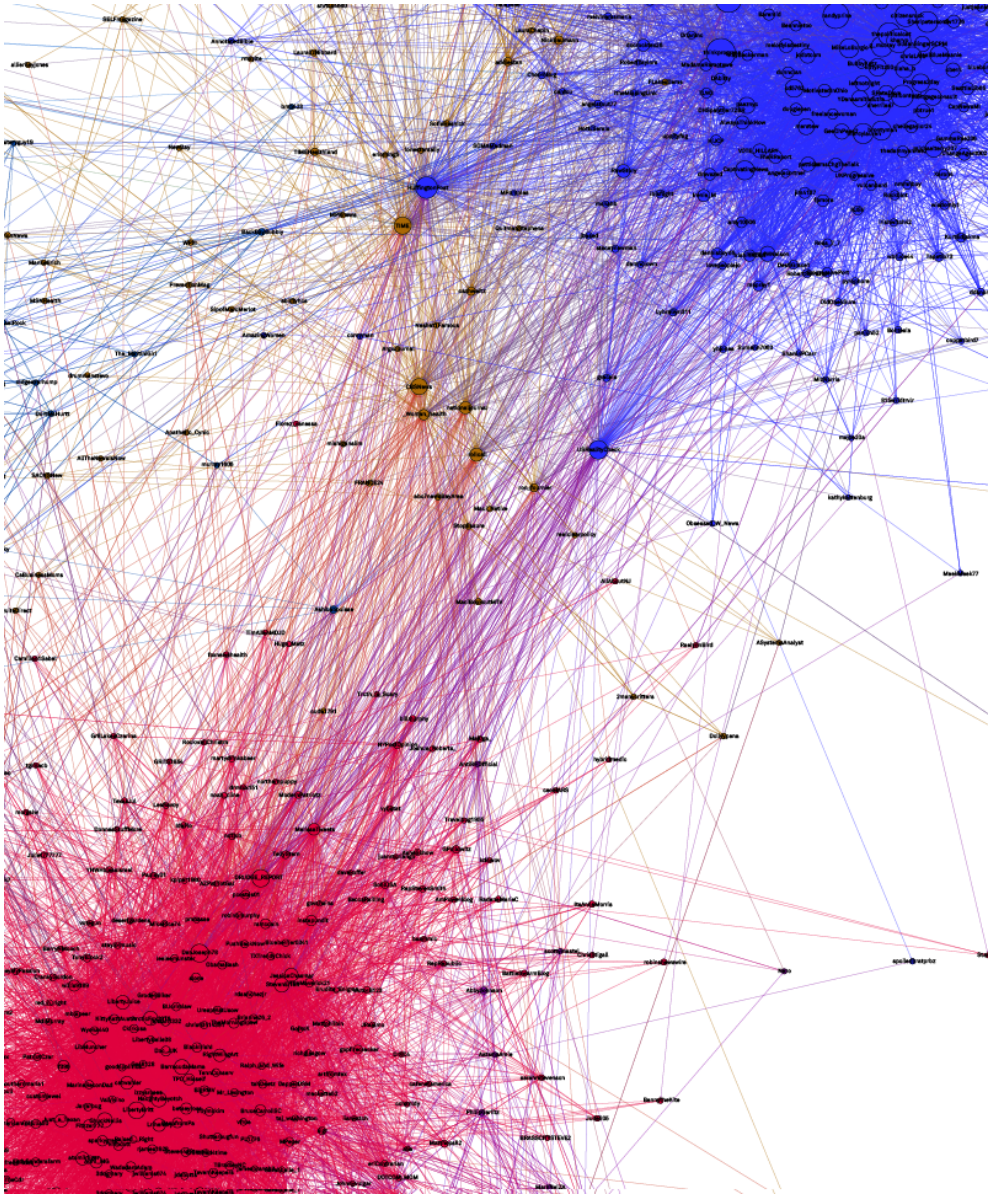
- kaaviokuvat voidaan usein ajatella graafeiksi
- asioiden välisiä suhteita voi usein esittää graafeina
- monet tehtävät voidaan palauttaa graafitehtäviksi



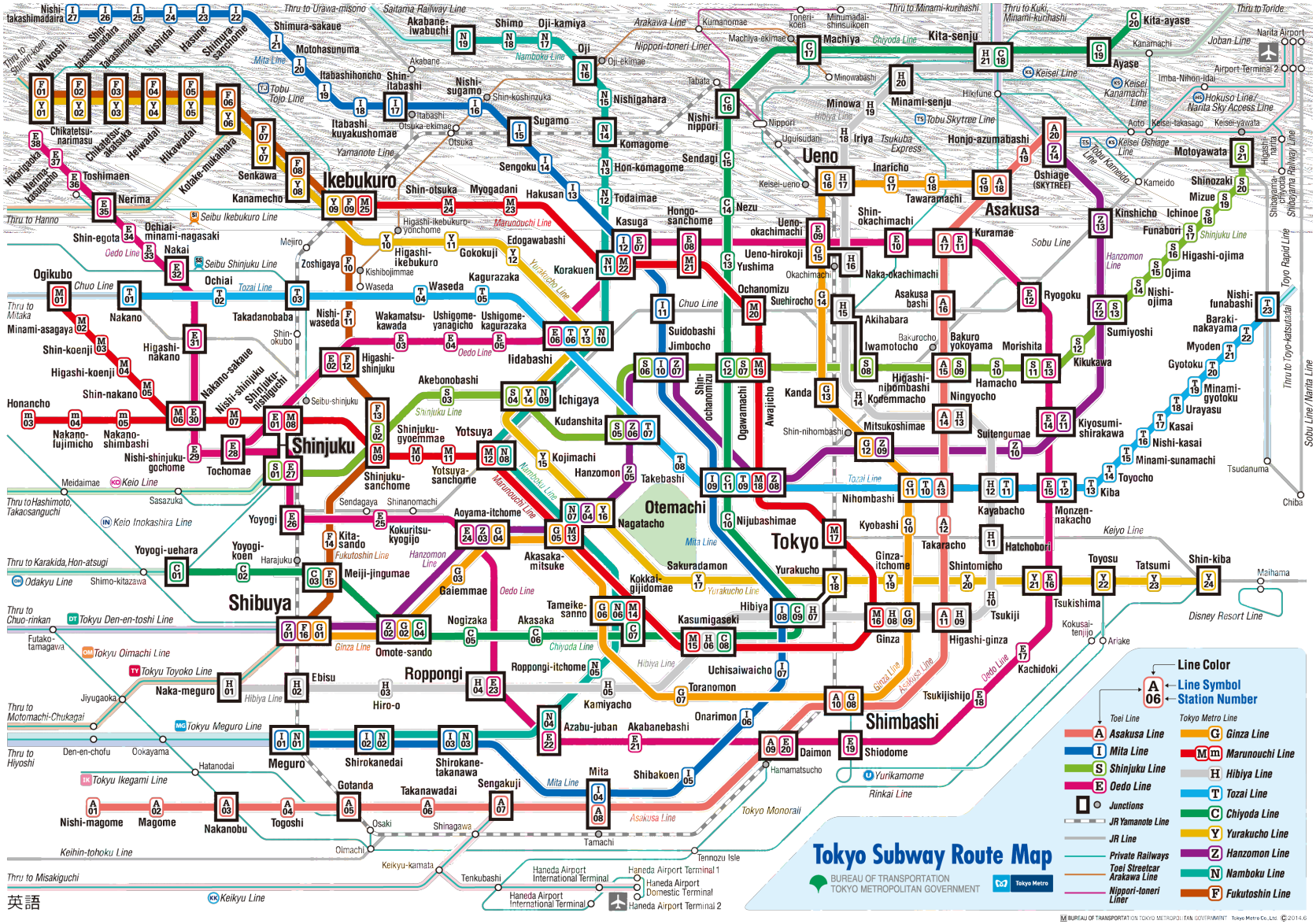
Kuva 19: Kuva: Wikimedia Commons, User:Jpatokal



