

TIES4211 Algoritmit 2

Antti Valmari

Jyväskylän yliopisto

1	Johdanto	2
2	Silmukkainvariantit ja lisäysjärjestäminen	20
3	Kekojärjestäminen ja prioriteettijono	41
4	Sovellusesimerkki: reitin etsintä	76
5	Lisää graafialgoritmeja	115
6	Ahneet algoritmit ja dynaaminen ohjelmointi	160
7	Pikajärjestäminen ja mediaanin etsiminen	177
8	Hieman linkitetyistä listoista	205
9	Lisää järjestämisalgoritmeista	214
10	Hajautustaulut	232
11	Binääripuut	250
	Kysymysten vastauksia	273

1 Johdanto

1.1	Tästä tekstistä	3
1.2	Arvosanan laskemisen testaaminen tietokoneella	6
1.3	Puolitushaku	10
1.4	Puolitushakualiohjelman testaaminen tietokoneella	12
1.5	Toimiva puolitushaku	17

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 tunsikin 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 koordinoita 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, ..., 10

```
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 suunnittelemisen olisi vaikeaa

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

Luotettavia ohjelmia saa aikaan vain testaamisen ja mieltimisen 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 loppupuolikkaalla

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:ssä 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, ..., 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

⇒ 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,...,8,...,8]

Virheellinen puolitushakumme jäi ikuisen 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 $\leadsto$ 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	...
<i>ala</i>	0			0			0	...
<i>vali</i>			0			0		...
<i>yla</i>	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

⇒ 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 puolitusshaku

Olisi toisinaan hyödyllistä, jos

- jos avain on taulukossa, niin puolitusshaku 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?

⇒ 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 ($i-1, i-2, \dots$ eivät voi olla oikein): $i = 0$ tai $A[i-1] < \text{avain}$
3. i ei ole liian pieni ($i+1, i+2, \dots$ eivät voi olla oikein): $i = n$ tai $A[i] \geq \text{avain}$

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

(*): ($ala = 0$ 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:iin
- koska $ala < yla$ rivillä 3, pätee $ala < \frac{ala+yla}{2} < yla$ ja $ala \leq \lfloor \frac{ala+yla}{2} \rfloor < yla$
- 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 $i = ala$

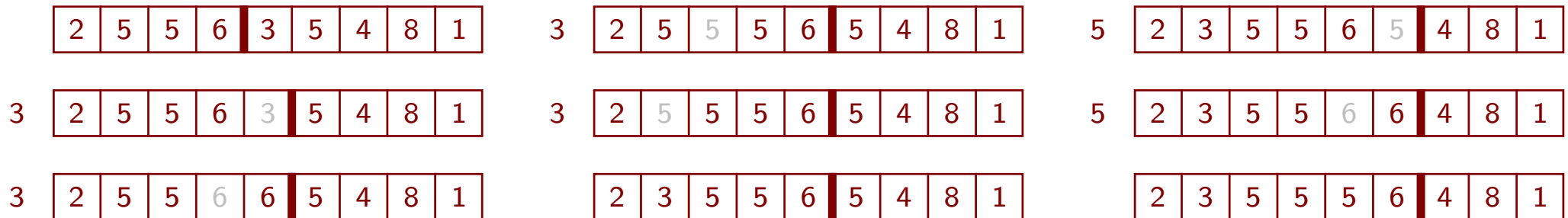
- ainoan indeksoinnin `A[vali]` laillisuus seuraa suoraan siitä, että $0 \leq ala \leq vali < yla \leq n$
- rivillä 1 on ylivuodon vaara, jos `A.size()` ei mahdu `int`-tyyppiin
- rivillä 7 ei ole ylivuotovaaraa, koska $vali < yla$
- rivillä 3 on Blochin ylivuotovaara
 - korjaantuu muuttamalla rivi muotoon `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	21
2.2	Todistuksen jakaminen tapauksiin ja tapaus $n = 0$	22
2.3	Ulompi silmukka paitsi sen runko	23
2.4	Ulomman silmukan lopettaminen	24
2.5	Ei laittomia toimintoja	25
2.6	Ulomman silmukan runko	26
2.7	Vakaus	29
2.8	O -, Ω - ja Θ -merkinnät	31
2.9	Lisäysjärjestämisen suoritus aika	35
2.10	Muistin tarve	39
2.11	Lisää kysymyksiä	40

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       $apu := A[i]; j := i$ 
3      while  $j > 0 \ \&\& \ A[j-1].x > apu.x$  do
4           $A[j] := A[j-1]; j = j - 1$ 
5       $A[j] := apu$ 

```

```

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 $n = 0$

⇒ 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 $A[a \dots y]$, missä $y \geq a - 1$, osataulukkoa $A[a]$, $A[a + 1]$, $\dots$, $A[y]$

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

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

Sytä epäonnistua loppuun pääsemisessä

- ikuinen silmukka
- kaatuminen suoritusajalliseen virheeseen

Aito **for**-silmukka (perusversio)

for $i := \text{alku}$ **to** loppu **do** runko

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

⇒ enintään $\text{MAX}(\text{loppu} - \text{alku} + 1, 0)$ 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

Taulukoiden indeksoinnit

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

⇒ siellä pätee $0 \leq j \leq i < n$

⇒ jokainen $A[i]$ ja $A[j]$ on laillinen

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

⇒ nekin ovat lailliset

Muistin loppuminen

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

⇒ vaikea keksiä muuta järkevää kuin antaa ohjelman kaatua

Lukualueen ylivuodot

- riippuu muuttujien tyypeistä
- tarkastamme yksityiskohdat

⇒ tarpeeksi isolla kokonaislukutyypillä ei vaaraa

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

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       $A[j] := apu$ 
```

2.6 Ulomman silmukan runko

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

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

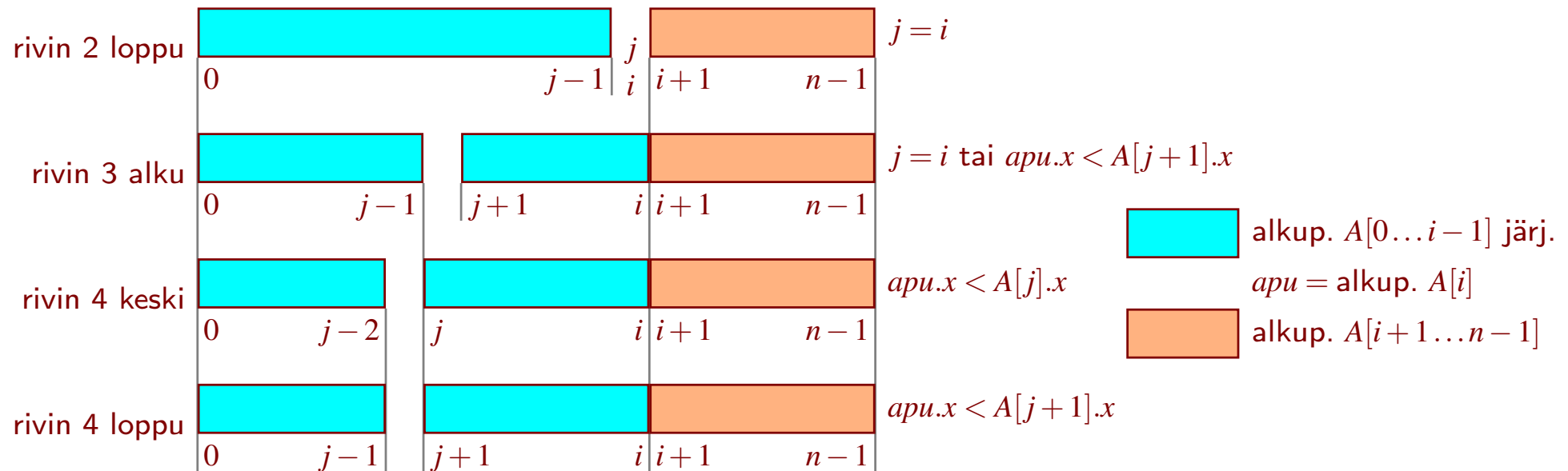
1. Osassa $A[0 \dots i]$ on alkuperäiset alkiot kasvavassa järjestyksessä.
2. Osassa $A[i + 1 \dots n - 1]$ on alkuperäiset alkiot alkuperäisessä järjestyksessä.

Jälkimmäinen säilyy, koska

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

Sisemmän silmukan invariantti:

1. Osissa $A[0 \dots j-1]$ ja $A[j+1 \dots i]$ on yhteensä osan $A[0 \dots i-1]$ 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. $A[j+1 \dots i]$ on tyhjä; $A[0 \dots j-1] = A[0 \dots i-1] = \text{alkup. } A[0 \dots i-1]$ kasvavasti
2. $apu = A[i] = \text{alkup. } A[i]$ $\Uparrow$ ulompi inv. 1
3. $j = i$ $\Uparrow$ ulompi inv. 2

Säilyy rungossa:

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

Toteuttaa halutun: rivin 5 jälkeen

- $A[0 \dots i]$:ssä on alkuperäiset alkiot (järjestys voi olla muuttunut)
 - I1 $\Rightarrow A[0 \dots j-1]$ ja $A[j+1, \dots, i]$ ovat yhdessä $A[0 \dots i-1]$:n alkuperäiset
 - $A[j] = \text{apu} =$ alkuperäinen $A[i]$ (I2)
- järjestys on kasvava
 - $j = 0$ tai $A[j-1].x \leq A[j].x$ silmukan lopetusehdon vuoksi ja koska $A[j] = \text{apu}$
 - $j = i$ tai $A[j].x < A[j+1].x$ I3:n vuoksi ja koska $A[j] = \text{apu}$
 - 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       $\text{apu} := A[i]; j := i$ 
3      while  $j > 0 \ \&\& \ A[j-1].x > \text{apu}.x$  do
4           $A[j] := A[j-1]; j = j - 1$ 
5       $A[j] := \text{apu}$ 
```

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

⇒ voidaan järjestää usean kriteerin mukaan aloittamalla viimeisenä sovellettavasta

⇒ 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 tulokorkeuden 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       $apu := A[i]; j := i$ 
3      while  $j > 0 \ \&\& \ A[j-1].x > apu.x$  do
4           $A[j] := A[j-1]; j = j - 1$ 
5       $A[j] := apu$ 
```

2.8 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

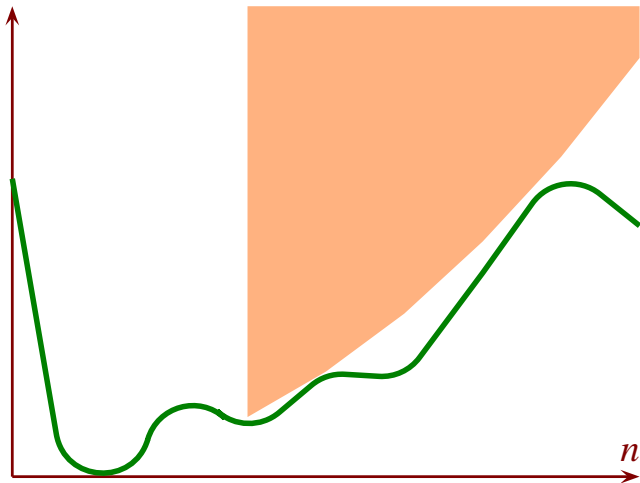
Ohjelman suoritusaikaan vaikuttaa moni muukin asia

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

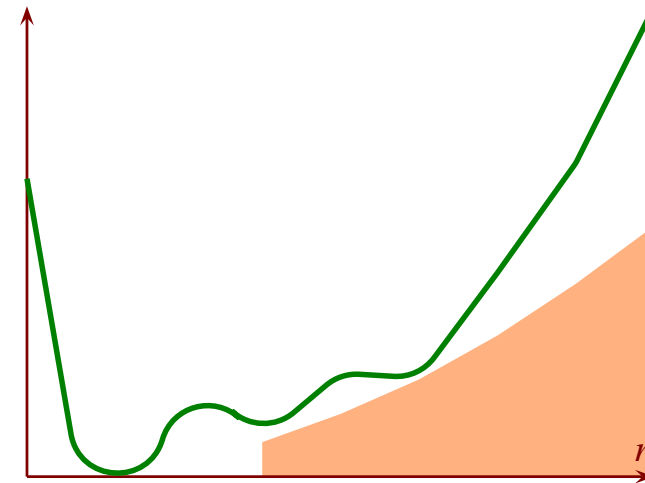
Suoritusajan kasvukäyrän muotoa kuvataan O -, Ω - ja Θ -merkinnöillä

- 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

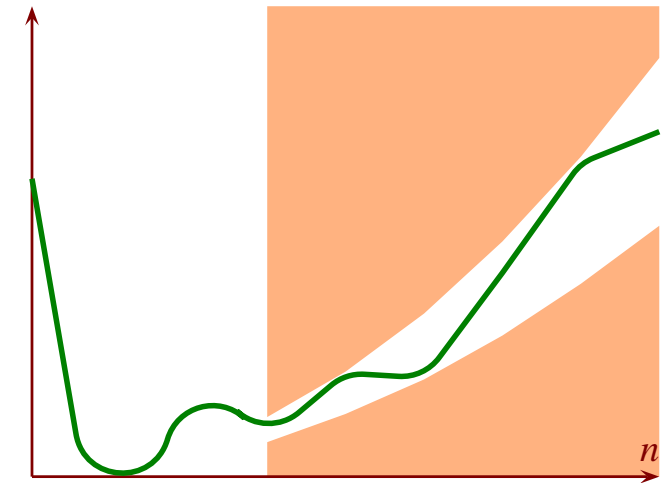
$O(\dots)$: enintään niin paljon



$\Omega(\dots)$: ainakin niin paljon



$\Theta(\dots)$: 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"

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

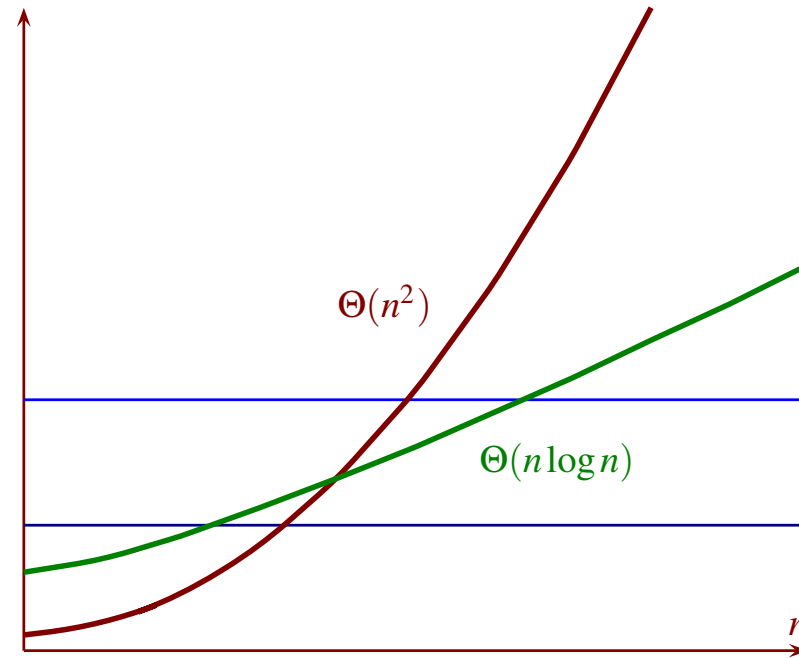
- toisin kuin mittaukset, ne ovat käytettävissä ennen kuin mitään on toteutettu

⇒ voidaan käyttää kun valitaan, mikä vaihtoehto toteutetaan

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

⇒ tyypillisesti ei tarvita tietoa mitä tapahtuu pienillä syötteillä

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



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

Käyrän muoto voi riippua syötteen laadusta

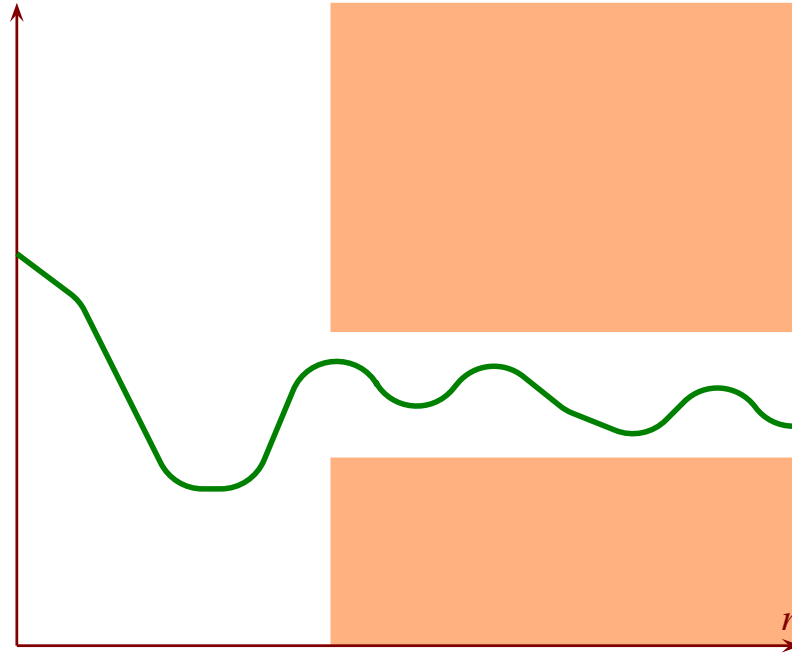
- jos tosielämän syöte on erilaatuista kuin oletettiin, Θ - jne. voivat antaa väärän kuvan
- jos tosielämän syöte on erilaatuista kuin mittauksissa, mittaukset ...
- tavallisesti keskitytään Θ -merkintään hitaimmassa tapauksessa
- jos se on riittävän hyvä, niin suoritus aika on riittävän hyvä kaikenlaisilla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

Ajan käytön näkökulmasta lisäysjärjestämisessä on seuraavat:

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

Alkioiden sijoituksen ja vertaamisen ajan kulutus riippuu alkioden koosta

- silti ajatellaan olevan $\Theta(1)$, koska
 - ei riipu n :stä
 - ei muutu suorituksen aikana
 - luvussa 7.3 nähdään, että suoritus aika voi riippua olennaisesti alkioden koosta
 - jos halutaan, niin voidaan ottaa käyttöön vaikka k ilmaisemaan alkion kokoa $\Rightarrow$ alkioden 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 9.4 (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 $\leq 2 + \dots + n \leq (n - 1)n \leq n^2$ kertaa
 - kukin muu rivi enintään $\text{MAX}(n + 1, n^2)$ 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 yläkiarvo, ja todellinen nopeus olisi parempi?
- jos A on takaperin järjestyksessä, niin kukin alkio siirretään i askelta
 - yhteensä $1 + 2 + \dots + n - 1 = \frac{1}{2}n^2 - \frac{1}{2}$ 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 $A[j-1].x > \text{apu}.x$ ei koskaan toteudu $\Rightarrow$ rivi 3 suoritetaan vain kerran kullekin i , eikä riviä 4 suoriteta kertaakaan

Siis INSERTIONSORT:n suoritusaika vaihtelee välillä $\Theta(n), \dots, \Theta(n^2)$

Mikä on suoritus aika 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 suoritus aika on $\Theta(n^2)$**
 - kukin alkio siirtyy keskimäärin puolet A:n jo järjestetyn osan koosta $\Rightarrow$ suoritus aika suunnilleen puolet hitaimman tapauksen suoritus ajasta

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

```
1 void SelectionSort( taulukko & A ){
2     for( unsigned i = 0; i+1 < A.size(); ++i ){
3         unsigned p = i;
4         for( unsigned j = i+1; j < A.size(); ++j ){
5             if( A[j].x < A[p].x ){ p = j; }
6         }
7         alkio apu = A[i]; A[i] = A[p]; A[p] = apu;
8     }
9 }
```


3 Kekojärjestäminen ja prioriteettijono

3.1	Keot	42
3.2	Kekoon lisääminen ja jonkin suurimman poistaminen	46
3.3	Kasvavat taulukot ja tasattu ajan kulutus	51
3.4	Hieman muistinhallinnasta	56
3.5	Kekoon lisäämisen ja poistamisen ajan kulutus	58
3.6	Kekojärjestäminen	61
3.7	Prioriteettijono	68

3.1 Keot

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

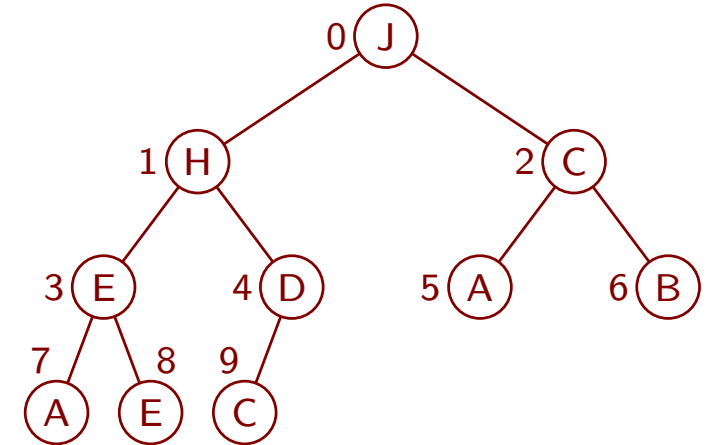
- muistialue, josta varataan muistia mm. `new`:lla luotaville olioille
- tietorakenne, jota käsitellään tässä luvussa

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 **tason**
- 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 ensimmäinen."

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 $\lfloor \dots \rfloor$

- 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 kokonaisluvulla n/m laskee $\lfloor \frac{n}{m} \rfloor$ eikä $\frac{n}{m}$, kun $n \geq 0$ ja $m > 0$
- mitä n/m laskee kun n tai m tai molemmat ovat negatiivisia vaihtelee eri kielissä
 - ks. Wikipedia "Modulo" $\rightsquigarrow$ "In programming languages"

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

Jokaisella i väliltä $1, \dots, n-1$ 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 $A[0 \dots n-1]$	$1 \leq i \leq n-1$	i	$\lfloor \frac{i-1}{2} \rfloor$
taulukolle $A[1 \dots n]$	$1 \leq i \leq n-1$	$i+1$	$\lfloor \frac{i-1}{2} \rfloor + 1$
$i-1$ 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	$\lfloor \frac{i}{2} \rfloor$

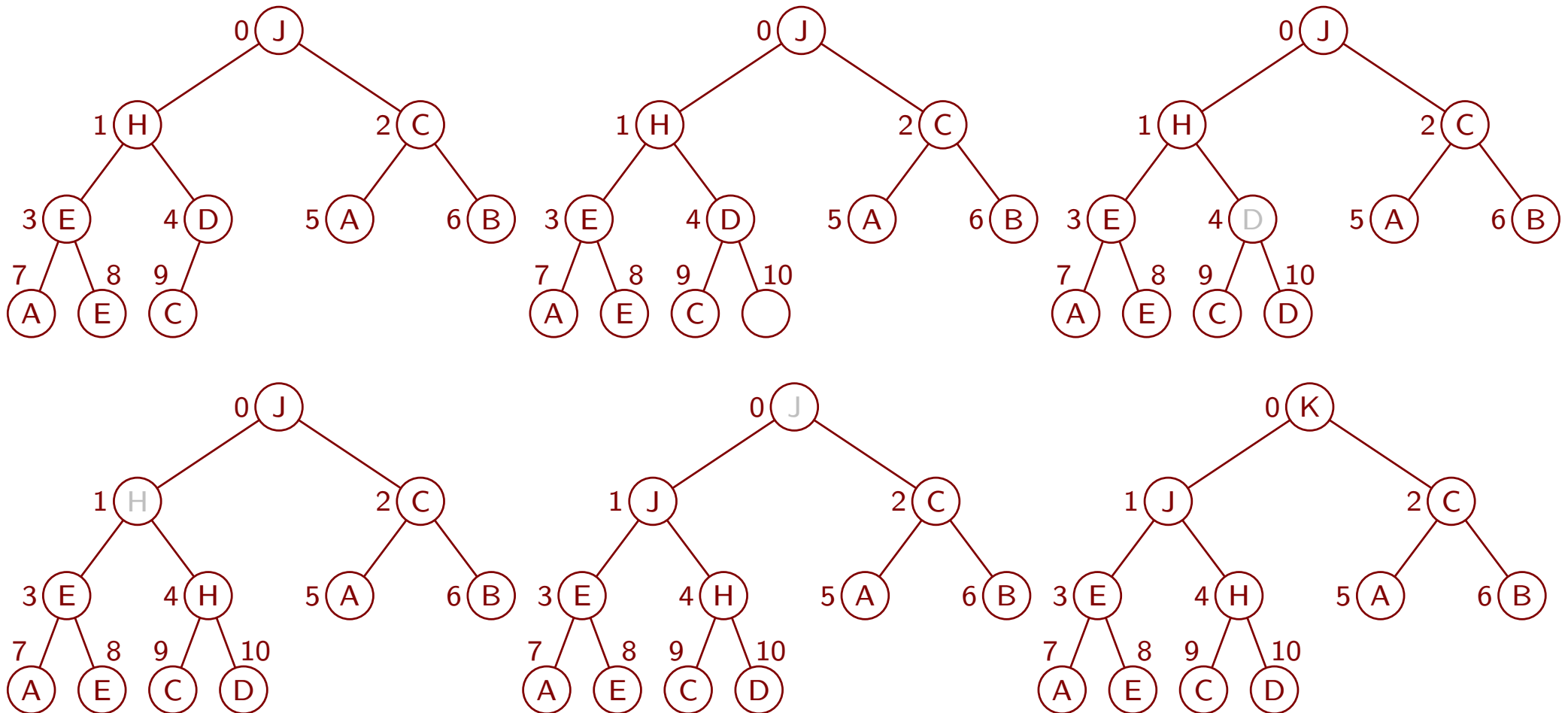
- päättelyvirheitä voi tapahtua, joten testaamme

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

```
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       $A[i] := A[j]; i := j; j := \lfloor \frac{i-1}{2} \rfloor$   
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 $A[0 \dots n-1]$ 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 $A[0 \dots i-1]$ ja $A[i+1 \dots n-1]$ on yhteensä alkuperäiset alkiot.
3. Jokaisella k väliltä $1, \dots, n-2$ 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 $A[0 \dots n-1] \rightsquigarrow A[0 \dots n-2]$ 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 alkioista, 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
 - $A[n - 1] = uusi$ $\Rightarrow A[0 \dots n - 2]$ on I3:n vuoksi keko, ja $uusi$ on oikealla paikallaan viimeisenä

```
3  while  $i > 0 \ \&\& \ A[j].x < uusi.x$  do
4       $A[i] := A[j]; i := j; j := \lfloor \frac{i-1}{2} \rfloor$ 
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 mainitsevat alkuperäiset alkiot ovat $A[0 \dots j - 1, j + 1 \dots n - 1]$, ja $i := j \rightsquigarrow A[0 \dots i - 1, i + 1 \dots n - 1]$
- 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

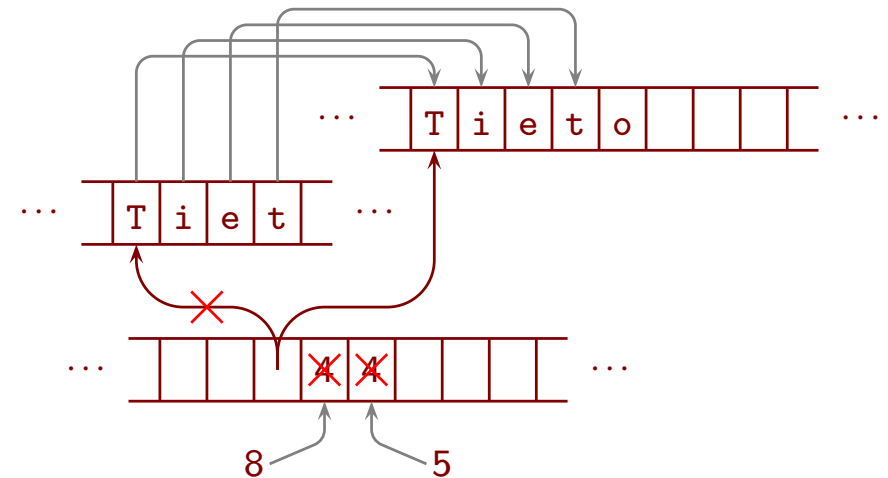
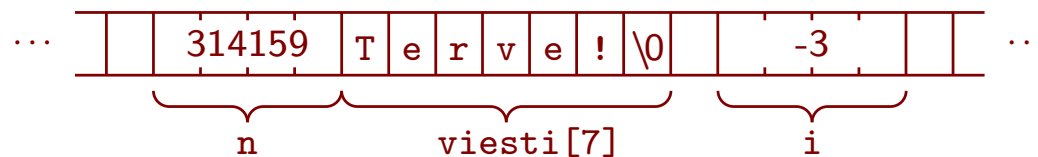
(Yhden) suurimman poistaminen

- 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 $n = 1$

⇒ enintään $(n - 1)$ ainakin kerran

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

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

...

