

Koe

- Koe ma 1.3 klo 16-19 salissa A111, koeaika kuten tavallista 2h 30min
- Kokeessa saa olla mukana A4:n kokoinen kaksipuolinen käsin tehty, itse kirjoitettu lunttilappu

Tärkeää ja vähemmäntärkeää

- Ensimmäisen periodin kolme ydinteemaa
 - aika- ja tilavaativuusanalyysi
 - linkitetty lista: erikoistapaukset ja listan eri versiot
 - tasapainottomat ja tasapainoiset hakupuut
- Kokeessa on kysymyksiä kaikista ydinteemoista
- Ensimmäisen välikokeen kannalta vähemmän tärkeää
 - yleisen puun esitystavat
 - puut ongelmanratkaisussa
 - Java-spesifiset asiat (ArrayList, LinkedList, TreeSet, TreeMap ...)
 - Invariantti
 - Rekursioyhtälöt

Aika- ja tilavaativuusanalyysi

- Mitä tarkoitetaan aika- ja tilavaativuudella? s. 26-30
 - Aikavaativuus: algoritmin käyttämä laskenta-"aika" syötteen koon funktiona
 - Tilavaativuus: algoritmin käyttämien apumuistipaikkojen/tilan määrä syötteen koon funktiona
- Tarkastellaan (lähes) koko kurssin ajan **pahimman tapauksen vaativuutta**, miten paljon aikaa/tilaa algoritmin suorituksessa kuluu enimmillään
- Ei olla kiinnostuneita liian tarkkoista laskelmista (s. 31-33 esimerkki liian tarkasta analyysistä)
- Kiinnostavampaa on mihin **vaativuusluokkaan** algoritmi kuuluu
 - vakioaikainen $\mathcal{O}(1)$
 - logaritminen $\mathcal{O}(\log n)$
 - lineaarinen $\mathcal{O}(n)$
 - neliöllinen $\mathcal{O}(n^2)$
 - eksponentiaalinen $\mathcal{O}(2^n)$

Aika- ja tilavaativuusanalyysi

- vaativuusluokkiin liittyen kannattaa muistaa
 - motivaatio sille, että "karkeampi analyysi" riittää, s. 34-37
 - mitä vaativuusluokka tarkoittaa, eli $\mathcal{O}$:n määritelmä, s. 38-42
- Pikakertaus $\mathcal{O}$ -käsitteestä:

jos ohjelman tarkka aikavaativuus on $3n^2 + 4n + 11$ niin sanotaan, että se on $\mathcal{O}(n^2)$, koska on olemassa funktio $d \cdot n^2$ joka kasvaa nopeammin kuin $3n^2 + 4n + 11$ kun valitaan vakio d sopivasti, esim. $d = 4$ ja tarkastellaan vain suuria n :n arvoja

usein riittää "hiha-analyysi": pyyhi pois muut kuin korkeimman asteen termi, ja pyyhi korkeimman asteen termin vakio pois, näin saat $\mathcal{O}$ -luokan
- Viikon 2 laskarien tehtävissä 1-3 tutkittiin $\mathcal{O}$ -käsitettä varsin matemaattisesti. Käsitteen matemaattinen osaaminen on hyödyksi, mutta vielä tärkeämpää on
 - **Ymmärtää miksi käsite on mielekäs** (esim. algoritmi jonka aikavaativuus $\mathcal{O}(\log n)$ on todella paljon nopeampi kuin $\mathcal{O}(n)$ -algoritmi kaikilla isoilla syötteillä)
 - **Osata analysoida algoritmien aika- ja tilavaativuuksia**

Aika- ja tilavaativuusanalyysi käytännössä

- Käytännön algoritmianalyysissa s. 43-50 nyrkkisääntöjen soveltaminen on avainasemassa
 - säännöt perustuvat s. 38-42 matemaattiseen kalustoon, ja
 - tuottavat saman vastauksen kuin s. 31-33 tyylinen tarpeettoman tarkka analyysi
- kolme ensimmäistä aikavaativuusanalysoijan nyrkkisääntöä
 - yksinkertaisten käskyjen (sijoitus, vertailu, tulostus, laskutoimitukset) aikavaativuus vakio eli $\mathcal{O}(1)$
 - peräkkäin suoritettavien käskyjen aikavaativuus on sama kuin käskyistä eniten aikaavievän aikavaativuus
 - ehtolauseen aikavaativuus on $\mathcal{O}(\text{ehdon testausaika} + \text{suoritetun vaihtoehdon aikavaativuus})$

