- Joskus on tarpeen esittää *ala*rajoja:

  $f(n) = \Omega(g(n))$ jos ja vain jos on vakiot $c > 0$ ja n_0 siten, että kaikilla $n \geq n_0$ pätee ehto $0 \leq c \cdot g(n) \leq f(n)$.

  Eli $f(n)$ nousee kohdassa n_0 kuvaajan $c \cdot g(n)$ *ylä*puolelle.

- Merkintä $f(n) = \Theta(g(n))$ tarkoittaa "$f(n) = \mathcal{O}(g(n))$ ja $f(n) = \Omega(g(n))$".
  - Siis on vakiot $c_1 \leq c_2$ ja kohta n_0 siten, että $f(n)$ jää kuvaajien $c_1 \cdot g(n)$ ja $c_2 \cdot g(n)$ *väliin*.
  - Siis $f(n)$ käyttäytyy "oleellisesti" kuten $g(n)$.
  - Funktion $f(n)$ heilahtelu voi ulottua vain vakiokertoimen päähän funktiosta $g(n)$.

- Joskus halutaan korostaa, että *funktion $f(n)$ kertaluokka-arvio $g(n)$ ei ole tarkka* vaan niiden väliin jää "ilmaa".
  - $f(n) = o(g(n))$ jos *jokaiselle* vakiolle $c > 0$ on kohta n_0 josta alkaen $0 \leq f(n) < c \cdot g(n)$.
    - Funktio $f(n)$ hiipuu merkityksettömäksi funktion $g(n)$ rinnalla kun n kasvaa.
    - Intuitio: $f(n) = \mathcal{O}(g(n))$ mutta $g(n) \neq \mathcal{O}(f(n))$.
  - Vastaavasti merkinnälle Ω:

    $f(n) = \omega(g(n))$ jos jokaiselle vakiolle $c > 0$ on kohta n_0 josta alkaen $0 \leq c \cdot g(n) < f(n)$.
    - Funktio $g(n)$ hiipuu merkityksettömäksi funktion $f(n)$ rinnalla kun n kasvaa.
    - Intuitio: $g(n) = o(f(n))$.

- Asymptoottiset merkinnät sallitaan myös *lausekkeiden sisällä*.

  Yhtäsuuruusmerkin (tai epäyhtälömerkin)

  joko oikealla tai vasemmalla puolella sellainen tarkoittaa *tuntematonta funktiota kyseisestä kertaluokasta*.

  Esimerkiksi *Stirlingin kaava*

  $$n! = \sqrt{2\pi \cdot n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right) \qquad (2)$$

  tarkoittaa jotakin lauseketta

  $$\sqrt{2\pi \cdot n}\left(\frac{n}{e}\right)^n (1 + f(n))$$

  missä $f(n) = \Theta(\frac{1}{n})$, mutta tarkemmin emme tiedä (emmekä välitä).

  kummallakin puolella samaa, mutta tuntemattomat valitaan *ensin vasemmalle, sitten oikealle*:

  $$\text{yhtälö (2)} = \mathcal{O}\left(\sqrt{2\pi \cdot n}\left(\frac{n}{e}\right)^n\right)$$

  1. Vasemmalle jokin $f(n) \leq \frac{c}{n}$ kun $n \geq n_0$.
  2. Silloin $f(n) \leq c$ joka katoaa oikealla.

- Tarkastelut lyhentyvät:

Lause 3.3.5. *Binäärihaun aikavaatimus lauseesta 3.2.2 on* $\lceil \log_2(n+1) \rceil + 2 = \mathcal{O}(\log_2(n))$.

Todistus.

$$\begin{aligned}\lceil \log_2(n+1) \rceil + 2 &\leq \log_2(n+1) + 3 \\ &= \mathcal{O}(\log_2(n+1)) \\ &\quad \text{lauseesta 3.3.3} \\ &= \mathcal{O}(\log_2(n)) \\ &\quad \text{lauseista 3.3.3} \\ &\quad \text{ja 3.3.4.}\end{aligned}$$

□

3.3.1 Ohjelmoijan $\mathcal{O}$-sääntöjä

Perusoperaatiot vievät vakioajan $\mathcal{O}(1)$.

Peräkkäisyys käsiteltiin lauseessa 3.3.1.