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

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

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

$$1 = 2 - 1 = 2 - (\frac{1}{2})^0$$

$$1 + \frac{1}{2} = 2 - (\frac{1}{2})^0 + (\frac{1}{2})^1 = 2 - (\frac{1}{2})^1$$

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

$$1 + \frac{1}{2} + (\frac{1}{2})^2 + (\frac{1}{2})^3 = 2 - (\frac{1}{2})^2 + (\frac{1}{2})^3 = 2 - (\frac{1}{2})^3$$

$$1 + \frac{1}{2} + (\frac{1}{2})^2 + \dots + (\frac{1}{2})^{k-1} = 2 - (\frac{1}{2})^{k-2} + (\frac{1}{2})^{k-1} = 2 - (\frac{1}{2})^{k-1}$$

- 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

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

⇒ ei $\Theta(n^2)$, vaan $\Theta(n)$

Siis

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

⇒ lisäyksen ajan kulutus on **tasatusti** $O(1)$

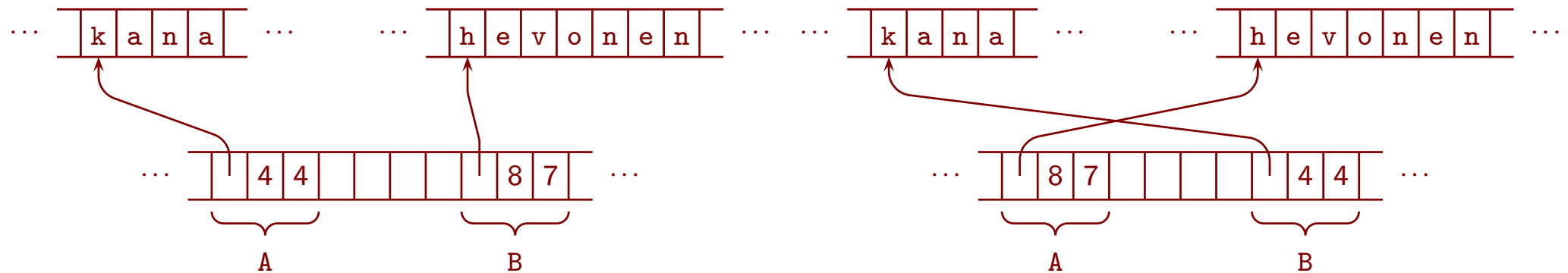
Ajan kulutuksen ja tasatun ajan kulutuksen ero on tärkeä ymmärtää!

- 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

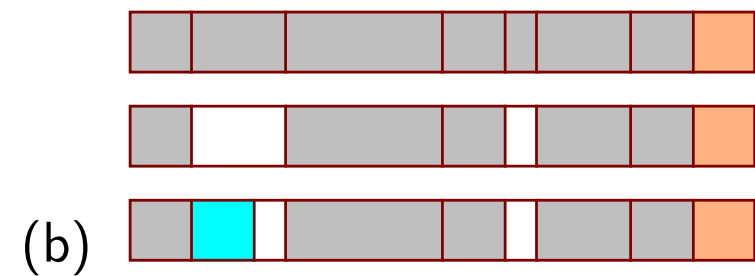
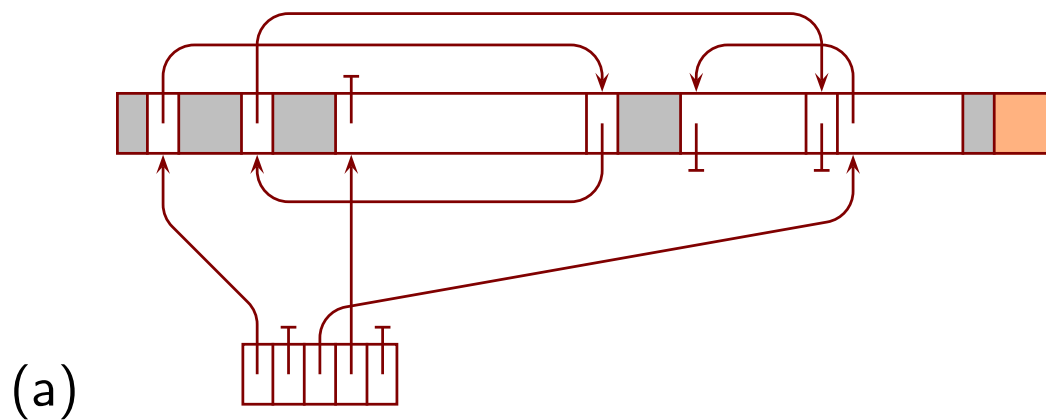
Kahden kasvavan taulukon sisältöjen vaihtaminen keskenään



3.4 Hieman muistinhallinnasta

Vapautetun muistin kierrättäminen tehokkaasti on vaikeaa

- käyttöjärjestelmä (ja ohjelmointikielen suoritussympäristö) 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
 - ⇒ 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 mutta 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
- ⇒ 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

Kuinka monta kertaa enintään silmukka voi kiertää?

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

⇒ 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 $\rightsquigarrow \dots \rightsquigarrow$ vajaa 1

⇒ 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 $O(\log n)$

LISÄÄKEKOON(&A, uusi)

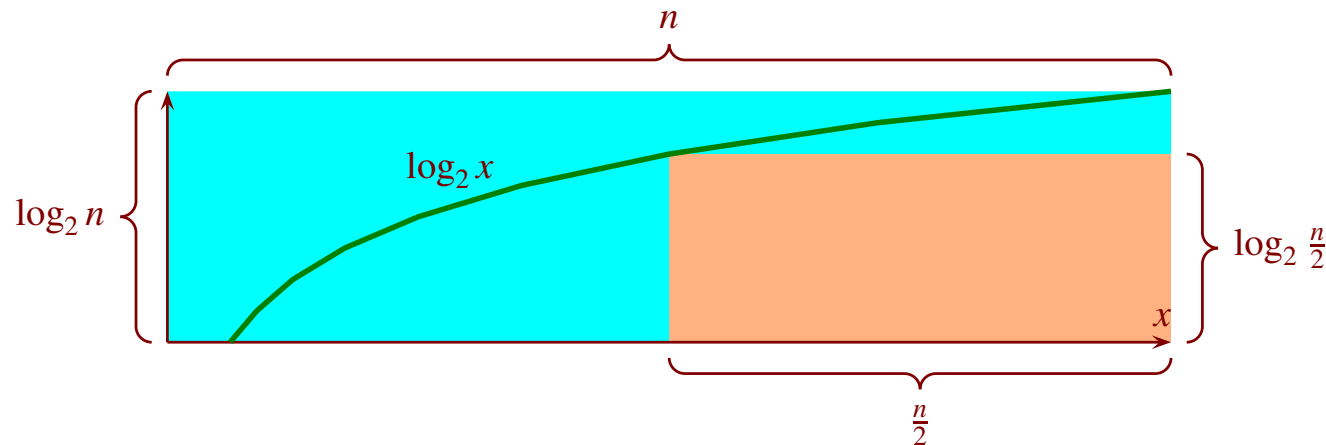
```
1  i := A.koko; j := ⌊ $\frac{i-1}{2}$ ⌋
2  A.kooksi(i+1)
3  while i > 0 && A[j].x < uusi.x do
4      A[i] := A[j]; i := j; j := ⌊ $\frac{i-1}{2}$ ⌋
5  A[i] := uusi
```

Kauanko kestää aloittaa tyhjällä keolla ja lisätä n alkioita?

- 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

$\Rightarrow \log_2 1 + \log_2 2 + \dots + \log_2 n$ 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 $\log_2 1 + \log_2 2 + \dots + \log_2 n > \frac{n}{2} \log_2 \frac{n}{2} = \frac{1}{2} n \log_2 n - \frac{1}{2} n$

$\Rightarrow$ 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:

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

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

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 $O(n \log n)$

- lisään 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 $A[0 \dots h]$ kekona, loppuosa $A[h + 1 \dots n - 1]$ lopullisessa tilassa $\Rightarrow$ lisämuistin tarve vain $\Theta(1)$
- lisäysvaiheen voi korvata algoritmilla, joka kuluttaa $\Theta(n)$ aikaa

HEAPSORT(&A)

```
1  TEEKKO(A)
2  for  $h := A.koko - 1$  downto 1 do
3       $apu := A[0]$ 
4      MUUTETTUPOISKEOSTA(A, h)
5       $A[h] := apu$ 
```

- silmukan alku unsigned:lla:
for(unsigned h = A.size(); h-- > 1;)

TEEKEKO(&A)

```
1   $n := A.koko$ 
2  for  $k := \lfloor n/2 \rfloor - 1$  downto 0 do
3       $apu := A[k]; i := k; j := 2i + 1$ 
4      while true do
5          if  $j + 1 < n \ \&\& \ A[j + 1].x \geq A[j].x$  then
6               $j := j + 1$ 
7          if  $j \geq n \ || \ A[j].x \leq apu.x$  then break
8           $A[i] := A[j]; i := j; j := 2i + 1$ 
9       $A[i] := apu$ 
```

TEEKEKO:n rivillä 2 $A[k+1 \dots n-1]$: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 $A[0 \dots n-1]$ 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

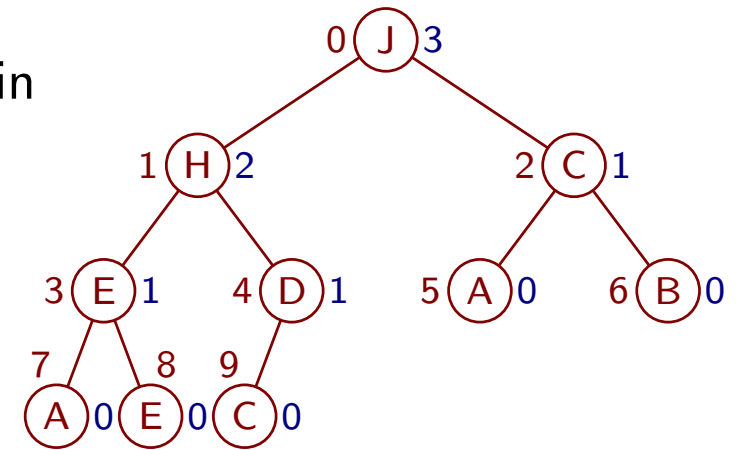
- kunkin ottelun häviöjää putoaa pois ja voittaja jatkaa, kunnes jäljellä on 4 joukkuetta
 - ei tasapelejä
 - loppujen neljän joukkueen kesken pelataan neljä ottelua
- kukin kärkijoukkue pelaa $\approx \log_2 n$ ottelua, kukin joukkue $O(\log n)$ ottelua
- joukkueiden määrän pudottamiseksi neljään tarvitaan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



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

⇒ vähintään korkeudella k on $m + y$ solmua

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

⇒ vähintään korkeudella k on $m + y - 1$ lasta

- edeltä korkeuden muilla solmuilla ja viimeisellä solmulla ...

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

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

⇒ $m = y + 1$ tai $m = y$

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

⇒ korkeintaan puolet solmuista voi valua edes yhden askeleen, korkeintaan neljäsosa toisenkin askeleen, korkeintaan kahdeksasosa kolmannenkin jne.

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

⇒ TEEKEKO toimii ajassa $O(n)$

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

⇒ 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
- ⇒ 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 $0, \dots, n-1$, missä n on juuri sen taulukon koko
 - ⇒ 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 suunnilleen yhtä nopea
- teoria ennustaa, että LISÄÄKEKOON on hitaimmillaan, kun taulukko on valmiiksi järjestyksessä

⇒ kokeiltiin eri järjestyksiä

0...4 999	TEEKEKO				<i>n</i> kertaa LISÄÄKEKON			
	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
- TEEKEKON vaikuttaa jonkin verran paremmalta kuin *n* kertaa LISÄÄKEKON
- 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 $\Theta(n)$

⇒ 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 kekkoon uudelleen aina kun sen prioriteetti nousee

- sama alkio voi olla keossa monena kappaleena
- keinoja tunnistaa keosta tuleva 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

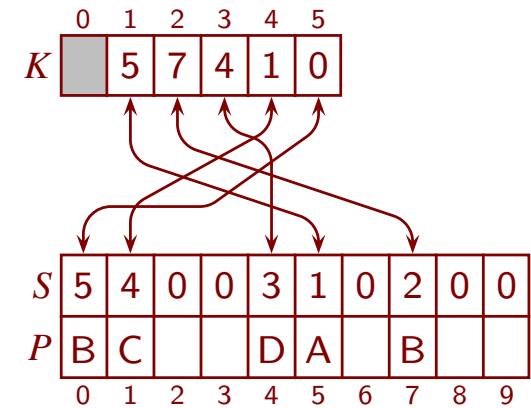
- kotitehtävinä

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

- esim. liian kauan kestänyt työ keskeytetään ja laitetaan kekkoon 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 7.3

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)

```
1  if  $k \geq S.koko$  then
2       $S.kooksi(k + 1, 0); P.kooksi(k + 1)$ 
3  if  $S[k] = 0$  then
4       $i := K.koko; K.kooksi(i + 1)$ 
5  else
6       $i := S[k]$ 
7   $P[k] := p; j := \lfloor \frac{i}{2} \rfloor$ 
8  while  $j > 0 \ \&\& \ P[K[j]] < p$  do
9       $K[i] := K[j]; S[K[i]] := i; i := j; j := \lfloor \frac{i}{2} \rfloor$ 
10 while true do
11      $j := 2i$ 
12     if  $j + 1 < K.koko \ \&\& \ P[K[j + 1]] \geq P[K[j]]$  then
13          $j := j + 1$ 
14     if  $j \geq K.koko \ || \ P[K[j]] \leq p$  then break
15      $K[i] := K[j]; S[K[i]] := i; i := j$ 
16  $K[i] := k; S[k] := i$ 
```

POISPJONOSTA(k)

```
17 if  $k \geq S.koko \ || \ S[k] = 0$  then return
18  $i := K.koko - 1; h := K[i]$ 
19  $K.kooksi(i)$ 
20  $S[h] := S[k]; S[k] := 0$ 
21 if  $h \neq k$  then
22     MUUTAPRIORITEETTI( $h, P[h]$ )
```

POISPJONOSTA

```
23 if  $K.koko > 1$  then
24     POISPJONOSTA( $K[1]$ )
```

ALUSTAPJONO

```
25  $S.kooksi(0); P.kooksi(0)$ 
26  $K.kooksi(1)$ 
```

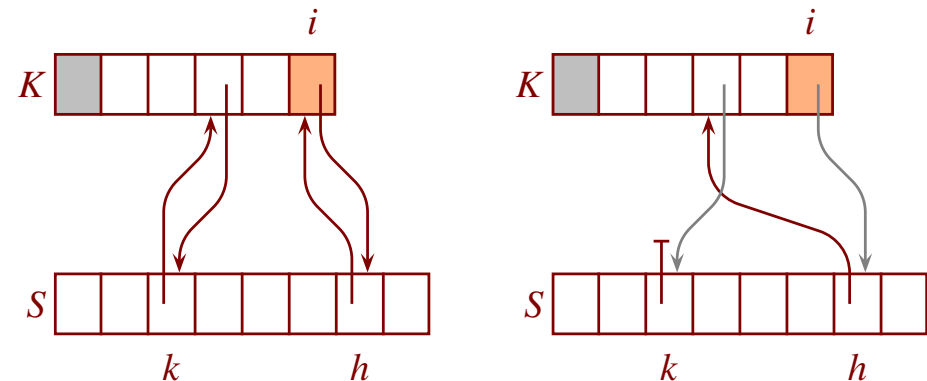
MUUTAPRIORITEETTI(k, p):n toiminta

- 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
- ⇒ 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  $k \geq S.koko \mid S[k] = 0$  then return
18   $i := K.koko - 1; h := K[i]$ 
19   $K.kooksi(i)$ 
20   $S[h] := S[k]; S[k] := 0$ 
21  if  $h \neq k$  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

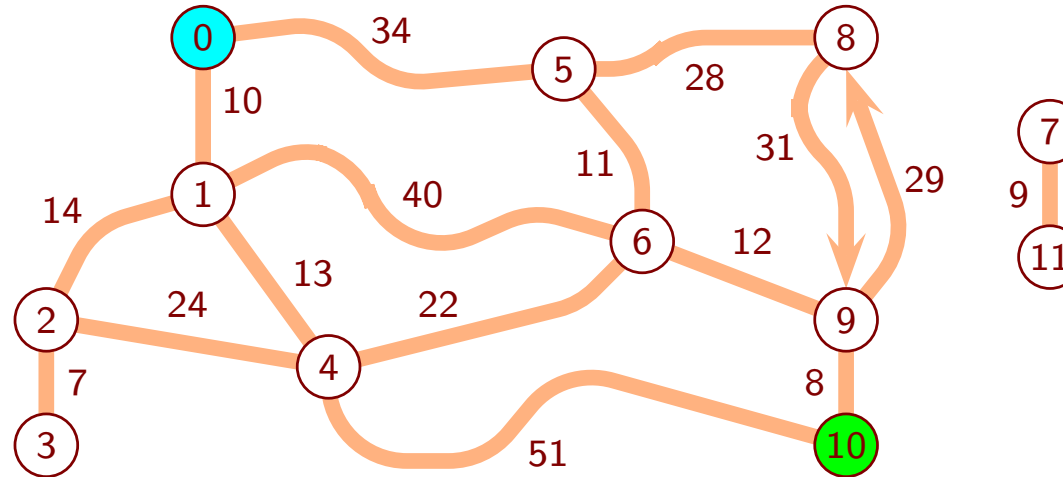
- S ei koskaan pienene
- rivillä 2 uusien alkioden paikaksi tulee 0 eli "ei keossa", K ei muutu
- K :n kasvatusta rivillä 4 vastaavat päivitykset K :hon ja S :ään tehdään rivillä 9, 15 tai 16
 - sitä ennen $K[i]$:tä ei käytetä
 - pätee myös rivillä 6 löydetylle i
- rivillä 9
 - K ja S päivitetään toisiaan 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	77
4.2	Syötteen lukeminen	81
4.3	Kekotoiminnot	85
4.4	Reitin etsintä	87
4.5	Ohjelman loppuosat	96
4.6	Etsintä yhtäaikaa kahteen suuntaan	98
4.7	A* reitinetsintäalgoritmi	100
4.8	Floyd–Warshall reitinetsintäalgoritmi	104
4.9	Bellman–Ford reitinetsintäalgoritmi	108
4.10	Bellman–Fordin nopeuttaminen rengaspuskurilla	113

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
⇒ 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
 - ⇒ 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 etaisuus;  
8     solmutyyppi( unsigned kl ):  
9         kaarten_loppu( kl ), reitti( ~0u ), etaisuus( 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
 - kaarella 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*) $\Rightarrow$ rivin 27 ehto ei ole sama kuin `0. > kaari.pituus || kaari.pituus == aareton`
- rivillä 31 luodaan tarvittaessa uusia solmutietueita
 - $\text{solmu} \geq \text{vanha} \geq n - 1$
 - 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 loppuksi alustetaan senhetkinen kaarten määrä eli edellisen solmun `kaarten_loppu` $\Rightarrow$ eivät vielä saa yhtään kaarta
- rivillä 32 uusi 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  }
```

Olennot erot aikaisempaan

- keossa on jokin pienin ensin eli ylimmäisenä
- (keon indeksointi alkaa nolasta)
- 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 etäisyys 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 löydetyt reitin pituus, eli se mitä muualla ohjelmassa kutsutaan etäisyydeksi

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

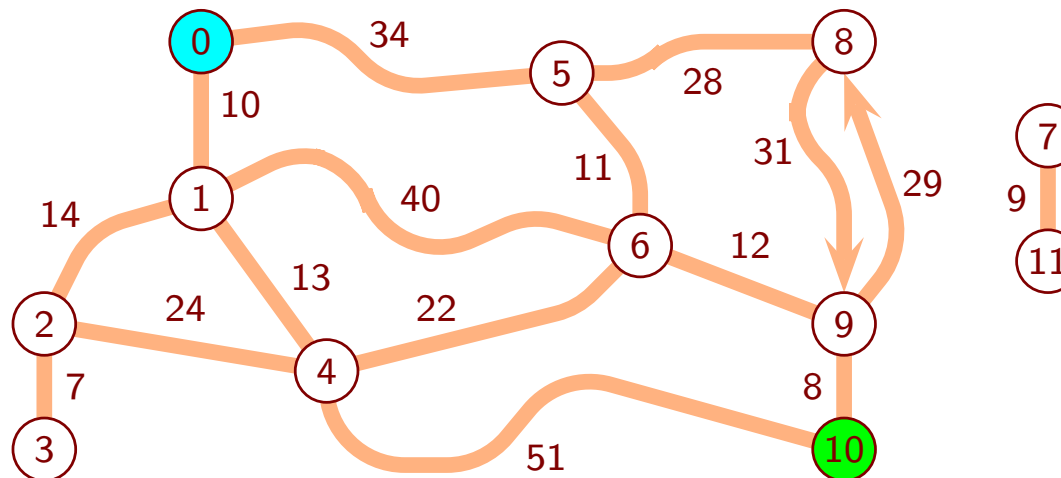
4.4 Reitin etsintä

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 $\Rightarrow$ solmu laitetaan kekoon uudelleen aina kun löytyy entistä lyhyempi reitti siihen
- $\Rightarrow$ 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 etäisyys 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 etäisyys ei ole minimietäisyys
- reitin kärjen etäisyys 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 etäisyys:kseen minimietäisyyden

Invariantti 3

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

- jos solmuun ei ole reittiä, niin sen etaisyyys 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 etaisyyys 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 maa1i, 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 etaisyyys ei ole minimietäisyys (kärjen määritelmä)

⇒ siihen vievää kaarta ei hylätä rivillä 69

⇒ siitä tulee reitin kärki rivillä 72
- jos jokin reitillä myöhäisempi solmu saa etaisyyys:kseen minimietäisyyden, niin se lisätään kekoon rivillä 72
 - ⇒ siitä tulee reitin kärki

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

- kutsumme tätä solmua "määränpääksi"
- jollei määränpään etäisyys 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 minimipituuisella reitillä ennen määränpäätä
 - kaarten pituudet eivät ole negatiivisia
- määränpään etäisyys on suurempi, koska se ei ole minimietäisyys
- määränpään etäisyydet keossa ovat ainakin yhtäsuuret

⇒ reitin kärki on keosta ulostulojärjestyksessä ennen määränpäätä

⇒ 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 ⇒ maali:n etäisyys on ääretön
 - keko on tyhjä ⇒ jokaisen solmun etäisyys on minimietäisyys (invariantti 3)

⇒ 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 etäisyys minimietäisyys
- ⇒ rivi 69 estää lisäämästä solmua keoon 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 tyhjää 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)
- ⇒ tarvittaessa lisämuistin käyttöksi 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 suoritus aika-, muisti- ja laillisuustarkasteluihin
- koko ohjelma kuluttaa $O(n + m \log m)$ aikaa ja $\Theta(n + m)$ muistia

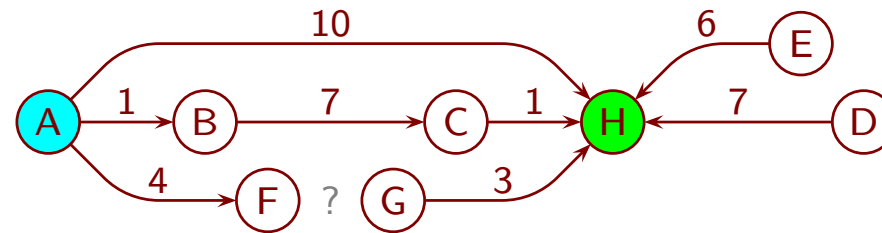
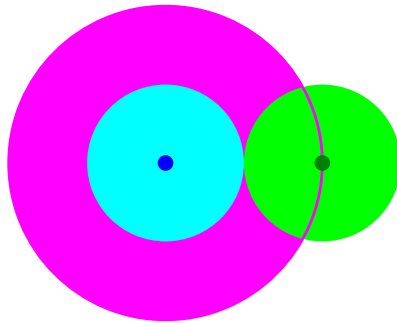
Dijkstran algoritmi tarvitsee kartasta vain osan

- lähdöstä joka suuntaan maalin etäisyydelle saakka
- ⇒ karttaa ei tarvitse hakea tallennustilasta kerralla kokonaan, vaan tarpeen mukaan
- ⇒ 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 risteyskiä 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 käsitelty yhden solmun
 - H on etuperin etsinnän keossa etäisyydellä 10
 - takaperin etsintä on käsitelty H:n
 - mahdollisimman lyhyttä reittiä ei ole vielä löytynyt
- ⇒ ei riitä, että toinen etsintä on käsitelty 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

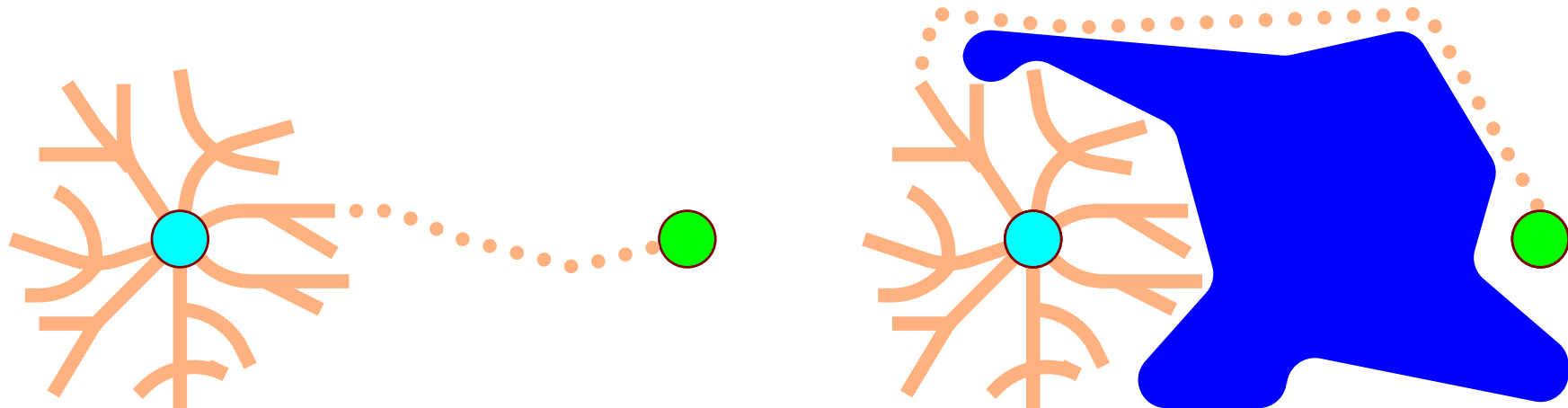
- lyhin löydetyn reitin pituus on enintään etuperin keon ylimmän etäisyys plus takaperin keon ylimmän etäisyys
- tutkimattomille kaarille pätee silloin
 - häntäpää on vähintään yhtä kaukana lähdöstä kuin etuperin keon ylimmän etäis.
 - kärkipää on vähintään yhtä kaukana maalista kuin takaperin keon ylimmän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

⇒ 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 tietiselokuvien madonreikiä
- tutkii todellisten maantiekarttojen tapauksessa paljon väärään suuntaan vieviä teitä



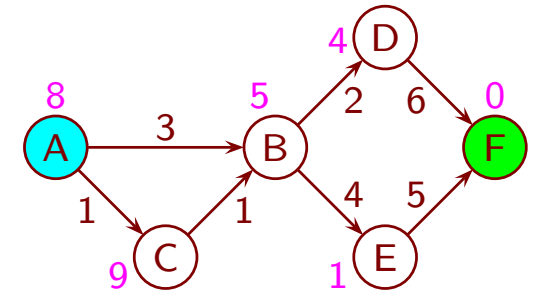
A* muistuttaa Dijkstraa, mutta ottaa huomioon myös arvion jäljellä olevasta matkasta

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

```

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

- kaarten pituudet saavat olla negatiivisia, mutta silmukoiden pituudet eivät saa
⇒ solmun sisällyttäminen reittiin kahdesti ei voi lyhentää reittiä
 - aluksi M sisältää kaarten pituudet
 - kukin $M[i, i]$ voidaan alustaa nolllaksi tai solmusta i itseensä vievän kaaren mukaan
 - kierroksen k jälkeen kukin $M[i, j]$ sisältää lyhimmän pituuden, kun matkan varrella
 - saa käyttää vain solmuja $0, \dots, k$, 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 $0, \dots, k - 1$
 - $i \rightsquigarrow x \rightsquigarrow x \rightsquigarrow j$ ei voi olla lyhyempi kuin $i \rightsquigarrow x \rightsquigarrow j$
- ⇒ rivin 5 ehto estää kielletyn toiston tulemisen mukaan
⇒ ei haittaa jos $M[i, k]$ tai $M[k, j]$ on jo päivitetty kierroksella k
- suoritusaika on $\Theta(n^3)$

Jonkin lyhimmän reitin tulostaminen

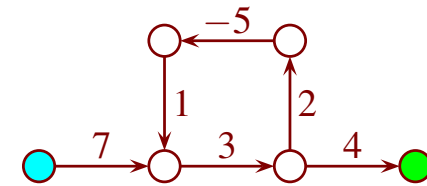
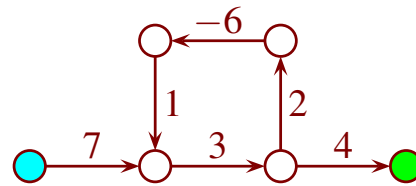
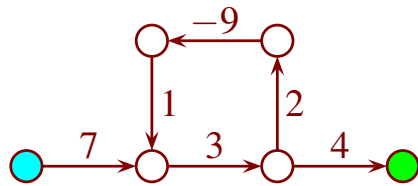
- ehdotus: $R[i, j]$ sisältää valitun k tai tiedon, että ei valittu mitään k
 - toimii, mutta tunnetaan näppärämpi keino
- näppärämpi: $R[i, j]$ sisältää seuraavan solmun numeron reitillä i :stä k :hon
 - jos i :stä on kaari j :hin, niin alustetaan $R[i, j] := j$, ja muutoin ei tarvitse alustaa
 - kun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

Suoritus aika

- $\Theta(n^3)$ 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, ...