Insertion-Smaller(A) // A on kokonaislukutaulukko

```
1  x = readInt() // luetaan syöte käyttäjältä
2  y = readInt() // luetaan toinen syöte käyttäjältä
3  z = readInt() // luetaan kolmas syöte käyttäjältä
4  n = A.length
5  if x < y and x < z
6      smallest = x
7  elif y < z
8      smallest = y
9  else
10     smallest = z
11  A[1] = smallest
12  A[n] = smallest
```

- syötteenä kokonaislukutaulukko A
- syötteen koko taulukon koko n , aikavaativuus ilmaistaan siis suhteessa syötteen kokoon
- aikavaativuus on vakio, eli $\mathcal{O}(1)$, sovellettu edellisen kalvon kolmea nyrkkisääntöä, tarkempi analyysi, ks s. 45
- tilavaativuus on selvästi vakio $\mathcal{O}(1)$, sillä riippumatta syötteenä olevan taulukon A koosta, metodi käyttää neljää apumuuttujaa

- kannattaa huomata, että vakioajassa toimivassa algoritmissa ei välttämättä suoriteta *aina* samaa määrää komentoja:

Insertion-If-Nonzero(A, x) // A on kokonaislukutaulukko

```
1  n = A.length
2  if x ≠ 0 ∧ n > 3
3      A[n-3] = x
4      A[n-2] = x
5      A[n-1] = x
6      A[n] = x
```

- jos parametri on x on nolla, ei rivejä 3-6 suoriteta, eli riippuen x:n arvosta komentoja suoritetaan joko 2 tai 6 kpl
- suoritettavien komentojen määrä **pahimmassa tapauksessa** on 6, eli riippumaton toisen parametrin eli taulukon A koosta
- koska komennot ovat yksinkertaisia komentoja, on pahimmassa tapauksessa suoritettavan käskyjonon 1-6 vaativuus $\mathcal{O}(1)$

- jos koodissa kutsutaan aliohjelmaa, on vaativuusanalyysissä huomioitava **aliohjelman suorituksen aikavaativuus**, joka on $\mathcal{O}(\text{parametrien evaluointiaika} + \text{parametrien sijoitusaika} + \text{aliohjelman käskyjen suoritus aika})$

- **Insertion-Smaller2(A)**

```
1  x = readInt()
2  y = readInt()
3  z = readInt()
4  n = A.length
5  smallest = min(x, y, z)
6  A[1] = smallest
7  A[n] = smallest
```

min(x,y,x)

```
1  if x<y and x<z
2      return x
3  elif y<z
4      return = y
5  else
6      return = z
```

- aliohjelman min aikavaativuus selvästi $\mathcal{O}(1)$
- Insertion-Smaller2 koostuu peräkkäisistä $\mathcal{O}(1)$ osista, eli sen aikavaativuus $\mathcal{O}(1)$

- **looppeja sisältävän algoritmin aikavaativuuden** arvioiminen:
 - arvioi sisimmän loopin runko-osan aikavaativuus: $\mathcal{O}(runko)$
 - arvioi kuinka monta kertaa sisin looppi yhteensä suoritetaan: lkm
 - aikavaativuus on $\mathcal{O}(lkm \times runko)$

Find(A,x) // A on kokonaislukutaulukko ja x etsittävä luku

```

1  i = 1
2  while i ≤ A.length
3      if A[i] == x
4          return true
5      i = i+1
6  return false

```

- loopin runko koostuu vakioaikaisista komennoista, eli runko vie $\mathcal{O}(1)$
- runko suoritetaan taulukon pituuden verran
- jos merkitään taulukon pituutta n :llä, on loopin vaativuus $\mathcal{O}(n \times 1) = \mathcal{O}(n)$
- koko aliohjelman vaativuus siis $\mathcal{O}(n)$ koska loopin ulkopuoleiset rivit 1 ja 6 vakioaikaisia ja looppi dominoi aikavaativuutta
- tilavaativuus on vakio sillä vain 1 apumuuttuja käytössä

Bubble-Sort(A)

```
1  for i = n to 2
2      for j = 1 to i-1
3          if A[j] > A[j+1]
4              // vaihdetaan A[j]:n ja A[j+1]:n sisältöä
5              apu = A[j]
6              A[j] = A[j+1]
7              A[j+1] = apu
```

- toimintaidea:

Taulukon osa $A[i + 1, \dots, n]$ järjestyksessä ja järjestetyn osan alkiot vähintään yhtä suuria kuin osan $A[1, \dots, i]$ alkiot

