- Proseduuri ps sisältää nyt vain takarekursiivisia kutsuja, joten sen rungosta saadaan silmukka:

```
while r ≠ NULL and s ≠ NULL do
  if r.aste > s.aste then
    enqueue(q, ⟨r.aste, r.kerroin⟩);
    r := r.seuraava
  else if r.aste < s.aste then
    enqueue(q, ⟨s.aste, s.kerroin⟩);
    s := s.seuraava
  else if r.kerroin + s.kerroin = 0 then
    r := r.seuraava;
    s := s.seuraava
  else
    enqueue(q, ⟨r.aste, r.kerroin + s.kerroin⟩);
    r := r.seuraava;
    s := s.seuraava
  end if
end while;
if r = NULL then
  kopioi(s)
else if s = NULL then
  kopioi(r)
end if
```

- Aikavaatimus on $\mathcal{O}(n)$ missä $n =$ syötelistojen yhteispituus = monomien kokonaismäärä syötepolynomeissa.

- Rekursion vaatima lisätila laitteiston ylläpitämässä aliohjelmakutsupinossa on
  - $\mathcal{O}(n)$ ennen
  - $\mathcal{O}(1)$ jälkeen

  rekursion poiston.

- On olemassa ohjelmointikielten
  - standardimäärittelyjä (kuten Scheme) jotka vaativat
  - toteutuksia (kuten `gcc`) jotka tekevät

  takarekursion poistoa.

- Jos siis puhutaan suorituksen tarvitsemasta laitepinosta, niin ensin pitää erottaa
  - takarekursiiviset
  - aidot rekursiiviset

  kutsut ja algoritmit toisistaan.

4.6 Rekursion poisto

- Tarkastellaan *rekursiivisen aliohjelman muuttamista iteratiiviseksi* esimerkkinä kalvojen 4.2 pinotietorakenteen käytöstä.
- Ongelmaa sivuttiin jo kalvoilla 4.5.
- Perusidea:
  - Tietokoneessa aliohjelmien kutsut talletetaan laitteiston ylläpitämään pinoon.
  - Korvataan tämä pino itse ylläpidetyllä.

Aluksi on funktio

```
function f(x1: α1, ..., xk: αk): β
  runko joka on sisäkkäinen if-lause;
  return (...).
```

Lopuksi on funktio

```
function f(x1: α1, ..., xk: αk): β
  Alusta ajopino σ syöteparametreina
  x1, ..., xk saaduilla arvoilla;
  while not isEmpty(σ) do
    Funktion f rungosta sellainen versio,
    joka käyttää ajopinon σ päällimmäistä
    alkiota kaikkeen kirjanpitoon ja jossa
    rekursiiviset kutsut on korvattu
    ajopinon σ suoralla käsittelyllä
  end while;
  return vastausmuuttujan y lopullinen
  arvo.
```

Välissä muunnamme funktion f runkoa vaihe vaiheelta samaan tapaan kuin ohjelmointikielen kääntäjä voisi tehdä.

1. Otetaan käyttöön *kirjanmerkit:*
   - Rungon alkuun pannaan kirjanmerkki

     0: runko
   - Muita kirjanmerkkejä $1, 2, 3, \ldots$ otetaan käyttöön sitä mukaa kun tarvitaan.

2. Ainoa sallittu rekursiokutsun muoto on *yksin sijoituslauseessa*:
   u := $f(\ldots)$

   Jos kutsu on sijoituslauseen sisällä

   v := $\ldots f(\ldots)\ldots$

   niin
   - luodaan uusi muuttujannimi z_j: β
   - korvataan sisäkutsu muotoon

     z_j := $f(\ldots)$;
     v := $\ldots z_j \ldots$

   Kutsu nousee samaan tapaan ulos myös **if**-ehdosta ja **return**-paluuarvosta.