Kuva 20: Kuva: Flickr, yaph,http://exploringdata.github.io/vis/footballers-search-relations/



Kuva 21: Kuva: Flickr, Andy Lamb



Kuva 22: Kuva: Tokyo Metro

Matematiikassa graafi G ilmoitetaan usein parina $G = (V, E)$.

- V = solmujen joukko
- E = kaarien joukko
- tällöin samojen solmujen välillä saa yleensä olla vain yksi kaari molempiin suuntiin
  - aina tämä ei kuitenkaan käytännön sovelluksessa riitä
  - esimerkiksi juna-aikatauluja esittävässä graafissa kaupunkien välillä on yleensä useampia vuoroja
  - tällaista graafia kutsutaan *monigraafiksi (multigraph)*

Kuva: HSL

- jos solmujen välillä sallitaan vain yksi kaari suuntaansa $\Rightarrow$ $E \subseteq V^2$
  - suunnatulla graafilla $|E|$ voi vaihdella välillä $0, \ldots, |V|^2$
  - laskettaessa graafialgoritmien suoritusaikoja oletamme tämän

  Graafialgoritmin suorituskyky ilmoitetaan yleensä sekä $|V|$:n että $|E|$:n funktiona
  - helppouden vuoksi jätämme itseisarvomerkit pois kertaluokkamerkintöjen sisällä ts. $O(VE) = O(|V| \cdot |E|)$

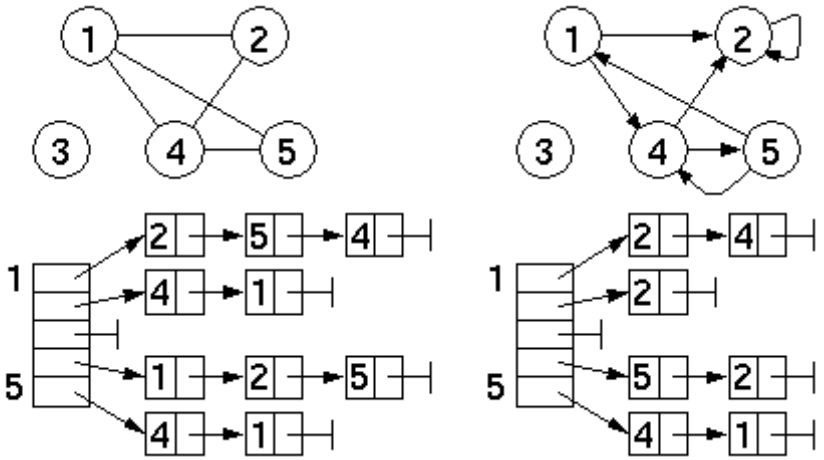
Graafin esittämiseen tietokoneessa on kaksi perusmenetelmää *kytkentälista (adjacency list)* ja *kytkentämatriisi (adjacency matrix)*.

Näistä kahdesta kytkentälistaesitys on yleisempi, ja tällä kurssilla keskitytään siihen.

- Solmut on talletettu johonkin tietorakenteeseen (valittu tietorakenne riippuu siitä, mitä oheisdataa solmut sisältävät, miten niitä pitää pystyä hakemaan, lisäämään jne.)
- Yksinkertaisimmillaan jokaisessa solmussa on tietorakenne, jossa on tallessa mihin solmuihin tästä solmusta on kaari.
  - Valittu tietorakenne riippuu siitä, paljonko kaaria arvellaan olevan, lisätäänkö/poistetaanko niitä jatkuvasti, täytyykö tiettyä kaarta pystyä hakemaan nopeasti jne.)
  - Tieto kohdesolmusta voidaan tallettaa osoittimena, solmun indeksinä (jos solmut indeksoitavassa tietorakenteessa) tms.
  - Solmujen järjestyksellä kytkentälistassa ei yleensä ole väliä

- Graafin kytkentälistojen yhteiskoko on
  - $|E|$, jos graafi on suunnattu, $2 \cdot |E|$, jos graafi on suuntaamaton

  $\Rightarrow$ kytkentälistaesityksen muistin kulutus on $O(\max(V, E)) = O(V + E)$



- Tiedon "onko kaarta solmusta v solmuun u" haku edellyttää yhden kytkentälistan läpi selaamista mikä vie hitaimmillaan aikaa $\Theta(V)$ (ellei solmusta lähteviä kaaria talleteta johonkin tietorakenteeseen, josta niistä pystytään nopeasti hakemaan kohdekaaren perusteella)
- Jos kaari on painotettu tai siihen liittyy oheisdataa, täytyy kaaresta tallettaa kohdesolmun lisäksi myös oheisdata (esim. struct, jossa kohdesolmu ja oheisdata)
- Joskus on tarpeen tallettaa myös tieto solmuun *tulevista* kaarista samaan tapaan (esim. solmujen ja kaarien poistamisen helpottamiseksi)

Kytkentämatriisiesityksen avulla edelliseen kysymykseen pystytään vastaamaan helposti.

