TIEA211 Algoritmit 2

Antti Valmari

Jyväskylän yliopisto

1 Johdanto

1.1 Tästä tekstistä

Tavoitteita

- harjoitella ohjelmointia
- oppia saamaan ohjelmia aina tai melkein aina oikein toimiviksi
- oppia menetelmiä ja yksityiskohtia, jotka ovat hyödyllisiä ohjelmoinnissa laajalti
- saada yleiskäsitys tärkeimmistä algoritmeista ja tietorakenteista
- kehittyä ohjelmoinnin ja ohjelmistosuunnittelun ajattelutavassa

Esimerkki tosielämässä tehdyistä virheistä: opintojaksojen kokonaisuuden tietomalli

- esimerkki koskee erästä jo kuopattua järjestelmää
- vaadittiin, että jos opintojakso pidetään, niin sen pakolliset esitiedotkin pidetään
- järjestelmä tunsi vain yhden lukuvuoden kerrallaan
- kevään opintojakso A oli syksyn opintojakson B pakollinen esitieto

⇒ kun B pidettiin viimeisen kerran, järjestelmä vaati A:n seuraavalle keväälle

- sen sijaan järjestelmä ei huomannut, jos A:lta puuttui toteutus edellisenä keväänä
- opettajien ja oppiaineiden täytyi koordinoida vahvistamispäiviä keskenään

⇒ järkevältä kuulostava tavoite kääntyi järjestelmän toteutuksessa itseään vastaan!

Sama esimerkki, toinen vika

- aikataulua ei esitetty viikkorytminä vaan luetteloina erilaisia oppimistapahtumia
- tiedot siirrettiin seuraavalle vuodelle kopioimalla ja korjailemalla
- jos edellisvuoden luento kopioitui seuraavan vuoden vappuun, niin se piti poistaa
- poistetun luennon kohdalta kopioitui sitä seuraavaan vuoteen aukko

⇒ valtava määrä työtunteja tarkastamiseen ja puuttuvien oppimistapahtum. lisäämiseen!

Parempi tietomalli

- tavallisen opintojakson esitystavan lähtökohta on viikkoaikataulu
  - viikkojen määrä
  - viikottaisten tapahtumien luettelo: ti 10–12 luento sali 3, to 14–16 luento sali 3, ma 12–14 harjoitus sali 231, ...
- lisäksi poikkeusten luettelo, esim. yksittäiset salin vaihdot
- järjestelmä kopioi seuraavalle vuodelle
  - vapun yms. aiheuttamia aukkoja ei poisteta viikkoaikataulusta vaan vain näytettävästä versiosta
  - vielä helpompi ratkaisu: vapulle yms. osuvia tapahtumia ei poisteta
- esitietojen tarkastamista varten muistetaan edellisten toteutusten päättymispäivät
- miten oppia ja opettaa tällaista ajattelutapaa?

Lämmittelykysymyksiä

```
1   int arvosana = 0;
2   while( arvosana < 5 && pisteet >= pisteraja[ arvosana ] ){
3     ++arvosana;
4   }
```

1.2 Arvosanan laskemisen testaaminen tietokoneella

Miten voi testata arvosanan laskevia ohjelmia?

- täysin varmaa keinoa ei ole olemassa
- tässä luvussa esitellään helppo, melko tehokas keino

Arvosanan 1 raja on `rajat[0]`, arvosanan 2 raja on `rajat[1]` jne. arvosanaan 5 saakka

- arvosana on väärä, jos ja vain jos pisteet eivät riitä siihen tai ne riittäisivät korkeampaan arvosanaan
- ei ihan toimiva testi

```
int arvosana = testattava( rajat, pist );
if( pist < rajat[ arvosana-1 ] || pist >= rajat[ arvosana ] )
  { ... ilmoita virheestä }
```

- on helppo kokeilla kaikki pistemäärät kohtuulliseen rajaan saakka

```
for( int pist = 0; pist <= 30; ++pist ){
  ... edellä ollut koodinpätkä
}
```

Esim. Javalla tällainen testaaja kaatuu poikkeukseen ennemmin tai myöhemmin!

- jos testattava palauttaa 0, niin `rajat[ arvosana-1 ]` on laiton indeksointi
- jos testattava palauttaa 5, niin `rajat[ arvosana ]` on laiton indeksointi

Indeksoinnin tarkastava `rajat[...]` voidaan C++:lla toteuttaa kuten riveillä 2, . . . , 13

```
1   #include <iostream>
2   class taulukko{
3     static int A[5];
4   public:
5     int size(){ return 5; }
6     int operator[]( int i ){
7       if( 0 <= i && i < 5 ){ return A[i]; }
8       std::cout << i << " laiton indeksi\n"; return -1;
9     }
10  };
11  int taulukko::A[5] = {14, 17, 20, 23, 26};
```

Korjattu vastauksen tarkastus on riveillä 22, . . . , 24

- rivi 22 ei lisää kiinni jäävien virheiden määrää, mutta selkeyttää ilmoituksia

⇒ rivi 22 ei ole välttämätön

```
12  int testattava( taulukko & rajat, int pist ){
13    for( int i=0; i < rajat.size()-1; i++ )
14      if( pist >= rajat[i] && pist < rajat[i+1] ) return i+1;
15    return 0;
16  }

17  int main(){
18    taulukko rajat;
19    for( int pist = 0; pist <= 30; ++pist ){
20      int arvosana = testattava( pist );
21      if(
22        arvosana < 0 || arvosana > 5 ||
23        ( arvosana > 0 && pist < rajat[ arvosana-1 ] ) ||
24        ( arvosana < 5 && pist >= rajat[ arvosana ] )
25      ){ std::cout << pist << " " << arvosana << "\n"; }
26    }
27  }
```

Testiympäristömme ei saa kiinni esimerkiksi alla olevaa virheellistä ohjelmanpätkää

```
if( pist < rajat[0] ){ return 0; }
if( pist >= rajat[4] ){ return 5; }
return ( pist - rajat[0] ) / ( rajat[1] - rajat[0] ) + 1;
```

Mahdollisimman tehokkaan pisterajojen joukon suunnitteleminen olisi vaikeaa

- vaatisi samankaltaista ajattelua kuin virheiden löytäminen miettimällä
- edellä oleva testaaja oli helppo keksiä
- se hyödyntää tietokoneen kykyä laskea paljon lyhyessä ajassa

Luotettavia ohjelmia saa aikaan vain testaamisen ja miettimisen yhdistelmällä

1.3 Puolitushaku

Erittäin nopea keino löytää avain järjestetystä taulukosta

Toista kunnes avain löytyi tai etsintäalue kapeni nollan kokoiseksi

- vertaa avainta keskimmäiseen alkioon
- $=$: löytyi
- $<$: jatka etsintäalueen alkupuolikkaalla
- $>$: jatka etsintäalueen loppuupuolikkaalla

20 kierrosta riittää etsimään miljoonan kokoisesta taulukosta

Puolitushaku on vaikea toteuttaa niin, että se toimii luotettavasti

- Bentley (1986): $\leq 10\,\%$ ohjelmistoammattilaisista onnistui
- Knuth (1973): puolitushaku julkaistiin 1946, virheetön puolitushaku vasta 1962
- Pattis (1988): 20:stä oppikirjasta 15:ssa puolitushaku oli väärin
- Bloch (2006): "Extra, Extra - Read All About It:
  Nearly All Binary Searches and Mergesorts are Broken"
  - lukualueen ylivuoto
  - vain valtavan suurilla taulukoilla

Löydä kaksi virhettä (Blochin kertoman ylivuotovirheen lisäksi)

```
1   int p_haku( const taulukko & A, int avain ){
2     int ala = 0, yla = A.size()-1;
3     while( ala < yla ){
4       int vali = (ala + yla)/2;
5       if( A[vali] == avain ){ return vali; }
6       else if( A[vali] > avain ){ yla = vali; }
7       else{ ala = vali; }
8     }
9     if( A[ala] == avain ){ return ala; }
10    else{ return -1; }
11  }
```

1.4 Puolitushakualiohjelman testaaminen tietokoneella

```
1   void testaa( const taulukko & A ){
2     for( int avain = 0; avain <= 9; ++avain ){
3       int tulos = p_haku( A, avain ), nn = A.size();
4       bool ok = ( -1 <= tulos && tulos < nn );
5       if( ok ){
6         bool mukana = false;
7         for( int ii = 0; ii < nn; ++ii ){
8           if( A[ii] == avain ){ mukana = true; break; }
9         }
10        if( tulos == -1 ){ ok = !mukana; }
11        else{ ok = ( A[ tulos ] == avain ); }
12      }
13      if( !ok ){ ilmoita virheestä }
14    }
15  }
```

Jos saadussa taulukossa on vain lukuja $1, \ldots, 8$, niin tämä testaa seuraavia tapauksia

- avain on pienempi kuin mikään taulukossa
- avain on suurempi kuin mikään taulukossa
- avain on taulukossa, testaa kaikki taulukon kohdat
- nolla tai useampia avain ei ole taulukossa -tapauksia

Osataanko kirjoittaa luotettava testaaja?

- ei, mutta se ei ole kovin iso ongelma
- testaajassa voidaan käyttää tehottomia yksinkertaisia ratkaisuja
- jos testaaja raportoi olemattoman virheen, niin testaajaa voidaan korjata

Testiaineiston valinta on iso ongelma!

- kattavan testiaineiston laatiminen on tyypillisesti mahdotonta
- voidaan laittaa tietokone laatimaan testiaineistoa automaattisesti

$\Rightarrow$ voidaan tehdä tuhansia testejä sekunnissa

Alkuvaiheessa ei kannatta miettiä liikaa testiaineiston laatua

- huonokin testiaineisto paljastaa osan ohjelmointivirheistä
- vasta sitten kun ne on korjattu, tarvitsee miettiä testiaineistoa huolellisemmin

Testausohjelman alku ja pääohjelma

```
1   #include <iostream>
2   #include <vector>
3   typedef std::vector< int > taulukko;
    ... tähän testikohde ja aliohjelma testaa
4   int main(){
5     taulukko A;
6     A.push_back(3); testaa(A); A.push_back(6); testaa(A);
7     A.push_back(7); testaa(A); A.clear(); testaa(A);
8     A.push_back(3); A.push_back(3); testaa(A);
9     ...
10    A.clear();
11    for( unsigned ii = 1; ii < 9; ++ii ){
12      for( unsigned jj = 0; jj < ii; ++jj ){ A.push_back(ii); }
13    }
14  }
```

Testaa taulukoilla $[3]$, $[3,6]$, $[3,6,7]$, $[]$, $[3,3]$, ..., $[1,2,2,3,3,3,\ldots,8,\ldots,8]$

Virheellinen puolitushakumme jäi ikuiseen silmukkaan

- sen aiheuttanut testisyöte paljastuu laittamalla testaaja rivin 3 edessä kertomaan, millä se testaa
- taulukko $[3,6]$, avain 4
  - rivi 3: $ala < yla$ on tosi $\rightsquigarrow$ riville 4
  - rivi 4: $vali$ saa arvon $\lfloor \frac{0+1}{2} \rfloor = 0$
  - rivit 5 ja 6 ohitetaan, koska $3 = A[0] = A[vali] < avain = 4$
  - rivi 7: ala saa arvon $vali$ eli 0

rivi	2	3	4	7	3	4	7	...
ala	0			0			0	...
$vali$			0			0		...
yla	1							...

Siis samaa alkiota $A[vali]$ tutkitaan uudelleen ja uudelleen

Korjaus: jätetään $vali$ hakualueen ulkopuolelle muuttamalla rivit 6 ja 7:

```
5   if( A[vali] == avain ){ return vali; }
6   else if( A[vali] > avain ){ yla = vali-1; }
7   else{ ala = vali+1; }
```

Sitten testikohde jäi kiinni tyhjällä taulukolla

- väittää löytäneensä avaimen 3 tyhjän taulukon paikasta 0
- tyhjässä taulukossa ei ole paikkaa 0
- sitä vastaavissa muistipaikoissa on edellisen testin $[3, 6, 7]$ jäljiltä 3

Tyhjällä taulukolla testaaminen siirrettiin ensimmäiseksi testiksi

$\Rightarrow$ testikohde kaatui

- selitys tarvitsee asioita, joita käsitellään luvussa 3.3

Korjaus: rivi 9 muotoon

```
if( A.size() > 0 && A[ala] == avain ){ return ala; }
```

Enää tiedossa ei ole muita virheitä kuin Blochin ylivuotovirhe

1.5 Toimiva puolitushaku

Olisi toisinaan hyödyllistä, jos

- jos avain on taulukossa, niin puolitushaku löytäisi ensimmäisen (eikä minkä tahansa)
- jollei löydy täsmälleen avaimen suuruista, niin palautettaisiin lähinnä suuremman paikka
  - vrt. tentin arvosana pistemäärän ja pisterajojen perusteella
  - mitä se tarkoittaa, kun avain on suurempi kuin mikään taulukossa?

$\Rightarrow$ palautettava se paikka, minkä vasemmalta alkava selaus palauttaisi (löytyi tai ohi)

- (käyttäjälle hieman lisätyötä, vaan niin on myös sen testaaminen, tuliko -1)

On siis palautettava luku, jolle pätee:

1. i on tulokselle sallitulla välillä: $0 \leq i \leq n$
2. i ei ole liian suuri: $i = 0$ tai $A[i-1] < avain$
3. i ei ole liian pieni: $i = n$ tai $A[i] \geq avain$

Sama on helppo ohjelmoida testaajaan, kunhan varoo tekemästä laittomia indeksointeja:

```
bool ok = ( 0 <= tulos && tulos <= nn );
if( ok ){ ok &= ( tulos == 0 || A[ tulos-1 ] < avain ); }
if( ok ){ ok &= ( tulos == nn || A[ tulos ] >= avain ); }
```

Toimiva uusien vaatimusten mukainen puolitushaku ja sen oikeellisuuden perustelu

- kukin väite pätee aina sen rivin lopussa, jonka kohdalla se on esitetty
- $0 \leq ala$ ja $yla \leq n$ pätevät rivin 1 jälkeen aina, mutta niitä ei toisteta siellä, missä se ei ole olennaista päättelylle

1	`int ala = 0, yla = A.size();`	$ala = 0 \leq n = yla$	$ala = 0$ ja $yla = n$
2	`while( ala < yla ){`	$0 \leq ala \leq yla \leq n$	(*)
3	`  int vali = (ala + yla)/2;`	$ala \leq vali < yla$	
4	`  if( A[vali] >= avain )`	$ala \leq vali < yla$	$A[vali] \geq avain$
5	`  { yla = vali; }`	yla pienenee; $ala \leq yla$	$A[yla] \geq avain$
6	`  else`	$ala < vali + 1 \leq yla$	$A[vali] < avain$
7	`  { ala = vali+1; }`	ala kasvaa; $ala \leq yla$	$A[ala - 1] < avain$
8	`}`		
9	`return ala;`	$0 \leq ala = yla \leq n$	(*)

(*): ($ala = 0$ tai $A[ala - 1] < avain$) ja ($yla = n$ tai $A[yla] \geq avain$)

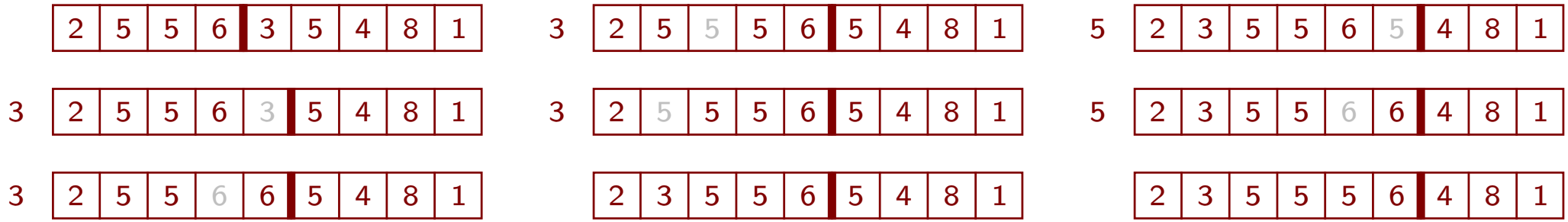
- $\frac{ala+yla}{2}$ on kokonaisluku tai puoliluku, esim. $5\frac{1}{2}$
- tarvittaessa se pyöristetään kokonaiseksi alaspäin, jotta voitaisiin sijoittaa `vali`:in
- koska $ala < yla$ rivillä 3, pätee $ala < \frac{ala+yla}{2} < yla$ ja $ala \leq \left\lfloor \frac{ala+yla}{2} \right\rfloor < yla$
- rivillä 9 $ala = yla$, koska sinne ei tulla jos $ala < yla$, ja osoitimme $ala \leq yla$
- edellä mainitut 1., 2. ja 3. pätevät rivillä 9, kun $i = ala$

- ainoan indeksoinnin `A[vali]` laillisuus seuraa suoraan siitä, että $0 \leq ala \leq vali < yla \leq n$
- rivillä 1 on ylivuodon vaara, jos `A.size()` ei mahdu `int`-tyyppiin
- rivillä 7 ei ole ylivuotovaaraa, koska $vali < yla$
- rivillä 3 on Blochin ylivuotovaara
  - korjaantuu muuttamalla rivi muotoon `int vali = ala + (yla-ala)/2`
- silmukka ja samalla koko aliohjelma lopettaa, koska
  - $yla - ala$ pienenee joka kierroksella mutta ei voi mennä alle nollan
  - muuta ikuisesti suorittamiselle altista ei aliohjelmassa ole

2 Silmukkainvariantit ja lisäysjärjestäminen

2.1 Lisäysjärjestäminen

- paras tunnettu yleiskäyttöinen menetelmä laittaa pieni taulukko suuruusjärjestykseen
- alkuosa kasvavassa järjestyksessä, loppuosa alkuperäisessä tilassa
- alkuosaa kasvatetaan lokero kerrallaan siirtämällä loppuosan ensimmäinen sinne



InsertionSort(&A)

1 **for** $i := 1$ **to** $A.koko - 1$ **do**
2 $\quad apu := A[i];\ j := i$
3 $\quad$ **while** $j > 0$ && $A[j-1].x > apu.x$ **do**
4 $\quad\quad A[j] := A[j-1];\ j = j-1$
5 $\quad A[j] := apu$

```
typedef std::vector< alkio > taulukko;
void InsertionSort( taulukko & A ){
  for( unsigned i=1; i<A.size(); ++i ){
    alkio apu = A[i]; unsigned j = i;
    while( j > 0 && A[j-1].x > apu.x ){
      A[j] = A[j-1]; --j;
    }
    A[j] = apu;
  }
}
```

2.2 Todistuksen jakaminen tapauksiin ja tapaus $n = 0$

- merkitsemme taulukon kokoa n:llä, siis $n = A.koko$
- jatkossa esitettävä todistus ei toimi kun $n = 0$

$\Rightarrow$ käsittelemme tapauksen $n = 0$ erikseen

- ilmiselvä tapaus
  - tyhjä taulukko on automaattisesti järjestyksessä
  - `InsertionSort` lopettaa heti alkuunsa tekemättä A:lle mitään

2.3 Ulompi silmukka paitsi sen runko

Silmukkainvariantti on väite, josta voidaan osoittaa kolme asiaa:

IE Se on tosi, kun silmukan alkuun tullaan silmukan edeltä.

IS Jos se on tosi kun ollaan silmukan alussa, ja jos lisäksi silmukan ehto on tosi, niin silmukkainvariantti on tosi myös kun alkuun tullaan uudelleen silmukan rungon suorittamisen jälkeen.

IH Jos se on tosi kun silmukan kiertäminen lopetetaan tai silmukka ohitetaan, niin silmukalta haluttu asia on tosi.

IE ja IS $\Rightarrow$ silmukkainvariantti on tosi aina silmukan alussa

- koskaan ei ole ***ensimmäinen kerta***, jolloin se ei ole tosi silmukan alussa

Tarkoittakoon $A[a \ldots y]$, missä $y \geq a-1$, osataulukkoa $A[a]$, $A[a+1]$, $\ldots$, $A[y]$

InsertionSort:n ulomman silmukan invariantti, kun $n > 0$ (mikä ei toimi, kun $n = 0$?):

1. Osassa $A[0 \ldots i-1]$ on alkuperäiset alkiot kasvavassa järjestyksessä.
2. Osassa $A[i \ldots n-1]$ on alkuperäiset alkiot alkuperäisessä järjestyksessä.

IE ja IH toteutuvat, koska ... (IS osoitetaan luvussa 2.6)

2.4 Ulomman silmukan lopettaminen

IE, IS ja IH

- takaavat, että ***jos*** päästään loppuun, niin lopputulos on oikea
- ***eivät takaa***, että päästään loppuun

Syitä epäonnistua loppuun pääsemisessä

- ikuinen silmukka
- kaatuminen suoritusaikaiseen virheeseen

Aito **for**-silmukka (perusversio)

$$\textbf{for } i := \mathit{alku} \textbf{ to } \mathit{loppu} \textbf{ do } \mathit{runko}$$

- silmukkamuuttujaan ei saa sijoittaa silmukassa (ei ainakaan vähentää)
- loppu lasketaan silmukan aloitushetkellä, eikä joka kierroksella erikseen
  - vrt. $\textbf{for } i := n \textbf{ to } n+5 \textbf{ do } n := n+1$ ja `for( i = n; i <= n+5; ++i ){ ++n; }`
- saa poistua kesken esim. **return**:lla
- loppu saa olla paljon pienempi kuin alku

$\Rightarrow$ enintään $\max\{\mathit{loppu} - \mathit{alku} + 1, 0\}$ kierrosta

- (on erikseen selvitettävä, voiko kierroksen sisällä eli rungossa olla ikuinen silmukka)

Yksityiskohdat tarkastamalla näkee, että `InsertionSort`:n `for`-silmukka on aito

2.5 Ei laittomia toimintoja

```
InsertionSort(&A)
1  for i := 1 to A.koko − 1 do
2      apu := A[i]; j := i
3      while j > 0 && A[j − 1].x > apu.x do
4          A[j] := A[j − 1]; j = j − 1
5      A[j] := apu
```

Taulukoiden indeksoinnit

- riveillä 2, . . . , 5 pätee $1 \leq i < n$

$\Rightarrow$ siellä pätee $0 \leq j \leq i < n$

$\Rightarrow$ jokainen $A[i]$ ja $A[j]$ on laillinen

- ennen kumpaakin $A[j-1]$ on todettu, että $j > 0$

$\Rightarrow$ nekin ovat lailliset

Muistin loppuminen

- harvinaista pienillä ohjelmilla nykykoneissa
- emme ole nyt tekemässä lentokonetta lentävää tai ydinvoimalaa valvovaa ohjelmistoa

$\Rightarrow$ vaikea keksiä muuta järkevää kuin antaa ohjelman kaatua

Lukualueen ylivuodot

- riippuu muuttujien tyypeistä
- tarkastamme yksityiskohdat

$\Rightarrow$ tarpeeksi isolla kokonaislukutyypillä ei vaaraa

- etumerkittömällä tyypillä huoledittava, että rivi 1 toimii myös kun $n = 0$

2.6 Ulomman silmukan runko

Osoitettava, että ulomman silmukan invariantti säilyy voimassa, kun runko suoritetaan

Toisin sanoen, rungon lopussa sen täytyy päteä muutettuna i:n tilalle $i+1$:

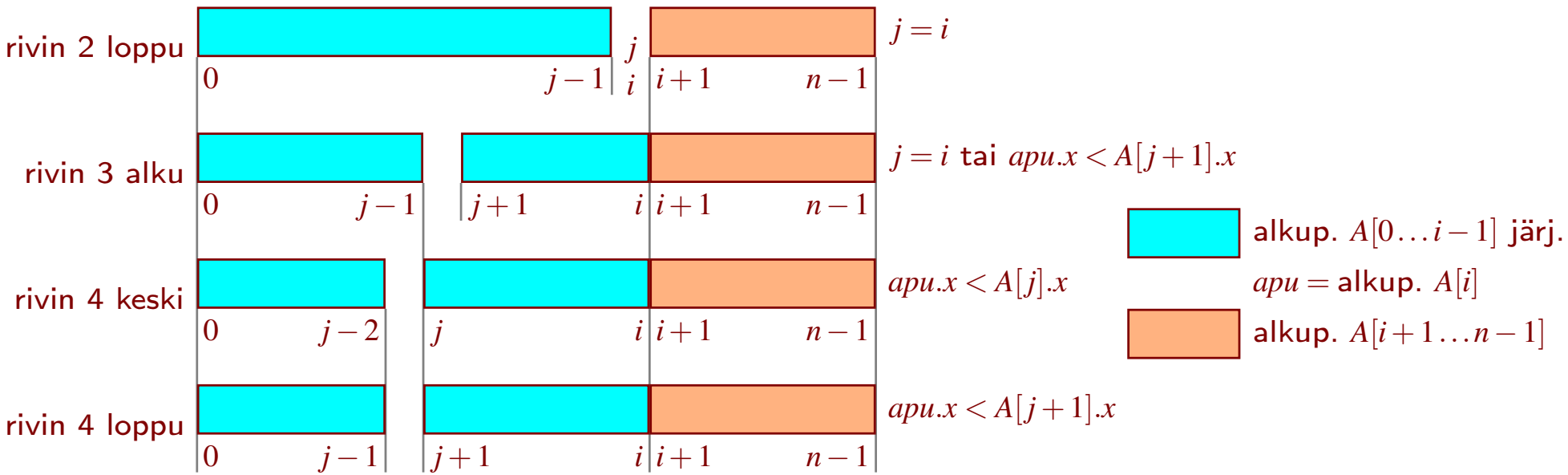
1. Osassa $A[0 \ldots i]$ on alkuperäiset alkiot kasvavassa järjestyksessä.
2. Osassa $A[i+1 \ldots n-1]$ on alkuperäiset alkiot alkuperäisessä järjestyksessä.

Jälkimmäinen säilyy, koska

- se on osa siitä mikä luvattiin rivin 1 alussa
- sen kannalta mikään ei muutu

Sisemmän silmukan invariantti:

1. Osissa $A[0 \ldots j-1]$ ja $A[j+1 \ldots i]$ on yhteensä osan $A[0 \ldots i-1]$ alkuperäiset alkiot kasvavassa järjestyksessä.
2. Alkuperäinen $A[i]$ on muuttujassa apu.
3. Joko $j = i$ tai $apu.x < A[j+1].x$.



Tullaan edeltä:

1. $A[j+1 \ldots i]$ on tyhjä; $A[0 \ldots j-1] = A[0 \ldots i-1] =$ alkup. $A[0 \ldots i-1]$ kasvavasti
2. $apu = A[i] =$ alkup. $A[i]$ $\Uparrow$ ulompi inv. 1
3. $j = i$ $\Uparrow$ ulompi inv. 2

Säilyy rungossa:

1. rivillä 4 $A[0\ldots j-1, j+1\ldots i] \rightsquigarrow A[0\ldots j-2, j\ldots i] \rightsquigarrow A[0\ldots(j+1)-2,(j+1)\ldots i]$
2. apu ja alkuperäinen $A[i]$ eivät muutu
3. rivin 3 vuoksi rivillä 4 $A[j-1].x > apu.x \rightsquigarrow A[j].x > apu.x \rightsquigarrow A[j+1].x > apu.x$

Toteuttaa halutun: rivin 5 jälkeen

- $A[0\ldots i]$:ssä on alkuperäiset alkiot (järjestys voi olla muuttunut)
  - I1 $\Rightarrow A[0\ldots j-1]$ ja $A[j+1,\ldots,i]$ ovat yhdessä $A[0\ldots i-1]$:n alkuperäiset
  - $A[j] = apu =$ alkuperäinen $A[i]$ (I2)
- järjestys on kasvava
  - $j = 0$ tai $A[j-1].x \leq A[j].x$ silmukan lopetusehdon vuoksi ja koska $A[j] = apu$
  - $j = i$ tai $A[j].x < A[j+1].x$ I3:n vuoksi ja koska $A[j] = apu$
  - muissa kohdissa kasvavuus seuraa I1:stä

Sisempi silmukka lopettaa viimeistään i kierroksen jälkeen

- j aloittaa arvosta i
- j pienenee joka kierroksella
- j ei voi mennä alle nollan
- silmukan rungossa ei ole silmukoita tms.

```
    InsertionSort(&A)
1   for i := 1 to A.koko − 1 do
2       apu := A[i]; j := i
3       while j > 0 && A[j − 1].x > apu.x do
4           A[j] := A[j − 1]; j = j − 1
5       A[j] := apu
```

2.7 Vakaus

Alkiossa on usein muutakin tietoa kuin järjestämisessä käytettävä avain

- esim. sähköpostit järjestetään usein vain lähetysajan mukaan

Toisinaan halutaan järjestää usean kriteerin mukaan, esim. korkeushyppy:

- ensisijaisesti ratkaisee suurin ylitetty korkeus
- tasatilanteessa voittaa hän, joka käytti ylimpään korkeuteen vähiten yrityksiä
- jos yhä tasatilanne, niin seuraavaksi verrataan aikaisempien pudotusten yhteismäärää

Voidaan toteuttaa monimutkaisella alkioiden vertaamistoiminnolla

- jokaista yhdistelmää varten tarvitaan oma toiminto

Järjestämisalgoritmi on ***vakaa***, jos ja vain jos se ei koskaan muuta sellaisten alkioiden keskinäistä järjestystä, joilla on sama avain

$\Rightarrow$ voidaan järjestää usean kriteerin mukaan aloittamalla viimeisenä sovellettavasta

$\Rightarrow$ ei tarvita erillistä vertailutoimintoa jokaiselle kriteerien yhdistelmälle

Esim. korkeushyppy

- ensin järjestetään alempien korkeuksien pudotusten yhteismäärän mukaan
- seuraavaksi ylimmän ylitetyn korkeuden pudotusten määrän mukaan
- lopuksi takaperin tuloskorkeuden mukaan

Esimerkki: sähköpostien järjestäminen vakaalla algoritmilla

viestit aikajärjestyksessä

9:15	Späde Spämmäri	Tsekkaa tämä!
11:46	Yvonne Ystävä	Syömään?
12:03	Nippe Nälkäinen	Syömään?
13:30	Späde Spämmäri	Juoru...
17:41	Kalle Kaveri	Ongelma ratkesi
22:28	Späde Spämmäri	Kissavideo :-)

klikattu aihe

13:30	Späde Spämmäri	Juoru...
22:28	Späde Spämmäri	Kissavideo :-)
17:41	Kalle Kaveri	Ongelma ratkesi
11:46	Yvonne Ystävä	Syömään?
12:03	Nippe Nälkäinen	Syömään?
9:15	Späde Spämmäri	Tsekkaa tämä!

klikattu lähettäjä

17:41	Kalle Kaveri	Ongelma ratkesi
12:03	Nippe Nälkäinen	Syömään?
13:30	Späde Spämmäri	Juoru...
22:28	Späde Spämmäri	Kissavideo :-)
9:15	Späde Spämmäri	Tsekkaa tämä!
11:46	Yvonne Ystävä	Syömään?

klikattu aihe

13:30	Späde Spämmäri	Juoru...
22:28	Späde Spämmäri	Kissavideo :-)
17:41	Kalle Kaveri	Ongelma ratkesi
12:03	Nippe Nälkäinen	Syömään?
11:46	Yvonne Ystävä	Syömään?
9:15	Späde Spämmäri	Tsekkaa tämä!

InsertionSort on vakaa

- kahden alkion järjestys voi vaihtua vain rivillä 4
- toinen niistä on $A[j-1]$ ja toinen on apu
- rivin 3 vuoksi riviä 4 ei suoriteta, jos $A[j-1].x = apu.x$

InsertionSort(&A)

1 **for** $i := 1$ **to** $A.koko - 1$ **do**
2 $\quad apu := A[i];\ j := i$
3 $\quad$ **while** $j > 0$ && $A[j-1].x > apu.x$ **do**
4 $\quad\quad A[j] := A[j-1];\ j = j-1$
5 $\quad A[j] := apu$

2.8 O-, Ω- ja Θ-merkinnät

Ohjelman suoritusaikaan voi vaikuttaa tietenkin syötteen koko, mutta myös syötteen laatu

- InsertionSort on paljon nopeampi järjestyksessä kuin takaperin järjestyksessä olevalle taulukolle

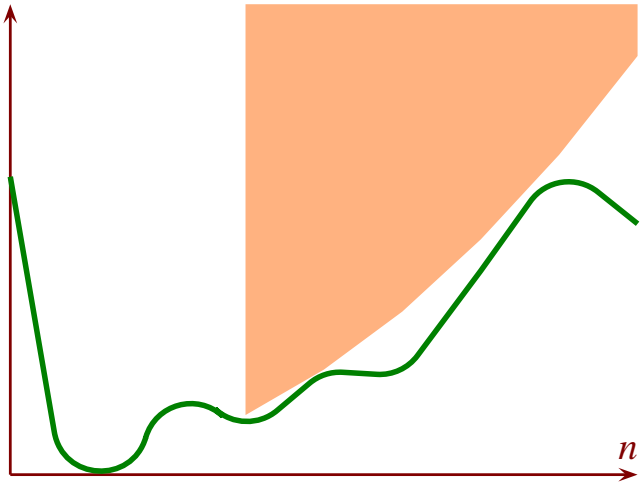
Ohjelman suoritusaikaan vaikuttaa moni muukin asia

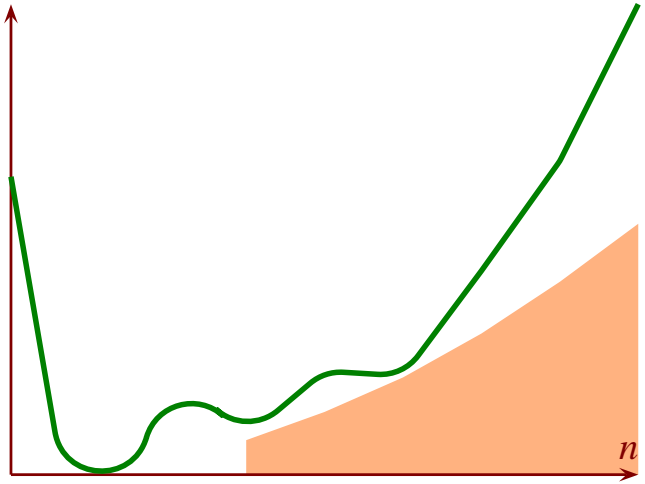
- tietokone, jossa ohjelma suoritetaan
- kääntäjä, jolla ohjelma käännettiin
- mitä muuta tietokone tekee samalla
- ...
- näiden merkitys on usein pieni verrattuna syötteen kokoon ja kasvukäyrän muotoon

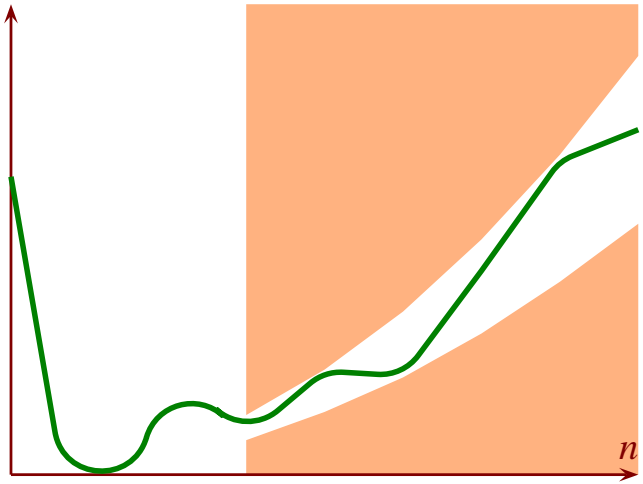
Suoritusajan kasvukäyrän muotoa kuvataan O-, Ω- ja Θ-merkinnöillä

- huomioon otetaan vain mitä tapahtuu suurilla syötteillä
- huomioon otetaan vain suoritusaikaan suurilla syötteillä eniten vaikuttavat tekijät
- unohdetaan ajan yksiköt (sekunti, vuosi) ja vakiokertoimet
- syötteen kokoa kuvataan usein muuttujalla n
- voi olla esim. että n on tieverkon risteysten määrä ja m on tienpätkien määrä

$O(\ldots)$: enintään niin paljon $\Omega(\ldots)$: ainakin niin paljon $\Theta(\ldots)$: ainakin ja enintään ...







Puuttuu aikayksikkö ja tieto kuinka suuri on "suuri", joten mitä merkitystä näillä on?

