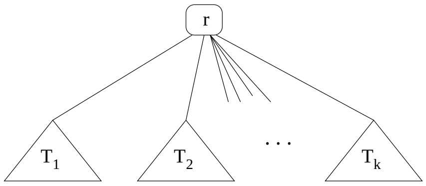
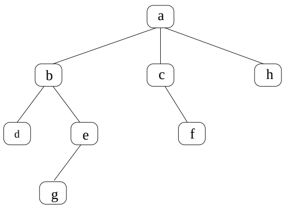
3. Hakupuut

- Hakupuu on listaa huomattavasti edistyneempi tapa toteuttaa abstrakti tietotyyppi joukko
- puurakenteelle on tietojenkäsittelyssä myös muutakin käyttöä, esim. algoritmin suoritusajan analysoinnissa, ohjelman laskennan etenemisen kuvailussa ym.
- ennen kuin menemme hakupuihin, tutustutaan yleiseen puihin liittyvään käsitteistöön
- *Puu* on kokoelma *solmuja* ja niitä yhdistäviä *kaaria*, siten että:
  - kokoelma on tyhjä, tai
  - yksi solmuista r, on *juuri* johon kaaret liittävät nolla tai useampia *alipuita* $T_1, \ldots T_k$, jotka itsekin ovat puita



  - Puiden $T_1, \ldots, T_k$ juuret ovat r:n *lapsia* ja r on lastensa *vanhempi*
  - jokaisella solmulla paitsi juurella on tasan yksi vanhempi, tästä seuraa että n solmuisessa puussa on täsmälleen $n-1$ kaarta

  - solmu jolla ei ole lapsia on *lehti*
  - solmut joilla on yhteinen vanhempi, ovat *sisaruksia*
  - *isovanhempi*, *lapsenlapsi*, *setä* ja *serkku* määräytyvät luonnollisella tavalla
- esim:



  - puun juuri a, jolla lapset b, c ja h
  - puun lehtiä ovat solmut d, g, f ja h
  - d, e ja f ovat a:n lapsenlapsia, ja a on d, e ja f isovanhempi
  - d ja e ovat sisaruksia joiden serkku on f
  - b on f:n setä
- lisää määritelmiä:
  - *polku* solmusta x_1 solmuun x_k on jono solmuja $x_1, x_2, \ldots, x_k$ siten että x_i on x_{i+1}:n vanhempi kun $1 \leq i \leq k-1$, polun pituus on sen särmien lukumäärä

    esim: a, b, e, g on polku solmusta a solmuun g jonka pituus on 3

    huom: erikoistapaus, jokaisesta solmusta on nollan pituinen polku itseensä!

  - jos on olemassa polku solmusta x_1 solmuun x_2, sanotaan että x_1 on x_2, *edeltäjä* ja x_2 on x_1:n *jälkeläinen*

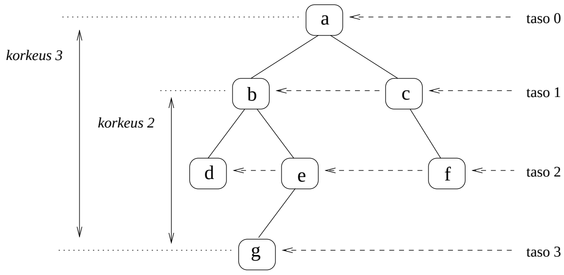
    jos $x_1 \neq x_2$ ovat edeltäjyys ja jälkeläisyys *aitoja*

    esim. a, b ja e ovat solmun e edeltäjät, joista a ja b ovat aitoja edeltäjiä

    solmun e seuraajia ovat e ja g, joista jälkimmäinen on aito seuraaja
  - solmun x *taso* (engl. depth) on polun pituus juuresta x:ään, juuren taso on 0

    joissain lähteissä tasosta käytetään termiä *syvyys*
  - solmun x *korkeus* on pisimmän x:stä lehteen vievän polun pituus

    puun korkeus on sen juuren korkeus
- esim: kuvassa solmujen tasot sekä solmun a ja b korkeus



