Koe

- Koe ma 28.2 klo 16.00-19 salissa A111, koeaika kuten tavallista 2h 30min
- **Kokeessa saa olla mukana A4:n kokoinen kaksipuolinen käsin tehty, itse kirjoitettu lunttilappu**
- Neuvontapaja (Joel ja Nyman) järjestävät TiRa-kertauksen pe klo 14.15-18 salissa D122
- Kokeeseen valmistautuessa kannattaa huomioida, että koetehtävät tulevat muistuttamaan huomattavasti enemmän normaaleja laskareita kuin pajatehtäviä
- Edellisten vuosien kokeita löytyy jonkun verran kurssien kotisivuilta, esim. kevät 2010: `http://www.cs.helsinki.fi/u/mluukkai/tirak2010/`

Tärkeää ja vähemmäntärkeää

- Ensimmäisen periodin kolme ydinteemaa teorian kannalta:
  - aika- ja tilavaativuusanalyysi
  - linkitetty lista: erikoistapaukset ja listan eri versiot
  - tasapainottomat ja tasapainoiset hakupuut
- Kokeessa on kysymyksiä kaikista ydinteemoista
- Erityisesti pajatehtävissä toistuvana teemana on ollut rekursio
  - kokeessakaan rekursiota ei ole syytä unohtaa . . .
- Ensimmäisen välikokeen kannalta ei niin tärkeää ovat
  - Java-spesifiset asiat (ArrayList, LinkedList, TreeSet, TreeMap . . .)
  - yleisen puun esitystavat (maanantain luennon kalvoissa, ei kuitenkaan)
  - invariantit ja rekursioyhtälöt

Aika- ja tilavaativuusanalyysi

- Mitä tarkoitetaan aika- ja tilavaativuudella? s. 28-32
  - Aikavaativuus: algoritmin käyttämä laskenta-"aika" syötteen koon funktiona
  - Tilavativuus: 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 tarkoista laskelmista (s. 33-34 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. 37-38
  - mitä vaativuusluokka tarkoittaa, eli $\mathcal{O}$:n matemaattinen määritelmä, s. 39-43

- Pikakertaus $\mathcal{O}$-käsitteesta:

  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

- 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 *nyrkkisääntöjen* (s. 44-59, kootusti sivulla 84) soveltaminen on avainasemassa
  - säännöt perustuvat s. 39-43 matemaattiseen kalustoon, ja
  - tuottavat saman vastauksen kuin s. 33-34 tyylinen tarpeettoman tarkka analyysi
- kolme ensimmäistä aikavaativuusanalysoijan nyrkkisääntöä
  - yksinkertaisten käskyjen (sijoitus, vertailu, tulostus, laskutoimitukset) aikavaativuus vakio eli $\mathcal{O}(1)$
  - peräkkäin suoritettavien käskyjen/ohjelmanosien aikavaativuus on sama kuin käskyistä/ohjelmanosista eniten aikaavievän aikavaativuus
  - ehtolauseen aikavaativuus on $\mathcal{O}$(ehdon testausaika + suoritetun vaihtoehdon aikavaativuus)

```
insertionSmaller(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   elsif 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. 46
- 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:

```
insertionIfNonzero(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 $\mathcal{O}$(parametrien evaluointiaika + parametrien sijoitusaika + aliohjelman käskyjen suoritusaika)

- **insertionSmaller2(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   elsif y<z
4       return = y
5   else
6       return = z
```

- aliohjelman min aikavaativuus selvästi $\mathcal{O}(1)$

- insertionSmaller2 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

bubbleSort(A)

```
1   for i = n to 2  // i:n arvo aluksi n eli taulukon pituus, i pienenee toistoissa 2:n asti
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 $A[i+1,\ldots,n]$ *järjestyksessä ja järjestetyn osan alkiot vähintään yhtä suuria kuin osan* $A[1,\ldots,i]$ *alkiot*

- loopin runko vakioaikainen sillä rungon suoritus tarkoittaa pahimmillaan neljän koodirivin suorittamista

- runko suoritetaan $1+2+3+\ldots+n=\frac{1}{2}n(n+1)$ kertaa (ks. s. 53)

- 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 $\mathcal{O}(n^2)$

- Pelkästään sisäkkäisten looppien määrä ei aina ratkaise aikavaativuutta

- Tarkastellaan seuraavaa metodia:

```
weirdSum(A)
1   sum = 0
2   for i = 1 to n-3
3       allSame = true
4       for j = 1 to 3
5           if A[i] ≠ A[j]
6               allSame = false
7       if allSame
8           sum = sum + 1
9   return sum
```

- sisempi looppi suoritetaan nyt ainoastaan kolme kertaa jokaisella i:n arvolla, eli yhteensä noin $3 \times n$ kertaa

- eli aikavaativuus sisäkkäisistä loopeista huolimatta $\mathcal{O}(n)$

- Entä peräkkäiset toistolauseet?