- hyvin usein ne kertovat suoritusajoista niin paljon, että enempää ei tarvita!
- tyypillisesti (mutta ei aina) "tavallisen iso" on "suuri"

$\Rightarrow$ O-, Ω- ja Θ-merkinnät eivät kerro mistä syötteen koosta alkaen hitaus on ongelma, mutta hyvin usein ne kertovat, tuleeko se olemaan ongelma

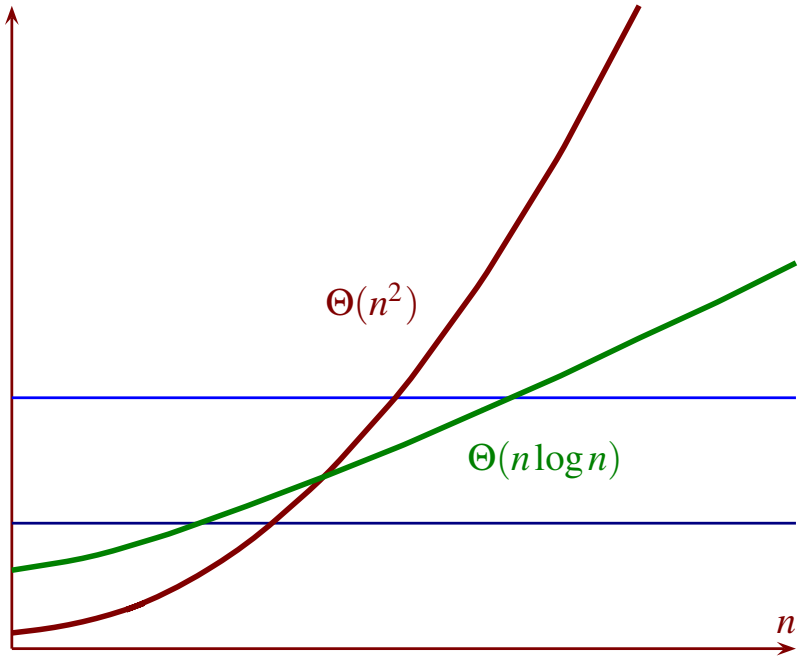
- toisin kuin mittaukset, ne ovat käytettävissä ennen kuin mitään on toteutettu

$\Rightarrow$ voidaan käyttää kun valitaan, mikä vaihtoehto toteutetaan

- melkein aina mikä tahansa edes jossain määrin järkevä ratkaisu on pienillä syötteillä tarpeeksi nopea

$\Rightarrow$ tyypillisesti ei tarvita tietoa mitä tapahtuu pienillä syötteillä

Yksinkertainen, pienillä syötteillä nopea vastaan monimutkainen, Θ-mielessä parempi



- tummansininen: askelten määrä ajassa, joka ei tunnu loppukäyttäjästä liian pitkältä
- vaaleansininen: sama nopeammalla tietokoneella

Käyrän muoto voi riippua syötteen laadusta

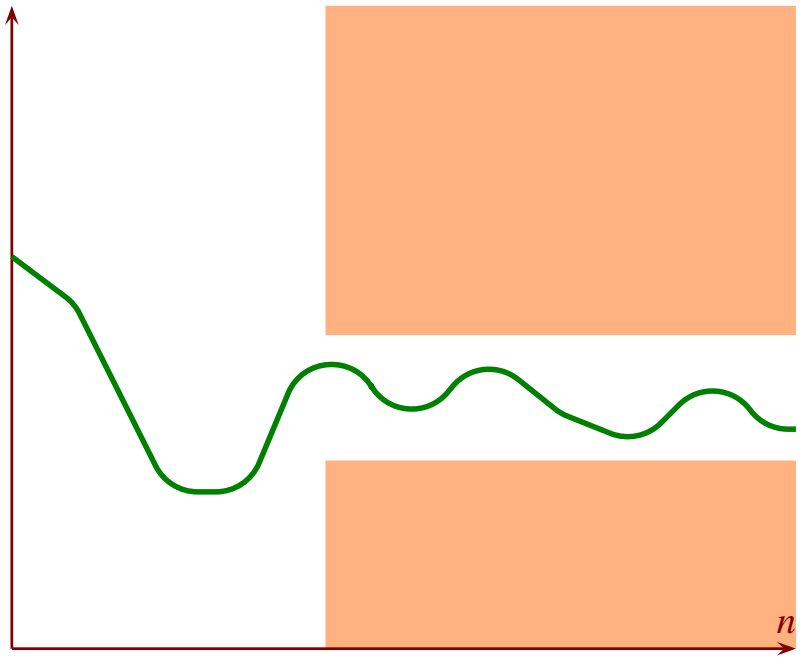
- jos tosielämän syöte on erilaatuista kuin oletettiin, Θ- jne. voivat antaa väärän kuvan
- jos tosielämän syöte on erilaatuista kuin mittauksissa, mittaukset ...
- tavallisesti keskitytään Θ-merkintään hitaimmassa tapauksessa
- jos se on riittävän hyvä, niin suoritusaika on riittävän hyvä kaikenlaisilla syötteillä

On myös tapauksia, joissa Θ-merkintä antaa käytännön kannalta väärän kuvan

- ääriesimerkki: galaktiset algoritmit

Näitä merkintöjä käytetään myös kuvaamaan muistin kulutusta

Mitä tarkoittaa $\Theta(1)$?



- ajan kulutus pysyy kahden vakion välissä, paitsi ehkä pienillä syötteillä
- usein sanotaan että ajan kulutus on vakio, mutta se ei ole tarkalleen ottaen oikein

Vapautus pienistä syötteistä tekee käsitteestä toimivan, vaikka

- pinosta poistaminen ei ole määritelty kun $n = 0$
- $\log n$ ei ole määritelty kun $n = 0$

O, Θ ja Ω ***eivät ole*** suunnilleen sama kuin matematiikan raja-arvo

2.9 Lisäysjärjestämisen suoritusaika

Ajan käytön näkökulmasta lisäysjärjestämisessä on seuraavat:

- kaksi silmukkaa: $\Theta(n)$ ja sen sisällä $\Omega(1) \ldots O(n)$
- lukujen ja taulukon alkioiden sijoituksia: kukin $\Theta(1)$
- apumuuttujan ja taulukon alkioiden avainten suuruusjärjestysvertailuja: kukin $\Theta(1)$
- $+1$, -1, `&&`: kukin $\Theta(1)$

Alkioiden sijoituksen ja vertaamisen ajan kulutus riippuu alkioiden koosta

- silti ajatellaan olevan $\Theta(1)$, koska
  - ei riipu n:stä
  - ei muutu suorituksen aikana
- luvussa 5.3 nähdään, että suoritusaika voi riippua olennaisesti alkioiden koosta
  - jos halutaan, niin voidaan ottaa käyttöön vaikka k ilmaisemaan alkion kokoa

  $\Rightarrow$ alkioiden sijoitus onkin $\Theta(k)$
- entä vertaaminen
  - voi olla luku $\ll$ avain $\ll$ alkio

  $\Rightarrow$ otetaanko käyttöön kolmaskin muuttuja?
- järjestämisalgoritmien analyysissä on syötekoon lisämuuttujista hyötyä vain, jos on poikkeuksellisen tiukat nopeusvaatimukset

$\Rightarrow$ emme ota niitä käyttöön ennen lukua 7.4 (ja sielläkin vain toisen, ja eri syystä)

for-silmukka kiertää (ainakin) $n-1$ kierrosta $\Rightarrow$ $\Omega(n)$

- myös $\Omega(\sqrt{n})$ ja $\Omega((\log n)^3)$, mutta $\Omega(n)$ kertoo eniten
- ehkä myös $\Omega(n\sqrt{n})$ ja se kertoisi enemmän, mutta emme vielä tiedä onko se totta

Vertaa "juna IC99 on myöhässä, arvioitu lähtöaika 9:20"

- ei lupaa etteikö lähtö tapahtuisi vasta 9:30
- lupaa että ei lähde ennen 9:20
- kuuluttaja kertoo myöhäisimmän ajan, josta tietää että juna ei lähde sitä ennen

Siis tavallisesti kerrotaan informatiivisin, jonka tiedetään olevan totta

while-silmukka kiertää enintään i kierrosta

- tiedämme $i < n$

$\Rightarrow$ $O(n)$ **for**-silmukan kierrosta kohti

$\Rightarrow$ kaikkiaan $O(n^2)$

- työläämmin
  - rivi 3 suoritetaan $\leq 2 + \ldots + n \leq (n-1)n \leq n^2$ kertaa
  - kukin muu rivi enintään $\max\{n+1, n^2\}$ kertaa

InsertionSort(&A)

```
1  for i := 1 to A.koko − 1 do
2      apu := A[i]; j := i
3      while j > 0 && A[j − 1].x > apu.x do
4          A[j] := A[j − 1]; j = j − 1
5      A[j] := apu
```

$O(n^2)$ on tyypillisesti ikävän hidas suurilla syötteillä

- onkohan se vain pessimistinen ylälikiarvo, ja todellinen nopeus olisi parempi?
- jos A on takaperin järjestyksessä, niin kukin alkio siirretään i askelta
  - yhteensä $1+2+\ldots+n-1 = \frac{1}{2}n^2 - \frac{1}{2}$ askelta $\Rightarrow$ $\Omega(n^2)$ ja $\Theta(n^2)$
  - (luvussa 3.5 kerrotaan, miten tällaisesta saa helposti Ω-merkinnän)
- siis $O(n^2)$ on paras, minkä voi ilmaista O-muodossa
- toinen tapa sanoa sama: InsertionSort on ***hitaimmillaan*** $\Theta(n^2)$

Huomaa ero:

- suoritusaika on $O(n^2)$: ei koskaan hitaampi, ei välttämättä edes niin hidas
- suoritusaika on hitaimmillaan $\Theta(n^2)$:
  ei koskaan hitaampi, on niin hidas ***äärettömän monella eri syötteellä***
- suoritusaika on $\Omega(n)$: ei koskaan nopeampi, ei välttämättä edes niin nopea
- suoritusaika on nopeimmillaan $\Theta(n)$:
  ei koskaan nopeampi, on niin nopea ***äärettömän monella eri syötteellä***

InsertionSort on nopeimmillaan $\Theta(n)$

- toteutuu, jos A on alun perin kasvavassa järjestyksessä
  - silloin rivin 3 ehdon osuus $A[j-1].x > apu.x$ ei koskaan toteudu
  - $\Rightarrow$ rivi 3 suoritetaan vain kerran kullekin i, eikä riviä 4 suoriteta kertaakaan

Siis InsertionSort:n suoritusaika vaihtelee välillä $\Theta(n), \ldots, \Theta(n^2)$

Mikä on suoritusaika tyypillisessä tapauksessa?

- ensin pitäisi tietää, kuinka paljon avaimet saavat toistua
- jos avaimet eivät saa yhtään toistua ja alkuperäinen järjestys on satunnainen, niin InsertionSort:n keskimääräinen suoritusaika on $\Theta(n^2)$
  - kukin alkio siirtyy keskimäärin puolet A:n jo järjestetyn osan koosta

  $\Rightarrow$ suoritusaika suunnilleen puolet hitaimman tapauksen suoritusajasta

Esimerkki, miten ei kannata tehdä

```
1   class vertaaja{
2   public:
3     bool operator()( const alkio & eka, const alkio & toka ){
4       return eka.x < toka.x;
5     }
6   };

7   void EraseInsert( taulukko & A ){
8     vertaaja vrt;
9     for( unsigned i = 0; i < A.size(); ++i ){
10      alkio apu = A[i]; A.erase( A.begin()+i );
11      taulukko::iterator j =
12        std::upper_bound( A.begin(), A.begin()+i, apu, vrt );
13      A.insert( j, apu );
14    }
15  }
```

2.10 Muistin tarve

Muistia tarvitaan järjestettävän taulukon lisäksi vain muuttujille apu, i, j ja n

- lisämuistin tarve on $\Theta(1)$ ja tarkemminkin ilmaistuna erittäin pieni

2.11 Lisää kysymyksiä

```
    void SelectionSort( taulukko & A ){
1     for( unsigned i = 0; i+1 < A.size(); ++i ){
2       unsigned p = i;
3       for( unsigned j = i+1; j < A.size(); ++j ){
4         if( A[j].x < A[p].x ){ p = j; }
5       }
6       alkio apu = A[i]; A[i] = A[p]; A[p] = apu;
7     }
8   }
```

3 Kekojärjestäminen ja prioriteettijono

3.1 Keot

Keko (***heap***) tarkoittaa kahta aivan eri asiaa:

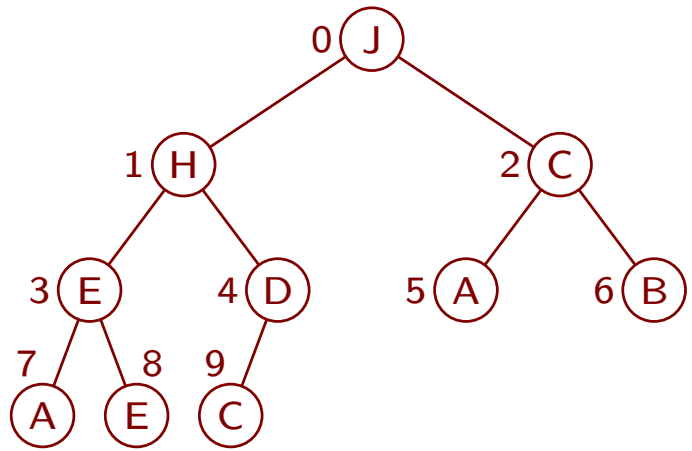
- muistialue, josta varataan muistia mm. `new`:lla luotaville olioille
- tietorakenne, jota käsitellään tässä luvussa

Keko tietorakenteena

- tietyllä tavalla järjestetty taulukko
- alkion lisääminen $O(\log n)$
- alkion poistaminen $O(\log n)$
- suurimman alkion katsominen $\Theta(1)$ (vaihtoehtoisesti pienimmän, ei molempien)
- toisinaan (harvoin) tarkoittaa myös samankaltaista linkitettyä rakennetta

(Binääri)keko jossa on jokin suurin ensimmäisenä, ja havainnollistus binääripuuna

J	H	C	E	D	A	B	A	E	C
0	1	2	3	4	5	6	7	8	9



- binääripuun ylin solmu on ***juuri***
- solmun ***syvyys*** on juuresta solmuun vievän polun pituus
  - juuren syvyys on 0
- samalla syvyydellä olevat solmut muodostavat ***taso***n
- alin taso on täytetty vasemmalta alkaen, muut tasot ovat täydet
- kunkin solmun avain on enintään yhtäsuuri kuin lähinnä ylemmän solmun avain
- lähinnä ylempi solmu on ***vanhempi***, lähinnä alempi on ***lapsi***
- joko ei lapsia, pelkkä ***vasen lapsi*** tai sekä vasen että ***oikea lapsi***
- tässä luvussa, ellei toisin sanota, "keko" tarkoittaa "binäärikeko jokin suurin ensimm."

Lapset ja vanhempi taulukon lokeroina

- lokeron 0 lapset, siltä osin kuin ovat olemassa, ovat lokeroissa 1 ja 2
- lokeron 1 lokeroissa 3 ja 4, lokeron 2 lokeroissa 5 ja 6, lokeron 3 lokeroissa 7 ja 8, ...
- jos $0 \leq i \leq ?$, niin lokeron i lapset ovat lokeroissa $2i+1$ ja $2i+2$
- jos i on parillinen ja $1 \leq i < n$, niin lokeron i vanhempi on lokerossa $\frac{i-2}{2}$
- jos i on pariton ja $1 \leq i < n$, niin lokeron i vanhempi on lokerossa $\frac{i-1}{2}$

Lattiafunktio $\lfloor \ldots \rfloor$

- puhuminen erikseen parillisista ja erikseen parittomista n on kömpelöä
- sen voi välttää merkinnällä $\lfloor x \rfloor$
  - suurin kokonaisluku, joka on enintään x
  - toisin sanoen, x pyöristettynä alaspäin lähimpään kokonaislukuun
- jos n on parillinen, niin $\lfloor \frac{n}{2} \rfloor = \frac{n}{2}$
- jos n on pariton, niin $\lfloor \frac{n}{2} \rfloor = \frac{n-1}{2}$
- monissa ohjelmointikielissä kokonaisluvuilla `n/m` laskee $\lfloor \frac{\mathtt{n}}{\mathtt{m}} \rfloor$ eikä $\frac{\mathtt{n}}{\mathtt{m}}$, kun $\mathtt{n} \geq 0$ ja $\mathtt{m} > 0$
- mitä `n/m` laskee kun `n` tai `m` tai molemmat ovat negatiivisia vaihtelee eri kielissä
  - ks. Wikipedia "Modulo" $\rightsquigarrow$ "In programming languages"

Keon $A[0 \ldots n-1]$ perusominaisuus:

Jokaisella i väliltä $1, \ldots, n-1$ pätee $A[\lfloor \frac{i-1}{2} \rfloor].x \geq A[i].x$.

- jos i on parillinen, niin $i-1$ on pariton ja $\lfloor \frac{i-1}{2} \rfloor = \frac{i-2}{2}$
- jos i on pariton, niin $i-1$ on parillinen ja $\lfloor \frac{i-1}{2} \rfloor = \frac{i-1}{2}$

Indeksointi ykkösestä alkaen

- tavallista algoritmikirjallisuudessa
- vanhempi ja lapset pitää laskea hieman toisin kuin edellä

	jos	, niin lokeron	vanhempi on lokerossa
taulukolle $A[0 \ldots n-1]$	$1 \leq i \leq n-1$	i	$\lfloor \frac{i-1}{2} \rfloor$
taulukolle $A[1 \ldots n]$	$1 \leq i \leq n-1$	$i+1$	$\lfloor \frac{i-1}{2} \rfloor + 1$
$i-1$ eikä i juoksee	$1 \leq (i-1) \leq n-1$	$(i-1)+1$	$\lfloor \frac{(i-1)-1}{2} \rfloor + 1$
sievennys	$2 \leq i \leq n$	i	$\lfloor \frac{i}{2} \rfloor$

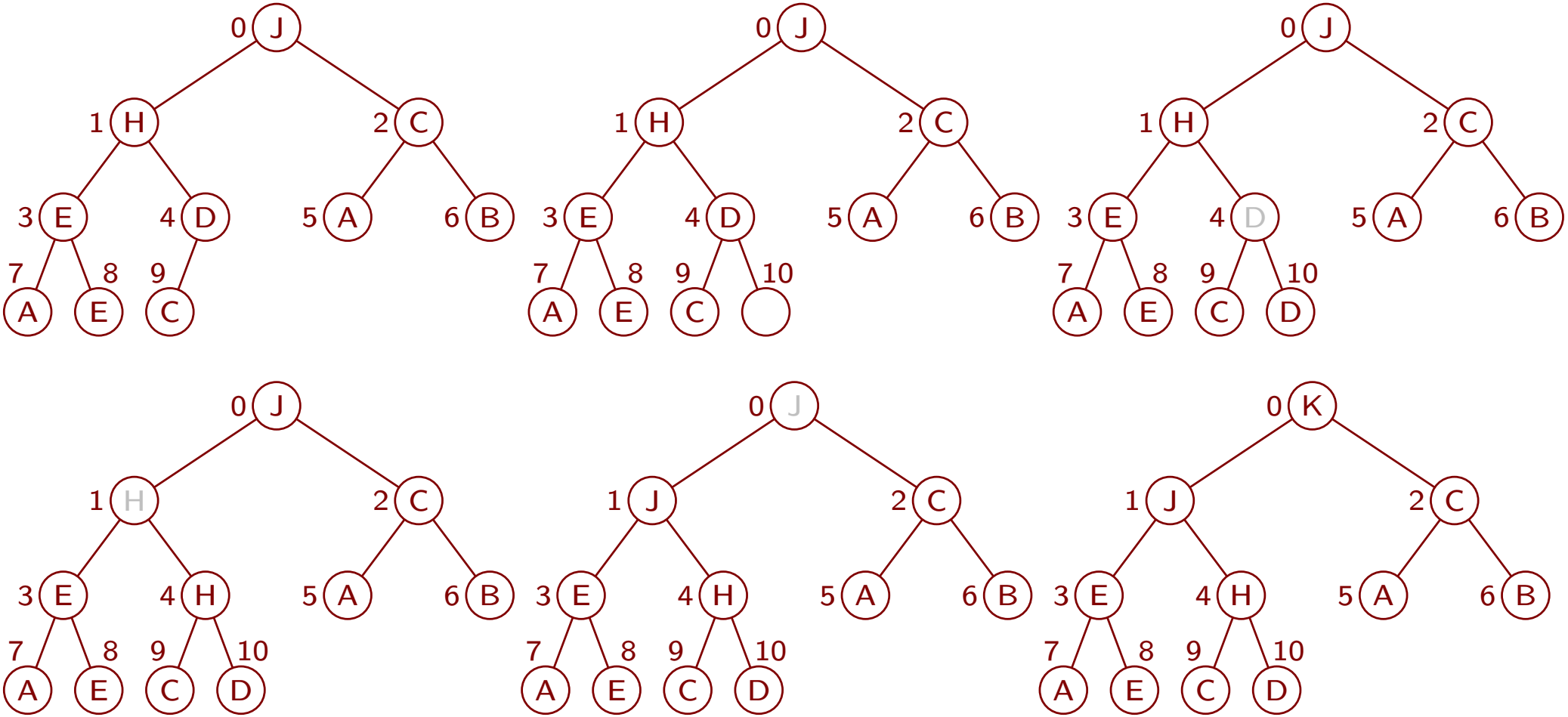
- päättelyvirheitä voi tapahtua, joten testaamme

lokero i	2	3	4	5	6
vanhemman lokero $\lfloor \frac{i}{2} \rfloor$	1	1	2	2	3

3.2 Kekoon lisääminen ja jonkin suurimman poistaminen

Kekoon lisääminen (esim. lisätään K)

- keon loppuun lisätään lokero
- sen vanhempi kopioidaan uuteen lokeroon, isovanhempi vanhemman tilalle jne. kunnes tyhjä kohta on lisätyn alkion avaimen mukaisessa kohdassa



Lisäys pseudokoodina

$\underline{\textsc{LisääKekoon}(\&A, uusi)}$

1 $i := A.koko;\ j := \lfloor \frac{i-1}{2} \rfloor$
2 $A.kooksi(i+1)$
3 **while** $i > 0$ && $A[j].x < uusi.x$ **do**
4 $\quad A[i] := A[j];\ i := j;\ j := \lfloor \frac{i-1}{2} \rfloor$
5 $A[i] := uusi$

Indeksointien laillisuus

- rivin 1 lopussa $i \geq 0$
- rivin 2 lopussa $0 \leq i = n-1$
- rivin 3 alussa jos $0 < i < n$, niin $i-1 \geq 0$
  $\Rightarrow\ j = \lfloor \frac{i-1}{2} \rfloor \geq 0$ ja $j = \lfloor \frac{i-1}{2} \rfloor \leq \frac{i-1}{2} \leq i-1 < i < n$
  $\Rightarrow$ rivin 4 lopussa $0 \leq i < n$

$\Rightarrow$ rivien 3 ja 4 silmukalla on invariantti $0 \leq i < n$

$\Rightarrow$ kaikki $A[i]$:t ja $A[j]$:t ovat lailliset

- jatkoa varten toteamme, että i pienenee rivillä 4

Muuta

- pysähtyy, koska i pienenee rivillä 4 ja rivi 3 testaa $i > 0$
- muisti voi loppua kesken rivillä 2
- C++:n `unsigned` ei kaadu alivuodosta $\Rightarrow$ `(i-1)/2` ei haittaa kun `i` $= 0$

Olettaen, että $A[0 \ldots n-1]$ on keko kun LISÄÄKEKOON aloittaa, on rivien 3 ja 4 silmukalla seuraava invariantti:

1. Alkio j on alkion i vanhempi, eli $j = \lfloor \frac{i-1}{2} \rfloor$.
2. Osissa $A[0 \ldots i-1]$ ja $A[i+1 \ldots n-1]$ on yhteensä alkuperäiset alkiot.
3. Jokaisella k väliltä $1, \ldots, n-2$ on alkion k vanhempi vähintään yhtäsuuri kuin alkio k, eli $A[\lfloor \frac{k-1}{2} \rfloor].x \geq A[k].x$.
4. $i = n-1$ tai osan 3 väite pätee myös kun $1 \leq k = n-1$.
5. $i = n-1$ tai $A[i].x < uusi.x$.

IE: Riville 3 tullaan sen edeltä

- I1 asetettiin voimaan rivillä 1
- I4 ja I5 pätevät, koska $i = n-1$
- I2 pätee, koska $i = n-1$ ja n kasvoi yhdellä rivillä 2
- I3 pätee, koska $A[0 \ldots n-1] \rightsquigarrow A[0 \ldots n-2]$ oli keko

1 $i := A.koko;\ j := \lfloor \frac{i-1}{2} \rfloor$
2 $A.kooksi(i+1)$

IH: Rivin 5 lopussa

- I2:n ja rivin 5 vuoksi A:ssa on alkuperäiset alkiot ja uusi alkio
- jos $i \neq n-1$, niin I4:n ja I3:n vuoksi A on keko
- jos $i = n-1 = 0$, niin A muodostuu pelkästään uudesta alkiosta, joten se on keko
- muutoin $i = n-1 > 0$
  - riviä 4 ei ole suoritettu kertaakaan
  - $uusi.x \leq A[j].x$, missä j on $(n-1)$:n vanhempi
  - $A[n-1] = uusi$

  $\Rightarrow$ $A[0 \ldots n-2]$ on I3:n vuoksi keko, ja $uusi$ on oikealla paikallaan viimeisenä

3 **while** $i > 0$ && $A[j].x < uusi.x$ **do**

4 $\quad A[i] := A[j]$; $i := j$; $j := \lfloor \frac{i-1}{2} \rfloor$

5 $A[i] := uusi$

IS: Invariantin säilyminen rivin 3 ehdolla rivillä 4

- I1 asetetaan voimaan rivin 4 lopussa
- I2 säilyy, koska lauseen $A[i] := A[j]$ jälkeen sen mainitsemat alkuperäiset alkiot ovat $A[0 \ldots j-1, j+1 \ldots n-1]$, ja $i := j \rightsquigarrow A[0 \ldots i-1, i+1 \ldots n-1]$
- I3 säilyy, koska lauseessa $A[i] := A[j]$
  - $A[i]$ tulee yhtä suureksi kuin vanhempansa
  - I3:n vuoksi $A[i]$ säilyy entisen suuruisena tai kasvaa

  $\Rightarrow$ $A[i]$ on yhä lapsiinsa nähden vähintään yhtä suuri
- I5:n osa $A[i].x < uusi.x$ astuu voimaan testin $A[j].x < uusi.x$ ja lauseen $i := j$ vuoksi
- I4:n loppuosa astuu ensimmäisellä kierroksella voimaan muodossa $A[\lfloor \frac{n-2}{2} \rfloor].x = A[n-1].x$, ja sen jälkeen $A[n-1]$ ei muutu ja $A[\lfloor \frac{n-2}{2} \rfloor]$ ei muutu tai kasvaa

(Yhden) suurimman poistaminen

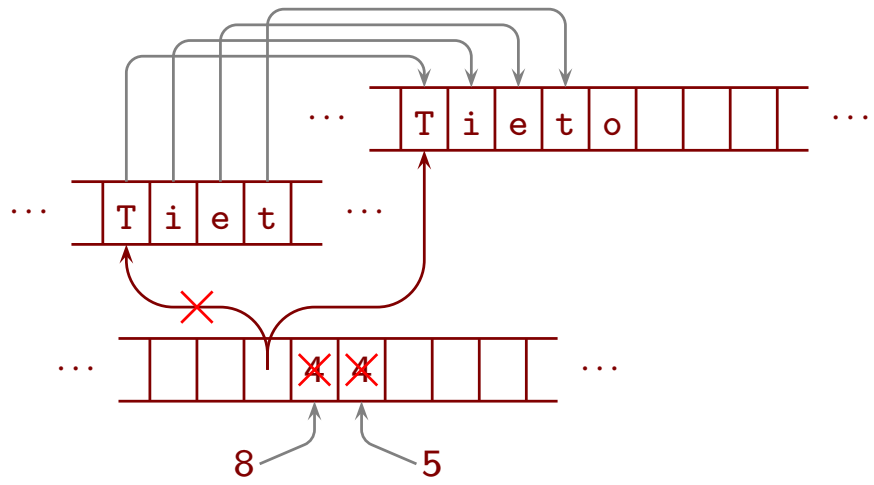
- kopioidaan ylimmän lokeron lapsi ylimmäksi (kumpi lapsi)?
- kopioidaan niin vapautuneen lokeron lapsi vapautuneeseen lokeroon (kumpi lapsi)?
- jatketaan kunnes vapaa lokero on keon viimeiselle alkiolle sopivassa paikassa
- keon viimeinen alkio laitetaan vapaaseen lokeroon
- kannattaa ajatella, että keko loppuu juuri ennen viimeistä alkiotaan
- ykstyiskohdat kotitehtävänä tai luvussa 4.3

3.3 Kasvavat taulukot ja tasattu ajan kulutus

Vanhoissa ohjelmointikielissä taulukoille varattiin tilaa samoin kuin "pienille" muuttujille

- muuttujat ovat muistissa (melkein) peräkkäin

⇒ maksimikoko piti tietää käännösaikana



Nykyisin monissa kielissä voidaan käyttää kasvavia taulukoita

- varsinaiselle sisällölle varataan tilaa sieltä mistä `new`:kin varaa
- muiden muuttujien seassa on vain osoitin varsinaiseen sisältöön, koko ja kapasiteetti
  - koon sijaan voi olla osoitin sisällön loppuun
  - kapasiteetin sijaan voi olla osoitin varatun tilan loppuun

⇒ hitaampaa koska joudutaan menemään osoittimen kautta, mutta joustavampaa

- kasvatuksen ajan kulutus
  - jos mahtuu kasvamaan varatussa muistissa, niin $\Theta(1)$
  - jos tarvitsee varata uusi, isompi alue, niin $\Theta(n)$ sisällön kopioinnin vuoksi

Miksi ei kasvateta esim. 100 alkiota kerrallaan?

- jos n on sadan monikerta, niin alkioita siirretään yhteensä $\frac{1}{200}n^2 - \frac{1}{2}n$ kertaa
  - $\Theta(n^2)$
- $\frac{1}{200}$ on pieni, mutta esim. 100 000 alkion taulukolla merkittävä
- toisaalta, monelle pienelle taulukolle 100 alkiota / taulukko tuhlaa muistia

Siksi tapana on kahdentaa kasvatuksessa (paitsi kun $n = 0$)

- silloin muistialueen kasvatuksia tapahtuu harvoin

Esimerkki

- n
- sijoitusten määrä kasvatuskerralla
- sijoitusten määrä yhteensä

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	2	3	1	5	1	1	1	9	1	1	1	1	1	1	1	17
1	3	6	7	12	13	14	15	24	25	26	27	28	29	30	31	48

- taulukkoa katsomalla on helppo muodostaa arvaus sijoitusten määrän vaihteluvälistä
  - kun n on kahden potenssi, sijoitusten määrä näyttää olevan $2n - 1$
  - kun n on kahden potenssi plus 1, se näyttää olevan $3n - 3$

On melko helppo nähdä, että $3n-3$ todella on sijoitusten määrän yläraja

- kun $n>1$, niin enintään $n-1$ alkiota siirretään koskaan muistialueesta toiseen
  - sitä ei siirretä, joka lisättiin kun viimeisen kerran kasvatettiin aluetta
- siirrettävistä täsmälleen puolet oli siirretty edelliselläkin kasvatuskerralla, paitsi $n=1$

$\Rightarrow$ enintään $(n-1)$ ainakin kerran

enintään $\frac{1}{2}(n-1)$ toisenkin kerran

enintään $(\frac{1}{2})^2(n-1)$ kolmannenkin kerran

$\ldots$ $\ldots$ $\ldots$

enintään $(\frac{1}{2})^{k-1}(n-1)$ myös k:nnen kerran, missä k on kasvatuskertojen määrä

- enintään yhteensä $\left(1+\frac{1}{2}+(\frac{1}{2})^2+\ldots+(\frac{1}{2})^{k-1}\right)(n-1)<2(n-1)$ siirtoa, koska

$$(\tfrac{1}{2})^{i+1}-(\tfrac{1}{2})^i=(\tfrac{1}{2}-1)(\tfrac{1}{2})^i=-(\tfrac{1}{2})^{i+1}$$

$$\begin{aligned}
1 &= 2-1 &&= 2-(\tfrac{1}{2})^0\\
1+\tfrac{1}{2} &= 2-(\tfrac{1}{2})^0+(\tfrac{1}{2})^1 &&= 2-(\tfrac{1}{2})^1\\
1+\tfrac{1}{2}+(\tfrac{1}{2})^2 &= 2-(\tfrac{1}{2})^1+(\tfrac{1}{2})^2 &&= 2-(\tfrac{1}{2})^2\\
1+\tfrac{1}{2}+(\tfrac{1}{2})^2+(\tfrac{1}{2})^3 &= 2-(\tfrac{1}{2})^2+(\tfrac{1}{2})^3 &&= 2-(\tfrac{1}{2})^3\\
1+\tfrac{1}{2}+(\tfrac{1}{2})^2+\ldots+(\tfrac{1}{2})^{k-1} &= 2-(\tfrac{1}{2})^{k-2}+(\tfrac{1}{2})^{k-1} &&= 2-(\tfrac{1}{2})^{k-1}
\end{aligned}$$

- lisäksi n alkuperäistä sijoitusta, joten yhteensä enintään $n+2(n-1)-1=3n-3$

Niinpä, kun aloitetaan tyhjällä taulukolla ja lisätään n alkiota, aikaa kuluu yhteensä $\Theta(n)$

- aikaa kuluu hieman myös kasvatustarpeen testaamiseen, uuden koon laskemiseen, ...
- isoilla taulukoilla sijoitukset hallitsevat työmäärää

$\Rightarrow$ aikaa kuluu noin kaksin- tai kolminkertaisesti verrattuna siihen mitä kuluisi jos tilaa olisi alun perin varattu tarpeeksi

$\Rightarrow$ ei $\Theta(n^2)$, vaan $\Theta(n)$

Siis

- ei voida luvata, että lisäys kuluttaa $O(1)$
- voidaan luvata, että n peräkkäistä lisäystä kuluttaa $O(n)$

$\Rightarrow$ lisäyksen ajan kulutus on ***tasatusti*** $O(1)$

Ajan kulutuksen ja tasatun ajan kulutuksen ero on tärkeä ymmärtää!

- jos syntyy kolari siksi, että auton jarruilla kesti 3 s reagoida, niin ei paljoa lohduta, että melkein aina muulloin jarrut reagoivat sekunnin murto-osassa
- jos jarrut reagoivat hitaasti vain hyvin harvoin, niin
  - vian etsiminen on hankalaa, koska vikaa on vaikea saada toistumaan
  - voi olla vaikea saada valmistaja myöntämään, että mitään vikaa edes on

Tilan vapauttaminen taulukon pienentyessä

- ei ole yhtä tarpeellista kuin varaaminen kasvaessa

⇒ esim. C++:n `vector` ei vapauta automaattisesti

- jos pienennys- ja kasvatusrajat olisivat samat, niin vuorotellen pienentämällä ja kasvattamalla kuluisi hyvin paljon aikaa

⇒ jos tilaa vapautetaan automaattisesti, niin pienennysrajan pitää olla riittävän kaukana kasvatusrajasta

3.4 Hieman muistinhallinnasta

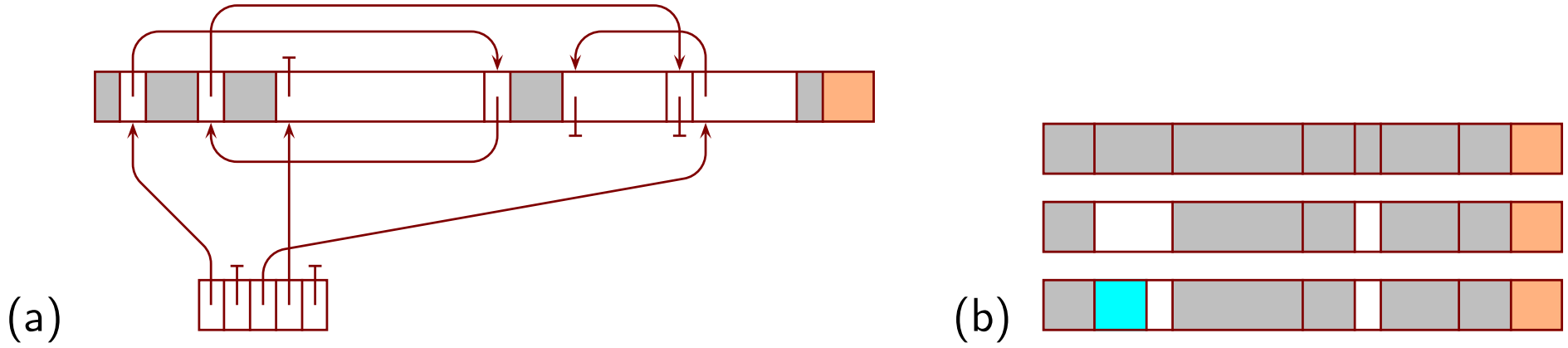
Vapautetun muistin kierrättäminen tehokkaasti on vaikeaa

- käyttöjärjestelmä (ja ohjelmointikielen suoritusympäristö) hoitavat
  - mutta eivät niin hyvin, että asian voisi kokonaan unohtaa!
- samankokoisia ***muistilohkoja*** on helppo ja tehokas kierrättää vapaiden listan avulla
- jos muistia varataan minkä kokoisina lohkoina tahansa
  - listoja saattaa tulla liian paljon
  - on epätodennäköistä, että tarkalleen sopivan kokoinen lohko olisi vapaana

  ⇒ joudutaan antamaan liian iso lohko, antamaan siitä osa tai luomaan uusi lohko

⇒ vapaat alueet saattavat ***pirstoutua*** pieniksi osiksi

⇒ lopulta ei ehkä ole isoa yhtenäistä vapaata aluetta, vaikka vapaata muistia olisi paljon



Muistin antaminen kahden potenssien kokoisina lohkoina auttaa

- vapaa muisti voidaan kierrättää tehokkaasti pienellä määrällä linkitettyjä listoja
  - yksi kullekin koolle
- eri kokoja on vähän
  $\Rightarrow$ todennäköisyys sopivan kokoisen vapaan lohkon olemassaololle kasvaa
- helpompi kehittää menetelmiä yhdistää vierekkäisiä vapaita lohkoja
  - esim. "buddy memory allocation"

Kahden potensseilla melkein puolet annetusta muistista voi jäädä käyttämättä

- se on parempi, kuin että yritetään käyttää tarkemmin muttta epäonnistutaan
- olennaista ei ole minimoida muistin käyttöä vaan voida tehdä mahdollisimman paljon muistilla joka on käytettävissä, käyttämättä kohtuuttomasti aikaa muistinhallintaan

Kotitehtävän inspiroima sivuhuomautus

- sanoja "kilo", "mega", "giga" jne. ja lyhenteitä "k", "M", "G" jne.
  käytetään sekä kahden että kymmenen potensseista

$\Rightarrow$ sekaannusta ja jopa oikeusjuttuja

- laajalti suositeltu kansainvälinen standardi määrittelee, että
  - "kilo", "mega", "giga" jne. ja "k", "M", "G" jne. vain kymmenen potensseille
  - suunnilleen samansuuruisista kahden potensseista käytetään
    "kibi", "mebi", "gibi" jne. ja "Ki", "Mi", "Gi" jne.
- tällä kurssilla noudatetaan tätä standardia

3.5 Kekoon lisäämisen ja poistamisen ajan kulutus

Rivillä 2 kuluu aikaa ***tasatusti*** $\Theta(1)$

- toisinaan $\Theta(n)$
- tyypillisesti $\Theta(1)$
- koko algoritmin ajan kulutukseen $\Theta(1)$:n mukaan

Kukin muu toiminto kuluttaa $\Theta(1)$ aikaa

LisääKekoon($\&A, uusi$)

1 $i := A.koko;\ j := \lfloor \frac{i-1}{2} \rfloor$
2 $A.kooksi(i+1)$
3 **while** $i > 0$ && $A[j].x < uusi.x$ **do**
4 $\quad A[i] := A[j];\ i := j;\ j := \lfloor \frac{i-1}{2} \rfloor$
5 $A[i] := uusi$

Kuinka monta kertaa enintään silmukka voi kiertää?

