8.2 Suunnitteluperiaate: Muunna ja hallitse

Muunna ja hallitse on suunnitteluperiaate, joka

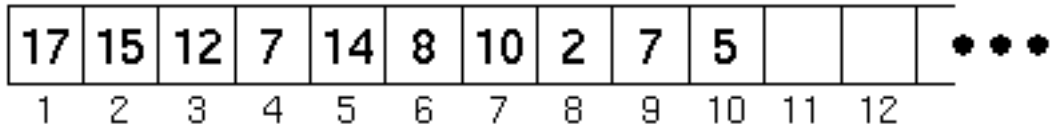
- Ensin muokkaa ongelman instanssia muotoon, joka on helpompi ratkaista – muunnosvaihe
- Sitten ongelma voidaan ratkaista – hallintavaihe

Ongelman instanssi voidaan muuntaa kolmella eri tavalla:

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

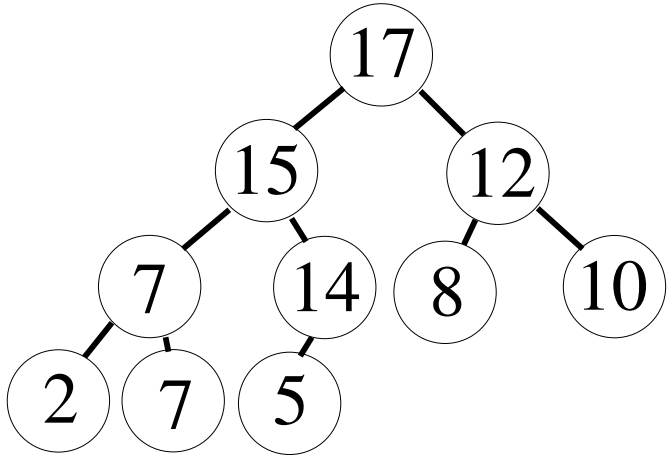
Keko

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



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

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



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

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

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

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

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

LEFT(i)
    return $2i$

RIGHT(i)
    return $2i + 1$

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

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

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

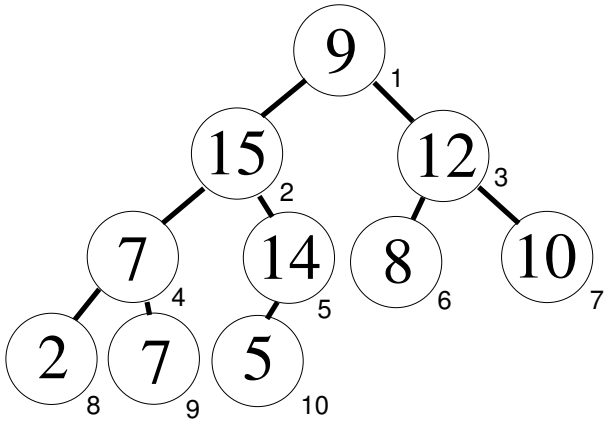
Keko-ominaisuudesta seuraa, että keon suurin alkio on aina keon juuressa, siis taulukon ensimmäisessa lokerossa.

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

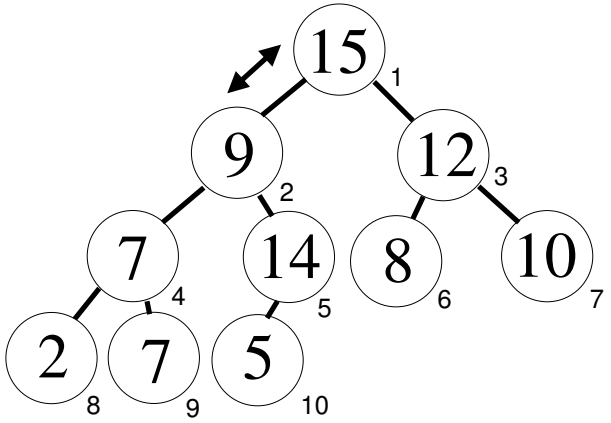
$\Rightarrow$ Jos keossa n solmua, sen korkeus on $\Theta(\lg n)$.