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

- ⇒ voidaan rajoittaa reitteihin, joissa mikään solmu ei toistu
- (niin on rajoitettu 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 $n - 1$
- ⇒ kierroksen i jälkeen on löydetty ainakin jotkin lyhimmat 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 $S[0 \dots n - 1]$

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

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

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

```
1  for  $i := 0$  to  $n - 1$  do  $S[i].e := \text{ääretön}$ 
2   $S[\text{lähtö}].e := 0$ 
3   $\text{kierros} := 1$ ;  $\text{muuttui} := \text{true}$ 
4  while  $\text{kierros} \leq n$  \&\&  $\text{muuttui}$  do
5       $\text{kierros} := \text{kierros} + 1$ ;  $\text{muuttui} := \text{false}$ 
6      for  $j := 0$  to  $m - 1$  do
7           $d := S[K[j].h].e + K[j].p$ ;  $i := K[j].k$ 
8          if  $d < S[i].e$  then
9               $S[i].e := d$ ;  $S[i].r := K[j].h$ 
10          $\text{muuttui} := \text{true}$ 
```

Jos ja vain jos lopussa $muuttui = \text{true}$, 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 $muuttui = \text{false}$
- osoitamme seuraavaksi, että muussa tapauksessa lopussa $muuttui = \text{true}$
- olkoot silmukan solmut $s_0, s_1, \dots, s_\ell$ ja kaarten pituudet $p_{0,1}, p_{1,2}, \dots, p_{\ell-1,\ell}$
 - s_ℓ on sama solmu kuin s_0
- jos rivin 8 ehto ei toteudu silmukan millekään kaarelle, niin
 - $S[s_0].e + p_{0,1} \geq S[s_1].e$
 - $S[s_0].e + p_{0,1} + p_{1,2} \geq S[s_1].e + p_{1,2} \geq S[s_2].e$
 - $\dots$
 - $S[s_0].e + p_{0,1} + \dots + p_{\ell-1,\ell} \geq S[s_\ell].e$
- jos lähdöstä on reitti s_0 :aan, niin $S[s_0].e$ on äärellinen, joten $p_{0,1} + \dots + p_{\ell-1,\ell} \geq 0$

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 5
 - myös Dijkstran algoritmia voidaan käyttää käyttämällä kaarten pituuksina nollaa
 - rivi 63 pois
 - Dijkstran algoritmi on siihen tarpeettoman hidas ja monimutkainen
- ⇒ keon tilalle järjestämätön taulukko jonka alkiossa on vain solmun numero, etaisyy:n ja reitti:n tilalle bitti joka kertoo onko solmu löydetty
- ⇒ muuttuu melkein erääksi luvun 5 algoritmiksi, nopeus $O(n + m)$

4.10 Bellman–Fordin nopeuttaminen rengaspuskurilla

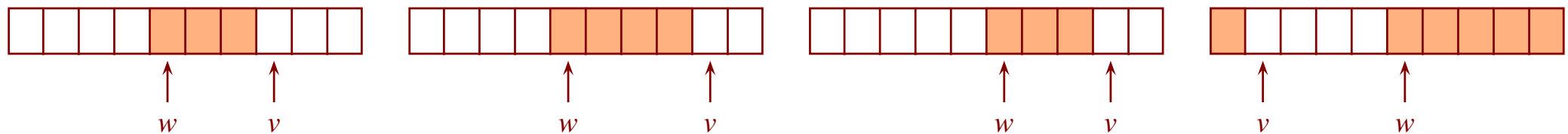
Kaaren käsittelystä on hyötyä vain jos sen häntäpään solmun e on muuttunut

- solmut, joiden e on muuttunut kuluvalle kierroksella, taulukkoon peräkkäin
⇒ ne ja niistä lähtevät kaaret voi selata tehokkaasti
 - kaaret esitetään kuten Dijkstran algoritmossa
- samaa solmua ei saa käsitellä samalla kierroksella uudelleen
⇒ kuluvalle ja seuraavalle 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 yhtäaikaan alkioita
- lisäyksen ja poiston koodit kotitehtävänä

Bellman–Ford-sovelluksessa

- solmun uudesta käsittelystä on hyötyä vain jos sen $.e$ on muuttunut edellisen jälkeen
- ⇒ samaa alkiota ei kannata laittaa jonoon useasti
- ⇒ jonossa on yhtäaikaa enintään n alkiota
- ⇒ 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

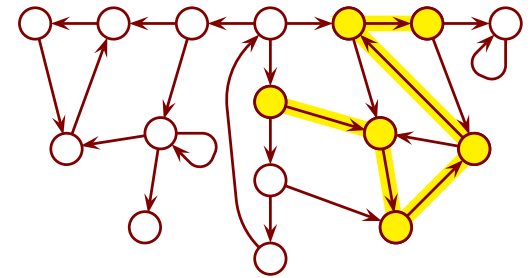
5 Lisää graafialgoritmeja

5.1	Leveyshaku	116
5.2	Sovellusesimerkki: 11-peli	119
5.3	Sovellusesimerkki: maksimi vuo	134
5.4	Yleinen graafihaku	142
5.5	Syvyyshaku	145
5.6	Minimi virittävä puu	155

5.1 Leveyshaku

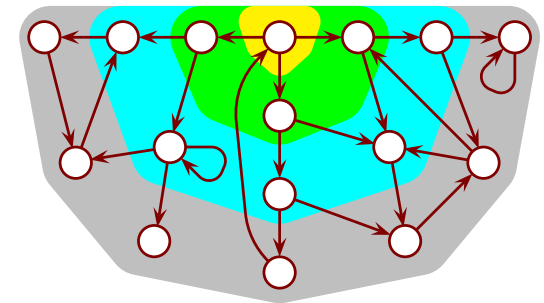
Suunnattu graafi (*directed graph*), kertausta ja uutta

- koostuu **solmuista** (*vertex, node*) ja **kaarista** (*edge, arc*)
 - solmut piirretään ympyröinä ja kaaret nuolina
 - kaari alkaa jostakin solmusta ja päättyy johonkin solmuun
 - solmujen joukkoa merkitään yleensä V ja kaarten joukkoa E
- **polku** (*path*) on jono solmuja siten, että edellisestä seuraavaan on aina kaari
 - polun **pituus** (*length*) on polun kaarten määrä eli solmujen määrä miinus yksi
 - yksittäinen solmu on polku, jonka pituus on 0
- solmu2 on **saavutettavissa** (*reachable*) solmu1:stä, jos ja vain jos solmu1:stä on polku solmu2:een



Leveyshaku etenee rintamana lähtösolmusta poispäin

- löytää kaarien määrässä mitattuna mahdollisimman lyhyet polut lähtösolmusta jokaiseen saavutettavaan solmuun



Voidaan tulkita Dijkstran algoritmin erikoistapaukseksi

- jokaisen kaaren pituus on 1
- ⇒ prioriteettijonon tilalla voi olla yksinkertaisempi tietorakenne
- ⇒ helppo toteuttaa tehokkaasti, ajassa $O(|V| + |E|)$ (missä $|X|$ on joukon X koko)

Algoritmi: laita lähtösolmu jonoon, merkitse se löydetyksi ja toista seuraavia kunnes jono on tyhjä

- poista jonon ensimmäinen solmu
- tutki sen jokainen lähtökaari seuraavasti: jos kärkipään solmu ei vielä ole löydetty, niin laita se jonoon ja merkitse löydetyksi

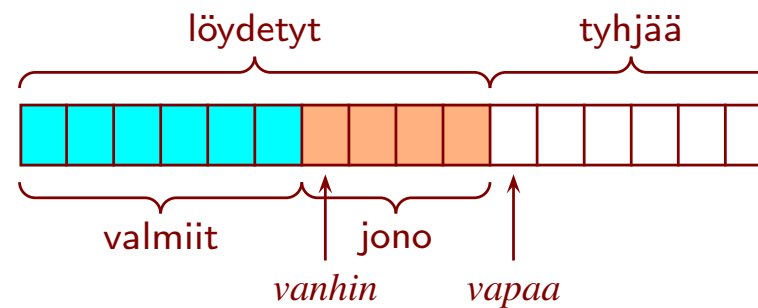
Konkreettisemmin

- solmutietue löydetään numerolla, viitteellä, osoittimella tms.
- löydetty-tieto sijaitsee solmussa
 - tai solmun numerolla indeksoitavassa taulukossa, jos solmutietue löytyy numerolla
- jonon voi toteuttaa helposti taulukkona, jonka koko on solmujen määrä
 - suhteessa pieni määrä lisää muistia eksplisiittisesti esitettyyn graafiin verrattuna
 - jos lähtösolmusta on kaari jokaiseen muuhun, niin melkein sen verran tarvitaan
- " $\neg$ " on logiikan symboli, joka tarkoittaa "ei" (siis ohjelmoinnin "!")

```
1  for  $v := 0$  to  $Solmut.koko - 1$  do  $Solmut[v].löydetty := false$ 
2   $vanhin := 0$  ;  $jono[0] := lähtö$  ;  $vapaa := 1$  ;  $Solmut[lähtö].löydetty := true$ 
3  while  $vanhin < vapaa$  do
4       $u := jono[vanhin]$  ;  $vanhin := vanhin + 1$ 
5      for  $v := Solmut[u].eka\_naapuri$  to  $Solmut[u].viim\_naapuri$  do
6          if  $\neg Solmut[v].löydetty$  then
7               $jono[vapaa] := v$  ;  $vapaa := vapaa + 1$  ;  $Solmut[v].löydetty := true$ 
```

while-silmukan invariantteja

- $vanhin \leq vapaa$
- solmun *löydetty* on tosi jos ja vain jos solmun numero on taulukossa *jono*
 - solmu merkitään löydetyksi ja sen numero laitetaan *jono*:on yhtäaikaan
 - *löydetty*-tietoja ei pyyhitä eikä solmujen numeroita poisteta *jono*:sta
- kunkin solmun numero on jonossa enintään kerran
 - solmun numero laitetaan jonoon vain jos solmu ei ollut löydetty



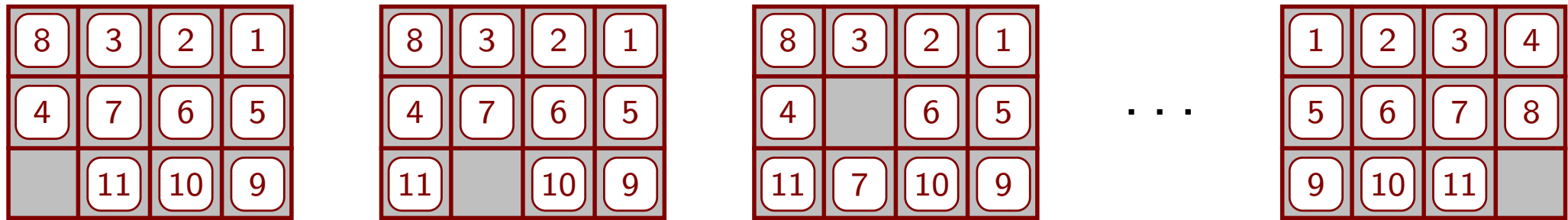
Suoritus aika $O(|V| + |E|)$, jos solmun lähtökaarten selaus on nopeaa

- kukin solmu laitetaan jonoon enintään kerran
- ⇒ rivit 1, 3 ja 4 suoritetaan enintään kerran solmua kohti
ja 5, 6 ja 7 enintään kerran kaarta (ja solmua) kohti
- muu on vakioaikaista

Muilta osin algoritmin todistus palautuu luvussa 5.4 esitettävään

Seuraavissa alaluvuissa on esimerkkitehtäviä, joita on mukautettu sovelluksen tarpeisiin

5.2 Sovellusesimerkki: 11-peli



Kuinka paljon on eri pelitilanteita?

- 11 lukua voi kirjoittaa 3×4 ruuduukkoon $12 \cdot 11 \cdot \dots \cdot 3 \cdot 2 = 479\,001\,600$ eri tavalla
- voikohan kaikki saavuttaa?



Tutkimme ensin järjestetystä tilasta saavutettavien tilojen muodostamaa graafia

- tavoitteena ei vielä ole tulostaa ratkaisuja, vaan muodostaa kaikki saavutettavat tilat
 - solmun **taso** on mahdollisimman lyhyen lähdöstä solmuun vievän polun pituus
- ⇒ jos siirto vie tasolta n tasolle m , niin $m \leq n + 1$
- mikä tahansa siirto voidaan peruuttaa
- ⇒ siirto ei voi viedä tasolta n tasolle m , missä $m < n - 1$
- muutoin sen peruuttava siirto kasvattaisi tasoa ainakin kahdella
- ⇒ riittää tallettaa kolme tasoa

Kirjastojen käyttöön ottaminen ja pelilaudan kokoa kuvaavat vakiot

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  const unsigned lev = 4, kork = 3, ruutuja = kork * lev;
```

Tilan esitys

- ruudun sisältö voidaan esittää 4 bitillä
 - tyhjä, 1, ..., 11

⇒ pelin tila voidaan esittää $12 \cdot 4 = 48$ bitillä

⇒ 64-bittinen etumerkitön kokonaisluku riittää

- C++-standardi tuntee sen nimellä `uint64_t`, mutta kaikki kääntäjät eivät tunne
- maskia käytetään poimimaan yksittäisen ruudun sisältö tilasta
 - koneessa, jossa koeajot tehtiin, `0xF` oli 32-bittinen, ja se ei riitä
 - rivi 6 huolehtii, että maski on 64-bittinen

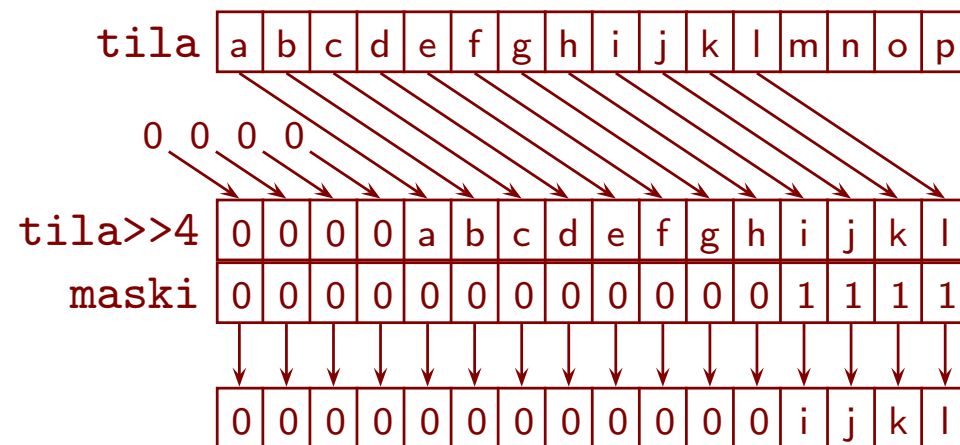
```
5  typedef unsigned long long uint64_t;
6  const uint64_t maski = 0xF;
```


Tulostusaliohjelmasta näkee, miten tilan osat saa kaivettua esiin

- osa `std::cout << ' '`; `std::cout.width(2)`; voisi olla `std::cout.width(3)`;
- valittu tapa on vikasietoisempi ohjelmointivirheiden suhteen
 - varmistaa välilyönnin tulostetun luvun eteen, jos se on isompi kuin luultiin

```
7 void tulosta( uint64_t tila ){
8     for( unsigned ii = 0; ii < lev; ++ii ){ std::cout << "---"; }
9     std::cout << '\n';
10    for( unsigned ii = 0; ii < ruutuja; ++ii ){
11        unsigned arvo = (tila >> 4*ii) & maski;
12        if( arvo ){ std::cout << ' '; std::cout.width(2); std::cout << arvo; }
13        else{ std::cout << "  "; }
14        if( (ii+1) % lev == 0 ){ std::cout << '\n'; }
15    }
16 }
```

Mitä `(tila >> 4*ii) & maski` tekee?



Seuraava aliohjelma etsii tyhjän ruudun paikan

```
17  unsigned etsi_tyhja( uint64_t tila ){
18      unsigned ii = 0;
19      while( tila & maski ){ tila >>= 4; ++ii; }
20      return ii;
21  }
```

Haussa käytetään kolmea tasoa:

- **vanhat:** edellisen tason tilat
 - suuruusjärjestyksessä, jotta tilan löytäminen niiden joukosta olisi nopeaa
- **nykyiset:** tutkittavat tilat
- **uudet:** tällä tasolla löydetty tilat
 - kohta kerrottavasta syystä saattaa sisältää saman tilan useasti

```
22  std::vector< uint64_t > vanhat, nykyiset, uudet;
23  unsigned tiloja = 0;
```

Seuraava aliohjelma yrittää siirtää laatan annetusta paikasta tyhjään paikkaan

- jos siirto vie edelliselle tasolle, se hylätään
 - `binary_search` on puolitushaku, joka palauttaa vain löytymistiedon
 - `lower_bound` on puolitushaku, joka palauttaa löytymispaikan kuten luvussa 1.5
 - siirto ei voi viedä nykyiselle tasolle
 - kuvittele laudan ruudut väritetyiksi kuten shakkilaudassa
 - jokainen siirto vaihtaa tyhjän paikan mustasta ruudusta valkoiseen tai päinvastoin
- ⇒ kaikissa saman tason tiloissa tyhjät ovat keskenään samanvärisissä ruuduissa
- sama tila voi tulla monta kertaa uusien tasolle
 - ylimääräisten kopioiden tunnistaminen olisi tässä vaiheessa kallista, mutta on myöhemmin halpaa

```
24 void siirra( unsigned ii, unsigned tyhja, unsigned paikka ){
25     uint64_t uusi = nykyiset[ ii ];
26     tyhja *= 4; paikka *= 4;
27     uint64_t nro = ((uusi >> paikka) & maski) << tyhja;
28     uusi &= ~(maski << paikka); uusi |= nro;
29     if( !std::binary_search( vanhat.begin(), vanhat.end(), uusi ) ){
30         uudet.push_back( uusi );
31     }
32 }
```

Leveyshaun pääsilmut alustuksineen

- liiat laskee uusien tilojen ylimääräisten löytämiskertojen määrän
- rivi 45 siirtää nykyiset vanhoiksi ja nolaa nykyiset
- rivi 47 kopioi uudet nykyisiksi jättäen saman tilan toistot pois

```
33 void tutki( uint64_t tila ){
34     nykyiset.push_back( tila ); ++tiloja; unsigned taso = 0, liiat = 0;
35     while( !nykyiset.empty() ){
36         std::cout << taso++ << ": " << tiloja << ' ' << nykyiset.size() << ' '
37             << liiat << '\n';
38         for( unsigned ii = 0; ii < nykyiset.size(); ++ii ){
39             unsigned tyhja = etsi_tyhja( nykyiset[ ii ] );
40             if( tyhja >= lev ){ siirra( ii, tyhja, tyhja - lev ); }
41             if( tyhja + lev < ruutuja ){ siirra( ii, tyhja, tyhja + lev ); }
42             if( tyhja % lev ){ siirra( ii, tyhja, tyhja - 1 ); }
43             if( (tyhja+1) % lev ){ siirra( ii, tyhja, tyhja+1 ); }
44         }
45         vanhat.clear(); vanhat.swap( nykyiset );
46         std::sort( uudet.begin(), uudet.end() ); liiat = uudet.size();
47         unique_copy( uudet.begin(), uudet.end(), back_inserter( nykyiset ) );
48         uudet.clear(); tiloja += nykyiset.size(); liiat -= nykyiset.size();
49     }
50 }
```

Pääohjelma

- muodosta ja tulosta järjestetty tila
- tutki saavutettavat tilat tulostaen tasoittain tilastoja
- tulosta tilojen määrä ja ne tilat, jotka ovat mahdollisimman kaukana lähtötilasta

```
51  int main(){
52      uint64_t tila = 0;
53      for( uint64_t ii = 1; ii < ruutuja; ++ii ){ tila |= ii << 4*(ii-1); }
54      tulosta( tila ); tutki( tila ); std::cout << tiloja << " tilaa\n";
55      for( unsigned ii = 0; ii < vanhat.size(); ++ii ){ tulosta( vanhat[ ii ] ); }
56  }
```

Havaintoja

- ohjelman suoritus kesti noin 52 s
- tärkeimpien tasojen tuloksia

taso	tiloja yhteensä	uusia tiloja	liikoja	
0	1	1	0	aloitustaso
26	8 052 888	2 782 396	479 364	keskimmäiset tasot
27	12 062 610	4 009 722	750 022	keskimmäiset tasot
36	145 628 724	21 841 159	9 325 834	eniten uusia tiloja
37	166 535 629	20 906 905	9 723 913	eniten liikoja tiloja
53	239 500 800	18	18	viimeinen taso

Saavutettavia tiloja on täsmälleen puolet edellä lasketusta ylärajasta 479 001 600

- **permutaatio** (**permutation**) voidaan ajatella taulukkona $P[1 \dots n]$, jossa on luvut $1, \dots, n$ jossain järjestyksessä
 - permutaatiot jakautuvat parillisiin ja parittomiin
 - kahden alkion vaihto keskenään vaihtaa parillisesta parittomaan tai toisinpäin
 - niiden parien määrä, joille $i < j$ ja $P[i] > P[j]$, muuttuu vaihdossa parittomasti
 - 11-pelin siirto on jonkin laatan vaihto tyhjän paikan kanssa
- ⇒ jos kahden tilan ainoa ero on kahden ruudun sisältö, niin toisen muuttaminen toiseksi vaatii parittoman määrän siirtoja
- edellä todettiin, että jokainen siirto vaihtaa tyhjän paikan erivärisen
- ⇒ jos siirtosarja ei vaihda tyhjän paikan sijaintia, niin sen pituus on parillinen
- ⇒ ei ole olemassa siirtosarjaa, joka vaihtaa kahden epätyhjän ruudun sisällöt keskenään eikä tee muuta
- näköjään muita rajoitteita ei ole!

Keskimmäisiin tasoihin mennessä on tutkittu $8\,052\,888 + 12\,062\,610 = 20\,115\,498$ tilaa

- alle kymmenesosa saavutettavien tilojen kokonaismäärästä
- ⇒ pelin ratkaiseva ohjelma kannattanee toteuttaa kaksisuuntaisella haulla
- etuperin haun lähtötila on eri, joten tutkittavien tilojen määräkin voi olla eri
 - teemme sellaisen!

Ensimmäiset 21 riviä ovat kuten edellisessä ohjelmassa

Nyt tilan yhteydessä tarvitaan

- tieto reitillä edellisestä / seuraavasta tilasta (etuperin / takaperin haussa)
- aputoimintoja kirjastotoimintojen käytön mahdollistamiseksi
- riveillä 25 ja 26 parametrin tyyppi voisi olla myös `const solmutyyppi &`
 - `solmutyyppi` on pieni (16 tavua), joten kopiointi on nopeaa

```
22  struct solmutyyppi{
23      uint64_t tila; unsigned reitti;
24      solmutyyppi( uint64_t tl, unsigned rt ): tila( tl ), reitti( rt ) {}
25      bool operator<( solmutyyppi toinen ) const { return tila < toinen.tila; }
26      bool operator==( solmutyyppi toinen ) const { return tila == toinen.tila; }
27  };
```

Jono muodostuu kolmesta osasta: nykyinen taso, edellinen taso ja vanhemmat tasot

- tasojen rajat ovat indekseinä eikä iteraattoreina, vaikka niitä tarvitaan jatkossa iteraattoreina
 - iteraattorit vanhenevat kun kasvavan taulukon sisältö muuttaa uuteen paikkaan
- rivit 33 ja 34 selkeyttävät koodia jatkossa

```
28  typedef std::vector< solmutyyppi >::iterator tilaattori;
29  struct jonotyyppi{
30      std::vector< solmutyyppi > tilat;
31      unsigned edell, nyk;
32      jonotyyppi(): edell(0), nyk(0) {}
33      unsigned koko() const { return tilat.size(); }
34      const solmutyyppi & operator[]( unsigned ii ) const { return tilat[ ii ]; }
35      const tilaattori itr_edell(){ return tilat.begin() + edell; }
36      const tilaattori itr_nyk(){ return tilat.begin() + nyk; }
37  };
```

Etsinnän tarvitsemat globaalit muuttujat

```
38  jonotyyppi jono_etup, jono_takap;    // etuperin ja takaperin haun jono
39  std::vector< solmutyyppi > uudet;    // uutena löydetyt tilat
40  unsigned etup = 0, takap = 0;        // tilat, joissa haut kohtaavat
```


Paikassa paikka olevan laatan siirtäminen tyhjään paikkaan

- kutsutaan molemmissa hauissa
⇒ jono1 ja jono2 eikä jono_etup ja jono_takap
 - hakujen kohdattua etup ja takap voivat olla väärinpäin, se korjataan jatkossa
- riveillä 51, ..., 55 havaitaan hakujen kohtaaminen ja talletetaan toisen haun tila

```
41  bool siirra(  
42      unsigned tyhja, unsigned paikka, jonotyyppi & jono1, jonotyyppi & jono2  
43  ){  
44      solmutyyppi uusi( jono1[ etup ].tila, etup );  
45      tyhja *= 4; paikka *= 4;  
46      uint64_t nro = ((uusi.tila >> paikka) & maski) << tyhja;  
47      uusi.tila &= ~(maski << paikka); uusi.tila |= nro;  
48      if( std::binary_search( jono1.itr_edell(), jono1.itr_nyk(), uusi ) ){  
49          return false;  
50      }  
51      takap = std::lower_bound( jono2.itr_nyk(), jono2.tilat.end(), uusi )  
52          - jono2.tilat.begin();  
53      if( takap < jono2.koko() && jono2[ takap ].tila == uusi.tila ){  
54          return true;  
55      }  
56      uudet.push_back( uusi ); return false;  
57  }
```

Seuraava aliohjelma käsittelee yhden tason jompaankumpaan suuntaan

- riveillä 59, ..., 67 käydään nykyisen tason tilat läpi
 - riveillä 60, ..., 66 kokeillaan siirtoa jokaiseen suuntaan
 - jos siirra havaitsee hakujen kohdanneen, tee_taso keskeyttää ja palauttaa true
- rivillä 68 muutetaan nykyinen taso edelliseksi
- riveillä 69, ..., 70 uusista tiloista tehdään nykyinen taso järjestettynä ja tilojen ylimääräiset kopiot poistettuina

```
58  bool tee_taso( jonotyyppi & jono1, jonotyyppi & jono2 ){
59      for( etup = jono1.nyk; etup < jono1.koko(); ++etup ){
60          unsigned th = etsi_tyhja( jono1[ etup ].tila );
61          if( th >= lev && siirra( th, th-lev, jono1, jono2 ) ){ return true; }
62          if( th+lev < ruutuja && siirra( th, th+lev, jono1, jono2 ) ){
63              return true;
64          }
65          if( th % lev && siirra( th, th-1, jono1, jono2 ) ){ return true; }
66          if( (th+1) % lev && siirra( th, th+1, jono1, jono2 ) ){ return true; }
67      }
68      jono1.edell = jono1.nyk; jono1.nyk = jono1.koko();
69      std::sort( uudet.begin(), uudet.end() );
70      unique_copy( uudet.begin(), uudet.end(), back_inserter( jono1.tilat ) );
71      uudet.clear(); return false;
72  }
```

Leveyshaun pääsilmutka alustuksineen

- rivi 81 keskeyttää, kun tasoja on tutkittu yhteensä maksimi siirtosarjan pituuteen 53 saakka
 - ohjelma on testattu muutamalla tilalla, joiden ratkaiseminen vaatii 53 siirtoa
- rivillä 83 korjataan etup ja takap, jos haut kohtasivat takaperin haussa
- `emplace_back` rakentaa alkion jonon jatkeeksi

```
73  unsigned taso = 0;  // haun taso lähdöstä tai maalista lukien
74  bool ratkaise( uint64_t lahto, uint64_t maali ){
75      jono_etup.tilat.emplace_back( lahto, 0 );
76      jono_takap.tilat.emplace_back( maali, 0 );
77      while( true ){
78          std::cout << taso++ << ": " << jono_etup.koko() + jono_takap.koko() <<
79              '\n';
80          if( tee_taso( jono_etup, jono_takap ) ){ return true; }
81          if( taso >= 27 ){ return false; }
82          if( tee_taso( jono_takap, jono_etup ) ){
83              std::swap( etup, takap ); return true;
84          }
85      }
86  }
```

Ratkaisusuunnassa löydetyt reitin osan tulostus