- kytkentämatriisi on $|V| \times |V|$ -matriisi A, jonka alkio a_{ij} on
  - 0, jos solmusta i ei ole kaarta solmuun j
  - 1, jos solmusta i on kaari solmuun j
- edellisen esimerkin graafien kytkentämatriisit ovat

	1	2	3	4	5
1	0	1	0	1	1
2	1	0	0	1	0
3	0	0	0	0	0
4	1	1	0	0	1
5	1	0	0	1	0

	1	2	3	4	5
1	0	1	0	1	0
2	0	1	0	0	0
3	0	0	0	0	0
4	0	1	0	0	1
5	1	0	0	1	0

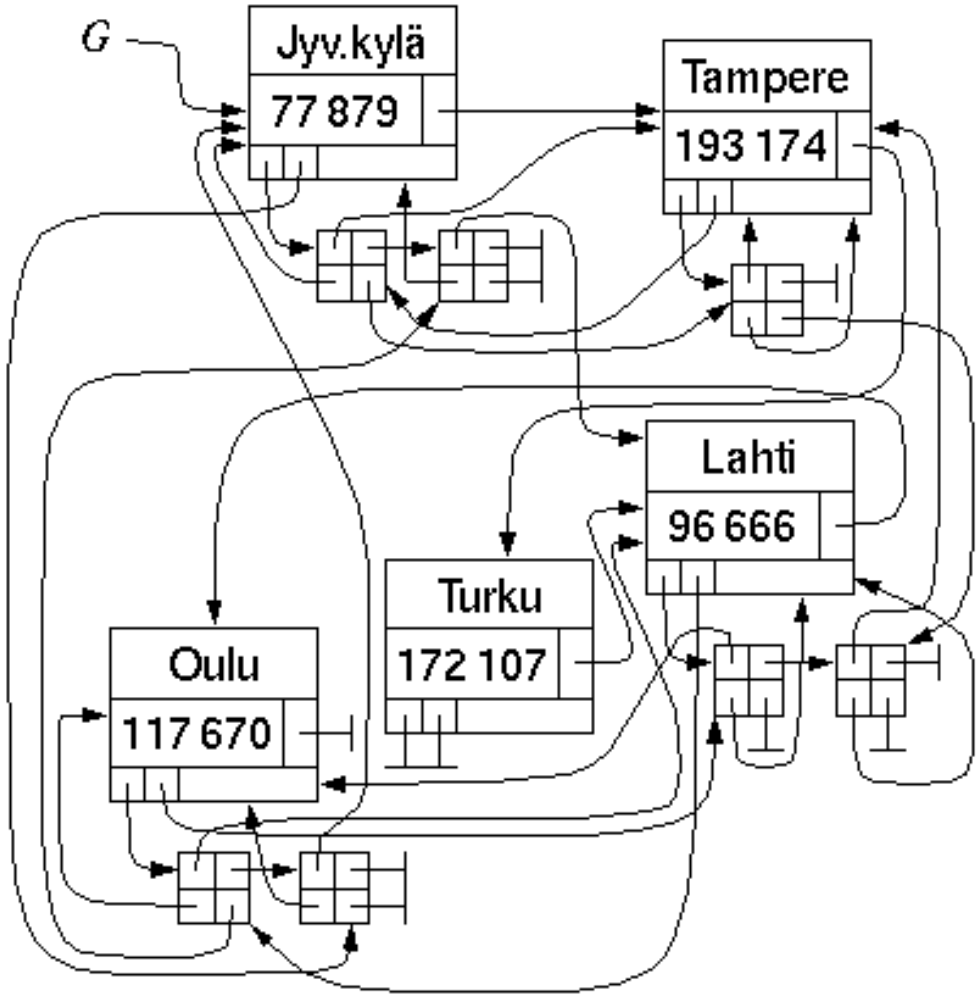
- muistin kulutus on aina $\Theta(V^2)$
  - jokainen alkio tarvitsee vain bitin muistia, joten useita alkioita voidaan tallettaa yhteen sanaan $\Rightarrow$ vakiokerroin saadaan aika pieneksi
- kytkentämatriisiesitystä kannattaa käyttää lähinnä vain hyvin tiheiden graafien yhteydessä.

Tarkastellaan tarkemmin kytkentälistaesitystavan toteutusta:

- käytännön sovelluksissa solmuun kertyy usein monenlaista tehtävän tai algoritmin vaatimaa tietoa
  - nimi
  - bitti, joka kertoo, onko solmussa käyty
  - osoitin, joka kertoo, mistä solmusta tähän solmuun viimeksi tultiin
  - ...

  $\Rightarrow$ solmusta kannattaa tehdä itsenäinen alkio, jossa on tarpeelliset kentät
- yleensä sama pätee myös kaariin

- pääperiaate:
  - talleta jokainen asia yhteen kertaan
  - ota käyttöön osoittimet, joilla voit helposti kulkea haluamiisi suuntiin



13.2 Yleistä graafialgoritmeista

Käsitteitä:

- askel = siirtyminen solmusta toiseen yhtä kaarta pitkin
  - suunnatussa graafissa askel on otettava kaaren suuntaan
- solmun v_2 *etäisyys* (*distance*) solmusta v_1 on lyhimmän v_1:stä v_2:een vievän polun pituus.
  - jokaisen solmun etäisyys itsestään on 0
  - merkitään $\delta(v_1, v_2)$
  - suunnatussa graafissa on mahdollista (ja tavallista), että $\delta(v_1, v_2) \neq \delta(v_2, v_1)$
  - jos v_1:stä ei ole polkua v_2:een, niin $\delta(v_1, v_2) = \infty$

Algoritmien ymmärtämisen helpottamiseksi annamme usein solmuille värit.

