5. Keko (eng. heap, suomeksi joskus myös kasa)

- tarkastellaan vielä yhtä tapaa toteuttaa sivulla 22 määritelty joukkotietotyyppi
- tällä kertaa emme kuitenkaan toteuta normaalia operaatiorepertoaaria
- olemme kiinnostuneita ainoastaan kolmesta operaatiosta:

- heap-insert(A,k)	lisää joukkoon avaimen k
- heap-min(A)	palauttaa joukon pienimmän avaimen arvon
- heap-del-min(A)	poistaa ja palauttaa joukosta pienimmän avaimen

- nämä operaatiot tarjoavaa kekoa sanotaan *minimikeoksi* (engl. minheap)
- toinen vaihtoehto, eli *maksimikeko* (engl. maxheap) taas tarjoaa operaatiot:

- heap-insert(A,k)	lisää joukkoon avaimen k
- heap-max(A)	palauttaa joukon suurimman avaimen arvon
- heap-del-max(A)	poistaa ja palauttaa joukosta suurimman avaimen

- näitä kolmea operaatiota sanotaan (maksimi/minimi) *keko-operaatioiksi*
- keskitymme tässä luvussa ensisijaisesti maksimikekoon ja sen toteutukseen

- pystyisimme luonnollisesti toteuttamaan operaatiot käyttäen jo tuntemiamme tietorakenteita:
  - käyttämällä *järjestämättömän listan* insert, delete ja max-operaatiota heap-insert olisi vakioaikainen mutta heap-del-max veisi aikaa $\mathcal{O}(n)$
  - käyttämällä *järjestetyn rengaslistan* insert, delete ja max-operaatiota heap-insert veisi aikaa $\mathcal{O}(n)$ ja heap-del-max olisi $\mathcal{O}(1)$
  - käyttämällä *punamustan puun* insert, delete ja max-operaatiota sekä heap-insert että heap-del-max veisivät aikaa $\mathcal{O}(log\ n)$

- seuraavassa esittämämme keko-tietotyypin toteutuksessa sekä heap-del-max (tai minimikeossa heap-del-min) että heap-insert toimivat ajassa $\mathcal{O}(log\ n)$ ja heap-max (tai minimikeossa heap-min) toimii vakioajassa

- tässä vaiheessa saattaa herätä kysymys mihin tarvitsemme näin spesialisoitunutta tietorakennetta, eikö riittäisi että käyttäisimme esim. tasapainoitettua hakupuuta, sillä näin saavutettu keko-operaatioiden tehokkuus olisi (operaatiota heap-max lukuun ottamatta) yhtä hyvä kuin pian esitettävällä varsinaisella keko-toteutuksella?
- tulemme kurssin aikana näkemään algoritmeja mitkä käyttävät aputietorakenteenaan kekoa ja näiden algoritmien tehokkaan toteutuksen kannalta keko-operaatioiden tehokkuus on oleellinen

- vaikka keko ei tuokaan kertaluokkaparannusta operaatioihin, on se käytännössä esim. punamustaan puuhun perustuvaa toteutusta huomattavasti nopeampi
- esimerkkejä keon sovelluksista:
- minimi-keon avulla saamme toteutettua tehokkaasti *prioriteettijonon*:
  - heap-insert vie jonottajan jonoon, avain vastaa jonottajan prioriteettia (mitä pienempi numeroarvo sitä korkeampi prioriteetti, ykkösluokka, kakkosluokka, jne)
  - heap-del-min ottaa jonosta seuraavaksi palveltavaksi korkeimman prioriteetin omaavan jonottajan
- keon avulla saamme myös toteutettua erittäin tehokkaan järjestämisalgoritmin
  - oletetaan että A on n-paikkainen kokonaislukutaulukko
  - seuraava algoritmi järjestää A:n alkiot suuruusjärjestykseen käyttäen kekoa H aputietorakenteena

    ```
    heap-sort(A,n)
    1  for i ←1 to n do
    2      heap-insert(H,A[i])
    3  for i ←1 to n do
    4      A[i] ←heap-del-min(H)
    ```

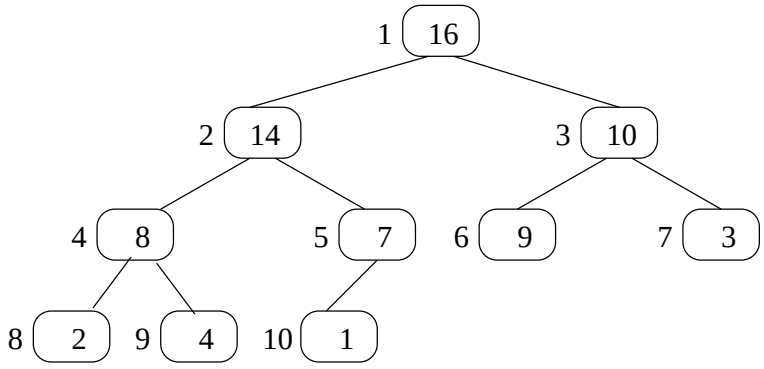
  - algoritmin aikavaativuus on $\mathcal{O}(n\ log\ n)$
  - palaamme tarkemmin kekojärjestämiseen luvussa 6
- myös verkkoalgoritmien yhteydessä (luvussa 7) löydämme käyttöä keolle