```
87 void tulosta_takaperin( unsigned ii ){
88     if( ii ){ tulosta_takaperin( jono_etup[ ii ].reitti ); }
89     tulosta( jono_etup[ ii ].tila );
90 }
```

Pääohjelma alkaa: syötteen lukeminen ja tarkastus

- tyhjää ruutua edustaa syötteessä 0
- nro on 64-bittinen, jotta myös $nro \ll 4*ii$ olisi

```
91 int main(){
92     /* Lue lähtötila. */
93     uint64_t lahto = 0; bool saatiin[ ruutuja ] = {};
94     for( unsigned ii = 0; ii < ruutuja; ++ii ){
95         uint64_t nro = 0; std::cin >> nro;
96         if( nro >= ruutuja ){ std::cout << nro << " on liian suuri\n"; return 2; }
97         if( saatiin[ nro ] ){ std::cout << nro << " toistui\n"; return 2; }
98         lahto |= nro << 4*ii; saatiin[ nro ] = true;
99     }
100     tulosta( lahto );
```

Loput pääohjelmasta

- rivit 105, ..., 107 tarvitaan, koska siirra ei pysty havaitsemaan 0:n pituista polkua

```
101     /* Muodosta tavoite ja yritä ratkaista peli. */
102     uint64_t maali = 0;
103     for( uint64_t ii = 1; ii < ruutuja; ++ii ){ maali |= ii << 4*(ii-1); }
104     if( lahto == maali ){
105         std::cout << "lähtö = maali, ratkesi!\n"; tulosta( lahto ); return 0;
106     }
107     bool ratkesi = ratkaise( lahto, maali );
108     std::cout << taso << " tasoa, "
109         << jono_etup.koko() + jono_takap.koko() + uudet.size() << " tilaa, ";
110     if( !ratkesi ){ std::cout << "ei ratkennut.\n"; return 1; }
111     std::cout << "ratkesi!\n";

112     /* Tulosta ratkaisu. */
113     tulosta_takaperin( etup );
114     tulosta( jono_takap[ takap ].tila );
115     while( takap ){
116         takap = jono_takap[ takap ].reitti; tulosta( jono_takap[ takap ].tila );
117     }

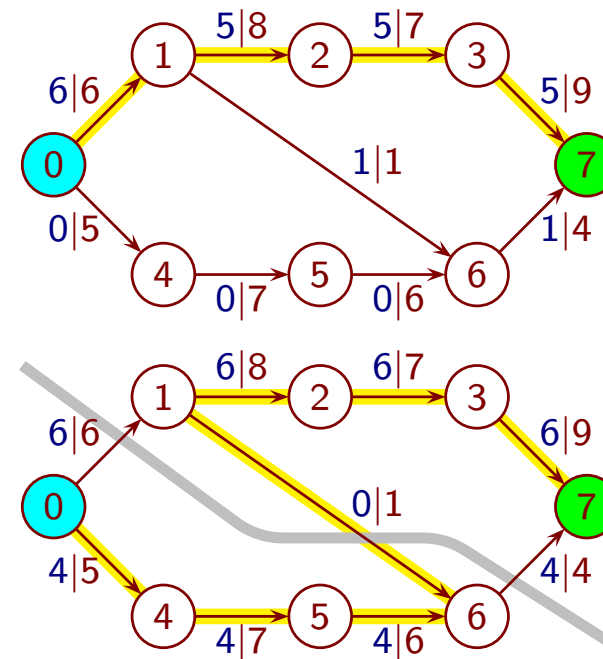
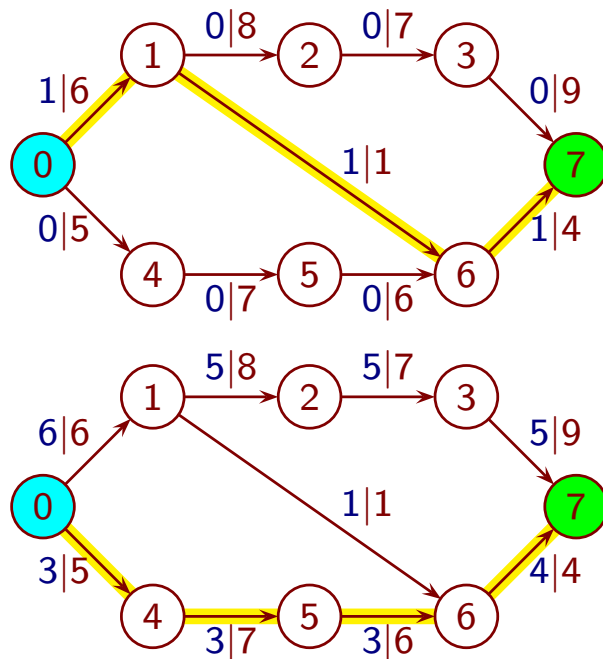
118 }
```

Ohjelma selvisi jokaisesta koeajosta alle 5 sekunnin

5.3 Sovellusesimerkki: maksimi vuo

Etsittävä mahdollisimman suuri vuo (**flow**) lähdöstä maaliin

- nesteen virtaus putkistossa, autojen virrat maantieverkossa, ...
- kutakin kaarta pitkin voi virrata enintään kaaren **kapasiteetin** (**capacity**) verran
- lähdöstä virtaa vain poispäin, maaliin virtaa vain sisään
- kuhunkin muuhun solmuun tulee yhteensä yhtä paljon kuin lähtee yhteensä
- minkään kahden solmun välillä ei saa olla kaarta molempiin suuntiin
 - tämä rajoitus voidaan kiertää lisäämällä jommankumman kaaren keskelle solmu
- jokainen solmu on jollain polulla lähdöstä maaliin

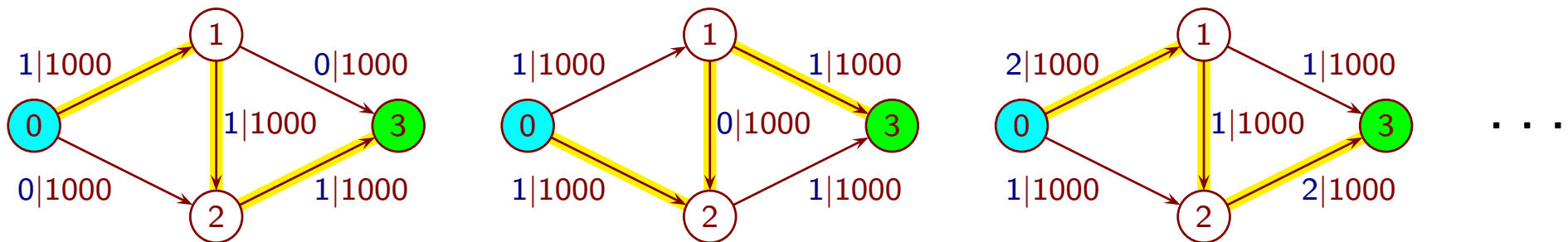


Edmonds–Karp-algoritmi (1972): toistetaan niin kauan kuin mahdollista

- etsitään mahdollisimman lyhyt polku lähdöstä maaliin, jossa kapasiteettia jäljellä
 - saattaa sisältää takaperoisen kaaren vuon vähentämisen
- kasvatetaan voita polun mukaisesti
- (ilman vaatimusta "mahdollisimman lyhyt" tämä tunnetaan Ford–Fulkerson-menetelmänä (1962))

Miksi mahdollisimman lyhyt polku?

- siten vältetään vuon kasvaminen alla olevan esimerkin tapaan hyvin hitaasti
- ⇒ algoritmi käyttää "vain" $O(|V||E|^2)$ aikaa
- (todistus on monimutkainen)



Tietorakenteet voidaan laskemiseksi toistuvilla leveyshauilla

- solmusta $u > 0$ lähtevät kaaret ovat vektorin kaaret lokeroissa $\text{solmut}[u-1].\text{kaarten_loppu}, \dots, \text{solmut}[u].\text{kaarten_loppu} - 1$
– jos i on em. välillä, niin kaari on $\text{kaaret}[i]$
- solmuun $u > 0$ tulevien kaarten numerot ovat vektorin vastak lokeroissa $\text{solmut}[u-1].\text{vastak_loppu}, \dots, \text{solmut}[u].\text{vastak_loppu} - 1$
– jos i on em. välillä, niin vastakaari on $\text{kaaret}[\text{vastak}[i]]$
- solmulla 0 lausekkeen $\text{solmut}[u-1].\dots_loppu$ tilalle tulee 0

```
1  struct kaarityyppi{
2      const unsigned alku, loppu, kapas; unsigned vuo;
3      kaarityyppi( unsigned s1, unsigned s2, unsigned kapas ):
4          alku( s1 ), loppu( s2 ), kapas( kapas ), vuo( 0 ) {}
5  };
6  std::vector< kaarityyppi > kaaret;

7  struct solmutyyppi{
8      unsigned kaarten_loppu, vastak_loppu, loytyi, reitti;
9      solmutyyppi( unsigned kl ):
10         kaarten_loppu( kl ), vastak_loppu( 0 ), loytyi( 0 ), reitti( ~0u ) {}
11  };
12  std::vector< solmutyyppi > solmut;
13  std::vector< unsigned > vastak;
```


Kaarten ja vastakaarten selaamista helpottavia toimintoja

```
14 unsigned kaari_solmu = 0, kaari_nyt = 0;
15 unsigned seur_kaari(){
16     if( kaari_nyt >= solmut[ kaari_solmu ].kaarten_loppu ){ return ~0u; }
17     return kaari_nyt++;
18 }
19 unsigned eka_kaari( unsigned s1 ){
20     kaari_solmu = s1; kaari_nyt = s1 ? solmut[ s1-1 ].kaarten_loppu : 0;
21     return seur_kaari();
22 }
23 unsigned vastak_solmu = 0, vastak_nyt = 0;
24 unsigned seur_vastak(){
25     if( vastak_nyt >= solmut[ vastak_solmu ].vastak_loppu ){ return ~0u; }
26     return vastak[ vastak_nyt++ ];
27 }
28 unsigned eka_vastak( unsigned s1 ){
29     vastak_solmu = s1; vastak_nyt = s1 ? solmut[ s1-1 ].vastak_loppu : 0;
30     return seur_vastak();
31 }
```

Vastakaaret voi muodostaa tehokkaasti laskentajärjestämisellä (luku 9.4)

- riveillä 43, ..., 46 korjataan se, että laskentajärjestäminen asettaa kunkin vastak_loppu osoittamaan vastakaarten alkua eikä loppua

```
32 void muodosta_vastakaaret(){
33     for( unsigned k1 = 0; k1 < kaaret.size(); ++k1 ){
34         ++solmut[ kaaret[ k1 ].loppu ].vastak_loppu;
35     }
36     for( unsigned s1 = 1; s1 < solmut.size(); ++s1 ){
37         solmut[ s1 ].vastak_loppu += solmut[ s1-1 ].vastak_loppu;
38     }
39     vastak.resize( kaaret.size() );
40     for( unsigned k1 = kaaret.size(); k1-->0; ){
41         vastak[ --solmut[ kaaret[ k1 ].loppu ].vastak_loppu ] = k1;
42     }
43     for( unsigned s1 = 1; s1 < solmut.size(); ++s1 ){
44         solmut[ s1-1 ].vastak_loppu = solmut[ s1 ].vastak_loppu;
45     }
46     solmut.back().vastak_loppu = kaaret.size();
47 }
```

Laskenta-algoritmin pääsilmmukan käynnistys

```
48 void laske_vuot(){
49     /* Käynnistä täydennyskierrokset. */
50     std::vector< unsigned > jono( solmut.size() );
51     unsigned kierros = 0;
52     while( true ){
```

Laskenta-algoritmin pääsilmmukan alkuosa: täydentävän polun etsintä leveyshaulla

- jos löydetty-tieto olisi bitti, niin se täytyisi nollata joka solmusta joka kierroksella
 - saattaa olla iso osa kokonaistyöstä, koska leveyshaku ei aina käy joka solmussa

⇒ käytämme bitin sijasta kokonaislukua

- solmu on löydetty jos ja vain jos sen loytyi = kierros

```
53      /* Etsi täydentävä polku */
54      ++kierros; jono[0] = lahto; unsigned nyt = 0, vapaa = 1;
55      while( nyt < vapaa ){
56          unsigned s1 = jono[ nyt++ ];
57          if( s1 == maali ){ break; }
58          for( unsigned k1 = eka_kaari( s1 ); k1 != ~0u; k1 = seur_kaari() ){
59              kaarityyppi & K1 = kaaret[ k1 ]; solmutyyppi & S1 = solmut[ K1.loppu ];
60              if( S1.loytyi == kierros || K1.vuo >= K1.kapas ){ continue; }
61              S1.loytyi = kierros; jono[ vapaa++ ] = K1.loppu; S1.reitti = k1;
62          }
63          for( unsigned k1 = eka_vastak( s1 ); k1 != ~0u; k1 = seur_vastak() ){
64              kaarityyppi & K1 = kaaret[ k1 ]; solmutyyppi & S1 = solmut[ K1.alku ];
65              if( S1.loytyi == kierros || K1.vuo == 0 ){ continue; }
66              S1.loytyi = kierros; jono[ vapaa++ ] = K1.alku; S1.reitti = k1;
67          }
68      }
69      if( solmut[ maali ].loytyi < kierros ){ break; }
```

Laskenta-algoritmin pääsilmmukan loppuosa

```
70      /* Etsi reitin lisäyksen arvo. */
71      unsigned s1 = maali, lisays = ~0u;
72      while( s1 != lahto ){
73          kaarityyppi & K1 = kaaret[ solmut[ s1 ].reitti ];
74          unsigned lisaysvara = s1 == K1.alku ? K1.vuo : K1.kapas - K1.vuo;
75          lisays = std::min( lisays, lisaysvara );
76          s1 = s1 == K1.alku ? K1.loppu : K1.alku;
77      }

78      /* Lisää lisäys voihin. */
79      s1 = maali;
80      while( s1 != lahto ){
81          kaarityyppi & K1 = kaaret[ solmut[ s1 ].reitti ];
82          if( s1 == K1.alku ){ K1.vuo -= lisays; s1 = K1.loppu; }
83          else{ K1.vuo += lisays; s1 = K1.alku; }
84      }

85  }

86  }
```

5.4 Yleinen graafihaku

Johdanto

- löytää kaikki solmut, jotka ovat saavutettavissa lähtösolmusta
- abstrakti algoritmi, joka voidaan toteuttaa monella eri tavalla
- konkreettisia toteutuksia mm. leveyshaku ja syvyyshaku

Solmujen värit

- **valkoinen**: ei vielä löydetty
- **harmaa**: löydetty, mutta ei loppuun käsitelty
- **musta**: loppuun käsitelty
- alussa lähtösolmu on harmaa ja muut solmut ovat valkoisia
- algoritmista näkee, että **värit voivat muuttua vain valkoinen $\rightsquigarrow$ harmaa $\rightsquigarrow$ musta**
- osoitamme, että **lopussa lähtösolmusta saavutettavat ovat mustia ja muut valkoisia**

Solmun tietoihin on tyypillisesti talletettu onko solmu löydetty

- harmaille solmuille on oma tietorakenne
 - leveyshaussa jono, syvyysaussa (mahdollisesti ajoympäristön) pino
- harmaa: löydetty ja harmaiden tietorakenteessa
- musta: löydetty ja ei harmaiden tietorakenteessa
- silmukoita etsittäessä solmuihin kannattaa tallettaa myös onko harmaa vai musta

Algoritmi: toista kunnes mikään solmu ei ole harmaa

- valitse jokin harmaa solmu
 - leveyshaussa valitaan jonon ensimmäinen
 - syvyysshaussa valitaan viimeisimpänä löydetty
- jos sillä ei ole tutkimattomia lähtökaaria, niin väritä se mustaksi (eli poista harmaiden tietorakenteesta)
- muussa tapauksessa tutki jokin sen tutkimaton lähtökaari
 - tutkiminen: jos sen kärkipään solmu on valkoinen, väritä se harmaaksi (eli merkitse löydetyksi ja lisää harmaiden tietorakenteeseen)
- kuinka löydetään tutkimaton lähtökaari?
 - leveyshaussa lähtökaaret voidaan tutkia peräkanaa yksinkertaisella silmukalla
 - syvyysshaussa asia on monimutkaisempi, ks. luku 5.5

Pysähtyminen

- joka kierroksella joko mustien solmujen määrä tai tutkittujen kaarten määrä kasvaa
- kukin solmu voidaan värittää mustaksi enintään kerran
- kukin kaari voidaan tutkia enintään kerran

Suoritus aika

- jos perustoiminnot ovat vakioaikaisia, niin suoritus aika on $O(|V| + |E|)$

Invariantti: Jokainen harmaa tai musta solmu on saavutettavissa lähtösolmusta.

- aluksi vain lähtösolmu on harmaa
- kun muu solmu väritetään harmaaksi, siihen on kaari harmaasta solmusta
- kun solmu väritetään mustaksi, se oli harmaa

Invariantti: Mistään mustasta solmusta ei ole kaarta valkoiseen solmuun.

- mustan solmun jokainen lähtökaari on tutkittu
 - tämä oli mustaksi värittämisen ehto
- tutkitun kaaren kärkisolmu ei ole valkoinen
 - oli ei-valkoinen tai väritettiin harmaaksi
 - muunvärinen ei muutu valkoiseksi

Lopussa kaikki lähtösolmusta saavutettavat solmut ovat mustia

- lopetusehdon vuoksi mikään solmu ei ole harmaa
 - solmu ei voi muuttua valkoiseksi, lähtösolmu on ollut harmaa $\Rightarrow$ se on musta
 - mistään mustasta solmusta ei ole kaarta valkoiseen solmuun
- $\Rightarrow$ ei voi olla polkua lähtösolmusta harmaaseen tai valkoiseen solmuun
- sellaisella polulla täytyisi olla jossakin kohtaa kaari mustasta solmusta valkoiseen

Koko graafin läpikäymiseksi kutsu hakua jokaisesta solmusta, joka on yhä valkoinen

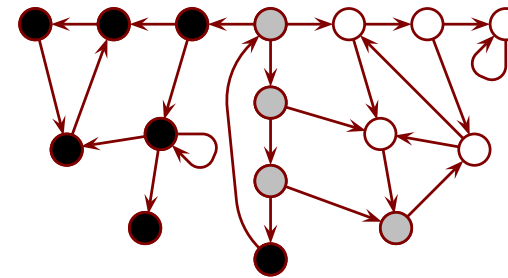
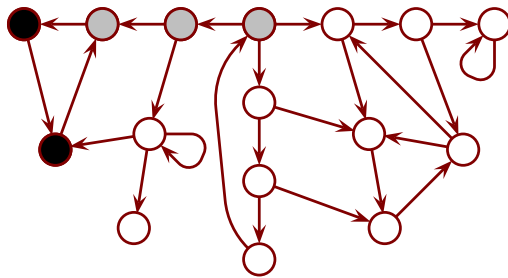
- esimerkki seuraavassa luvussa

5.5 Syvyyshaku

Syvyyshauulla voidaan löytää silmukoita

Syvyyshaussa harmaat solmut muodostavat polun lähtösolmusta nykyiseen solmuun

- tutkittavaksi otetaan aina polun viimeisen solmun lähtökaari
 - jos siten löytyy uusi solmu, se lisätään polun jatkoksi
- ⇒ saman solmun lähtökaaria ei tutkita peräkkäin (paitsi jos solmusta ei löydy uusia solmuja)
- kun solmun kaikki lähtökaaret on tutkittu, solmu poistetaan polusta



Helppo toteuttaa rekursiivisesti

```
SYVYYSHAKU(u)  
1  u.väri := harmaa  
2  for v := Solmut[u].eka_naapuri to Solmut[u].viim_naapuri do  
3      if v.väri = valkoinen then SYVYYSHAKU(v)  
4      else if v.väri = harmaa then löytyi silmukka  
5  u.väri := musta
```

Ei-rekursiivinen toteutus tarvitsee esim.

- pinon, jossa ylläpidetään harmaiden solmujen polkua
- solmuun tiedon, mitkä siitä lähtevät kaaret on tutkittu (esim. montako ensimmäistä)

Löytäessään harmaan solmun syvyyshaku löytää lähtösolmusta saavutettavan silmukan

- jos tutkittavan kaaren kärkipäässä on harmaa solmu, siitä pääsee em. polkua pitkin nykyiseen solmuun
- tutkittava kaari sulkee silmukan

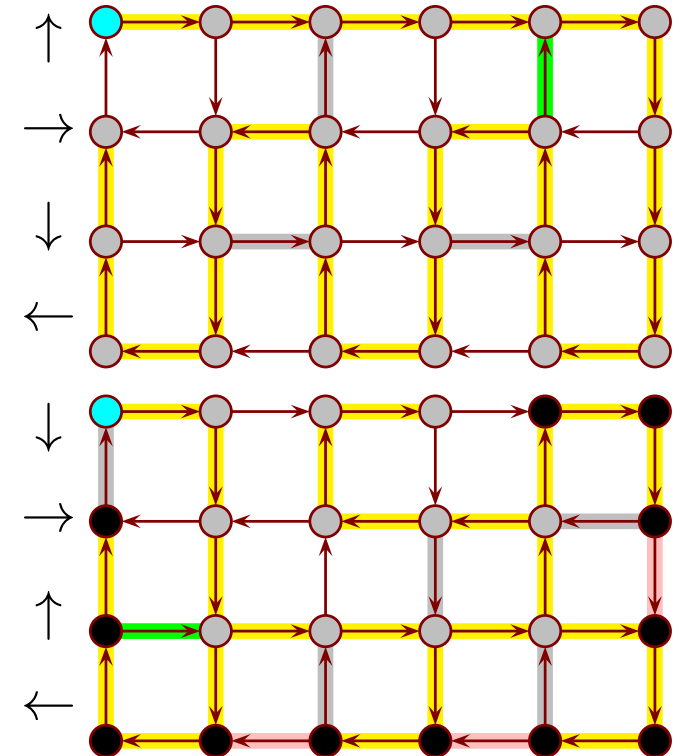
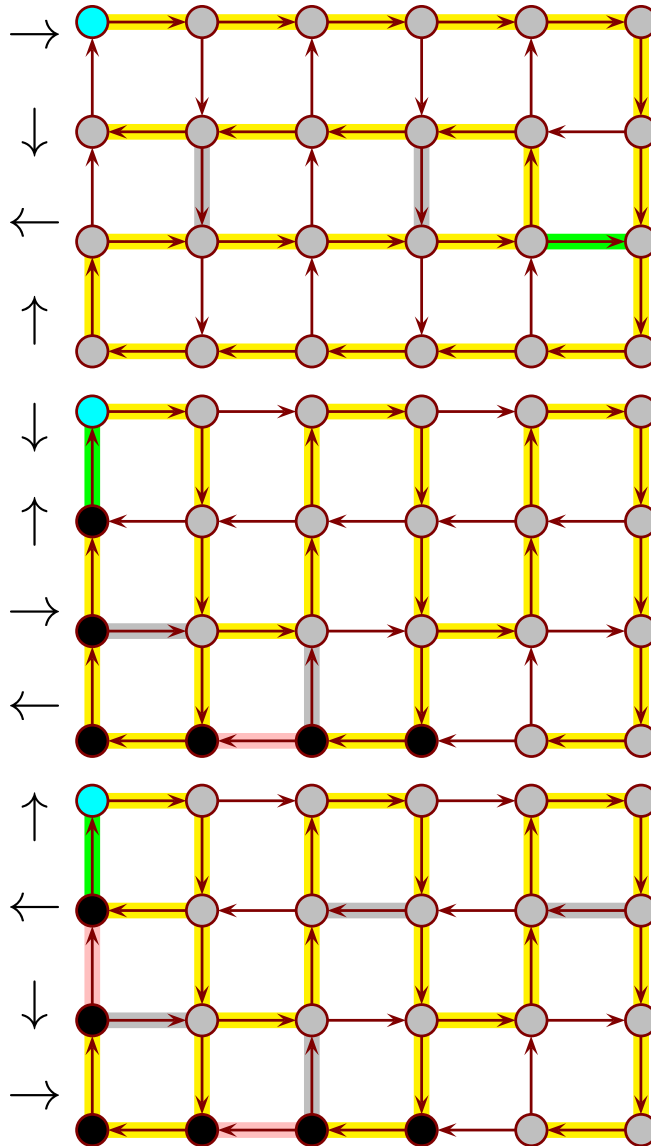
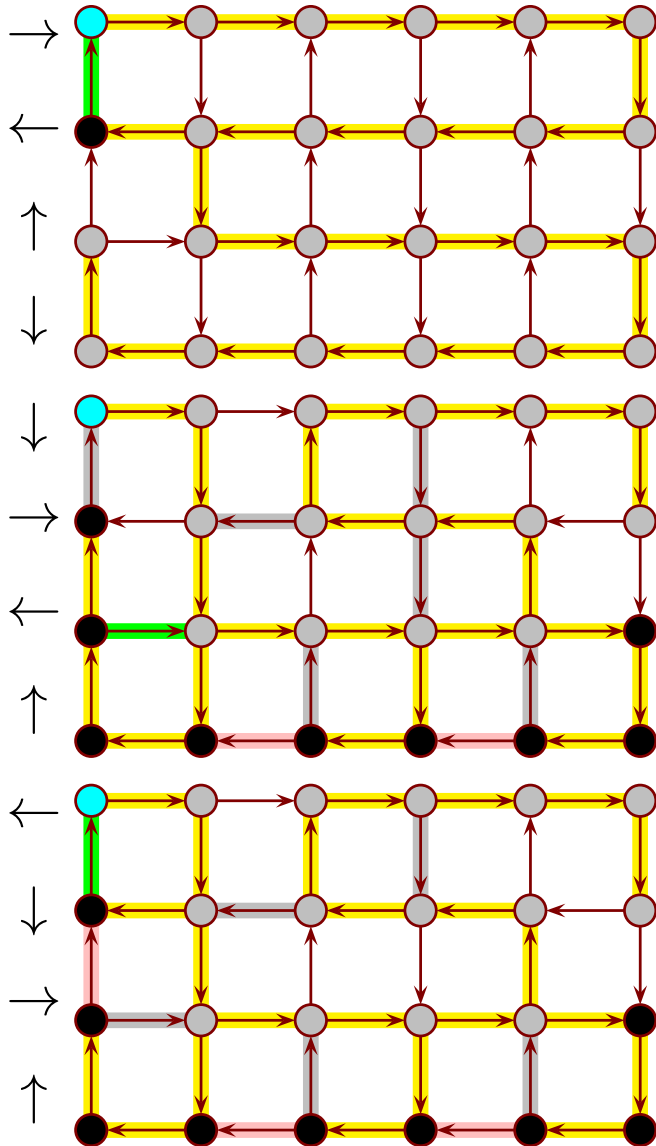
Syvyyshaku löytää jokaisesta lähtösolmusta saavutettavasta silmukasta ainakin yhden kaaren

- lähtösolmusta saavutettavan silmukan solmut väritetään aikanaan mustaksi
- jokin silmukan solmuista u on ensimmäinen, joka siitä silmukasta väritetään mustaksi
- olkoon v silmukassa seuraava solmu
- kun kaari $u \rightsquigarrow v$ tutkittiin, v ei ollut
 - musta, koska muutoin u ei olisi silmukan ensimmäinen mustaksi väritetty
 - valkoinen, koska muutoin haku olisi jatkanut v :hen ja aikanaan peruuttanut sieltä u :hun, jolloin se olisi värittänyt v :n mustaksi ennen u :ta

$\Rightarrow v$ oli harmaa

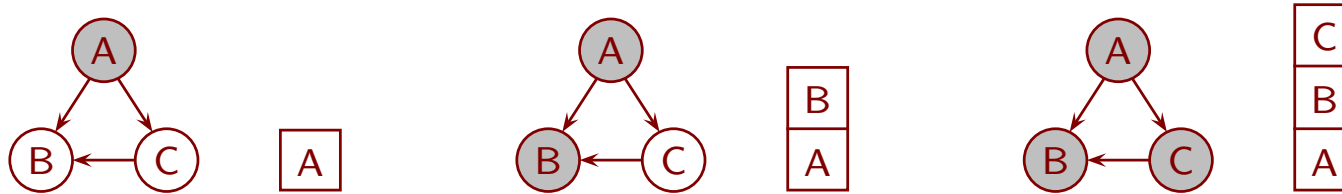
Polkuja viimeisen solmun löydyttyä, kun kaaret käsitellään eri järjestyksissä

- vihreät, harmaat ja pinkit kaaret tutkittiin, mutta niitä pitkin ei menty
- vihreä kaari sulkee ensimmäisenä löydetyn silmukan, harmaa sulkee muun silmukan



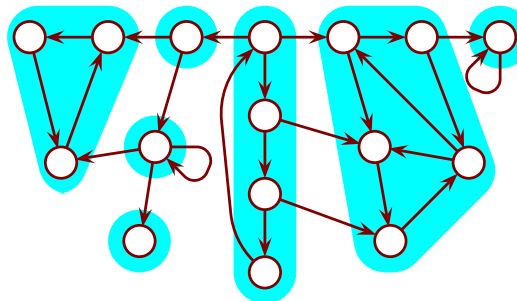
Kaikkien naapureiden laittaminen yhtäaikaan pinoon ei takaa silmukoiden löytymistä oikein

- lähtösolmu A pinoon
- A:sta lähtevät kaaret tutkitaan, B ja C pinoon
- C tutkitaan, siitä on kaari harmaaseen solmuun B vaikka ei ole silmukkaa



Suunnatun graafin toistottomien silmukoiden määrä voi olla valtava

- toistoton tarkoittaa, että mikään solmu ei toistu
 - edellisen ruudun 4×6 graafeissa on 88 toistotonta silmukkaa (selvitetty kokeellisesti)
 - vastaavassa 8×10 -graafissa on 199 387 034 toistotonta silmukkaa (selv. kokeellisesti)
- ⇒ tavallisesti ei kannata käsitellä jokaista toistotonta silmukkaa erikseen, vaan jokainen **vahvasti kytketty komponentti** (**strongly connected component**) erikseen
- mahdollisimman iso sellainen joukko solmuja, että jokaisesta on polku jokaiseen

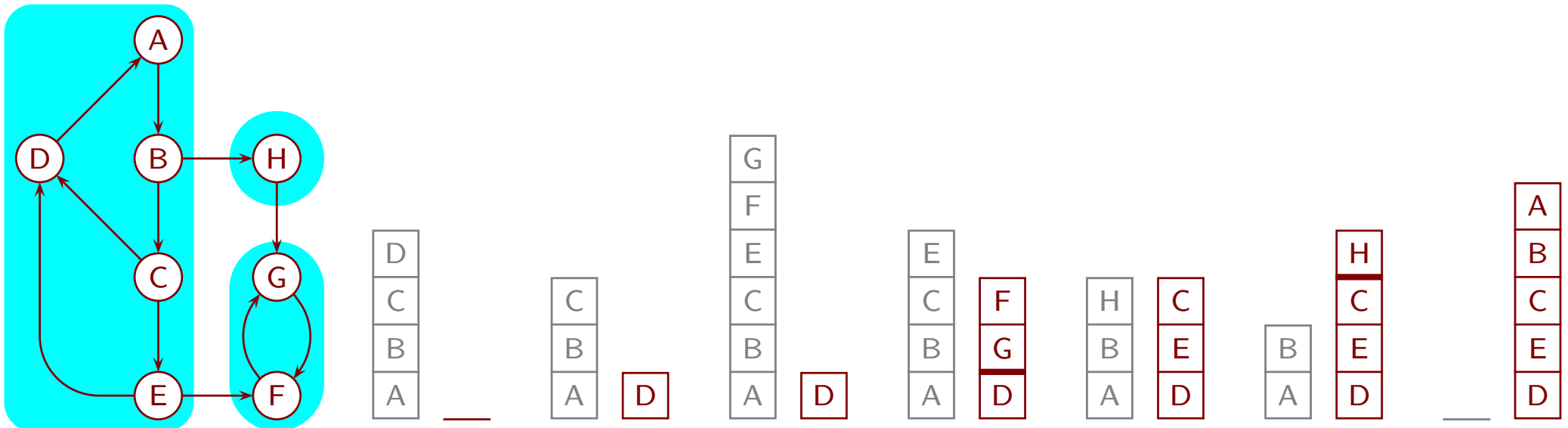


Kun syvyysshaun peruuttaessa komponentti vaihtuu, on jokainen nykyisestä solmusta saavutettava solmu loppuun käsitelty

- jos jokin nykyisestä saavutettava muu solmu olisi harmaa, olisi se silmukassa nykyisen solmun kanssa
⇒ komponentti ei vaihtuisi peruutuksessa
- loppuun käsitellyistä ei ole kaaria valkoisiin ⇒ nykyisestä saavutettavat eivät valkoisia

Seurauksia

- kun komponentista on peruutettu pois, on kompon. valmis eikä siihen enää palata
- jos komponentista pois peruuttamiset voidaan tunnistaa, voidaan komponentit löytää "ruskean pinon" avulla (eri pino kuin syvyysshaun pino)
 - solmu pannaan ruskeaan pinoon peruutettaessa (tai jo löydettyessä)
 - kun komponentti valmistuu, sen kaikki solmut poistetaan ruskeasta pinosta



Tarjanin algoritmi (1972) vahvasti kytkettyjen komponenttien löytämiseksi (parannuksin)

- syvyyshaku täydennettynä ruskealla pinolla ja kaksinkertaisella solmujen numeroinnilla