- loopin runko vakioaikainen sillä rungon suoritus tarkoittaa pahimmillaan neljän koodirivin suorittamista
- runko suoritetaan $1 + 2 + 3 + \dots + n = \frac{1}{2}n(n + 1)$ kertaa (ks. s. 50)
- aikavaativuus siis $\mathcal{O}(n^2)$
- myös karkeampi analyysi toisi saman tuloksen:
ulommainen for suoritetaan $n-1$ kertaa, sisempi for suoritetaan joka kerta *enimmillään* $n-2$ kertaa, eli kokonaisuudessaan runkoa ei suoriteta ainakaan enempää kuin $(n - 1)(n - 2) = n^2 - 3n + 2$ kertaa joka on myös $\mathcal{O}(n^2)$

- rekursiota käyttävien algoritmien aikavaativuuden arvioiminen menee hyvin samaan tyyliin kuin looppeja sisältävien algoritmien analyysi:
 - arvioi pelkän rekursiivisen aliohjelman runko-osan aikavaativuus: $\mathcal{O}(\text{runko})$
 - arvioi kuinka monta kertaa rekursiokutsu yhteensä suoritetaan: lkm
 - aikavaativuus on $\mathcal{O}(lkm \times \text{runko})$
- esim. rekursiivinen etsintä taulukosta

Stupid-Find(A,i,x)

```

1  if i > A.length
2      return false
3  elif A[i] = x
4      return true
5  else
6      return Stupid-Find(A,i+1,x)
```

- Jos halutaan etsiä taulukosta A lukua 7, tapahtuu etsintä kutsulla Stupid-Find(A,1,7), toinen parametri kertoo, että etsintä alkaa taulukon paikasta numero 1
- Pelkän rungon aikavaativuus vakio $\mathcal{O}(1)$
- Jos taulukon koko on n tehdään rekursiivisia kutsuja $n + 1$, eli algoritmin suorituksen aikavaativuus on $\mathcal{O}(n)$

- **Rekursiivisen algoritmin tilavaativuus**

- apumuuttujia ei ole käytössä, mutta yllättäen tilavaativuus onkin $\mathcal{O}(n)$, ks. s. 61
- rekursion syvyys määrittää tilavaativuuden

- Tässä esitetyillä nyrkkisäännöillä pärjää aika pitkälle TiRa:n ensimmäisessä välikokeessa
- Monisteen s. 55-56 ja 59-60 mainitut rekursioyhtälöt voi ensimmäisessä välikokeessa unohtaa, asiaan palataan seuraavassa periodissa
- Monisteen s. 62-70 "vaativuusanalyysiä kansantajuisesti" kannattaa lukea

Linkitetty lista

- **linkitettyt rakenteet (s. 94-126) on kaikkien TiRa-opiskelijoiden syytä hallita**
 - miten rakenne muodostuu (listasolmut viittaavat toisiin listasolmuihin)
 - miten rakenteita käydään läpi
 - miten alkioita lisätään ja poistetaan
 - ym.
 - operaatioiden aikavaativuus (yleensä joko $\mathcal{O}(1)$ esim. lisää alkuun tai $\mathcal{O}(n)$ käy pahimmassa tapauksessa kaikki läpi)
- listan eri versiot
 - yhteen ja kahteen suuntaan linkitettyt
 - alkiot järjestyksessä vai ei?
- listan "erikoistapaukset": pino ja jono
 - materiaalissa myös taulukkoon perustuvat toteutukset (s. 82-93). ne ovat ok, mutta TiRa:n kannalta kovaa ydintä ovat nimenomaan linkitettyt toteutukset

Linkitetty lista: mihin tarvitaan?

- ohjelma haluaa ylläpitää suoritusaikana vaihtelevakokoista joukkoa:
 - **search(S,k)** jos joukosta löytyy avaimella k varustettu alkio, operaatio palauttaa viitteen alkioon, muuten operaatio palauttaa viitteen NIL
 - **insert(S,x)** lisää joukkoon alkion johon x viittaa
 - **delete(S,x)** poistaa tietueen johon x viittaa
 - **min(S)** palauttaa viitteen joukon pienimpään alkioon
 - **max(S)** palauttaa viitteen joukon suurimpaan alkioon
 - **succ(S,x)** palauttaa viitteen alkiota x seuraavaksi suurimpaan alkioon, jos x oli suurin palauttaa NIL
 - **pred(S,x)** palauttaa viitteen alkiota x seuraavaksi pienempään alkioon, jos x oli pienin palauttaa NIL
- listat ovat yksi (ei kovin tehokas) tapa toteuttaa tietotyyppi joukko
- kuten toisessa periodissa nähdään, monet tietorakenteet tarvitsevat listoja aputietorakenteena (hajautus: ylivuotoketjut, verkko: vieruslistat)
- myös pino ja jono ovat tarpeellisia osia tietyissä algoritmeissa (esim. puun leveyssuuntainen läpikäynti, sulutuksen tarkastus, ...)