- rivin 2 lopussa $i = n-1$
- $\lfloor \frac{i-1}{2} \rfloor < \frac{i}{2}$ kun $i > 0$

$\Rightarrow$ enintään niiden kertojen määrä, jotka n täytyy puolittaa, jotta tulos olisi alle 1
  - $\log_2 n + 1$, jos n on kahden potenssi, esim. $16 \rightsquigarrow 8 \rightsquigarrow 4 \rightsquigarrow 2 \rightsquigarrow 1 \rightsquigarrow \frac{1}{2}$
  - $\lfloor \log_2 n \rfloor + 1$, jos n ei ole kahden potenssi, esim. $15 \rightsquigarrow$ vajaa $8 \rightsquigarrow \ldots \rightsquigarrow$ vajaa 1

  $\Rightarrow$ aina enintään $1 + \log_2 n$ (paitsi kun $n = 0$)

Siksi LisääKekoon:n suoritusaika on tasatusti $O(\log_2 n)$ eli tasatusti $O(\log n)$

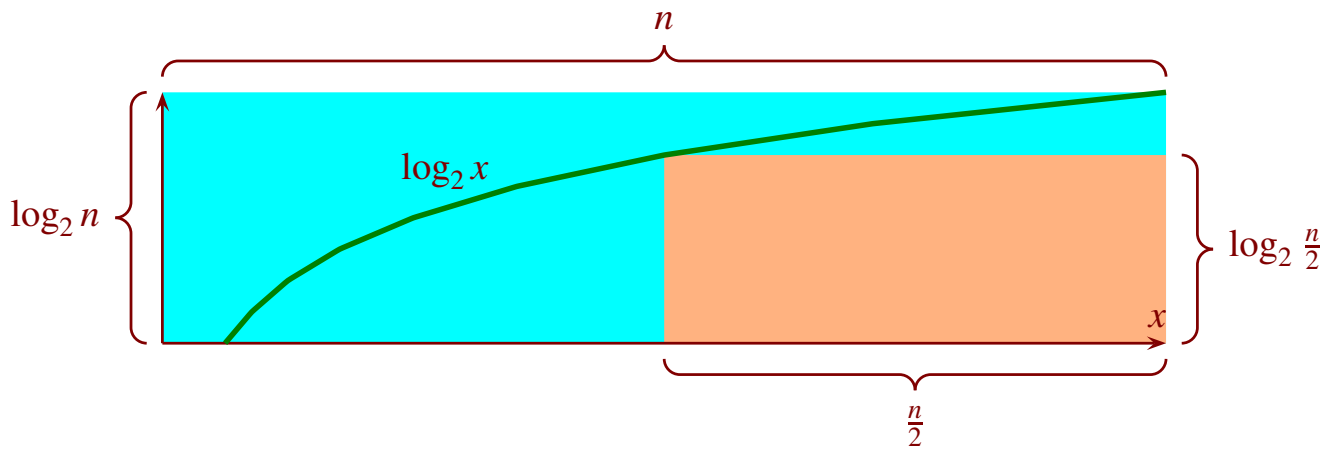
- $O(\log_2 n)$ tarkoittaa samaa kuin $O(\log n)$, koska $\log_2 n = \frac{1}{\log 2} \log n$
- tyypillisesti $O(\log n)$, toisinaan $\Theta(n)$
  - jälkimmäisellä voi olla merkitystä esim. lentokonetta ohjaavassa ohjelmassa
- isomman algoritmin ajan kulutukseen saa laskea $O(\log n)$

Kauanko kestää aloittaa tyhjällä keolla ja lisätä n alkiota?

- selvästi $O(n \log n)$
  - viimeinen lisäys on $O(\log n)$
  - kukin muu vie hitaimmillaan enintään saman ajan kuin viimeinen hitaimmillaan
- ensimmäisiä alkioita lisättäessä keko on pieni

$\Rightarrow$ $\log_2 1 + \log_2 2 + \ldots + \log_2 n$ antaa tarkemman kuvan kuin $n \log_2 n$

- jos jokainen uusi alkio kiipeää keon ylimmäksi, niin aikaa todella kuluu niin paljon
  - kotitehtävänä on keksiä syöte, jolla niin käy
- tämä summa on vaikea laskea tarkasti!



- toisaalta kuvan (ja seuraavan ruudun) mukaan kun $n \geq 1$, pätee $\log_2 1 + \log_2 2 + \ldots + \log_2 n > \frac{n}{2} \log_2 \frac{n}{2} = \frac{1}{2} n \log_2 n - \frac{1}{2} n$

$\Rightarrow$ summa on $\Omega(n \log n)$

  - se on myös $O(n \log n)$, joten se on $\Theta(n \log n)$

Kuvan x-akselilla käytetään reaalilukuja, mutta summan termien indeksit ovat kokonaislukuja

- onko kuvaan vetoaminen oikein?
- pienet epätarkkuudet eivät haittaa, koska O-, Θ- ja Ω-merkintä abstrahoivat paljon
- asia on helppo selvittää tarkastikin:

$$\log_2 1 + \log_2 2 + \underbrace{\log_2 3 + \log_2 4 + \log_2 5} \qquad \log_2 1 + \underbrace{\log_2 2 + \log_2 3 + \log_2 4}$$

Keosta poistamisen ajan kulutus määräytyy samaan tyyliin

- jos kekotaulukolle varattua muistilohkoa ei koskaan pienennetä, niin poisto on aina $O(\log n)$ (eikä vain tasatusti $O(\log n)$)
- ainakaan C++:n `vector`:in muistilohkoa ei pienennetä automaattisesti

"Fibonaccin keko" on Θ-merkinnän tasolla edellä kuvattua kekoa nopeampi

- varsinkin jos tarvitaan toiminto yhdistämään kaksi kekoa
- Fibonaccin keot ovat kuitenkin niin monimutkaisia, että ajan vakiokerroin on suuri

$\Rightarrow$ tuskin koskaan käytännössä kilpailukykyisiä

3.6 Kekojärjestäminen

On helppo keksiä, miten keon avulla voi järjestää taulukon ajassa $O(n \log n)$

- lisätään alkiot kekoon
- siirretään toistuvasti ylin tulostaulukon loppuosaan kunnes keko on tyhjä

Tätä voi parantaa

- tulos voi sijaita samassa taulukossa kuin keko ja alkuperäiset alkiot
  - alkuosa $A[0 \ldots h]$ kekona, loppuosa $A[h+1 \ldots n-1]$ lopullisessa tilassa

  $\Rightarrow$ lisämuistin tarve vain $\Theta(1)$
- lisäysvaiheen voi korvata algoritmilla, joka kuluttaa $\Theta(n)$ aikaa

```
HEAPSORT(&A)
TEEKEKO(A)
for h := A.koko − 1 downto 1 do
    apu := A[0]
    MUUTETTUPOISKEOSTA(A, h)
    A[h] := apu
```

```
TEEKEKO(&A)
n := A.koko
for k := ⌊n/2⌋ − 1 downto 0 do
    apu := A[k]; i := k; j := 2i + 1
    while true do
        if j + 1 < n && A[j + 1].x ≥ A[j].x then
            j := j + 1
        if j ≥ n || A[j].x ≤ apu.x then break
        A[i] := A[j]; i := j; j := 2i + 1
    A[i] := apu
```

- silmukan alku `unsigned`:lla:

```
for( unsigned h = A.size(); h-- > 1; )
```

TeeKeko:n rivillä 2 $A[k+1 \ldots n-1]$:n jokainen alkio toteuttaa keon ehdon alaspäin

- riviltä 1 tultaessa minkään $A[i]$, missä $k < i < n$, alapuolella ei ole alkioita
- riveillä 3, ..., 8 valutetaan $A[k]$ kekoehdon mukaiseen paikkaan

$\Rightarrow$ silmukan lopetettua $k = -1$ ja $A[0 \ldots n-1]$ on keko

Millä perusteella luvataan $O(n)$ eikä $O(n \log n)$?

- $\approx \frac{n}{2}$ valutusta, valutus on hitaimmillaan $\Theta(\log n)$ $\Rightarrow$ hitaimmillaan $\Theta(n \log n)$?
- äskeinen päättely ei ole pätevä, koska
  - kun sanotaan, että valutus on hitaimmillaan $\Theta(\log n)$, on n valutusalueen koko
  - äskeisessä päättelyssä n oli koko ajan koko taulukon koko
- tarkempi analyysi osoittaa, että suurin osa alkioista voi valua vain vähän matkaa

Cup-turnauksen otteluiden määrässä on samankaltainen ilmiö

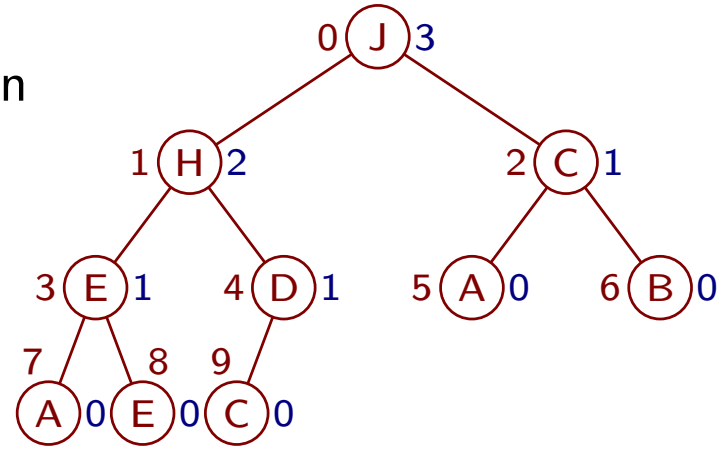
- kunkin ottelun häviäjä putoaa pois ja voittaja jatkaa, kunnes jäljellä on 4 joukkuetta
  - ei tasapelejä
  - loppujen neljän joukkueen kesken pelataan neljä ottelua
- kukin kärkijoukkue pelaa $\approx \log_2 n$ ottelua, kukin joukkue $O(\log n)$ ottelua
- joukkueiden määrän pudottamiseksi neljään tarvitaan tasan $n-4$ ottelua

$\Rightarrow$ otteluita on kaikkiaan täsmälleen n

$\Rightarrow$ joukkue pelaa keskimäärin kaksi ottelua joukkueiden määrästä riippumatta!

Solmun ***korkeus***

- mahdollisimman pitkän matkan pituus solmusta alaspäin
  - eri asia kuin syvyys ja taso
- lapsettoman solmun korkeus on 0



Keossa

- solmun i korkeus $\geq$ solmun $i+1$ korkeus
- kullakin korkeudella (paitsi 0) muilla kuin viimeisellä solmulla on kaksi lasta
  - ne ovat yhtä matalammalla kuin solmu itse
- sama pätee korkeuden viimeisen solmun vasempaan lapseen
- korkeuden viimeisen solmun oikea lapsi voi
  - puuttua (solmu itse on silloin korkeudella 1) (esim. 4 D)
  - olla yhtä matalammalla kuin solmu itse (esim. 1 H)
  - olla kahta matalammalla kuin solmu itse (esim. 0 J)

Kullakin korkeudella on sama tai sama plus 1 määrä solmuja kuin korkeammalla yhteensä

- olkoot m korkeudella k ja y sitä korkeammalla olevien solmujen määrä

$\Rightarrow$ vähintään korkeudella k on $m+y$ solmua

- niistä jokainen paitsi juuri on vähintään korkeudella $k+1$ olevan solmun lapsi

$\Rightarrow$ vähintään korkeudella k on $m+y-1$ lasta

- edeltä korkeuden muilla solmuilla ja viimeisellä solmulla ...

$\Rightarrow$ vähintään korkeudella $k+1$ olevien solmujen lapsista korkeintaan yksi puuttuu tai on matalammalla kuin korkeudella k

- $m+y-1=2y$ tai $m+y-1=2y-1$

$\Rightarrow$ $m=y+1$ tai $m=y$

Siis ainakin puolet solmuista on korkeudella 0, ainakin puolet lopuista korkeudella 1 jne.

$\Rightarrow$ korkeintaan puolet solmuista voi valua edes yhden askeleen, korkeintaan neljäsosa toisenkin askeleen, korkeintaan kahdeksasosa kolmannenkin jne.

- valumisia tapahtuu yhteensä enintään $\left(\frac{1}{2}+(\frac{1}{2})^2+\ldots+(\frac{1}{2})^k\right)n$ $< 1n = n$ askelta, missä k on juuren korkeus

$\Rightarrow$ TeeKeko toimii ajassa $O(n)$

- toisaalta jo rivillä 2 (**for**-silmukan alkurivi) kuluu aikaa $\Omega(n)$

$\Rightarrow$ TeeKeko toimii ajassa $\Theta(n)$

TeeKeko vai n kertaa LisääKekoon?

- hitaimmillaan olennainen mutta ei valtava ero
  - TeeKeko $\Theta(n)$, n kertaa LisääKekoon $\Theta(n \log n)$
- mutta koko Heapsort on silti hitaimmillaan $\Theta(n \log n)$ rivien 2, ..., 5 vuoksi ...
- ... ja tyypillinen tapaus voi olla parempi kuin hitain tapaus
  - satunnainen alkio on suunnilleen puolivälissä aikaisempien suuruusjärjestystä
  - ainakin kolme neljäsosaa alkioista on kahdella alimmalla korkeudella

  $\Rightarrow$ saattaa olla, että satunnainen alkio ei kiipeä paljoa LisääKekoon tapauksessa

Siispä mittaamaan!

- yksi taulukko kutakin ilmoitettua kokoa
- alkiossa oli joko pelkkä avain tyyppiä `int`, tai myös 400 tavua tyyppiä `char`
- avaimet oli arvottu väliltä $0, \ldots, n-1$, missä n on juuri sen taulukon koko

  $\Rightarrow$ avaimen arvo voi toistua, jolloin jotain toista arvoa ei tule lainkaan
- kukin aika on kolmen mittauksen mediaani, ja sisältää myös taulukoiden luonnit

	0...4 999		10 000...19 999		10^6...1 000 019	
TeeKeko ja suoristus	0,6 s	5,6 s	8,7 s	1 min 25 s	1,8 s	36 s
TeeKeko, ei suoristus	0,6 s	5,6 s	9,1 s	1 min 25 s	1,8 s	36 s
LisääKekoon ja suoristus	0,6 s	6,0 s	9,1 s	1 min 28 s	1,9 s	35 s
LisääKekoon, ei suoristus	0,7 s	6,0 s	9,6 s	1 min 29 s	1,9 s	35 s

Mittaustulosten luotettavuudesta

- ajoissa on mukana myös taulukon luonti ja täyttäminen
- mediaanin kolmen mittauksen kesken toinen numero vaihteli, mutta vain vähän
- mittauksia myöhemmin toistettaessa esiintyi hieman isompaa vaihtelua
- aikaisemmasta tiedetään, että esim. puolen vuoden päästä voi tulla isohkojakin eroja
  - kääntäjän ja käyttöjärjestelmän päivityksiä?
- mittaustulosten vaihtelua mittausolosuhteiden pysyessä samana kutsutaan ***kohinaksi***

Suoristus (***inline***) on aliohjelmakutsun korvaaminen aliohjelman koodilla

- C++:n `inline` pyytää mutta ei velvoita kääntäjää suoristamaan
- mittauksissa vaikutus oli hyvin pieni, paitsi pienillä alkioilla keskikokoisilla taulukoilla

TeeKeko verrattuna n kertaa LisääKekoon mittauksissa

- edellä TeeKeko oli toisinaan hieman nopeampi, useimmiten suunnillen yhtä nopea
- teoria ennustaa, että LisääKekoon on hitaimmillaan,
  kun taulukko on valmiiksi järjestyksessä

$\Rightarrow$ kokeiltiin eri järjestyksiä

0...4 999	TeeKeko				n kertaa LisääKekoon			
	suoristus		ei suoristus		suoristus		ei suoristus	
etuperin	0,4 s	5,3 s	0,3 s	5,4 s	0,5 s	7,6 s	0,5 s	7,7 s
satunnainen	0,6 s	5,6 s	0,6 s	5,6 s	0,6 s	6,0 s	0,7 s	6,0 s
takaperin	0,4 s	5,0 s	0,4 s	5,1 s	0,4 s	5,3 s	0,5 s	5,4 s

- tulokset vastaavat ennustetta
- TeeKekoon vaikuttaa jonkin verran paremmalta kuin n kertaa LisääKekoon
- suoristuksen vaikutus, jos sitä edes on, hukkuu kohinaan

Heapsort ei ole vakaa

3.7 Prioriteettijono

Prioriteettijono on tietorakenne, joka tarjoaa seuraavat palvelut

- ainakin
  - alkion lisääminen
  - jonkin korkeimman prioriteetin alkion lukeminen ja poistaminen
- ehkä myös yksi tai useampi seuraavista
  - jonkin korkeimman prioriteetin alkion lukeminen poistamatta sitä
  - alkion prioriteetin kasvattaminen, kun alkio on prioriteettijonossa
  - alkion prioriteetin pienentäminen, kun alkio on prioriteettijonossa

Hitaita toteutuksia

- järjestämätön taulukko: korkeimman etsiminen $\Theta(n)$
- järjestetty taulukko: lisääminen hitaimmillaan $\Theta(n)$

$\Rightarrow$ prioriteettijonon käsittelyn suoritusajaksi saattaa tulla yhteensä $\Theta(n^2)$

Jos alkion prioriteettia ei tarvitse voida muuttaa alkion ollessa prioriteettijonossa, niin LisääKekoon ja PoisKeosta ovat hyvä toteutus

- lisääminen tasatusti $O(\log n)$
- poistaminen $O(\log n)$
- lukeminen poistamatta $\Theta(1)$

Jos alkion prioriteettia tarvitsee voida korottaa mutta ei alentaa ja alkio tarvitsee käsitellä vain kerran, niin alkio voidaan lisätä kekoon uudelleen aina kun sen prioriteetti nousee

- sama alkio voi olla keossa monena kappaleena
- keinoja tunnistaa keosta tulevan alkio vääräksi kappaleeksi
  - alkio on jo käsitelty
  - alkion prioriteetti on tallennettu myös keon ulkopuolelle ja on siellä erisuuri
- koska keon toiminnot ovat (tasatusti) $O(\log n)$, se hidastaa vain vähän
  - $\log n^2 = 2 \log n$

  $\Rightarrow$ jos kukin alkio on keossa n kappaleena, niin aika vain suunnilleen kaksinkertaistuu
- esim. reitin etsintää käsittelevä luku

Jos prioriteettia tarvitsee voida myös alentaa, niin kumpikaan keino ei riitä yksinään

- kotitehtävinä

Jos alkio tarvitsee voida käsitellä monesti, niin väärien kappaleiden hylkääminen menee vielä vaikeammaksi

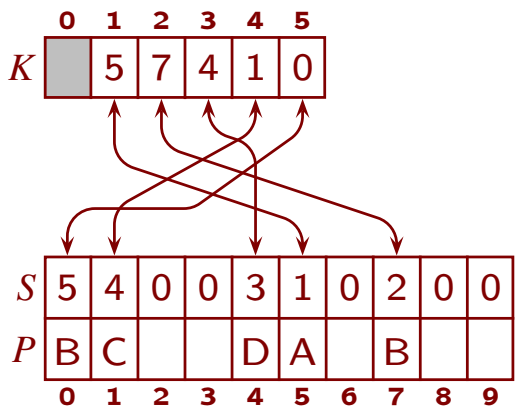
- esim. liian kauan kestänyt työ keskeytetään ja laitetaan kekoon uudelleen

$\Rightarrow$ näytämme, miten keossa olevan alkion prioriteetin voi muuttaa tehokkaasti

- samankaltaisia keinoja käytetään
  - graafialgoritmeissa ja äärellisten automaattien minimoinnissa
  - nopeuttamaan isoista alkioista muodostuvien taulukoiden järjestämistä luvussa 5.3

Tietorakenne

- jokaisella alkiolla on numero nollasta alkaen
  - muistia kuluu suurimpaan numeroon verrannollisesti
- keossa K on vain alkion numero
- prioriteetit ovat taulukossa P
- S kertoo alkion paikan keossa
  - K indeksoidaan ykkösestä alkaen, jotta 0 jäisi vapaaksi tarkoittamaan "ei keossa"



Algoritmit ovat seuraavassa ruudussa

- AlustaPjono luo tyhjän prioriteettijonon
  - kekotaulukkoon yksi, käyttämättömäksi jäävä alkio
- PoisPjonosta(k) poistaa alkion numero k
  - toiminta kerrotaan pian
- PoisPjonosta poistaa jonkin korkeimman prioriteetin alkion
  - rivillä 23 estetään tyhjästä prioriteettijonosta poistaminen
- MuutaPrioriteetti(k, p) tekee pääosan työstä
  - luottaa, että k on alkion numero
  - lisää alkion prioriteettijonoon, jollei se ole siellä jo
  - voidaan käyttää myös muuttamaan prioriteettijonossa olevan alkion prioriteettia
  - toiminta kerrotaan pian

```
MuutaPrioriteetti(k, p)
if k ≥ S.koko then
    S.kooksi(k+1, 0); P.kooksi(k+1)
if S[k] = 0 then
    i := K.koko; K.kooksi(i+1)
else
    i := S[k]
P[k] := p; j := ⌊i/2⌋
while j > 0 && P[K[j]] < p do
    K[i] := K[j]; S[K[i]] := i; i := j; j := ⌊i/2⌋
while true do
    j := 2i
    if j+1 < K.koko && P[K[j+1]] ≥ P[K[j]] then
        j := j+1
    if j ≥ K.koko || P[K[j]] ≤ p then break
    K[i] := K[j]; S[K[i]] := i; i := j
K[i] := k; S[k] := i
```

```
PoisPjonosta(k)
if k ≥ S.koko || S[k] = 0 then return
i := K.koko − 1; h := K[i]
K.kooksi(i)
S[h] := S[k]; S[k] := 0
if h ≠ k then
    MuutaPrioriteetti(h, P[h])
```

```
PoisPjonosta
if K.koko > 1 then
    PoisPjonosta(K[1])
```

```
AlustaPjono
S.kooksi(0); P.kooksi(0)
K.kooksi(1)
```

MuutaPrioriteetti(k, p):n toiminta

- jos k:ta ei ole aiemmin käytetty, niin riveillä 1 ja 2 laajennetaan S ja P kattamaan se
  - S:n uudet alkiot nollataan, P:n uusia alkioita ei tarvitse alustaa
  - k:ta ei ole rajoitettu muuten kuin että muistia kuluu siihen verrannollisesti

  $\Rightarrow$ tarpeettoman suuri k tuhlaa muistia, ja hyvin suuri voi johtaa muistin loppumiseen
- riveillä 3, . . . , 6 tallennetaan k:n paikka keossa i:hin, tarvittaessa luodaan paikka
- alkion k (uusi) prioriteetti asetetaan rivillä 7
- riveillä 7, . . . , 9 tarvittaessa siirretään k:n paikka ylemmäs keossa S ylläpitäen
- riveillä 10, . . . , 15 tarvittaessa siirretään k:n paikka alemmas keossa S ylläpitäen
- rivillä 16 alkio k laitetaan paikalleen keossa ja sen paikka merkitään S:ään

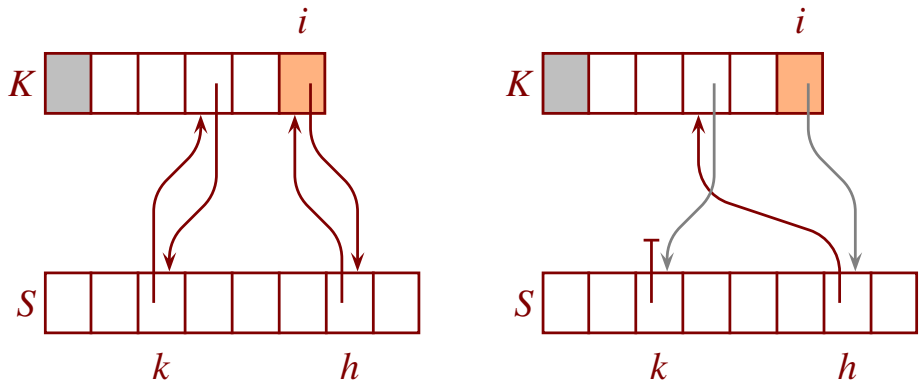
PoisPjonosta(k):n toiminta

- rivillä 17 estetään sellaisen poistaminen, joka ei ole prioriteettijonossa
- keon viimeisen alkion numero tallennetaan h:hon ja alkio poistetaan K:sta
- jos poistettava oli keon viimeinen eli $h = k$, niin asetetaan sen paikaksi 0
- muussa tapauksessa keon viimeisen paikaksi merkitään poistetun paikka, siirretään siinä oleva prioriteettinsa mukaiseen paikkaan, ja poistetun paikaksi merkitään 0

$\Rightarrow$ keon viimeinen siirtyy poistetun tilalle K:ssa ja prioriteettinsa mukaiselle paikalleen

- $K[S[h]]$ on hetken aikaa väärin, mutta se ei haittaa, koska MuutaPrioriteetti($h, P[h]$) korjaa sen ennen kuin käyttää sitä

```
17  if k ≥ S.koko || S[k] = 0 then return
18  i := K.koko − 1; h := K[i]
19  K.kooksi(i)
20  S[h] := S[k]; S[k] := 0
21  if h ≠ k then
22      MuutaPrioriteetti(h, P[h])
```



Luokkainvariantti on väittämä, joka on voimassa aina kun minkään aliohjelman suoritus ei ole kesken

1. $P.koko = S.koko$
2. Jokaisella $1 \leq i < K.koko$ pätee: $0 \leq K[i] < S.koko$ ja $S[K[i]] = i$.
   - K:n alkiot ovat laillisia indeksejä S:lle
   - keossa kohdassa i oleva alkio on S:nkin mukaan siinä
3. Jokaisella $0 \leq i < S.koko$ pätee: jos $S[i] \neq 0$, niin $1 \leq S[i] < K.koko$ ja $K[S[i]] = i$.
   - jos $S[i] \neq 0$, niin $S[i]$ on laillinen indeksi K:lle, ja keon kohdassa $S[i]$ on i
4. Jokaisella $2 \leq i < K.koko$ pätee: $P[K[\lfloor \frac{i}{2} \rfloor]] \geq P[K[i]]$.
   - keko-ominaisuus indeksoinnilla ykkösestä alkaen, ja ottaen huomioon, että K:ssa ei ole avain vaan sen sijainti P:ssä

AlustaPjono asettaa invariantin kaikki osat voimaan

Osa 1 säilyy voimassa, koska S:n ja P:n kokoja muutetaan vain samalla tavalla

Osan 4 voi tarkastaa kuten LisääKekoon ja PoisKeosta

Osat 2 ja 3 säilyvät voimassa, koska siitä huolehditaan aina kun S tai K muuttuu

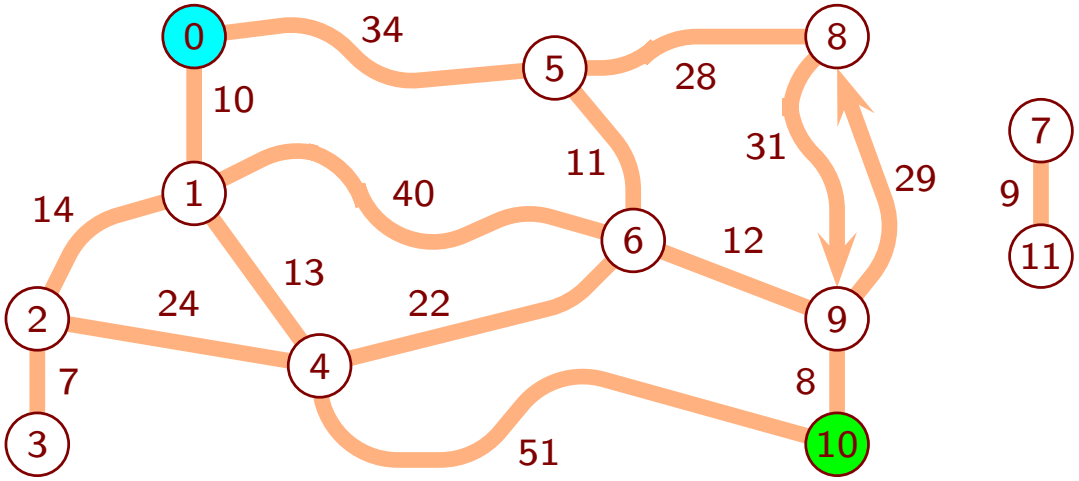
- S ei koskaan pienene
- rivillä 2 uusien alkioiden paikaksi tulee 0 eli "ei keossa", K ei muutu
- K:n kasvatusta rivillä 4 vastaavat päivitykset
  K:hon ja S:ään tehdään rivillä 9, 15 tai 16
  - sitä ennen $K[i]$:tä ei käytetä
  - pätee myös rivillä 6 löydetylle i
- rivillä 9
  - K ja S päivitetään toisiaan vastaavasti
  - i on laillinen indeksi K:lle eri syistä rivin 4, 6 tai 9 vuoksi
  - j on laillinen indeksi K:lle ehdon $j > 0$ vuoksi ja koska $j = \lfloor \frac{i}{2} \rfloor$ ja i on laillinen
  - $K[i]$ on laillinen indeksi S:lle koska $K[j]$ oli
- rivillä 15 syyt ovat samankaltaiset kuin rivillä 9
- rivillä 16
  - K ja S päivitetään toisiaan vastaavasti
  - k on laillinen indeksi rivin 2 vuoksi
  - i:lle pätee vastaava kuin riveillä 9 ja 15
- rivillä 19 i lakkaa olemasta laillinen keon indeksi, vastaavasti
  - jos $h = k$, niin $S[K[i]]$ nollaantuu rivillä 20
  - muutoin $S[k]$ nollaantuu ja $S[K[i]]$ lakkaa olemasta i rivillä 20,
    ja $K[S[K[i]]]$ korjaantuu rivillä 22

4 Sovellusesimerkki: reitin etsintä

4.1 Ohjelman alkuosat

Teemme ajokelpoisen, melkein kurssin ohjelmointiharjoitusten tasoisen ohjelman

Tiekartta



- kaksisuuntainen tienpätkä esitetään tietokoneelle kahtena yksisuuntaisena
- ympyrät ovat ***solmuja*** ja yksisuuntaiset tienpätkät ***kaaria***
- muutamia reittejä 0:sta 10:een
  - 101 km pitkä: 0 5 8 9 10
  - 74 km pitkä: 0 1 4 10
  - 65 km pitkä: 0 1 4 6 9 10
  - 65 km pitkä: 0 5 6 9 10
- ei yhtään reittiä 0:sta 7:aan
  $\Rightarrow$ etäisyys 0:sta 7:aan on (reaalilukujen) ääretön eli ∞

Otetaan käyttöön C++:n syöte- ja tulostustoiminnot sekä kasvavat taulukot

```
1  #include <iostream>
2  #include <vector>
```

Otetaan käyttöön edustaja äärettömälle

- jokaisesta solmusta pidetään kirjaa pienimmästä jo löydetyn reitin pituudesta
  - aluksi ääretön
- tietotyypin suurin mahdollinen arvo käyttäytyy vertailuissa kätevästi
  $\Rightarrow$ ei tarvitse käsitellä erikoistapauksena sitä, että vielä ei ole löydetty yhtään reittiä
- `float` ja `double` -tyypeissä on ääretön ja miinus ääretön
- desimaalipiste on tärkeä

```
3  const double aareton = 1/.0;
```

Tälle käyttökohteelle hyvä tapa esittää kaaret

- taulukko, jossa on ensin solmusta 0 lähtevät kaaret, sitten solmusta 1, ...
- `minne` sisältää kaaren kärkipään solmun numeron
- esitystapa ei sisällä lainkaan tietoa, mistä solmusta kaari alkaa!

```
4   struct kaarityyppi{ unsigned minne; double pituus; };
5   std::vector< kaarityyppi > kaaret;
```

- koko ohjelmassa on vain yksi kaaritaulukko

⇒ kätevää esittää globaalina muuttujana
  – mutta salliiko hyvä ohjelmointityyli sen?