- Seuraavassa metodi, joka kääntää taulukon sisällön pinon s avulla:

```
ReverseArray(A)
1   Stack s
2   for i = 1 to A.length
3       Push(s, T[i])
4   for j = 1 to A.length
5       T[i] = Pop(s)
```

- pino-operaatiot vietät vakioajan, eli molempien for:ien runko vakioaikainen

- molemmat for:it suoritetaan n kertaa (n taulukon pituus)

- koska metodi koostuu kahdesta *peräkkäisest* $\mathcal{O}(n)$ aikaa vievästä osasta, sen aikavaativuus on $\mathcal{O}(n)$

- pinossa on enimmillään n alkiota joten tilavaativuus myös $\mathcal{O}(n)$

- kuudes sääntö: rekursiota käyttävien algoritmien aikavaativuuden arvioiminen etenee hyvin samaan tyyliin kuin looppeja sisältävien algoritmien analyysi
  - arvioi pelkän rekursiivisen aliohjelman runko-osan aikavaativuus: $\mathcal{O}(runko)$
  - arvioi kuinka monta kertaa rekursiokutsu yhteensä suoritetaan: lkm
  - aikavaativuus on $\mathcal{O}(lkm \times runko)$

- esim. tulostetaan taulukon sisältö käänteisesti rekursiivisen metodin avulla

```
recPrint(A,i)
1   if i > A.length
2       return
3   recPrint(A,i+1)
4   println( A[i] )
```

- 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. 59
  - rekursion syvyys määrittää tilavaativuuden

- Tässä esitetyillä nyrkkisäännöillä pärjää aika pitkälle TiRa:n ensimmäisessä välikokeessa

- Monisteen s. 60-62 ja 68-70 mainitut rekursioyhtälöt voi ensimmäisessä välikokeessa unohtaa, asiaan palataan seuraavassa periodissa

- Monisteen s. 72-81 "vaativuusanalyysiä kansantajuisesti" kannattaa lukea

Linkitetty lista

- **linkitetyt rakenteet (s. 105-126) 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 linkitetyt
  - alkiot järjestyksessä vai ei?
- listan "erikoistapaukset": pino ja jono
  - materiaalissa myös taulukkoon perustuvat toteutukset (s. 91-104). ne ovat ok, mutta TiRa:n kannalta kovaa ydintä ovat nimenomaan linkitetyt 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)