3. Nyt tiedetään ne kentät, jotka pinoon σ vietävässä *aktivaatiotietueessa* (activation record) a täytyy olla:
   - $a.x_i$: α_i jokaiselle parametrille x_i: α_i
   - $a.z_j$: β jokaiselle vaiheessa 2 luodulle uudelle muuttujannimelle z_j: β
   - $a.\ell$: $\mathbb{N}$.

   Koko pino σ tallettaa riittävät tiedot jokaisesta keskeneräisestä rekursiokutsusta, jotta sitä voidaan myöhemmin jatkaa.

   Yksi aktivaatiotietue a tallettaa yhden kutsun tiedot:
   - kutsun sisäisten muuttujien $x_1, x_2, x_3, \ldots, x_k$ ja $z_1, z_2, z_3, \ldots$ arvot joilla jatketaan
   - kirjanmerkin ℓ josta runkoa jatketaan.

   Päällimmäinen aktivaatiotietue top(σ) kertoo käynnissä olevan kutsun tiedot.

4. Korvataan jokainen parametrin x_i maininta rungossa sitä vastaavan kentän top(σ)$.x_i$ maininnalla.

   Samoin muuttujille z_j.

5. Korvataan jokainen rekursiokutsu

   w := $f(e_1, \ldots, e_k)$

   koodinpätkällä

   top(σ)$.\ell$:=δ;
   b := **new** aktivaatiotietue;
   $b.x_1$:= e_1;
   ⋮
   $b.x_k$:= e_k;
   $b.\ell$:= 0;
   push(σ, b);
   δ: w := y

   missä
   - δ on uusi kirjanmerkki
   - suorituksen on tarkoitus hypätä takaisin alkuun juuri sitä ennen
   - ja palata myöhemmin takaisin juuri siihen.

6. Suorituksen pitää voida palata vaiheessa 5 luotuun kirjanmerkkiin δ.
   - Tehdään lauseelle **switch (**top(σ)$.\ell$**)** haara **case** δ**:** haara$_\delta$
   - Ohjelmanpätkä haara$_\delta$ saadaan kopioimalla kontrollireittejä kohdasta δ alkaen kunnes reitti päättyy...

     toiseen kirjanmerkkiin η**:** Haara loppuu siihen.

     Suoritus palaa aikanaan kirjanmerkkiä η vastaavaan haaraan haara$_\eta$.

     vastaukseen return e**:** Haaran loppuun laitetaan

     y := e;
     pop(σ)

     joka palauttaa kontrollin vastaavaan vaiheessa 5 luotuun kirjanmerkkiin.

   Suoraviivaista nyt kun runko oletettiin yksinkertaiseksi, yleisesti hankalampaa.

7. Silmukan sisältö on nyt valmis:

switch (top(σ).ℓ)
  case 0: haara$_0$;
  case 1: haara$_1$;
  case 2: haara$_2$;
  case 3: haara$_3$;
   ⋮
end switch

kaikille vaiheessa 5 luoduille kirjanmerkeille $1, 2, 3, \ldots$

8. Silmukan alustus mukailee vaihetta 5:

σ := makeEmptyStack( );
b := **new** aktivaatiotietue;
$b.\ell$:= 0;
$b.x_1$:= x_1;
⋮
$b.x_k$:= x_k;
push(σ, b);

Eli alustetaan pinon pohja alkuperäisellä rekursiokutsulla.

- Kalvojen 4.5 takarekursiivinen kutsu

  return $f(e_1, \ldots, e_k)$

  voidaan lisäksi optimoida siten, että

  – pinoon ei viedä uutta aktivaatiotietuetta

  – vaan nykyinen käytetäänkin uudelleen

  – koska sehän on nyt turha.

  Silloin kutsu ei lisää pinotilan tarvetta.

- Vaiheessa 5 se jätetäänkin ennalleen.

- Vaiheeseen 6 lisätään vastaava vaihtoehto

  vastaukseen return $f(e_1, \ldots, e_k)$**:** Haaran loppuun laitetaankin vaihetta 5 mukaillen

  top(σ).ℓ := 0;
  top(σ).x_1 := e_1;
  ⋮
  top(σ).x_k := e_k

  joka kirjoittaakin nykyiseen aktivaatiotietueeseen.