5.1 maksimi-keon toteuttaminen

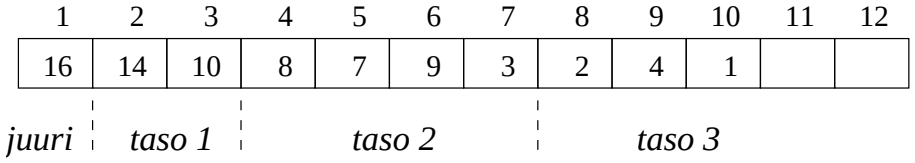
- keko kannattaa ajatella binääripuuna joka on talletettu muistiin taulukkona
- binääripuu on keko jos

  (K1) kaikki lehdet ovat kahdella vierekkäisellä tasolla k ja $k+1$ siten että tason $k+1$ lehdet ovat niin vasemmalla kuin mahdollista ja kaikilla k:ta ylempien tasojen solmuilla on kaksi lasta

  (K2) jokaiseen solmuun talletettu arvo on suurempi kuin solmun lapsiin talletetut arvot
- seuraava binääripuu on keko



- keko-ominaisuuden **(k1)** ansiosta puu voidaan esittää taulukkona missä solmut on lueteltu tasoittain vasemmalta oikealle
- yllä oleva puu taulukkoesityksenä:



- käytännössä keko kannattaa aina tallentaa taulukkoa käyttäen
- keko-taulukkoon A liittyy kaksi attribuuttia
  - length[A] kertoo taulukon koon
  - heap-size[A] kertoo montako taulukon paikkaa (alusta alkaen) kuuluu kekoon
  - huom: taulukossa A voi siis kohdan heapsize jälkeen olla "kekoon kuulumattomia" lukuja
- keon juurialkio on talletettu taulukon ensimmäiseen paikkaan A[1]
- taulukon kohtaan i talletetun solmun vanhemman sekä lapset tallettavat taulukon indeksit saadaan selville seuraavilla seuraavilla apu-operaatioilla:

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

  left(i)
  return 2i

  right(i)
  return 2i+1

- vaikka varsinaisia linkkejä ei ole, on keossa liikkuminen todella helppoa,
  - tarkastellaan edellisen sivun esimerkkitapausta
  - juuren $A[1]$ vasen lapsi on paikassa $A[2*1] = A[2]$ ja oikea lapsi paikassa $A[2*1+1] = A[3]$

  - solmun $A[5]$ vanhempi on $A[\lfloor 5/2 \rfloor] = 2$, vasen lapsi $A[10]$ mutta koska heap-size[A]=10, niin oikeaa lasta ei ole

- voimme lausua kekoehdon **(K2)** nyt seuraavasti:
  kaikille $1 < i \leq$ heap-size pätee $A[parent(i)] \geq A[i]$

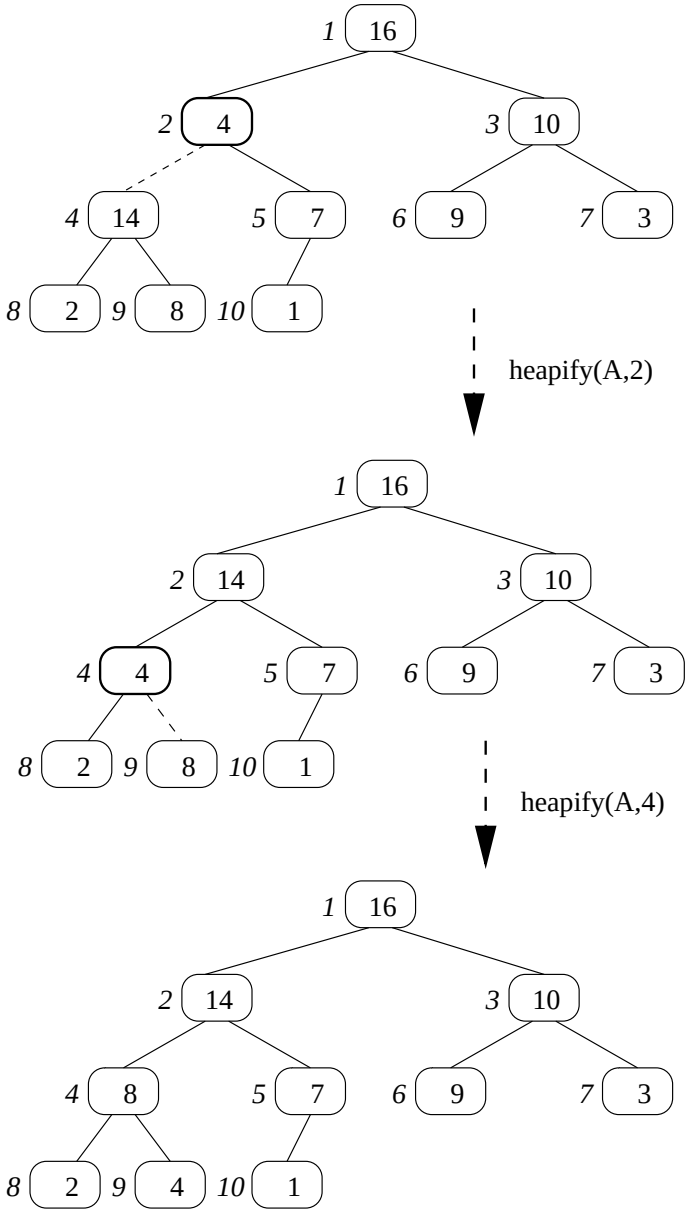
- ennen kuin voimme toteuttaa varsinaiset keko-operaatiot, toteutamme apuoperaation heapify
  - parametreina taulukko A ja indeksi i
  - oletuksena on että left(i) ja right(i) viittaavat jo kekoja olevien alipuiden A[left(i)] A[right(i)] juuriin
  - operaatio kuljettaa alkiota A[i] alaspäin kunnes alipuusta jonka juurena A[i] on tulee keko

```
heapify(A,i)
1  l ←left(i)
2  r ←right(i)
3  if r ≤ heap-size[A] then
4       if A[l] > A[r] then largest ←l else largest ←r
5       if A[i] < A[largest] then
6            vaihda A[i] ja A[largest]
7            heapify(A,largest)
8  else if l=heap-size and A[i]<A[l] then
9            vaihda A[i] ja A[l]
```