Binääripuu

- jos puun solmuilla on korkeintaan kaksi lasta, on kyseessä *binääripuu*
  - binääripuun alipuiden juuria kutsutaan *vasemmaksi* ja *oikeaksi* lapseksi, solmun x vasempaan lapseen viitataan left[x] ja oikeaan right[x]
  - solmusta left[x] alkavaa puuta kutsutaan solmun x *vasemmaksi alipuuksi* ja solmusta right[x] alkavaa x:n *oikeaksi alipuuksi*
  - ed. kalvolla oleva puu on binääripuu, esim solmun b vasen lapsi on d ja oikea lapsi e, eli left[b] $= d$ ja right[b] $= e$
- **Lause 1**:

  Olkoon T binääripuu jonka korkeus on k. Tällöin

  (1) puun tasolla i on enintään 2^i solmua, $i \geq 0$

  (2) puussa T on enintään $2^{k+1} - 1$ solmua

  todistus

  Väite 1:

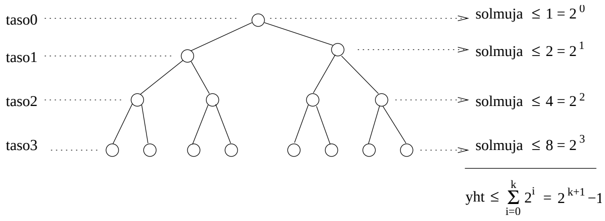
  Induktiolla tason korkeuden suhteen suhteen. Tasolla 0 vain juurisolmu, ja $2^0 = 1$ joten kaava voimassa tasolla 1.

  Oletetaan että väite pätee tasolla $n-1$ ja osoitetaan että se pätee myös tasolla n. Tasolla $n-1$ siis enintään 2^{n-1} solmua. Jokaisella näistä enintään kaksi lasta, siis tason n lapsimäärä on enintään $2 \times 2^{n-1} = 2^n$. Kaava siis voimassa myös tasolla n.

todistus jatkuu ...

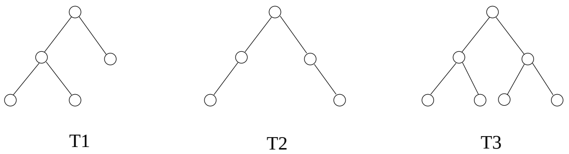
Väite 2:

binääripuun solmujen lukumäärä on summa eri tason solmujen lukumärästä, eli edellisen kohdan perusteella enintään $\sum_{i=0}^{k} 2^i = \frac{1-2^{k+1}}{1-2} = 2^{k+1} - 1$, missä käytimme geometrisen sarjan summakaavaa.

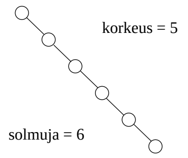


- muutama termi
  - binääripuu on *aito* jos jokaisella solmulla joko 0 tai 2 lasta
  - binääripuu on *täysi* jos sen jokainen lehti on samalla tasolla
  - jos binääripuu on aito ja täysi, sanotaan että se on täydellinen

- esim:



  - T1 on aito muttei täysi,
  - T2 on täysi muttei aito
  - T3 on aito ja täysi, eli *täydellinen*
- lausetta 1 soveltamalla helppo huomata että k:n korkuisella täydellisellä binääripuulla on täsmälleen 2^k lasta ja $2^{k+1} - 1$ solmua
- toisaalta binääripuussa jonka korkeus on k voi olla vähimmillään $k + 1$ solmua:



- lausutaan seuraavassa vielä täsmällisesti mikä on puun korkeus suhteessa solmujen lukumäärään
- **Lause 2**:

  Olkoon T binääripuu jonka solmujen lukumäärä on n. Tällöin

  (1) puun korkeus on enintään $n - 1$

  (2) puun korkeus on vähintään $\Omega(log\ n)$

  todistus

  Väite 1:

  jos jokaisella ei-lapsisolmulla on ainoastaan yksi lapsi, on puu kuin lista ja korkeus silloin suurin mahdollinen eli $n - 1$