Ehtolause muotoa

if ehto joka vie ajan $\mathcal{O}(f(n))$ **then**
  jotakin joka vie ajan $\mathcal{O}(g(n))$
else
  jotakin muuta joka vie ajan $\mathcal{O}(h(n))$
end if

- *Perusoletus:* Suoritus etenee *joka kerran raskainta reittiä*, jolloin

  $$\mathcal{O}(\mathcal{O}(f(n)) + \max(\mathcal{O}(g(n)), \mathcal{O}(h(n))))$$

  eli *korkein* kertaluokista $\mathcal{O}(f(n))$, $\mathcal{O}(g(n))$ ja $\mathcal{O}(h(n))$.

- Jos tiedetään muuta, niin voidaan saada tarkempi arvio.

Rekursiivinen aliohjelma käsitellään palautuskaavoilla lauseen 3.1.2 tapaan:

1. Oletetaan että meillä jo olisi $\mathcal{O}$-lauseke nimeltään $T(n)$ ="rekursiossa käytetty aika kun parametrina saadun syötteen pituus on n".

2. Kirjoitetaan $\mathcal{O}$-lauseke $f(n)$ ="aliohjelman rungossa käytetty aika kun parametrina saadun syötteen pituus on n".

   Kun rungossa on rekursiokutsu, niin lausekkeeseen laitetaan $T(m)$ missä $m < n$ on kutsussa lähtevän syötteen pituus.

   Jos $m \geq n$, niin

   - joko rekursio ei pääty — eli algoritmi ei toimi
   - tai rekursiota ohjaa jokin toinen suure k kuin n — eli analyysissä pitääkin laskea miten k riippuu suureesta n.

3. Yritetään ratkaista palautuskaava $T(n) = f(n)$.

Toisto on rekursion erikoistapaus kuten kalvoilla 2.3.

Jos toistokerroille tiedetään *etukäteen yläraja*, niin helpompaa:

r := lkm(n);
for all i := 1 **to** r **do**
  jotakin joka vie ajan $\mathcal{O}(f(n))$
end for

on selvästi
lkm$(n) \cdot \mathcal{O}(f(n)) = \mathcal{O}(\text{lkm}(n) \cdot f(n))$.

Joskus silmukan rungon suoritusaika riippuu (myös) silmukkaindeksistä i:

r := lkm(n);
for all i := 1 **to** r **do**
  jotakin joka vie ajan $\mathcal{O}(f(i))$
end for

Silloin tarkempi arvio on summa

$$\sum_{1 \leq i \leq \text{lkm}(n)} \mathcal{O}(f(i)) = \mathcal{O}\left(\sum_{1 \leq i \leq \text{lkm}(n)} f(i) \right).$$

Huomaa: Ei lauseesta 3.3.1!

Sisäkkäisten toistosilmukoiden

r := lkm$_1(n)$;
for all i := 1 **to** r **do**
  s := lkm$_2(i)$;
  for all j := 1 **to** s **do**
    jotakin joka vie ajan $\mathcal{O}(f(j))$
  end for
end for

analyysi antaa sisäkkäisen summan

$$\sum_{1 \leq i \leq \text{lkm}_1(n)} \sum_{1 \leq j \leq \text{lkm}_2(i)} \mathcal{O}(f(j)).$$

Usein sisäsilmukan raja ja runko eivät riipukaan ulkosilmukasta:

r := lkm$_1(n)$;
for all i := 1 **to** r **do**
  s := lkm$_2(n)$;
  for all j := 1 **to** s **do**
    jotakin joka vie ajan $\mathcal{O}(f(n))$
  end for
end for

antaa $\mathcal{O}(\text{lkm}_1(n) \cdot \text{lkm}_2(n) \cdot f(n))$.

3.4 Lisäyslajittelu

- Käsitellään *lisäyslajittelu* (Insertion Sort) esimerkkinä algoritmin
  - kehittämisestä invariantein
  - analysoimisesta $\mathcal{O}$-säännöin.
- Kehitetään siis koodinpätkää, jonka

  alussa saadaan kokonaislukutaulukko $A[1 \ldots N]$

  lopussa A sisältää samat luvut kuin alussakin, mutta nyt järjestyksessä.
- Keksitään ulkosilmukalle "hyvä" invariantti:

  for j := 2 **to** N **do**
  Alkuosa $A[1 \ldots j-1]$ sisältää samat luvut kuin alussakin, mutta nyt järjestyksessä;
  Miten laajennat sen myös paikkaan $A[j]$*?*
  end for

- Laajentamisellekin keksitään invariantti:

  Haetaan indeksi $0 \leq i \leq j-1$ jolla paikan $A[j]$ alkuperäinen sisältö a on

  vähintään yhtä suuri kuin osan $A[1 \ldots i]$ suurin luku ($= A[i]$ jos $1 \leq i$)

  pienempi kuin osan $A[i+1 \ldots j-1]$ pienin luku ($= A[i+1]$ jos $i+i \leq j-1$).