- jos molemmat lapset ovat olemassa (if rivillä 3), vaihdetaan tarvittaessa (if rivillä 4) $A[i]$:n arvo lapsista suuremman arvoon ja kutsutaan lapselle heapify-operaatiota

- jos vain vasen lapsi on olemassa ja tämän arvo suurempi kuin A[i]:n, vaihdetaan arvot keskenään (rivi 8)

- seuraava kuvasarja valottaa operaation toimintaa



- heapify-operaation suoritusaika riippuu ainoastaan puun korkeudesta, rekursiivisia kutsuja tehdään pahimmassa tapauksessa puun korkeuden verran
- n alkioisen keon korkeus selvästi $\mathcal{O}(log\ n)$ sillä keko on lähes täydellinen binääripuu

- n alkiota sisältävälle keolle tehdyn heapify-operaation pahimman tapauksen aikavaativuus on siis $\mathcal{O}(\log n)$

- kekoehdosta **(K2)** seuraa suoraan että keon maksimialkio on talletettu paikkaan A[1]
- operaatio heap-max siis on triviaali ja vie vakioajan

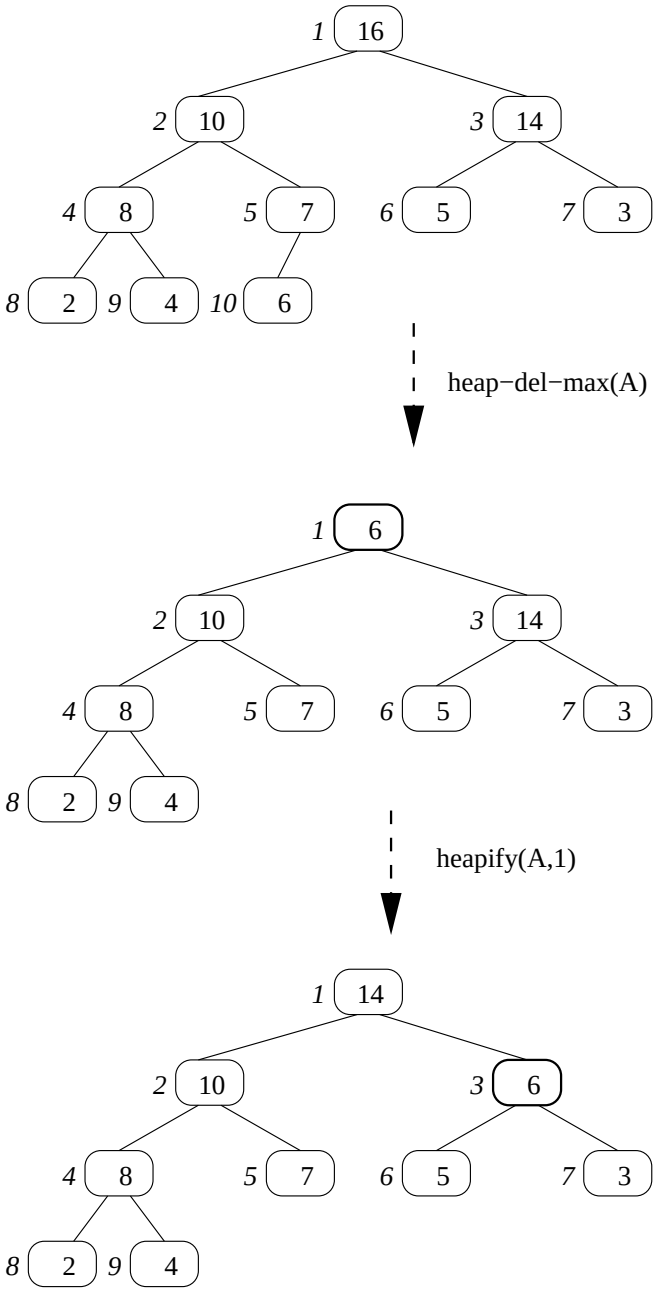
```
heap-max(A)
    return A[1]
```

- alkion poistaminen keosta

```
heap-del-max(A)
1   max ←A[1]
2   last ←heap-size[A]
3   A[1] ←A[last]
4   heap-size[A] ←last-1
5   heapify(A,1)
6   return max
```

- toimintaidea
  - operaatio palauttaa kohdassa A[1] olleen avaimen
  - keon viimeisessä paikassa oleva alkio A[last] viedään poistetun alkion tilalle ja keon koko pienennetään yhdellä (rivit 3 ja 4)
  - keko on muuten kunnossa mutta kohtaan A[1] siirretty avain saattaa rikkoa keko-ominaisuuden, kutsutaan heapify operaatiota korjaamaan tilanne
- operaation aikavaativuus sama kuin heapifyllä, eli $\mathcal{O}(log\ n)$