- valkoinen = solmua ei ole vielä löydetty
- harmaa = solmu on löydetty, mutta ei loppuun käsitelty
- musta = solmu on löydetty ja loppuun käsitelty
- solmun väri muuttuu järjestyksessä valkoinen $\rightarrow$ harmaa $\rightarrow$ musta
- värikoodaus on lähinnä ajattelun apuväline, eikä sitä välttämättä tarvitse toteuttaa täysin, yleensä riittää tietää, onko solmu löydetty vai ei.
  - usein tämäkin informaatio on pääteltävissä nopeasti muista kentistä

Monet graafialgoritmit perustuvat graafin tai sen osan läpikäyntiin tietyssä järjestyksessä.

- perusläpikäyntijärjestyksiä on kaksi, **leveyteen ensin** -haku (*BFS*) ja **syvyyteen ensin** -haku (*DFS*).
- läpikäynnillä tarkoitetaan algoritmia, jossa
  - käydään kerran graafin tai sen osan jokaisessa solmussa
  - kuljetaan kerran graafin tai sen osan jokainen kaari

Hakualgoritmit käyttävät lähtökohtana jotain annettua graafin solmua, *lähtösolmua* (*source*) ja etsivät kaikki ne solmut, joihin pääsee lähtösolmusta nollalla tai useammalla askeleella.

13.3 Leveyteen ensin -haku (breadth-first)

Leveyteen ensin -hakua voi käyttää esimerkiksi

- kaikkien solmujen etäisyyden määrittämiseen lähtösolmusta
- (yhden) lyhimmän polun löytämiseen lähtösolmusta jokaiseen solmuun

Leveyteen ensin -haun nimi tulee siitä, että se tutkii tutkitun ja tuntemattoman graafin osan välistä rajapintaa koko ajan koko sen leveydeltä.

Solmujen kentät:

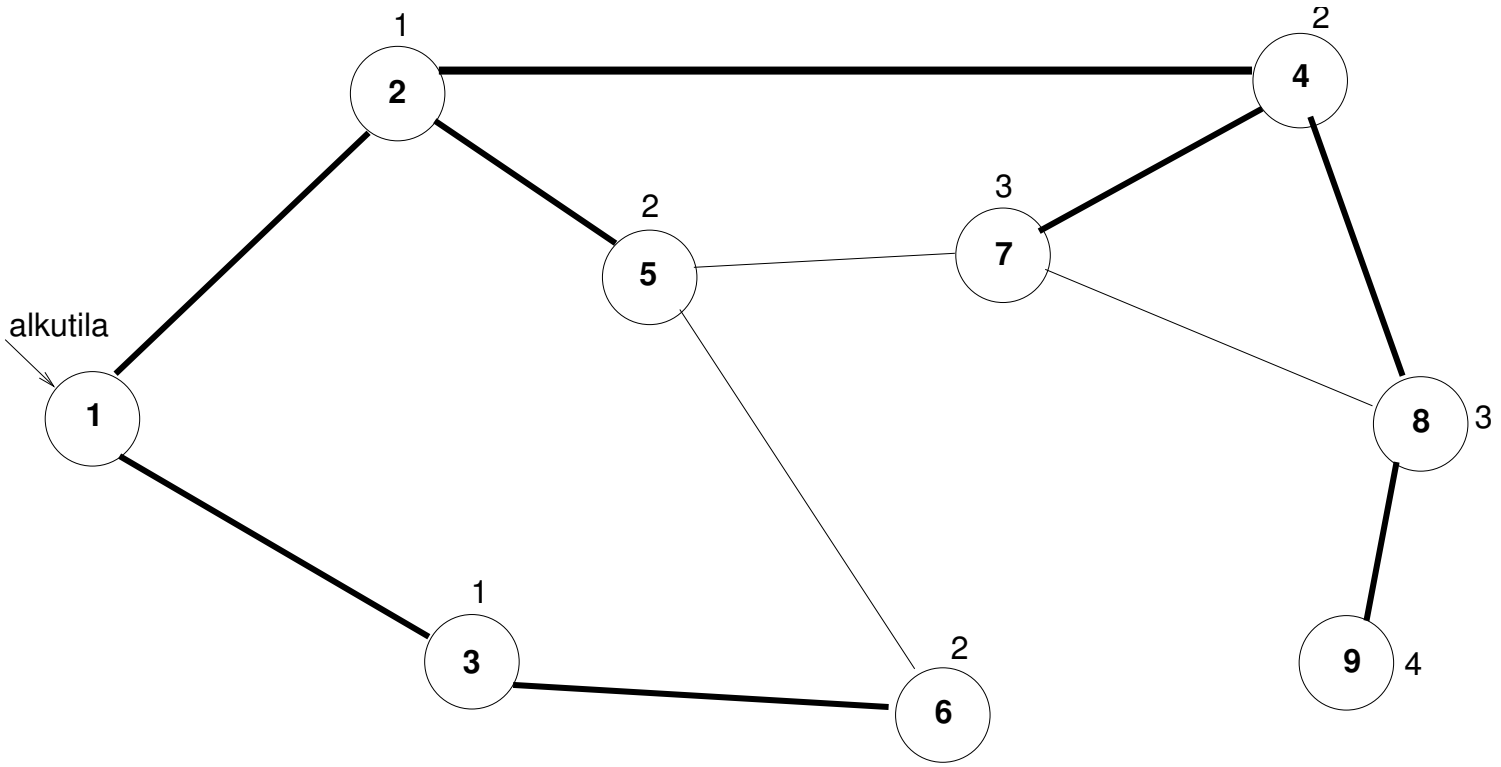
- $v \rightarrow d$ = jos solmu v on löydetty niin sen etäisyys s:tä, muutoin ∞
- $v \rightarrow \pi$ = osoitin solmuun, josta haku ensi kerran tuli v:hen, löytämättömille solmuille NIL
- $v \rightarrow colour$ = solmun v väri
- $v \rightarrow Adj$ = solmun v naapurisolmujen joukko