  Lisätään a paikkojen $A[i]$ ja $A[i+1]$ väliin.

  Siirretään sen tieltä osa $A[i+1 \ldots j-1]$ paikkoihin $A[i+2 \ldots j]$.

  for j := 2 **to** N **do**
  a := $A[j]$;
  i := $j-1$;
  while $i > 0$ **do**
  Osa $A[1 \ldots i]$ on yhä hakematta, mutta osa $A[i+1 \ldots j-1]$ on jo siirretty paikkoihinsa $A[i+2 \ldots j]$;
  Miten haet ja siirrät kohdassa i*?*
  end while
  end for

- Hakeminen ja siirto voidaan toteuttaa lisäämällä ne koodinpalat, jotka saavat invariantit voimaan:

```
1: for j := 2 to N do
2:     a := A[j];
3:     i := j − 1;
4:     while i > 0 and A[i] > a do
5:         A[i + 1] := A[i];
6:         i := i − 1;
7:     end while;
8:     A[i + 1] := a
9: end for
```

- $\mathcal{O}$-analyysi:

  Sisäsilmukan runko (rivit 5–6) on $\mathcal{O}(1)$.

  Sisäsilmukka (rivit 4–7) pyörähtää $\mathcal{O}(j)$ kertaa kullakin ulkosilmukan j.

  Ulkosilmukan runko (rivit 2–8) on siis myös $\mathcal{O}(j)$.

  Ulkosilmukka (rivit 1–9) pyörähtää $\mathcal{O}(N)$ kertaa.

- Saadaan summa

$$\begin{aligned}\sum_{2 \leq j \leq N} \mathcal{O}(j) &= \mathcal{O}\left(\sum_{2 \leq j \leq N} j\right) \\ &= \mathcal{O}\left((N-1) \cdot \frac{2+N}{2}\right) \\ &= \mathcal{O}\left(\frac{N^2}{2} + \frac{N}{2} - 1\right) \\ &= \mathcal{O}(N^2).\end{aligned}$$

- Saatiin *polynomi*

  $p(x) = a_d \cdot n^d + a_{d-1} \cdot n^{d-1} + a_{d-2} \cdot n^{d-2} + \cdots + a_0$.

  Sellaisen $\mathcal{O}$-arvio on *korkeimman asteen termi* n^d *ilman kerrointa* $a_d \neq 0$.

  Loput termit voidaan jättää laskematta heti kun aste on selvinnyt!

3.5 Logaritmeista

- Logaritmifunktiot $\log(n)$ ovat yleisiä algoritmien analyysissä.
- Logaritmi $\log_p(n)$ on eksponenttifunktion p^n käänteisfunktio: $\log_p(p^n) = n = p^{\log_p(n)}$.
- Logaritmit *kasvavat todella hitaasti!*

  Koska eksponenttifunktiot kasvavat todella *nopeasti!*
- Varsinkin 2-kantainen logaritmi $\log_2(n)$.
- Esimerkiksi kun kalvojen 2.1 hajoita ja hallitse -periaatteessa *onnistutaan jakamaan syöte (melkein) tasan kahtia*.

  Kuten kalvojen 2.4.2 binäärihaussa.
- Logaritmin *kantaluvun voi unohtaa* $\mathcal{O}$-tarkkuudella:

$$\log_p(n) = \frac{\log_q(n)}{\log_q(p)}.$$

- Funktio $\lceil \log_p(m+1) \rceil$ kertoo, kuinka monta merkkiä pitkä luvun $m \in \mathbb{N}$ *tekstiesitys* on kirjoitettuna p-kantaisessa numerojärjestelmässä:

  binäärijärjestelmässä $p = 2$ eli bitteinä

  kymmenjärjestelmässä $p = 10$ eli tuttuina numeroina.
- *Numeerisissa* algoritmeissa se on usein järkevämpi syötteen koon mitta:

```
for i := 2 to ⌈√n⌉ do
  if (n mod i) = 0 then
    return i
  end if
end for;
return 1.
```

  – Silmukka pyörii $\mathcal{O}(\sqrt{n})$ kertaa.

  – Eikö sen siis tarvitse käydä läpi edes koko syötelukua n?

  – Intuitiivisesti kyllä, jopa monta kertaa.