- esimerkki heap-del-max operaation toiminnasta



- alkion lisääminen kekoon

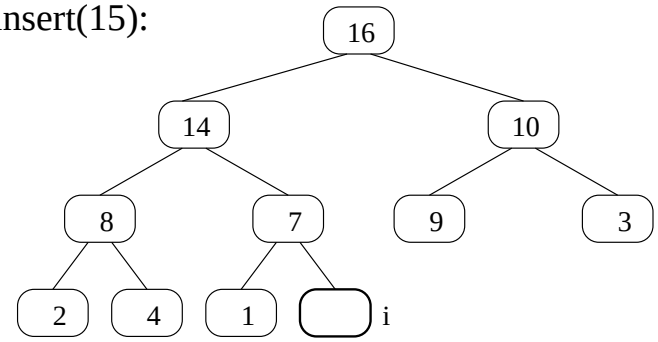
```
heap-insert(A,k)
1  i ←heap-size[A]+1
2  heap-size[A] ←i
3  while i>1 and A[parent(i)] < k do
4       A[i] ←A[parent(i)]
5       i ←parent(i)
6  A[i] ←k
```

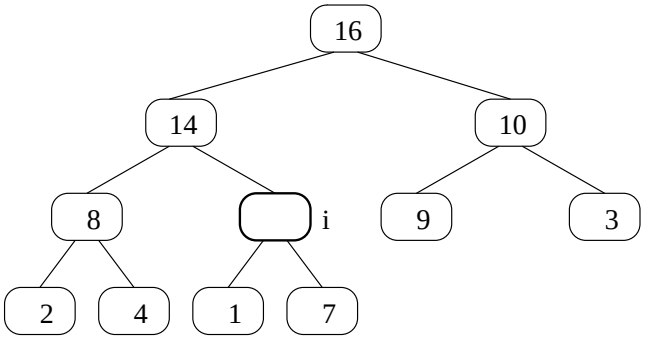
- toimintaidea
  - kasvatetaan keon kokoa yhdellä solmulla eli tehdään paikka uudelle avaimelle
  - kuljetaan nyt keon uudesta solmusta ylöspäin ja siirretään arvoja samaan aikaan yhtä alemmas niin kauan kunnes uudelle alkiolle löydetään paikka joka ei riko kekoominaisuutta **(K2)**

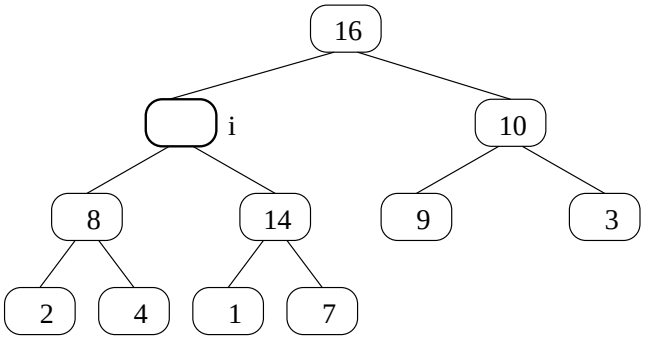
- pahimmassa tapauksessa avain lisätään juureen jolloin puun korkeudellisen verran avaimia on valutettu alaspäin
- operaation aikavaativuus siis on $\mathcal{O}(log\ n)$

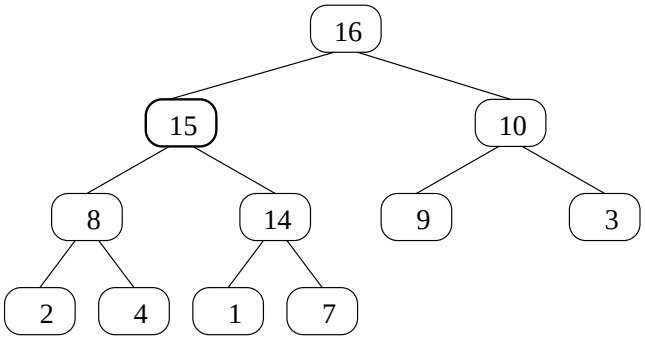
- seuraavalla sivulla esimerkki operaation toiminnasta

heap−insert(15):









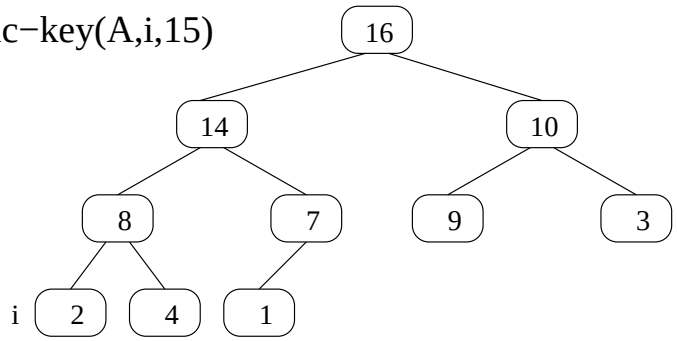
- jotkut sovellukset tarvitsevat seuraavaa keko-operaatiota joka kasvattaa annetussa indeksissä olevan avaimen arvoa