```
1  unsigned nro_nyt = 0;
2  unsigned DFS( unsigned uu ){
3      solmutyyppi & UU = solmut[ uu ];
4      if( UU.oma_nro != ~0u ){ return UU.pieni_nro; }
5      UU.oma_nro = UU.pieni_nro = nro_nyt++;
6      unsigned kk = uu ? solmut[ uu-1 ].kaarten_loppu : 0;
7      for( ; kk < UU.kaarten_loppu; ++kk ){
8          UU.pieni_nro = std::min( UU.pieni_nro, DFS( kaaret[ kk ] ) );
9      }
10     pino.push_back( uu );
11     if( UU.pieni_nro < UU.oma_nro ){ return UU.pieni_nro; }
12     while( !pino.empty() ){
13         unsigned vv = pino.back();
14         if( solmut[ vv ].pieni_nro < UU.oma_nro ){ break; }
15         std::cout << ' ' << solmut[ vv ].nimi << ' ' << solmut[ vv ].pieni_nro;
16         solmut[ vv ].pieni_nro = ~0u; pino.pop_back();
17     }
18     std::cout << '\n'; return ~0u;
19 }
```

Aliohjelman DFS kutsut

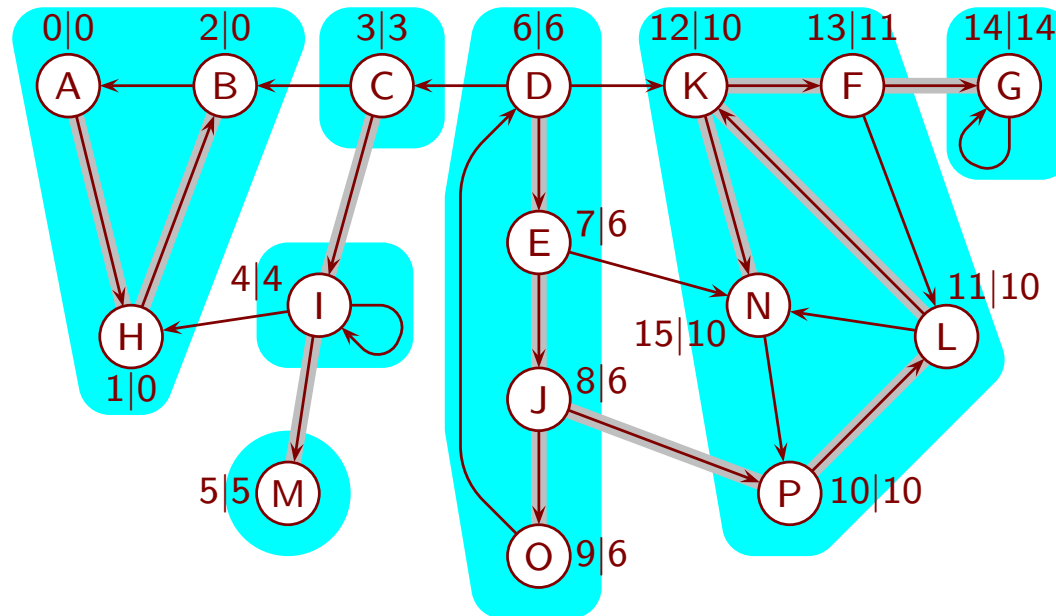
```
for( unsigned uu = 0; uu < solmut.size(); ++uu ){ DFS( uu ); }
```

Toiminnan perusideat

- solmujen värit
 - **valkoinen**: ei vielä löydetty; jos ja vain jos $\text{oma_nro} = \sim 0u$
 - **harmaa**: löydetty, mutta ei peruutettu
 - **ruskea**: peruutettu, mutta solmun komponentti ei vielä ole valmis
 - **musta**: loppuun käsitelty; jos ja vain jos $\text{pieni_nro} = \sim 0u$ ja $\text{oma_nro} \neq \sim 0u$
- oma_nro on harmaaksi värittämisen hetkellä asetettu juokseva numero
- mustan solmun pieni_nro on $\sim 0u$ ja valkoiselle ei ole määritelty
- muun solmun pieni_nro on pienin solmusta mustia välttämällä saavutetun oma_nro
 - aluksi solmun oma_nro
 - sen jälkeen voi pienentyä mutta ei kasvaa ennen kuin solmusta tulee musta
 - päivitetään kun lähtökaaren päästä löytyy harmaa tai ruskea solmu, tai solmuun peruutetaan sellaisesta (mustasta tuleva $\sim 0u$ ei voi aiheuttaa päivitystä)
- vahvasti kytketyn komponentin valmistuminen tunnistetaan siitä, että solmusta peruutettaessa sen oma_nro on yhtäsuuri kuin pieni_nro
 - perustelemme kohta, että tämä testi toimii oikein
 - silloin ruskeasta pinosta poistetaan solmut, joiden pieni_nro on vähintään nykyisen oma_nro

Esimerkki syntyvästä numeroinnista

- tässä ja jatkossa mustan solmun `pieni_nro`:na käytetään viimeisintä `pieni_nro`:a ennen mustaksi värittämistä



Helppoja invariantteja riveillä 1, ..., 9

- kustakin solmusta on polku `pieni_nro`-solmuunsa
- solmu on ruskeassa pinossa jos ja vain jos se on ruskea
- ruskealle solmulle `pieni_nro` < `oma_nro`

Invariantti: Jos u peri `pieni_nro`:n kaarta $u \rightsquigarrow v$ pitkin, niin u ja v ovat samassa komponentissa

- perimishetkellä u on uusin yhä harmaa
- koska invariantti päti aiemmin, on v samassa komponentissa `pieni_nro`-solmunsa kanssa, se oman `pieni_nro`-solmunsa kanssa jne.
 - v on perinyt `pieni_nro`:sa jotain polkua pitkin, jne.
- koska numerot eivät voi pienentyä loputtomiin, jollekin niistä `pieni_nro` = `oma_nro`
 - se ei ole musta, koska muutoin myös v olisi musta, joten u ei perisi siitä
 - se ei ole ruskea, koska ruskeille `pieni_nro` < `oma_nro` $\Rightarrow$ se on harmaa

$\Rightarrow$ siitä on harmaa polku u :hun

$\Rightarrow$ se, u ja v ovat samassa komponentissa

Jos testi katsoo solmun komponentin keskeneräiseksi, niin se on oikeassa

- edellä kerrotun perusteella u on samassa komponentissa `pieni_nro`-solmunsa `pieni_nro`-solmun `pieni_nro`-solmun ... kanssa
- se on harmaa

Jos testi katsoo solmun komponentin valmistuneeksi, niin se on oikeassa

- olkoon juuri ennen peruutusta solmusta (toistoton) polku muuhun harmaaseen
- jokin solmu on ensimmäinen tämän polun muu harmaa
- sen `oma_nro` on pienempi kuin nykyisen solmun, koska nykyinen on uusin yhä harmaa
- se on löytynyt nykyisestä käsin, koska ruskean, mustan ja nykyisen naapurit eivät ole valkoisia

⇒ sen numero korvautui vielä pienemmällä, tai periytyi nykyiseen sitä polkua pitkin, jota pitkin se löytyi nykyisestä

⇒ nykyiselle solmulle `pieni_nro` < `oma_nro`

Ajan ja lisämuistin kulutus $O(|V| + |E|)$

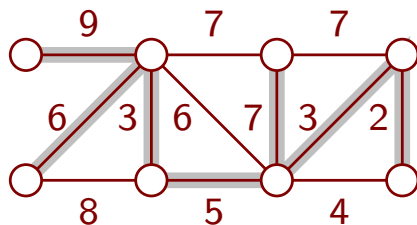
5.6 Minimi virittävä puu

Taustakäsitteitä

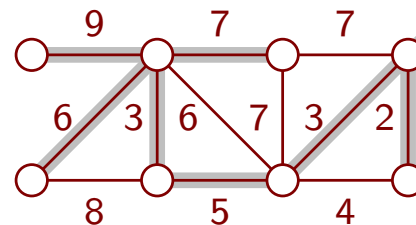
- **suuntaamaton graafi** (*undirected graph*) on muuten kuten suunnattu graafi, mutta kaarilla ei ole suuntaa
⇒ ei piirretä nuolenkärkiä
- **painotetun** (*weighted*) graafin jokaiseen kaareen on liitetty luku
– voi tarkoittaa esim. kaaren esittämän tienpätkän pituutta
- suuntaamaton graafi on **kytketty** (*connected*), jos ja vain jos jokaisesta solmusta on polku jokaiseen
- kytketyn suuntaamattoman graafin **virittävä puu** (*spanning tree*) on (kaikki solmut sekä) sellainen silmukaton joukko kaaria, että jokaisesta solmusta on polku jokaiseen
- virittävässä puussa on kaaria yksi vähemmän kuin graafissa solmuja
– jokainen solmu paitsi ensimmäinen tarvitsee yhden kaaren kytkemään se muihin

Määritelmä

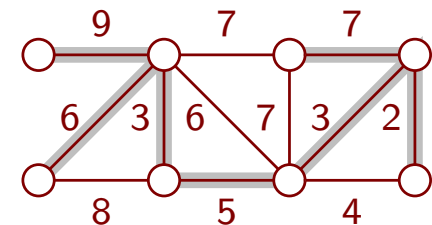
- **minimi virittävä puu** (*minimum spanning tree*) on sellainen, jossa kaarten painojen summa on mahdollisimman pieni



tai



tai



Kruskalin algoritmi (1956)

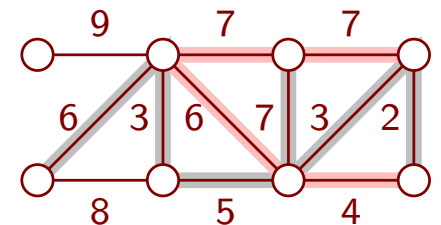
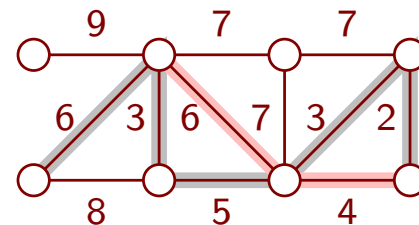
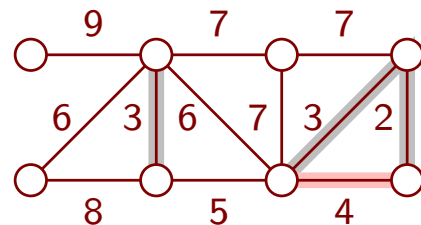
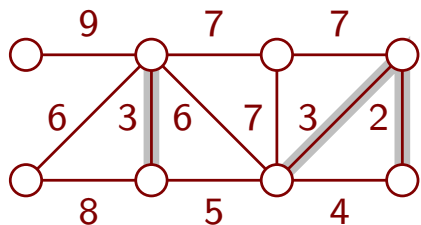
- käy kaaret läpi painon mukaisessa järjestyksessä
 - ota mukaan ne ja vain ne, jotka yhdistävät kaksi siihen asti erillistä komponenttia

Löytää virittävän puun

- yhdistää vain erillisiä komponentteja $\Rightarrow$ ei muodosta silmukoita
- kunkin kaaren päät yhdistetään polulla viimeistään kun kaari tulee tutkintavuoroon $\Rightarrow$ jokaiselle polulle syntyy korvaava polku

Invariantti: on olemassa minimi virittävä puu, johon jokainen valittu kaari kuuluu

- aluksi triviaalisti totta, koska ei ole valittu yhtään kaarta
 - uusi valittu kaari ei välttämättä kuulu edellisellä kierroksella ajateltuun puuhun
 - siinä puussa on uuden kaaren päiden komponentit yhdistävä polku ja siinä ainakin yksi tutkimaton kaari
 - uuden kaaren paino on enintään sama kuin ko. kaaren
- $\Rightarrow$ korvaamalla ko. kaari uudella kaarella saadaan yhtä hyvä puu
- $\Rightarrow$ invariantti säilyy uuden kaaren lisäyksessä

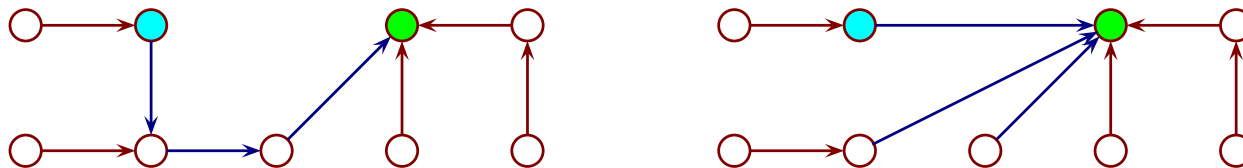


Miten tunnistetaan tehokkaasti ovatko kaaren päiden komponentit sama vai eri?

- "union-find-tietorakenne"
- jokaiseen solmuun liitetään linkki kohti "edustajaa" sekä "korkeusarvio"
- kaksi solmua kuuluu samaan komponenttiin jos ja vain jos niillä on sama edustaja
- aluksi jokainen linkki osoittaa solmuun itseensä (tai tyhjään) ja korkeusarvio on 0
- kun kaksi komponenttia yhdistetään, käännetään arviolta matalamman edustajan linkki osoittamaan arviolta korkeamman edustajaa
 - jos korkeusarviot ovat yhtäsuuret, edustajan korkeusarviota kasvatetaan yhdellä



- etsinnässä kohdatut linkit käännetään osoittamaan suoraan edustajaa
 - ⇒ myöhemmät etsinnät nopeutuvat
 - jollei linkkejä käännettäisi etsinnässä, olisivat korkeusarviot aitoja korkeuksia



- kunkin testin tasatuksi suoritusajaksi tulee hitusen hitaampi kuin vakio

Tällä toteutuksella Kruskalin algoritmin ajan kulutus on $O(|E|\log|V|)$

- hitainta on kaarten järjestäminen, $O(|E|\log|E|)$
- nyt $|E| \leq |V|^2$, joten $\log|E| \leq 2\log|V| = O(\log|V|)$

Primin algoritmi (Jarník 1930, Prim 1957)

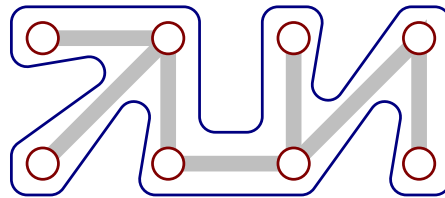
- aloitetaan mielivaltaisesta solmusta
- joka kierroksella otetaan mukaan mahdollisimman lyhyt kaari, jonka yksi mutta ei toinen pää on jo mukana
- voidaan toteuttaa Dijkstra'n algoritmin kaltaisesti

Minimin virittävän puun soveltaminen kauppamatkustajan ongelmaan

- kauppamatkustajan ongelmassa pitää löytää mahdollisimman lyhyt polku, joka käy jokaisessa solmussa ja palaa aloitussolmuun
- kaarten painot (= pituudet) ei-negatiivisia
- ongelmasta on useita muunnelmia
 - graafi on suunnattu / suuntaamaton
 - solmussa saa / ei saa käydä useasti
 - kysytään polkua / onko olemassa polkua, jonka pituus on enintään annettu luku

Minimin virittävän puun avulla saadaan käyttökelpoinen likiarvo versiolle suuntaamaton, saa käydä useasti, kysytään polkua

- kierretään minimi virittävä puu kulkemalla sen jokainen kaari edestakaisin



- tämä reitti on korkeintaan kaksi kertaa mahdollisimman lyhyen reitin pituinen
 - kun kauppamatkustajan reitistä poistetaan kaaria niin että jäljelle jää puu, saadaan virittävä puu
- ⇒ kauppamatkustajan reitti on ainakin yhtä pitkä kuin minimi virittävä puu
- likiarvoa voi parantaa heuristiikoilla ja järjestelmällisellä haualla

6 Ahneet algoritmit ja dynaaminen ohjelmointi

6.1	Ahneet algoritmit	161
6.2	Hitaat rekursiiviset	165
6.3	Dynaaminen ohjelmointi	172

6.1 Ahneet algoritmit

Ahne algoritmi (*greedy algorithm*) tarkoittaa algoritmia, joka tekee aina sen mikä näyttää välittömästi parhaalta

- ei välttämättä löydä optimaalista eli kaiken kaikkiaan parasta tulosta
- joistakin ahneista algoritmeista voidaan todistaa, että ne löytävät optimaalisen

Esimerkki: rahasumma mahdollisimman pienellä määrällä kolikoita

- rahasumma on ei-negatiivinen kokonaisluku
- taulukko *arvot* sisältää kolikoiden arvot laskevassa järjestyksessä, viimeisenä 1

```
1  i := 0
2  while jäljellä > 0 do
3      while arvot[i] > jäljellä do i := i + 1
4      tulosta arvot[i] ; jäljellä := jäljellä − arvot[i]
```

- jos *arvot* = [50, 25, 10, 2, 1], niin
 - ym. algoritmi tuottaa $30 = 25 + 2 + 2 + 1$, siis neljä kolikkoa
 - $30 = 10 + 10 + 10$ olisi vain kolme kolikkoa
- jos *arvot* = [50, 20, 10, 5, 2, 1], niin ym. algoritmi tuottaa optimaalisen tuloksen
 - 1-senttisiä kannattaa olla korkeintaan 1 (muutoin kannattaisi vaihtaa $1+1 \rightsquigarrow 2$)
 - 1- ja 2-senttisiä kannattaa olla korkeintaan 2 (muutoin $2+2+2 \rightsquigarrow 5+1$ ja ...)
 - 5-senttisiä kannattaa olla korkeintaan 1 (muutoin $5+5 \rightsquigarrow 10$)
 - ...

Aiemmin käsiteltyjä ahneita algoritmeja

- Dijkstra ja admissible-heuristiikalla A*: valitse aina tutkimaton solmu, johon on lähdöstä mahdollisimman lyhyt matka / mahdollisimman pieni kokonaismatkan arvio
- Kruskal ja Prim: valitse aina mahdollisimman lyhyt ehdot täyttävä kaari

Juomapaikkatehtävä

- juoksureitin varrelle halutaan laittaa *tauvoja* paikkaa, joissa tarjotaan juotavaa
- ne vaativat tilaa ja kulkuyhteyksiä
⇒ ei voi laittaa mihin tahansa
- ⇒ mahdolliset paikat on ilmoitettu matkana lähdöstä lukien taulukossa *sijainnit*
 - *sijainnit*[0] = 0 on lähtö, *sijainnit*[*sijainnit.koko* – 1] on maali
- ”osuus” on matka lähdöstä ensimmäiseen juomapaikkaan, juomapaikasta seuraavaan tai viimeisestä juomapaikasta maaliin
- halutaan minimoida pisimmän osuuden pituus

Juomapaikkojen sijoittaminen mahdollisimman lähelle tasajakoa ei välttämättä ole optimaalista

- esim. kolme juomapaikkaa, *sijainnit* = [0, 9, 18, 27, 35, 44, 52]
- tasajako tuottaisi [0, 13, 26, 39, 52]
- niitä lähinnä ovat [0, 9, 27, 35, 52], jolloin pisimmän osuuden pituus on 18
- valinnoilla [0, 9, 18, 35, 52] pisimmän osuuden pituus on 17

Ahne algoritmi, joka selvittää, riittääkö pisimmän osuuden pituudeksi *pituus*, jos juomapaikkoja on enintään *tauvoja* kpl

- sijoita seuraava juomapaikka niin kauas edellisestä kuin *pituus* sallii
- *i* on yhtä suurempi kuin viimeisimmän juomapaikan indeksi taulukossa *sijainnit*
- vaikka silmukat ovat sisäkkäin, on ajan kulutus vain $O(\text{tauvoja} + \text{sijainnit.koko})$
 - rivin 4 loppuosa suoritetaan kaikkiaan alle *sijainnit.koko* kertaa

```
1  i := 1
2  for j := 0 to tauvoja do
3      raja := sijainnit[i - 1] + pituus
4      while i < sijainnit.koko && sijainnit[i] ≤ raja do i := i + 1
5  if i = sijainnit.koko return true else return false
```

Juomapaikat voidaan sijoittaa optimaalisesti kutsumalla tätä algoritmia eri pituuksilla kunnes lyhin *pituus* on löytynyt, jolla se palauttaa true

- koko reitin *pituus* *sijainnit*[*sijainnit.koko* - 1] riittää varmasti
- se jaettuna (*tauvoja* + 1):llä pyöristettynä ylös on pienin, jota kannattaa kokeilla
- ikävä kyllä seuraava on aika hidas sijainneilla [0, 1, 1000000000] ja yhdellä tauolla

```
1  for pituus :=  $\left\lceil \frac{\text{koko\_reitin\_pituus}}{\text{tauvoja}+1} \right\rceil$  to koko_reitin_pituus do
2      edeltä rivit 1, ..., 4
3      if i = sijainnit.koko then return pituus
```

- kokeiltavien pituuksien valinta puolitushaun kaltaisesti on parempi

```
1  ylä := koko_reitin_pituus ; ala :=  $\left\lceil \frac{\textit{ylä}}{\textit{tauvoja}+1} \right\rceil$ 
2  while ala < ylä do
3      pituus :=  $\left\lfloor \frac{\textit{ala}+\textit{ylä}}{2} \right\rfloor$  ; edeltä rivit 1, ..., 4
4      if i = sijainnit.koko then ylä := pituus else ala := pituus + 1
5  return ala
```

6.2 Hitaat rekursiiviset

Vaihtorahatehtävälle on helppo laatia rekursiivinen vaihtoehtot kokeileva algoritmi

- lanttien arvot laitetaan taulukkoon pienimmästä alkaen
- $\sim 0_u$ tarkoittaa, että summaa ei voi muodostaa
 - esim. summaa 3 ei voi muodostaa, jos lantit ovat $[2, 5, 10, 20, 50]$
- jos tavoitellaan summaa 0, niin ei tarvita yhtään lanttia
- muutoin päätetään joko käyttää lanttia nyt tai ei enää koskaan
 - käyttäminen myöhemmin mutta ei nyt ei tuottaisi lisää eri summia
- ⇒ ohjelmaa voi nopeuttaa jättämällä sellaiset mahdollisuudet tutkimatta
- jos jotakin lanttia käytetään nyt, niin jäljelle jää vaihtorahatehtävä lantin arvon verran pienemmälle summalle (rivit 7, ..., 10)
 - onnistuneeseen tulokseen pitää lisätä 1 kuvaamaan sen lantin käyttöä
- jos jotakin lanttia ei enää käytetä, niin jäljelle jää vaihtorahatehtävä entiselle summalle mutta se lantti kiellettyinä (rivit 11, ..., 13)
- jos oli mahdollista sekä käyttää että jättää käyttämättä ko. lanttia, niin valitaan niin saaduista tuloksista pienempi

Tässä vaiheessa ei tulosteta valintaa vaan vain vertaillaan suoritusajkoja

- tehtävä ratkaistaan aina vain isommalle summalle kunnes aika loppuu
- rivillä 17 tulostetaan käynnistyskäsky, eli suoritettavan ohjelman tiedostonimi

Perusversio C:stä perittyjä taulukoita käyttäen