Alkion lisääminen kekoon ylhäältä:

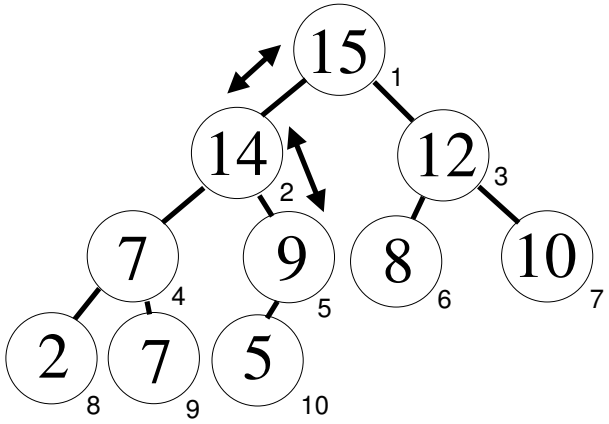
- oletetaan, että $A[1 \ldots n]$ on muuten keko, mutta keko-ominaisuus ei päde kekopuun juurelle
  - toisin sanoen $A[1] < A[2]$ tai $A[1] < A[3]$



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



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



Sama pseudokoodina:

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

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

Keon rakentaminen

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

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

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

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

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

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

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

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

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

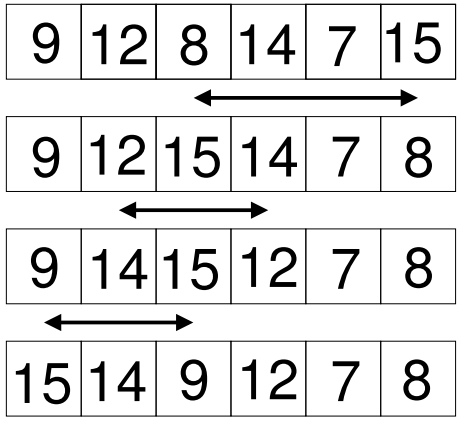
8.3 Järjestäminen keon avulla

Taulukon alkioiden ärjestäminen voidaan toteuttaa tehokkaasti kekoa hyödyntäen:

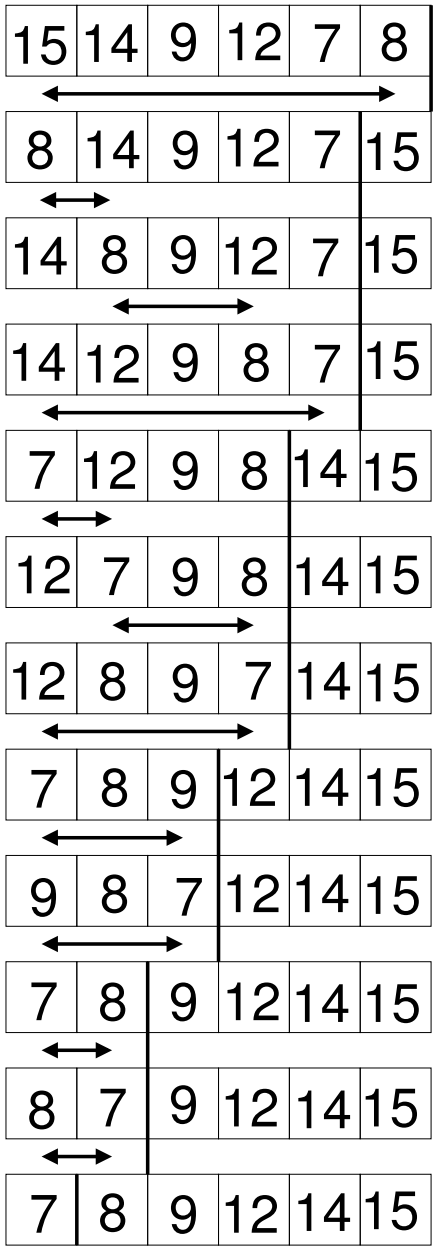
```
HEAPSORT( A )
1  BUILD-HEAP( A )                    (muutetaan taulukko keoksi)
2  for i := A.length downto 2 do      (käydään taulukko läpi lopusta alkuun)
3      exchange A[ 1 ] ↔ A[ i ]       (siirretään keon suurin alkio keon viimeiseksi)
4      A.heapsize := A.heapsize − 1   (siirretään suurin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

Esitetäänpä sama kuvien avulla:

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



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



HEAPSORTIN suoritusaika koostuu seuraavista osista.

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

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

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

- Kekoa kannattaa käyttää ainoastaan silloin!

HEAPSORTIN etuja ja haittoja

Etuja:

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

Haittoja:

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

8.4 Prioriteettijono

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

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

Prioriteettijonoilla on monia käyttökohteita

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

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

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

Sen sijaan keon avulla prioriteettijonon voi toteuttaa tehokkaasti.

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

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

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

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

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

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

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

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

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

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