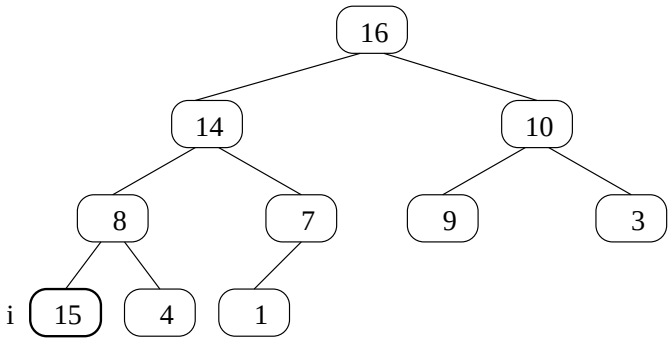
```
heap-inc-key(A,i,newk)
1   if newk > A[i] then
2       A[i] ←newk
3       while i>1 and A[parent(i)] < A[i] do
4           vaihda A[i] ja A[parent[i]]
5           i ←parent(i)
```

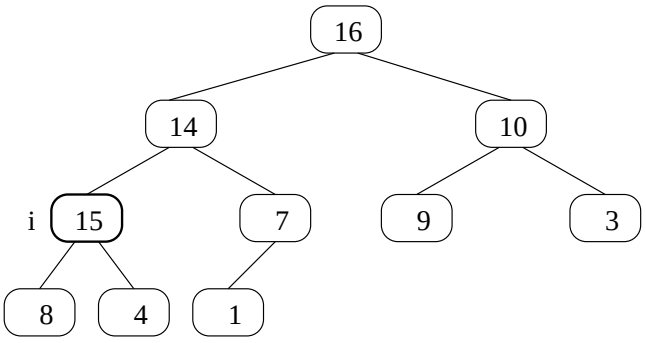
- toimintaidea
  - jos yritetään pienentää avaimen arvoa, operaatio ei tee mitään
  - kopioidaan keon kohtaan i uusi avaimen arvo (rivi 2)
  - jos kasvatettu avain rikko keko-ominaisuuden **(K2)**, vaihdetaan sen arvo vanhemman kanssa niin monta kertaa kunnes oikea paikka löytyy (rivit 3-5)

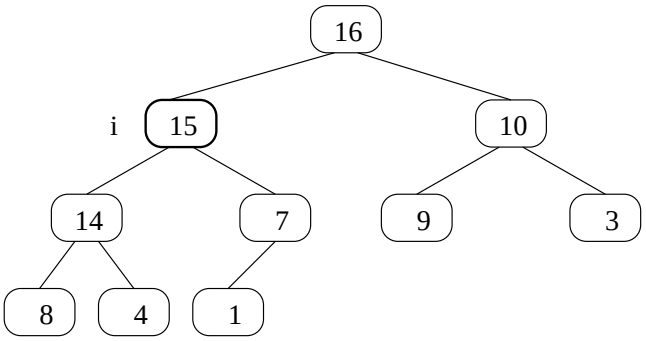
- pahimmassa tapauksessa lehdessä olevaa avainta muutetaan ja muutettu avain joudutaan kuljettamaan aina puun juureen saakka
- operaation aikavaativuus on $\mathcal{O}(log\ n)$
- seuraavalla sivulla esimerkki operaation toiminnasta

hash−inc−key(A,i,15)







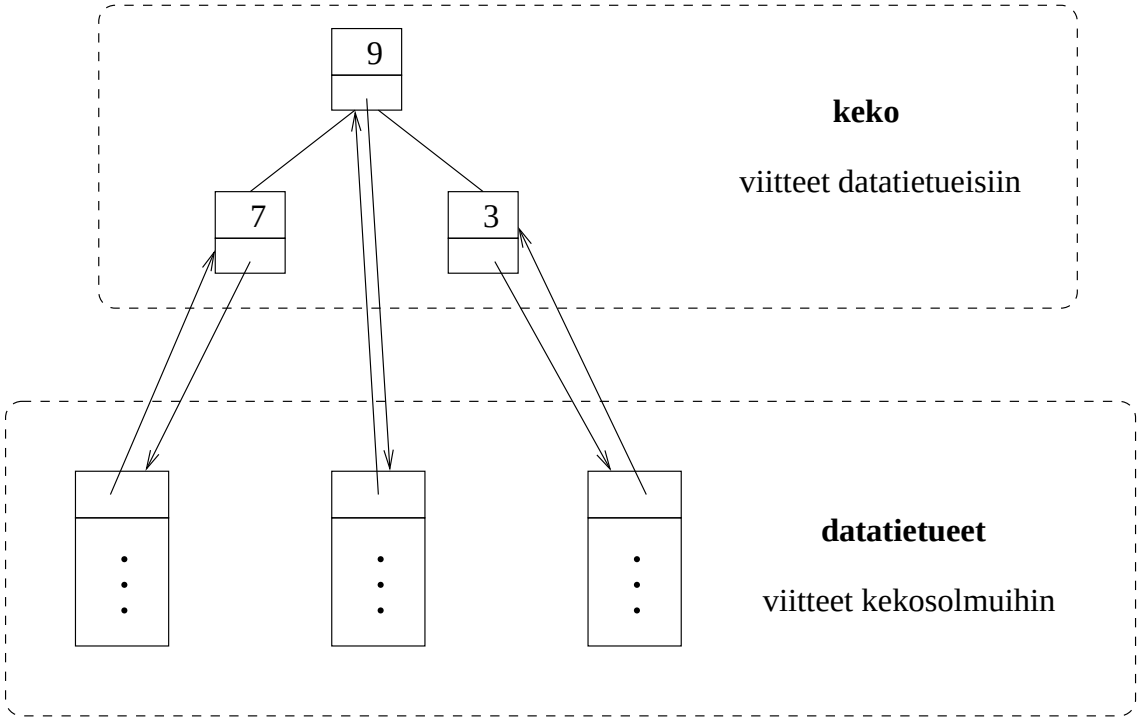


5.2 Keko käytännössä

- monissa käytännön sovelluksissa, esim. käytettäessä kekoa prioriteettijonoa, keottavat alkiot sisältävät muitakin kenttiä kuin pelkän avaimen
- tällöin itse kekoon ei välttämättä kannata tallettaa muuta kuin avaimet sekä linkit avaimeen liittyvään datakenttään
  - jos esim. toteuttaisimme käyttöjärjestelmässä prosessien skedulointijonon käyttäen kekoa ei koko prosessikuvaajaa kannattane viedä kekoon
- tällaisessa käytännön tilanteessa keko-operaatioiden parametrit kannattanee valita seuraavasti