Algoritmin käyttämä tietorakenne Q on jono (noudattaa FIFO-jonokuria).

```
 BFS(s)                                          (algoritmi saa parametrinaan aloitussolmun s)
1   ▷ alussa kaikkien solmujen kentät ovat arvoiltaan colour = WHITE, d = ∞, π = NIL
2   s→colour := GRAY                            (merkitään alkutila löydetyksi)
3   s→d := 0                                    (etäisyys alkutilasta alkutilaan on 0)
4   PUSH(Q, s)                                  (työnnetään alkutila jonoon)
5   while Q ≠ ∅ do                              (toistetaan niin kauan kun tiloja riittää)
6       u := POP(Q)                             (vedetään jonosta seuraava tila)
7       for each v ∈ u→Adj do                   (käydään u:n naapurit läpi)
8           if v→colour = WHITE then            (jos solmua ei ole vielä löydetty . . . )
9               v→colour := GRAY                (. . . merkitään se löydetyksi)
10              v→d := u→d + 1                  (kasvatetaan etäisyyttä yhdellä)
11              v→π := u                        (tilaan v tultiin tilan u kautta)
12              PUSH(Q, v)                      (työnnetään tila jonoon odottamaan käsittelyä)
13      u→colour := BLACK                       (merkitään tila u käsitellyksi)
```

Kaikkia algoritmin käyttämiä solmun kenttiä ei välttämättä käytännön toteutuksessa tarvita, vaan osan arvoista voi päätellä toisistaan.

Alla olevassa kuvassa graafin solmut on numeroitu siinä järjestyksessä, jossa BFS löytää ne. Solmujen etäisyys alkutilasta on merkitty solmun viereen ja haun kulkureitti tummennettu.



Suoritusaika solmujen (V) ja kaarien (E) määrien avulla ilmaistuna:

- ennen algoritmin kutsumista solmut pitää alustaa
  - järkevässä ratkaisussa tämä on tehtävissä ajassa $O(V)$
- rivillä 7 algoritmi selaa solmun lähtökaaret
  - onnistuu käyttämällämme kytkentälistaesityksellä solmun kaarien määrään nähden lineaarisessa ajassa
- kukin jono-operaatio vie vakiomäärän aikaa
- while-silmukan kierrosten määrä
  - vain valkoisia solmuja laitetaan jonoon
  - samalla solmun väri muuttuu harmaaksi
    $\Rightarrow$ kukin solmu voi mennä jonoon korkeintaan kerran

  $\Rightarrow$ while-silmukka pyörähtää siis korkeintaan $O(V)$ määrän kertoja

- for-silmukan kierrosten määrä
  - algoritmi kulkee jokaisen kaaren korkeintaan kerran molempiin suuntiin

  $\Rightarrow$ for-silmukka käydään läpi yhteensä korkeintaan $O(E)$ kertaa

$\Rightarrow$ koko algoritmin suoritusaika on siis $O(V + E)$

Algoritmin lopetettua π-osoittimet määrittelevät puun, joka sisältää löydetyt solmut, ja jonka juurena on lähtösolmu s.

- *leveyteen ensin -puu* (*breadth-first tree*)
- π-osoittimet määräävät puun kaaret "takaperin"
  - osoittavat juurta kohti
  - $v \rightarrow \pi$ = v:n *edeltäjä* (*predecessor*) eli *isä* (*parent*)
- kaikki lähtösolmusta saavutettavissa olevat solmut kuuluvat puuhun
- puun polut ovat mahdollisimman lyhyitä polkuja s:stä löydettyihin solmuihin

Lyhimmän polun tulostaminen

- kun BFS on asettanut π-osoittimet kohdalleen, lyhin polku lähtösolmusta s solmuun v voidaan tulostaa seuraavasti:

```
PRINT-PATH(G, s, v)
1  if v = s then                    (rekursion pohjatapaus)
2      print s
3  else if v→π = NIL then           (haku ei ole saavuttanut solmua v lainkaan)
4      print "ei polkua"
5  else
6      PRINT-PATH(G, s, v→π)        (rekursiokutsu . . . )
7      print v                      (. . . jonka jälkeen suoritetaan tulostus)
```

- ei-rekursiivisen version voi tehdä esim.
  - kokoamalla solmujen numerot taulukkoon kulkemalla π-osoittimia pitkin, ja tulostamalla taulukon sisällön takaperin
  - kulkemalla polku kahdesti, ja kääntämällä π-osoittimet takaperin kummallakin kertaa (jälkimmäinen käännös ei tarpeen, jos π-osoittimet saa turmella)

13.4 Syvyyteen ensin -haku (depth-first)

Syvyyteen ensin -haku on toinen perusläpikäyntijärjestyksistä.

Siinä missä leveyteen ensin -haku tutkii koko hakurintamaa sen koko leveydeltä, syvyyteen ensin -haku menee yhtä polkua eteen päin niin kauan kuin se on mahdollista.

- polkuun hyväksytään vain solmuja, joita ei ole aiemmin nähty
- kun algoritmi ei enää pääse eteenpäin, se peruuttaa juuri sen verran kuin on tarpeen uuden etenemisreitin löytämiseksi, ja lähtee sitä pitkin
- algoritmi lopettaa, kun se peruuttaa viimeisen kerran takaisin lähtösolmuun, eikä löydä enää sieltäkään tutkimattomia kaaria

Algoritmi muistuttaa huomattavasti leveyteen ensin -haun pseudokoodia.

Merkittäviä eroja on oikeastaan vain muutama:

- jonon sijasta käsittelyvuoroaan odottavat tilat talletetaan pinoon
- algoritmi ei löydä lyhimpiä polkuja, vaan ainoastaan jonkin polun
  - tästä syystä esimerkkipseudokoodia on yksinkertaistettu jättämällä π-kentät pois

Algoritmin käyttämä tietorakenne S on pino (noudattaa LIFO-jonokuria).

```
 DFS(s)                                          (algoritmi saa parametrinaan aloitussolmun s)
1   ▷ alussa kaikkien (käsittelemättömien) solmujen värikenttä colour = WHITE
2   PUSH(S, s)                                  (työnnetään alkutila pinoon)
3   while S ≠ ∅ do                              (jatketaan niin kauan kun pinossa on tavaraa)
4       u := POP(S)                             (vedetään pinosta viimeisin sinne lisätty tila)
5       if u→colour = WHITE then                (jos solmua ei ole vielä käsitelty . . . )
6           u→colour := GRAY                    (merkitään tila käsittelyssä olevaksi)
7           PUSH(S, u)                          (työnnetään taas pinoon (mustaksi värjäys))
8           for each v ∈ u→Adj do               (käydään u:n naapurit läpi)
9               if v→colour = WHITE then        (jos solmua ei ole vielä käsitelty . . . )
10                  PUSH(S, v)                  (. . . työnnetään se pinoon odottamaan käsittelyä
11              else if v→colour = GRAY then    (harmaa solmu! Sykli löytynyt! . . . )
12                  ????                        (käsittele sykli, jos se kiinnostaa)
13      else
14          u→colour := BLACK                   (kaikki lapset käsitelty, solmu on valmis)
```

Jos halutaan tutkia koko graafi, voidaan kutsua syvyyteen ensin -hakua kertaalleen kaikista vielä tutkimattomista solmuista.