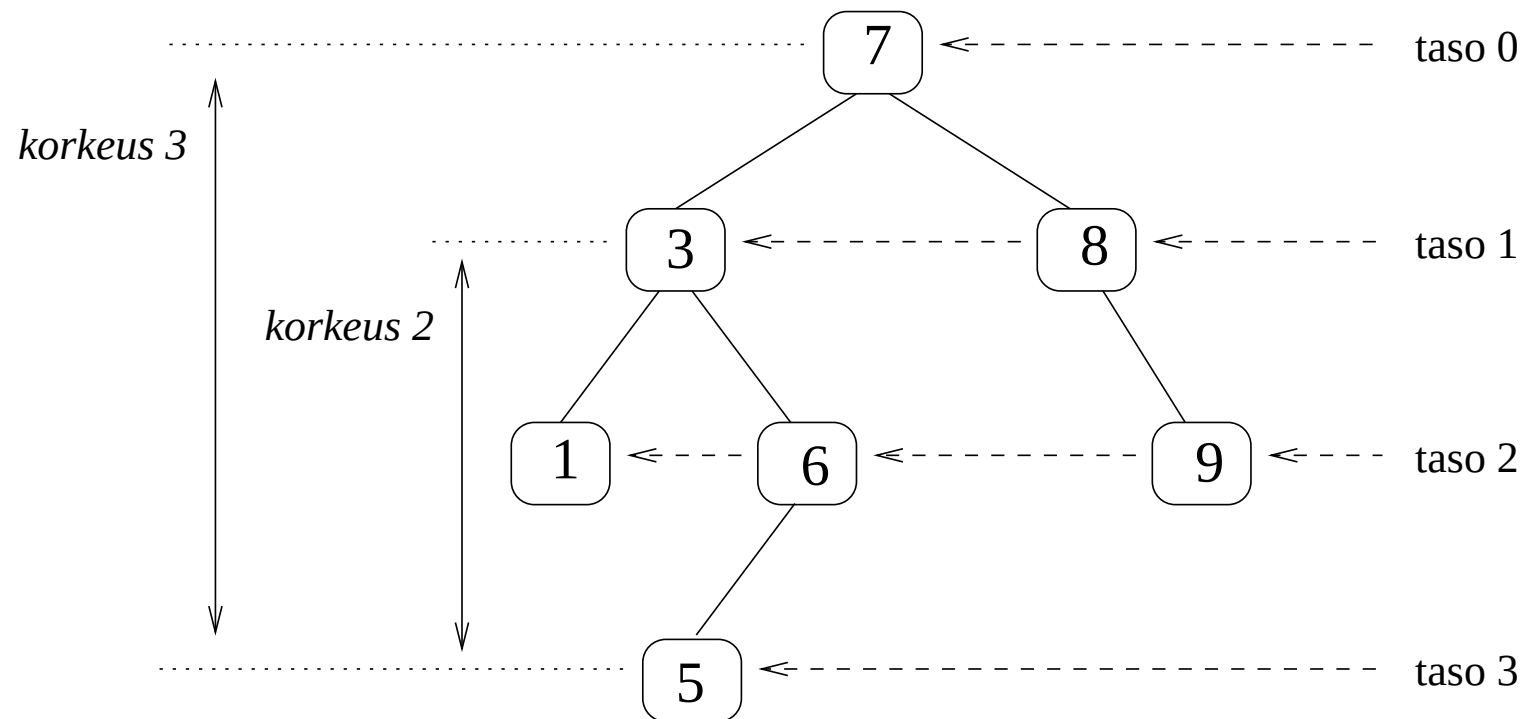
- On paljon listaa tehokkaampia tapoja toteuttaa tietotyyppi joukko

	lista1	lista2	tasap.puu	hajautusrak
search	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$
insert	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$
delete	$O(1)$	$O(1)$	$O(\log n)$	$O(1)$
min	$O(n)$	$O(1)$	$O(\log n)$	$O(n)$
max	$O(n)$	$O(1)$	$O(\log n)$	$O(n)$
succ	$O(n)$	$O(1)$	$O(\log n)$	$O(n)$
pred	$O(n)$	$O(1)$	$O(\log n)$	$O(n)$