Tälle käyttökohteelle hyvä tapa esittää solmut

- kukin solmu tietää, missä seuraavan solmun kaaret alkavat taulukossa `kaaret`
  – jatkossa kätevämpää kuin missä omat kaaret alkavat
- ohjelman toiminta ei tarvitsisi alkuarvoa `reitti`-muuttujille
  – virheiden etsimistä helpottaa alkuarvo, joka ei voi olla solmun numero

```
6   struct solmutyyppi{
7     unsigned kaarten_loppu, reitti; double etaisyys;
8     solmutyyppi( unsigned kl ):
9       kaarten_loppu( kl ), reitti( ~0u ), etaisyys( aareton ) {}
10  };
11  std::vector< solmutyyppi > solmut;
```

Muuttujat etsittävän reitin alulle ja lopulle

```
12   unsigned lahto = ~0u, maali = ~0u;
```

4.2 Syötteen lukeminen

Syötteen rakenne

- lähtösolmun numero, maalisolmun numero, kaaret
  - kaaressa on häntäpään solmun numero, kärkipään solmun numero ja pituus
- kaaret on annettava häntäpään solmun numeron mukaisessa kasvavassa järjestyksessä
- pituus ei saa olla negatiivinen eikä ääretön
- normaali käytäntö valkoisen tilan osalta
- syötevirhe ilmaistaan palauttamalla `false`

Lähdön ja maalin numeroiden lukeminen

- samalla aloitetaan kirjanpito suurimmasta solmun numerosta, joka kohdataan muuten kuin kaaren alkuna

```
13  bool lue_syote(){
14    std::cin >> lahto >> maali;
15    if( !std::cin ){
16      std::cout << "!!! Lähtö tai maali puuttuu\n"; return false;
17    }
18    unsigned solmu = 0, max_muu = lahto > maali ? lahto : maali;
```

Syötteen lukemisen pääsilmukka

- `solmu` nollattiin rivillä 18
- yritetään lukea kaaren häntäpään solmu `solmu`:un ja muut tiedot muuttujaan `kaari`
  - jollei saatu, niin rivillä 22 lopetetaan kaarten lukeminen
- riveillä 23, . . . , 26 lopetetaan, jos häntäp. solmut eivät tule kasvavassa järjestyksessä

```
19    while( true ){
20      kaarityyppi kaari; unsigned vanha = solmu;
21      std::cin >> solmu >> kaari.minne >> kaari.pituus;
22      if( !std::cin ){ break; }
23      if( solmu < vanha ){
24        std::cout << "!!! Väärä mistä-solmu " << solmu << "\n";
25        return false;
26      }
27      if( !( 0. <= kaari.pituus && kaari.pituus < aareton ) ){
28        std::cout << "!!! Laiton pituus " << kaari.pituus << "\n";
29        return false;
30      }
31      solmut.resize( solmu + 1, solmutyyppi( kaaret.size() ) );
32      kaaret.push_back( kaari ); ++solmut[ solmu ].kaarten_loppu;
33      if( max_muu < kaari.minne ){ max_muu = kaari.minne; }
34    }
```

- riveillä 27, . . . , 30 tarkastetaan, että `pituus` on laillinen
  - double sisältää arvoja ***epäluku*** (***not-a-number***)
  - ⇒ rivin 27 ehto ei ole sama kuin `0. > kaari.pituus || kaari.pituus == aareton`
- rivillä 31 luodaan tarvittaessa uusia solmutietueita
  - `solmu` $\geq$ `vanha` $\geq n-1$
  - koska `solmu` ei ole pienentynyt, ei `resize` tee mitään, jos `solmu` ei ole kasvanut
  - voidaan luoda monta, jos solmunumeroita on välistä käyttämättä
  - jokaisen luodun solmun kaarten lopuksi alustetaan senhetkinen kaarten määrä eli edellisen solmun `kaarten_loppu`
  - ⇒ eivät vielä saa yhtään kaarta
- rivillä 32 uusin kaari lisätään kaaritaulukkoon ja merkitään uusimman solmun kaariin
- rivillä 33 ylläpidetään suurinta solmun numeroa, joka on kohdattu muuten kuin kaaren alkuna

Riveillä 35, . . . , 37 luodaan tarvittaessa solmut, joita ei luotu rivillä 31

- tämä `resize` tarvitsee suojata vähentämästä solmuja
- lopuksi ilmoitetaan syötteen lukemisen onnistuneen

```
35     if( max_muu >= solmut.size() ){
36       solmut.resize( max_muu + 1, solmutyyppi( kaaret.size() ) );
37     }
38     return true;
39   }
```

Toimintojen laillisuus

- rivin 32 indeksointi on laillinen rivin 31 vuoksi
- jos syöte voi olla suuri, niin
  - muisti voi loppua riveillä 31, 32 tai 36
  - ehkä kannattaa vaihtaa `unsigned` $\rightsquigarrow$ `std::size_t` ja `~0u` $\rightsquigarrow$ `~0uz`

Ajan kulutus

- kutakin kaarta kohti menee tasatusti $\Theta(1)$ aikaa
- lisäksi kutakin solmua kohti menee tasatusti $\Theta(1)$ aikaa, vaikka solmua ei olisi mainittu syötteessä
  - jokaiselle numerolle suurimpaan syötteessä mainittuun saakka on solmu
  - sekin solmu alustetaan rivillä 31 tai 36

$\Rightarrow$ ajan kulutus on kaikkiaan $\Theta(n+m)$

4.3 Kekotoiminnot

```
40  std::vector< kaarityyppi > keko;

41  inline void lisaa_kekoon( unsigned solmu, double pituus ){
42    unsigned i = keko.size(), j = (i-1) / 2;
43    keko.resize(i+1);
44    while( i > 0 && keko[j].pituus > pituus ){
45      keko[i] = keko[j]; i = j; j = (i-1) / 2;
46    }
47    keko[i].minne = solmu; keko[i].pituus = pituus;
48  }

49  inline void poista_keosta(){
50    unsigned h = keko.size() - 1, i = 0, j = 1;
51    while( true ){
52      if( j+1 < h && keko[j+1].pituus <= keko[j].pituus ){ ++j; }
53      if( j >= h || keko[j].pituus >= keko[h].pituus ){ break; }
54      keko[i] = keko[j]; i = j; j = 2*i + 1;
55    }
56    keko[i] = keko[h]; keko.resize(h);
57  }
```

Olennaiset erot aikaisempaan

- keossa on jokin pienin ensin eli ylimmäisenä
- (keon indeksointi alkaa nollasta)
- keko on globaali muuttuja
- `lisaa_kekoon` ei saa kaaritietuetta vaan sen kummankin osan erikseen
  - kutsujan ei tarvitse rakentaa tietuetta
  - `lisaa_kekoon` ei tarvitse poimia pituutta tietueesta

Pituuksia vai ***etäisyyksiä***?

- kaarella on `pituus`
- solmulla on `etaisyys` lähdöstä mahdollisimman lyhyttä löydettyä reittiä pitkin

⇒ keon yhteydessä pitäisi puhua etäisyydestä eikä pituudesta

- se olisi vaatinut uuden tietotyypin luomista, joka eroaa vain yhden nimen osalta

⇒ kekotoimintojen `pituus` on mahdollisimman lyhyen löydetyn reitin pituus, eli se mitä muualla ohjelmassa kutsutaan etäisyydeksi

Ajan kulutus on tasatusti $O(\log \text{mikä})$?

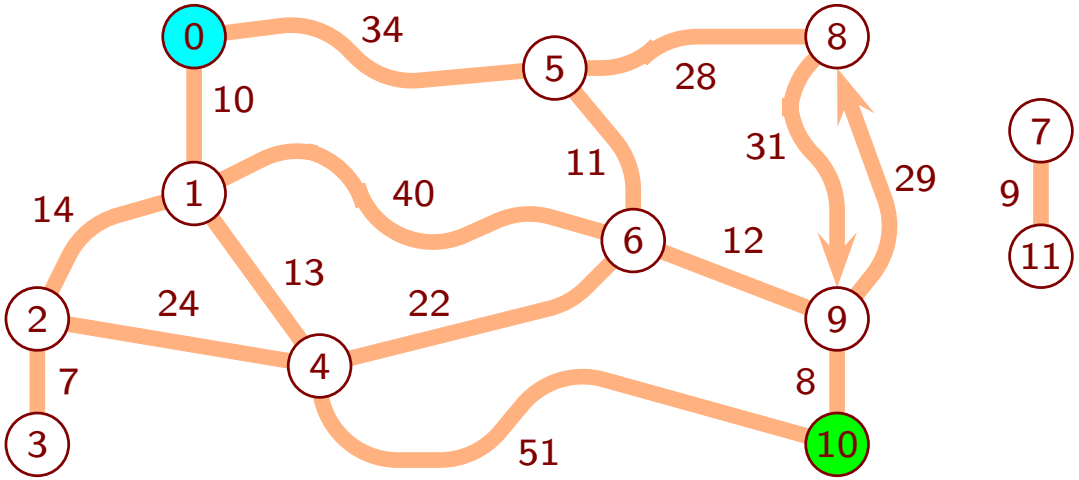
4.4 Reitin etsintä

Dijkstran algoritmi (1950-luku)

```
58  bool etsi_reitti(){
59    solmut[ lahto ].etaisyys = 0.; lisaa_kekoon( lahto, 0. );
60    while( !keko.empty() ){
61      unsigned solmu1 = keko[0].minne; double etaisyys = keko[0].pituus;
62      poista_keosta();
63      if( solmu1 == maali ){ return true; }
64      if( solmut[ solmu1 ].etaisyys < etaisyys ){ continue; }
65      unsigned i = solmu1 ? solmut[ solmu1 - 1 ].kaarten_loppu : 0;
66      for( ; i < solmut[ solmu1 ].kaarten_loppu; ++i ){
67        unsigned solmu2 = kaaret[i].minne;
68        etaisyys = kaaret[i].pituus + solmut[ solmu1 ].etaisyys;
69        if( etaisyys < solmut[ solmu2 ].etaisyys ){
70          solmut[ solmu2 ].etaisyys = etaisyys;
71          solmut[ solmu2 ].reitti = solmu1;
72          lisaa_kekoon( solmu2, etaisyys );
73        }
74      }
75    }
76    return false;
77  }
```

Toimintaperiaate

- käy läpi solmuja kasvavan ***minimietäisyyden*** mukaisessa järjestyksessä
  - mahdollisimman lyhyen reitin pituus lähdöstä, vaikka reittiä ei olisi löydetty
- lopettaa, kun käsittelyyn tulee `maali` tai käsiteltävät solmut loppuvat
- löydetyt, käsittelemättömät solmut odottavat käsittelyvuoroaan keossa
- solmu voi alun perin löytyä liian suurella etäisyydellä
  - esim. solmu 6 löytyy ensin 1:n kautta etäisyydellä 50, myöhemmin 4:n kautta 45
  - ⇒ solmu laitetaan kekoon uudelleen aina kun löytyy entistä lyhyempi reitti siihen
  - ⇒ sama solmu voi olla keossa useasti eri etäisyyksillä
  - keon alkiossa on solmun numero ja etäisyys silloin, kun solmu pantiin kekoon
  - lisäksi keon ulkopuolella solmun tiedoissa on solmun pienin tunnettu etäisyys
- keossa olevista ulos tulee ensin sellainen, jonka etäisyys on mahdollisimman pieni
  - tulemme osoittamaan, että se on ko. solmun minimietäisyys



Ohjelmakoodista

- rivillä 59 `lahto` merkitään löydetyksi etäisyydellä 0 ja laitetaan odottamaan käsittelyä
- riveillä 61 ja 62 otetaan keon ylin solmu käsiteltäväksi ja se poistetaan keosta
  - solmusta saattaa jäädä kekoon muita kappaleita, joilla on suurempi etäisyys
- jos se on `maali`, niin lopetetaan ja palautetaan `true` merkiksi, että reitti löytyi
- muutoin, jos solmuun on jo löytynyt lyhyempi reitti, niin sille ei tehdä nyt mitään
  - lyhyempää reittiä edustava keon alkio on ohittanut keossa nyt keosta tulleen

  $\Rightarrow$ solmu on jo käsitelty
- muussa tapauksessa solmu käsitellään riveillä 65, . . . , 74
  - siitä lähtevät kaaret käydään läpi
- jos kaaren kautta kulkee aikaisemmin löydettyjä lyhyempi reitti kaaren kärkipään solmuun, niin
  - kärkipään solmuun merkitään tämän reitin tiedot
  - kärkipään solmu laitetaan kekoon tämän reitin mukaisella etäisyydellä
- jos keko tyhjeni, niin palautetaan `false` merkiksi että `maali`:in ei ole reittiä

Minimipituisten reittien löytyminen voidaan osoittaa seuraavilla rivin 60 invarianteilla:

1. Jokainen keossa oleva tieto ja jokainen alustuksen jälkeen mihinkään solmuun merkitty tieto on jonkin reitin mukainen.

- rivillä 59 sen reitin, joka johtaa lähdöstä lähtöön nollaa kaarta pitkin
- riveillä 70, . . . , 72 lähdöstä jotenkin `solmu1`:een ja sieltä juuri tutkittua kaarta pitkin `solmu2`:een vievän reitin

2. Jos minkään solmun `etaisyys` on koskaan solmun minimietäisyys, niin se on siitä eteenpäin `etsi_reitti`:n loppuun saakka solmun minimietäisyys.

- etäisyyksiä ei koskaan kasvateta (rivit 59 ja 69)
- etäisyys ei voi pienentyä alle minimietäisyyden invariantti 1:n vuoksi

3. (seuraava ruutu)

Minimipituisen reitin kärki

- tarkoittakoon minimipituisen ***reitin kärki*** solmua, joka on keossa minimietäisyytensä kanssa, ja jonka jälkeen reitin minkään solmun `etaisyys` ei ole minimietäisyys
- reitin kärjen `etaisyys` on minimietäisyys (kotitehtävä)
- reitin kärki voi lakata olemasta reitin kärki kahdella tavalla:
  - se tulee pois keosta (se kappale, joka oli siellä minimietäisyytensä kanssa)
  - reitin jokin myöhempi solmu saa `etaisyys`:kseen minimietäisyyden

Invariantti 3

3. Jokaiselle solmulle pätee: solmun `etaisyys` on minimietäisyys tai solmuun johtaa minimipituinen reitti, jolla on kärki.

- jos solmuun ei ole reittiä, niin sen `etaisyys` ja minimietäisyys ovat ääretön
- muutoin solmuun on minimipituinen reitti
  - sen solmujen minimietäisyydet ovat äärellisiä, koska kaarten pituudet ovat
- rivillä 59 solmusta `lahto` tulee reitin kärki
  - reitin muiden solmujen `etaisyys` on ääretön $\neq$ minimietäisyys
- jos reitin kärki poistuu keosta minimietäisyytensä kanssa (rivi 62), niin
  - jos se on reitin määränpää, niin invariantin 3 yläosa pätee (ja säilyy inv2 vuoksi)
  - jos se on `maali`, niin riville 60 ei enää tulla, joten väitteen ei enää tarvitse päteä
  - muutoin jatketaan riville 65, koska keosta tuli minimietäisyys
  - reitin seuraavan solmun minimietäisyys lasketaan rivillä 68
  - reitin seuraavan solmun `etaisyys` ei ole minimietäisyys (kärjen määritelmä)

  $\Rightarrow$ siihen vievää kaarta ei hylätä rivillä 69

  $\Rightarrow$ siitä tulee reitin kärki rivillä 72
- jos jokin reitillä myöhäisempi solmu saa `etaisyys`:kseen minimietäisyyden, niin se lisätään kekoon rivillä 72

  $\Rightarrow$ siitä tulee reitin kärki

Osoitamme, että aina kun solmu tulee ulos keosta, on sen `etaisyys` minimietäisyys

- kutsumme tätä solmua "määränpääksi"
- jollei määränpään `etaisyys` ole minimietäisyys, niin invariantin 3 mukaan rivillä 60 määränpäähän johtaa minimipituinen reitti, jolla on kärki
- reitin kärki on keossa minimietäisyytensä kanssa
- tämä minimietäisyys on enintään määränpään minimietäisyys
  - reitin kärki on minimipituisella reitillä ennen määränpäätä
  - kaarten pituudet eivät ole negatiivisia
- määränpään `etaisyys` on suurempi, koska se ei ole minimietäisyys
- määränpään etäisyydet keossa ovat ainakin yhtäsuuret

$\Rightarrow$ reitin kärki on keosta ulostulojärjestyksessä ennen määränpäätä

$\Rightarrow$ määränpää ei voi tulla keosta ulos rivillä 62

Kun `etsi_reitti` lopettaa

- jos se lopettaa rivillä 63, niin löydetty reitti on minimipituinen
- muutoin se lopettaa rivillä 76
  - `maali` ei käynyt keossa $\Rightarrow$ `maali`:n `etaisyys` on ääretön
  - keko on tyhjä $\Rightarrow$ jokaisen solmun `etaisyys` on minimietäisyys (invariantti 3)

$\Rightarrow$ `maali`:n minimietäisyys on ääretön, eli `maali`:in ei ole reittiä

Lopettaahan `etsi_reitti` varmasti?

- koodissa ei ole muuta potentiaalisesti ikuisesti suorittavaa kuin silmukat
- rivien 66, . . . , 74 silmukka on aito **for**-silmukka
- osoitamme, että myös rivien 60, . . . , 75 **while**-silmukka lopettaa
- osoitimme, että aina kun solmu tulee keosta, on sen `etaisyys` minimietäisyys

⇒ rivi 69 estää lisäämästä solmua kekoon sen jälkeen kun se on kerran tullut sieltä

- muut keosta tulevat kopiot hylätään rivillä 64

⇒ rivi 65 suoritetaan kullekin solmulle enintään kerran

⇒ rivit 67, . . . , 73 suoritetaan kullekin kaarelle enintään kerran

- keosta voidaan poistaa enintään yhtä monta kertaa kuin siihen lisätään

⇒ rivit 61, . . . , 64 suoritetaan korkeintaan kerran
lähtösolmulle ja korkeintaan kerran kaarta kohti

⇒ silmukka lopettaa

Ajan kulutus

- n on solmujen ja m kaarten määrä
- `lisaa_kekoon` ja `poista_keosta` vievät tasatusti $O(\log k)$ aikaa
  - k on keon koko
  - $k \leq m+1$, joten ne vievät tasatusti $O(\log m)$ aikaa
- kukin muu toiminto vie $\Theta(1)$ aikaa

$\Rightarrow$ kaikkiaan $O(m \log m)$ aikaa

Laittomat toiminnot

- rivi 60 varmistaa, että rivillä 62 ei yritetä poistaa tyhjästä keosta
- muisti voi loppua rivillä 59 tai 72
- indeksointien laillisuus on sen varassa, että syöte on tallennettu oikein
  - mm. että kunkin solmun `kaarten_loppu` $\leq$ `kaaret.size()`
  - (rivin 65 indeksointi on suojattu)

Pyöristysvirheet

- liukuluvuilla laskennassa voi syntyä pyöristysvirheitä

⇒ algoritmi voi valita hieman pitemmän kuin mahdollisimman lyhyen reitin

- `float`:lla virheen suuruusluokka 1 mm / 1000 km, `double`:lla paljon pienempi
  ⇒ ei merkitystä maantiekarttasovelluksessa
  – (jossain tarkkuussovelluksessa voi olla)
- rivillä 68 tulos voi olla ääretön, vaikka oikea tulos olisi äärellinen
  – tälläkään ei ole merkitystä kuin äärimmäisissä sovelluksissa
- kokonaisluvuilla ei tapahdu pyöristysvirheitä, mutta voi tapahtua lukualueen ylivuoto
  – se voi sekoittaa algoritmin toiminnan perinpohjin

4.5 Ohjelman loppuosat

Löydetyn reitin tulostaminen

- `reitti`-muuttujat kertovat sen takaperin
- sen saa helposti tulostettua etuperin rekursiolla
- yksinkertainen ja tehokas tapa (mutta ei äärimmäisen tehokas)
- $O(n)$ aikaa ja lisämuistia
  - reitin voi selata takaperin ja samalla kääntää etuperin $\Theta(1)$ muistissa (myöhemmin kotitehtävä)

  $\Rightarrow$ tarvittaessa lisämuistin käytöksi saa $\Theta(1)$

```
78  void tulosta_reitti( unsigned solmu ){
79    if( solmu == lahto ){ std::cout << lahto; return; }
80    unsigned edell = solmut[ solmu ].reitti;
81    tulosta_reitti( edell );
82    std::cout << " "
83      << solmut[ solmu ].etaisyys - solmut[ edell ].etaisyys
84      << "km " << solmu;
85  }
```

Pääohjelma

```
86  int main(){
87    if( !lue_syote() ){ return 1; }
88    if( !etsi_reitti() ){ std::cout << "Ei reittiä\n"; return 2; }
89    std::cout << solmut[ maali ].etaisyys << "km: ";
90    tulosta_reitti( maali ); std::cout << "\n";
91  }
```

- ei oleellista lisää suoritusaika-, muisti- ja laillisuustarkasteluihin
- koko ohjelma kuluttaa $O(n+m\log m)$ aikaa ja $\Theta(n+m)$ muistia

Dijkstran algoritmi tarvitsee kartasta vain osan

- lähdöstä joka suuntaan maalin etäisyydelle saakka

$\Rightarrow$ karttaa ei tarvitse hakea tallennustilasta kerralla kokonaan, vaan tarpeen mukaan

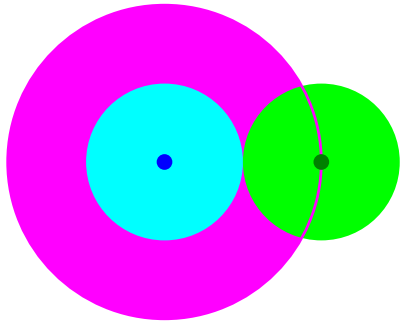
$\Rightarrow$ ajan kulutus on $O(n+m\log m)$, missä n ja m ovat ***tarvitun osan*** solmujen ja kaarten määrä

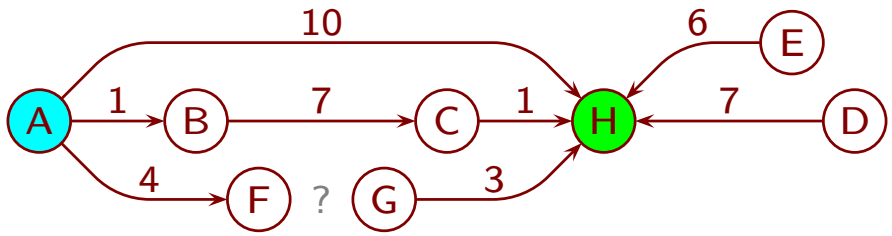
- voi olla merkittävä ajan säästö!

4.6 Etsintä yhtäaikaa kahteen suuntaan

Etsitään yhtäaikaa lähdöstä maaliin ja toisinpäin kunnes etsinnät kohtaavat

- jos risteyksiä on etsintäalueella suunnilleen tasaisesti,
  niin tutkittavien määrä suunnilleen puolittuu





Mitä tarkoittaa etsintöjen kohtaaminen?

- kuvassa lyhin reitti on A B C H, pituus 9
- kun kumpikin etsintä on käsitellyt yhden solmun
  - H on etuperin etsinnän keossa etäisyydellä 10
  - takaperin etsintä on käsitellyt H:n
  - mahdollisimman lyhyttä reittiä ei ole vielä löytynyt

⇒ ei riitä, että toinen etsintä on käsitellyt ja toinen löytänyt saman solmun

- kuvassa sama solmu on tullut ulos molemmista keoista vasta etäisyydellä 8
  - siihen mennessä myös E ja D on tutkittu

⇒ jatkaminen kunnes solmu on tullut ulos molemmista keoista teettää turhaa työtä

Paras hetki lopettaa

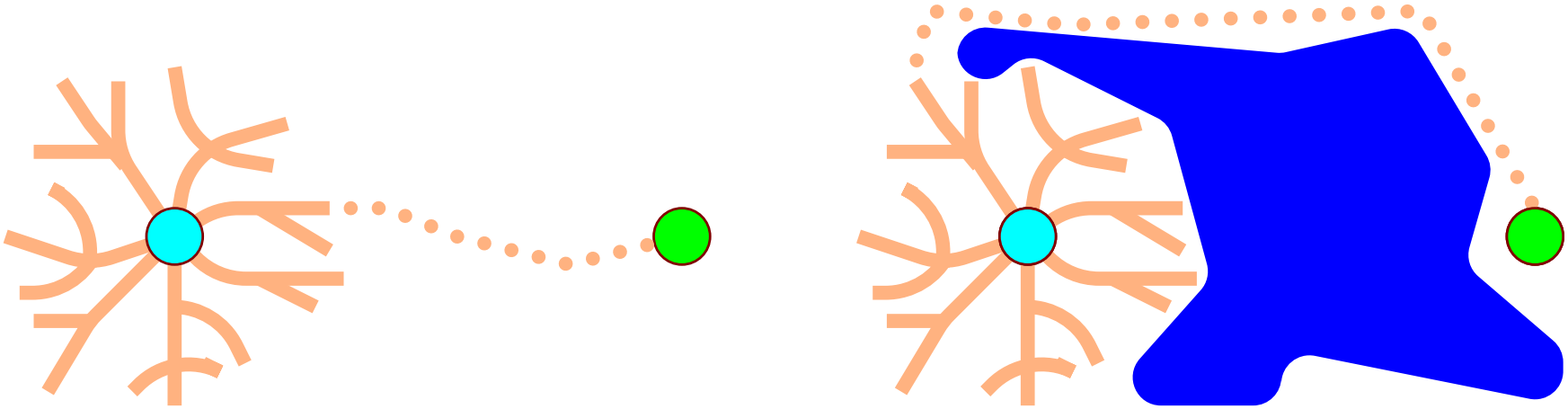
- lyhin löydetyn reitin pituus on enintään etuperin keon ylimmän etäisyys plus takaperin keon ylimmän etäisyys
- tutkimattomille kaarille pätee silloin
  - häntäpää on vähintään yhtä kaukana lähdöstä kuin etuperin keon ylimmän etäis.
  - kärkipää on vähintään yhtä kaukana maalista kuin takaperin keon ylimmän etäis.

  $\Rightarrow$ kaaren kautta ei voi kulkea lyhyempää reittiä
- ennen kuin ehto toteutuu, löydettyjä lyhyempi reitti saattaa olla olemassa
  - kun kuvassa F ja G ovat kumpikin kekonsa ylimpinä, ei vielä tiedetä, kulkeeko niiden kautta löydettyjä lyhyempi reitti
- ehto toteutuu kuvassa kun etuperin keon ylin on F,4 ja takaperin keon ylin E,6

4.7 A* reitinetsintäalgoritmi

Dijkstran algoritmi ei katso, mihin suuntiin tutkitut tienpätkät vievät

- toimii oikein, vaikka kartassa olisi tieteiselokuvien madonreikiä
- tutkii todellisten maantiekarttojen tapauksessa paljon väärään suuntaan vieviä teitä



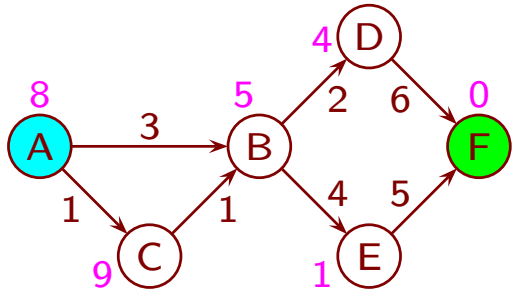
A* muistuttaa Dijkstraa, mutta ottaa huomioon myös arvion jäljellä olevasta matkasta

- tasokartassa hyvä arvio on linnuntie-etäisyys maaliin $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$
- ***kokonaismatkan arvio*** = jo löydetyn reitin pituus solmuun + loppumatkan arvio
  - aina seuraavaksi käsitellään solmu, jolla on mahdollisimman pieni kokonaismatkan arvio
  - usein ohjaa hakemaan reittiä oikeasta suunnasta, mutta ei aina

Admissible

- loppumatkan arvio on aina oikein tai alakanttiin
- takaa mahdollisimman lyhyen reitin löytymisen, jos sama solmu käsitellään tarvittaessa useasti

⇒ saattaa olla hyvin hidas



Esimerkki

- kun A on käsitelty
  - B odottaa käsittelyvuoroa kokonaismatkan arviolla $(3)+5=8$
  - C arviolla $(1)+9=10$
- kun B on käsitelty
  - D arviolla $(3+2)+4=9$
  - E arviolla $(3+4)+1=8$
  - kun ne on käsitelty, on maali löytynyt matkoilla 12 ja 11
- sitten käsitellään C

⇒ B löytyy uudelleen arviolla $(1+1)+5=7$

- se on vähemmän kuin aiemmin käytetty 8

⇒ kaikki B:stä oikealle täytyy käsitellä uudelleen

Consistent

- kaaren pituus + loppumatkan arvion muutos ≥ 0, ja loppumatkan arvio maalissa $= 0$
  - ei päde esimerkin kaarelle (C, B)
- $\Leftrightarrow$ "admissible" ja kaaren lopussa arvio on vähintään yhtä tarkka kuin kaaren alussa
- samaa solmua ei tarvitse käsitellä useasti

$\Rightarrow$ Dijkstran algoritmi kelpaa pienin muunnoksin, ajan kulutus $O(n + m \log m)$ säilyy

Miksi Dijkstran algoritmi kelpaa pienin muunnoksin?

- A* toimii ikään kuin Dijkstran algoritmilla etsittäisiin reittiä muunnetussa kartassa
  - kaarten pituudet on korvattu kokonaismatkan arvioiden muutoksilla:
    kaaren pituus + loppumatk. arvio kaaren lopussa − loppumatk. arv. kaaren alussa
- minimoi (reitin kaarten pituuksien summa − loppumatkan arvio lähdössä)
  - loppumatkan arvio lähdössä on kaikille reiteille yhtäsuuri

  $\Rightarrow$ sama kuin minimoida reitin kaarten pituuksien summa

- Dijkstran algoritmissa (muunnettu) kaaren pituus ei saa olla negatiivinen
  - tämä on sama kuin "consistent"-ehdon alkuosa

Miten Dijkstran algoritmia taritsee muuttaa?

- kaarten pituuksia ei oikeasti kannata korvata muunnetuilla pituuksilla
  - se olisi $\Theta(m)$ työ, joka pitäisi toistaa aina kun lähtö tai maali vaihtuu
  - $\Rightarrow$ menetettäisiin se nopeusetu, että jos lähtö ja maali ovat lähekkäin, niin käsitellään vain pientä osaa kartasta
- lisätään solmun tietoihin linnuntie-etäisyys maaliin
  - lasketaan se kun solmu löytyy rivillä 59 tai 70
  - kekoon pannaan sen ja lähdöstä solmuun löydetyn reitin pituuden summa eli kokonaismatkan arvio
- miksi solmussa on loppumatkan arvion lisäksi alkumatka eikä kokonaismatkan arvio?
  - pienempi riski, että pyöristysvirheet aiheuttavat ongelmia
  - (tosin tässä sovelluksessa riski lienee riittävän pieni kummallakin tavalla)

4.8 Floyd–Warshall reitinetsintäalgoritmi

Reitinetsintäalgoritmien välisiä eroja

- syötteeltä edellyttämänsä ominaisuudet
  - toteutuuko "admissible"-ehto
  - saako olla silmukoita, joilla on negatiivinen pituus (Bellman–Ford sallii)
  - saako kaarilla mutta ei silmukoilla olla negatiivisia pituuksia (Floyd–Warshall sallii)
- toimintaperiaate
- ajan kulutus Dijkstra $O(n+m\log m)$, Bellman–Ford $O(nm)$, Floyd–Warshall $\Theta(n^3)$
- lähtöjen ja maalien määrät
  - A* yhdestä yhteen
  - Dijkstra ja Bellman–Ford yhdestä kaikkiin
  - Floyd–Warshall kaikista kaikkiin

Lähtöjen tai maalien määrän kasvatus kutsumalla samaa algoritmia useasti on hidasta

- kertoo suoritusajan n:llä

Sopivin kaarten esitystapa vaihtuu kun tehtävänä on kaikista kaikkiin

- kaaria voi järkevästi olla enintään n^2 (tai $n^2 - n$)
  - jokaisesta solmusta jokaiseen (paitsi kukin kaari solmusta itseensä)
- edellä käytetty tapa on hyvä kun kaaria on paljon vähemmän kuin n^2
  - ***kytkentälista***, tarkemmin sanottuna ***forward star***
  - vie muistia $\Theta(m)$ eikä $\Theta(n^2)$
  - kaikkien kaarten selaaminen vie aikaa $\Theta(m)$ eikä $\Theta(n^2)$
  - esim. maantiekartassa $m \lesssim 4n \ll n^2$
- kaikista kaikkiin -tapauksessa vastaus sisältää n^2 alkiota

$\Rightarrow$ muistia kuluu joka tapauksessa $\Theta(n^2)$

$\Rightarrow$ kaaret kannattaa esittää ***kytkentämatriisina***

- $n \times n$ taulukko
- $M[i, j]$ sisältää kaaren pituuden solmusta i solmuun j tai äärettömän

- lopputuloksena esim. $M[i, j]$ sisältää reitin pienimmän pituuden solmusta i solmuun j

Floyd–Warshall-algoritmi

```
FloydWarshall(&M)
1  for k := 0 to n−1 do
2      for i := 0 to n−1 do
3          for j := 0 to n−1 do
4              d := M[i,k] + M[k,j]
5              if M[i,j] > d then M[i,j] := d
```

- kaarten pituudet saavat olla negatiivisia, mutta silmukoiden pituudet eivät saa
  $\Rightarrow$ solmun sisällyttäminen reittiin kahdesti ei voi lyhentää reittiä
- aluksi M sisältää kaarten pituudet
  - kukin $M[i,i]$ voidaan alustaa nollaksi tai solmusta i itseensä vievän kaaren mukaan
- kierroksen k jälkeen kukin $M[i,j]$ sisältää lyhimmän pituuden, kun matkan varrella
  - saa käyttää vain solmuja $0, \ldots, k$, kutakin enintään kerran
  - ei saa käyttää lähtöä eikä maalia
- perustelu
  - valitaan parempi reiteistä $i \rightsquigarrow k \rightsquigarrow j$ ja $i \rightsquigarrow j$,
    missä "$\rightsquigarrow$" saa käyttää vain solmuja $0, \ldots, k-1$
  - $i \rightsquigarrow x \rightsquigarrow x \rightsquigarrow j$ ei voi olla lyhyempi kuin $i \rightsquigarrow x \rightsquigarrow j$

  $\Rightarrow$ rivin 5 ehto estää kielletyn toiston tulemisen mukaan

  $\Rightarrow$ ei haittaa jos $M[i,k]$ tai $M[k,j]$ on jo päivitetty kierroksella k
- suoritusaika on $\Theta(n^3)$

Jonkin lyhimmän reitin tulostaminen

- ehdotus: $R[i,j]$ sisältää valitun k tai tiedon, että ei valittu mitään k
  - toimii, mutta tunnetaan näppärämpikin keino
- näppärämpi: $R[i,j]$ sisältää seuraavan solmun numeron reitillä i:stä k:hon
  - jos i:stä on kaari j:hin, niin alustetaan $R[i,j] := j$, ja muutoin ei tarvitse alustaa
  - kun sijoitetaan $M[i,j] := d$, niin samalla sijoitetaan $R[i,j] := R[i,k]$
- tulostaminen kotitehtävänä

Suoritusaika

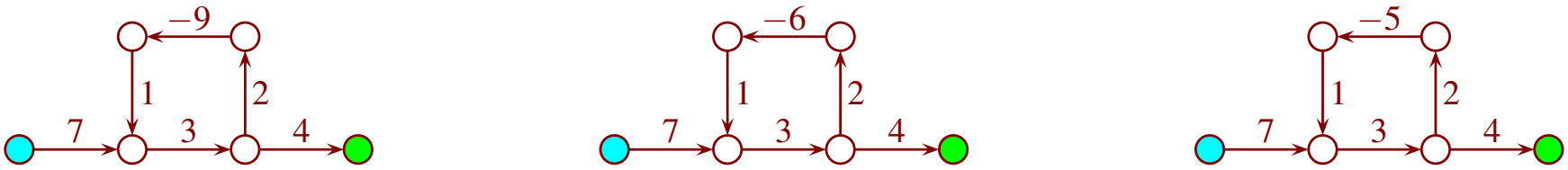
- $\Theta(n^3)$ riippumatta siitä, ovatko R ja tulostaminen mukana
  - reittejä on enintään n^2 ja kunkin tulostaminen vie aikaa $O(n)$
- tavallisesti $m \leq n^2$

$\Rightarrow$ $\Theta(n^3)$ on tavallisesti selvästi huonompi kuin Dijkstra yhdestä kaikkiin, joka on $\Theta(n + m \log m)$

- jos $m \approx n^2$, niin $\Theta(n^3)$ on hieman parempi kuin Dijkstra kaikista kaikkiin, joka on $\Theta(n^3 \log n)$

Muistin kulutus on $\Theta(n^2)$ riippumatta siitä, ovatko R ja tulostaminen mukana

4.9 Bellman–Ford reitinetsintäalgoritmi



Negatiivinen silmukka

- silmukka, jonka kaarten pituuksien summa on negatiivinen
- solmuun, johon pääsee negatiivisen silmukan kautta, ei ole lyhintä reittiä
  - kiertämällä silmukkaa tarpeeksi monesti saadaan aina vain lyhyempi reitti
  - esimerkissä reitin pituus voi olla $14, 11, 8, 5, 2, -1, -4, -7, \ldots$

Jos silmukka ei ole negatiivinen, niin sen kiertäminen ei lyhennä reittiä

- jos silmukan pituus on positiivinen, niin sen kiertäminen pidentää reittiä
  - mikään lyhin reitti ei voi kiertää sitä
- jos silmukan pituus on nolla, niin sen kiertäminen ei lyhennä eikä pidennä
  - lyhin reitti voi kiertää sen 0, 1, 2, ... kertaa

$\Rightarrow$ voidaan rajoittua reitteihin, joissa mikään solmu ei toistu
  - (niin on rajoituttu tähän astikin)
  - sellaisessa on enintään $n-1$ kaarta

Bellman–Ford-algoritmin ominaisuuksia

- jos syötteen lähdöstä saavutettavassa osassa on negatiivinen silmukka, niin palauttaa tiedon että niin on
- muutoin etsii jokaiselle solmulle jonkin lyhimmän reitin lähdöstä siihen
- ajan kulutus $\Theta(nm)$

Toimintaperiaate

- algoritmi tekee enintään n kierrosta
- jokaisella kierroksella käsitellään jokainen kaari
  - jos kaaren kautta saadaan aikaisempia lyhyempi reitti kaaren kärkipään solmuun, niin se merkitään kärkipään solmun tietoihin
  - jokainen solmun tieto on aina jonkin todellisen reitin mukainen
- tarkastellaan jotakin mahdollisimman lyhyttä toistotonta reittiä
  - sen ensimmäinen kaari käsitellään kierroksella 1 (ja muutkin kaaret)
  - sen toinen kaari käsitellään kierroksella 2 (ja muutkin kaaret)
  - sen viimeinen kaari käsitellään viimeistään kierroksella $n-1$

$\Rightarrow$ kierroksen i jälkeen on löydetty ainakin jotkin lyhimmät reitit, joissa on lähdön lisäksi enintään i solmua

- jos kierroksella minkään solmun tiedot eivät muutu, niin voidaan lopettaa
  - ne eivät muuttuisi myöhemmilläkään kierroksilla, koska niiden laskennan lähtötiedot eivät ole muuttuneet

Solmut $S[0 \ldots n-1]$

- $S[i].e$ on solmun etäisyys lähdöstä
- $S[i].r$ on edellinen solmu reitillä lähdöstä solmuun

Kaaret $K[0 \ldots m-1]$

- $K[j].h$ on sen solmun numero, josta kaari alkaa ("häntä")
- $K[j].k$ on sen solmun numero, johon kaari päättyy ("kärki")
- $K[j].p$ on pituus

```
for i := 0 to n−1 do S[i].e := ääretön
S[lähtö].e := 0
kierros := 1; muuttui := true
while kierros ≤ n && muuttui
    kierros := kierros + 1; muuttui := false
    for j := 0 to m−1 do
        d := S[K[j].h].e + K[j].p; i := K[j].k
        if d < S[i].e then
            S[i].e := d; S[i].r := K[j].h
            muuttui := true
```

Jos ja vain jos lopussa $muuttui =$ true, niin lähdöstä on reitti negatiiviseen silmukkaan

- jollei sellaista reittiä ole, niin viimeistään kierroksen $n-1$ jälkeen kunkin solmun e- ja r-kentässä on jonkin lyhimmän reitin tiedot
  $\Rightarrow$ lopussa $muuttui =$ false
- osoitamme seuraavaksi, että muussa tapauksessa lopussa $muuttui =$ true
- olkoot silmukan solmut s_0, s_1, ..., s_ℓ ja kaarten pituudet $p_{0,1}$, $p_{1,2}$, ..., $p_{\ell-1,\ell}$
  - s_ℓ on sama solmu kuin s_0
- jos rivin 8 ehto ei toteudu silmukan millekään kaarelle, niin
  - $S[s_0].e + p_{0,1} \geq S[s_1].e$
  - $S[s_0].e + p_{0,1} + p_{1,2} \geq S[s_1].e + p_{1,2} \geq S[s_2].e$
  - ...
  - $S[s_0].e + p_{0,1} + \ldots + p_{\ell-1,\ell} \geq S[s_\ell].e$
- jos lähdöstä on reitti s_0:aan, niin $S[s_0].e$ on äärellinen, joten $p_{0,1} + \ldots + p_{\ell-1,\ell} \geq 0$

Rivi 7 täytyy toteuttaa siten, että jos $S[K[j].h].e$ on ääretön, niin myös d on ääretön!