  Väite 2:

  täydellisen binääripuun tapauksessa puun korkeus on pienimmillään, tällöin lauseen 1 nojalla $n = 2^{k+1} - 1$, eli $n + 1 = 2^{k+1}$ otetaan kaksikantainen logaritmi molemmilta puolilta $log_2\ (n + 1) = k + 1$ eli $k = log_2\ (n + 1) - 1 = \Omega(log_2 n)$
- huom: koska kyse puun korkeuden alarajasta, käytimme asymptoottisen alarajan merkintää Ω merkinnän $\mathcal{O}$ sijasta

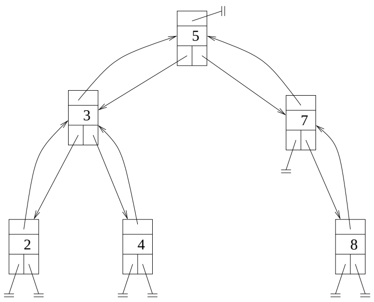
2.1 Binäärihakupuut

- toteutetaan sivun 22 abstrakti tietotyyppi joukko siten että joukossa olevat alkiot talletetaan binääripuun solmuihin
- rajoitumme jälleen yksinkertaistettuun tapaukseen missä talletettavat tietoalkiot sisältävät ainoastaan avaimen
- puu rakentuu puusolmu-tietueista, joilla seuraavat kentät:

key	talletettu tietoalkio
left	viite vasempaan lapseen
right	viite oikeaan lapseen
p	viite vanhempaan

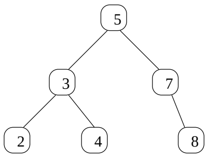
- puulla T attribuutti $root[T]$ joka osoittaa juurisolmuun
- jos solmulla x ei ole vasenta lasta, $left[x] = NIL$, vastaavasti oikealle lapselle
- puun juurisolmulla parent-kentän arvo NIL
- puun solmuille on voimassa *binäärihakupuuehto:* jos x on hakupuun solmu niin
  - x:n mille tahansa vasemman alipuun solmulle y: $key[y] \leq key[x]$
  - solmun x:n mille tahansa oikean alipuun solmulle z: $key[x] \leq key[z]$

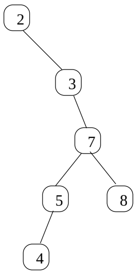
- esim: hakupuu missä alkiot 2, 3, 4, 5 ,7 ja 8



- yleensä piirrämme puut ilman linkkikenttien eksplisiittistä sisältöä

  oikealla samat alkiot sisältävä erimuotoinen binäärihakupuu





- puun T alkiot voidaan tulostaa suuruusjärjestyksessä kutsumalla seuraavaksi esitettävää rekursiivista algoritmia parametrilla $root[T]$

```
inorder-tree-walk(x)
    if x ≠ NIL then
        inorder-tree-walk(left[x])
        print key[x]
        inorder-tree-walk(right[x])
```

- algoritmin toiminnan eteneminen syötteenä kuvan vasen puu

```
inorder-tree-walk(5)
    inorder-tree-walk(3)
        inorder-tree-walk(2)
            inorder-tree-walk(NIL)
            print(2)                        2
            inorder-tree-walk(NIL)
        print(3)                            3
        inorder-tree-walk(4)
            inorder-tree-walk(NIL)
            print(4)                        4
            inorder-tree-walk(NIL)
    print(5)                                5
    inorder-tree-walk(7)
        inorder-tree-walk(NIL)
        print(7)                            7
        inorder-tree-walk(8)
            inorder-tree-walk(NIL)
```



```
            print(8)                        8
            inorder-tree-walk(NIL)
```

edellä esim. kutsu inorder-tree-walk(5) tarkoittaa että proseduuria kutsutaan *viitteenään* solmu joka sisältää avaimenaan numeron 5