- tällöin solmuja ei väritetä valkoisiksi kutsukertojen välillä

Rivin 5 perään voitaisi lisätä operaatio, joka kaikille graafin alkioille halutaan tehdä. Voidaan esimerkiksi

- tutkia onko tila maalitila, ja lopettaa jos on
- ottaa talteen solmuun liittyvää oheisdataa
- muokata solmuun liittyvää oheisdataa

Suoritusaika voidaan laskea samoin kuin leveyteen ensin -haun yhteydessä:

- ennen algoritmin kutsumista solmut pitää alustaa
  - järkevässä ratkaisussa tämä on tehtävissä ajassa $O(V)$
- rivillä 6 algoritmi selaa solmun lähtökaaret
  - onnistuu käyttämällämme kytkentälistaesityksellä solmun kaarien määrään nähden lineaarisessa ajassa
- kukin pino-operaatio vie vakiomäärän aikaa
- while-silmukan kierrosten määrä
  - vain valkoisia solmuja laitetaan pinoon
  - samalla solmun väri muuttuu harmaaksi
    $\Rightarrow$ kukin solmu voi mennä pinoon korkeintaan kerran

  $\Rightarrow$ while-silmukka pyörähtää siis korkeintaan $O(V)$ määrän kertoja

- for-silmukan kierrosten määrä
  - algoritmi kulkee jokaisen kaaren korkeintaan kerran molempiin suuntiin

  $\Rightarrow$ for-silmukka käydään läpi yhteensä korkeintaan $O(E)$ kertaa

$\Rightarrow$ koko algoritmin suoritusaika on siis $O(V + E)$

DFS on myös mahdollista toteuttaa rekursiivisesti, jolloin algoritmin pinona toimii funktioiden kutsupino.

- Rekursiivinen versio on itse asiassa jonkin verran yksinkertaisempi kuin iteratiivinen versio!
- (Keksitkö, miksi?)

Huom! Ennen algoritmin kutsumista kaikki solmut tulee alustaa valkoisiksi!

```
 DFS(u)
1   u→colour := GRAY                    (merkitään tila löydetyksi)
2   for each v ∈ u→Adj do               (käydään kaikki u:n naapurit läpi)
3       if v→colour = WHITE then        (jos ei olla vielä käyty v:ssä...)
4           DFS(v)                      (...jatketaan etsintää rekursiivisesti tilasta v)
5       else if v→colour = GRAY then    (jos on jo käyty, muttei loppuun käsitelty ...)
6           ▷ silmukka on löytynyt      (...silmukka on löytynyt)
7   u→colour := BLACK                   (merkitään tila käsitellyksi)
```

Suoritusaika:

- rekursiivinen kutsu tehdään ainoastaan valkoisille solmuille
- funktion alussa solmu väritetään harmaaksi
  $\Rightarrow$ DFS:ää kutsutaan korkeintaan $O(V)$ kertaa
- kuten aiemmassakin versiossa for-silmukka kiertää korkeintaan kaksi kierrosta kutakin graafin kaarta kohden koko algoritmin suorituksen aikana
  $\Rightarrow$ siis for-silmukan kierroksia tulee korkeintaan $O(E)$ kappaletta
- muut operaatioista ovat vakioaikaisia

$\Rightarrow$ koko algoritmin suoritusaika on edelleen $O(V + E)$

Leveyteen ensin -haku vai syvyyteen ensin -haku:

- lyhimmän polun etsimiseen täytyy käyttää leveyteen ensin -hakua
- jos graafin esittämä tilavaruus on hyvin suuri, käyttää leveyteen ensin -haku yleensä huomattavasti enemmän muistia
  - syvyyteen ensin -haun pinon koko pysyy yleensä pienempänä kuin leveyteen ensin -haun jonon koko
  - useissa sovelluksissa esimerkiksi tekoälyn alalla jonon koko estää leveyteen ensin -haun käytön
- mikäli graafin koko on ääretön, ongelmaksi nousee se, ettei syvyyteen ensin -haku välttämättä löydä ikinä maalitilaa, eikä edes lopeta ennen kuin muisti loppuu
  - näin tapahtuu, jos algoritmi lähtee tutkimaan hedelmätöntä äärettömän pitkää haaraa
  - tätä ongelmaa ei kuitenkaan esiinny äärellisten graafien yhteydessä

- syvyyteen ensin -haun avulla voi ratkaista joitakin monimutkaisempia ongelmia, kuten graafin silmukoiden etsintä
  - harmaat solmut muodostavat lähtösolmusta nykyiseen solmuun vievän polun
  - mustista solmuista pääsee vain mustiin ja harmaisiin solmuihin

    $\Rightarrow$ jos nykyisestä solmusta pääsee harmaaseen solmuun, niin graafissa on silmukka

14 Lyhimmät painotetut polut

BFS löytää lyhimmän polun lähtösolmusta graafin saavutettaviin solmuihin.

Se ei kuitenkaan enää suoriudu tehtävästä, jos kaarien läpi kulkeminen maksaa askelta enemmän.

Tässä luvussa käsitellemme lyhimpien painotettujen polkujen etsintää graafista, jonka kaaripainot ovat positiivisia ja voivat poiketa ykkösestä.