- `float` ja `double`: toteutuu automaattisesti
- `int` ja `unsigned`: huolehdittava erikseen

Jälkimmäisessä tapauksessa lisätyöllä voidaan selvittää, mitkä $.e$ ja $.r$ ovat pätevät

- etsitään kaaret, joille rivin 8 ehto toteutuu
  - kuten riveillä 6, . . . , 9
  - algoritmin kierros n voidaan korvata tällä
- tehdään graafihaku käyttäen näiden kaarten kärkipäitä lähtökohtina
  - graafihakuja on selostettu luvussa 11
- myös Dijkstran algoritmia voidaan käyttää käyttämällä kaarten pituuksina nollaa
  - rivi 63 pois
  - Dijkstran algoritmi on siihen tarpeettoman hidas ja monimutkainen

  $\Rightarrow$ keon tilalle järjestämätön taulukko jonka alkiossa on vain solmun numero, `etaisyys`:n ja `reitti`:n tilalle bitti joka kertoo onko solmu löydetty

  $\Rightarrow$ muuttuu melkein erääksi luvun 11 algoritmiksi, nopeus $O(n+m)$

4.10 Bellman–Fordin nopeuttaminen rengaspuskurilla

Kaaren käsittelystä on hyötyä vain jos sen häntäpään solmun $.e$ on muuttunut

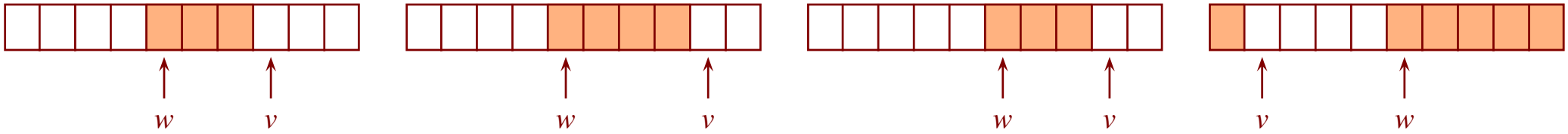
- solmut, joiden $.e$ on muuttunut kuluvalla kierroksella, taulukkoon peräkkäin
  $\Rightarrow$ ne ja niistä lähtevät kaaret voi selata tehokkaasti
  – kaaret esitetään kuten Dijkstran algoritmissa
- samaa solmua ei saa käsitellä samalla kierroksella uudelleen

$\Rightarrow$ kuluvalla ja seuraavalla kierroksella käsiteltävien solmujen luettelot pidettävä erillään

Jono on tietorakenne, joka tarjoaa seuraavat palvelut:

- alkion lisääminen jonoon
- eniten aikaa jonossa olleen alkion katsominen ja poistaminen

Rengaspuskurina toteutettu jono



- rengaspuskuri on tyhjä jos ja vain jos $v = w$
- jotta $v = w$ ei olisi totta myös kun rengaspuskuri on täysi, varataan taulukolle yksi lokero enemmän kuin siinä on enintään yhtäaikaa alkioita
- lisäyksen ja poiston koodit kotitehtävänä

Bellman–Ford-sovelluksessa

- solmun uudesta käsittelystä on hyötyä vain jos sen $.e$ on muuttunut edellisen jälkeen

$\Rightarrow$ samaa alkiota ei kannata laittaa jonoon useasti
  $\Rightarrow$ jonossa on yhtäaikaa enintään n alkiota
  $\Rightarrow$ rengaspuskuritaulukon kooksi $n+1$

- lisätään solmun tietoihin bitti, joka kertoo onko se jonossa
- kierroksen valmistuminen voidaan tunnistaa siitä, että vanhimman solmun paikka saavutti sen, missä seuraavan vapaan solmun paikka oli kierroksen alussa

Muutetun Bellman–Fordin koodi kotitehtävänä

5 Pikajärjestäminen ja mediaanin etsiminen

5.1 Lippmanin versio

Pikajärjestäminen lienee maailman eniten käytetty isojen taulukoiden järjestämisalgoritmi

- yleensä käytännössä kilpailijoitaan nopeampi
- ei ole vakaa
- hitaimmillaan todella hidas
  - hyvällä toteutuksella hitain tapaus saadaan lähes merkityksettömän harvinaiseksi
- vaikea ohjelmoida kunnolla
  - esim. netin funktionaalisissa toteutuksissa on yleensä olennaisia heikkouksia
- muistin tarve on suurempi kuin kekojärjestämisellä
- (lomitusjärjestäminen on nousussa vakavaksi kilpailijaksi)

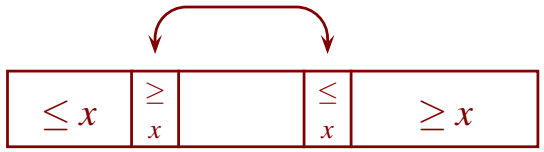
Toimintaperiaate

$\leq x$	$\geq x$

- taulukoille, joissa on enintään yksi alkio, ei tehdä mitään
  - muunnelmissa raja voi olla isompi kuin yksi
- muussa tapauksessa taulukko ***ositetaan*** alku- ja loppuosaan (ja keskiosaan)
  - vaihdetaan alkioiden paikkoja siten, että jokainen alkuosan alkio on enintään yhtäsuuri kuin mikään loppuosan alkio
  - joissakin versiossa syntyy tai voi syntyä myös keskiosa, jossa on yksi alkio
- järjestetään alku- ja loppuosa pikajärjestämisellä
  - kaksi rekursiivista kutsua, tai silmukka ja rekursiivinen kutsu

Osittaminen

- valitaan jokin osataulukon alkio ***jakoalkioksi***
- selataan osataulukkoa alusta alkaen kunnes löytyy alkio, joka on liian suuri alkuosaan
  - liian suuria ovat ainakin jakoalkiota suuremmat
  - tilanteesta riippuen myös jakoalkion kanssa yhtäsuuri voi olla liian suuri
- selataan lopusta alkaen kunnes löytyy alkio, joka on liian pieni loppuosaan
  - nytkin jakoalkion suuruinen on tai ei ole liian pieni
- vaihdetaan löydetyt alkiot keskenään
- toistetaan kunnes selaukset kohtaavat
- on olemassa muunnelma, jossa taulukkoa selataan pelkästään alusta alkaen



Lippman, C++ Primer, s. 128 (1989)

- ladontaa muutettu, jotta mahtuu valkokankaalle
  - muuten suora kopio kirjasta
- käyttää C:stä peräisin olevia taulukoita eikä `vector`:eita
- järjestää `ia`[`low...high`]
- `swap(ia,`x`,`y`)` vaihtaa keskenään lokeroiden x ja y sisällöt
  - (`static` rajaa näkyvyyttä)
- jakoalkioksi `elem` osataulukon ensimmäinen alkio
  - selaus alkaa sen perästä
- osituksen jälkeen pienet kohdissa: `low...hi` -1, jakoalkio: `hi`, suuret: `hi` $+1$`...high`

Lippman, C++ Primer, s. 128 (1989)

```
1   static void swap( int *ia, int i, int j )
2     { int tmp = ia[i]; ia[i] = ia[j]; ia[j] = tmp; }
3   void qsort( int *ia, int low, int high ){
4     if( low < high ){
5       int lo = low; int hi = high + 1; int elem = ia[ low ];
6       for (;;){
7         while ( ia[ ++lo ] <= elem ) ;
8         while ( ia[ --hi ] > elem ) ;
9         if( lo < hi ) swap( ia, lo, hi );
10          else break;
11      } // end, for(;;)
12      swap( ia, low, hi );
13      qsort( ia, low, hi - 1 ); qsort( ia, hi + 1, high );
14    } // end, if ( low < high )
15  }
```

Koeajojen tuloksia

- $0, \ldots, 199\,999$ sekalaisessa järjestyksessä: silmänräpäys
- $0, \ldots, 199\,999$ kasvavassa järjestyksessä: 6,7 s
- 200 000 kertaa sama alkio: 9,8 s
- eräs pieni taulukko: kaatui, ”Muistialueen ylitys”

Toisessa painoksessa rivi 7 oli muutettu

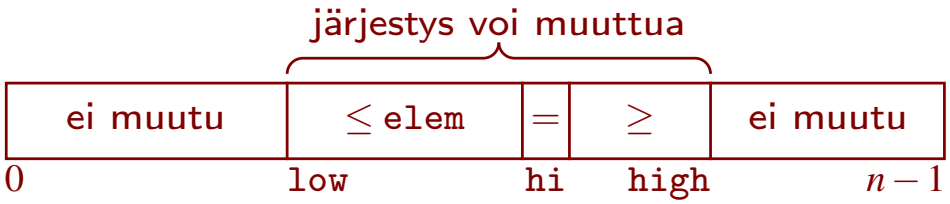
- `while ( ia[ ++lo ] < elem ) ;`
- 200 000 kertaa sama alkio: silmänräpäys
- muut viat säilyivät

Silmukoihin perustuvan ohjelman tarkastaminen

- tarkasta silmukat
  - keksi invariantti, jollei sitä ole annettu valmiiksi
  - tarkasta, että invariantti pätee, kun silmukkaan tullaan sen edeltä
  - tarkasta, että invariantti säilyy voimassa silmukan rungon suorituksessa
  - tarkasta, että invariantti ja silmukan ehdon negaatio takaavat tavoitteen
  - tarkasta, että silmukka ei kierrä ikuisesti (eikä rekursio)
- tarkasta mahdollisesti laittomat toiminnot
  - taulukoiden indeksoinnit
  - lukualueiden ylivuodot
  - ...
- vaiheet saa tehdä lomittain

Jotta `ia` olisi oikeassa järjestyksessä rivin 13 lopussa, täytyy rivin 13 alussa päteä:

1. osassa $\texttt{ia}[\texttt{low}\ldots\texttt{high}]$ on samat alkiot kuin rivin 3 lopussa
2. osissa $\texttt{ia}[0\ldots\texttt{low}-1]$ ja $\texttt{ia}[\texttt{high}+1\ldots n-1]$ on samat alkiot samassa järjestyksessä kuin rivin 3 lopussa, missä n on koko taulukon `ia` koko
3. jos $\texttt{low} \leq x < \texttt{hi}$ niin $\texttt{ia}[x] \leq \texttt{elem}$
4. $\texttt{ia}[\texttt{hi}] = \texttt{elem}$
5. jos $\texttt{hi} < x \leq \texttt{high}$ niin $\texttt{ia}[x] \geq \texttt{elem}$



Osat 1 ja 2

- `ia`:n sisältöä muutetaan vain vaihtamalla kahden alkion paikkaa `swap`:illa
  $\Rightarrow$ riittää tarkastaa, että ne käsittelevät vain osaa $\texttt{ia}[\texttt{low}\ldots\texttt{high}]$
- rivi 9: `swap( ia, lo, hi )`
  - rivit 5 ja 7: $\texttt{low} < \texttt{lo}$, rivi 9: $\texttt{lo} < \texttt{hi}$, rivit 5 ja 8: $\texttt{hi} \leq \texttt{high}$

  $\Rightarrow$ $\texttt{low} < \texttt{lo} < \texttt{hi} \leq \texttt{high}$
- rivi 12: `swap( ia, low, hi )`
  - $\texttt{ia}[\texttt{low}]$ ei muutu edellä, koska $\texttt{low} < \texttt{lo} < \texttt{hi}$

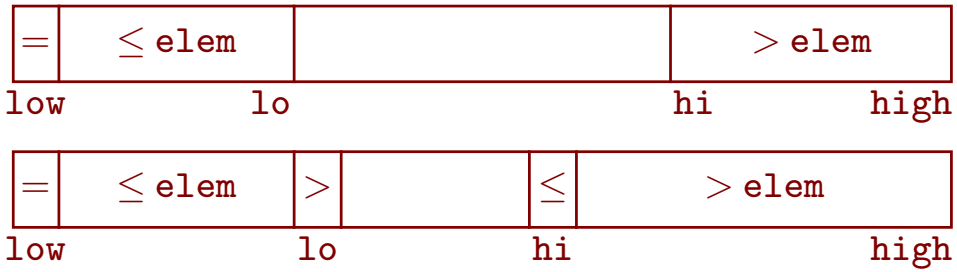
  $\Rightarrow$ riveillä 6, ..., 11 $\texttt{ia}[\texttt{low}] = \texttt{elem}$

  $\Rightarrow$ rivi 8 pysähtyy viimeistään kun $\texttt{hi} = \texttt{low}$
  - silloin $\texttt{lo} > \texttt{low} = \texttt{hi}$, joten rivit 9 ja 10 vievät pois silmukasta

  $\Rightarrow$ rivillä 12 $\texttt{low} \leq \texttt{hi} \leq \texttt{high}$

Osat 3, 4 ja 5

3. jos $\texttt{low} \leq x < \texttt{hi}$ niin $\texttt{ia}[x] \leq \texttt{elem}$
4. $\texttt{ia}[\texttt{hi}] = \texttt{elem}$
5. jos $\texttt{hi} < x \leq \texttt{high}$ niin $\texttt{ia}[x] \geq \texttt{elem}$



- apuväite (a) rivin 6 alussa:
  - jos $\texttt{low} + 1 \leq x \leq \texttt{lo}$ niin $\texttt{ia}[x] \leq \texttt{elem}$
  - jos $\texttt{hi} \leq x \leq \texttt{high}$ niin $\texttt{ia}[x] > \texttt{elem}$
- apuväite (b) rivin 9 alussa:
  - $\texttt{ia}[\texttt{lo}] > \texttt{elem}$, ja jos $\texttt{low} + 1 \leq x < \texttt{lo}$ niin $\texttt{ia}[x] \leq \texttt{elem}$
  - $\texttt{ia}[\texttt{hi}] \leq \texttt{elem}$, ja jos $\texttt{hi} < x \leq \texttt{high}$ niin $\texttt{ia}[x] > \texttt{elem}$
- rivi 5 asettaa (a):n voimaan tyhjillä väleillä
- (b) seuraa siitä että (a) päti sekä rivien 7 ja 8 silmukoista
- jos riviltä 9 palataan riville 6, niin `swap( ia, lo, hi )` muuttaa (b):n (a):ksi
- muussa tapauksesssa $\texttt{lo} \geq \texttt{hi}$, ja mennään riville 12
  - jos $\texttt{low} + 1 \leq x < \texttt{hi}$ niin $\texttt{ia}[x] \leq \texttt{elem}$, koska $\texttt{lo} \geq \texttt{hi}$
  - 5. pätee, jopa niin että $>$ eikä $\geq$
  - edelliseltä ruudulta $\texttt{low} \leq \texttt{hi} \leq \texttt{high}$ ja $\texttt{ia}[\texttt{low}] = \texttt{elem}$
- `swap( ia, low, hi )` asettaa $\texttt{ia}[\texttt{low}] \leq \texttt{elem}$ ((b)) ja $\texttt{ia}[\texttt{hi}] = \texttt{elem}$, eikä riko muuta

$\Rightarrow$ 3., 4. ja 5. pätevät

Rekursion pysähtyminen

- rivillä 12 `low` $\leq$ `hi` $\leq$ `high`

$\Rightarrow$ kumpikin rekursiivinen kutsu saa alkuperäistä pienemmän osataulukon

```
13   qsort( ia, low, hi - 1 ); qsort( ia, hi + 1, high );
```

Silmukoiden pysähtyminen

- rivien 6, . . . , 11 silmukka
  - rivillä 7 `lo` kasvaa ja rivillä 8 `hi` vähenee ainakin yhdellä

  $\Rightarrow$ lopulta rivin 9 ehto `lo` $<$ `hi` lakkaa olemasta voimassa
- näimme edellä, että rivin 8 silmukka pysähtyy viimeistään kun `hi` $=$ `low`
- mikä pysäyttää rivin 7 silmukan?

```
 6   for (;;){
 7     while ( ia[ ++lo ] <= elem ) ;
 8     while ( ia[ --hi ] > elem ) ;
 9     if( lo < hi ) swap( ia, lo, hi );
10       else break;
11   } // end, for(;;)
```

Rivin 7 silmukan käyttäytyminen

- etenee, kunnes kohtaa suuremman alkion kuin `elem`
  - `elem` saattaa olla taulukon suurin tai jaetulla ykköstilalla
  - niin on varmasti, jos taulukon kaikki alkiot ovat keskenään yhtäsuuret

⇒ saattaa edetä taulukon reunan ohi

- C++:ssa `[ ]` ei ole suojattu ohi-indeksoinnilta (`.at( )` on)

⇒ tapahtuu jokin seuraavista:

1. kohdataan muistipaikkojen ryhmä, jonka sisältö `int`:ksi tulkittuna on suurempi kuin `elem`
   - silmukka lopettaa
   - ei näytä tapahtuvan mitään outoa, ehkä aikaa kuluu enemmän
2. ohitetaan käyttäjän muistialueen loppu, jolloin käyttöjärjestelmä kaataa ohjelman
   - koeajoissa tapahtui vain kun jokin taulukon alkio on melkein suurin `int`
3. jotain muuta
   - hyvin harvinaista, mutta periaatteessa mahdollista

⇒ virhettä on vaikea havaita testaamalla!

- testiaineistossa pitää olla hyvin suuri luku

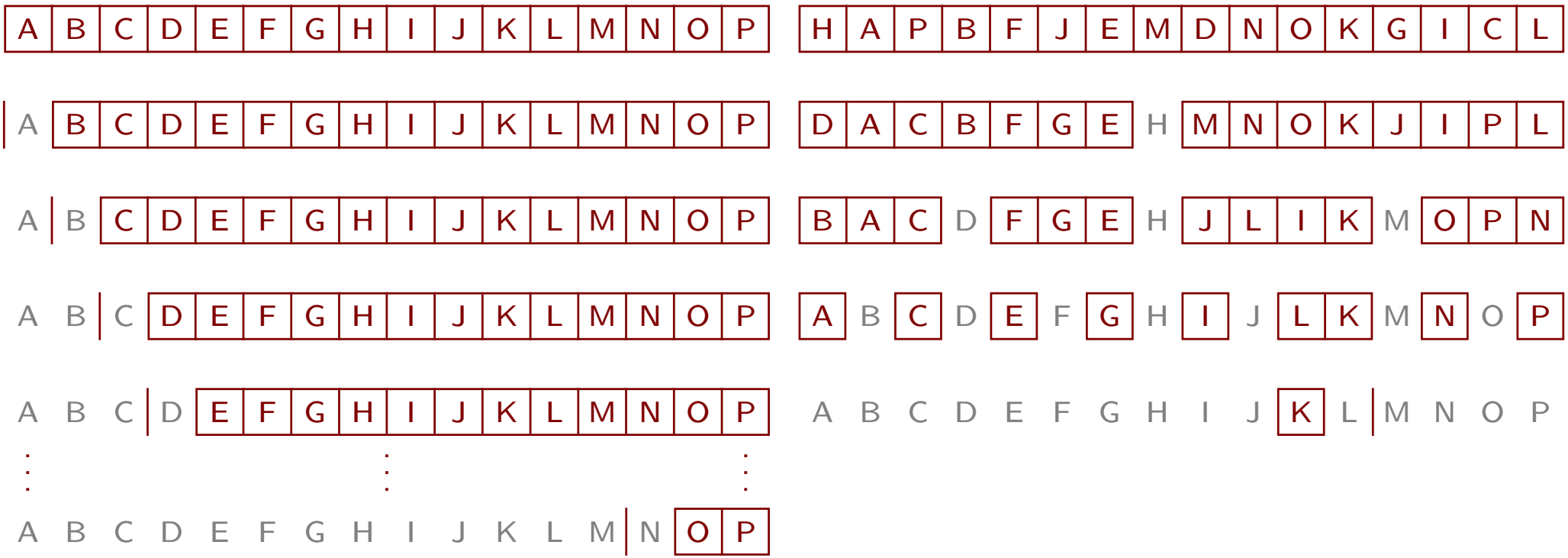
Ohi-indeksointi

- laillinen: ei osu osataulukkoon `ia`[`low`...`high`], osuu koko taulukkoon `ia`$[0 \ldots n-1]$
- laiton: osuu koko taulukon ulkopuolelle

Taulukolla $[0,1,\ldots,199\,999]$ kului paljon enemmän aikaa kuin epäjärjestyksessä olevalle

- rivi 7 pysähtyy aina heti, ohi-indeksointeja ei tapahdu eikä alkioita kopioida rivillä 9
  $\Rightarrow$ eikö suorituksen pitäisi olla nopea?
- rivi 8 lopettaa vasta kun `hi` $=$ `low`
  $\Rightarrow$ rivillä 13 aina tyhjä osataulukko `ia`$[$`low`$\ldots$`low`$-1]$
  ja yhtä vaille alkuperäinen `ia`$[$`low`$+1\ldots$`high`$]$

$\Rightarrow$ alkuperäinen ja rekursiiviset kutsut selaavat $n+(n-1)+\ldots+2 \approx \frac{1}{2}n^2$ alkiota



- epäjärjestyksellä osataulukot ovat usein melko samankokoiset
  $\Rightarrow$ selataan yhteensä keskimäärin alle $5n\log_2 n$ alkiota (työläs lasku)

Kun kaikki alkiot olivat yhtäsuuret, aikaa kului vielä enemmän

- 9,8 s vastaan 6,7 s
- nytkin ositus tuottaa tyhjä vastaan yhtä vaille kaikki
  - tällä kertaa alkuosassa on yhtä vaille kaikki
- rivin 7 silmukka jatkaa joka kerta taulukon lopun ohi
  $\Rightarrow$ paljon laillisia ja jokin määrä laittomia ohi-indeksointeja

Mitä vaikuttaa rivin 7 muutos `<=` $\rightsquigarrow$ `<`?

- kun kaikki alkiot ovat keskenään erisuuret
  - taulukon sisällä muutos ei vaikuta lainkaan
  - oikean reunan ohi menemisen jälkeen pysähtyminen voi tapahtua aiemmin
- kun kaikki alkiot ovat keskenään yhtäsuuret, myös rivi 7 pysähtyy aina heti

$\Rightarrow$ `lo` ja `hi` etenevät tasatahtia kohti toisiaan

$\Rightarrow$ ositus jakaa taulukon mahdollisimman tarkasti tasan

$\Rightarrow$ suoritusaika muuttuu silmänräpäykseksi

Lippmanin tavoite ei ollut esitellä pikajärjestäminen
vaan havainnollistaa C++:n aliohjelmia

- silti tapaus on esimerkki siitä, että pikajärjestäminen on vaikea toteuttaa kunnolla
- ja siitä, että joitakin virheitä on vaikea löytää testaamalla

5.2 Perusversio

Laadukas perusversio (ei mukana tehostuskeinoja)

- järjestää osataulukon $A[a \ldots y]$
- voi syntyä yhden alkion kokoinen keskiosa
- jakoalkion suuruiset voivat päätyä kumpaan osaan vaan (ja yksi keskelle)
  - ositus on hyvä silloinkin, kun kaikki alkiot ovat keskenään yhtäsuuret
- jakoalkio valitaan osataulukosta satunnaisesti
  - mikään järjestys ei johda systemaattisesti huonoon ositukseen

```
QUICKSORT(&A, a, y)
if a ≥ y then return
x := A[RANDOM(a, y)].x
i := a; j := y
while true do
    while A[i].x < x do i := i + 1
    while A[j].x > x do j := j − 1
    if i ≥ j then break
    SWAP(A, i, j)
    i := i + 1; j := j − 1
QUICKSORT(A, a, i − 1); QUICKSORT(A, j + 1, y)
```

Tarvitsee osoittaa seuraavat asiat:

- lopussa on samat alkiot kuin alussa
  - sen saamme hoidettua heti: ainoa tapa, jolla A:n sisältöä muutetaan, on SWAP
- rivillä 10 kunkin osan alkiot ovat enintään yhtäsuuria kuin seuraavien osien
  - jotta rivin 10 jälkeen koko taulukko olisi järjestyksessä
  - osat ovat $A[a \dots i-1]$, $A[i \dots j]$ ja $A[j+1 \dots y]$
  - kotitehtävä: miksi ei riitä sanoa "seuraavan osan"
- keskimmäisessä osassa on enintään yksi alkio, eli $i \geq j$
- osat eivät mene lomittain, eli $i-1 < j+1$ rivillä 10
  - jotta ajan käytön analyysi olisi pätevä
  - lisäksi varmistaa, että keskimmäinen osa on hyvin määritelty, eli $j \geq i-1$
- jokainen osa on pienempi kuin alkuperäinen osataulukko
  - jotta vältettäisiin ikuinen rekursio
  - $i-1 < y$ ja $j+1 > a$ ja $i \dots j \neq a \dots y$
- jokainen silmukka pysähtyy
- A:ta ei indeksoida laittomasti
- laittomia aritmeettisia ylivuotoja ei tapahdu

Rivin 3 lopussa tilanne näyttää tältä:

Rivien 4, . . . , 9 silmukalla on seuraava invariantti:

1. $a \leq i \leq y$ ja $a \leq j \leq y$ ja $i \leq j+1$.
2. Kukin osan $A[a \ldots i-1]$ alkio on enintään x ja kukin osan $A[j+1 \ldots y]$ alkio on vähintään x.
3. Osassa $A[i \ldots y]$ on ainakin yksi alkio, joka on vähintään x. Osassa $A[a \ldots j]$ on ainakin yksi alkio, joka on enintään x.

Voimaan astuminen kun tullaan rivin 4 edeltä

- 1: $i = a$ (rivi 3), $a < y$ (rivi 1) ja $j = y$ (rivi 3)
- 2: osat ovat tyhjät
- 3: rivillä 2 valittu alkio toteuttaa vaatimukset

```
1   if a ≥ y then return
2   x := A[Random(a,y)].x
3   i := a; j := y
```

Osan 1 säilyminen voimassa

- i ei voi vähentyä eikä j kasvaa, joten $a \leq i$ ja $j \leq y$ säilyvät voimassa
- jos riville 4 tullaan uudelleen, niin $i < j$ rivin 9 alussa, joten
  - $i+1 \leq j \leq y$ ja $a \leq i \leq j-1$
  - $i+1 \leq (j-1)+1$

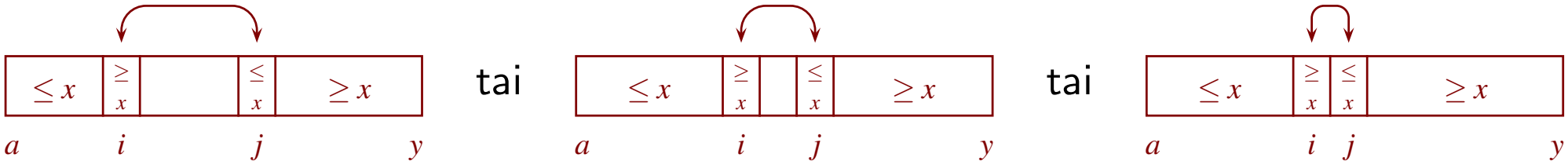
$\Rightarrow$ rivin 9 lopussa ja rivin 4 alussa $i \leq y$ ja $a \leq j$ ja $i \leq j+1$

- (viimeistään osan 3 lupaamat alkiot pysäyttävät i:n kasvun ja j:n vähenemisen, mutta emme tarvitse tätä tietoa vielä)

Osien 2 ja 3 säilyminen voimassa

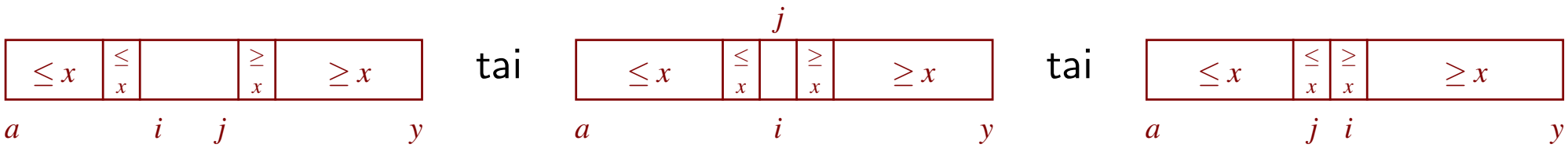
- rivit 5 ja 6
  - säilyttävät osat 2 ja 3 voimassa
  - saattavat voimaan $A[i].x \geq x$ ja $A[j].x \leq x$
- jos riville 4 tullaan uudelleen, niin rivin 7 alussa päti $i < j$
  $\Rightarrow$ tilanne näytti tältä:

```
4  while true do
5      while A[i].x < x do i := i+1
6      while A[j].x > x do j := j-1
7      if i ≥ j then break
8      SWAP(A, i, j)
9      i := i+1; j := j-1
```



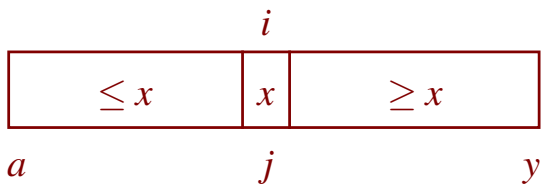
- rivin 8 SWAP asetti $A[i].x \leq x$ ja $A[j].x \geq x$

$\Rightarrow$ 2 ja 3 säilyivät voimassa rivillä 9



Tilanne rivien 4, . . . , 9 silmukan lopetettua, jos $i = j$

- $A[i].x \geq x$ ja $A[i].x \leq x$, joten $A[i].x = x$

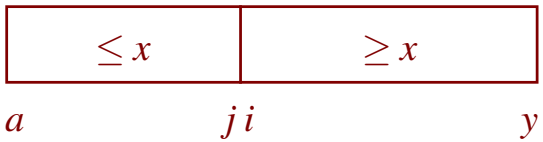


- osat eivät mene lomittain, ja alkiot ovat oikeassa järjestyksessä
- jokainen osa on alkuperäistä pienempi
  - keskimmäisessä on vain yksi alkio, vaikka sisään tuli ainakin kaksi (rivi 1)
  - muista osista puuttuu ainakin keskimmäisen alkio

Tilanne rivien 4, . . . , 9 silmukan lopetettua muussa tapauksessa

- $i \geq j$ (rivi 7) mutta ei $i = j$, joten $i > j$
- rivillä 4 päti $i \leq j+1$ eli $i \leq j$ tai $i = j+1$
- osan 3 vuoksi rivin 5 silmukka lopetti viimeistään kun $i = y$
  $\Rightarrow$ jos $i > j$ astui voimaan, niin silloin $j+1 = i \leq y$
  $\Rightarrow$ rivin 5 silmukka ei jatkanut osan 2 vuoksi
- osan 3 vuoksi rivin 6 silmukka lopetti viimeistään kun $j = a$
  $\Rightarrow$ jos $i > j$ astui voimaan, niin silloin $a \leq j = i-1$
  $\Rightarrow$ rivin 6 silmukka ei jatkanut osan 2 vuoksi

$\Rightarrow$ rivien 4, . . . , 9 silmukan lopetettua $i = j+1$ ja $i \leq y$ ja $a \leq j$



- osat eivät mene lomittain, ja alkiot ovat oikeassa järjestyksessä
- keskisosa on tyhjä
- kumpikin muu osa on alkuperäistä pienempi
  - osasta $A[a \ldots i-1]$ puuttuu ainakin $A[i]$
  - osasta $A[j+1 \ldots y]$ puuttuu ainakin $A[j]$

Silmukoiden pysähtyminen

- rivit 5 ja 6: osa 3
- rivit 4, . . . , 9: tutkittava alue kapenee muilla kuin viimeisellä kierroksella ainakin kahdella (rivi 9)

Indeksointien laillisuus

- rivi 2: $\textsc{Random}(a,y)$ tuottaa satunnaisluvun väliltä a, . . . , y
- rivit 5, 6, 8 ja 10: olemme osoittaneet, että $a \leq i \leq y$ ja $a \leq j \leq y$

Aritmeettiset ylivuodot

- rivit 5, 6 ja 9: ei vaaraa, koska aina $a \leq i \leq y$ ja $a \leq j \leq y$
- rivi 10
  - ylivuodon vaara, kun $j = y$
  - `unsigned`-tyypillä alivuodon vaara, kun $a = i = 0$

$\Rightarrow$ rivi 1 pois, rivin 10 ja alkuperäisen kutsun sijaan

if $i > a+1$ **then** $\textsc{Quicksort}(A,a,i-1)$
if $j < y-1$ **then** $\textsc{Quicksort}(A,j+1,y)$
if $a < y$ **then** $\textsc{Quicksort}(A,a,y)$

- myös voi muuttaa koko koodin siten että y on viimeinen indeksi plus 1, mutta
  - rivi 1 on muutettava monimutkaisemmaksi
  - koodin symmetria menetetään

Osituksen laatu

- jos osataulukon kaikki alkiot ovat yhtäsuuret, niin se jaetaan täsmälleen keskeltä
- jos osataulukossa on paljon sekä jakoalkion että muun suuruisia alkioita, niin osituksen lopputulos on hyvä mutta ei paras mahdollinen
  - tarkempi analyysi kotitehtävinä
- koska jakoalkio valitaan satunnaisesti, mikään syöte ei johda aina huonoon ositukseen

Paikallisuus

- pikajärjestäminen pysyttelee osataulukon alueella kunnes on saanut sen valmiiksi
  - se toimii paljolti ***paikallisesti*** eli käyttää pitkiä aikoja enimmäkseen vain muutamaa samaa pientä muistialuetta
  - vertaa kekojärjestämiseen, joka pomppii $i \rightsquigarrow 2i+1$ ja $2i+2$
- tietokoneissa on suuren muistin lisäksi pienempi nopeampi ***välimuisti***
  - tai jopa monta eri kerrosta välimuisteja
- siltä osin kuin ohjelman toiminta mahtuu välimuistiin, ohjelma nopeutuu
- jos alkioiden avaimet ovat järjestettävässä taulukossa eivätkä osoittimien takana, niin pikajärjestäminen hyötyy tästä
  - joissakin uusissa oliokielissä taulukossa eivät ole alkiot, vaan osoittimet niihin

$\Rightarrow$ C++:n `sort` perustuu pikajärjestämiseen, mutta Javan `sort` ei

Muistin tarve

- järjestettävä taulukko on rekursiivisille kutsuille yhteinen
- kukin rekursiivinen kutsu tarvitsee $\Theta(1)$ omaa muistia
- samalla rekursiotasolla olevat kutsut eivät tarvitse omaa muistia samanaikaisesti

$\Rightarrow$ lisämuistin tarve on verrannollinen rekursiotasojen määrään

$\Rightarrow$ keskimäärin $\Theta(\log n)$ ja enimmillään $\Theta(n)$

Pikajärjestäminen ei ole vakaa

Elegantti mutta muuten ei kovin laadukas pikajärjestäminen Haskellilla

- https://wiki.haskell.org/Introduction#Quicksort_in_Haskell [2.2.2024]

```
1  quicksort :: Ord a => [a] -> [a]
2  quicksort []     = []
3  quicksort (p:xs) = (quicksort lesser) ++ [p] ++ (quicksort greater)
4      where
5          lesser  = filter (< p) xs
6          greater = filter (>= p) xs
```

5.3 Tehostuskeinoja

Pienet osataulukot kannattaa järjestää InsertionSort:illa

- yksinkertaisuutensa vuoksi nopeampi, kun taulukko on tarpeeksi pieni
- pienillä alkioilla hyöty on suurempi kuin suurilla alkioilla
- voidaan toteuttaa Quicksort:ssa jättämällä pienet osataulukot järjestämättä ja ajamalla lopuksi InsertionSort
- jos rajana on k, niin
  - kukin alkio liikkuu enintään $k-1$ askelta