- algoritmi käy puun läpi *sisäjärjestyksessä*
  - tultaessa solmuun x ensin käsitellään vasen lapsi left[x] sitten itse solmu x ja tämän jälkeen oikea lapsi right[x]
  - vasenta lasta left[x] käsitellessä tulostetaan kaikki siitä alkavan alipuun alkiot ja binäärihakupuuehdon perusteella näiden arvo on korkeintaan sama kuin key[x]:n arvo
  - oikeaa lasta right[x] käsitellessä tulostetaan kaikki siitä alkavan alipuun alkiot ja binäärihakupuuehdon perusteella näiden arvo on vähintään yhtä suuri kuin key[x]:n arvo
  - arvo key[x] tulostetaan siis oikeassa kohdassa
  - samanlaisen päättelyn perusteella jokainen alkio tulostetaan oikeassa kohdassa
- jos puussa on n solmua, algoritmin aikavaativuus on $\mathcal{O}(n)$ sillä jokaisessa solmussa käydään tasan kolme kertaa, tultaessa vanhemmasta rekursiokutsulla sekä palattaessa vasemman ja oikean alipuun rekursiosta
- algoritmin tilavaativuus
  - algoritmin edetessä rekursiopinoon on talletuttuna reitti juurisolmusta tarkasteltavaan solmuun

  - algoritmin suoritusaikana rekursiopinossa on siis tietoa "piilossa"
  - rekursiopinon koko on pahimmillaan sama kuin puun korkeus h, eli algoritmin *tilavaativuus* on $\mathcal{O}(h)$
  - Lauseen 2 perusteella puun korkeus pahimmassa tapauksessa $n-1$, missä n solmujen lukumäärä, eli
  - tilavaativuus on siis pahimmassa tapauksessa $\mathcal{O}(n)$

joukko-operaatioiden toteuttaminen

- avaimen k etsiminen puusta tapahtuu kutsumalla rekursiivista operaatiota search(root[T],k)

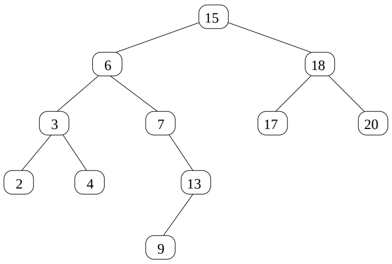
```
search(x, k)
    if x = NIL or key[x] = k
        then return x
    if k < key[x]
        then return search(left[x],k)
        else return search(right[x],k)
```

- etsintä siis etenee juuresta alaspäin, pahimmassa tapauksessa puun korkeuden verran eli aikavaativuus $\mathcal{O}(h)$ missä h puun korkeus

  Lauseen 2 perusteella puun korkeus pahimmassa tapauksessa $n-1$, missä n solmujen lukumäärä, eli

  searchin pahin tapaus vie ajan $\mathcal{O}(n)$
- rekursiopinon korkeus pahimmillaan sama kuin puun korkeus, eli myös tilavaativuus $\mathcal{O}(n)$

- esim:



search(root[T],13) etenee polkua 15 → 6 → 7 → 13
search(root[T], 10) etenee 15 → 6 → 7 → 13 → 9 → *NIL*

- avaimen etsiminen on helppo toteuttaa myös ilman rekursiota

```
search(x, k)
    while x ≠ NIL and key[x] ≠ k do
        if k < key[x]
            then x ←left[x]
            else x ←right[x]
    return x
```

- aikavaativuus on edelleen $\mathcal{O}(n)$ mutta koska rekursiopinosta on päästy eroon, tilavaativuus onkin $\mathcal{O}(1)$

- binäärihakupuuehdosta seuraa suoraan että kulkemalla mahdollisimman paljon vasemmalle, päädymme pienimpään puussa olevaan alkioon

  seuraavassa operaatio mikä palauttaa viitteen solmun x alipuun pienimpään alkioon

```
min(x)
    while left[x] ≠ NIL do
        x ←left[x]
    return x
```

- vastaavasti, maksimialkio löytyy menemällä oikealle niin kauan kuin mahdollista:

```
max(x)
    while right[x] ≠ NIL do
        x ←right[x]
    return x
```

- kuten ei-rekursiivisella search:illa, sekä min että max -operaatioiden pahimman tapauksen aikavaativuus on $\mathcal{O}(h)$ ja tilavaativuus $\mathcal{O}(1)$
- kalvon 58 esimerkissä minimin haku etenee polkua 15 → 6 → 3 → 2 ja maksimin polkua 15 → 18 → 20

- operaation succ toteutus:

```
succ(x)
    if right[x] ≠ NIL
        then return min(right[x])
    y ←p[x]
    while y ≠ NIL and x = right[y] do
        x ←y
        y ←p[x]
    return y
```