- negatiivisten kaaripainojen hallitsemiseen tarvitaan monimutkaisempia algoritmeja, esimerkiksi Bellman-Ford algoritmi

14.1 Lyhin polku

Graafin kaarilla voi olla ominaisuus nimeltä *paino(weight)*.

- paino voi edustaa vaikkapa reitin pituutta tai tilasiirtymän kustannusta
- Graafin $G = (V, E)$ painofunktio $w : E \to \mathbb{R}$ kaarilta reaalilukupainoille
- Polun $p = \langle v_0, v_1, ..., v_k \rangle$ paino $w(p)$ on sen muodostavien kaarten painojen summa $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$.

Määritelmä: lyhimmän polun paino $\delta(u, v)$:

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{jos on polku} u\text{:sta } v\text{:hen,} \\ \infty & \text{muuten.} \end{cases}$$

ja siten lyhin polku u:sta v:hen mikä tahansa polku p, jolle $w(p) = \delta(u, v)$

Tämä mutkistaa lyhimmän reitin etsintää merkittävästi.

- lyhin reitti on se lähtösolmusta etsittyyn solmuun kulkeva polku, jonka kaarien painojen summa on mahdollisimman pieni
- jos jokaisen kaaren paino on 1, tehtävä voidaan ratkaista käymällä lähtösolmusta saavutettavissa oleva graafin osa läpi leveyteen ensin -järjestyksessä
- jos painot saattavat olla < 0, voi olla, että tehtävään ei ole ratkaisua, vaikka polkuja olisi olemassakin
  - jos graafissa on silmukka, jonka kaaripainojen summa on negatiivinen saadaan mielivaltaisen pieni painojen summa kiertämällä silmukkaa tarpeeksi monta kertaa

14.2 Dijkstran algoritmi

Suunnatun, painotetun graafin $G = (V, E)$, jossa kaaripainot ovat ei-negatiivisia, lyhimmät painotetut polut lähtösolmusta voi etsiä Dijkstran algoritmilla.

- etsii lyhimmät polut lähtösolmusta s kaikkiin saavutettaviin solmuihin, painottaen kaarien pituuksia w:n mukaan
- valitsee joka tilanteessa tutkittavakseen lyhimmän polun, jota se ei ole vielä tutkinut

  $\Rightarrow$ se on siis ahne algoritmi
- oletus: $w(u, v) \geq 0 \; \forall (u, v) \in E$

```
DIJKSTRA(s, w)                          (algoritmi saa parametrinaan aloitussolmun s)
1   ▷ alussa kaikkien solmujen kentät ovat arvoiltaan colour = WHITE, d = ∞, π = NIL
2   s→colour := GRAY                    (merkitään alkutila löydetyksi)
3   s→d := 0                            (etäisyys alkutilasta alkutilaan on 0)
4   PUSH(Q, s)                          (työnnetään alkutila prioriteettijonoon)
5   while Q ≠ ∅ do                      (jatketaan niin kauan kun solmuja riittää)
6       u := EXTRACT-MIN(Q)             (otetaan prioriteettijonosta seuraava tila)
7       for each v ∈ u→Adj do           (käydään u:n naapurit läpi)
8           if v→colour = WHITE then    (jos solmussa ei ole käyty . . . )
9               v→colour := GRAY        (. . . merkitään se löydetyksi)
10              PUSH(Q, v)              (työnnetään tila jonoon odottamaan käsittelyä)
11          RELAX(u, v, w)
12      u→colour := BLACK               (merkitään tila u käsitellyksi)

RELAX(u, v, w)
1   if v→d > u→d + w(u, v) then         (jos löydettiin uusi lyhyempi reitti tilaan v...)
2       v→d := u→d + w(u, v)            (...pienennetään v:n etäisyyttä lähtösolmusta)
3       v→π := u                        (merkitään, että v:n tultiin u:sta)
```

Algoritmin käyttämä tietorakenne Q on prioriteettijono (luentomonisteen kohta 3.2).

w sisältää kaikkien kaarien painot.

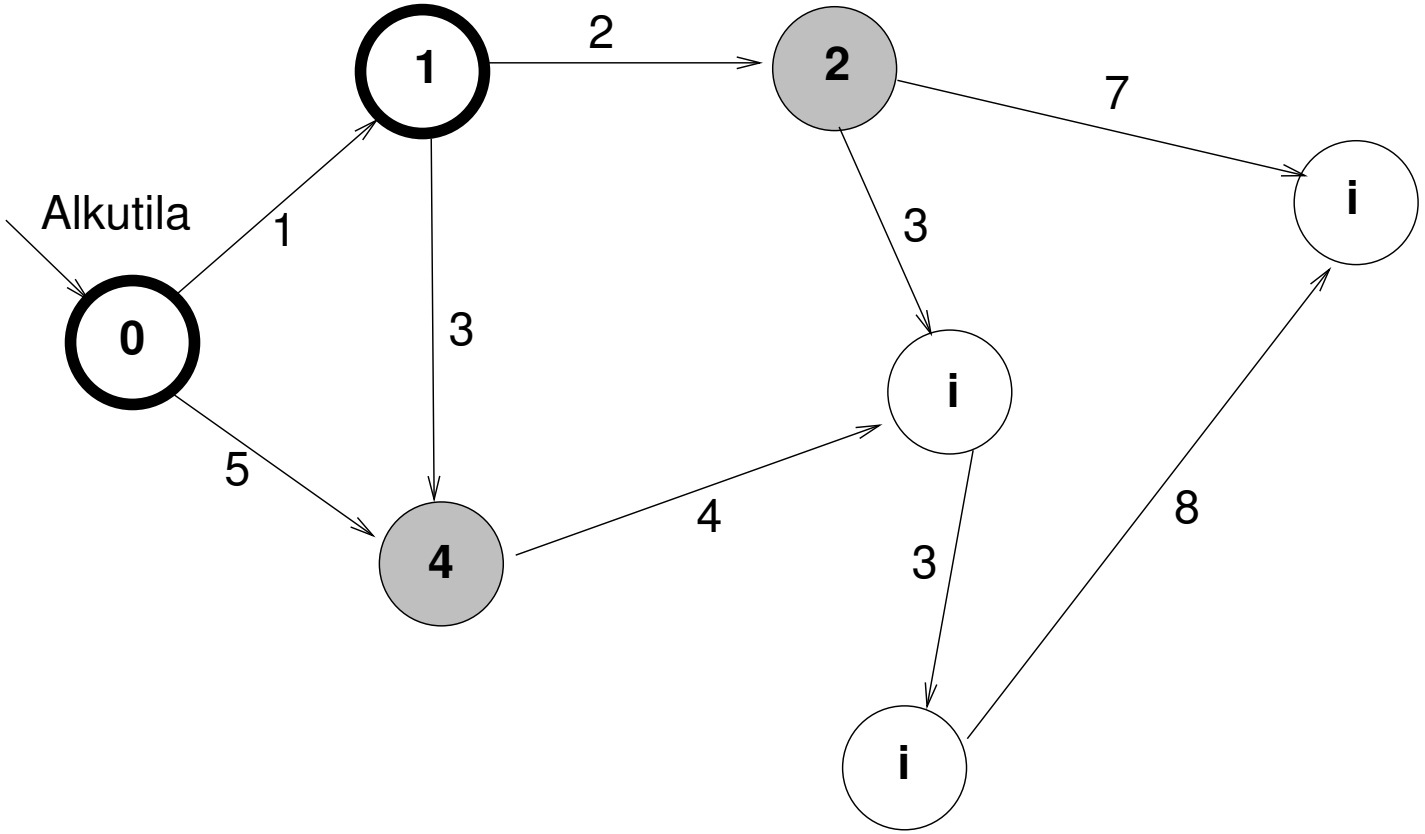
Dijkstran algoritmi käyttää apufunktiota RELAX