  $\Rightarrow$ InsertionSort:in ajan käyttö on $O(nk)$

Suurempi osataulukko kannattaa käsitellä silmukalla eikä rekursiolla

$\Rightarrow$ rekursiopinon syvyys ja muistin tarve huonoimmillaan $\Theta(\log n)$

```
Quicksort(&A, a, y)
1   while y − a ≥ 50 do
        ... edeltä rivit 2, ..., 9
10      if i − a ≤ y − j then
11          if i > a then Quicksort(A, a, i − 1)
12          a := j + 1
13      else
14          if j < y then Quicksort(A, j + 1, y)
15          y := i − 1
```

```
Quicksort(&A)
1   if A.koko < 2 then return
2   Quicksort(A, 0, A.koko − 1)
3   InsertionSort(A)
```

Suoritusaikamittauksia

- mittausolosuhteet kuten ruudulla 65, * tarkoittaa vakaata

	0...4 999		10 000...19 999		10^6...1 000 019	
InsertionSort *	6,1 s	5 min 35 s	5 min 36 s	-	48 min	-
Heapsort	0,6 s	5,5 s	8,9 s	1 min 23 s	1,8 s	35 s
rekursiivinen Quicksort	0,6 s	4,1 s	8,6 s	59 s	1,5 s	16 s
silmukka-Quicksort	0,6 s	4,1 s	8,5 s	59 s	1,5 s	16 s
MergeSort *	0,5 s	7,2 s	7,5 s	2 min 18 s	1,5 s	27 s
C++ `stable_sort` *	0,5 s	5,2 s	6,8 s	1 min 25 s	1,4 s	27 s
Quicksort-16	0,5 s	4,6 s	6,6 s	1 min 06 s	1,2 s	17 s
Quicksort-50	0,4 s	6,4 s	6,0 s	1 min 27 s	1,1 s	20 s
C++ `sort`	0,4 s	4,7 s	6,0 s	1 min 06 s	1,1 s	17 s
RadixSort *	0,1 s	4,6 s	1,6 s	1 min 16 s	0,3 s	12 s
CountingSort *	0,04 s	3,6 s	0,9 s	54 s	0,4 s	10 s
aputaul. vakautettu `sort` *	0,7 s	2,3 s	9,7 s	34 s	2,4 s	13 s
aputaul. `stable_sort` *	0,6 s	2,2 s	8,7 s	32 s	2,2 s	14 s
aputaulukko-Quick-50	0,5 s	2,1 s	7,6 s	31 s	2,1 s	13 s
aputaulukko C++ `sort`	0,5 s	2,1 s	7,7 s	30 s	2,0 s	12 s

Huomautuksia

- vakaus on lisävaatimus
  $\Rightarrow$ ei yllätä, että kaikkein nopeimmat algoritmit eivät ole vakaita
- InsertionSort on taulukossa ainoa, joka käyttää aikaa keskimäärin $\Theta(n^2)$
  – huomattavasti muita hitaampi
- erot Heapsort:in ajan kulutuksessa ruutuun 65 verrattuna johtunevat kohinasta

0,6 s	5,6 s	8,7 s	1 min 25 s	1,8 s	36 s
0,6 s	5,5 s	8,9 s	1 min 23 s	1,8 s	35 s

- silmukka-Quicksort ei eronnut mainittavasti nopeudessa rekursiivisesta
  $\Rightarrow$ satunnaisella aineistolla rekursiotasojen määrä pysynee melkein aina pienenä
- pienillä alkioilla kannatti järjestää pienet osataulukot lisäysjärjestämisellä, isoilla ei
  – isoilla alkioilla kannattaa minimoida alkioiden kopiointien määrä

$\Rightarrow$ muutos, joka nopeuttaa yhdenlaisilla taulukoilla, saattaa hidastaa toisenlaisilla
  $\Rightarrow$ on vaikeaa optimoida äärimmilleen
  – mutta ei se tyypillisesti ole tarpeenkaan

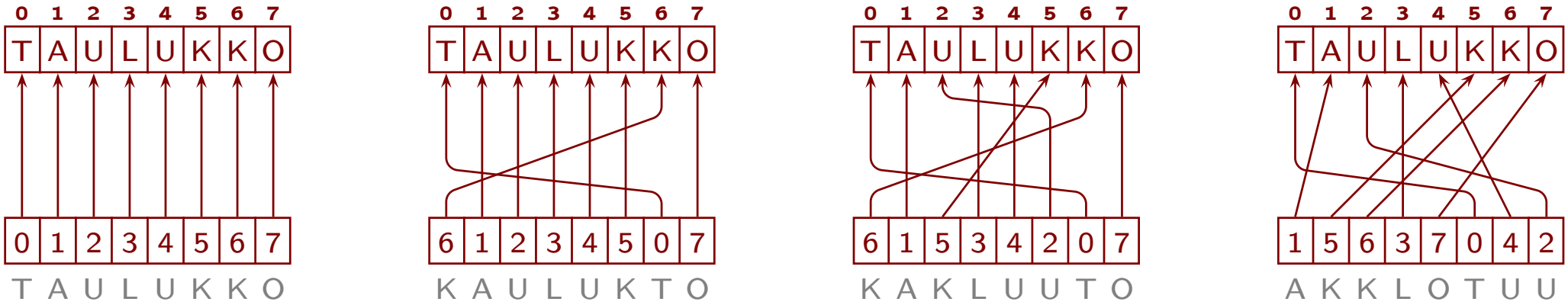
- MergeSort tulee luvussa 7.1
- RadixSort ja CountingSort olettavat avaimista paljon
  – se on otettava huomioon, jotta niiden vertaaminen muihin olisi reilua
  – luvut 7.5 ja 7.4

C++-standardi ei kerro, mitä C++:n algoritmit ovat

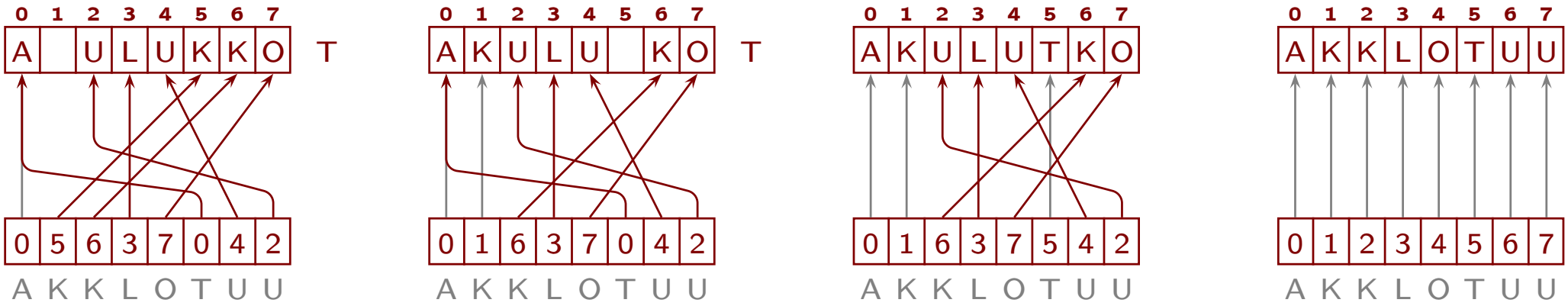
- C++ `stable_sort` perustuu melko varmasti kahteen lomitusjärjestämisen (merge sort) muunnokseen
  - ajan kulutus $\Theta(n \log n)$ jos muistia on paljon, $\Theta(n(\log n)^2)$ jos niukasti
- C++ `sort` perustuu melko varmasti pikajärjestämiseen
  - Wikipedia "Introsort" ja `g++ -E`
  - enintään 16 alkion osataulukot järjestetään lisäysjärjestämisellä
  - jos rekursiotasojen määrä ylittää suunnilleen $2\log_2 n$, siirrytään kekojärjestämiseen
- pienillä alkioilla nopeimmat yleiskäyttöiset olivat C++ `sort` ja QUICKSORT-50

Järjestäminen aputaulukko ensin, lopuksi sen avulla alkiot suoraan paikoilleen

- minimoi alkioiden kopioimisen määrän paikallisuuden
  ja vertailun nopeuden kustannuksella
  $\Rightarrow$ eduksi isoilla alkioilla
- aluksi aputaulukko täytetään luvuilla $0, 1, \ldots, n-1$
- aputaulukko järjestetään käyttäen vertailuina $A[N[i]].x < A[N[j]].x$ jne.



- nyt $A[N[0]].x \leq A[N[1]].x \leq \ldots \leq A[N[n-1]].x$
- alkiot kopioidaan lopullisille paikoilleen "kierroksittain" (yksityiskohdat kotitehtävinä)



- aputaulukon alkiot ovat pieniä verrattuna varsinaisen taulukon alkioihin
  $\Rightarrow$ lisämuistin tarve on suhteessa pieni, vaikka onkin $\Theta(n)$
- aputaulukkoa käyttävän järjestämisen saa helposti vakaaksi,
  vaikka varsinainen järjestämisalgoritmi ei olisi vakaa
  – kotitehtävä

5.4 Nopea etsiminen järjestysluvun perusteella

QUICKSELECT$(\&A, k)$

- palauttaa suuruusjärjestyksessä $(k+1)$:nnen alkion
  - jos on monta yhtäsuurta, saa palauttaa niistä minkä tahansa
- toimii muuten kuten QUICKSORT, mutta jatkaa vain sillä osataulukolla, johon etsittävä alkio osuu
  - yksityiskohdat kotitehtävänä

$\Rightarrow$ saattaa muuttaa taulukon järjestystä

- ajan kulutus on keskimäärin $\Theta(n)$
  - huonoin tapaus on $\Theta(n^2)$, mutta on hyvällä toteutuksella hyvin harvinainen
- lisämuistin tarve on $\Theta(1)$
- kotitehtävä: luonnostele algoritmi, joka hyödyntää aikaisempaa järjestämistyötään

QUICKSELECT(A, k)

- muuten kuten edellinen, mutta A välitetään kopioimalla eikä viitteellä

$\Rightarrow$ ei muuta kutsujan taulukon järjestystä

- lisämuistin tarve $\Theta(n)$

Tunnetaan monimutkainen keino saada ajan kulutukseksi hitaimmillaan $\Theta(n)$

6 Hieman linkitetyistä listoista

Yksisuuntainen linkitetty lista

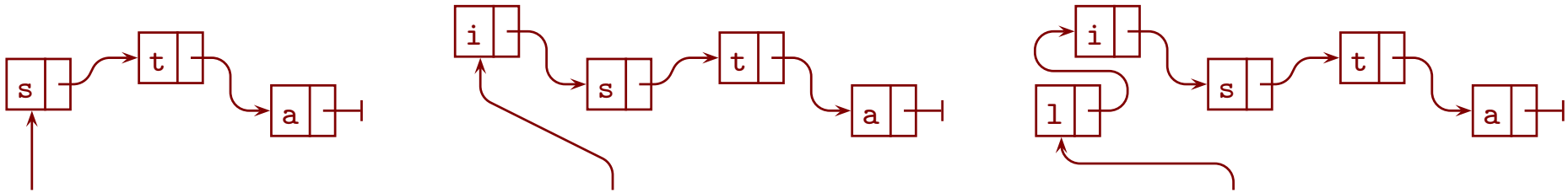
- tietorakenne, jossa jokainen alkio sisältää tiedon, missä seuraava alkio sijaitsee
- linkki voi olla osoitin, ei-negatiivinen kokonaisluku, iteraattori, ...

Kaksisuuntainen: myös tieto, missä edellinen alkio sijaitsee

Perusmuotoisen linkitetyn listan alkuun voi lisätä ja poistaa alkion tehokkaasti

$\Rightarrow$ helppo toteuttaa tehokkaasti ***pino***

- lisäys ja poistaminen vievät $\Theta(1)$ aikaa



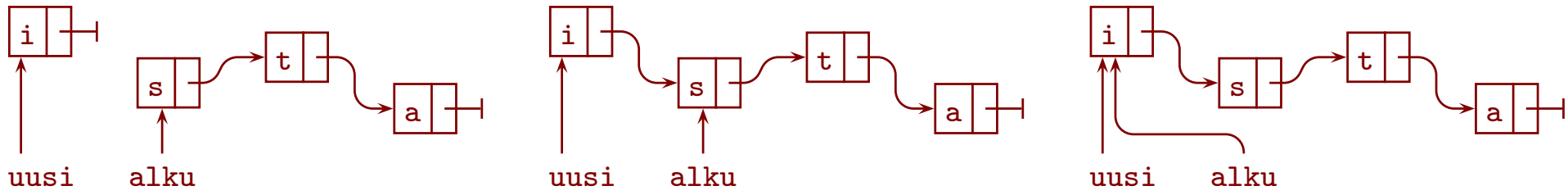
Pino listalla C++:lla

- alkion tietotyyppi

```
1   struct alkio{
2     char mrk; alkio *seur;
3     alkio( char mrk ): mrk(mrk), seur(0) {}
4   };
```

- lisääminen listan alkuun

```
5   alkio *uusi = new alkio( 'i' );
6   uusi->seur = alku; alku = uusi;
```

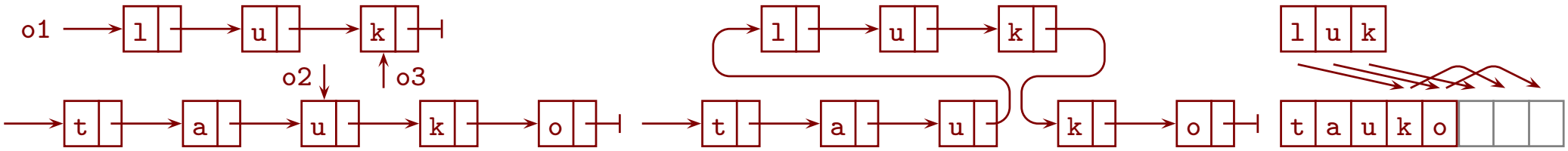


- ensimmäisen tulostaminen, poistaminen ja muistin palautus käyttöjärjestelmälle

```
if( alku ){
  std::cout << alku->mrk << "\n";
  alkio *poistettava = alku; alku = alku->seur; delete poistettava;
}else{
  std::cout << "Lista oli tyhjä.\n";
}
```

Linkitettyä listaa voi käsitellä tehokkaasti keskeltä, jos on osoittimet oikeisiin paikkoihin

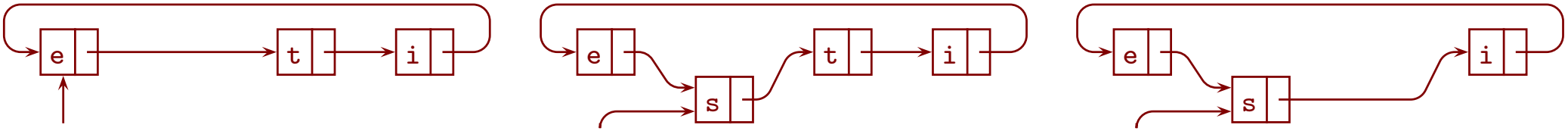
- lisääminen, jos on osoitin lisäyskohtaa edeltävään alkioon (ja lisättävien loppuun)
- taulukkoon ei voi lisätä tehokkaasti keskelle



- poistaminen, jos on osoitin poistettavaa ***edeltävään*** alkioon
  - edeltävä alkio on usein vaikea löytää tehokkaasti
  - kaksisuuntaisessa listassa riittää osoitin poistettavaan
- osalistan poistaminen tarvitsee osoittimen myös viimeiseen poistettavaan alkioon

Jonon voi toteuttaa tehokkaasti ***rengaslistalla***

- ulkopäin osoittava osoitin osoittaa viimeiseen alkioon
- esimerkissä poistuu ensin `t`, sitten `i`, sitten `e` ja sitten `s`
- lisäys ja poisto vievät $\Theta(1)$ aikaa



Rengaslistalla toteutettuun jonoon lisääminen

```
void lisaa( alkio *uusi ){
  if( !viim ){ uusi->seur = uusi; viim = uusi; }
  else{ uusi->seur = viim->seur; viim->seur = uusi; viim = uusi; }
}
```

Pakka sallii lisäämisen ja poiston molempiin päihin

- rengaslistalla voidaan toteuttaa näistä kolme ajassa $\Theta(1)$
- kaksisuuntaisella linkitetyllä listalla saadaan kaikki neljä ajassa $\Theta(1)$
- rengaspuskurilla saadaan kaikki neljä tasatussa ajassa $\Theta(1)$

Linkitetyn listan alkioihin pääsee käsiksi vain selaamalla tai tallennetulla osoittimella tms.

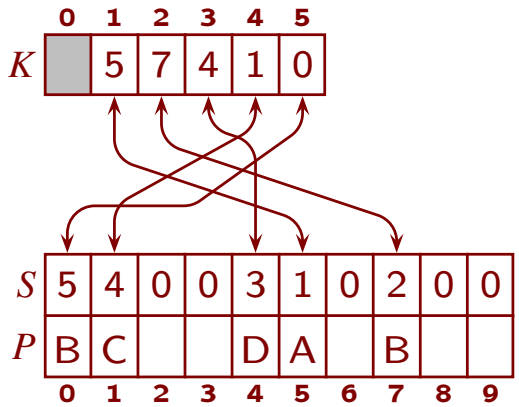
- vähentää hyötyä, joka saadaan siitä, että keskellekin voi lisätä ja poistaa ajassa $\Theta(1)$
- kekojärjestäminen olisi linkitetyillä listoilla hidas
  - vaatii kykyä siirtyä alkiosta i alkioihin $2i+1$ ja $2i+2$
- pinon ja jonon voi toteuttaa taulukoillakin

$\Rightarrow$ linkitetyille listoille on niukasti käyttöä itsenäisenä tietorakenteena

Linkeille on enemmän käyttöä isomman rakenteen osana

- binääripuut luku 9
  - voi toteuttaa monia asioita tehokkaasti
- apuvälineenä rakenteessa, joka käyttää myös taulukoita
  - tällöin voi olla luontevaa käyttää linkkeinä kokonaislukuja
  - esim. `reitti` luvussa 4
- kun linkitettyä rakennetta muutetaan, ei alkioiden sijaintia muistissa tarvitse muuttaa
- taulukoilla ei ole tätä etua

  $\Rightarrow$ luvussa 3.7 prioriteettijonossa olevan alkion prioriteetin muuttaminen oli monimutkaista



Linkitetyn listan lopun ilmoittaminen

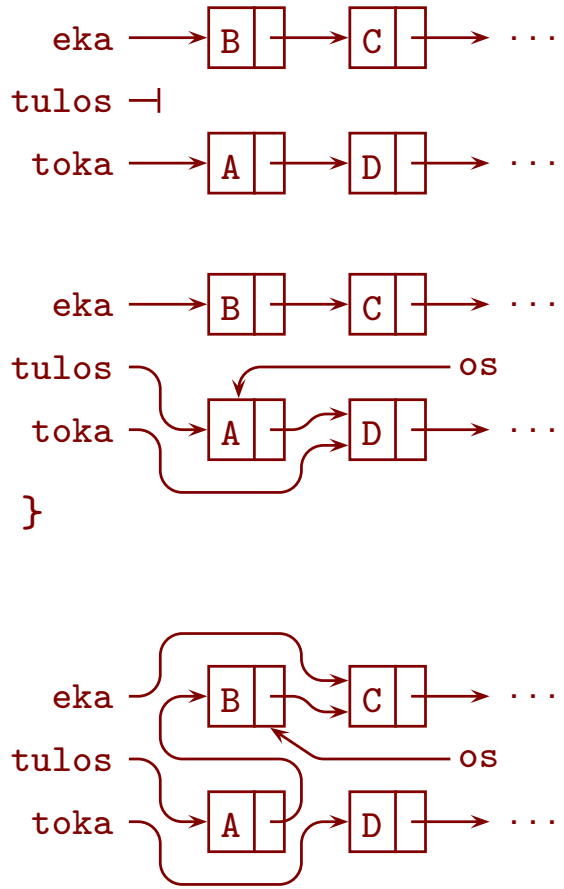
- osoitin ei minnekään
- luku 0 tai -1 tai `~0u`
- linkki alkioon itseensä

Tyhjiä rakenteita ja listojen päitä joudutaan usein käsittelemään erikoistapauksina

$\Rightarrow$ esitämme niksin, joka vähentää erikoistapausten tarvetta

Esimerkki erikoistapauksista: kahden järjestetyn linkitetyn listan yhdistäminen

```
1   alkio *lomita( alkio *eka, alkio *toka ){
2     if( !eka ){ return toka; }
3     if( !toka ){ return eka; }
4     alkio *tulos = 0;
5     if( eka->x <= toka->x ){ tulos = eka; eka = eka->seur; }
6     else{ tulos = toka; toka = toka->seur; }
7     alkio *os = tulos;
8     while( eka && toka ){
9       if( eka->x <= toka->x ){ os->seur = eka; eka = eka->seur; }
10      else{ os->seur = toka; toka = toka->seur; }
11      os = os->seur;
12    }
13    os->seur = eka ? eka : toka;
14    return tulos;
15  }
```



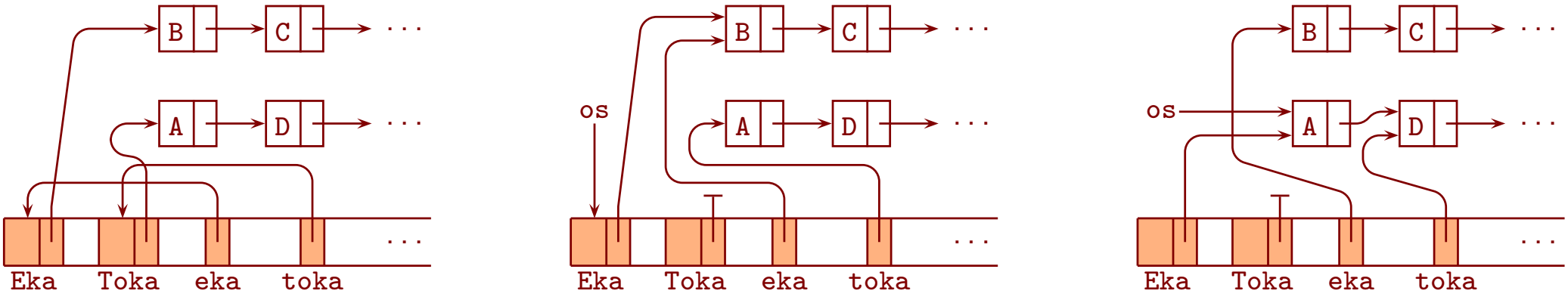
- tyhjät syötelistat käsitellään erikoistapauksina riveillä 2 ja 3
- kaikkein pienin alkio siirretään tuloslistan ensimmäiseksi riveillä 4, . . . , 6
- riveillä 7, . . . , 12 toistuvasti siirretään pienempi syötelistojen seuraavana vuorossa olevista alkioista tuloslistaan kunnes jompikumpi syötelista loppuu
- sen syötelistan, joka ei vielä loppunut, jäljellä oleva osa lisätään tuloslistaan rivillä 13
- rivit 5 ja 6 ovat melkein samanlaiset kuin 9 ja 10!

Niksi: ***päätemerkki***

- tietue, joka on muuten kuten rakenteen alkio, mutta jossa ei ole hyötykuormaa
  - hyötykuormalle varattu muisti jätetään käyttämättä (tai kikkaillaan pois)
- voi olla kiinteä muuttuja tai luotu `new`:lla
- niiden käytöllä voidaan toisinaan vähentää erikoistapausten määrää
- aliohjelmasta `lomita` saadaan 5 riviä pois:

```
1   void pm_lomita( alkio *eka, alkio *toka ){
2     alkio *os = toka; toka = toka->seur; os->seur = 0;
3     os = eka; eka = eka->seur;
      edeltä rivit 8, ..., 13
10  }
```

- päätemerkit voidaan luoda komennolla `alkio Eka('\0'), Toka('\0');`
  - hyötykuormalle varattuun muistiin menee tyhjä merkki
  - niistä tulee tavallisia muuttujia, eikä `new`:lla luotuja



- `pm_lomita( &Eka, &Toka );` lomittaa päätemerkeistä `Eka` ja `Toka` alkavat listat
  - `Eka` muuttuu yhdistetyksi listaksi
  - `Toka` muuttuu tyhjäksi listaksi
  - (`&` muodostaa osoittimen perässään olevaan alkioon)
- lisäetu: `Toka`:n kautta ei pääse sotkemaan lomitettua listaa

7 Lisää järjestämisalgoritmeista

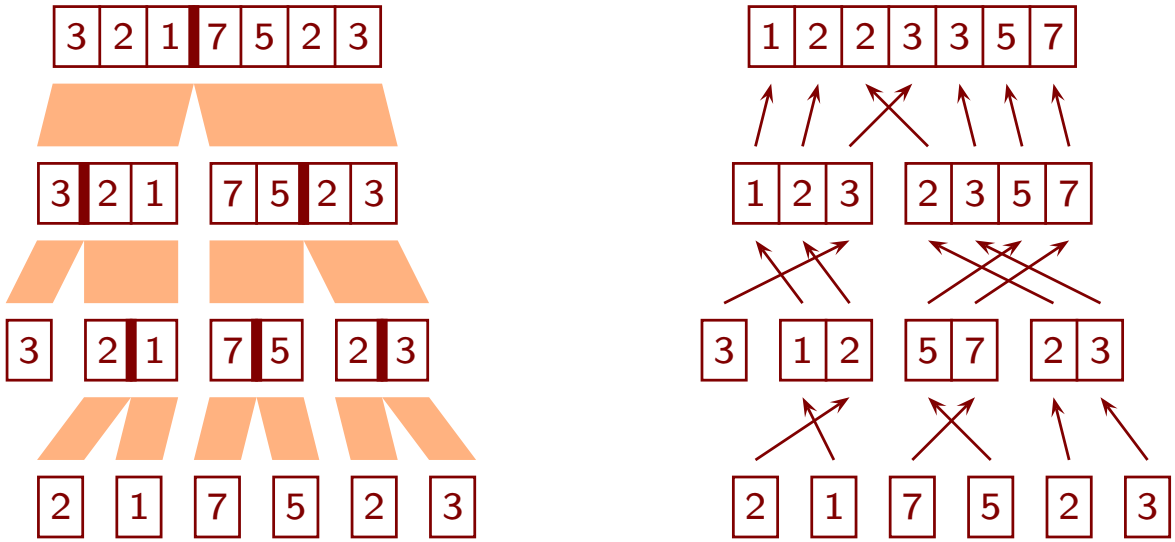
7.1 Lomitusjärjestäminen

Eivätkö jo esitellyt järjestämisalgoritmit riitä?

- lomitusjärjestäminen on ainoa vakaa laajalti tunnettu $O(n \log n)$ järjestämisalgoritmi
  - (aputaulukoilla saadaan mistä tahansa vakaa)
  - C++:n `stable_sort` on sen kaksi muunnelmaa (tätä ei voi luvata varmasti)
- toisin kuin kekojärjestäminen, se toimii hyvin myös linkitetyille listoille
- sen ainoa olennainen heikkous taulukoilla on $\Theta(n)$ lisämuistin tarve
  - linkitetyillä listoilla riittää $\Theta(1)$ lisämuistia
- paikallisuuden tuoma nopeusetu menetetään kun alkiot ovat osoittimien takana
  - Javan ja Pythonin `sort` eivät perustu pikajärjestämiseen vaan lomitusjärjestämiseen (tätä ei voi luvata varmasti)
  - "Timsort", "powersort"

Rekursiivinen lomitusjärjestäminen

- helpoin ymmärtää, mutta ei tehokkain
- etenee ylhäältä alas
- jaetaan taulukko mahdollisimman tarkasti tasan
- osataulukot järjestetään rekursiivisesti
- järjestetyt osat lomitetaan yhdeksi järjestetyksi taulukoksi



Rekursiivinen lomitusjärjestäminen ohjelmana

```
1   void mergerec( taulukko & A, taulukko & B, unsigned aa, unsigned ll ){
2     if( ll - aa <= 1 ){ return; }
3     unsigned vv = (aa + ll) / 2;
4     mergerec( A, B, aa, vv ); mergerec( A, B, vv, ll );
5     unsigned ii = aa, jj = aa, kk = vv;
6     while( jj < vv && kk < ll ){
7       if( A[ jj ].x <= A[ kk ].x ){ B[ ii ] = A[ jj ]; ++ii; ++jj; }
8       else                         { B[ ii ] = A[ kk ]; ++ii; ++kk; }
9     }
10    while( jj < vv ){ B[ ii ] = A[ jj ]; ++ii; ++jj; }
11    while( kk < ll ){ B[ ii ] = A[ kk ]; ++ii; ++kk; }
12    for( unsigned ii = aa; ii < ll; ++ii ){ A[ii] = B[ii]; }
13  }

14  void mergerec( taulukko & A ){
15    taulukko B( A.size() ); mergerec( A, B, 0, A.size() );
16  }
```

- `aa` on osan alku ja `ll` on loppu plus yksi
- `ii` selaa tulostaulukkoa, `jj` ensimmäistä ja `kk` toista osaa, `vv` on toisen osan alku
- muuttamalla rivi 3 muotoon `unsigned vv = aa + (ll-aa)/2;` vältetään ylivuodon vaara
  - sen sijaan ... `= aa/2 + ll/2;` ei toimi

Muunnelmien suoritusaikoja

- tulostaulukon lomitusaliohjelmassa varaava ja sieltä takaisin kopioiva on hitaampi

```
taulukko B( ll-aa );
...
for( unsigned ii = aa; ii < ll; ++ii ){ A[ ii ] = B[ ii-aa ]; }
```

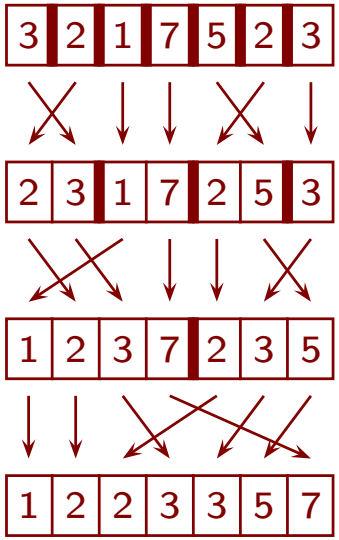
  ⇒ ei kannata toistuvasti varata ja vapauttaa muistia, jos voi käyttää kerran varattua

- `mergesort` on nopeampi, koska, kuten kohta näemme, se välttää takaisin kopioinnin
- pienten osataulukoiden järjestäminen lisäysjärjestämisellä nopeuttaa pienillä alkioilla
  - paikoitellen myös isoilla alkioilla

	0...4 999		10 000...19 999		10^6...1 000 019	
muistia toistuvasti varaava	0,8 s	14 s	11 s	3 min 45 s	2,0 s	52 s
`mergerec`	0,6 s	9,2 s	8,6 s	2 min 26 s	1,7 s	32 s
`mergesort`	0,5 s	7,2 s	7,7 s	2 min 18 s	1,5 s	28 s
`mergesort-8`	0,5 s	7,4 s	7,5 s	2 min 14 s	1,5 s	26 s
`mergesort-16`	0,5 s	7,9 s	7,1 s	2 min 19 s	1,4 s	26 s
`mergesort-32`	0,5 s	9,1 s	6,7 s	2 min 30 s	1,4 s	27 s
`mergesort-64`	0,5 s	12 s	6,8 s	3 min 00 s	1,4 s	31 s
listojen	1,2 s	3,3 s	18 s	1 min 12 s	15 s	44 s

Iteratiivinen lomitusjärjestäminen

- etenee alhaalta ylös
- lomitetaan aina kaksi peräkkäistä yhden mittaista osaa kahden mittaiseksi
- lomitetaan aina kaksi peräkkäistä kahden mittaista osaa neljän mittaiseksi
- lomitetaan aina kaksi peräkkäistä neljän mittaista osaa kahdeksan mittaiseksi
- ...
- viimeinen osa saa olla muita lyhyempi
- kun pääsilmukka on kiertänyt h kierrosta, koostuu taulukko 2^h pituisista järjestetyistä osista (ja viimeisestä osasta)



- jos käytetään lisäysjärjestämistä tehostamaan, se tehdään heti aluksi
  - jaetaan taulukko esim. 8:n pituisiin osiin, jotka lisäysjärjestetään

Iteratiivinen lomitusjärjestäminen ohjelmana

```
 1  inline void merge( const taulukko & A, taulukko & B, unsigned dd ){
 2    unsigned ii = 0;
 3    while( ii < A.size() ){
 4      unsigned jj = ii, kk = ii + dd, vv = kk, ll = kk + dd;
 5      if( A.size() < vv ){ vv = A.size(); }
 6      if( A.size() < ll ){ ll = A.size(); }
        ... edeltä rivit 6, ..., 11
13    }
14  }

15  void mergesort( taulukko & A ){
16    taulukko B( A.size() );
17    for( unsigned dd = 1; dd < A.size(); dd *= 2 ){
18      merge( A, B, dd ); A.swap( B );
19    }
20  }
```

- `const` estää `merge`:ä vahingossa sijoittamasta `A`:han
  - `&` tarvitaan estämään `A`:n kopiointi aliohjelmakutsussa
- C++:n `swap` vaihtaa kahden `vector`:in sisällön vakioajassa
  - mahdollista, koska varsinainen sisältö on osoittimen takana omassa muistialueessa
  - korvike: `merge( A, B, dd ); dd *= 2; merge( B, A, dd );`

Iteratiivinen lomitusjärjestäminen linkitetyille listoille

- listojen lomittaminen ei tarvitse aputaulukkoa
  - rekursiivisen version lisämuistin kulutus on $\Theta(\log n)$
  - iteratiivisen version lisämuistin kulutus on $\Theta(1)$
  - toisaalta osoittimet vievät $\Theta(n)$ lisämuistia hyötykuormaan nähden
- osalistojen rajat pitää etsiä selaamalla listaa
- osalistojen rajoina voi käyttää myös kohtia, joissa alkio on edeltäjäänsä pienempi
  - jokainen pääsilmukan kierros pienentää osalistojen määrän suunnilleen puoleen
  - jos listassa on alun perin pitkiä osalistoja, niin tarvitaan vähemmän kierroksia
- edellä olleet mittaukset tehtiin tällaisella toteutuksella
  - mittausteknisistä syistä aikoihin sisältyy listan luonti (ja purkaminen)

  $\Rightarrow$ ajat eivät ole vertailukelpoisia muihin

7.2 Satunnaisen järjestyksen tuottaminen

Tarvitaan esimerkiksi lottorivin tuottamiseksi

Toimintaperiaate

- arvotaan koko taulukosta alkio, ja vaihdetaan se ensimmäiseksi
- arvotaan muista kuin ensimmäisestä alkio, ja vaihdetaan se toiseksi
- arvotaan muista kuin kahdesta ensimmäisestä alkio, ja vaihdetaan se kolmanneksi
- ...
- alla $\textsc{Random}(i, n-1)$ tuottaa tasan jakautuneen satunnaisluvun väliltä $i, \ldots, n-1$

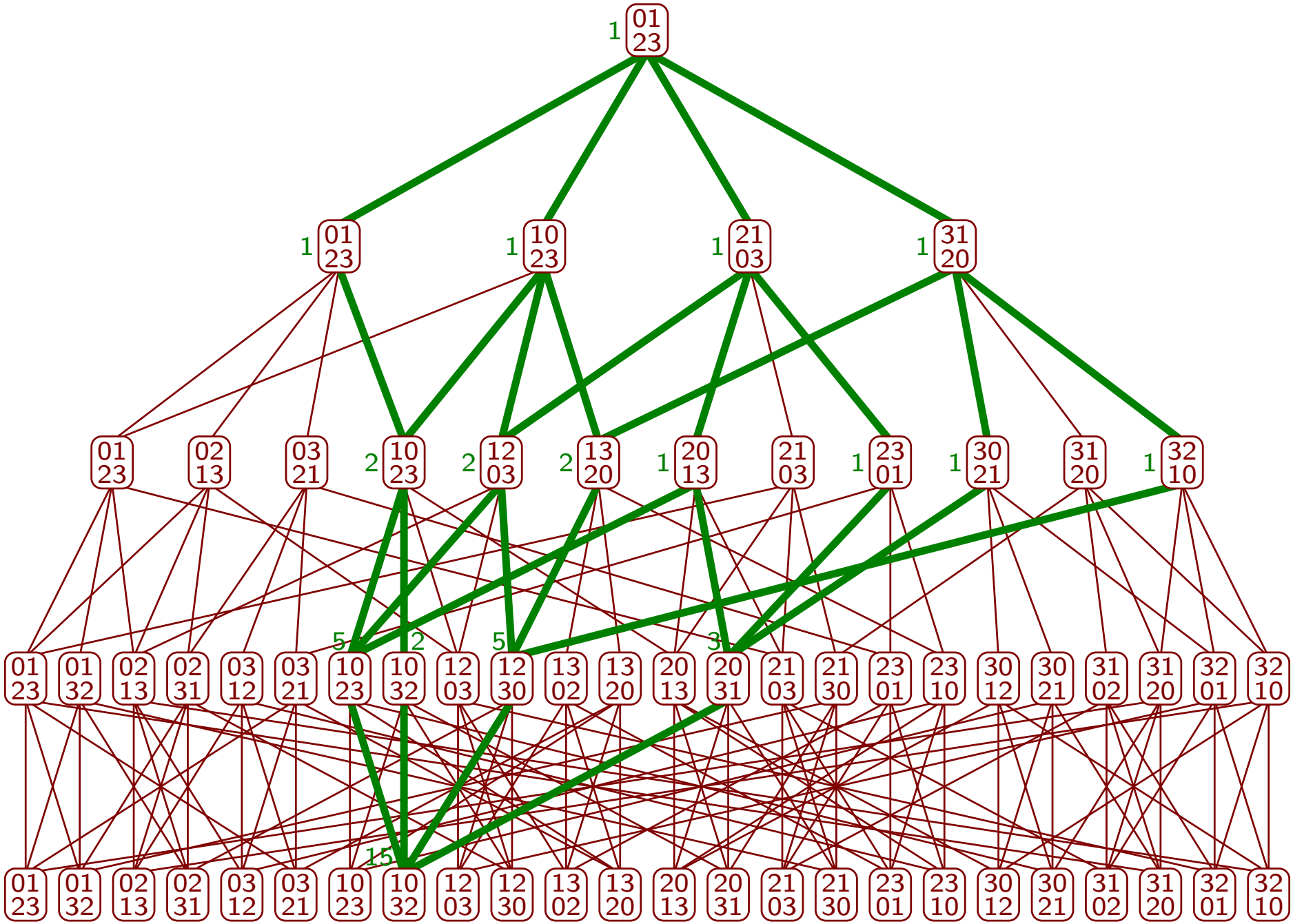
```
for i := 0 to n−1 do A[i] := i
for i := 0 to n−2 do
    Swap(A, i, Random(i, n−1))
```