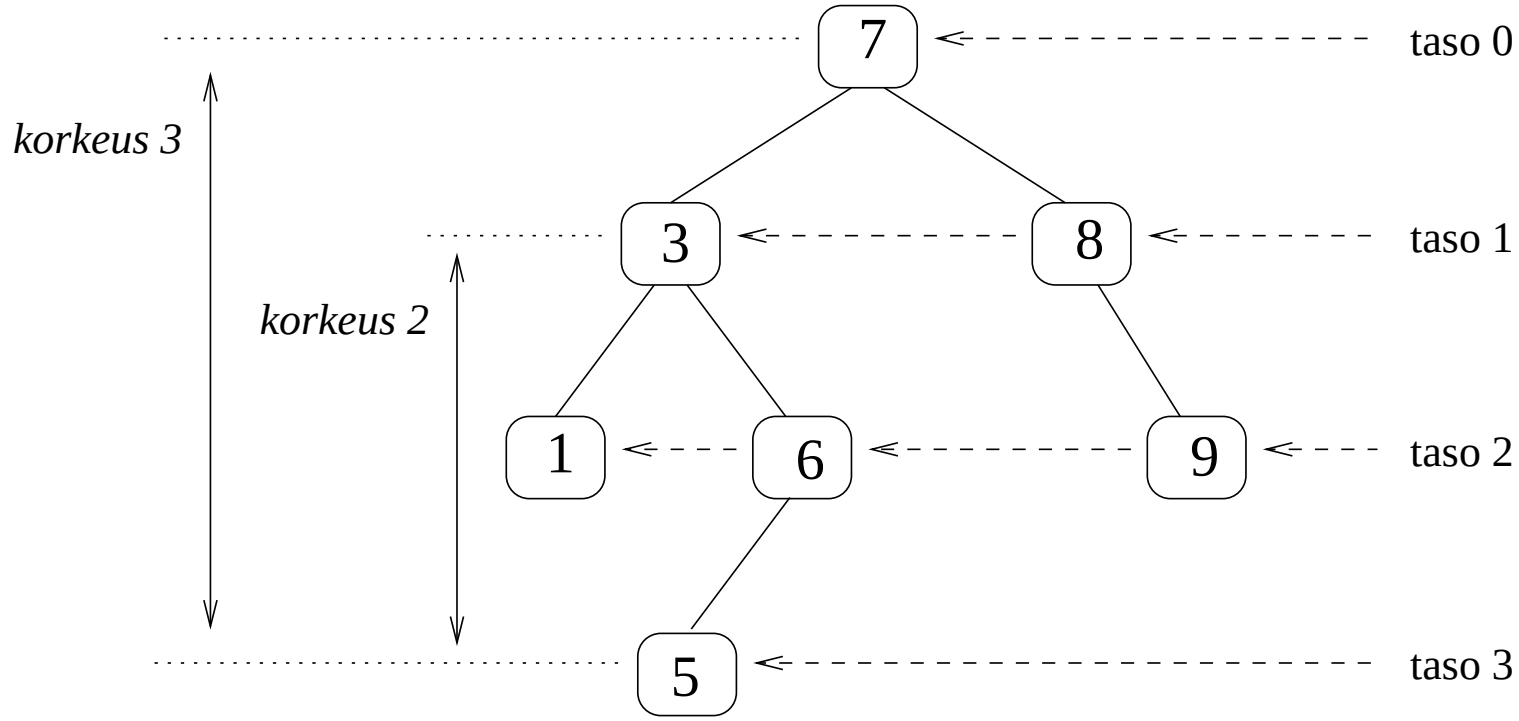
- myös pino ja jono ovat tarpeellisia osia tietyissä algoritmeissa (esim. puun leveyssuuntainen läpikäynti, sulutuksen tarkastus, . . .)

- On paljon listaa tehokkaampia tapoja toteuttaa tietotyyppi joukko

	taulukko	järj.taulukko	lista	järj.lista	tasap.puu	haj.taul
search	O(n)	O(log n)	O(n)	O(n)	O(log n)	O(1)
insert	O(1)	O(n)	O(1)	O(n)	O(log n)	O(1)
delete	O(n)	O(n)	O(1)	O(1)	O(log n)	O(1)
min	O(n)	O(1)	O(n)	O(1)	O(log n)	O(n)
max	O(n)	O(1)	O(n)	O(1)	O(log n)	O(n)
succ	O(n)	O(1)	O(n)	O(1)	O(log n)	O(n)
pred	O(n)	O(1)	O(n)	O(1)	O(log n)	O(n)

Tasapainottamattomat ja tasapainoiset binäärihakupuut

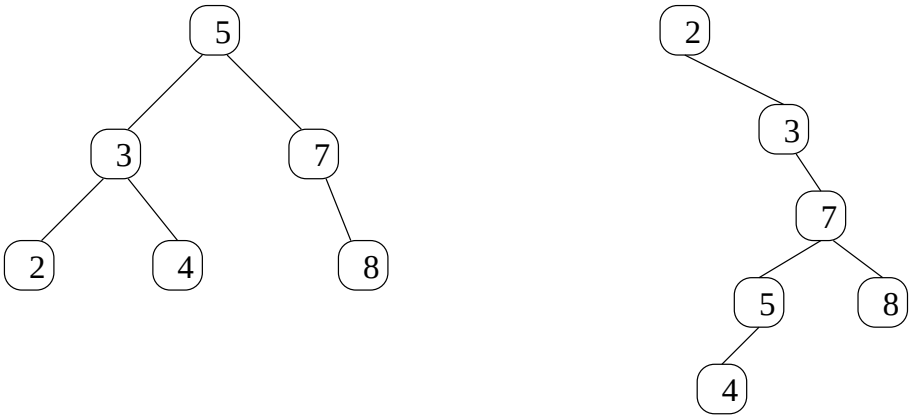
- tärkeitä käsitteitä
  - solmu, juuri, lehti, vanhempi, vasen lapsi, oikea lapsi, vasen alipuu, oikea alipuu
  - korkeus ja tasot:



  - 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

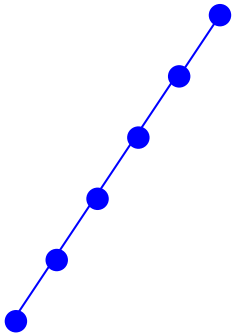
- operaatioiden search, insert, delete, min, succ, max ja pred toimintaperiaate (varsinkin 3 ensimmäistä) syytä hallita (s. 157-177)
  - operaatioiden toimintaperiaate ja niiden yhteys binäärihakupuuehtoon
  - operaatioiden aikavaativuus ja tilavaativuus
- esim. operaatio insert
  - ensin etsitään paikka mihin lisättävä avain väistämättömästi binäärihakupuuehdon 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
```

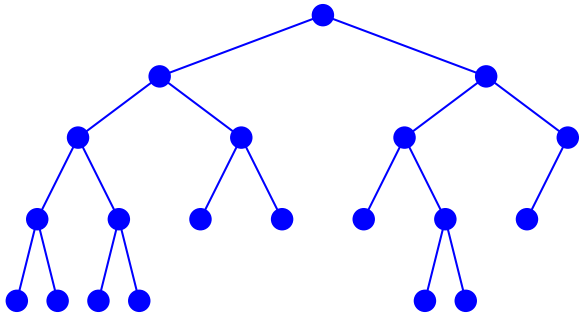
- huomataan (s. 157-177), että kaikkien muidenkin operaatioiden vaativuus riippuu suoraan puun korkeudesta h, eli **puun korkeus on kriittinen asia tehokkuuden kannalta**

puun korkeus vs. solmumäärä

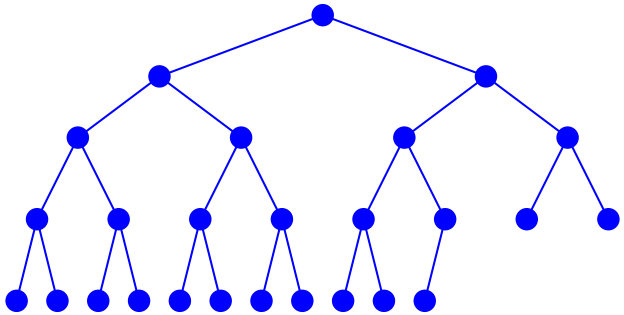
- puita on hyviä ja huonoja



lineaarinen



tasapainoinen?



melkein täydellinen

- **tärkeitä tuloksia**: jos puussa n solmua, niin
  - tasapainoittamattoman puun korkeus parhaimmillaan $\mathcal{O}(\log n)$ mutta pahimmilaan $\mathcal{O}(n)$ (s. 142-147)
  - AVL-puun korkeus parhaimmillaan $\mathcal{O}(\log n)$ ja myös pahimmillaan $\mathcal{O}(\log n)$ (s. 185-190)
- tulokset ja niiden seuraukset on pakko kaikkien tietää ja on hyvä ymmärtää vähintään maalaisjärjen tasolla mistä on kysymys (esim. s. 144 ja 146 kuvat)
- AVL-puun korkeuden ylärajan matemaattista todistusta ei tarvitse kokeessa osata

puun solmujen läpikäynti: esi-, sisä- ja jälkijärjestys

- joukko-operaatioiden lisäksi on erittäin tärkeää osata soveltaa puiden solmujen läpikäyntejä
- 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 ≠ 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. 151-156)

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 sisäsolmujen (eli kaikkien paitsi lehtisolmujen) lukumäärän selvittäminen

```
sisasolmut(x)
    if x == NIL
        return 0
    if x.left == NIL and  x.right == NIL
        return 0
    return 1 + sisasolmut(x.left) + sisasolmut(x.right)
```

- olemattoman solmun ja lehtisolmun (ei lapsia) tapauksessa metodi palauttaa 0

- muussa tapauksessa selvitetään lapsista alkavien alipuiden sisäsolmujen lukumäärä, lisätään siihen yksi ja palautetaan näin saatu summa kutsujalle

- opeaatiolle annetaan parametriksi puun juuri:
print "sisäsolmuja "+ sisasolmut(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 avaimeksi 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 tälläisen operaation seurauksena