- Lausutaankin aikavaatimus sen suhteen, montako *bittiä* algoritmi sai syötteenään:

$$\begin{aligned}\mathcal{O}\left(\sqrt{n}\right) &= \mathcal{O}\left(\sqrt{2^{\log_2(n)}}\right) \\ &= \mathcal{O}\left(2^{(\log_2(n)/2)}\right).\end{aligned}$$

  – Algoritmi onkin *eksponentiaalinen* saamiensa bittien lukumäärän suhteen!

  – Kuulostaa järkevämmältä: aritmetiikka (mod, + ja =) tekee paljon työtä yksittäistä bittiä kohden.

  – Keskeinen tehtävä *kryptografiassa* ja *vaativuusteoriassa*.
- Toinen tapa ajatella samaa asiaa:

  – $\mathcal{O}(\sqrt{n})$-analyysi oletti, että luku n mahtuu *yhteen* muistipaikkaan.

  – Mutta kun n kasvaa, niin se vuotaakin yli: yhden muistipaikan koolla on (laskentamallikohtainen) yläraja.

  – Siksi syöte pitikin tulkita $\mathcal{O}(\log(n))$ bitin *taulukoksi*.

4 Listarakenteet

- Kurssin ensimmäinen varsinainen tietorakenne (taulukon jälkeen).
- Käytetään *kun informaation käsittelyjärjestys riippuu sen saapumisjärjestyksestä*.

  Kun käsittelyjärjestys on vapaa, niin taulukko on parempi.
- Luokittelu *päivityspaikan mukaan*:

rakenne	**lisäys**	**poisto**
pino	päälle	päältä
jono	perään	alusta
pakka	päälle ja pohjalle	päälle ja pohjalle

4.1 Osoittimet

- Taulukkoa monimutkaisempiin tietorakenteisiin tarvitaan *epäsuoraa osoittamista* (indirect addressing):
  - Tietoalkiota jonka sisältö osoittaa sen paikan, josta haluttu toinen alkio löytyy.
  - Piirroksissa *nuoli* lähtöpaikan sisältä kohdepaikan reunaan.
- Sellainen on *osoitin* (pointer):
  - "Sellaista ei olekaan" -arvo `NULL`, `NIL`,...
  - Muut arvot osoittavat paikkoihin.
  - Osoittimien perusoperaatiot:

    osoitetun sisällön haku eli "nuolen seuraaminen"

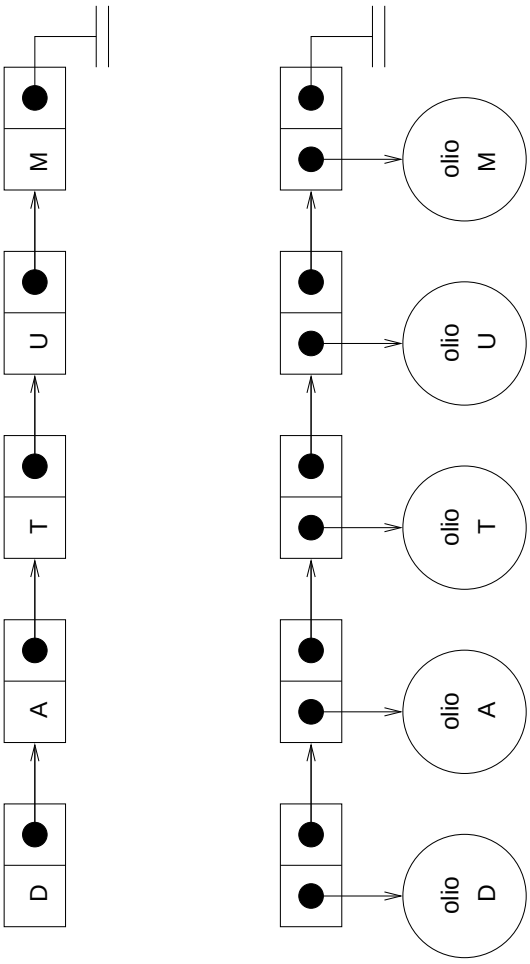
    paikan varaus osoitettavalle alkiolle `new`, `malloc`,...

    varatun paikan vapauttaminen `delete`, `dispose`, `free`,...

- Java-kielessä...
  - ei ole *näkyviä* osoittimia...
  - koska *jokaiseen olioon* kuljetaan piilotettua osoitinta pitkin
  - mutta perustyypit kuten `int` talletetaankin suoraan...
  - ja siksi nekin voidaan "korottaa" epäsuoriksi olioiksi kuten `Integer`.
- Peruslista koostuu lista-alkioista kenttinään
  - talletettu tietoalkio
  - osoitin seuraavaan lista-alkioon.
- Listarakenne on

  endogeeninen jos sen sisältämät tietoalkot on talletettu suoraan lista-alkioihin — Java-kielen perustyypit

  eksogeeninen jos lista-alkioihin on talletettu osoittimet tietoalkioihin — Java-kielen oliot.