- Esimerkkinä kalvojen 4.5 listarekursiivinen algoritmi.

- Vaiheen 2 jälkeen:

if r = NULL **then**
  return s
else if s = NULL **then**
  return r
else if r. aste > s. aste **then**
  z_1 := ps(r. seuraava, s);
  return new listaAlkio(r. aste,
                r. kerroin,
                z_1)
else if r. aste < s. aste **then**
  z_2 := ps(r, s. seuraava);
  return new listaAlkio(s. aste,
                s. kerroin,
                z_2)
else if r. kerroin + s. kerroin = 0 **then**
  return ps(r. seuraava, s. seuraava)
else
  z_3 := ps(r. seuraava, s. seuraava);
  return new
  listaAlkio(r. aste,
       r. kerroin + s. kerroin,
       z_3)
end if.

- Vaiheen 4 jälkeen:

if top(σ).r = NULL **then**
  return top(σ).s
else if top(σ).s = NULL **then**
  return top(σ).r
else if top(σ).r. aste > top(σ).s. aste **then**
  top(σ).z_1 := ps(top(σ).r. seuraava, top(σ).s);
  return new listaAlkio(top(σ).r. aste,
                top(σ).r. kerroin,
                top(σ).z_1)
else if top(σ).r. aste < top(σ).s. aste **then**
  top(σ).z_2 := ps(top(σ).r, top(σ).s. seuraava);
  return new listaAlkio(top(σ).s. aste,
                top(σ).s. kerroin,
                top(σ).z_2)
else if top(σ).r. kerroin + top(σ).s. kerroin = 0
then
  return ps(top(σ).r. seuraava, top(σ).s. seuraava)
else
  top(σ).z_3 := ps(top(σ).r. seuraava,
            top(σ).s. seuraava);
  return new
  listaAlkio(top(σ).r. aste,
       top(σ).r. kerroin + top(σ).s. kerroin,
       top(σ).z_3)
end if.

- Vaiheen 5 jälkeen:

```
if top(σ).r = NULL then
   return top(σ).s
else if top(σ).s = NULL then
   return top(σ).r
else if top(σ).r.aste > top(σ).s.aste then
   top(σ).ℓ := 1;
   b := new aktie(top(σ).r.seuraava, top(σ).s);
   push(σ, b);
   1: top(σ).z1 := y;
   return new listaAlkio(top(σ).r.aste,
                         top(σ).r.kerroin,
                         top(σ).z1)
else if top(σ).r.aste < top(σ).s.aste then
   top(σ).ℓ := 2;
   b := new aktie(top(σ).r, top(σ).s.seuraava);
   push(σ, b);
   2: top(σ).z2 := y;
   return new listaAlkio(top(σ).s.aste,
                         top(σ).s.kerroin,
                         top(σ).z2)
else if top(σ).r.kerroin + top(σ).s.kerroin = 0
then
   return ps(top(σ).r.seuraava, top(σ).s.seuraava)
else
   top(σ).ℓ := 3;
   b := new aktie(top(σ).r seuraava,
                  top(σ).s.seuraava);
   push(σ, b);
   3: top(σ).z3 := y;
   return new
   listaAlkio(top(σ).r.aste,
          top(σ).r.kerroin + top(σ).s.kerroin,
          top(σ).z3)
end if.
```


- Vaiheen 6 sivuhaarat:

```
case 1:
   top(σ).z1 := y;
   y:= new listaAlkio(top(σ).r.aste,
                      top(σ).r.kerroin,
                      top(σ).z1);
   pop(σ)

case 2:
   top(σ).z2 := y;
   y := new listaAlkio(top(σ).s.aste,
                       top(σ).s.kerroin,
                       top(σ).z2);
   pop(σ)

case 3:
   top(σ).z3 := y;
   y := new listaAlkio(top(σ).r.aste,
                       top(σ).r.kerroin
                       + top(σ).s.kerroin,
                       top(σ).z3);
   pop(σ)
```