- heap-insert(A,x,k)	lisää kekoon tietueen x jolla avain k
- heap-max(A)	palauttaa *viitteen* tietueeseen millä oli avaimena keon maksimi-arvo
- heap-del-max(A)	palauttaa viitteen tietueeseen millä oli avaimena keon maksimi-arvo ja poistaa tietueen keosta
- heap-inc-key(x,newk)	kasvattaa tietueen x avainta antaen sille uuden arvon $newk$

- jotta operaatio heap-inc-key saadaan toteutettua tehokkaasti data-tietueista on myös oltava linkki vastaavaan kekoalkioon
- tätä linkkiä kutsutaan joskus *kahvaksi* (engl. handle)
- muistin organisointi näyttää esim. seuraavalta:


9
7
3
keko
viitteet datatietueisiin
datatietueet
viitteet kekosolmuihin

6. Järjestäminen

- osaamme jo muutamia $\mathcal{O}(n^2)$ ajassa toimivia menetelmiä (mm. lisäysjärjestäminen luvusta 1) jotka järjestävät n lukua suuruusjärjestykseen
- tarkastellaan tässä luvussa muutamaa menetelmää joilla järjestämisessä päästään aikaan $\mathcal{O}(n\ log\ n)$

6.1 Kekojärjestäminen

- sivulla 151 hahmottelimme jo idean miten kekoa voidaan käyttää aikaansaamaan ajassa $\mathcal{O}(n\ log\ n)$ toimiva järjestämisalgoritmi
- esitetään tässä käytännön tasolla hieman tehokkaammin toimiva versio kekojärjestämisestä

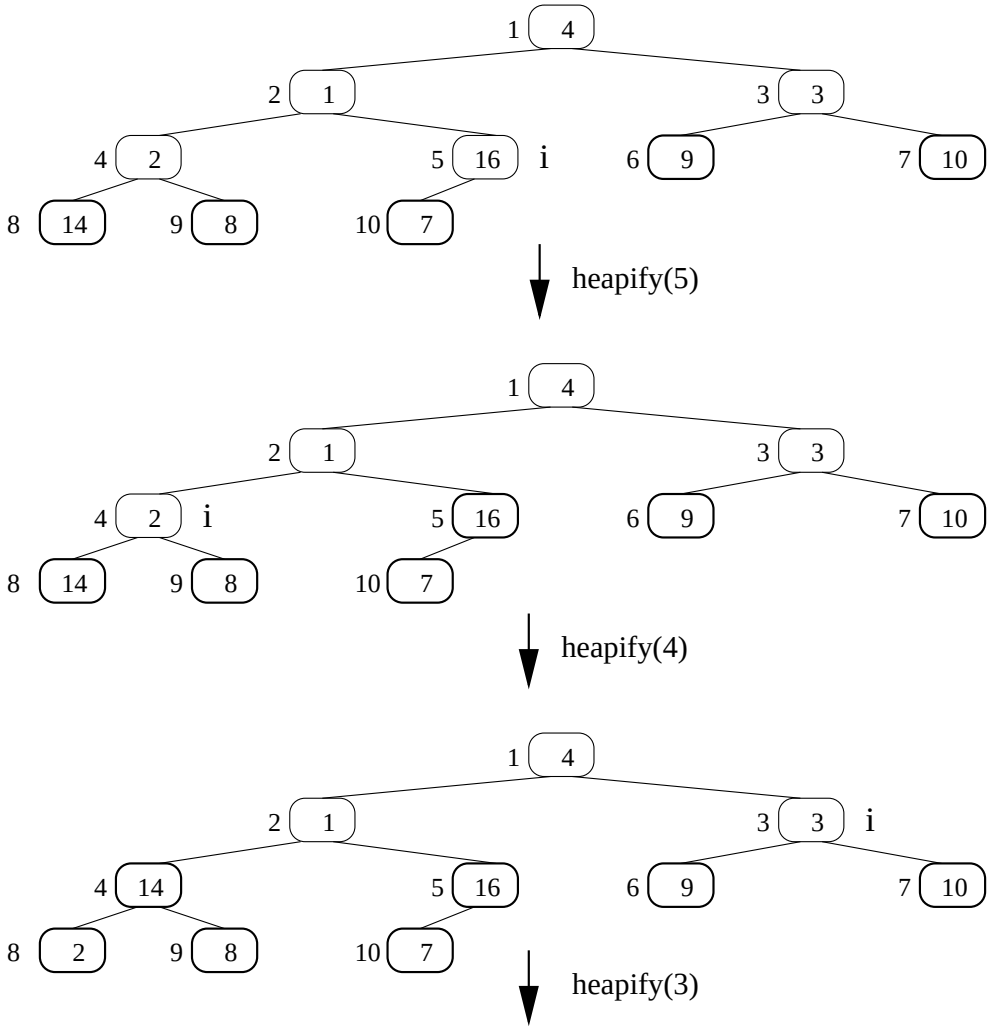
- sivulla 154 esittelimme operaation heapify(A,i) mikä toimii seuraavasti
  - oletuksena on että left(i) ja right(i) viittaavat jo kekoja olevien alipuiden A[left(i)] A[right(i)] juuriin
  - operaation suorituksen jälkeen alipuu A[i] on keko
- operaation avulla on helppo rakentaa mistä tahansa taulukosta A keko:
  - lehdet, eli paikossa A[$\lfloor n/2 \rfloor + 1$], ..., A[n] olevat yhden alkion alipuut ovat jo kekoja
  - kutsutaan heapify lapselliselle kekosolmulle alkaen A[$\lfloor n/2 \rfloor$]:sta aina juureen A[1] asti