- Listat (alkiotyyppiä τ) ovat *rekursiivisesti määritelty tietotyyppi:*
  - Tyhjä lista NULL on lista (alkiotyyppiä τ).
  - Jos
    - L on lista (alkiotyyppiä τ)
    - a on tietoalkio (tyyppiä τ)

    niin pari $\langle a, L \rangle$ on myös lista (alkiotyyppiä τ).
- Siis listoja voi käsitellä *listarekursiolla* joka on kalvojen 2.1 hajoita ja hallitse -periaatteen sovellus:
  - Tyhjän listan lukujen summa on 0.
  - Epätyhjän listan $\langle a, L \rangle$ lukujen summa on $a +$ loppulistan L lukujen summa.

  Kääntäen, listarekursiivisen koodin toiminta osoitetaan induktiolla.
- Periaate toimii muillakin osoitinrakenteilla *joissa ei ole kehiä*.

4.2 Pino

Listan variaatio *pino* (pushdown stack)

- Kuin lautaspino kaapissa:
  - Otetaan päällimmäinen kattaukseen.
  - Tiskattu pannaan päällimmäiseksi.

  Pinon sisälle ei kosketa.

- Tietoalkio kuin keltainen tarramuistilappu lautasen sisällä:
  - Päällimmäinen lappu voidaan lukea.
  - Alempi lappu voidaan lukea vasta kun sen päältä on otettu kaikki lautaset pois.

    Jos kaikki laput selattavissa, niin *avo*pino (stack).

  Lappuun ei kosketa kun sen lautanen on pinossa.

- Perusoperaatiot:

  Luonti makeEmptyStack( ) tyhjäksi.

  Tyhjyystesti isEmpty(s) kertoo onko pino s tyhjä vai ei.

  Lisäys push(s, a) lisää pinon s päälle lautasen jonka lappuun kirjoitetaan a.

  Luku top(s) palauttaa sen, mitä on kirjoitettu pinon s päällimmäisen lautasen muistilappuun.

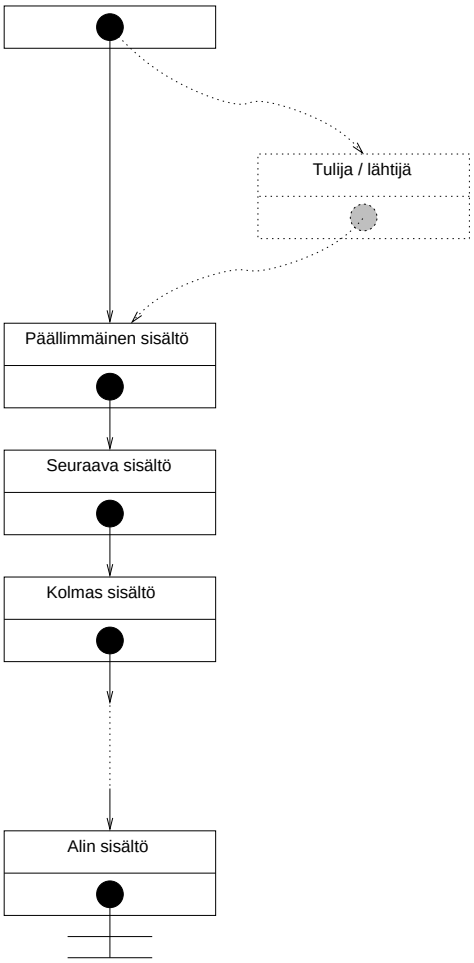
  Oletetaan $\neg$ isEmpty(s), muuten ajonaikainen virhe.

  Poisto pop(s) poistaa pinosta s sen päällimmäisen lautasen.

  Oletetaan $\neg$ isEmpty(s), muuten ajonaikainen virhe.

  Kaikki vievät *vakio*ajan $\mathcal{O}(1)$ jos ne toteutetaan *muokkaamalla pinoon vievää osoitinta*.

- Osoittimen invariantti: se osoittaa aina pinon päällimmäiseen lista-alkioon, jos sellainen on, muuten se on NULL.



- "*LIFO*- eli Last In, First Out -lista".