- Vaiheen 6 päähaara:

```
case 0:
   if top(σ).r = NULL then
      y := top(σ).s;
      pop(σ)
   else if top(σ).s = NULL then
      y := top(σ).r;
      pop(σ)
   else if top(σ).r.aste > top(σ).s.aste then
      top(σ).ℓ := 1;
      b := new aktie(top(σ).r.seuraava, top(σ).s);
      push(σ, b)
   else if top(σ).r.aste < top(σ).s.aste then
      top(σ).ℓ := 2;
      b := new aktie(top(σ).r, top(σ).s.seuraava);
      push(σ, b)
   else if top(σ).r.kerroin + top(σ).s.kerroin = 0
   then
      top(σ).r := top(σ).r.seuraava;
      top(σ).s := top(σ).s.seuraava;
      top(σ).ℓ := 0
   else
      top(σ).ℓ := 3;
      b := new aktie(top(σ).r seuraava,
                     top(σ).s.seuraava);
      push(σ, b)
   end if.
```


- Lopputulos vaiheiden 7 ja 8 jälkeen:

```
function ps(r, s: listaosoitin): listaosoitin
   σ := makeEmptyStack( );
   push(σ, new aktie(r, s));
   while not isEmpty(σ) do
      switch (top(σ).ℓ)
         case 0: ...;
         case 1: ...;
         case 2: ...;
         case 3: ...
      end switch
   end while;
   return y.
```

4.7 Iteraattorit

- *Iteraattori* peittää osoittimien käsittelyn abstraktin rajapinnan taakse.
- Peittää allaolevan tietorakenteen yksityiskohdat kutsuvalta algoritmilta, joka vain käy läpi sen sisältämiä tietoja järjestyksessä.
- Sallii *geneeriset algoritmit*
  - jotka toimivat kaikilla sellaisilla tietorakenteilla
  - joille voidaan luoda sopivat iteraattorit.
- Tärkeä käsite tietorakennekirjastoissa.

  Periytyminen hyödyllistä:
  - listaiteraattori on 1-suuntainen iteraattori
  - taulukkoiteraattori on 2-suuntainen iteraattori
  - ...

- Perusoperaatiot iteraattorille, joka *lukee yhteen suntaan* ketjutettua listaa:

  Alustus *osoittamaan* annetun listan ensimmäiseen lista-alkioon.

  Talletetun tietoalkion lukeminen osoitetusta lista-alkiosta.

  Seuraavaan lista-alkioon siirtyminen pitkin listan osoitinketjua.

  Lopputesti joka kertoo onko iteraattori ohittanut listansa viimeisen alkion. Silloin
  - vastaavaa tieto-
  - seuraavaa lista-

  alkiota ei ole.
- Muut operaatiot riippuvat

  allaolevasta tietorakenteesta: 2-suuntaisella ketjutuksella (kuten kalvojen 4.4 pakoissa) paluu **edelliseen**.

  käyttötarkoituksesta: listan *sisäosan* muuttaminen iteraattorin kohdalta.

- Kalvojen 2.4.1 peräkkäishaku iteraattorilla:

  function
  phait(I: **iteraattori**, b: $\mathbb{N}$) : **boolean**
    while (not `atEnd`(I)**) and** `retrieve`(I) $< b$ **do**
      `next`(I)
    end while;
    return (not `atEnd`(I)**) and** `retrieve`(I) $= b$**.**

- Parametrina taulukon A sijaan (perus)iteraattori I josta alkaen hakua suoritetaan.
- Sopii erilaisille järjestetyille A:

  taulukolle kutsulla
  phait(**new** `arrayIterator`(A), b)

  listalle kutsulla
  phait(**new** `listIterator`(A), b)

  ...