```
1  #include <iostream>
2  const unsigned lantit[] = { 1, 2, 10, 25, 50 }, lantteja = 5;
3  const unsigned max_summa_1 = 1 + 100000;
4  unsigned vaihtoraha( unsigned summa, unsigned suurin ){
5      if( summa == 0 ){ return 0; }
6      unsigned tulos = ~0u;
7      if( summa >= lantit[ suurin ] ){
8          tulos = vaihtoraha( summa - lantit[ suurin ], suurin );
9          if( tulos != ~0u ){ ++tulos; }
10     }
11     if( suurin > 0 ){
12         tulos = std::min( tulos, vaihtoraha( summa, suurin - 1 ) );
13     }
14     return tulos;
15 }
16 int main( int, char *argv[] ){
17     std::cout << argv[0] << '\n';
18     for( unsigned summa = 0; summa < max_summa_1; ++summa ){
19         std::cout << summa << ": " << vaihtoraha( summa, lantteja-1 ) << '\n';
20     }
21 }
```

Kuinka pitkälle ehtii annetussa ajassa?

- kolmen mittauksen mediaani
- lisäsekunti tuo vain vähän lisähyötyä, ja uusi lisäsekunti vielä vähemmän

suoritus aika	1 s	2 s	3 s	4 s	5 s	6 s	7 s
perusversio	377	426	458	482	501	518	532
rajaava 1	575	681	752	806	850	889	922
rajaava 2	1 807	2 046	2 195	2 310	2 401	2 476	2 545
ennuste 2	1 809	2 043	2 194	2 308	2 400	2 478	2 546

Miksi ohjelma on näin tehoton?

- koska se ratkaisee samoja osaongelmia uudelleen ja uudelleen
- esim. jos aluksi $summa = 100$, niin se kutsuu seuraavasti siten, että $summa = 50$
 - 1 kerran niin että $suurin = 4$ (eli 50-senttinen)
 - 2 kertaa niin että $suurin = 3$: edellisestä ja kutsusta (75,3)
 - 3 kertaa niin että $suurin = 2$: kahdesti edellisestä ja kerran kutsusta (60,2)
 - 8 kertaa niin että $suurin = 1$: kolmesti edellisestä ja viidesti kutsusta (52,2)
 - 107 kertaa niin että $suurin = 0$
- jos aluksi $summa = 100$, niin se kutsuu 637 kertaa (0,0)

Ennustettu suorituskyky

- jos aluksi $summa = n$ ja $lantteja = \ell$, niin kutsuja tulee karkeasti $\sim n^\ell$
- pätee

$$\left(\frac{n}{2}\right)^{\ell+1} \leq \left\lceil \frac{n}{2} \right\rceil \underbrace{\left(\frac{n}{2}\right)^\ell}_{\lfloor \frac{n}{2} \rfloor} = \underbrace{0 + \dots + 0}_{\lfloor \frac{n}{2} \rfloor} + \underbrace{\left(\frac{n}{2}\right)^\ell + \dots + \left(\frac{n}{2}\right)^\ell}_{\lceil \frac{n}{2} \rceil} \leq 1^\ell + 2^\ell + \dots + n^\ell \leq nn^\ell = n^{\ell+1}$$

$\Rightarrow$ kun suoritetaan kasvavalla $summa$ kunnes t sekuntia on kulunut,

$$t \sim 1^\ell + 2^\ell + \dots + n^\ell \sim n^{\ell+1} \Rightarrow n \sim t^{\frac{1}{\ell+1}}$$

- edellä $\ell = 5$, joten tämä ennustaa $n \sim t^{0,167}$
- pienimmän neliösumman menetelmä on keino löytää mittaustuloksia mahdollisimman hyvin vastaava suora
- sillä saadaan $\ln n \approx \ln 376,936 + 0,177146 \ln t$
 - siksi $n \approx 376,936 t^{0,177146}$
 - tämä lauseke kokonaisluvuksi pyöristettynä tuottaa tarkalleen rivin "perusversio"!
 - $t = 20$ tuottaa ennuste 641, mitattu 639
 - $t = 100$ tuottaa ennuste 852, mitattu 845

$\Rightarrow$ eksponentin ennustimme vain likimain, mutta tuloksen muoto at^b toimii hyvin

Idea: keskeytetään haku, kun lantteja on käytetty yhtä paljon kuin parhaassa tunnetussa

- ohjelma "rajaava 2"
- lisätään parametri raja kertomaan, montako lanttia saa enintään käyttää
- aluksi raja = summa
- jos kutsussa raja = 0, niin poistutaan heti ja ilmoitetaan epäonnistuminen
- rivillä 8 rekursiivisessa kutsussa raja:a vähennetään yhdellä
- jos rivillä 8 saatiin raja:a parempi tulos, niin korvataan raja sillä
- merkittävä parannus nopeuteen (siihen, mihin saakka ehditään annetussa ajassa)
- nyt pienimmän neliösumman menetelmä tuottaa $n \approx 1809,13 t^{0,175684}$
 - $t = 20$ tuottaa ennuste 3062, mitattu 3032
 - $t = 100$ tuottaa ennuste 4063, mitattu 3957

"Rajaava 1" on melkein samanlainen

- se yrittää ensin ilman suurinta sallittua lanttia ja vasta sitten sen kanssa
- huomaamme, että sillä kumpaa yritetään ensin, on suuri vaikutus ajan kulutukseen
- sille suorituskyyennuste on $n \approx 575,403 t^{0,242709}$, 1191 / 1182, 1760 / 1704
 - sekin on lähes täydellisen tarkka kun t on välillä 1, ..., 7
 - eksponentissa on iso ero aiempiin

Kun $t = 20$ tai $t = 100$, kaikki kolme ennustetta tuottavat liian suuren tuloksen

Seuraavassa luvussa kehitämme nopeamman ratkaisun

Hidas rekursiivinen juomapaikkatehtävän ratkaisu

- valitsee juomapaikoille sijainnit lähdöstä kohtaan *paikka* saakka, kun juomapaikkoja pitää olla *tauvoja* kpl
 - palauttaa pisimmän osuuden pituuden
 - jos juomapaikkoja ei ole lupa lisätä, pisin osuus on tietenkin koko matka (rivi 2)
 - *lyhin* pitää kirjata parhaasta tuloksesta
 - rivillä 3 se asetetaan niin suureksi, että rivin 6 loppu varmasti suoritetaan joskus
 - sitten kokeillaan kaikki tutkittavalle välille mahdolliset juomapaikat
 - $\text{sijainnit}[\text{paikka}] - \text{sijainnit}[i]$ on osuuden pituus siitä tutkittavan matkan loppuun
 - $\text{PARAS}(i, \text{tauvoja} - 1)$ on optimoitu alkumatkan pisimmän osuuden pituus
 - optimaalinen juomapaikan sijainti talletetaan taulukkoon *valittu*
 - ei tarvita kun $\text{tauvoja} = 0$
- ⇒ säästetään yksi rivi indeksoimalla luvulla $\text{tauvoja} - 1$ eikä tauvoja

```
1  PARAS(paikka, tauvoja)
2  if tauvoja < 1 then return sijainnit[paikka]
3  lyhin := sijainnit[paikka] + 1
4  for i := 0 to paikka do
5      pisin := MAX( sijainnit[paikka] - sijainnit[i], PARAS(i, tauvoja - 1) )
6      if pisin < lyhin then lyhin := pisin ; valittu[paikka, tauvoja - 1] := i
7  return lyhin
```

- olisi parempi mutta monimutkaista etsiä optimi puolitushaulla
 - paras saattaa olla suurin jolla alkumatka on lyhyempi kuin viimeinen osuus, tai sitä seuraava
 - emme tule saamaan luvun 6.1 versioon nähden kilpailukykyistä edes luvun 6.3 keinoilla
- ⇒ emme tarkastele tätä ajatusta tämän enempää
- juomapaikkojen tulostaminen

```

8   TULOSTA__PAIKAT(paikka, taukoja)
9   if taukoja > 0 then TULOSTA__PAIKAT(valittu[paikka, taukoja - 1], taukoja - 1)
10  tulosta sijainnit[paikka]

```

- pääohjelma
 - tulostaa pisimmän osuuden pituuden
 - tulostaa juomapaikat mukaan lukien maali mutta ei lähtöä (joka on aina 0) (paitsi jos lähtö on valittu useaksi juomapaikaksi)

```

11  tulosta paras(sijainnit.koko - 1, taukoja)
12  TULOSTA__PAIKAT(sijainnit.koko - 1, taukoja)

```

6.3 Dynaaminen ohjelmointi

Kaksi tapaa: välitulokset muistetaan (*memoization*) tai lasketaan etukäteen

⇒ säästetään valtava määrä samojen välitulosten laskemista uudestaan ja uudestaan

Välitulokset muistava ratkaisu vaihtorahatehtävälle

- talletetaan jo lasketut välitulokset taulukkoon tulokset
 - ~1u tarkoittaa, että välitulosta ei ole vielä laskettu

```
1  unsigned tulokset[ max_summa_1 ][ lantteja ];
2  unsigned vaihtoraha( unsigned summa, unsigned suurin ){
3      unsigned tulos = tulokset[ summa ][ suurin ];
4      if( tulos != ~1u ){ return tulos; }
5      if( summa == 0 ){ tulokset[ summa ][ suurin ] = 0; return 0; }
6      tulos = ~0u;
7      if( summa >= lantit[ suurin ] ){
8          tulos = vaihtoraha( summa - lantit[ suurin ], suurin );
9          if( tulos < ~1u ){ ++tulos; }
10     }
11     if( suurin > 0 ){
12         tulos = std::min( tulos, vaihtoraha( summa, suurin - 1 ) );
13     }
14     tulokset[ summa ][ suurin ] = tulos; return tulos;
15 }
```

Mittaustulosten vertailukelpoisuuden vuoksi tulokset nollataan ~1u:ksi pääohjelman joka kierroksella

- kolmen mittauksen mediaani pyöristettynä kahteen merkitsevään numeroon

suoritus aika	1 s	2 s	3 s	4 s	5 s	6 s	7 s
muistava	11 000	17 000	22 000	26 000	30 000	33 000	36 000
taulukkoiva	16 000	27 000	35 000	43 000	49 000	55 000	61 000
yksiulotteinen	25 000	36 000	44 000	52 000	58 000	64 000	69 000

Kahdesta syystä emme laadi aikaennusteita näille ohjelmille

- edellä käytetyn ennustemallin perustelu ei ole niille pätevä
- emme saa "tarkastusmittauksia" $t = 20$ ja $t = 100$ kuin yhden, koska taulukolle varattu koko 100 000 ylittyy

Vaihtoehto välitulosten muistamiselle on täyttää taulukko ennalta päätetyssä järjestyksessä

- ensin reunat: käytössä on vain yhtä lanttia; tavoiteltu summa on nolla
- sitten muu taulukko jo täytettyjen avulla
- tällaisenaan ajo 100 000 saakka vei aikaa noin 16 s, rivit 6 ja 7 vaihdettuina noin 25 s

```
1  unsigned vaihtoraha( unsigned summa, unsigned suurin ){
2      for( unsigned ss = 0; ss <= summa; ++ss ){
3          if( ss % lantit[ 0 ] == 0 ){ tulokset[ ss ][ 0 ] = ss / lantit[ 0 ]; }
4      }
5      for( unsigned ll = 1; ll <= suurin; ++ll ){ tulokset[ 0 ][ ll ] = 0; }
6      for( unsigned ss = 1; ss <= summa; ++ss ){
7          for( unsigned ll = 1; ll <= suurin; ++ll ){
8              unsigned tulos = tulokset[ ss ][ ll - 1 ];
9              if( ss >= lantit[ ll ] ){
10                 unsigned tapa2 = tulokset[ ss - lantit[ ll ] ][ ll ];
11                 if( tulos > tapa2 ){ tulos = tapa2 + 1; }
12             }
13             tulokset[ ss ][ ll ] = tulos;
14         }
15     }
16     return tulokset[ summa ][ suurin ];
17 }
```

Tutkimalla edellistä ohjelmaa huomataan, että yksiulotteinen taulukko riittää

```
1  unsigned vaihtoraha( unsigned summa, unsigned suurin ){
2      for( unsigned ss = 0; ss <= summa; ++ss ){
3          if( ss % lantit[ 0 ] == 0 ){ tulokset[ ss ] = ss / lantit[ 0 ]; }
4      }
5      for( unsigned ll = 1; ll <= suurin; ++ll ){
6          for( unsigned ss = 1; ss <= summa; ++ss ){
7              if( ss >= lantit[ ll ] ){
8                  unsigned tapa2 = tulokset[ ss - lantit[ ll ] ];
9                  if( tulokset[ ss ] > tapa2 ){ tulokset[ ss ] = tapa2 + 1; }
10             }
11         }
12     }
13     return tulokset[ summa ];
14 }
```

Lanttien tulostaminen voidaan lisätä lisäämällä kaksiulotteinen bittitaulukko

- `bool kayta[ max_summa_1 ][ lantteja ];`
- alustetaan arvolla `false`
- riville 9 lisätään `kayta[ ss ][ ll ] = true;`

Välitulokset muistava ratkaisu juomapaikkatehtävälle

- ei ole kilpailukykyinen ahneeseen algoritmiin ja puolitushakuun perustuvaa ratkaisua vastaan, mutta havainnollistaa välitulosten muistamista
- $\text{muistettu}[\text{paikka}, \text{taukoja}]$ tallettaa optimituloksen
 - ∞ tarkoittaa, että sitä ei ole vielä laskettu
- aluksi jokainen $\text{muistettu}[\text{paikka}, \text{taukoja}]$ on ∞
- erot edelliseen versioon ovat pienet
- olkoon n mahdollisten juomapaikkojen määrä ja k haluttu määrä
 - ajan kulutus $O(n^2k)$ (puolitushaulla saataisiin $O(nk \log n)$)
 - muistin kulutus $\Theta(nk)$

```
1  PARAS(paikka, taukoja)
2  if taukoja < 1 then return sijainnit[paikka]
3  lyhin := muistettu[paikka, taukoja - 1]
4  if lyhin  $\neq \infty$  then return lyhin
5  for i := 0 to paikka do
6      pisin := MAX( sijainnit[paikka] - sijainnit[i], PARAS(i, taukoja - 1) )
7      if pisin < lyhin then lyhin := pisin ; valittu[paikka, taukoja - 1] := i
8  muistettu[paikka, taukoja - 1] := lyhin
9  return lyhin
```


7 Pikajärjestäminen ja mediaanin etsiminen

7.1	Lippmanin versio	178
7.2	Perusversio	189
7.3	Tehostuskeinoja	198
7.4	Nopea etsiminen järjestysluvun perusteella	204

7.1 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)

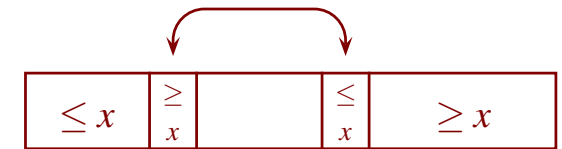
Toimintaperiaate

$\leq x$	$= x$	$\geq x$
----------	-------	----------

- 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 alkioden paikkoja siten, että jokainen alkuosan alkio on enintään yhtäsuuri kuin mikään loppuosan alkio
 - joissakin versiossa syntyy tai voi syntyä myös keskiosa, jossa on vain yhtä avainta
- 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`

$\leq x$	$= x$	$\geq x$
----------	-------	----------

```
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, ..., 199999 sekalaisessa järjestyksessä: silmänräpäys
- 0, ..., 199999 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

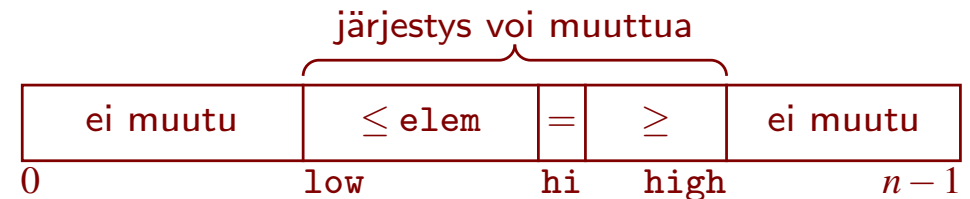
- `while ( ia[ ++lo ] < elem ) ;`
- 200 000 kertaa sama alkio: silmänräpäys
- muut viat säilyivät

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
- tarkasta rekursiokutsut samaan tyyliin kuin silmukat
 - invarianttia vastaa ehto, joka on voimassa jokaisen kutsun alussa
- 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 $ia[low \dots high]$ on samat alkiot kuin rivin 3 lopussa
2. osissa $ia[0 \dots low - 1]$ ja $ia[high + 1 \dots n - 1]$ on samat alkiot samassa järjestyksessä kuin rivin 3 lopussa, missä n on koko taulukon ia koko
3. jos $low \leq x < hi$ niin $ia[x] \leq elem$
4. $ia[hi] = elem$
5. jos $hi < x \leq high$ niin $ia[x] \geq elem$



Osat 1 ja 2

- ia :n sisältöä muutetaan vain vaihtamalla kahden alkion paikkaa swap:illa
⇒ riittää tarkastaa, että ne käsittelevät vain osaa $ia[low \dots high]$
- rivi 9: `swap( ia, lo, hi )`
 - rivit 5 ja 7: $low < lo$, rivi 9: $lo < hi$, rivit 5 ja 8: $hi \leq high$
 - ⇒ $low < lo < hi \leq high$
- rivi 12: `swap( ia, low, hi )`
 - $ia[low]$ ei muutu edellä, koska $low < lo < hi$
 - ⇒ riveillä 6, ..., 11 $ia[low] = elem$
 - ⇒ rivi 8 pysähtyy viimeistään kun $hi = low$
 - viimeistään silloin rivit 9 ja 10 vievät pois silmukasta, koska silloin $lo > low = hi$
 - ⇒ rivillä 12 $low \leq hi \leq high$

Osat 3, 4 ja 5

3. jos $\text{low} \leq x < \text{hi}$ niin $\text{ia}[x] \leq \text{elem}$

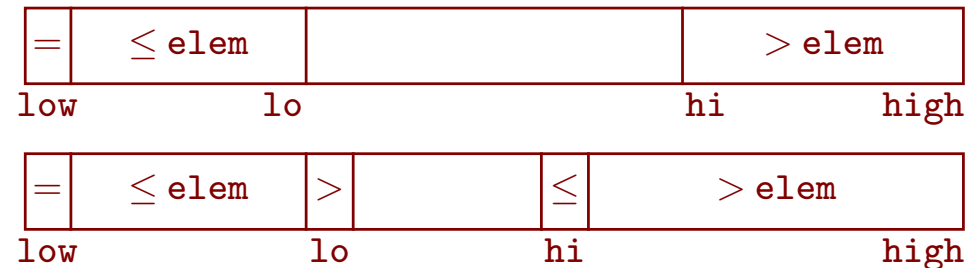
4. $\text{ia}[\text{hi}] = \text{elem}$

5. jos $\text{hi} < x \leq \text{high}$ niin $\text{ia}[x] \geq \text{elem}$

- apuväite (a) rivin 6 alussa:

- jos $\text{low} + 1 \leq x \leq \text{lo}$ niin $\text{ia}[x] \leq \text{elem}$

- jos $\text{hi} \leq x \leq \text{high}$ niin $\text{ia}[x] > \text{elem}$



- apuväite (b) rivin 9 alussa:

- $\text{ia}[\text{lo}] > \text{elem}$, ja jos $\text{low} + 1 \leq x < \text{lo}$ niin $\text{ia}[x] \leq \text{elem}$

- $\text{ia}[\text{hi}] \leq \text{elem}$, ja jos $\text{hi} < x \leq \text{high}$ niin $\text{ia}[x] > \text{elem}$

- 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 tapauksessa $\text{lo} \geq \text{hi}$, ja mennään riville 12
 - jos $\text{low} + 1 \leq x < \text{hi}$ niin $\text{ia}[x] \leq \text{elem}$, koska $\text{lo} \geq \text{hi}$
 - 5. pätee, jopa niin että $> \text{eikä} \geq$
 - edelliseltä ruudulta $\text{low} \leq \text{hi} \leq \text{high}$ ja $\text{ia}[\text{low}] = \text{elem}$
 - `swap( ia, low, hi )` asettaa $\text{ia}[\text{low}] \leq \text{elem}$ (b) ja $\text{ia}[\text{hi}] = \text{elem}$, eikä riko muuta
- ⇒ 3., 4. ja 5. pätevät

Rekursioiden pysähtyminen

- rivillä 12 $low \leq hi \leq high$

⇒ 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
- ⇒ 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 odottamattoman paljon
 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

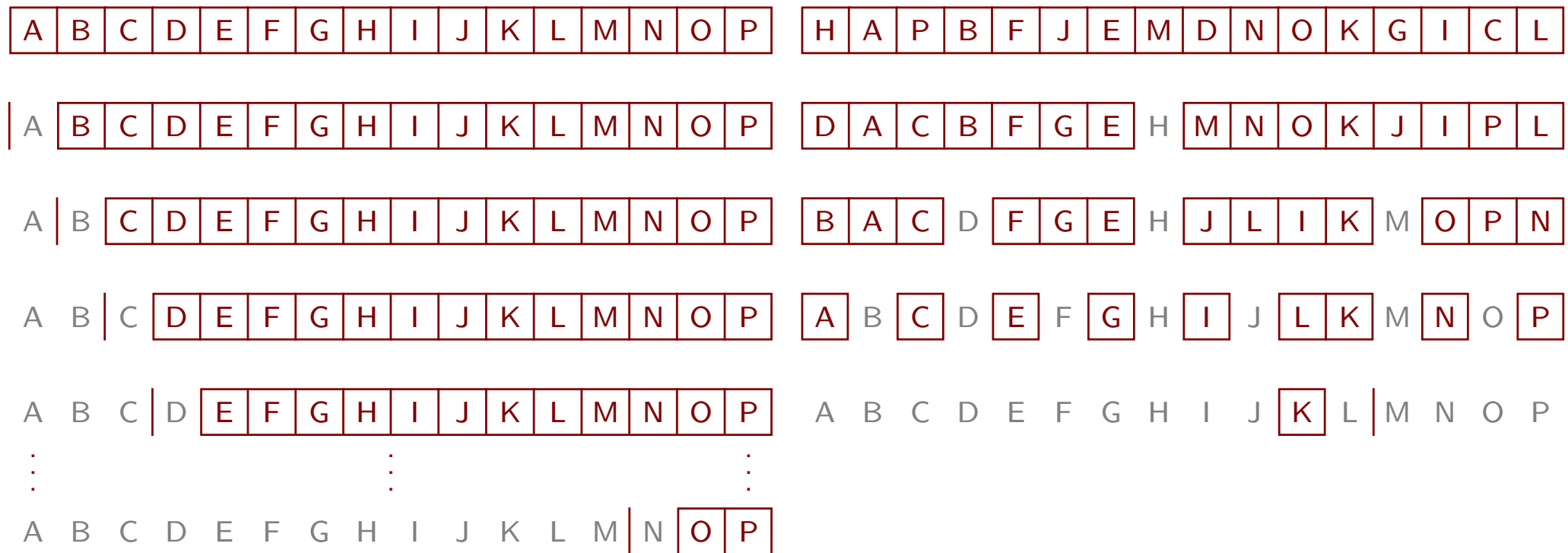
Ohi-indeksointi

- laillinen: ei osu osataulukkaan `ia[low...high]`, osuu koko taulukkaan `ia[0...n-1]`
- laiton: osuu koko taulukon ulkopuolelle

Taulukolla $[0, 1, \dots, 199999]$ kului paljon enemmän aikaa kuin epäjärjestyksessä olevalla

- 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 $ia[low \dots low - 1]$
 ja yhtä vaille alkuperäinen $ia[low + 1 \dots high]$

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



- epäjärjestyksellä osataulukot ovat usein melko samankokoiset
 $\Rightarrow$ selataan yhteensä keskimäärin alle $5n \log_2 n$ alkioita (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
⇒ paljon laillisia ja jokin määrä laittomia ohi-indeksointeja

Mitä vaikuttaa rivin 7 muutos $\leq \rightsquigarrow <?$

- 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
- ⇒ l_o ja h_i etenevät tasatahtia kohti toisiaan
- ⇒ ositus jakaa taulukon mahdollisimman tarkasti tasan
- ⇒ 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

7.2 Perusversio

Laadukas perusversio (ei mukana tehostuskeinoja)

- järjestää osataulukon $A[a \dots y]$
- pseudokoodi ottaa huomioon, että alkio voi sisältää muutakin kuin avaimen
- voi syntyä yhden alkion kokoinen keskiosa
- jakoalkion suuruiset voivat päätyä kumpaan osaan vain (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)
1  if  $a \geq y$  then return
2   $x := A[\text{RANDOM}(a, y)].x$ 
3   $i := a; j := y$ 
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  $i \geq j$  then break
8      SWAP( $A, i, j$ )
9       $i := i + 1; j := j - 1$ 
10 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"
- keskimmaisessä 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 keskimmainen 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 $A[a \dots i - 1]$ alkio on enintään x ja kukin osan $A[j + 1 \dots y]$ alkio on vähintään x .
3. Osassa $A[i \dots y]$ on ainakin yksi alkio, joka on vähintään x . Osassa $A[a \dots j]$ 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 tyhjä
- 3: rivillä 2 valittu alkio toteuttaa vaatimukset

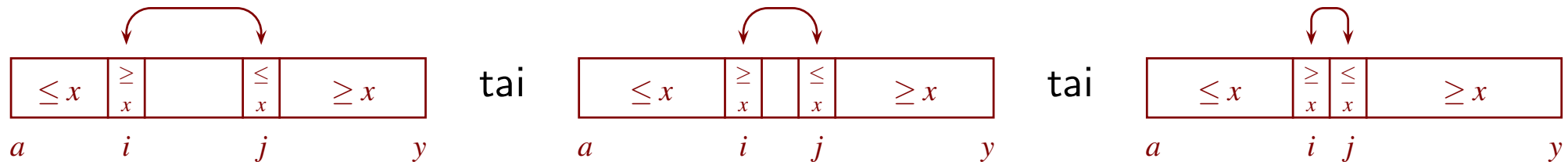
```
1  if  $a \geq y$  then return  
2   $x := A[\text{RANDOM}(a, y)].x$   
3   $i := a; j := y$ 
```

Osan 1 säilyminen voimassa

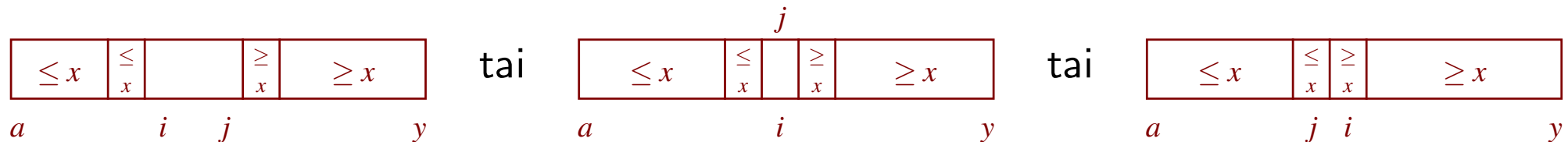
- i ei voi vähentyä eikä j kasvaa, joten $a \leq i$ ja $j \leq y$ säilyvät voimassa
 - jos riville 4 tullaan uudelleen, niin rivin 9 alussa päti $i < j$, joten
 - $i + 1 \leq j \leq y$ ja $a \leq i \leq j - 1$
 - $i + 1 \leq (j - 1) + 1$
- $\Rightarrow$ rivin 9 lopussa ja rivin 4 alussa $i \leq y$ ja $a \leq j$ ja $i \leq j + 1$
- (viimeistään osan 3 lupaamat alkiot pysäyttävät i :n kasvun ja j :n vähenemisen, mutta vielä emme tarvitse tätä tietoa)

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:



- rivin 8 SWAP asetti $A[i].x \leq x$ ja $A[j].x \geq x$
 $\Rightarrow$ 2 ja 3 säilyivät voimassa rivillä 9

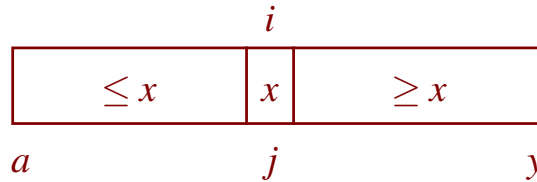


```

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  $i \geq j$  then break
8      SWAP( $A, i, j$ )
9       $i := i + 1; j := j - 1$ 
    
```


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

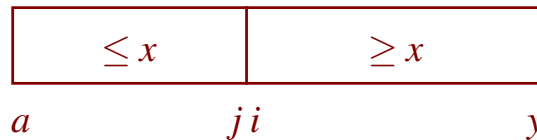
- $A[i].x \geq x$ ja $A[i].x \leq x$, joten $A[i].x = x$



- osat eivät mene lomittain, ja alkiot ovat oikeassa järjestyksessä
- jokainen osa on alkuperäistä pienempi
 - keskimmaisessä on vain yksi alkio, vaikka sisään tuli ainakin kaksi (rivi 1)
 - muista osista puuttuu ainakin keskimäinen 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 $A[a \dots i - 1]$ puuttuu ainakin $A[i]$
 - osasta $A[j + 1 \dots y]$ 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: $\text{RANDOM}(a,y)$ tuottaa satunnaisluvun väliltä $a, \dots, y$
- 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 $a = i = 0$

⇒ rivi 1 pois, rivin 10 ja alkuperäisen kutsun sijaan

```
if  $i > a + 1$  then QUICKSORT( $A, a, i - 1$ )  
if  $j < y - 1$  then QUICKSORT( $A, j + 1, y$ )  
if  $a < y$  then 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 alkioden avaimet ovat järjestettävässä taulukossa eivätkä osoittimien takana, niin pikajärjestäminen hyötyy tästä
 - joissakin uusissa oliokielissä taulukossa eivät ole alkiot, vaan osoittimet niihin
- ⇒ 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
- ⇒ lisämuistin tarve on verrannollinen rekursiotasojen määrään
- ⇒ 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
```

7.3 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
- ⇒ INSERTIONSORT:in ajan käyttö on $O(nk)$