- Pinon tyypillinen käyttö:
  1. Puuhaillaan työtehtävän A parissa.
  2. Vastaan tulee työtehtävä B joka täytyy hoitaa *heti*.
  3. Kirjoitetaan tehtävästä A sellainen muistilappu, jonka pohjalta tehtävää A voidaan jatkaa siitä, mihin nyt jäätiin.
  4. Viedään lappu pinon päälle.
  5. Siirrytään työtehtävään B.
  6. Kun B on valmis, otetaan pinon päältä lappu, ja jatketaan tehtävää A sen merkintöjen mukaan.

- Esimerkki: aliohjelma*kutsu*pino.

4.3 Jono

Listan variaatio *jono* (queue):

- Kuin kassajono kaupassa:
  - Jonon ensimmäinen asiakas menee kassalle, kun se vapautuu.
  - Asiakas menee jonon viimeiseksi, kun ostokset on kerätty kärryyn.
- "*FIFO*- eli First In, First Out -lista".
- Tarvitaan *tunnussolmu* (header):

  Kenttinä osoittimet jonon
  - ensimmäiseen
  - viimeiseen

  lista-alkioon (jos sellainen on).

  Tunnussolmussa voi ylläpitää listasta tietoja, jotka haluaa tietää ajassa $\mathcal{O}(1)$: esimerkiksi *pituus*.

- Perusoperaatiot:

  Luonti makeEmptyQueue( ) tyhjäksi — varaa tunnussolmun ja alustaa sen molemmat osoittimet.

  Tyhjyystesti isEmpty(q) kertoo onko jono q tyhjä vai ei.

  Lisäys enqueue(q, a) lisää jonon q loppuun jonottajan a.

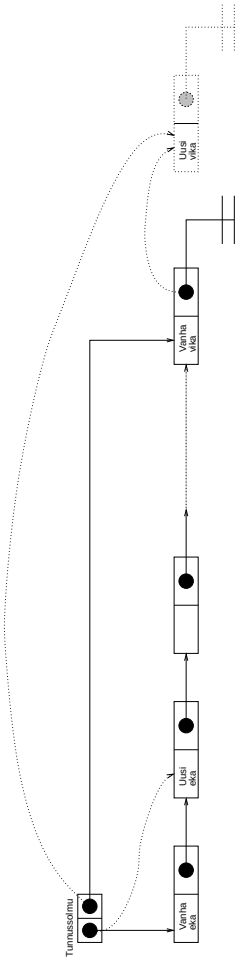
  Luku front(q) palauttaa jonon q ensimmäisen jonottajan (poistamatta sitä jonosta).

  Oletetaan $\neg$ isEmpty(q), muuten ajonaikainen virhe.

  Poisto dequeue(q) poistaa jonosta q sen ensimmäisen alkion.

  Oletetaan $\neg$ isEmpty(q), muuten ajonaikainen virhe.

  Kaikki vievät *vakio*ajan $\mathcal{O}(1)$ jos ne toteutetaan *muokkaamalla tunnussolmun osoitinkenttiä*.



- Jonon tyypillinen käyttö:
  1. Puuhaillaan työtehtävän A parissa.
  2. Vastaan tulee työtehtävä B jolla *ei ole kiire*.
  3. Kirjoitetaan tehtävästä B sellainen muistilappu, jonka pohjalta se voidaan aloittaa myöhemmin.
  4. Viedään lappu jonon perään.
  5. Jatketaan työtehtävää A.
  6. Kun A on valmis, otetaan jonosta ensimmäinen lappu, ja aloitetaan sen mukainen työtehtävä. Aikanaan vuoroon tulee myös B.
- Esimerkki: tuottaja-kuluttaja-pari (generate-and-test).