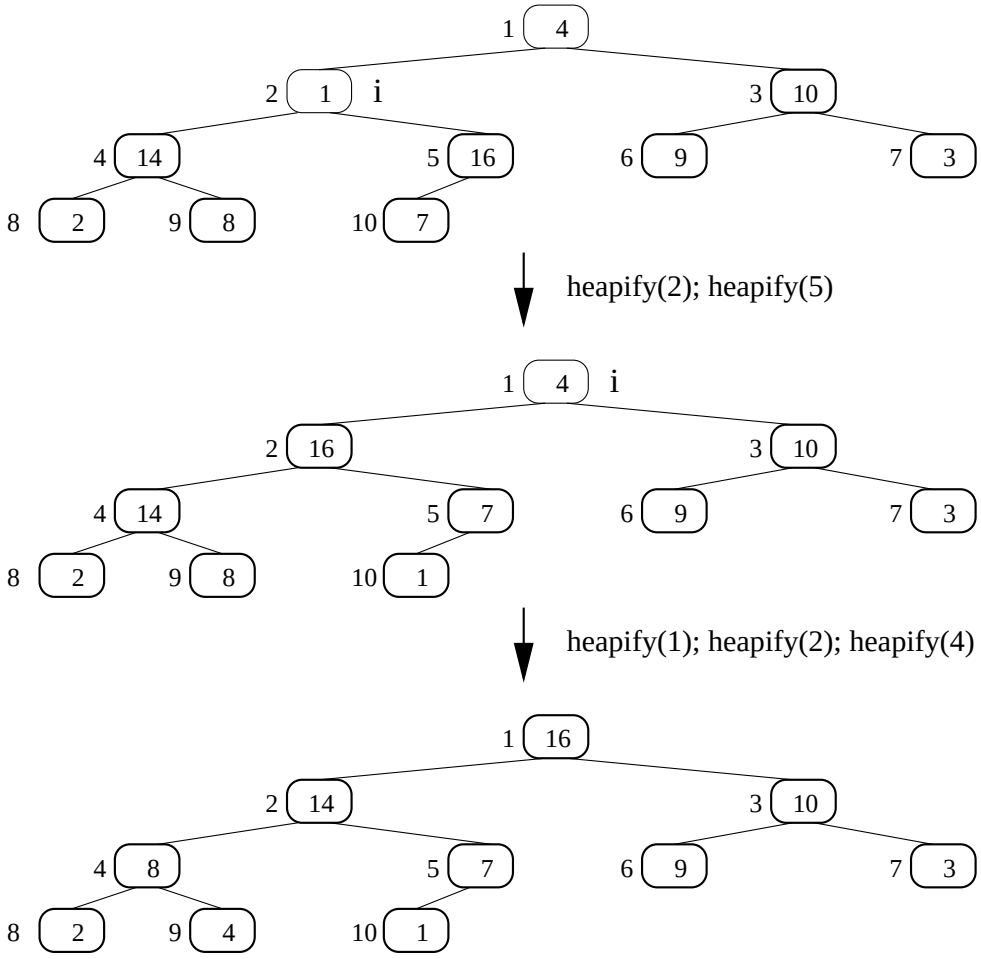
– näin taulukko A muuttuu keoksi

build-heap(A)
1 heap-size[A] ←length[A]
2 **for** i ←$\lfloor length[A]/2 \rfloor$ **downto** 1 **do**
3 heapify(A,i)

- seuraava kuvasarja valottaa operaation toimintaa

	1	2	3	4	5	6	7	8	9	10
alussa	4	1	3	2	16	9	10	14	8	7





tuloksena

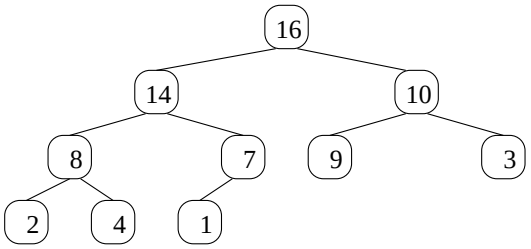
1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

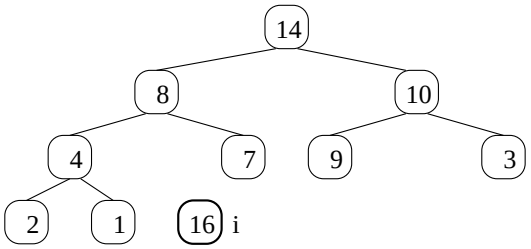
- kekojärjestäminen tapahtuu seuraavasti