Suurempi osataulukko kannattaa käsitellä silmukalla eikä rekursiolla

⇒ rekursiopinin syvyys ja muistin tarve huonoimmillaan $\Theta(\log n)$

```
1  QUICKSORT(&A, a, y)
   while  $y - a \geq 50$  do
       ... edeltä rivit 2, ..., 9
10  if  $i - a \leq y - j$  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       $y := i - 1$ 
```

```
1  QUICKSORT(&A)
   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
⇒ 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

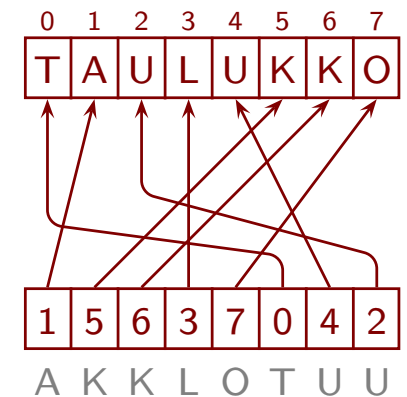
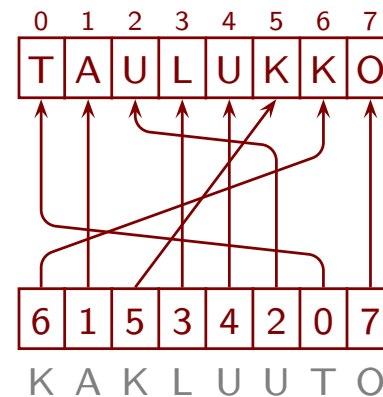
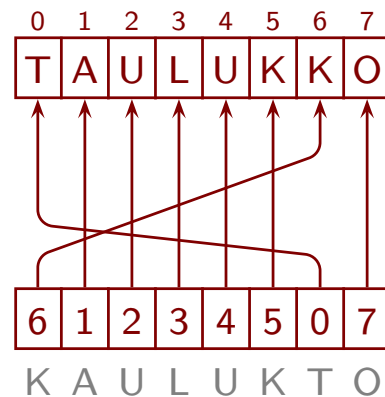
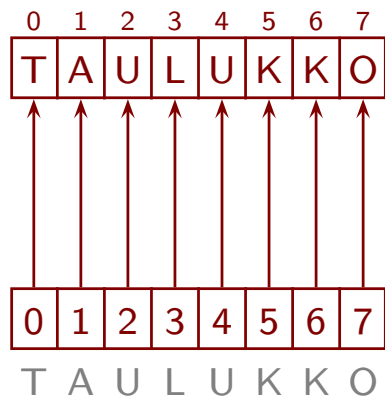
0,6 s	5,6 s	8,7 s	1 min 25 s	1,8 s	36 s
0,6 s	5,5 s	8,9 s	1 min 23 s	1,8 s	35 s
 - silmukka-QUICKSORT ei eronnut mainittavasti nopeudessa rekursiivisesta
⇒ satunnaisella aineistolla rekursiotasojen määrä pysynee melkein aina pienenä
 - pienillä alkioilla kannatti järjestää pienet osataulukot lisäysjärjestämisellä, isoilla ei
 - isoilla alkioilla kannattaa minimoida alkioden kopiointien määrä
- ⇒ muutos, joka nopeuttaa yhdenlaisilla taulukoilla, saattaa hidastaa toisenlaisilla
⇒ on vaikeaa optimoida äärimmilleen
 - mutta ei se tyypillisesti ole tarpeenkaan
- MERGESORT tulee luvussa 9.1
 - RADIXSORT ja COUNTINGSORT olettavat avaimista paljon
 - se on otettava huomioon, jotta niiden vertaaminen muihin olisi reilua
 - luvut 9.5 ja 9.4

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

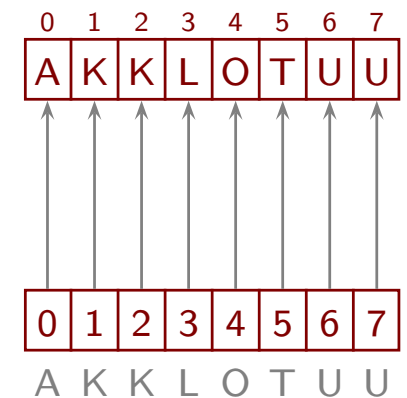
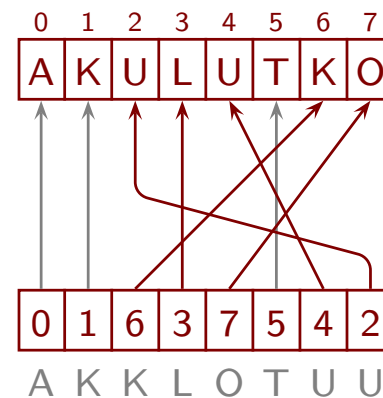
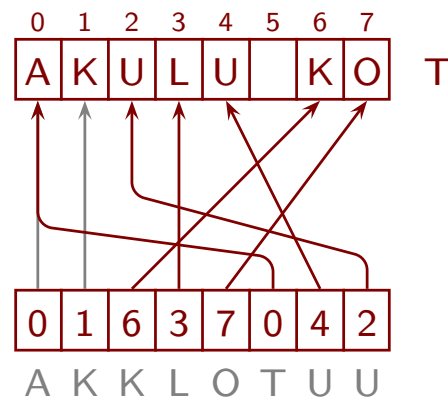
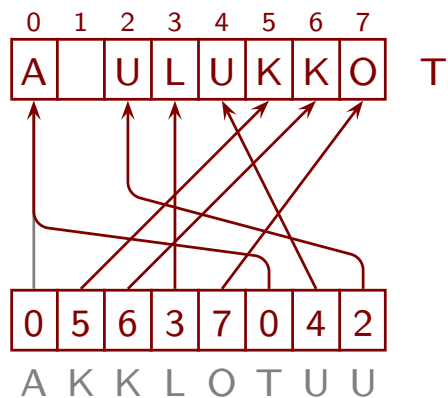
- C++ `stable_sort` perustuu melko varmasti kahteen lomitussjärjestämisen (merge sort) muunnokseen
 - ajan kulutus $\Theta(n \log n)$ jos muistia on paljon, $\Theta(n(\log n)^2)$ jos niukasti
- C++ `sort` perustuu melko varmasti pikajärjestämiseen
 - Wikipedia "Introsort" ja g++ -E
 - enintään 16 alkion 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 $0, 1, \dots, n-1$
- aputaulukko järjestetään käyttäen vertailuina $A[N[i]].x < A[N[j]].x$ jne.



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



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

7.4 Nopea etsiminen järjestysluvun perusteella

QUICKSELECT(&A, k)

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



⇒ 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: luonnostelee algoritmi, joka hyödyntää aikaisempaa järjestämistyötään

QUICKSELECT(A, k)

- muuten kuten edellinen, mutta A välitetään kopioimalla eikä viitteellä

⇒ ei muuta kutsujan taulukon järjestystä

- lisämuistin tarve $\Theta(n)$

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

8 Hieman linkitetyistä listoista

Yksisuuntainen linkitetty lista

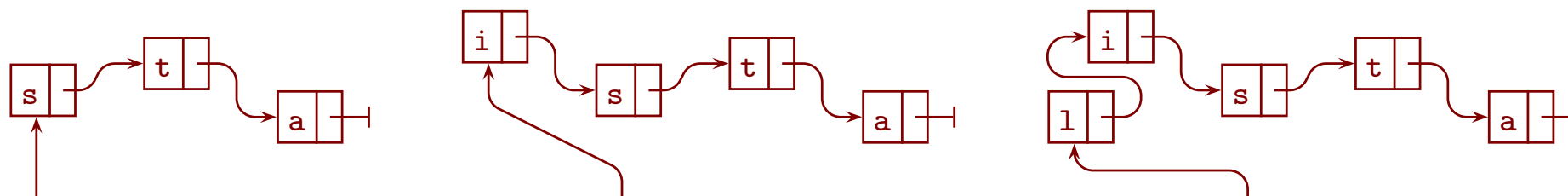
- tietorakenne, jossa jokainen alkio sisältää tiedon, missä seuraava alkio sijaitsee
- linkki voi olla osoitin, ei-negatiivinen kokonaisluku, iteraattori, ...

Kaksisuuntainen: myös tieto, missä edellinen alkio sijaitsee

Perusmuotoisen linkitetyn listan alkuun voi lisätä ja poistaa alkion tehokkaasti

⇒ helppo toteuttaa tehokkaasti *pino*

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



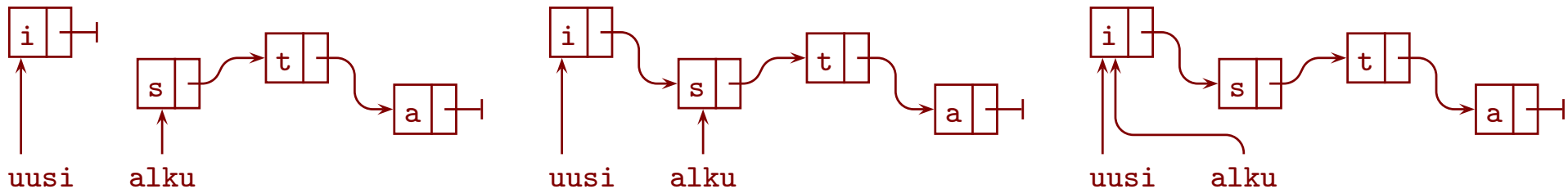
Pino listalla C++:lla

- 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;
```

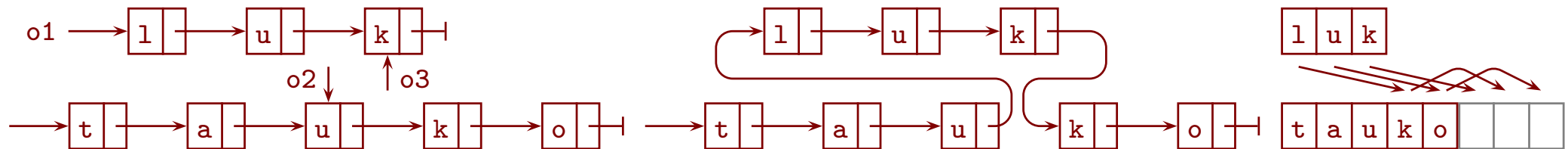


- ensimmäisen tulostaminen, poistaminen ja muistin palautus käyttöjärjestelmälle

```
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

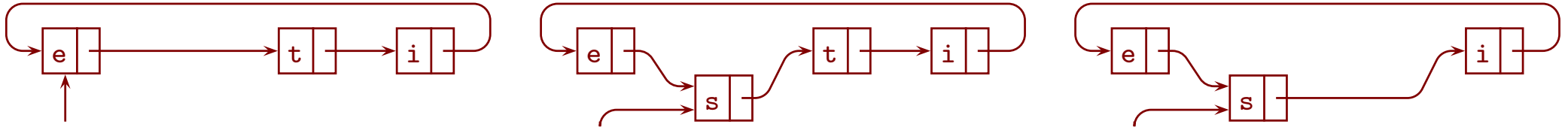
- lisääminen, jos on osoitin lisäyskohtaa edeltävään alkioon (ja lisättävien loppuun)
- taulukkoon ei voi lisätä tehokkaasti keskelle



- 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 tasatassa ajassa $\Theta(1)$

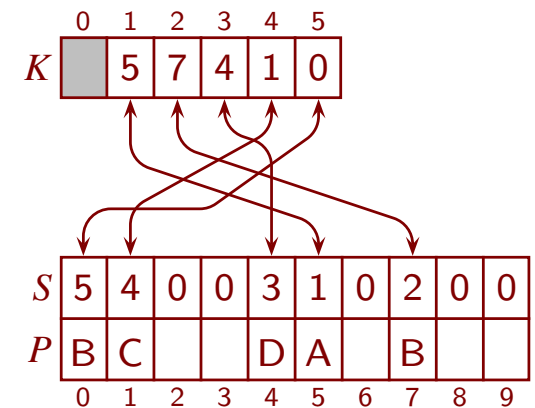
Linkitetyn listan alkioihin pääsee käsiksi vain selaamalla tai tallennetulla osoittimella tms.

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

⇒ linkitetyille listoille on niukasti käyttöä itsenäisenä tietorakenteena

Linkeille on enemmän käyttöä isomman rakenteen osana

- binääripuut luku 11
 - 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 alkioden sijaintia muistissa tarvitse muuttaa
 - ⇒ luvussa 3.7 prioriteettijonossa olevan alkion prioriteetin muuttaminen oli monimutkaista



Linkitetyn listan lopun ilmoittaminen

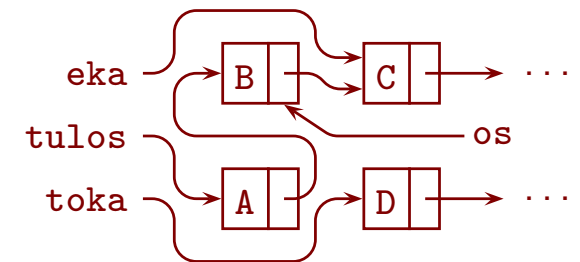
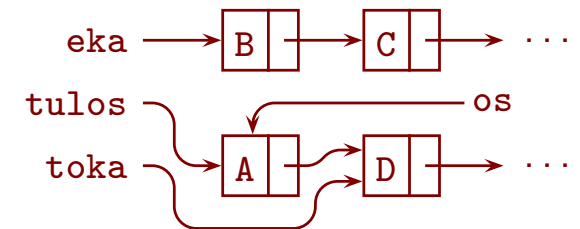
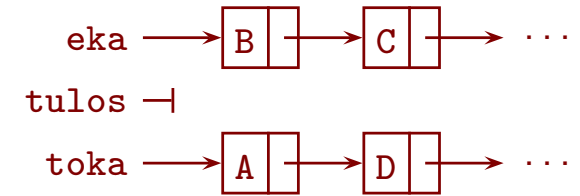
- osoitin ei minnekään
- luku 0 tai -1 tai $\sim 0_u$
- linkki alkioon itseensä

Tyhjiä rakenteita ja listojen päitä joudutaan usein käsittelemään erikoistapauksina

⇒ 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 }
```



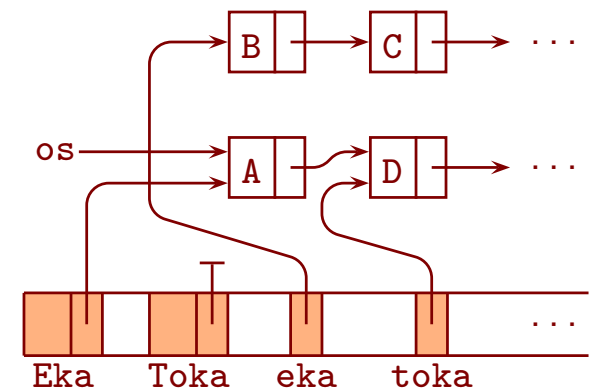
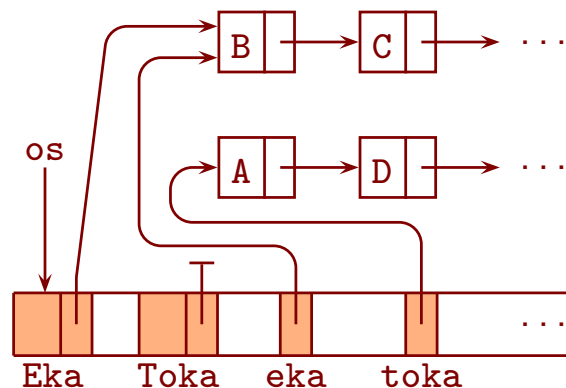
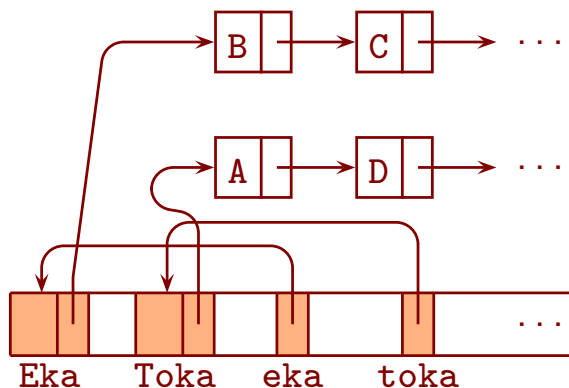
- tyhjät syötelistat käsitellään erikoistapauksina riveillä 2 ja 3
- kaikkein pienin alkio siirretään tuloslistan ensimmäiseksi riveillä 4, ..., 6
- riveillä 7, ..., 12 toistuvasti siirretään pienempi syötelistojen seuraavana vuorossa olevista alkioista tuloslistaan kunnes jompikumpi syötelista loppuu
- 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

9 Lisää järjestämisalgoritmeista

9.1	Lomitusjärjestäminen	215
9.2	Satunnaisen järjestyksen tuottaminen	222
9.3	Vertaamiseen perustuvan järjestämisen ajan alaraja	225
9.4	Laskentajärjestäminen	229
9.5	Kantalukujärjestäminen	231

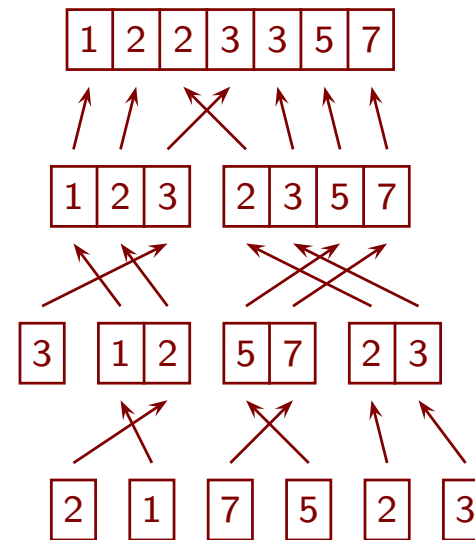
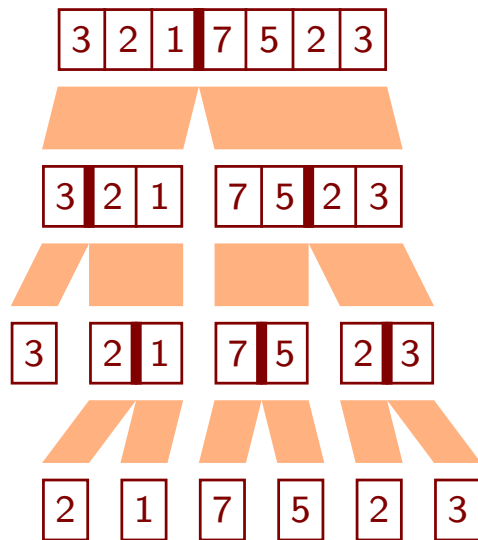
9.1 Lomitusjärjestäminen

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 suoritusajkoja

- 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 ]; }
```

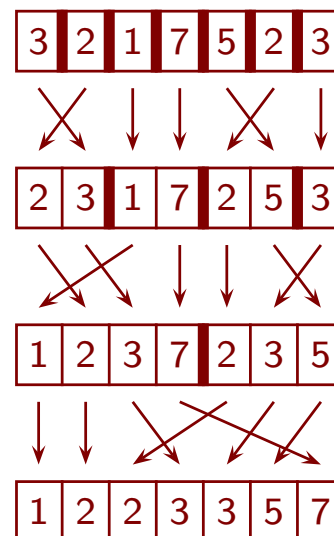
⇒ ei kannata toistuvasti varata ja vapauttaa muistia, jos voi käyttää kerran varattua

- 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 lomitussjärjestäminen

- 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)
- ⇒ ajat eivät ole vertailukelpoisia muihin

9.2 Satunnaisen järjestyksen tuottaminen

Tarvitaan esimerkiksi lottorivin tuottamiseksi

Toimintaperiaate

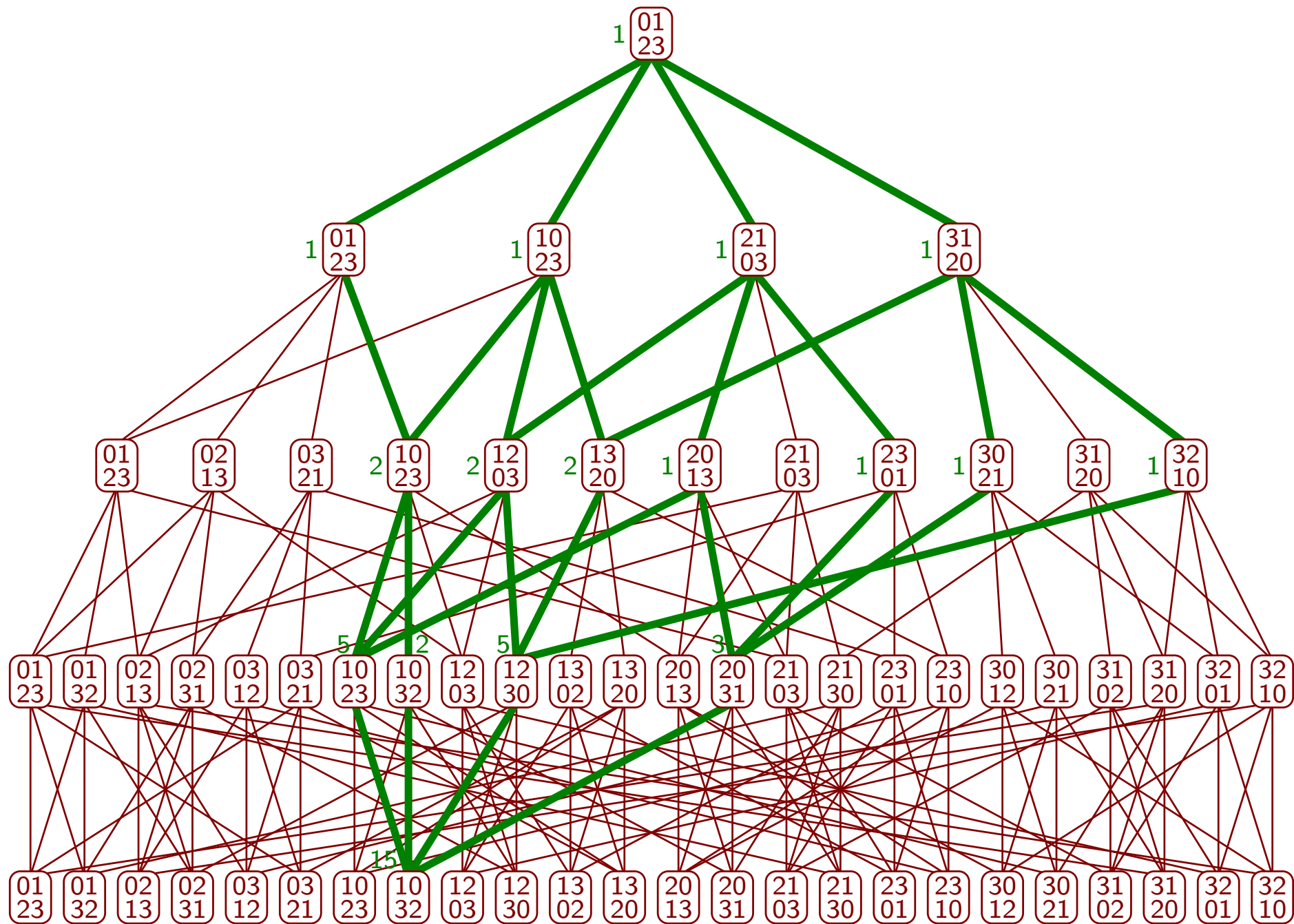
- arvotaan koko taulukosta alkio, ja vaihdetaan se ensimmäiseksi
- arvotaan muista kuin ensimmäisestä alkio, ja vaihdetaan se toiseksi
- arvotaan muista kuin kahdesta ensimmäisestä alkio, ja vaihdetaan se kolmanneksi
- ...
- alla $\text{RANDOM}(i, n-1)$ tuottaa tasan jakautuneen satunnaisluvun väliltä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