- Operoinnin
  - epäsuorasti iteraattoreilla
  - suoraan osoittimilla

  pitää olla yhtä tehokasta $\mathcal{O}$-tarkkuudella.

  Valmiskirjastoissa luvataankin usein aika- ja tilavaatimukset $\mathcal{O}$-tarkkuudella.
- Jos sallitaan uuden alkion lisääminen iteraattorin I osoittaman kohdan

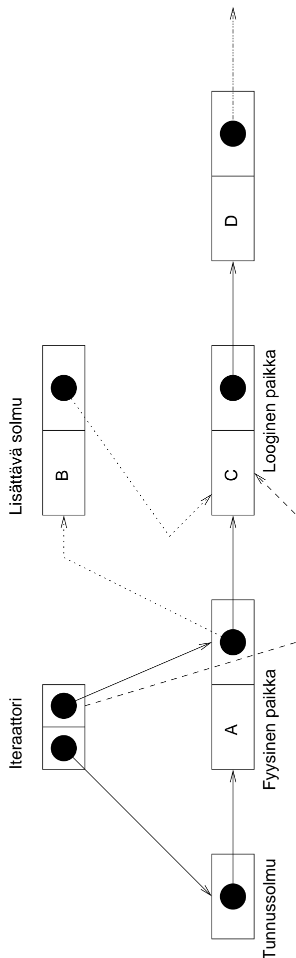
  perään niin mitä tehdään kun `atEnd`(I)?

  eteen niin mitä tehdään kun ollaan ensimmäisessä alkiossa?
- Ratkaisu: I osoittaa

  loogisesti nykyiseen, mutta

  fyysisesti edelliseen lista-alkioon, missä

  tunnussolmu on ensimmäisen edellinen.



- Kalvojen 3.4 lisäyslajittelu:

```
J := alkukohta;
while not atEnd(J) do
  I := alkukohta;
  while retrieve(I) < retrieve(J) do
    next(I)
  end while;
  if atSame(I, J) then
    next(J)
  else
    insert(I, retrieve(J));
    delete(J)
  end if
end while
```

  - Uusi operaatio `atSame`(I, J) testaa osoittavatko iteraattorit I ja J samaan vai eri alkioon.
  - Lisäysoperaatio `insert`(I, x) lisää alkion x iteraattorin I osoittaman kohdan *eteen*.
  - Poisto-operaatio `delete`(J) iteraattorin J osoittaman alkion y ja
  - jättää iteraattorin J osoittamaan poistuneen alkion y seuraajaan.

- Listan sisälle kohdistuvien lisäysten ja poistojen *synkronointi* voi olla vaikeaa:

  1. Poista alkio iteraattorin J kohdalta.
  2. Lisää alkio iteraattorin I kohdalle.

  Entäs jos aluksi kohdat J ja I olivatkin *samat*?

- Yleisempi *osoitinaliasongelma* (pointer aliasing):
  - Jos samaan muuttuvaan tietoon on useita eri viitteitä...
  - niin eri viittaajat eivät voi luottaa...
  - siihen että tieto pysyy samana kahden käyttökerran välissä.

- Ei-paikalliset riippuvuudet ohjelmakoodissa:
  - toisaalta nopeuttavat tiedonvälitystä...
  - toisaalta virheiden lähde!

5 Järjestäminen

- Yksi järjestämisalgoritmi nähtiin jo kalvoilla 3.4.

  Se vei $\mathcal{O}(n^2)$ askelta, missä n on lajiteltavien alkioiden lukumäärä.

- Esitellään 3 muuta järjestämisalgoritmia:

  1. *lomitus*järjestäminen
  2. *pika*järjestäminen
  3. *keko*järjestäminen

  Ne vievät vain

  $$\mathcal{O}(n \cdot \log(n))$$

  askelta (eräin lisäoletuksin).

- Toisaalta itse järjestämisongelma vie

  $$\Omega(n \cdot \log(n))$$

  askelta (eräin lisäoletuksin).