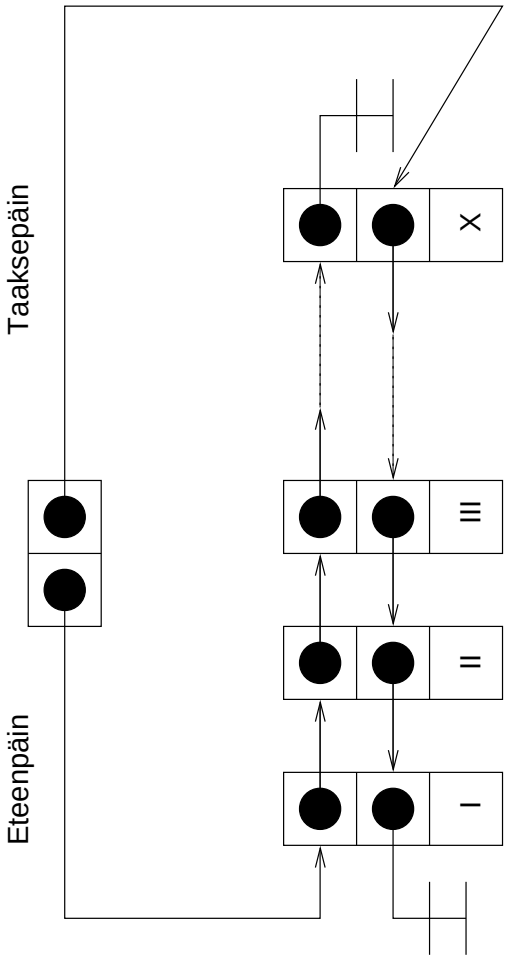
4.4 Pakka

Listan variaatio *pakka* (double-ended queue, dequeue):

- Kalvojen 4.3 jono jota voi
  - kasvattaa
  - lyhentää

  kummastakin päästä.
- Saadaan hitsaamalla yhteen 2 jonoa:
  - eteen-
  - taakse-

  päin.
- Siis jokaisessa listasolmussa 2 osoitinta:
  - seuraajaan
  - edeltäjään.



4.5 Polynomien listaesitys

- Tarkastellaan esimerkkinä *polynomien*

  $a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + a_{n-2} \cdot x^{n-2} + \cdots + a_0 \cdot x^0$

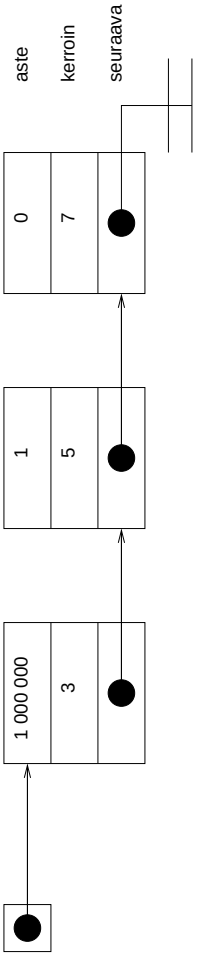
  esittämistä listoina.
- Silloin ei tarvitse esittää niitä monomeja $a_p \cdot x^p$ joilla $a_p = 0$.
- Silloin *harvan* (sparse) polynomin

  $$3 \cdot x^{1\,000\,000} + 5 \cdot x + 7$$

  esitys vie (luokkaa) 3 muistipaikkaa, ei 1 000 001.

  Käsittelykin on vastaavasti nopeampaa.
- Talletetaan lista-alkioon
  - aste k
  - kerroin a_k

  suurimmasta k alkaen.



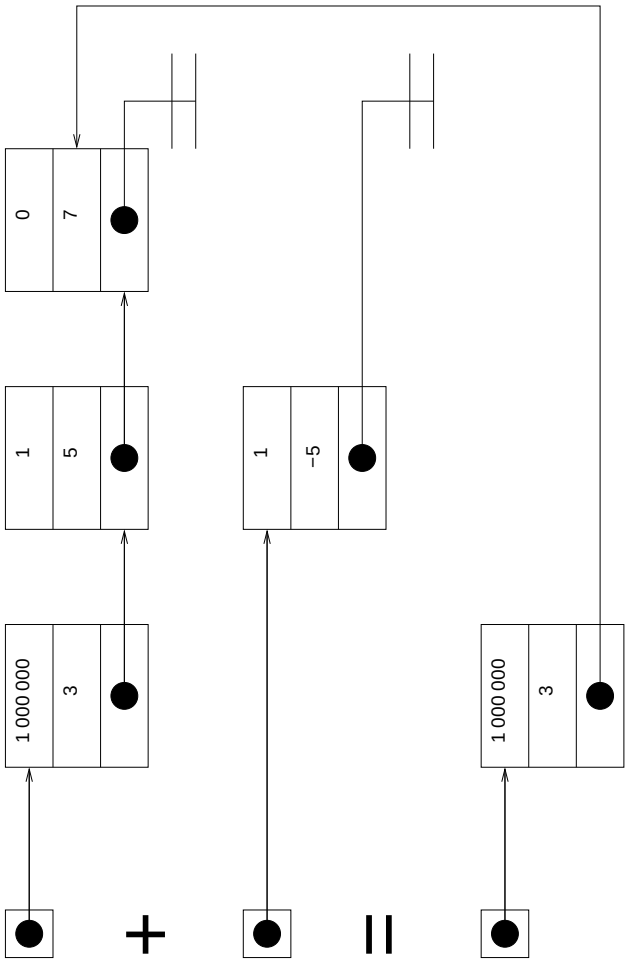
- Tarkastellaan polynomien yhteenlaskua

$$\left(\sum_{i\in\mathbb{N}} r_i \cdot x^i\right) + \left(\sum_{j\in\mathbb{N}} s_i \cdot x^j\right) = \left(\sum_{k\in\mathbb{N}} (r_k + s_k) \cdot x^k\right).$$

- Listaesityksessä:
  - Tyhjä lista on polynomi 0.
  - Puuttuvan monomin kerroin on 0.
  - Jos monomin kertoimeksi tulee 0, niin se jätetään pois.