- algoritmin toiminta jakautuu kahteen eri tapaukseen
  - jos solmun x oikea alipuu on epätyhjä, on solmun seuraaja oikean alipuun pienin alkio

    esim. kalvon 58 kuvassa solmun 15 seuraaja on oikean alipuun minimi, eli solmu 17
  - jos oikea alipuu on tyhjä, solmun x seuraaja on esi-isä joka löytyy siten että palaan puussa kohti juurta niin kauan kunnes tehdään yksi paluuaskel "yläviistoon oikealle"

    esim. kalvon 58 kuvassa solmun 13 seuraaja on 15 mikä löytyy palaamalla kolme askelta ylöspäin 13 → 7 → 6 → 15 ja askeleista viimeinen siis on "yläviistoon oikealle"

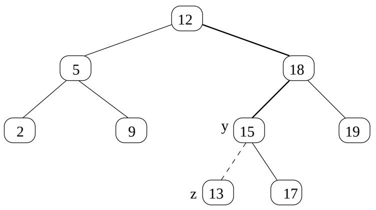
    esim. solmun 17 seuraaja on 18 koska heti ensimmäinen askel juureen päin on yläoikealle
- succ-operaation pahin tapaus vie aikaa $\mathcal{O}(h)$, voidaan joutua menemään juuresta aina koko puun korkeuden verran alas tai palata lehdestä aina juureen asti, tilavaativuus on $\mathcal{O}(1)$
- operaatio pred on symmetrinen succ-operaation kanssa, ja eli vaatii pahimmassa tapauksessa ajan $\mathcal{O}(h)$ ja tilan $\mathcal{O}(1)$

- alkion lisääminen binäärihakupuuhun käy melko helposti

```
insert(T, k)
1  z ←new puusolmu
2  key[z] ←k
3  left[z] ←NIL
4  right[z] ←NIL
5  x ←root[T]
6  y ←NIL
7  while x ≠ NIL do
8      y ←x
9      if k < key[x]
10         then x ←left[x]
11         then x ←right[x]
12 p[z] ←y
13 if y= NIL
14     then root[T] ←z
15     else if k < key[y]
16         then left[y] ←z
17         else right[y] ←z
```

- toimintaperiaate
  - rivien 7-11 while-toistolause etsii uudelle alkiolle paikan
  - toistolauseen jälkeen y osoittaa alkioon jonka lapseksi uusi alkio lisätään
  - rivi 13 huomioi erikoistilanteen missä lisättävä alkio on puun ensimmäinen
  - riveillä 15-17 uusi alkio laitetaan y:n joko vasemmaksi tai oikeaksi alkioksi

- puu mihin lisätty avain 13, polku juuresta solmuun jonka lapseksi uusi solmu lisätään on tummennettu

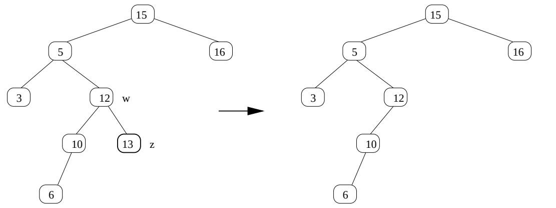


- kuten muillakin tähän asti kohtaamillamme operaatioilla, on lisäyksenkin aikavaativuus $\mathcal{O}(h)$ sillä pahimmassa tapauksessa lisäys tehdään alimmalla tasolla olevan solmun lapseksi
- tilavaativuus on $\mathcal{O}(1)$, sillä rekursio ei ole käytössä