5.1 Lomitusjärjestäminen

- *Lomitusjärjestäminen* (merge sort) saadaan kalvojen 2.1 hajoita ja hallitse -periaatteella:
  - On helppoa järjestää $n \leq 1$ alkiota.
  - Jaetaan $n \geq 2$ alkiota *2 yhtä suureen osaan* A ja B jotka lajitellaan rekursiivisesti.
  - Lajitellut osat A' ja B' lomitetaan toisiinsa, ja saadaan kokonaisratkaisu.
- Hyvä menetelmä, jos *aineisto tulee listana jonka osoittimia saa muuttaa*.

function mergesort(L: **lista**): **lista**
  $\langle A, B \rangle$:= split(L);
  if B = NULL **then**
    return A
  else
    return merge(mergesort(A), mergesort(B))
  end if.

5.1.1 Aineiston jakaminen

- Aineiston jakaminen 2 (melkein) yhtä suureen osaan käy
  - ”sulle-mulle-sulle-mulle-. . .”
  - ”Jako kahteen!”

  -periaatteella.
- Resursseina tarvitaan:

  Aikaa $\mathcal{O}(|L|)$: Käydään syöteaineisto L kerran läpi, missä $|L|$ on sen pituus.

  Aputilaa $\mathcal{O}(1)$: Jos
  - saa muokata syötteen osoitinkenttiä
  - (helppo) rekursio korvataan toistolla kalvojen 4.6 tapaan.

Invariantti: Alkuperäinen lista L on jaettu osiin A, B, M siten, että osassa A on alkioita

- joko yhtä monta
- tai 1 enemmän

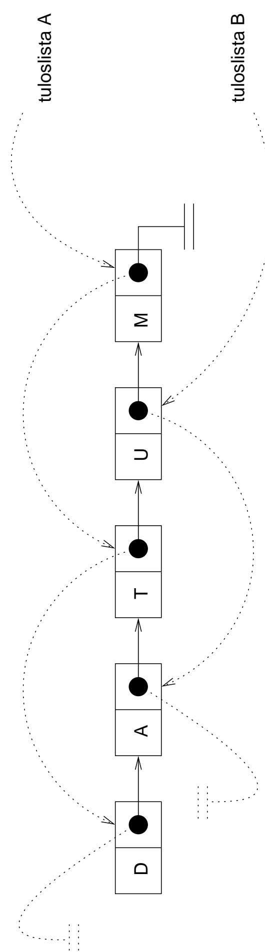
kuin osassa B.

Aluksi listat A ja B ovat tyhjiä.

Konvergentti: Jakamaton osa M lyhentyy (aikanaan tyhjäksi).

Nopeaa koska saamme itse valita mitä osoittimia muutamme — siis ensimmäisiä, koska ne ovat heti tarjolla.

function split(A, B, M: **lista**): **listapari**
  if M on tyhjä **then**
    return $\langle A, B \rangle$
  else
    Siirrä listasta M yksi lista-alkio listaan B;
    split(B, A, M)
  end if.



5.1.2 Aineiston yhdistäminen

- Kaksi (samoin) järjestettyä listaa A' ja B' voidaan *lomittaa* (merge) yhdeksi (samoin) järjestetyksi listaksi:
  - Jos listan A' ensimmäinen alkio $E_{A'}$ on pienempi kuin listan B', niin vie $E_{A'}$ tuloslistan loppuun.
  - Ja päinvastoin.
- Resursseina tarvitaan