Arpominen joka kerta koko taulukosta ei tuota tasajakaumaa

- esim. kolmen alkion taulukolla on 6 järjestystä ja 27 sellaista arvontatulosta
- 27 ei ole tasan jaollinen kuudella

⇒ osa järjestyksistä tulee 4 ja osa 5 kertaa

- neljän alkion taulukolla on 256 arvontatulosten yhdistelmää
  - 1 0 3 2 tulee 15:ssa
  - 3 0 1 2 tulee 8:ssa


1 01 23
1 01 23
1 10 23
1 21 03
1 31 20
01 23
02 13
03 21
2 10 23
2 12 03
2 13 20
1 20 13
21 03
1 23 01
1 30 21
31 20
1 32 10
5
2
5
3
01 23
01 32
02 13
02 31
03 12
03 21
10 23
10 32
12 03
12 30
13 02
13 20
20 13
20 31
21 03
21 30
23 01
23 10
30 12
30 21
31 02
31 20
32 01
32 10
15
01 23
01 32
02 13
02 31
03 12
03 21
10 23
10 32
12 03
12 30
13 02
13 20
20 13
20 31
21 03
21 30
23 01
23 10
30 12
30 21
31 02
31 20
32 01
32 10

Satunnaisluvun tuottamisesta väliltä $0, \ldots, m-1$

- $\textsc{Random}() \bmod m$ tuottaa tasajakauman vain jos $\textsc{Random}()$:in tuottamien eri satunnaislukujen määrä on jaollinen m:llä
- tyypillinen satunnaislukugeneraattori tuottaa 2^{31} tai 2^{32} eri arvoa
- esim. jos $\textsc{Random}()$ tuottaa luvut $0, \ldots, 2^{31}-1$, niin $\textsc{Random}() \bmod 1000$ tuottaa
  - $2\,147\,484$ kertaa arvot $0, \ldots, 647$
  - $2\,147\,483$ kertaa arvot $648, \ldots, 999$

$\Rightarrow$ jos jakauman pitää olla tarkasti tasainen, niin kun $\textsc{Random}()$ tuottaa lähellä maksimia olevan luvun, niin se pitää hylätä ja arpoa uusi
  - esim. $2\,147\,483\,000$ tai yli

- satunnaisen järjestyksen tuottamisessa m saa arvot $n, n-1, \ldots, 2$
  $\Rightarrow$ on usein muu kuin kahden potenssi

7.3 Vertaamiseen perustuvan järjestämisen ajan alaraja

Koko tämän luvun ajan oletetaan (paitsi kun erikseen sanotaan toisin)

- taulukon kaikki luvut ovat keskenään erisuuret
- algoritmi hankkii tietoa järjestyksestä vain vertaamalla alkioita toisiinsa

Järjestäminen sisältää oikean alkion valitsemisen jokaiseen paikkaan

- pienin ensimmäiseksi, toiseksi pienin toiseksi, ..., suurin viimeiseksi
- niistä syntyy yhteensä $n \cdot (n-1) \cdot \ldots \cdot 1 = n!$ mahdollisuutta
  - jo paikan saaneet eivät ole mukana myöhemmissä valinnoissa
  - n ***kertoma***

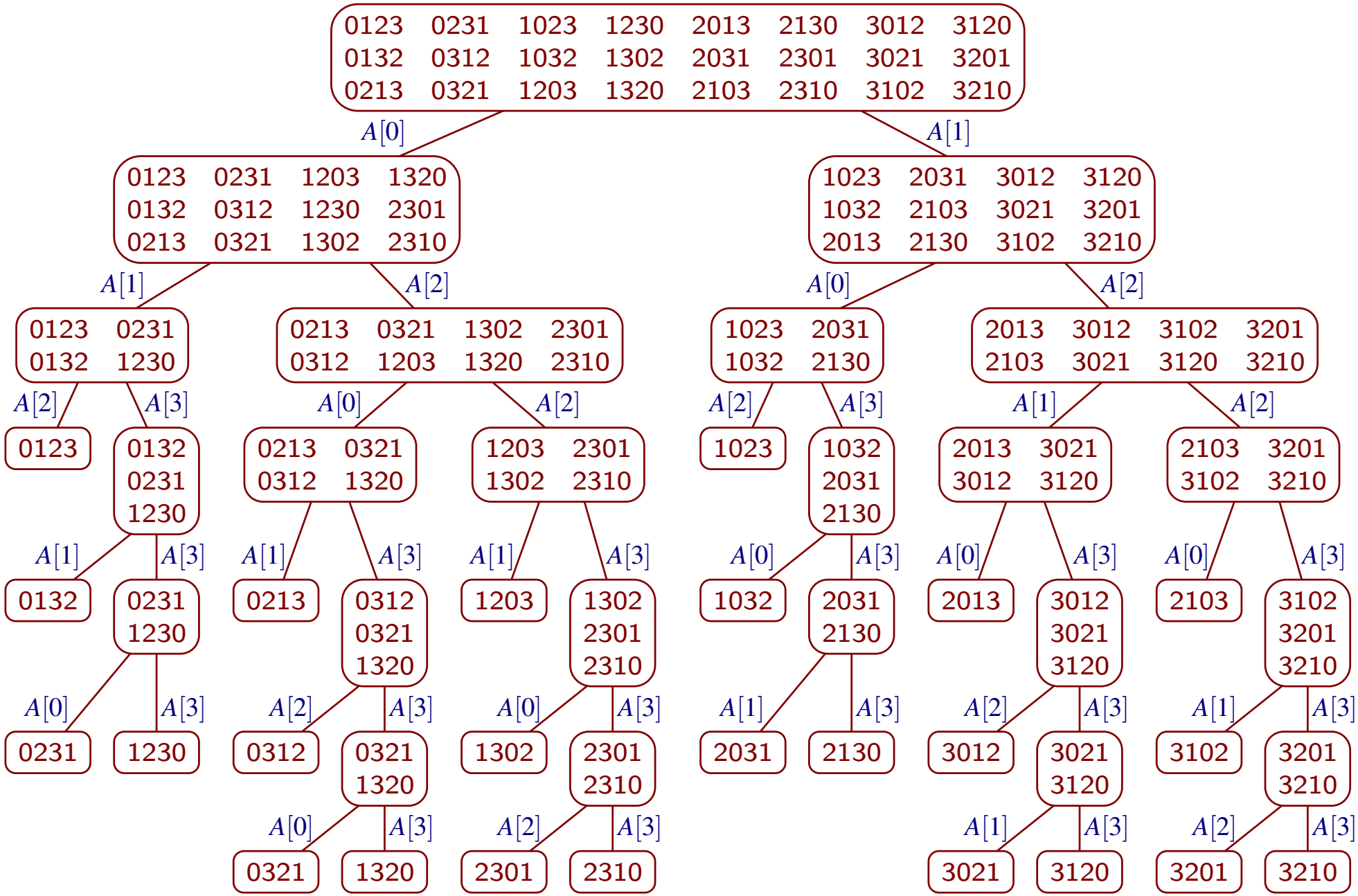
Jokainen vertailu jakaa jäljellä olevat mahdolliset järjestykset kahtia

- niihin, joilla vertailusta tuli se tulos mikä tuli, ja muihin

$\Rightarrow$ puu, jonka kussakin solmussa on siihenastisiin vertaamistuloksiin sopivat järjestykset

Seuraavassa ruudussa on lisäysjärjestämisen puu neljän alkion tapauksessa

- ensimmäinen kahtiajako sen mukaan, onko toinen alkio suurempi kuin ensimmäinen
- toinen kahtiajako sen mukaan, onko kolmas alkio suurempi kuin kaksi ensimmäistä
- jos kolmas on suurempi, niin seuraava sen mukaan onko neljäs suurempi kuin kolmas
- muutoin sen mukaan, onko kolmas suurempi kuin pienempi ensimmäisistä
- sitten neljättä verrataan muihin suurimmasta alkaen kunnes jäljellä on yksi järjestys


0123 0231 1023 1230 2013 2130 3012 3120
0132 0312 1032 1302 2031 2301 3021 3201
0213 0321 1203 1320 2103 2310 3102 3210
A[0]
A[1]
0123 0231 1203 1320
0132 0312 1230 2301
0213 0321 1302 2310
1023 2031 3012 3120
1032 2103 3021 3201
2013 2130 3102 3210
A[1]
A[2]
A[0]
A[2]
0123 0231
0132 1230
0213 0321 1302 2301
0312 1203 1320 2310
1023 2031
1032 2130
2013 3012 3102 3201
2103 3021 3120 3210
A[2]
A[3]
A[0]
A[2]
A[2]
A[3]
A[1]
A[2]
0123
0132
0231
1230
0213 0321
0312 1320
1203 2301
1302 2310
1023
1032
2031
2130
2013 3021
3012 3120
2103 3201
3102 3210
A[1]
A[3]
A[1]
A[3]
A[1]
A[3]
A[0]
A[3]
A[0]
A[3]
A[0]
A[3]
0132
0231
1230
0213
0312
0321
1320
1203
1302
2301
2310
1032
2031
2130
2013
3012
3021
3120
2103
3102
3201
3210
A[0]
A[3]
A[2]
A[3]
A[0]
A[3]
A[1]
A[3]
A[2]
A[3]
A[1]
A[3]
A[2]
A[3]
0231
1230
0312
0321
1320
1302
2301
2310
2031
2130
3012
3021
3120
3102
3201
3210
A[0]
A[3]
A[2]
A[3]
A[1]
A[3]
A[2]
A[3]
0321
1320
2301
2310
3021
3120
3201
3210

Jos puun korkeus on k, niin puussa on enintään 2^k lehteä

- lehti on solmu, jolla ei ole lapsia
- $n!$ järjestystä vastaa $n!$ lehteä

$\Rightarrow$ $n! \leq 2^k \;\Rightarrow\; k \geq \log_2 n! = \log_2 1 + \ldots + \log_2 n > \frac{1}{2} n \log_2 n - \frac{1}{2} n$, kun $n \geq 1$ (ruutu 59)

$\Rightarrow$ huonoimmassa tapauksessa tarvitaan $\Omega(n \log n)$ vertailua

$\Rightarrow$ jokainen vain vertaamiseen perustuva järjestämisalgoritmi käyttää hitaimmillaan $\Omega(n \log n)$ aikaa

Tarkempi laskelma (jota emme tee) osoittaa, että tulos pätee myös keskimäärin

- kuten edellä näkyi, vertailut eivät välttämättä jaa järjestyksiä tasan
- isompi kasa päätyy jatkoon todennäköisemmin kuin pienempi
- isompi kasa tarvitsee keskimäärin enemmän vertailuja kuin pienempi

$\Rightarrow$ mitä epätasaisempia jaot ovat, sitä enemmän keskimäärin tarvitaan vertailuja

Jos kaikki alkiot ovat keskenään erisuuret, niin jokainen järjestämisalgoritmi, joka hankkii tiedon oikeasta järjestyksestä vertaamalla alkioita (eikä muilla keinoin), käyttää keskimäärin $\Omega(n \log n)$ aikaa.

Jos alkioissa saa olla yhtäsuuria, niin pienempi työmäärä saattaa riittää

- pikajärjestämisen taulukko voidaan osittaa kuvan mukaisesti ajassa $\Theta(n)$
  - sellaisen kirjoittaminen oli aiemmin kotitehtävänä

$< x$	$= x$	$> x$

- sellainen pikajärjestäminen ei lähetä jakoalkion suuruisia alemmille rekursiotasoille

$\Rightarrow$ jos avaimissa on enintään viittä eri arvoa, niin rekursiotasoja on enintään viisi

$\Rightarrow$ ajan kulutus on $\Theta(n)$

- mikä tahansa pieni vakio kelpaa "viiden" tilalle
  - jos se ei ole pieni, niin tulos on silti oikein, mutta vastaa huonosti käytäntöä
  - jos vakiota merkitään k:lla, niin ajan kulutus on $O(nk)$
  - jos k kasvaa paljon hitaammin kuin $\log n$, niin se on parempi kuin $O(n \log n)$
  - vakio ei kasva lainkaan!
- binääripuilla voi vertailemiseen perustuen järjestää ajassa $O(n \log k)$

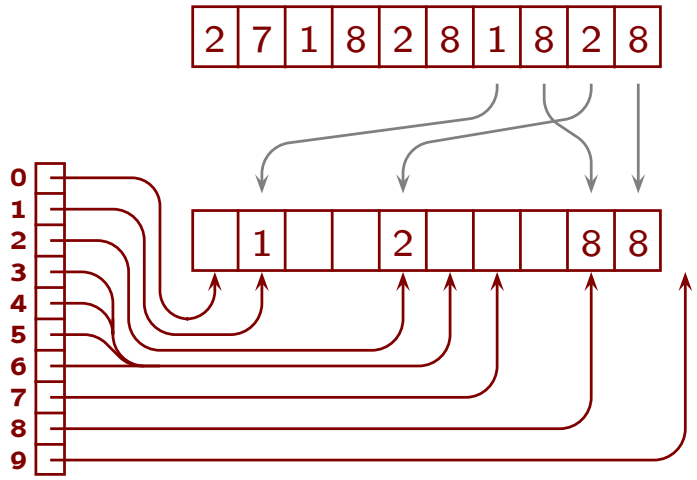
Seuraavaksi tarkastellaan kahta järjestämisalgoritmia, jotka eivät perustu vertaamiseen

7.4 Laskentajärjestäminen

Toimintaperiaate

- luodaan laskuritaulukko, jossa on lokero jokaiselle mahdolliselle avaimen arvolle
  - laskuri on (etumerkitön) kokonaisluku
  - laskurit nollataan rivillä 3 (ohjelmointikielissä voi sisältyä luontikomentoon)
- käydään järjestettävä taulukko läpi ja lasketaan, monestiko kukin avain esiintyy
- muutetaan kukin laskuri kertomaan enintään indeksinsä suuruisten avainten esiintymien määrä
- kopioidaan alkiot tulostaulukkoon suoraan oikeille paikoilleen
  - rivillä 6 kukin laskuri sisältää indeksiään pienempien ja kopioimattomien indeksinsä suuruisten alkioiden yhteismäärän
- rivillä 8 käytetty vakioaikainen taulukoiden vaihto on ainakin C++:ssa

```
1  CountingSort(&A,k)
2  luo taulukko B(A.koko); luo taulukko L(k)
3  for j := 0 to k−1 do L[j] := 0
4  for i := 0 to A.koko−1 do L[A[i].x] := L[A[i].x]+1
5  for j := 1 to k−1 do L[j] := L[j]+L[j−1]
6  for i := A.koko−1 downto 0 do
7      L[A[i].x] := L[A[i].x]−1; B[L[A[i].x]] := A[i]
8  A.vaihda(B)
```



Ominaisuuksia

- avainten on oltava peräisin pienehköstä joukosta
  - kokonaislukuja väliltä $0, \ldots, k-1$, tai koodattavissa sellaisiksi järjestys säilyttäen
  - esim. auton rekisteri maks. kolme kirjainta ja numeroa $\rightsquigarrow k = 25\,259\,000$

$\Rightarrow$ ei sovellu esimerkiksi ihmisten nimien järjestämiseen
  - esim. 7-kirjaimisia jonoja on yli 17 miljardia

- vakaa
  - viimeinen silmukka käy indeksit läpi takaperin
- suoritusaika ja lisämuistin tarve $\Theta(n+k)$
  - kutakin avainta katsotaan vain pari kertaa, kukin alkio siirretään vain kerran
  - muutenkin tehdään kaikkiaan vain vähän työtä

  $\Rightarrow$ erittäin nopea varsinkin pienillä alkioilla

Ruudussa 136 oli $k = n$

- avaimet oli valittu satunnaisesti väliltä $0, \ldots, n-1$

7.5 Kantalukujärjestäminen

(***Radix sort***)

Toimintaperiaate

- pilkotaan avaimet osiin
- järjestetään vakaalla algoritmilla vähiten merkitsevän osan mukaan, sitten toiseksi vähiten, sitten kolmanneksi vähiten, ...
  - usein tähän käytetään laskentajärjestämistä

Jos osia on d kappaletta ja osa voi saada k arvoa, niin laskentajärjestämisellä

- ajan kulutus on $\Theta(d(n+k))$
- lisämuistin tarve on $\Theta(k)$

Ruudussa 136 oli $k = 256$ ja $d = 4$

k	d	0...4 999		10 000...19 999		10^6...1 000 019	
16	8	0,3 s	5,8 s	3,2 s	1 min 43 s	0,4 s	16 s
256	4	0,1 s	4,6 s	1,6 s	1 min 17 s	0,3 s	12 s
4096	3	0,1 s	4,3 s	1,4 s	1 min 09 s	0,3 s	11 s
65536	2	0,3 s	4,3 s	1,7 s	1 min 02 s	0,3 s	10 s

8 Hajautustaulut

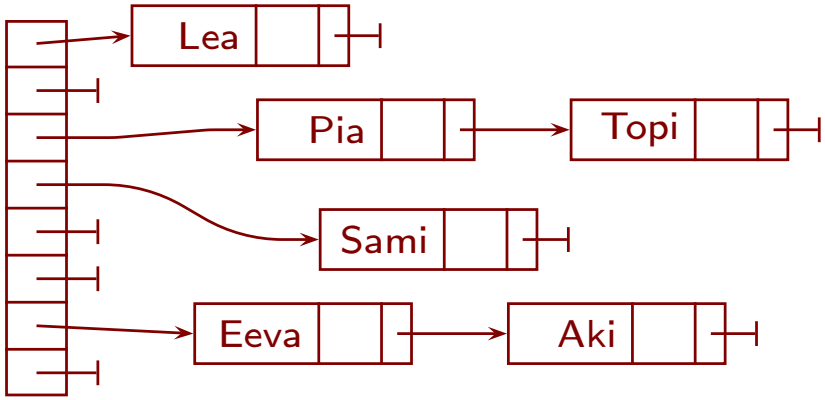
8.1 Ketjutetut hajautustaulut

Hajautustaulu (***hash table***) on tehokas rakenne tietojen tallentamiseen

- keskimäärin hyvin nopea lisääminen, avaimella etsiminen ja poistaminen
  - ajan kulutus kullakin keskimäärin $\Theta(1)$, huonoimmillaan $\Theta(n)$
  - avain voi olla esim. henkilön nimi
- ei tarjoa alkioiden etsimistä suuruusjärjestyksen perusteella
- käyttää kohtuullisesti lisämuistia

Ketjutetun hajautustaulun rakenne (***hashing with chaining***)

- alkio koostuu avaimesta, muusta hyötykuormasta ja linkistä seuraavaan alkioon
- alkioista on muodostettu keskimäärin lyhyitä linkitettyjä listoja
  $\Rightarrow$ jos alkioita on paljon, niin listojakin on paljon
- listojen alut ovat taulukossa osoittimia
- listojen sisäisellä järjestyksellä ei ole väliä



Taulukon koon valitseminen

- liian pieni taulukko johtaa pitkiin listoihin ja hitauteen
- liian iso taulukko tuhlaa muistia
- voidaan aloittaa esim. 256 alkion taulukolla
  - kohtuullinen määrä muistia, vaikka alkioita olisi vähän
- voidaan kaksinkertaistaa taulukko esim. aina kun n ylittää 2 kertaa taulukon koon

Hajautusfunktio (***hash function***)

- ottaa avaimen ja tuottaa luvun väliltä $0, \ldots, M-1$, missä M on taulukon koko
- tavoitteena satunnaiselta vaikuttava hajautus
  - ei saa syntyä pitkiä listoja, paitsi sattumalta
- mielellään oltava nopea laskea
  - saattaa muodostaa merkittävän osan toimintojen ajan kulutuksesta
- hajautusfunktio pitää suunnitella huolella!
  - ei esimerkiksi henkilön nimen viimeinen kirjain

Johdanto esimerkeille hajautusfunktioista merkkijonoille

- aineisto: Nykysuomen sanalista. Kotimaisten kielten keskus. Päivitetty 30.4.2024
  - viitattu 10.5.2024
  - saatavissa https://kaino.kotus.fi/lataa/nykysuomensanalista2024.csv
  - $104\,743$ sanaa kuten "algoritmi" tai sanaliittoa kuten "all stars -joukkue"
  - myös eksoottisia merkkejä kuten "à la carte -annos"
- $M = 2^{16} = 65\,536$
- hajautusfunktio laskettiin jokaiselle sana(liito)lle ja eri tulosten määrät kirjattiin
  - tuloksen esiintymien määrä = tulosta vastaavan listan pituus
- esimerkeissä käytettiin laadun mittareina suurinta listan pituutta ja lisätyön määrää
  - lisäystoiminnossa selataan lista läpi, jotta ei lisättäisi avainta, joka on jo mukana
  - lisätyön määrä on tästä syystä selattujen alkioiden kokonaismäärä

Kaksi ensimmäistä tavua UTF8-koodauksen mukaan

```
1  unsigned haj_fu( const std::string & sana ){
2    unsigned tulos = (unsigned char)( sana[ 0 ] );
3    if( sana.length() > 1 ){
4      tulos |= (unsigned char)( sana[ 1 ] ) << 8;
5    }
6    return tulos;
7  }
```

- `(unsigned char)` muuttaa `char`:n 8-bittiseksi ei-negatiiviseksi luvuksi
  - C++:ssa `char` on 8-bittinen, mutta voi olla joko etumerkillinen tai etumerkitön
- suurimmalle osalle Suomen kielen merkeistä jokainen merkki on tavu
  - Å, Ä, Ö, å, ä, ja ö ovat C3 85, C3 84, C3 96, C3 A5, C3 A4 ja C3 B6
  - useimmat eksoottiset merkit ovat ainakin kaksi tavua kukin
- nopea laskea
- maksimipituus $4\,208$, lisätyö $71\,508\,730$
  - todella huono
- pisimmmässä listassa kaadanta, kaadattaa, kaade, kaadella, . . . , kavuta

⇒ sanan monien (kaikkien?) merkkien pitää vaikuttaa, vaikka se hidastaa laskentaa

Jokainen tavu vaikuttaa xor:illa tuloksen jompaankumpaan tavuun

```
1  unsigned haj_fu( const std::string & sana ){
2    unsigned tulos = 0;
3    for( unsigned ii = 0; ii < sana.length(); ++ii ){
4      tulos ^= (unsigned char)( sana[ ii ] ) << 8*(ii%2);
5    }
6    return tulos & 0xFFff;
7  }
```

- maksimipituus 40, lisätyö 954 889
  - parani paljon, mutta yhä huono
- pisimmmässä aamiainen, aavistella, aavistuksenomainen, ahmia, . . . , öljynetsintä
- 55 181 tyhjää ja 10 355 epätyhjää listaa

⇒ eri merkkien samoissa bittikohdissa on säännöllisyyksiä
  - esim. eniten merkitsevä bitti on nolla kirjaimissa A, . . . , Z, a, . . . , z

Merkin bittien levittäminen bittipaikkoihin kertolaskulla

```
1  unsigned haj_fu( const std::string & sana ){
2    unsigned tulos = 0;
3    for( unsigned ii = 0; ii < sana.length(); ++ii ){
4      tulos ^= (1000*ii + 1) * (unsigned char)( sana[ ii ] );
5    }
6    return tulos & 0xFFff;
7  }
```

- maksimipituus 9, lisätyö 84 113, tyhjiä listoja 13 280
  - parani huomattavasti
- `ii` vaikuttaa kertoimeen, jotta eri kohdissa olevat merkit vaikuttaisivat eri tavalla
  - `1001` eikä `(1000*ii+1)` tuotti kehnon tuloksen 107, 393 320 ja 33 727
- kerroin on pariton, jotta alin bitti ei olisi aina 0
  - `1000*ii` eikä `(1000*ii+1)` tuotti 33, 681 727 ja 57 344
- vakio 1000 valittiin kokeilemalla muutamia eri arvoja ilman syvällistä viisautta

vakio	1	10	100	10 000	1 024	1 009
pisin	101	22	10	9	45	10
lisätyö	2 751 947	331 039	85 418	84 492	1 280 371	84 338
tyhjiä	61 682	43 915	13 548	13 459	59 393	13 347
huom.					2:n potenssi	alkuluku

Suurehkoilla alkuluvuilla kertominen ja summaus

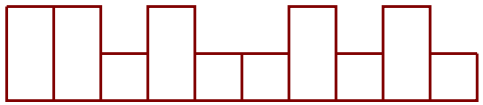
```
1  unsigned haj_fu( const std::string & sana ){
2    const unsigned kerroin[] =
3      { 7723, 7753, 7789, 7823, 7853, 7877, 7901, 7927, 7949, 7993 };
4    unsigned tulos = 0;
5    for( unsigned ii = 0; ii < sana.length(); ++ii ){
6      tulos *= kerroin[ ii % 10 ]; tulos += (unsigned char)( sana[ ii ] );
7    }
8    return tulos & 0xFFff;
9  }
```

- Wikipedia Hash function $\rightsquigarrow$ Character folding
- maksimipituus 9, lisätyö $84\,176$, tyhjiä listoja $13\,354$
  - ei parempi eikä huonompi kuin parhaat aikaisemmat

Netistä löytyisi paljon lisää hajautusfunktioita

Miksi tulosten paraneminen loppui?

- tasajakaumalla olisi maksimipituus 2, lisätyö $39\,207$ ja tyhjiä listoja 0
- viimeisillä alkioilla on suuri todennäköisyys osua "vääriin" paikkoihin
- viisi lisää tuottaa alla olevasta tasaisen todennäköisyydellä $\frac{5}{10} \cdot \frac{4}{10} \cdot \frac{3}{10} \cdot \frac{2}{10} \cdot \frac{1}{10} \approx 0{,}0012$



Listojen pituuksien jakaumasta

- kun $\frac{n}{M} = \ell$ on vakio ja $n \to \infty$, niin eripituisten listojen osuudet $\to e^{-\ell}\frac{\ell^k}{k!}$

ℓ	pituus	0	1	2	3	4	5	6
1	osuus %	37	37	18	6	2	0	0
1,598...	osuus %	20	32	26	14	5	2	0
2	osuus %	14	27	27	18	9	4	1

- nämä likiarvot ovat varsin päteviä jo kun $n = 1000$

$\Rightarrow$ täysin tasaista pituusjakaumaa ei kannata odottaa

- Nykysuomen sanalistalla $\ell = 1{,}598\ldots$, joten ennusteeksi tulee
  - pisin 9 tai 10
  - lisätyö $83\,702$
  - tyhjiä listoja $13\,254{,}6$
- satunnaisluvuilla tuli maksimipituus 9, lisätyö $83\,923$ ja tyhjiä listoja $13\,256$

$\Rightarrow$ aiemmin saatu maksimipituus 9, lisätyö $84\,113$ ja tyhjiä listoja $13\,280$ on erittäin hyvä!

Lisätyön ennuste on alkiota kohti $\approx \frac{\ell}{2}$ ja kaikkiaan $\frac{n(n-1)}{2M} \approx \frac{n^2}{2M}$

8.2 Sovellusesimerkki: reitin etsinnän syötteen uudistaminen

Syötteen erot aiempaan reitinetsintäohjelmaan

- solmuilla on nimet eikä numerot
- kaaret saavat tulla missä järjestyksessä tahansa

Solmut ja kaaret

```
5   struct solmutyyppi;
6   struct kaarityyppi{ solmutyyppi *minne; double pituus; };
7   struct solmutyyppi{
8     std::string nimi;
9     std::vector< kaarityyppi > kaaret;
10    double etaisyys; solmutyyppi *reitti, *haj_seur;
11    solmutyyppi( const std::string & nimi, solmutyyppi *haj_seur ):
12      nimi( nimi ), etaisyys( aareton ), reitti( 0 ), haj_seur( haj_seur ) {}
13  };
```

- solmuja ei enää osoiteta luvuilla vaan osoittimilla
- kullakin solmulla on oma siitä lähtevien kaarten taulukko
- solmussa on osoitin hajautustaulun listassa seuraavaan
  - saa arvonsa rakentajan kutsussa, kun hajautustauluun lisätään alkio

Hajautustaulu ja hajautusfunktio

```
14  std::vector< solmutyyppi* > haj_taulu( 8 );
15  unsigned haj_alkioita = 0, haj_maski = 0x7;

16  unsigned haj_fu( const std::string & sana ){
17    unsigned tulos = 0;
18    for( unsigned i = 0; i < sana.length(); ++i ){
19      tulos ^= (1000*i+1) * (unsigned char)( sana[ i ] );
20    }
21    return tulos & haj_maski;
22  }
```

- aloitetaan pienellä hajautustaululla ja kahdennetaan tarvittaessa
  - koko on aina 2:n potenssi
- `haj_alkioita` laskee alkioiden määrää kahdentamista varten
- `haj_maski` poimii hajautustaulun kokoa vastaavan määrän bittejä
  - kahden potenssi miinus yksi
- käytetään edellä 65 536:lla hyväksi havaittua hajautusfunktiota
  - emme tiedä kuinka hyvä se on muunkokoisilla hajautustauluilla, mutta sitä emme tiedä muistakaan hajautusfunktioista

Hajautustaulun kahdentaminen

```
23  void haj_kahdenna(){
24    haj_maski = 2*haj_maski + 1;
25    std::vector< solmutyyppi* > uusi_taulu( haj_maski + 1 );
26    for( unsigned i = 0; i <= haj_maski / 2; ++i ){
27      solmutyyppi *o1 = haj_taulu[ i ];
28      while( o1 ){
29        solmutyyppi *o2 = o1; o1 = o1->haj_seur;
30        unsigned h = haj_fu( o2->nimi );
31        o2->haj_seur = uusi_taulu[ h ]; uusi_taulu[ h ] = o2;
32      }
33    }
34    haj_taulu.swap( uusi_taulu );
35  }
```

- varataan kaksinkertainen taulukko ja siirretään alkiot sinne
  - siirtämisessä ei tarvitse kopioida alkioita, vaan ainoastaan muuttaa osoittimia
- taulukon koko on 2:n potenssi ja maski on sitä yhden pienempi
  $\Rightarrow$ rivi 24 tuottaa oikean maskin ja rivit 25 ja 26 käyttävät oikeita arvoja
- maski muutetaan alussa, jotta rivillä 30 käytettäisiin uuden taulun hajautusfunktiota
- selataan kaikkien listojen kaikki alkiot
  - ajan kulutus on $\Theta(n)$, koska alkioita on yhteensä n ja listoja on suunnilleen n

Hajautustauluun lisääminen tai siellä jo olevan löytäminen

```
36  solmutyyppi* haj_etsi_tai_lisaa( const std::string & nimi ){
37    unsigned h = haj_fu( nimi );
38    solmutyyppi *os = haj_taulu[ h ];
39    while( os && os->nimi != nimi ){ os = os->haj_seur; }
40    if( !os ){
41      os = new solmutyyppi( nimi, haj_taulu[ h ] );
42      haj_taulu[ h ] = os;
43      if( ++haj_alkioita > haj_maski ){ haj_kahdenna(); }
44    }
45    return os;
46  }
```

- jos nimi on jo hajautustaulussa, niin ei lisätä vaan palautetaan osoitin sen alkioon
- muussa tapauksessa lisätään alkio hajautustauluun ja palautetaan osoitin siihen
- jos alkioiden määrä saavuttaa 2:n potenssin, niin kasvatetaan hajautustaulua

Muita eroja aikaisemman version alkuosaan

```
3   #include <string>
47  solmutyyppi *lahto = 0, *maali = 0;
```

Solmun lukeminen syötteestä

```
48  solmutyyppi *lue_solmu(){
49    std::string nimi; std::cin >> nimi;
50    if( !std::cin ){ return 0; }
51    return haj_etsi_tai_lisaa( nimi );
52  }
```

- jos saadaan nimi, niin lisätään tai löydetään solmu ja palautetaan osoitin siihen
- muussa tapauksessa palautetaan osoitin ei minnekään

Syötteen lukeminen

```
53  bool lue_syote(){
54    lahto = lue_solmu();
55    if( !lahto ){ std::cout << "!!! Lähtö puuttuu\n"; return false; }
56    maali = lue_solmu();
57    if( !maali ){ std::cout << "!!! Maali puuttuu\n"; return false; }
58    while( true ){
59      solmutyyppi *hanta = lue_solmu();
60      if( !hanta ){ return true; }
61      kaarityyppi kaari;
62      kaari.minne = lue_solmu();
63      if( !kaari.minne ){ std::cout << "!!! Kärki puuttuu\n"; return false; }
64      std::cin >> kaari.pituus;
65      if( !std::cin ){ std::cout << "!!! Pituus puuttuu\n"; return false; }
66      if( !( 0. <= kaari.pituus && kaari.pituus < aareton ) ){
67        std::cout << "!!! Laiton pituus " << kaari.pituus << "\n"; return false;
68      }
69      hanta->kaaret.push_back( kaari );
70    }
71  }
```

- lopetetaan onnistuneesti, kun ei saada kaaren häntää
- jokaisesta muusta syötteen saamatta jäämisestä lopetetaan virheilmoituksella

Erot loppuosassa ohjelmaa

- solmumuuttujat ei `unsigned` vaan `solmutyyppi *`
- ei `solmut[ lahto ].etaisyys` vaan `lahto->etaisyys` jne.
- kaarten selaamisen aloitukseksi
  `for( unsigned i = 0; i < solmu1->kaaret.size(); ++i )`
- tulostetaan `lahto->nimi` jne.

8.3 Avoin osoitus

(***open addressing***)

Ei linkitettyjä listoja, vaan alkiot ovat suoraan taulukossa

- hajautusfunktio ottaa kaksi argumenttia: avaimen ja yrityskerran numeron
- avaimella x etsitään lokeroista $h(x,0)$, $h(x,1)$, $h(x,2)$, ... kunnes
  - löytyy,
  - kohdataan tyhjä lokero tai
  - on yritetty jokaista lokeroa

Heikkouksia

- suoritusajat nousevat rajusti, kun täyttöaste lähenee ykköstä
- avaimen paikalla voitava olla tyhjää lokeroa tarkoittava arvo
  - esim. henkilötunnuksena 000000A0000
- poistaminen edellyttää toisenkin erityisarvon
  - jos etsittäessä kohdataan poistettu, niin jatketaan etsintää
  - jos lisättäessä kohdataan poistettu, niin lisätään siihen

$\Rightarrow$ avointa osoitusta ei kannata käyttää

8.4 Bloom-suodatin

Erittäin vähän muistia käyttävä tietorakenne joukon likimääräiseksi esittämiseksi

- jäsenyystesti tuottaa "varmasti ei" tai "ehkä kyllä"
  - väärän "kyllä" todennäköisyys saadaan pieneksi (***Bloom filter***)
- alkioita voidaan lisätä
- perusmuotoisesta Bloom-suodattimesta ei voi poistaa alkioita

Esimerkiksi sanojen talletus oikolukuohjelmaa varten

- väärä "kyllä" tarkoittaa, että väärin kirjoitettu sana jää havaitsematta

Rakenne

- bittitaulukko $H[0 \ldots M-1]$
- k hajautusfunktiota $h_1(x), \ldots, h_k(x)$
- x talletetaan asettamalla jokainen $h_i(x)$ ykköseksi

Esimerkiksi 100 000 sanaa, 1 000 000 bittiä ja 7 hajautusfunktiota

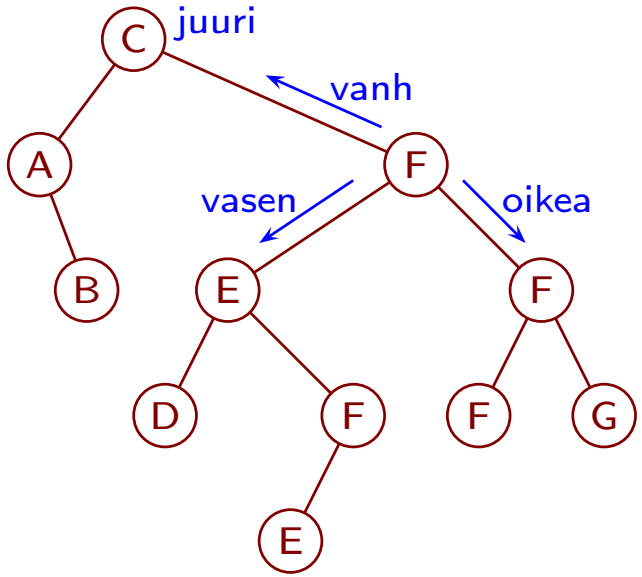
- bitti jää nollaksi todennäköisyydellä
  $(1-10^{-6})^{700000} = (1+\frac{-0{,}7}{700000})^{700000} \approx e^{-0{,}7} \approx 49{,}7\,\%$

$\Rightarrow$ väärän "kyllä" todennäköisyys $\approx (1-0{,}497)^7 \approx 0{,}8\,\%$

- käytetään vain 10 bittiä eli 1,25 tavua sanaa kohti!