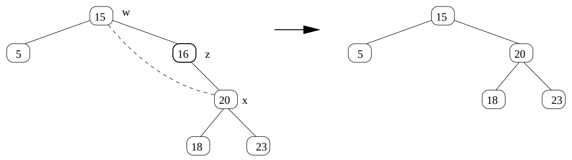
- poisto on puu-operaatioista monimutkaisin, argumenttina operaatiolla on viite poistettavaan solmuun z

delete(T, z)
1 **if** left[z] = NIL and right[z] = NIL
2   w ←p[z]
3   **if** w = NIL **then** root[T] = NIL
4   **else if** z = left[w]
5   **then** left[w] ←NIL
6   **else** right[w] ←NIL
7   **return** z
8 **if** left[z] = NIL or right[z] = NIL
9   **if** left[z] ≠ NIL **then** x ←left[z] **else** x ←right[z]
10   w ←p[z]
11   **if** w = NIL **then** root[T] = x
12   **else if** z = left[w]
13   **then** left[w] ←x
14   **else** right[w] ←x
15   p[x] ←w
16   **return** z
17 y ←succ(z)
18 x ←right[y]
19 w ←p[y]
20 **if** y = left[w]
21   **then** left[w] ←x
22   **else** right[w] ←x
23 p[x] ←w
24 key[z] ←key[y]
25 **return** y

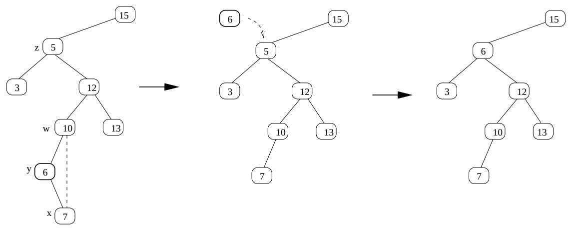
- algoritmilla kolme erillistä tapausta
- tapaus 1: *poistettavalla ei lapsia*, rivit 1-7
  - poistetaan solmu z
  - huomioidaan rivillä 7 erikoistapaus missä poistettava on juuri



- tapaus 2: *poistettavalla yksi lapsi*, rivit 8-16
  - korvataan poistettava z ainoalla lapsellaan x
  - huomioidaan rivillä 11 erikoistapaus missä poistettava on juuri



- tapaus 3: *poistettavalla kaksi lasta*, rivit 17-25
  - vaihdetaan poistettava solmun z *avain* sisäjärjestyksessä seuraavan solmun y avaimeen
  - solmu y mistä korvaava avain löytyi, poistetaan korvaamalla se lapsellaan x



- lopuksi palautetaan viite puusta poistettuun solmuun, näin sen varaama muistitila voidaan tarvittaessa vapauttaa
- poiston aikavaativuus selvästi $\mathcal{O}(h)$, kuljetaan yksi polku juuresta mahdollisesti lehtisolmuun ja kussakin solmussa suoritetaan vakiomäärä operaatioita, tilavaativuus $\mathcal{O}(1)$
- Huom: operaatio delete(T,z) siis poistaa puusta solmun z *sisällön*, tapauksessa 3 solmun z muistialue jää vielä käyttöön sisältäen kuitenkin toisen avaimen
- Cormenissa poisto on esitetty hieman eri tavalla, koodissa eri tapaukset ovat lomittuneet toisiinsa, koodi on lyhempi mutta vaikeampi ymmärtää

- jokaisen operaatioista search, insert, delete, max, min, succ ja pred pahimman tapauksen aikavaativuus oli siis $\mathcal{O}(h)$, missä h on puun korkeus
- kaikki operaatiot pystyttiin tekemään siten että tilavaativuus on vain $\mathcal{O}(1)$
- Lauseen 2 perusteella n solmuisen puun korkeus h vaihtelee välillä $log_2\,(n+1) - 1 \leq h \leq n - 1$
- eli jos puu on *tasapainoinen* on operaatioiden aikavaativuus $\mathcal{O}(log\ n)$ ja puu on oleellisesti listaa parempi joukon toteutuksessa
- kun taas hyvin epätasapainoisessa puussa operaatioiden vaativuus on $\mathcal{O}(n)$ ja puu on siis jopa listaa huonompi tapa abstraktin tietotyypin joukko toteuttamiseen
- huono puu syntyy esim. lisäämällä puuhun solmut $1, \ldots, n$ suuruusjärjestyksessä: insert(1), insert(2), ..., insert(n) tai käänteisessä järjestyksessä insert(n), insert(n-1), ..., insert(1)

  toisaalta huono puu voi syntyä myös patologisen insert/delete suoritusyhdistelmän takia

- ongelma: *miten voisimme taata että puu pysyy tasapainoisena*