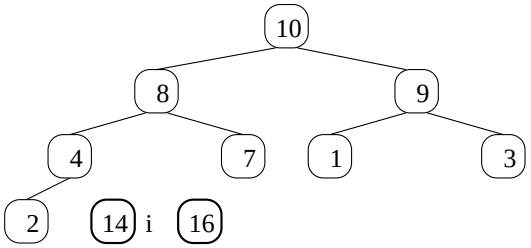
```
heap-sort(A)
1  build-heap(A)
2  for i ←length[A] downto 1 do
3      vaihda A[1] ja A[i]
4      heap-size[A] ←heap-size[A]-1
5      heapify(A,1)
```

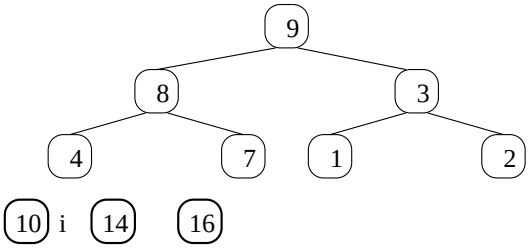
- toimintaidea

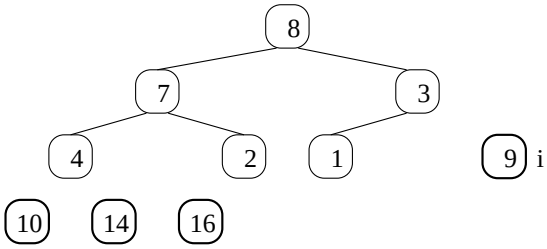
  - aineistosta tehdään ensin maksimikeko, näin suurin alkio on kohdassa A[1]
  - vaihdetaan keskenään keon ensim. ja viim. alkio
  - näin saamme yhden alkion vietyä taulukon loppuun "oikealle" paikalleen, pienentämällä keon kokoa yhdellä huolehditaan vielä että viimeinen alkio ei enää kuulu kekoon
  - paikkaan A[1] viety alkio saattaa rikkoa keko-ominaisuuden
  - huolehditaan vielä että keko-ominaisuus säilyy kutsumalla heapify(A,1)
  - toistetaan samaa niin kauan kun keossa on alkiota
- kekojärjestämisen aikavaativuus
  - heapify:n aikavaatimus $\mathcal{O}(log\ n)$ keolle jossa n alkiota
  - build-heap operaatiossa kutsutaan $n/2$ kertaa heapify keolle jossa *korkeintaan* n alkiota, siis build-heap käyttää aikaa *korkeintaan* $\mathcal{O}(n\ log\ n)$
  - heap-sortissa kutsutaan vielä $n-1$ kertaa heapify-operaatiota
  - kokonaisuudessaan kekojärjestämisen aikavaativuus on siis $\mathcal{O}(n\ log\ n)$
- hieman tarkempi analyysi (ks. Karvi tai Cormen) paljastaa että operaation build-heap vaativuus onkin vain $\mathcal{O}(n)$
- tilavaativuus
  - heapify kutsuu rekursiivisesti itseään pahimmillaan keon korkeudellisen verran, operaatio on kuitenkin helppo toteuttaa myös ilman rekursiota jolloin sen tilantarve vakio
  - muutkaan kekojärjestämisen toimet eivät aputilaa tarvitse, siis tilavaativuus $\mathcal{O}(1)$

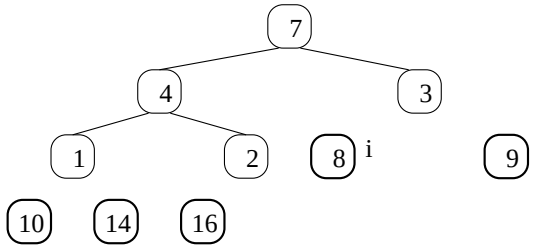


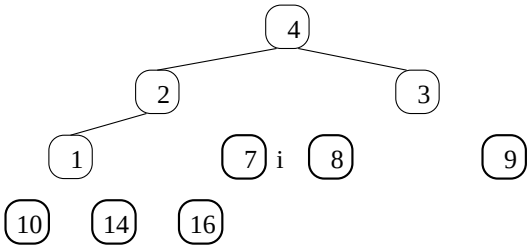


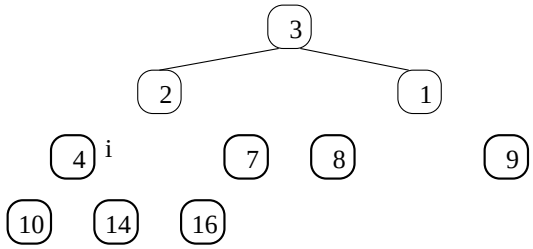


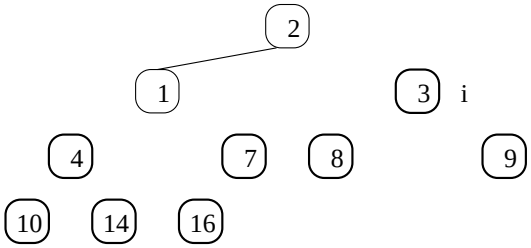


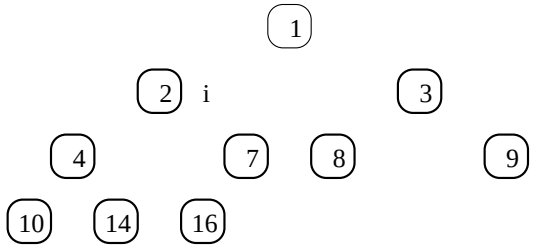












tuloksena

1	2	3	4	7	8	9	10	14	16