9 Binääripuut

9.1 Yleistä binääripuista



Solmun rakenne

```
1  struct solmu{
2    solmu *vasen, *oikea, *vanh;
3    // hyötykuormaa
4  }
```

- `juuri->vanh` ei osoita minnekään
- jos `s1->vasen` on olemassa, niin `s1->vasen->vanh` = `s1`
- jos `s1->oikea` on olemassa, niin `s1->oikea->vanh` = `s1`
- jos `s1->vanh` on olemassa, niin `s1->vanh->vasen` = `s1` tai `s1->vanh->oikea` = `s1`
- vain osa binääripuurakenteista käyttää `vanh`-osoittimia (***binary tree***)

Koko puun tulostaminen rekursiivisesti ilman `vanh`-osoittimia, kutsu `tulosta( juuri )`

```
1  void tulosta( solmu *s1 ){
2    if( !s1 ){ return; }
3    std::cout << "("; tulosta( s1->vasen ); std::cout << s1->x;
4    tulosta( s1->oikea ); std::cout << ")";
5  }
```

- tulostaa esim. `((A(B))C(((D)E((E)F))F((F)F(G))))`

Muun muassa matemaattisia lausekkeita voi esittää binääripuina

- esim. MathCheckin lausekepuun solmu

```
1   /* Node of the expression tree or expression DAG */
2   class expression{
3   private:
4     /* For a constant, val is its value as a number. For a variable, val is its
5       index in var_used. Left precedences must be even and right precedences
6       odd. */
7     op_type opr_;                    // operand
8     expression *left_, *right_;      // subexpressions
9     number val_;                     // number value or variable letter
10    unsigned h_val_;                 // hash value (needed of sub-expressions)
11    expression *h_next,              // next in hash list
12      *smpl;                         // either 0 or simplified version of the expr.
13    unsigned type_;                  // static information (e.g., may be < 0)
14    union{
15      void *extra_ptr_;              // pointer for specific needs of algorithms
16      unsigned extra_uns_;           // some bytes of workspace
17    };
      ··· monta sataa riviä toimintoja ja hieman staattista dataa
    }
```

Otteita MathCheckistä kohdasta, jossa se laskee derivaattoja

```
1  expression *derivative( expression *ee, unsigned vv ){
2    /* Variables and constant functions */
3    if( known_const( ee ) ){ return expr_0; }
4    op_type opr = ee->opr();
5    if( opr == op_var ){ return ee->var_idx() == vv ? expr_1 : expr_0; }
```

- vakion derivaatta on 0
  - `known_const` tunnistaa vakioiksi lausekkeet, joissa ei esiinny muuttujia, kuten $2\pi + \sqrt{5}$
- muuttujan, jonka suhteen derivoidaan, derivaatta on 1
- muiden muuttujien derivaatat ovat 0

```
1      if( opr == op_sqrt ){
2        return new_arit(
3          derivative( e2, vv ),
4          op_div,
5          new_arit( ee, op_vprod, expr_2 )
6        );
7      }
```

$$\frac{d}{dx}\sqrt{f(x)} = \frac{\frac{d}{dx}f(x)}{\sqrt{f(x)}\cdot 2}$$

- `new_arit` tuottaa uuden aritmeettisen lausekepuun solmun
- `op_vprod` on näkyvä kertolasku, kuten $2 \cdot x$

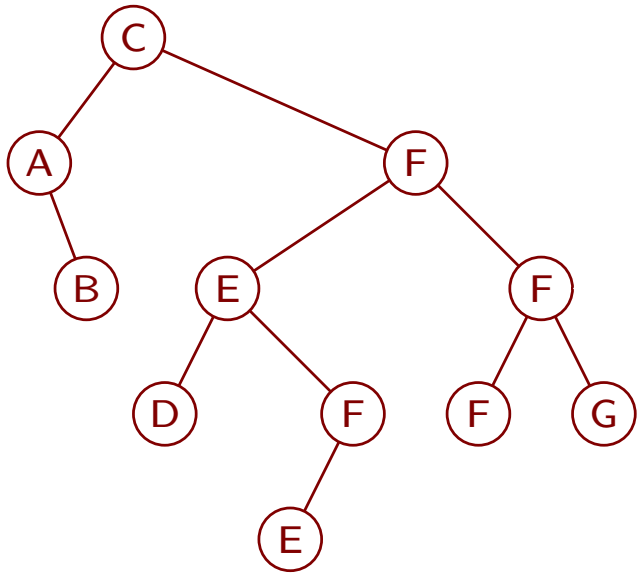
```
1     if( opr == op_vprod || opr == op_iprod ){
2       return new_arit(
3         new_arit( derivative( e1, vv ), op_vprod, e2 ),
4         op_plus,
5         new_arit( e1, op_vprod, derivative( e2, vv ) )
6       );
7     }
```

$$\frac{d}{dx}\big(f(x)\cdot g(x)\big) = \frac{d}{dx}f(x)\cdot g(x) + f(x)\cdot\frac{d}{dx}g(x)$$

- `op_iprod` on näkymätön kertolasku, kuten $2x$

Korkeus (*height*)

- solmun korkeus on mahdollisimman pitkän matkan pituus solmusta alaspäin
- eri asia kuin syvyys
  - matkan pituus juuresta solmuun
- lapsettoman solmun korkeus on 0
- koko puun korkeus on juuren korkeus
- jatkossa n on solmujen määrä ja k on korkeus



Siirtyminen solmusta seuraavaan `vanh`-osoittimien avulla

```
if( s1->oikea ){
  s1 = s1->oikea;
  while( s1->vasen ){ s1 = s1->vasen; }
}else{
  while( s1->vanh && s1->vanh->oikea == s1 ){ s1 = s1->vanh; }
  s1 = s1->vanh;
}
```

- yksittäinen siirtymä vie hitaimmillaan $\Theta(k)$ aikaa
- koko puun selaaminen vie $\Theta(n)$ aikaa
  - jokainen kaari kuljetaan kerran alas ja kerran ylös

9.2 Binäärihakupuut

Kukin avain vasemmassa alipuussa $\leq$ solmun avain $\leq$ kukin avain oikeassa alipuussa

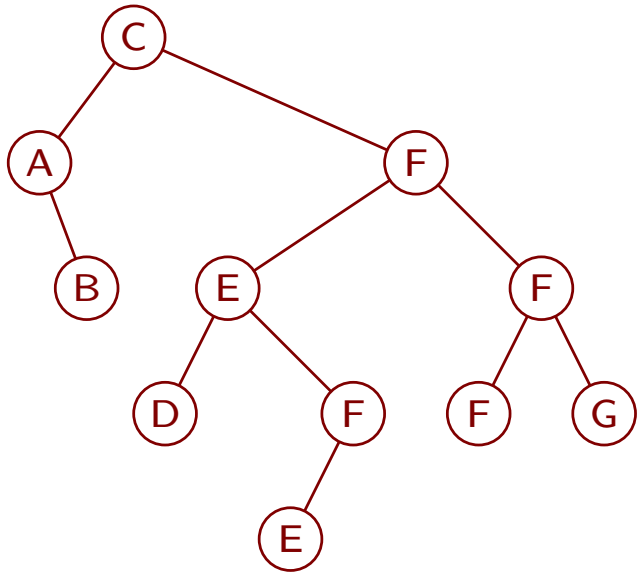
Etsintä avaimen perusteella, mikä tahansa osuma kelpaa, 0 = ei löytynyt

- kutsu `etsi_jokin( juuri, x )`

```
1   solmu *etsi_jokin( solmu *s1, avaintyyppi x ){
2     while( s1 && s1->x != x ){
3       if( s1->x < x ){ s1 = s1->oikea; }else{ s1 = s1->vasen; }
4     }
5     return s1;
6   }
```

Ensimmäisen yhtäsuuren tai lähinnä suuremman etsintä avaimen perusteella

```
1   solmu *etsi_eka( solmu *s1, avaintyyppi x ){
2     solmu *s2 = 0;
3     while( s1 ){
4       if( s1->x < x ){ s1 = s1->oikea; }
5       else{ s2 = s1; s1 = s1->vasen; }
6     }
7     return s2;
8   }
```



Lisääminen avaimen mukaiseen paikkaan, yhtäsuurista ensimmäiseksi

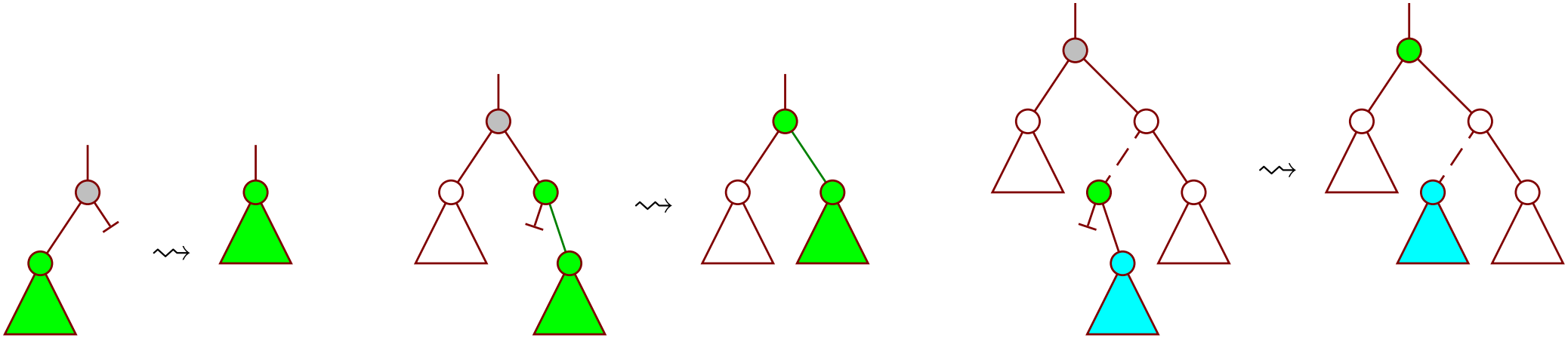
- `uusi` on solmutietue, jolle `uusi->vasen` $=$ `uusi->oikea` $= 0$

```
1   void lisaa( solmu *& juuri, solmu *uusi ){
2     if( !juuri ){ juuri = uusi; uusi->vanh = 0; return; }
3     solmu *s1 = juuri;
4     while( true ){
5       if( s1->x < uusi->x ){
6         if( s1->oikea ){ s1 = s1->oikea; }
7         else{ s1->oikea = uusi; uusi->vanh = s1; return; }
8       }else{
9         if( s1->vasen ){ s1 = s1->vasen; }
10        else{ s1->vasen = uusi; uusi->vanh = s1; return; }
11      }
12    }
13  }
```

- `*& juuri` saa aikaan, että kutsujan `juuri` muuttuu tarvittaessa rivillä 2

Poistaminen jakautuu eri tapauksiin

- jos poistettavan solmun jompikumpi lapsi puuttuu, niin poistettava solmu linkitetään pois vanhempansa ja ainoan lapsensa (jos on) välistä
- jos poistettavan solmun oikean lapsen vasen lapsi puuttuu, niin oikea lapsi linkitetään poistettavan solmun tilalle
- muussa tapauksessa linkitetään suuruusjärjestyksessä seuraava solmu pois vanhempansa ja ainoan lapsensa (jos on) välistä ja poistettavan solmun tilalle

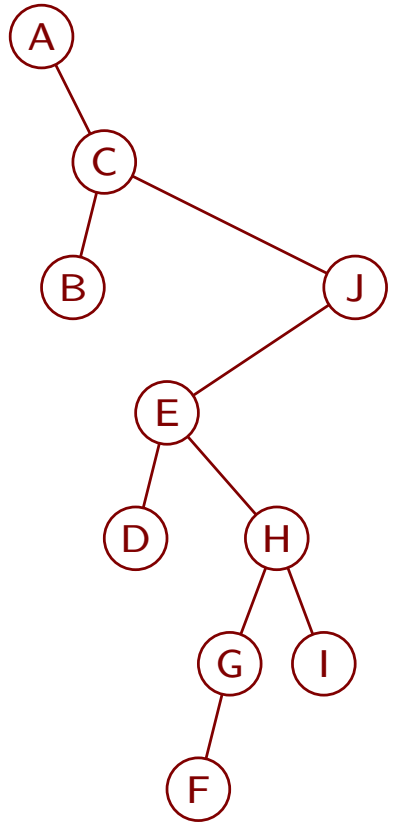


Jos solmuja lisätään suunnilleen suuruusjärjestyksessä, niin pienelläkin määrällä solmuja voi tulla kovin korkea puu

$\Rightarrow$ toiminnoista tulee hitaita

Tasapainottaminen

- sen varmistaminen, että puun korkeus on korkeintaan vakio kertaa $\log n$
  - tehdään (ainakin) solmu lisättäessä ja solmu poistettaessa
  - sen ansiosta lisäys, poisto ja avaimella etsintä saadaan ajassa $O(\log n)$
- on kehitetty monia tasapainottamismenetelmiä
- luvussa 9.3 esitellään eräs suosituimmista



Tasapainotetuilla binäärihakupuilla voidaan toteuttaa monipuolinen tietovarasto tehokkaasti

- tuotettu palvelu tunnetaan ainakin nimillä sorted map ja sorted associative array
- tallentaa avaimen ja muun hyötykuorman muodostamia pareja
- avain voi olla monimutkaista tietotyyppiä, vaikka merkkijono
- tarjoaa lisäyksen, poiston, etsinnän avaimen perusteella sekä ensimmäisen, viimeisen, seuraavan ja edellisen avainten järjestyksen perusteella
- tarjolla monissa ohjelmointikielissä
  - esim. seuraava C++-koodinpätkä tulostaa 4:

```
1  std::map< std::string, unsigned > sanalaskuri;
2  sanalaskuri[ "hauva" ] = 3; ++sanalaskuri[ "hauva" ];
3  std::cout << sanalaskuri[ "hauva" ] << "\n";
```

- kaikki edellä mainitut saadaan toimimaan ajassa $O(\log n)$
  - lisäksi koko rakenteen selaus avainten järjestyksessä ajassa $O(n)$
- tärkeimmät erot hajautustauluun
  - hajautustaulu ei tarjoa tehokkaita avainten järjestykseen perustuvia toimintoja
  - hajautustaulu tarjoaa muut mainitut toiminnot käytännössä nopeammin

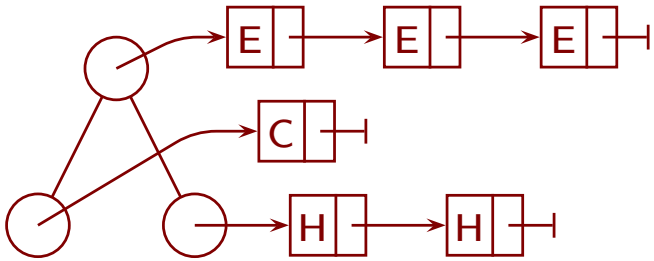
Esimerkki: sanojen esiintymien määrät laskeva tehokas ohjelma kokonaisuudessaan

```
1   #include <iostream>
2   #include <string>
3   #include <map>
4   int main(){
5     std::map< std::string, unsigned > sanalaskuri;
6     while( true ){
7       std::string sana; std::cin >> sana;
8       if( !std::cin ){ break; }
9       ++sanalaskuri[ sana ];
10    }
11    for( const auto & alkio : sanalaskuri ){
12      std::cout.width(4);
13      std::cout << alkio.second << " " << alkio.first << "\n";
14    }
15  }
```

- rivit 11, . . . , 14 selaavat `sanalaskuri`:n alkiot
  - `auto` hakee tyypin `sanalaskuri`:n tyypistä
- `.width(4)` asettaa seuraavan tulostuskentän leveydeksi (ainakin) 4
- `.first` ja `.second` ovat avain ja muu hyötykuorma
- (sanojen erottimena on tässä vain valkoinen tila, mikä ei riitä kaikkiin tarpeisiin)

Tasapainotetuilla binäärihakupuilla voidaan järjestää vakaasti ajassa $O(n \log h)$

- tässä h on avainten eri arvojen määrä
- binäärihakupuun solmussa ei ole yhtä alkiota, vaan linkitetty lista alkioita, joilla on sama avain
- lisätään taulukon alkiot binäärihakupuuhun ajassa $O(n \log h)$
  - jos alkio on jo puussa, niin lisätään listan alkuun
  - muussa tapauksessa luodaan puuhun uusi solmu ja sille yhden alkion lista
- käydään binäärihakupuu läpi takaperin ja kopioidaan listojen sisällöt taulukkoon lopusta alkaen ajassa $O(n)$



9.3 Punamustat puut

Binäärihakupuu on ***punamusta puu*** (***red-black tree***), jos ja vain jos:

- Jokainen solmu on joko punainen tai musta.
- Juuri on musta.
- ***Punaisten sääntö:*** Jos punaisella solmulla on lapsia, niin ne ovat mustia.
- ***Mustien sääntö:*** Jokainen juuresta alas ei minnekään -osoittimeen vievä polku sisältää saman määrän mustia solmuja.

Seurauksia

- punaisella solmulla on nolla tai kaksi lasta
- jos solmulla on täsmälleen yksi lapsi, niin se on punainen ja solmu itse on musta

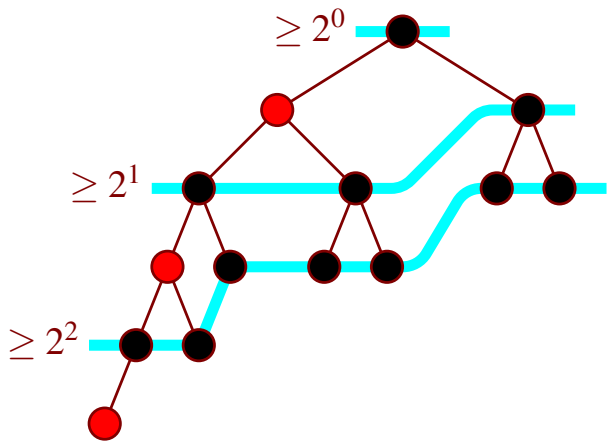
Punamustan puun korkeus on enintään $2\log_2 n$

- olkoon k korkeus ja n solmujen määrä
- pisimmällä polulla alas ei minnekään -osoittimeen
  - on kaikkiaan $k+1$ solmua
  - mustien solmujen määrä $\geq$ punaisten solmujen määrä

  $\Rightarrow$ mustien solmujen määrä on vähintään $\lceil \frac{k+1}{2} \rceil$

$\Rightarrow$ mustia solmuja on kaikkiaan vähintään $2^{\lceil \frac{k+1}{2} \rceil} - 1 \geq 2^{\frac{k}{2}}$

$\Rightarrow n \geq 2^{\frac{k}{2}} \Rightarrow k \leq 2\log_2 n$

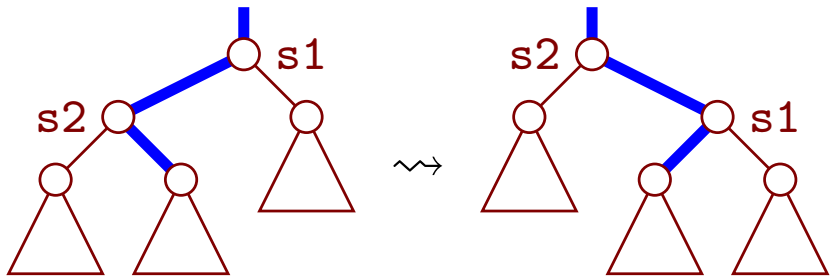


Perustoiminnot

- etsinnät ja selaus kuten binäärihakupuille
  - etsinnän ajan kulutus on $O(\log n)$, koska korkeus on $\Theta(\log n)$
- lisäys $O(\log n)$ ja jotakin poistosta $O(\log n)$ kerrotaan kohta
- toisin kuin usein kirjallisuudessa, emme käytä päätemerkkejä

Tasapainottamisen perustoiminto on ***kierto*** (***rotation***)

- kierto oikealle




```
1   s2 = s1->vasen; s1->vasen = s2->oikea;
2   if( s1->vasen ){ s1->vasen->vanh = s1; }
3   s2->vanh = s1->vanh; s2->oikea = s1; s1->vanh = s2;
4   if( !s2->vanh ){ juuri = s2; }
5   else if( s2->vanh->vasen == s1 ){ s2->vanh->vasen = s2; }
6   else{ s2->vanh->oikea = s2; }
```

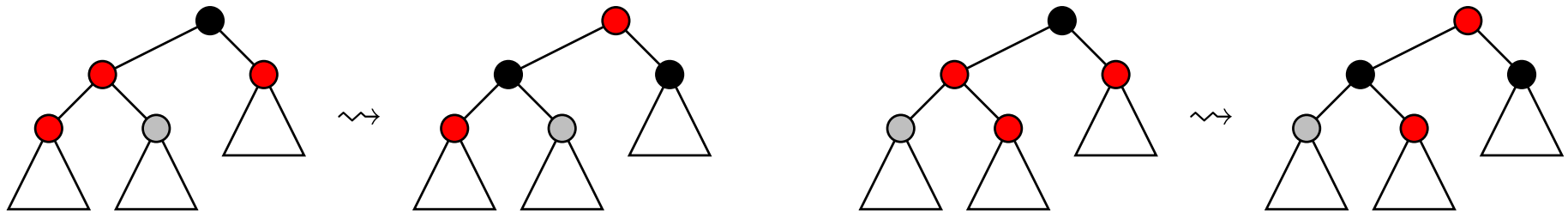
- kierto vasemmalle on symmetrinen

Solmun lisääminen ajassa $O(\log n)$

- solmu lisätään kuten edellä
- jos se on juuri, niin se väritetään mustaksi ja lopetetaan
- muussa tapauksessa se väritetään punaiseksi
- jos sen vanhempi on musta, niin lopetetaan
- muussa tapauksessa on kaksi punaista solmua päällekkäin
- ongelmaa siirretään ylöspäin kuten kohta kerrotaan, kunnes se katoaa tai saavuttaa juuren
- jos se saavuttaa juuren, niin juuri väritetään mustaksi
- kun ongelma ei ole juuressa, on solmun vanhemman vanhempi olemassa ja musta
- käsittelemme tapaukset, joissa solmun vanhempi on oman vanhempansa vasen lapsi
  - tapaukset joissa se on oikea lapsi ovat symmetriset
- kuvissa harmaa solmu tarkoittaa mustaa solmua tai ei lainkaan solmua

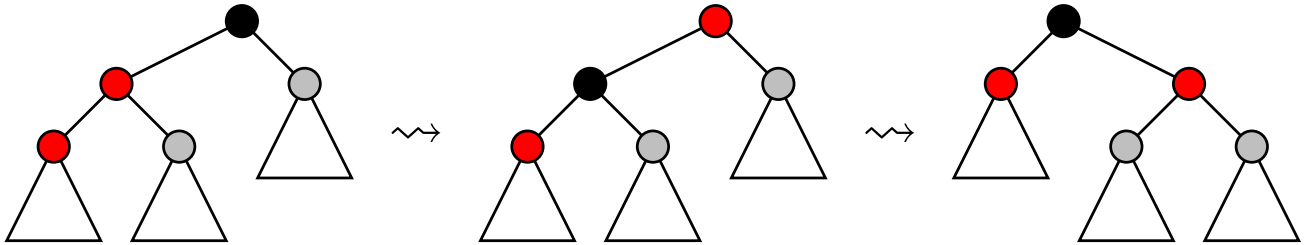
Tapaus 1: solmun vanhemman sisarus on punainen

- värittämällä kolme solmua ongelma siirtyy kaksi askelta ylöspäin tai katoaa



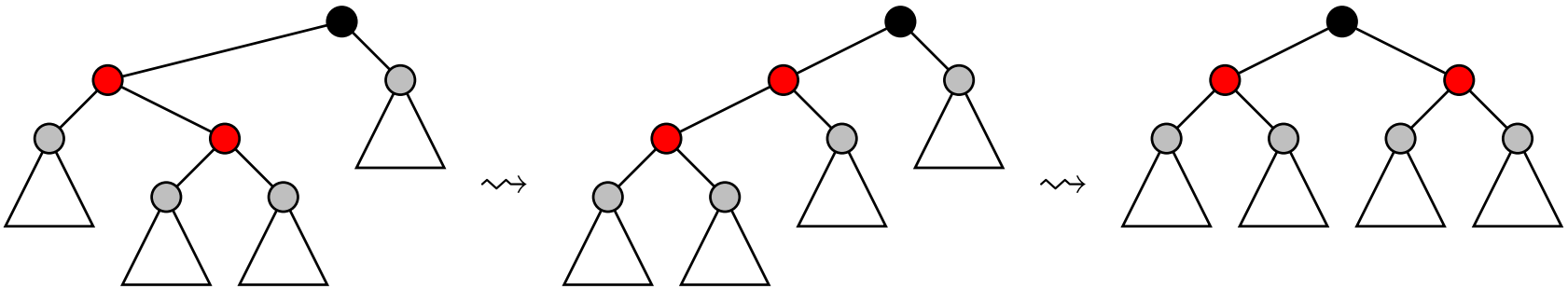
Tapaus 2: solmun vanhemman sisarus puuttuu tai
on musta, ja solmu on vanhempansa vasen lapsi

- punaisten solmujen säännön vuoksi vanhemman oikea lapsi puuttuu tai on musta
- värittämällä kaksi solmua ja tekemällä kierto oikealle ongelma katoaa



Tapaus 3: solmun vanhemman sisarus puuttuu tai
on musta, ja solmu on vanhempansa oikea lapsi

- punaisten solmujen säännön vuoksi punaisten lapset puuttuvat tai ovat mustia
- tekemällä kierto vasemmalle tilanne palautuu edelliseen



Poistaminen ajassa $O(\log n)$ käyttää samoja ideoita, mutta on vielä monimutkaisempaa

- solmu poistetaan kuten luvussa 9.2
- jos poistettu solmu oli musta, niin värejä korjataan maalaamalla ja kierroilla, kunnes ongelma katoaa tai saavuttaa juuren
- kiertoja tarvitaan enintään kolme

9.4 Kuinka mones ja valitse näin mones

Ylläpitämällä jokaisessa solmussa siitä alkavan alipuun kokoa, saadaan ajassa $O(\log n)$:

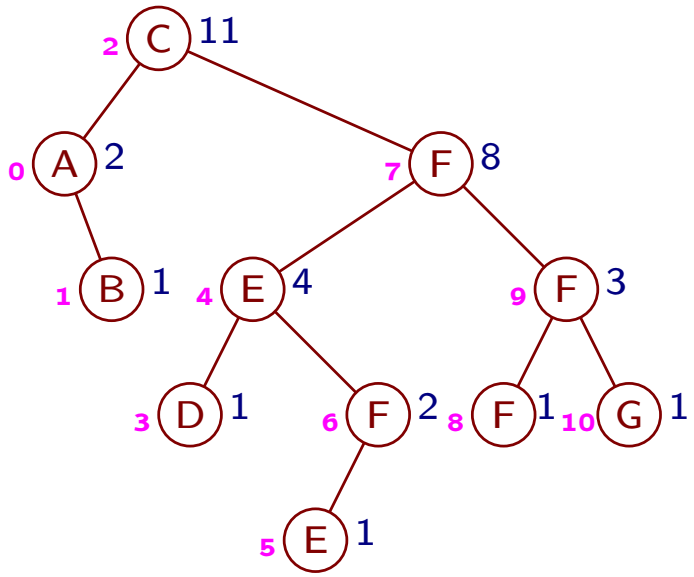
- solmun löytäminen järjestysluvun perusteella
  - saa kokonaisluvun i
  - palauttaa osoittimen siihen solmuun, jota ennen on i solmua
  - muistuttaa taulukon indeksointia $A[i]$ luvulla eikä avaimella, palauttaa osoittimen
- solmun järjestysluvun selvittäminen
  - saa osoittimen solmuun ja palauttaa tiedon, montako solmua on sitä ennen
  - vastaa sitä, että on osoitin $A[i]$:hin ja palauttaa i:n

Käyttöesimerkki: suunnistuskilpailu

- suunnistajat lähtevät eri aikoina
- kun suunnistaja tulee maaliin, hän on aikansa perusteella esimerkiksi kolmas
- hänen sijoituksensa voi huonontua, kun joku muu tulee maaliin
- hänen sijoituksensa paranee aina kun joku edellä ollut hylätään
  - kävi ilmi, että edellä ollut oli oikaissut kielletyn alueen kautta
- ”monesko hän on nyt”: vastaus saadaan järjestysluku selvittämällä plus yksi
- ”kuka on nyt viidentenä”: etsitään solmu järjestysluvun miinus yksi perusteella
- jaetut sijat saadaan selville edeltäjä- ja seuraaja -toiminnoilla

Solmun löytäminen järjestysluvun perusteella ajassa $O(k)$

```
1  solmu *valitse_kohdasta_i( solmu *s1, unsigned ii ){
2    while( s1 ){
3      unsigned mm = s1->vasen ? mm = s1->vasen->koko : 0;
4      if( ii == mm ){ return s1; }
5      if( ii < mm ){ s1 = s1->vasen; }
6      else{ ii = ii-mm-1; s1 = s1->oikea; }
7    }
8    return 0;
9  }
```



- kutsutaan `valitse_kohdasta_i( juuri, i )`

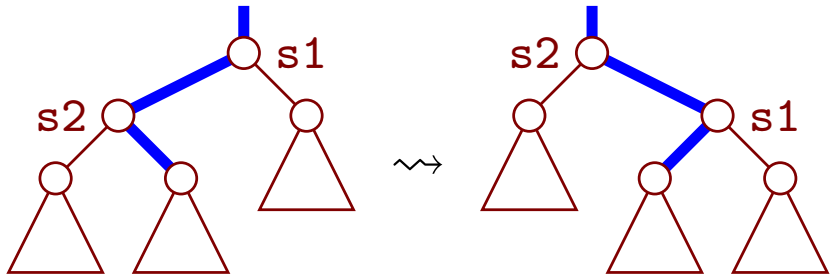
Solmun järjestysluvun selvittäminen ajassa $O(k)$

```
1  unsigned kuinka_mones( solmu *s1 ){
2    unsigned mm = 0;
3    if( s1->vasen ){ mm += s1->vasen->koko; }
4    while( s1->vanh ){
5      if( s1 == s1->vanh->oikea ){ mm += 1 + s1->vanh->vasen->koko; }
6      s1 == s1->vanh;
7    }
8    return mm;
9  }
```

Kokotietojen ylläpito ajassa $O(k)$

- kun solmu on lisätty, suoritetaan `while( s1 ){ ++s1->koko; s1 = s1->vanh; }`
  - vaihtoehtoisesti kasvatukset voi tehdä jo lisäyspaikkaa etsittäessä
  - lisätyn solmun kooksi asetetaan 1
- ennen poistamista
  - poistettavan solmun tilalle linkitettävän solmun alkuperäisestä vanhemmasta alkaen suoritetaan `while( s1 ){ --s1->koko; s1 = s1->vanh; }`
  - kopioidaan poistettavan solmun koko sen tilalle linkitettävän solmun kooksi
- kierron jälkeen päivitetään kiertosolmujen koot (`s1` oli ylempi ennen kiertoa)
  `s2->koko = s1->koko; paivita_koko( s1 );`

```
1   void paivita_koko( solmu* s1 ){
2     s1->koko = 1;
3     if( s1->vasen ){ s1->koko += s1->vasen->koko; }
4     if( s1->oikea ){ s1->koko += s1->oikea->koko; }
5   }
```



10 Välitulosten muistaminen

11 Lisää graafialgoritmeja

Kysymysten vastauksia

PoisKeosta(&A)

1 $h := A.koko - 1;\ i := 0;\ j := 1$
2 **while** true **do**
3 **if** $j+1 < h$ && $A[j+1].x \geq A[j].x$ **then** $j := j+1$
4 **if** $j \geq h$ || $A[j].x \leq A[h].x$ **then break**
5 $A[i] := A[j];\ i := j;\ j := 2i+1$
6 $A[i] := A[h];\ A.kooksi(h)$

```
1  unsigned tuplaa( unsigned alue, unsigned eksp ){
2    if( eksp >= maks_eksp ){ return ~0u; }
3    unsigned uusi = vapaat[ eksp+1 ];
4    if( uusi != ~0u ){ vapaat[ eksp+1 ] = muisti[ uusi ]; }
5    else if( (1u << (eksp+1)) > kaikki - loppu ){ return ~0u; }
6    else{ uusi = loppu; loppu += (1u << (eksp+1)); }
7    for( unsigned i = 0; i < (1u << eksp); ++i ){
8      muisti[ uusi + i ] = muisti[ alue + i ];
9    }
10   muisti[ alue ] = vapaat[ eksp ]; vapaat[ eksp ] = alue;
11   return uusi;
12 }
```

```
TulostaReitti(i, maali)
tulosta i
while i ≠ maali do
    j := R[i, maali]
    tulosta " "M[i, j] "km "j
    i := j
```

```
jonoon(i)
if S[i].m = false then
    J[v] := i; v := (v + 1) mod (n + 1); S[i].m := true
```

```
jonosta()
i := J[w]; w := (w + 1) mod (n + 1); S[i].m := false; return i
```

```
1   for i := 0 to n − 1 do
2       S[i].e := ääretön; J[i] := i; S[i].m := true
3   S[lähtö].e := 0; w := 0; v := n; kierros := 1
4   while kierros ≤ n && w ≠ v do
5       kierros := kierros + 1; u := v
6       while w ≠ u do
7           h := jonosta()
8           if h = 0 then j := 0 else j := S[h − 1].ℓ
9           while j < S[h].ℓ do
10              d := S[h].e + K[j].p; k := K[j].k; j := j + 1
11              if d < S[k].e then
12                  S[k].e := d; S[k].r := h; jonoon(k)
```

```
QS3(&A,a,y)
if a ≥ y then return
x := A[Random(a,y)].x
j := a; k := a
for i := a to y do
    if A[i].x ≤ x then
        apu := A[i]; A[i] := A[j]
        if apu.x = x then A[j] := apu
        else A[j] := A[k]; A[k] := apu; k := k+1
        j := j+1
QS3(A,a,k−1); QS3(A,j,y)
```

```
for i := 0 to A.koko−2 do
    if N[i] ≠ i then
        apu := A[i]; j := i; k := N[j]
        while k ≠ i do
            A[j] := A[k]; j := k; k := N[j]; N[j] := j
        A[j] := apu
```

QUICKSELECT($\&A, k$)

1 **if** $k < 0 \mid\mid k \geq A.koko$ **return** virhe
2 **while** true **do**
... kuvan 5.2 rivit 2, ..., 9
11 **if** $k < i$ **then** $y := i - 1$
12 **else if** $k > j$ **then** $a := j + 1$
13 **else return** $A[i]$

```
1  alkio *poista(){
2    if( !viim ){ return 0; }
3    alkio *vanhin = viim->seur;
4    if( vanhin == viim ){ viim = 0; }
5    else{ viim->seur = vanhin->seur; }
6    return vanhin;
7  }
```

```
1  unsigned i1 = maali, i2 = ~0u;
2  while( i2 != lahto ){
3    unsigned i3 = i2;
4    i2 = i1; i1 = solmut[ i1 ].reitti; solmut[ i2 ].reitti = i3;
5  }
```

```
1   void countingsort( taulukko & A ){
2     const unsigned k = 256;
3     taulukko B( A.size() ); unsigned L[k] = {};
4     for( unsigned i = 0; i < A.size(); ++i ){ ++L[ A[i].x ]; }
5     for( unsigned j = 1; j < k; ++j ){ L[j] += L[ j-1 ]; }
6     for( unsigned i = A.size(); i--; ){ B[ --L[ A[i].x ] ] = A[i]; }
7     A.swap(B);
8   }
```

```
1   unsigned tulos = 0;
2   for( unsigned ii = 0; ii < 3; ++ii ){
3     tulos *= 29;
4     if( R[ii] != ' ' ){ tulos += R[ii] - 'A'+ 1; }
5   }
6   --tulos;
7   for( unsigned ii = 3; ii < 6; ++ii ){
8     tulos *= 10;
9     if( R[ii] != ' ' ){ tulos += R[ii] - '0'; }
10  }
```