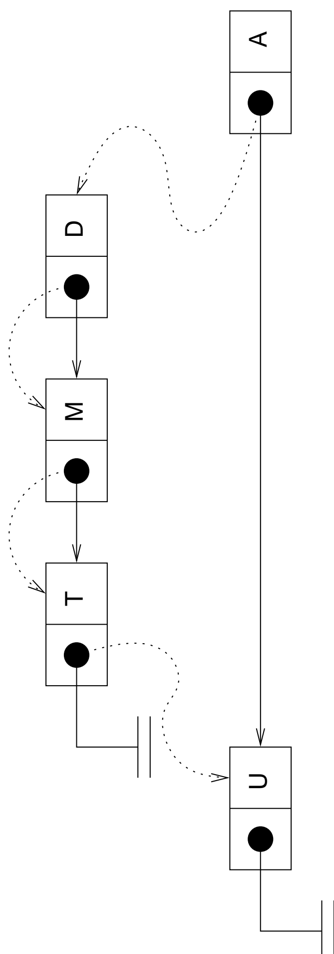
  aikaa $\mathcal{O}(|A'| + |B'|)$ — yksi läpikäynti.

  aputilaa $\mathcal{O}(1)$ samoin oletuksin kuin kalvoilla 5.1.1.

- Toistorakenteella osoittimia käsitellen:

function merge(A', B': **lista**): **lista**
  Q **:=** makeEmptyQueue( )**;**
  while $A' \neq$ NULL **and** $B' \neq$ NULL **do**
    if $E_{A'} < E_{B'}$ **then**
      Siirrä listan A' ensimmäinen
      lista-alkio tulosjonon Q perään
    else
      Siirrä listan B' ensimmäinen
      lista-alkio tulosjonon Q perään
    end if
  end while;
  if lista A' on tyhjä **then**
    Liitä B' jonon Q perään järjestystä
    muuttamatta
  else if lista B' on tyhjä **then**
    Liitä A' jonon Q perään järjestystä
    muuttamatta
  end if;
  return Q listana**.**

- Vertaa kalvojen 4.5 algoritmiin ja sen kehitysvaiheisiin.



5.1.3 Lomitusjärjestämisen ajantarve

- Analysoidaan sitten koko algoritmin aikavaatimusta:

function mergetime(n: $|L|$)
  $\mathcal{O}(n)$**;**
  if $n < 2$ **then**
    $\mathcal{O}(1)$
  else
    mergetime($\lceil n/2 \rceil$) +
    mergetime($\lfloor n/2 \rfloor$) + $\mathcal{O}(n)$
  end if.

- Rajoitutaan syötepituuksiin muotoa $n = 2^k$ (kuten kalvoilla 3.2).
- Saadaan palautuskaava

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

- Ratkaistaan se binääripuita tarkastelemalla.

5.1.3.1 Binääripuun idea

- *Binääripuu* (binary tree) on tietojenkäsittelyssä *hyvin* yleinen

  ajattelun apuneuvo kuten nyt

  konkreettinen tietorakenne kuten myöhemmin.

- Induktiivinen määritelmä: Binääripuussa on

  juuri(solmu) (root (node/vertex)) jolla on

  lapsia yhteensä

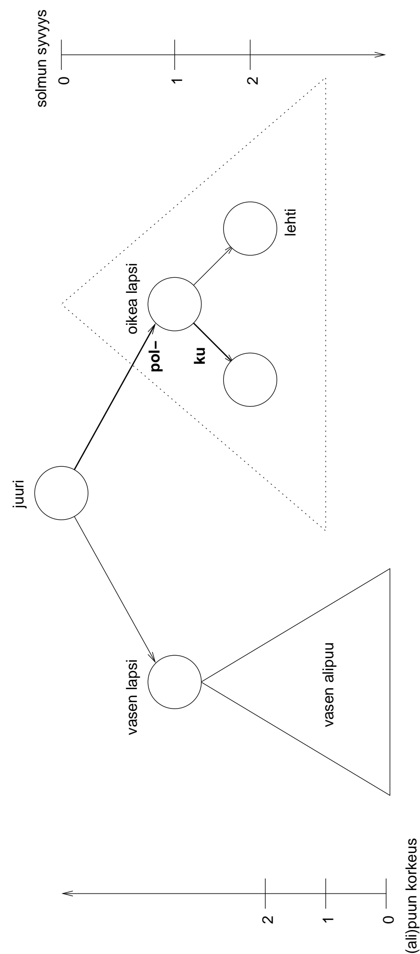
  – joko 0, jolloin se on *lehti* (leaf)

  – tai 2, jolloin se on *sisäsolmu* (interior node). Nämä lapset ovat

    vasen ja

    oikea

    ja kumpikin niistä on oman *alipuunsa* (subtree) juuri.