- kaaren (u, v) *relaksointi* (*relaxation*) testaa, voiko lyhintä löydettyä v:hen vievää polkua parantaa reitittämällä sen loppupää u:n kautta, ja tarvittaessa tekee niin

Muilta osin algoritmi muistuttaa huomattavasti leveyteen ensin -hakua.

- se löytää lyhimmät polut kasvavan pituuden mukaisessa järjestyksessä
- kun solmu u otetaan Q:sta, sen painotettu etäisyys s:stä varmistuu $u \rightarrow d$:ksi
  - jos prioriteettijonosta otettu tila on maalitila, voidaan algoritmin suoritus lopettaa saman tien

Alla olevassa kuvassa nähdään Dijkstran algoritmi tilanteessa, jossa mustalla ympyröidyt solmut on käsitelty.



Suoritusaika:

- while-silmukka käy enintään $O(V)$ kierrosta ja for-silmukka yhteensä enintään $O(E)$ kierrosta
- prioriteettijonon voi toteuttaa tehokkaasti keon avulla tai vähemmän tehokkaasti listan avulla

	kekototeutuksella:	listatoteutuksella:	
rivi 4:	$\Theta(1)$	$\Theta(1)$	(tyhjään tietorakenteeseen lisääminen)
rivi 5:	$\Theta(1)$	$\Theta(1)$	(onko prioriteettijono tyhjä)
rivi 6:	$O(\lg V)$	$O(V)$	(Extract-Min)
rivi 10:	$\Theta(1)$	$\Theta(1)$	(valkoisen solmun prioriteetti on ääretön, joten sen oikea paikka on keon lopussa)
rivi 11:	$O(\lg V)$	$\Theta(1)$	(relaksoinnissa solmun prioriteetti voi muuttua)

- käytettäessä kekototeutusta jokaisella while- ja for-silmukan kierroksella suoritetaan yksi $O(\lg V)$ aikaa kuluttava operaatio

  $\Rightarrow$ algoritmin suoritusaika on $O((V + E) \lg V)$

14.3 A*-algoritmi

Dijkstran algoritmi etsi lyhimmän painotetun reitin kartoittamalla solmuja lyhimmästä reitistä alkaen reitin pituusjärjestyksessä. Eli: Dijkstra käyttää hyväkseen vain jo kuljetuista kaarista saatavaa tietoa.

A*-algoritmi tehostaa tätä lisäämällä *heuristiikan* (=oletuksen) lyhimmästä mahdollisesta etäisyydestä maaliin. (Esim. maantiereitin haussa etäisyys linnuntietä).

- etsii lyhimmän painotetun polun lähtösolmusta s annettuun maalisolmuun g. **Ei** kartoita lyhintä reittiä kaikkiin solmuihin (kuten Dijkstra), vain maalisolmuun.
- edellyttää, että painot ovat ei-negatiivisia (kuten Dijkstrakin)
- edellyttää, että jokaiselle solmulle voidaan laskea sen minimietäisyys maalista (ts. löytynyt lyhin reitti ei voi olla lyhempi).
- valitsee joka tilanteessa tutkittavakseen ei-tutkitun solmun, jossa (lyhin etäisyys lähdöstä solmuun + arvioitu minimietäisyys maaliin) on pienin.

Ainoa ero A*:n ja Dijkstran välillä on relaksointi (ja se, että A* kannattaa lopettaa heti maalisolmun löydyttyä, koska se ei kartoita lyhimpiä etäisyyksiä kaikkiin solmuihin).

RELAX-A*(u, v, w)

```
1  if v→d > u→d + w(u, v) then     (jos löydettiin uusi lyhyempi reitti tilaan v...)
2       v→d := u→d + w(u, v)       (...uusi pituus tähän saakka...)
3       v→de := v→d + min_est(v, g)  (...ja minimiarvio koko reitistä)
4       v→π := u                   (merkitään, että v:n tultiin u:sta)
```

A*:n käyttämässä prioriteettijonossa käytetään prioriteettina koko reitin pituusarviota $v{\rightarrow}de$.

(Dijkstran algoritmi on A*:n erikoistapaus, jossa $min\_est(a, b)$ on aina 0.)

14.4 Kevyin virittävä puu

Graafin $G = (V, E)$ kevyin virittävä puu on sen asyklinen aligraafi, joka yhdistää kaikki graafin solmut niin, että aligraafin kaarien painojen summa on pienin mahdollinen.

Puun löytämiseksi on kaksi algoritmia: Primin ja Kruskalin

Prim muistuttaa Dijkstran algoritmin kun taas Kruskal lähestyy ongelmaa luomalla metsän, jossa on puu jokaiselle puulle ja sitten yhdistämällä näitä puuksi