- Lista 1 tarkoittaa järjestämätöntä linkitettyä listaa ja lista 2 järjestettyä linkitettyä listaa

tasapainottomat ja tasapainoiset binäärihakupuut

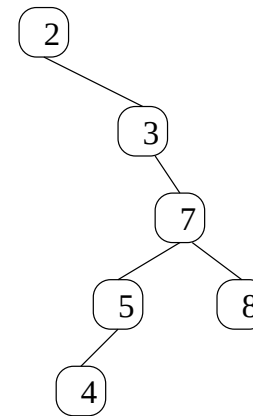
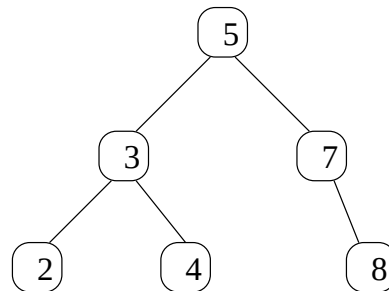
- tärkeitä käsitteitä
 - solmu, juuri, lehti, vanhempi, vasen lapsi, oikea lapsi, vasen alipuu, oikea alipuu
 - korkeus ja tasot:



- huom: puun korkeus = juuren korkeus = puun matalimmalla olevan tason numero

binäärihakupuuehto

- Binäärihakupuussa avaimet toteuttavat **binäärihakupuuehdon**:
 - jos solmu v on solmun x vasemmassa alipuussa, niin $v.key < x.key$
 - jos solmu o on solmun x oikeassa alipuussa, niin $x.key < o.key$
- eli **solmun vasemmassa alipuussa on ainoastaan sitä pienempiä ja oikeassa alipuussa sitä isompia avaimia**
- esim: kaksi rakenteeltaan erilaista binäärihakupuuta joissa alkiot 2, 3, 4, 5, 7 ja 8



- **hakupuuehto erittäin tärkeä ymmärtää**, sillä käytännössä kaikki puiden joukko-operaatiot (search, insert, delete, min, succ, max ja pred) perustuvat ehdon voimassaoloon

joukko-operaatiot tasapainottamattomassa puussa

- operaatioiden search, insert, delete, min, succ, max ja pred toimintaperiaate (varsinkin 3 ensimmäistä) syytä hallita (s. 144-165)
 - operaatioiden toimintaperiaate ja sen yhteys binäärihakupuuuehtoon
 - operaatioiden aikavaativuus ja tilavaativuus
- esim. operaatio insert
 - ensin etsitään paikka mihin lisättävä avain väistämättömästi binäärihakupuuuehdon nojalla kuuluu
 - lisäyspaikka on aina jonkin puun lehtisolmun vasen tai oikea alipuu
 - lisäys siis kulkee aina polun puun juuresta lehteen
 - lisäyksen aikavaativuus riippuu puun korkeudesta h sillä lisäyspaikan etsivä looppi suoritetaan h kertaa
 - loopin rungon aikavaativuus on $\mathcal{O}(1)$, eli koko loopin aikavaativuus $\mathcal{O}(h)$
 - muut osat (rivit 1-8 ja 14-17) operaatiosta vakioaikaisia, siispä operaation aikavaativuus $\mathcal{O}(h)$, missä h siis puun korkeus

```

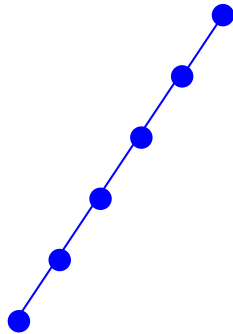
insert(T, k)
1  uusi = new puusolmu
2  uusi.key = k
3  uusi.left = uusi.right = uusi.parent = NIL
4  if T.root == NIL          // jos puu on tyhjä, tulee uudesta solmusta juuri
5      T.root = uusi
6      return
7  x = T.root
8  p = NIL
9  while x ≠ NIL              // etsitään kohta, johon uusi alkio kuuluu
10     p = x
11     if k < x.key
12         x = x.left
13     else x = x.right
14  uusi.parent = p           // viitteet uuden alkion ja sen vanhemman välille
15  if uusi.key < p.key
16     p.left = uusi
17  else p.right = uusi