- Esimerkkinä

$$\begin{array}{rl} & 3 \cdot x^{1\,000\,000} + 5 \cdot x + 7 \\ + & \qquad\qquad -5 \cdot x \\ = & 3 \cdot x^{1\,000\,000} \qquad\quad + 7. \end{array}$$



- Listarekursiolla:

function ps(r, s: **listaosoitin**): **listaosoitin**

```
 1: if r = NULL then
 2:     return s
 3: else if s = NULL then
 4:     return r
 5: else if r.aste > s.aste then
 6:     return new
        listaAlkio(r.aste,
                   r.kerroin,
                   ps(r.seuraava, s))
        eli uusi lista-alkio jonka vastaavat
        kentät alustetaan näin
 7: else if r.aste < s.aste then
 8:     return new
        listaAlkio(s.aste,
                   s.kerroin,
                   ps(r, s.seuraava))
 9: else if r.kerroin + s.kerroin = 0 then
10:     return ps(r.seuraava, s.seuraava)
11: else
12:     return new
        listaAlkio(r.aste,
                   r.kerroin + s.kerroin,
                   ps(r.seuraava, s.seuraava))
13: end if.
```


- Karvin luentomuistiinpanojen ja Javan merkintä

  osoitin_tietueeseen.kenttä_tietueessa = kentän sisältö

  seuraa *implisiittisesti* ensin osoitinnuolta tietueeseen.

- Cormenin et al kirjassa sama merkintä on

  kenttä_tietueessa[osoitin_tietueeseen] = kentän sisältö

  eli jokainen kenttä on oma *taulukko*, ja osoittimet indeksejä.

  Ideana on, että tietokoneen muisti on valtava taulukko, ja osoittimet indeksejä siitä varattuihin lohkoihin.

- *Rekursion muuntaminen silmukaksi* suoraan kalvojen 2.3 tapaan ei nyt onnistu:
  - Nyt rekursiokutsu riveillä 6, 8 ja 12 *ei ole viimeinen* toimitus ennen `return`-paluuta (kuten rivillä 10).
  - Eli ei olekaan *häntä- tai takarekursiivinen* kutsu (tail recursive call).
  - Takarekursiivinen kutsu voidaan muuntaa paluuksi silmukan alkuun.
  - Yleinen rekursiivinen kutsu tarvitsee kalvojen 4.2 pinoa.
- Tällä kertaa kutsut saadaan takarekursiivisiksi:
  - Tämä algoritmi luo tuloslistaansa syötelistojensa järjestyksessä.
  - Eli lisää tuloslistan loppuun.
  - Eli käytetäänkin tulos*jonoa* kalvoilta 4.3.
  - Esimerkki *kerääjän* (accumulator) käyttöönotosta.

- Tulos on
  q := makeEmptyQueue( );
  ps(r, s);
  Muunna tulosjono q tuloslistaksi unohtamalla tunnussolmun osoitin viimeiseen alkioon

jonka aliohjelmat ovat

procedure ps(r, s: **listaosoitin**)
  if $r =$ NULL **then**
    kopioi(s)
  else if $s =$ NULL **then**
    kopioi(r)
  else if $r.$aste $> s.$aste **then**
    enqueue($q, \langle r.\text{aste}, r.\text{kerroin}\rangle$);
    ps($r.$seuraava, s)
  else if $r.$aste $< s.$aste **then**
    enqueue($q, \langle s.\text{aste}, s.\text{kerroin}\rangle$);
    ps($r, s.$seuraava)
  else if $r.$kerroin $+ s.$kerroin $= 0$ **then**
    ps($r.$seuraava, $s.$seuraava)
  else
    enqueue($q, \langle r.\text{aste}, r.\text{kerroin} + s.\text{kerroin}\rangle$);
    ps($r.$seuraava, $s.$seuraava)
  end if.

procedure kopioi(t: **listaosoitin**)
  while $t \neq$ NULL **do**
    enqueue($q, \langle t.\text{aste}, t.\text{kerroin}\rangle$);
    $t := t.$seuraava
  end while.