```
muuta-arvoja(x, summa)
    if x ≠ NIL
        x.key = x.key+summa
        muuta-arvoja(x.left, x.key)
        muuta-arvoja(x.right, x.key)
```

  - kutsutaan operaatiota parametrina juuri ja 0: muuta-arvoja(T.root, 0)
  - Juuren avain säilyy entisellään, juuren lapsien avaimien arvoksi tulee juuren avaimen arvo + alkuperäinen avain, jne

- joskus puun läpikäyntijärjestyksellä ei ole väliä, 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

  $$|\,\textbf{Height(x.left)} - \textbf{Height(x.right)}\,| \leq 1 \qquad \textbf{kaikilla solmuilla x}$$

- **AVL-puun tasapainoehto on syytä hallita**

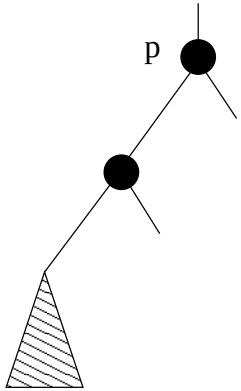
- tasapainoehto takaa, että puun korkeus on $\mathcal{O}(\log n)$ (s 185-190 todistus, jonka sisältöä ei siis tarivse osata)

- jos tasapainoehto rikkoutuu avaimen lisäyksen tai poiston yhteydessä, se voidaan saada taas voimaan ajassa $\mathcal{O}(\log n)$ kierto-operaatioiden avulla

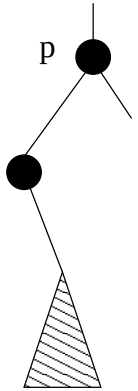
- koska operaatioiden aikavaativuus riippuu puun korkeudesta, on AVL-puussa kaikkien joukko-operaatioiden search, insert, delete, min, max, succ ja pred aikavaativuus $\mathcal{O}(\log n)$

AVL-puu

- epätasapaino korjataan vakioajassa toimivilla **kierto-operaatioilla** (s 194-218)
- Jos solmun p epätasapainon syy on sen vasemmassa alipuussa:

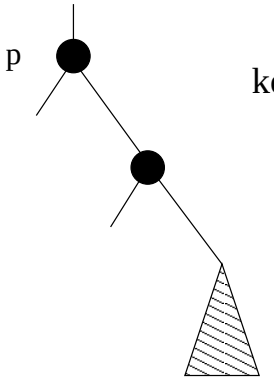


korjaava toimenpide:
rightRotate(p)

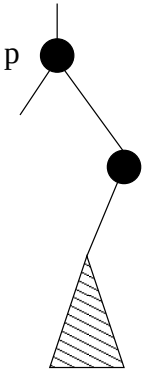


korjaava toimenpide:
leftRightRotate(p)
eli
leftRotate(p.left)
rightRotate(p)

- Jos solmun p epätasapainon syy on sen oikeassa alipuussa:



korjaava toimenpide:
leftRotate(p)



korjaava toimenpide:
rightLeftRotate(p)
eli
rightRotate(p.right)
leftRotate(p)

AVL-puun tasapainon säilytys

- Kun AVL-puuhun lisätään uusi solmu, voi puu mennä epätasapainoon
- AVL-insert (s. 217-220 ja 224)
  - 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. 225-231)
  - 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. 172)
  - AVL-puussa joudutaan tämän lisäksi kulkemaan polku poistetusta juureen siltä varalta, että epätasapainotilanteita syntyi
  - kierto-operaatiot ovat vakioaikaisia, eli niiden suorittaminen ei kasvata aikavaativuutta siitäkän huolimatta, että kiertoja voidaan joutua suorittamaan puun korkeuden verran, toisin kuin AVL-insertissä jossa max 2 kiertoa riittää

Entäs rekursio?

- Puiden rekursiivisten läpikäyntien soveltaminen syytä osata
- Rekursiivisten algoritmien toiminta hyvä ymmärtää ja niiden aika- ja tilavaativuuksia pitää pystyä analysoimaan
- Rekursio ei kuitenkaan ole kokeessa pääosassa, joten ei syytä panikointiin

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 $\mathcal{O}$(ehdon testausaika + suoritetun vaihtoehdon aikavaativuus)
- aliohjelman suorituksen aikavaativuus on $\mathcal{O}$(parametrien evaluointiaika + parametrien sijoitusaika + aliohjelman käskyjen suoritusaika)
- 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 $\mathcal{O}(lkm \times runko)$
- rekursiota käyttävien algoritmien aikavaativuuden arvioiminen:
  - arvioi pelkän rekursiivisen aliohjelman runko-osan aikavaativuus: $\mathcal{O}(runko)$
  - arvioi kuinka monta kertaa rekursiokutsu yhteensä suoritetaan: lkm
  - aikavaativuus on $\mathcal{O}(lkm \times runko)$