```

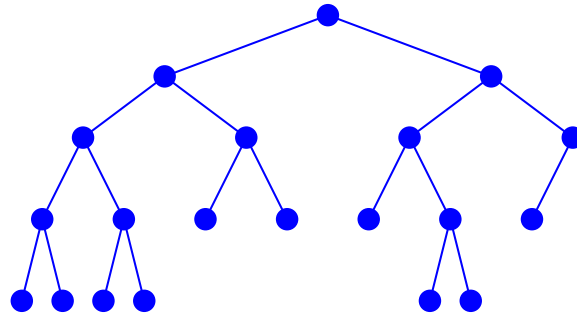
- huomataan (s. 144-165), että kaikkien muidenkin operaatioiden vaativuus riippuu suoraan puun korkeudesta h , eli **puun korkeus on kriittinen asia tehokkuuden kannalta**

puun korkeus vs. solmumäärä

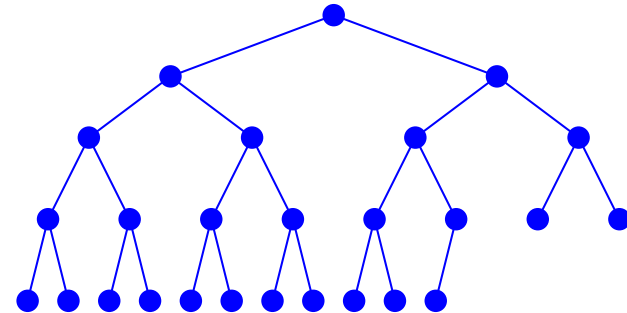
- puita on hyviä ja huonoja



lineaarinen



tasapainoinen?



melkein täydellinen

- **tärkeitä tuloksia:** jos puussa n solmua, niin
 - tasapainoittamattoman puun korkeus parhaimmillaan $\mathcal{O}(\log n)$ mutta pahimmillaan $\mathcal{O}(n)$ (s. 132-137)
 - AVL-puun korkeus parhaimmillaan $\mathcal{O}(\log n)$ ja myös pahimmillaan $\mathcal{O}(\log n)$ (s. 172-182)
- tulokset ja niiden seuraukset on pakko kaikkien tietää ja on hyvä ymmärtää vähintään maalaisjärjen tasolla mistä on kysymys (esim. s. 134 ja 136 kuvat)
- AVL-puun korkeuden ylärajan matemaattista todistusta ei tarvitse kokeessa osata

puun solmujen läpikäynti: esi-, sisä- ja jälkijärjestys

- joukko-operaatioiden lisäksi tärkeää ovat erilaiset puiden läpikäynnit
- läpikäynti voitaisiin periaatteessa suorittaa left-, right- ja parent-linkkejä pitkin kulkien, mutta lienee helpointa tehdä rekursion avulla
- tulostetaan puun solmut **sisäjärjestyksessä**: käsittele ensin vasen alipuu, sitten käsittele solmu, eli tulosta avaimen arvo ja sen jälkeen käsittele oikea alipuu

```
inorder-tree-walk(x)
```

```
  if  $x \neq \text{NIL}$ 
```

```
    inorder-tree-walk(x.left)    // käsitellään vasen alipuu
```

```
    print x.key
```

```
    inorder-tree-walk(x.right)   // käsitellään oikea alipuu
```

- kutsutaan parametrina puun juuri: `inorder-tree-walk(T.root)`
- rungon suoritus vie vakioajan sillä rungossa vain yksi komento operaatio tulee kutsutuksi kerran jokaiselle puun solmulle eli jos solmuja on n , on aikavaativuus $\mathcal{O}(n)$
- hakupuuehto takaa, että sisäjärjestyksessä tapauksessa käsitellään solmut niiden avaimien mukaisessa suuruusjärjestyksessä (s. 140-143)

puun solmujen läpikäynti: esi-, sisä- ja jälkijärjestys

- jos halutaan kuljettaa tietoa lehdistä juureen päin, on **jälkijärjestys** hyvä strategia, eli käsittele lapset ensin, ja vasta sitten solmu itse
- esim. puun korkeuden selvittäminen
 - solmun korkeus on sen lasten korkeuksien maksimi plus 1
 - lehtisolmujen korkeus on 0
 - algoritmi yksinkertaistuu hieman jos ajatellaan, että olemattomien alipuiden korkeus on -1

```
laske-korkeus(x)
  if x == NIL
    return -1                // olemattoman alipuun korkeus on -1
  k1 = laske-korkeus(x.left) // selvitetään vasemman lapsen korkeus
  k2 = laske-korkeus(x.right) // selvitetään oikean lapsen korkeus
  return max( k1, k2 )+1     // oma korkeus on lasten korkeuden maksimi +1
```

- eli ensin selvitetään lapsien korkeudet ja palautetaan sitten kysyjälle oma korkeus
- opeaatiolle annetaan parametriksi puun juuri:
`print "puun korkeus on"; print laske-korkeus(T.root)`