Nuolta isästä lapseen kutsutaan *kaareksi* (edge/arc).

Konkreettisessa tietorakenteessa nuoli on kalvojen 4.1 osoitin (ja solmu tietue).

Binääripuu on siis *haarautuva* tietorakenne (toisin kuin kalvojen 4 listat).

Polku (path) on (mahdollisesti tyhjä) jono peräkkäisiä nuolia.

Konkreettisesti siis osoitinketju.

Polun pituus on siinä olevien kaarten lukumäärä.

Konkreettisesti siis osoitinketjun pituus.

Solmun syvyys on (ainoan) juuresta siihen vievän polun pituus.

Konkreettisesti siis se määrä osoittimia jotka täytyy kulkea kun halutaan juuresta tähän solmuun.

Siis tähän solmuun pääsemiseen kuluvat askeleet.

(Ali)puun korkeus on siinä olevan solmun suurin syvyys.

Konkreettisesti siis pisin määrä osoittimia mikä täytyy koskaan kulkea.

Siis pahin koskaan tarvittava askelmäärä.

Täydessä (full) (ali)puussa kaikki lehdet ovat yhtä syvällä.

Siis puu on piirroksena "täysi kolmio".

- Määrittelimme itse asiassa erikoistapauksen "*epätyhjä aito* binääripuu":

  Tyhjä puu on solmuton — "ei mitään".

  Konkreettisesti se on NULL-osoitin.

  Silloin "lehti on solmu jonka molemmat alipuut ovat tyhjiä" jne.

  Muussa kuin aidossa binääripuussa sallitaan myös 1-lapsiset solmut.

- Aito täysi binääripuu on **täydellinen** (complete).

- Binääripuut ovat erikoistapauksia *yleisistä* puista:
  - solmulla voi olla lapsia myös > 2
  - lapsilla ei välttämättä keskinäistä järjestystä.
- Yleiset puut ovat erikoistapaus *verkoista* (graph) joissa
  - on solmuja
  - on solmujen välisiä kaaria
  - solmuun voi tulla monta eri kaarta
  - polut voivat muodostaa kehiä.
- Kaikkiin nähin palaamme kurssilla myöhemmin.

5.1.3.2 Rekursiokutsupuu

- Ajatellaan että rekursiivinen funktio *piirtää puuta* suorituksestaan:

  Aluksi piirretään tätä kutsua kuvaava juurisolmu.

  Juurisolmuun kerätään tarpeelliset tiedot tämän kutsun niistä askelista, jotka tehdään *ilman rekursiota*.

  Rekursiokutsu piirtää juurisolmulle aina seuraavan alipuun samalla periaatteella.
- Tällaista puuta kutsutaan *rekursiokutsupuuksi* (recursion call tree).
- Valitsemalla tarpeelliset tiedot sopivasti saadaan kerättyä tietoa *koko* rekursiivisesta laskennasta.

- Analysoidaan lomituslajittelun ajantarvetta rekursiokutsupuilla:
  - Tarpeellinen tieto on
    - kalvojen 5.1.1 jakamisessa
    - kalvojen 5.1.2 yhdistämisessä

    käytetty `aika`, joka on
    $$\mathcal{O}(|L|). \quad (3)$$
  - Eli jokaiselle kutsulle oma kalvojen 3 väijyntälaskuri.
- Nyt rekursiokutsupuu on

  aito koska jos rekursiiviseen haaraan mennään, niin tehdään 2 rekursiokutsua

  täysi koska aina
  $$|L| = 2^k$$
  eli **täydellinen** kalvojen 5.1.3.1 binääripuu.