```
1  for  $i := 0$  to  $n-1$  do  $A[i] := i$   
2  for  $i := 0$  to  $n-2$  do  
3       $\text{SWAP}(A, i, \text{RANDOM}(i, n-1))$ 
```



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:ssä
 - 3 0 1 2 tulee 8:ssä



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

- $\text{RANDOM}(\dots) \bmod m$ tuottaa tasajakauman vain jos $\text{RANDOM}(\dots)$:in tuottamien eri satunnaislukujen määrä on jaollinen m :llä
 - tyypillinen satunnaislukugeneraattori tuottaa 2^{31} tai 2^{32} eri arvoa
 - esim. jos RANDOM tuottaa luvut $0, \dots, 2^{31} - 1$, niin $\text{RANDOM} \bmod 1000$ tuottaa
 - 2147484 kertaa arvot $0, \dots, 647$
 - 2147483 kertaa arvot $648, \dots, 999$
- ⇒ jos jakauman pitää olla tarkasti tasainen, niin kun RANDOM tuottaa lähellä maksimia olevan luvun, niin se pitää hylätä ja arpoa uusi
- esim. 2147483000 tai yli
- satunnaisen järjestyksen tuottamisessa m saa arvot $n, n - 1, \dots, 2$
⇒ on usein muu kuin kahden potenssi

9.3 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 $n \cdot (n-1) \cdot \dots \cdot 1 = n!$ mahdollisuutta
 - jo paikan saaneet eivät ole mukana myöhemmissä valinnoissa
 - n **kertoma**

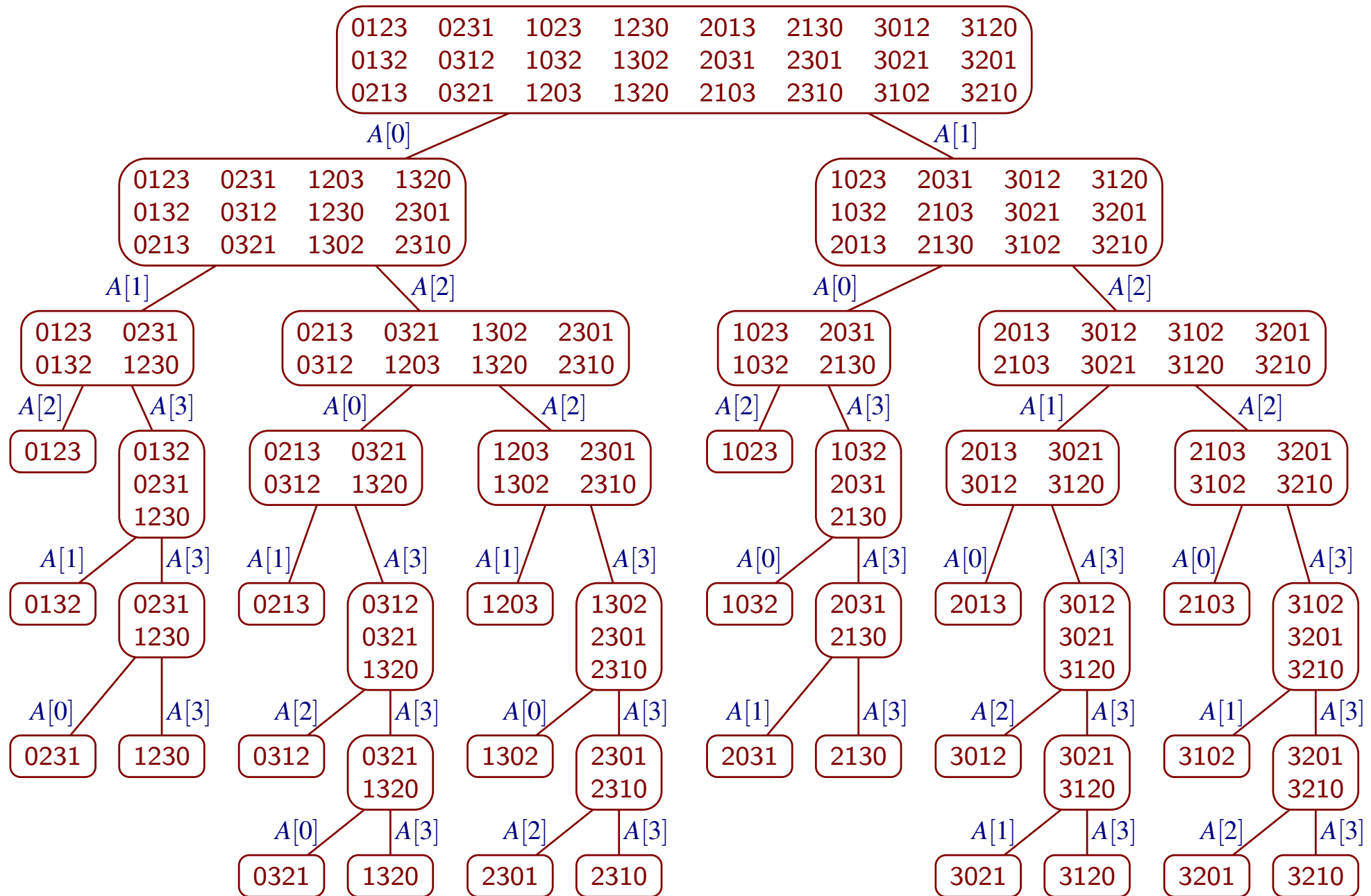
Jokainen vertailu jakaa jäljellä olevat mahdolliset järjestykset kahtia

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

⇒ 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



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

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

$\Rightarrow n! \leq 2^k \Rightarrow k \geq \log_2 n! = \log_2 1 + \dots + \log_2 n > \frac{1}{2}n \log_2 n - \frac{1}{2}n$, 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 päättyy 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 rekursiotasojille
- ⇒ jos avaimissa on enintään viittä eri arvoa, niin rekursiotasoja on enintään viisi
- ⇒ 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 $O(n \log k)$

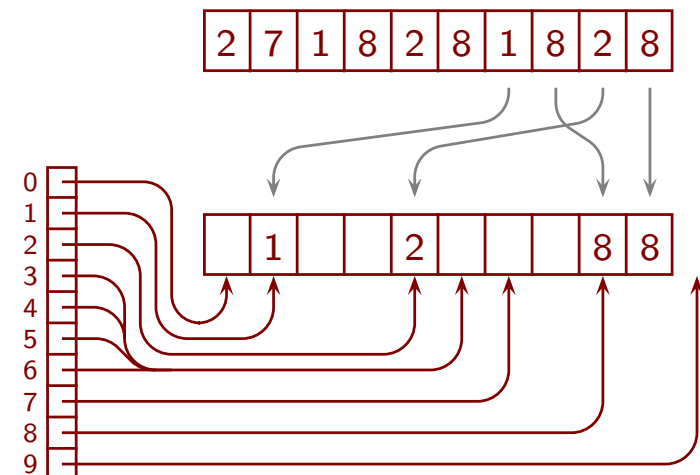
Seuraavaksi tarkastellaan kahta järjestämisalgoritmia, jotka eivät perustu vertaamiseen

9.4 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 indeksiansä pienempien ja kopioimattomien indeksinsä suuruisten alkioden 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 $0, \dots, k-1$, 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
 - suoritus aika 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 alkiolla

Ruudussa 199 oli $k = n$

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

9.5 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 199 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

10 Hajautustaulut

10.1	Ketjutetut hajautustaulut	233
10.2	Sovellusesimerkki: reitin etsinnän syötteen uudistaminen	241
10.3	Avoin osoitus	248
10.4	Bloom-suodatin	249

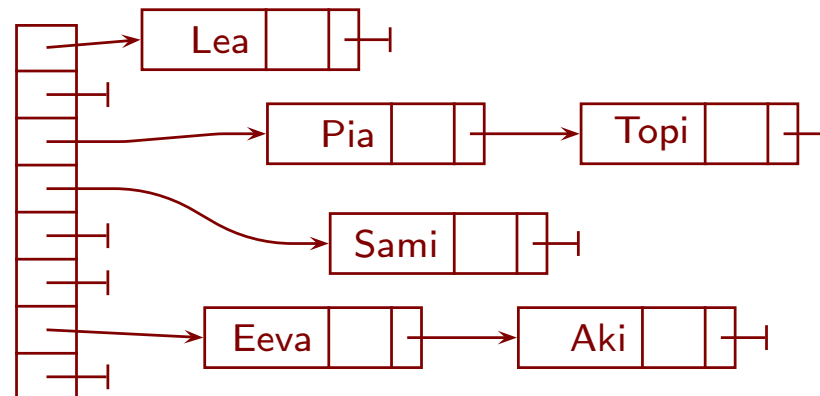
10.1 Ketjutetut hajautustaulut

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 alkoiden 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
⇒ jos alkioita on paljon, niin listojakin on paljon
- listojen alut ovat *kantataulukossa* (*bucket array*), joka on taulukko osoittimia
- listojen sisäisellä järjestyksellä ei ole väliä



Kantataulukon koon valitseminen

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

Hajautusfunktio (hash function)

- ottaa avaimen ja tuottaa luvun väliltä $0, \dots, M - 1$, missä M on kantataulukon 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>
 - 104743 sanaa kuten "algoritmi" tai sanaliittoa kuten "all stars -joukkue"
 - myös eksoottisia merkkejä kuten "à la carte -annos"
- $M = 2^{16} = 65536$
- 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 alkioden 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 4208, lisätyö 71508730
 - todella huono
 - pisimmässä 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 954889
 - parani paljon, mutta yhä huono
 - pisimmässä aamiainen, aavistella, aavistuksenomainen, ahmia, ..., öljynetsintä
 - 55181 tyhjää ja 10355 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 84113, tyhjiä listoja 13280
 - parani huomattavasti
- ii vaikuttaa kertoimeen, jotta eri kohdissa olevat merkit vaikuttaisivat eri tavalla
 - 1001 eikä $(1000*ii+1)$ tuotti kehnon tuloksen 107, 393320 ja 33727
- kerroin on pariton, jotta alin bitti ei olisi aina 0
 - $1000*ii$ eikä $(1000*ii+1)$ tuotti 33, 681727 ja 57344
- vakio 1000 valittiin kokeilemalla muutamia eri arvoja ilman syvällistä viisautta

vakio	1	10	100	10000	1024	1009
pisin	101	22	10	9	45	10
lisätyö	2751947	331039	85418	84492	1280371	84338
tyhjiä	61682	43915	13548	13459	59393	13347
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 84176, tyhjiä listoja 13354
 - ei parempi eikä huonompi kuin parhaat aikaisemmat

Netistä löytyisi paljon lisää hajautusfunktioita

Miksi tulosten paraneminen loppui?

- tasajakaumalla olisi maksimipituus 2, lisätyö 39207 ja tyhjiä listoja 0
- viimeisillä alkioilla on suuri todennäköisyys osua "väärin" 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 $n \rightarrow \infty$, niin eripituisten listojen osuudet $\rightarrow e^{-\ell} \frac{\ell^k}{k!}$

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 $\ell = 1,598\dots$, joten ennusteeksi tulee
 - pisin 9 tai 10
 - lisätyö 83 702
 - tyhjiä listoja 13 254,6

- satunnaisluvulla 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 alkia kohti $\approx \frac{\ell}{2}$ ja kaikkiaan $\frac{n(n-1)}{2M} \approx \frac{n^2}{2M}$

10.2 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 etaisuus; solmutyyppi *reitti, *haj_seur;
11     solmutyyppi( const std::string & nimi, solmutyyppi *haj_seur ):
12         nimi( nimi ), etaisuus( 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 alkioden määrää kahdentamista varten
- haj_maski poimii hajautustaulun kokoa vastaavan määrän bittejä
 - kahden potenssi miinus yksi
- käytetään edellä 65 536:lla hyväksi havaittua hajautusfunktioita
 - 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
 - ⇒ rivi 24 tuottaa oikean maskin ja rivit 25 ja 26 käyttävät oikeita arvoja
- maski muutetaan alussa, jotta rivillä 30 käytettäisiin uuden taulun hajautusfunktioita
- 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 alkioden 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.

10.3 Avoin osoitus

(*open addressing*)

Ei linkitettyjä listoja, vaan alkiot ovat suoraan kantataulukossa

- 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

⇒ avointa osoitusta ei kannata käyttää

10.4 Bloom-suodatin

Erittäin vähän muistia käyttävä tietorakenne joukon likimääräiseksi esittämiseksi

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

Esimerkiksi sanojen talletus oikolukuohjelmaa varten

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

Rakenne

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

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

- bitti jää nolllaksi 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!

11 Binääripuut

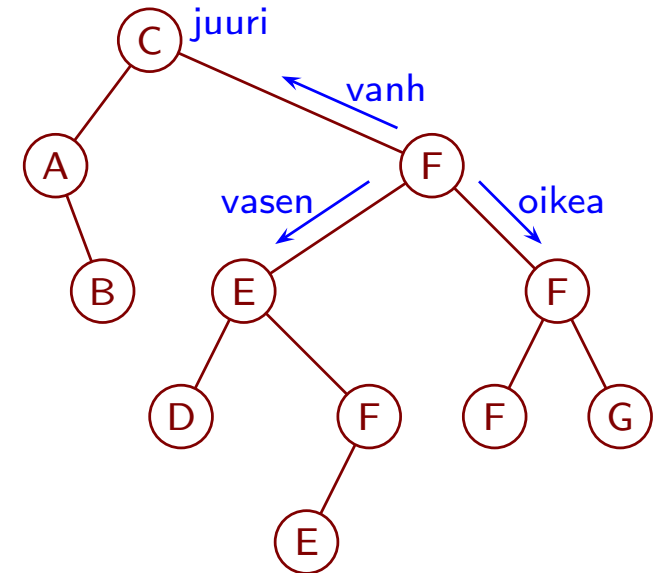
11.1	Yleistä binääripuista	251
11.2	Binäärihakupuut	256
11.3	Punamustat puut	265
11.4	Kuinka mones ja valitse näin mones	270

11.1 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     };
18     ... monta sataa riviä toimintoja ja hieman staattista dataa
19 }
```

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  }

```

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

- op_iprod on näkymätön kertolasku, kuten $2x$

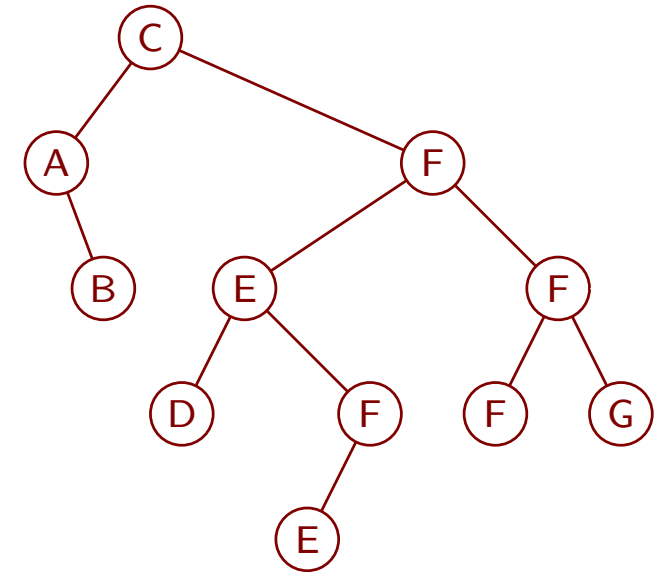
Korkeus (*height*)

- 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

```
1  if( s1->oikea ){
2      s1 = s1->oikea;
3      while( s1->vasen ){ s1 = s1->vasen; }
4  }else{
5      while( s1->vanh && s1->vanh->oikea == s1 ){ s1 = s1->vanh; }
6      s1 = s1->vanh;
7  }
```

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



11.2 Binäärihakupuut

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

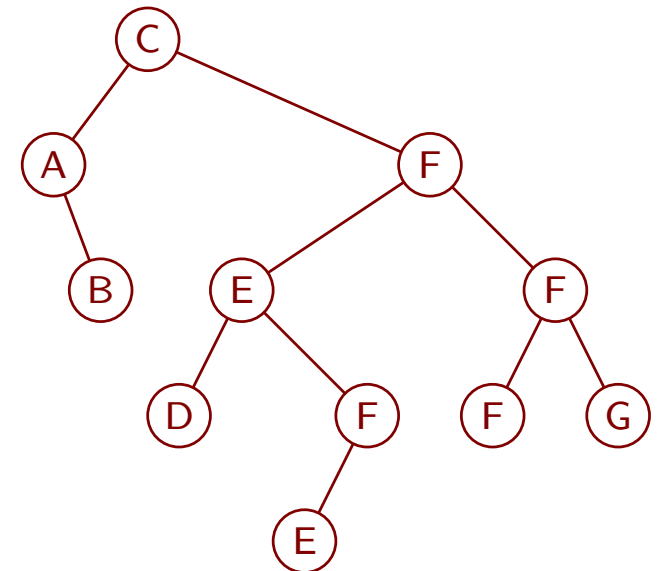
Etsintä avaimen perusteella, mikä tahansa osuma kelpaa, 0 = ei löytynyt

- 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

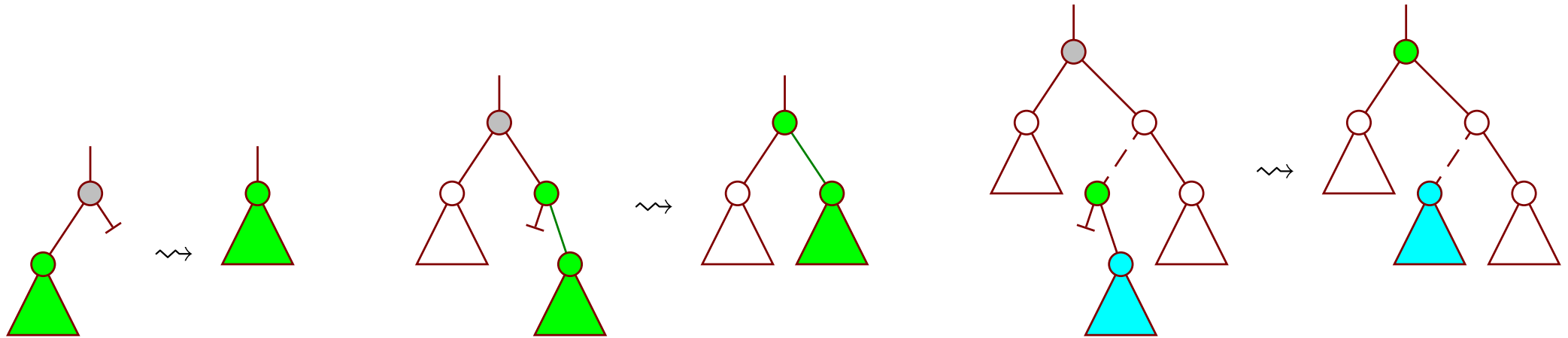
- `uusi` on solmutietue, jolle `uusi->vasen = uusi->oikea = 0`

```
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

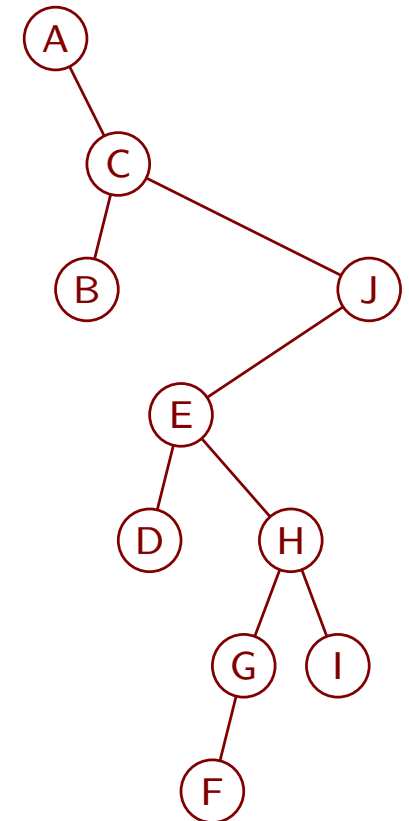


Jos solmuja lisätään suunnilleen suuruusjärjestyksessä, niin pienelläkin määrällä solmuja voi tulla kovin korkea puu

⇒ 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 11.3 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. C++
 - rakenne[avain] tarvittaessa lisää solmun
 - seuraava 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 helposti järjestää vakaasti ajassa $O(n \log n)$

```

1  std::multiset< tietue > alkiot;
2  for( unsigned ii = 0; ii < A.size(); ++ii ){ alkiot.insert( A[ ii ] ); }
3  unsigned ii = 0;
4  for( const auto & alkio : alkiot ){ A[ ii ] = alkio; ++ii; }

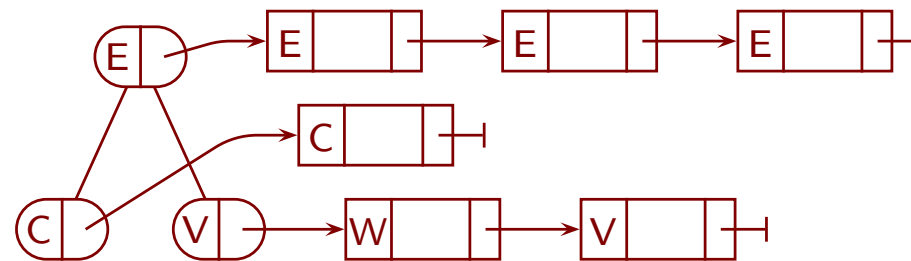
```

- multiset on muuten kuten joukko, mutta yhtäsuuria alkioita voi olla monta
– uusin tulokas lisätään yhtäsuurista viimeiseksi
- monimutkaisuutensa vuoksi hidas pienillä alkioilla
- aika nopea isoilla alkioilla, koska alkioita kopioidaan vähän
- riveillä " $h = 100$ " alkiot ovat väliltä $0, \dots, 99$

	0...4 999		10 000...19 999		10 ⁶ ...1 000 019	
monijoukko *	1,1 s	3,3 s	16 s	58 s	14 s	29 s
monijoukko $h = 100$ *	1,0 s	3,4 s	16 s	58 s	8,7 s	15 s
listoja puussa *	2,2 s	4,3 s	32 s	1 min 14 s	19 s	32 s
listoja puussa $h = 100$ *	1,0 s	3,0 s	11 s	46 s	4,3 s	9,5 s
stable_sort *	0,5 s	5,2 s	6,8 s	1 min 25 s	1,3 s	26 s
stable_sort $h = 100$ *	0,4 s	5,2 s	4,9 s	1 min 25 s	0,7 s	27 s
aputaul. stable_sort *	0,6 s	2,3 s	8,6 s	33 s	2,2 s	14 s
aputaul. stable_sort $h = 100$ *	0,5 s	2,3 s	6,5 s	31 s	1,5 s	13 s

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 yhtäsuuri 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 $O(n)$
- avaimet ovat myös listojen alkioissa mm. koska eri avaimet voivat olla yhtäsuuret
 - esim. Suomessa toisinaan samaistetaan V ja W
- monimutkaisuuden vuoksi lähtökohtaisesti hidas
- kilpailukykyinen pienellä h



Toteutus C++:lla

```
1  #include <map>
2  struct aputietue{
3      tietue data; aputietue *seur;
4      aputietue( const tietue & data, aputietue *seur ):
5          data(data), seur(seur) {}
6  };
7  void jarjesta_listoilla_puussa( taulukko & A ){
8      std::map< int, aputietue* > avaimet;
9      for( unsigned ii = 0; ii < A.size(); ++ii ){
10         aputietue *os = avaimet[ A[ii].x ];
11         avaimet[ A[ii].x ] = new aputietue( A[ii], os );
12     }
13     unsigned ii = A.size();
14     for( auto it = avaimet.rbegin(); it != avaimet.rend(); ++it ){
15         for( aputietue *os = it->second; os; ){
16             aputietue *vanha = os; os = os->seur;
17             A[ --ii ] = vanha->data; delete vanha;
18         }
19     }
20 }
```


11.3 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

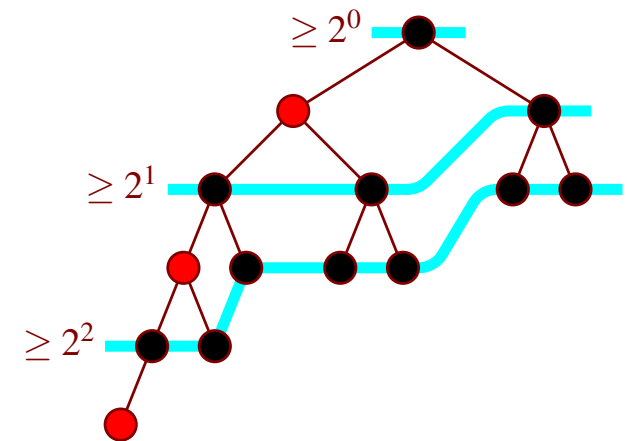
- punaisella solmulla on nolla tai kaksi lasta
- jos solmulla on täsmälleen yksi lapsi, niin se on punainen ja solmu itse on musta

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 $2^{\lceil \frac{k+1}{2} \rceil} - 1 \geq 2^{\frac{k}{2}}$

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

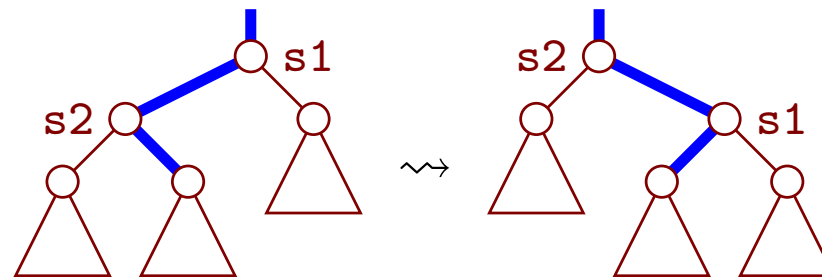


Perustoiminnot

- 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; }
```

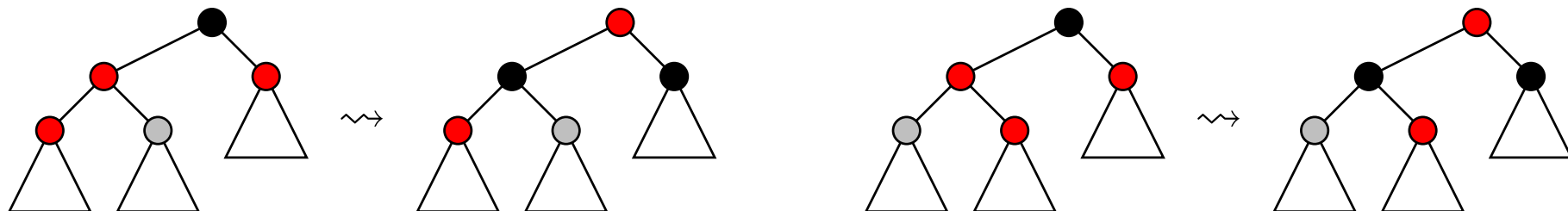
- kierto vasemmalle on symmetrinen

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

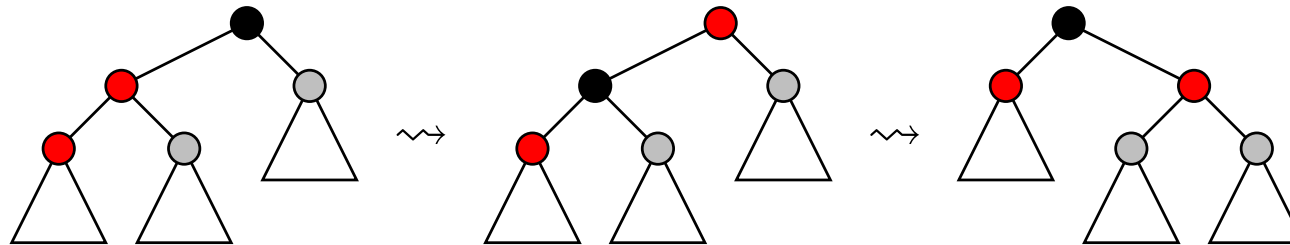
Tapaus 1: solmun vanhemman sisarus on punainen

- värittämällä kolme solmua ongelma siirtyy kaksi askelta ylöspäin tai katoaa



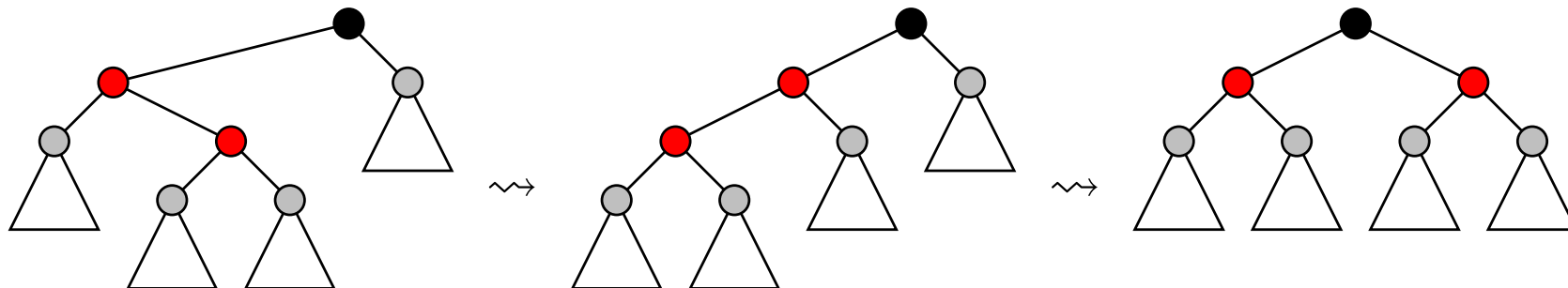
Tapaus 2: solmun vanhemman sisarus puuttuu tai on musta, ja solmu on vanhempansa vasen lapsi

- 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 11.2
- jos poistettu solmu oli musta, niin värejä korjataan maalaamalla ja kierroilla, kunnes ongelma katoaa tai saavuttaa juuren
- kiertoja tarvitaan enintään kolme

11.4 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

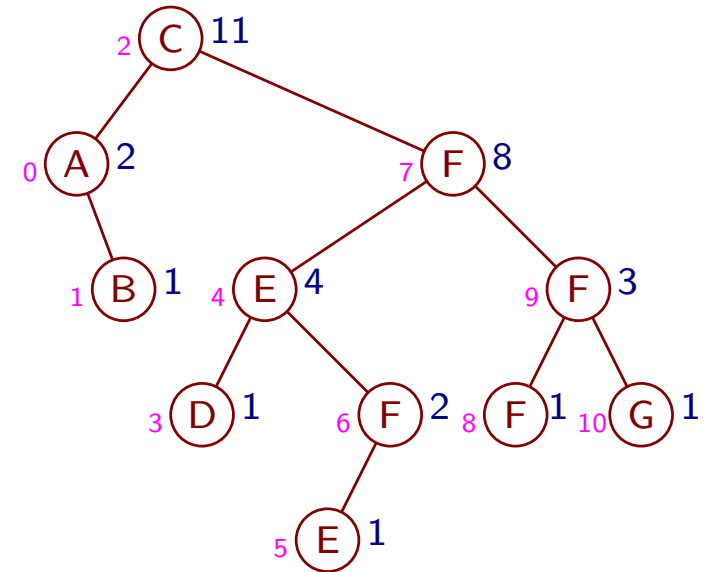
Solmun löytäminen järjestysluvun perusteella ajassa $O(k)$

```
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 $O(k)$

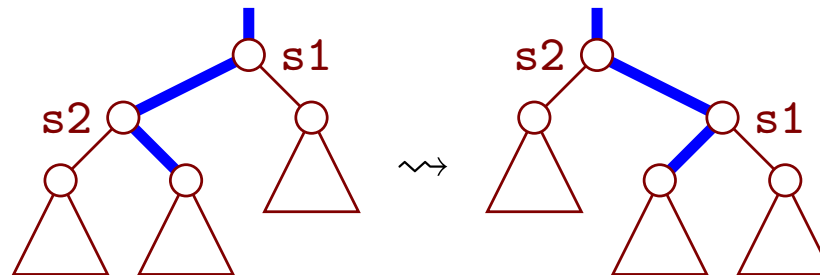
```
1  unsigned kuinka_mones( solmu *s1 ){
2      unsigned mm = 0;
3      if( s1->vasen ){ mm += s1->vasen->koko; }
4      while( s1->vanh ){
5          if( s1 == s1->vanh->oikea ){ mm += 1 + s1->vanh->vasen->koko; }
6          s1 == s1->vanh;
7      }
8      return mm;
9  }
```



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 }
```



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 ≥ A[j].x then j := j + 1
4      if j ≥ h || A[j].x ≤ 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 }
```

```

1  TULOSTAReitti(i, maali)
2  tulosta i
3  while  $i \neq \textit{maali}$  do
4       $j := R[i, \textit{maali}]$ 
5      tulosta " "  $M[i, j]$  "km " j
6       $i := j$ 

```

```

1  JONOON(i)
2  if  $S[i].m = \text{false}$  then
3       $J[v] := i; v := (v + 1) \bmod (n + 1); S[i].m := \text{true}$ 

```

```

1  JONOSTA
2   $i := J[w]; w := (w + 1) \bmod (n + 1); S[i].m := \text{false};$  return i

```

```

1  for  $i := 0$  to  $n - 1$  do
2       $S[i].e := \text{ääretön}; J[i] := i; S[i].m := \text{true}$ 
3   $S[\text{lähtö}].e := 0; w := 0; v := n; \text{kierros} := 1$ 
4  while  $\text{kierros} \leq n \ \&\& \ w \neq v$  do
5       $\text{kierros} := \text{kierros} + 1; u := v$ 
6      while  $w \neq u$  do
7           $h := \text{JONOSTA}$ 
8          if  $h = 0$  then  $j := 0$  else  $j := S[h - 1].\ell$ 
9          while  $j < S[h].\ell$  do
10              $d := S[h].e + K[j].p; k := K[j].k; j := j + 1$ 
11             if  $d < S[k].e$  then
12                  $S[k].e := d; S[k].r := h; \text{JONOON}(k)$ 

```

```

1  QS3(&A, a, y)
2  if  $a \geq y$  then return
3   $x := A[\text{RANDOM}(a, y)].x$ 
4   $j := a; k := a$ 
5  for  $i := a$  to  $y$  do
6      if  $A[i].x \leq x$  then
7           $apu := A[i]; A[i] := A[j]$ 
8          if  $apu.x = x$  then  $A[j] := apu$ 
9          else  $A[j] := A[k]; A[k] := apu; k := k + 1$ 
10          $j := j + 1$ 
11   $\text{QS3}(A, a, k - 1); \text{QS3}(A, j, y)$ 

1  for  $i := 0$  to  $A.koko - 2$  do
2      if  $N[i] \neq i$  then
3           $apu := A[i]; j := i; k := N[j]$ 
4          while  $k \neq i$  do
5               $A[j] := A[k]; j := k; k := N[j]; N[j] := j$ 
6               $A[j] := apu$ 

```

QUICKSELECT(&A,k)

```
1  if  $k < 0 \parallel k \geq A.koko$  return virhe
2  while true do
    ... kuvan 7.2 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 }

```