- jos halutaan kuljettaa tietoa juuresta lehtiin päin, on **esijärjestys** hyvä strategia, eli käsittele solmu ensin, ja vasta sen jälkeen lapset
- esim. halutaan muuttaa solmujen avaimiksi arvo, joka on vanha arvo $+$ niiden solmujen vanhojen avaimien arvot, jotka sijaitsevat polulla juuresta solmuun
 - kuljetetaan summaa mukana puussa ylhäältä alaspäin
 - huom: puu tuskin säilyy hakupuuna tällaisen operaation seurauksena

muuta-arvoja(x , summa)

if $x \neq \text{NIL}$

$x.\text{key} = x.\text{key} + \text{summa}$

muuta-arvoja($x.\text{left}$, $\text{summa} + x.\text{key}$)

muuta-arvoja($x.\text{right}$, $\text{summa} + x.\text{key}$)

- kutsutaan operaatiota parametrina juuri ja 0: $\text{muuta-arvoja}(T.\text{root}, 0)$
- Juuren avain säilyy entisellään, juuren lapsien avaimien arvoksi tulee juuren avaimen arvo $+$ alkuperäinen avain, jne
- joskus puun läpikäyntijärjestyksellä ei ole väliä (esim. lh5 teht 3), voidaan käydä puu läpi esi-, sisä- tai jälkijärjestyksessä
- jos puun korkeus on h , on **rekursiivisten läpikäyntialgoritmien tilavaativuus on $\mathcal{O}(h)$** , sillä rekursiivisia kutsuja on pahimmillaan suorituksen alla puun korkeuden verran ja jokainen menossa olevista kutsuista vie tilaa $\mathcal{O}(1)$:n verran

Tasapainon säilytys: AVL-puu

- Normaalin binääripuun solmussa olevien attribuuttien *key*, *left*, *right* ja *parent* lisäksi jokaisessa AVL-puun solmussa on kenttä *height*, joka ilmoittaa solmun korkeuden.
- AVL-puulta vaaditaan, että se toteuttaa seuraavan **tasapainoehdon**:
minkä tahansa solmun vasemman ja oikean alipuun korkeuksien erotus on joko -1 , 0 tai 1 .

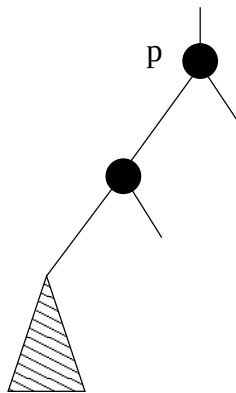
Toisin sanoen vaaditaan

$$| \text{Height}(x.\text{left}) - \text{Height}(x.\text{right}) | \leq 1 \quad \text{kaikilla solmuilla } x$$

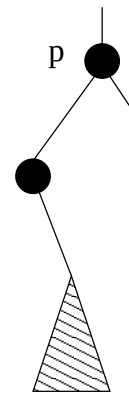
- **AVL-puun tasapainoehto on syytä hallita**
- tasapainoehto takaa, että puun korkeus on $\mathcal{O}(\log n)$ (s 172-181 todistus, jonka sisältöä ei siis tarvitse osata)
- jos tasapainoehto rikkoutuu avaimen lisäyksen tai poiston yhteydessä, se voidaan saada taas voimaan ajassa $\mathcal{O}(\log n)$ **kierto-operaatioiden** avulla
- koska operaatioiden aikavaativuus riippuu puun korkeudesta, on **AVL-puussa kaikkien joukko-operaatioiden** *search*, *insert*, *delete*, *min*, *max*, *succ* ja *pred* aikavaativuus $\mathcal{O}(\log n)$

AVL-puu

- epätasapaino korjataan vakioajassa toimivilla **kierto-operaatioilla** (s 183-207)
- Jos solmun p epätasapainon syy on sen vasemmassa alipuussa:

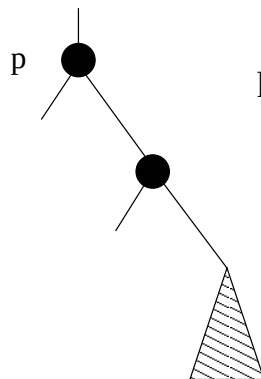


korjaava toimenpide:
 $\text{rightRotate}(p)$

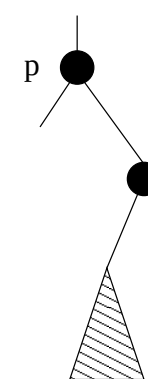


korjaava toimenpide:
 $\text{leftRightRotate}(p)$
eli
 $\text{leftRotate}(p.\text{left})$
 $\text{rightRotate}(p)$

- Jos solmun p epätasapainon syy on sen oikeassa alipuussa:



korjaava toimenpide:
 $\text{leftRotate}(p)$



korjaava toimenpide:
 $\text{rightLeftRotate}(p)$
eli
 $\text{rightRotate}(p.\text{right})$
 $\text{leftRotate}(p)$

AVL-puun tasapainon säilytys

- Kun AVL-puuhun lisätään uusi solmu, voi puu mennä epätasapainoon
- AVL-insert (s. 194-211 ja s. 215)
 - tehdään lisäys ensin kuten lisäys tehdään tasapainottamattomaan puuhun
 - polulla lisätystä juureen on saattanut mennä solmuja epätasapainoon
 - kuljetaan polku läpi ja jos törmätään epätasapainoon, tehdään tarvittavat kierto-operaatiot
 - jos kiertoja tehdään, on taattua, että kaikki solmut palaavat tasapainoon
 - matkan varrella on myös päivitettävä solmujen korkeuskentät
- operaation aikavaativuus on $\mathcal{O}(\log n)$ koska
 - normaali lisäys kulkee polun juuresta lehteen, näin löytyy lisäyskohta
 - AVL-puussa joudutaan tämän lisäksi kulkemaan polku lisätystä juureen siltä varalta, että epätasapainotilanteita syntyi
 - molempien polkujen pituus on puun korkeuden verran eli $\mathcal{O}(\log n)$ sillä puu on tasapainossa, suoritetaan siis kaksi $\mathcal{O}(\log n)$:n pituista polun läpikäyntiä
 - kierto-operaatiot ovat vakioaikaisia, eli niiden suorittaminen ei kasvata aikavaativuutta

AVL-puun tasapainon säilytys

- Kun AVL-puusta poistetaan solmu, voi puu mennä epätasapainoon
- AVL-delete (s. 216-222)
 - tehdään poisto ensin kuten poisto tehdään tasapainottamattomasta puusta
 - polulla poistetusta juureen on saattanut mennä solmuja epätasapainoon
 - kuljetaan polku läpi ja tehdään tarvittaessa kierto-operaatioita
 - kierto-operaatioita saatetaan joutua tekemään useiden solmujen kohdalla
 - matkan varrella on myös päivitettävä solmujen korkeuskentät
- operaation aikavaativuus on $\mathcal{O}(\log n)$ koska
 - normaali poisto joutuu pahimmassa tapauksessa kulkemaan polun juuresta lehteen (poiston kolmas tapaus, s. 162)
 - AVL-puussa joudutaan tämän lisäksi kulkemaan polku poistetusta juureen siltä varalta, että epätasapainotilanteita syntyi
 - kierto-operaatiot ovat vakioaikaisia, eli niiden suorittaminen ei kasvata aikavaativuutta siitäkään huolimatta, että kiertoja voidaan joutua suorittamaan puun korkeuden verran, toisin kuin AVL-insertissä jossa max 2 kiertoa riittää

Algoritmien aikavaativuusanalyysin nyrkkisäännöt

- yksinkertaisten käskyjen aikavaativuus on vakio eli $\mathcal{O}(1)$
- peräkkäin suoritettavien käskyjen aikavaativuus on sama kuin käskyistä eniten aikaavievän aikavaativuus
- ehtolauseen aikavaativuus on $\mathcal{O}(\text{ehdon testausaika} + \text{suoritetun vaihtoehdon aikavaativuus})$
- aliohjelman suorituksen aikavaativuus on $\mathcal{O}(\text{parametrien evaluointiaika} + \text{parametrien sijoitusaika} + \text{aliohjelman käskyjen suoritus aika})$
- looppeja sisältävän algoritmin aikavaativuuden arvioiminen:
 - arvioi sisimmän loopin runko-osan aikavaativuus: $\mathcal{O}(\text{runko})$
 - arvioi kuinka monta kertaa sisin looppi yhteensä suoritetaan: lkm
 - aikavaativuus on $\mathcal{O}(lkm \times \text{runko})$
- rekursiota käyttävien algoritmien aikavaativuuden arvioiminen:
 - arvioi pelkän rekursiivisen aliohjelman runko-osan aikavaativuus: $\mathcal{O}(\text{runko})$
 - arvioi kuinka monta kertaa rekursiokutsu yhteensä suoritetaan: lkm
 - aikavaativuus on $\mathcal{O}(lkm \times \text{runko})$