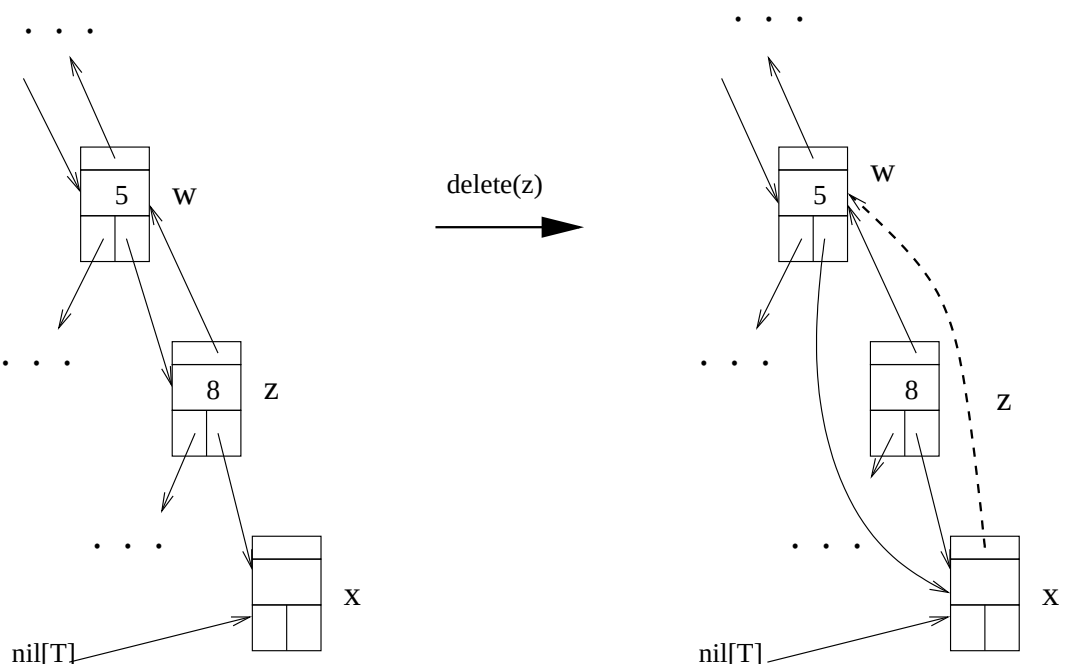
Alkion poisto

- aluksi uusi solmu poistetaan samoin kuin lisäys tehtäisiin normaaliin binäärihakupuuhun
- tämän jälkeen kutsutaan apurutiinia rb-delete-fixup joka korjaa mahdollisen punamustaehdon rikkoutumisen

```
delete(T, z)
1   if left[z] = nil[T] or right[z] = nil[T]
2       if left[z] ≠ nil[T] then x ←left[z] else x ←right[z]
3       w ←p[z]
4       if w = nil[T] then root[T] = x
5       else if z = left[w]
6               then left[w] ←x
7               else right[w] ←x
8       p[x] ←w
9       if color[z] = BLACK then rb-delete-fixup(T,x)
10      return z
11  y ←succ(z)
12  x ←right[y]
13  w ←p[y]
14  if y = left[w]
15      then left[w] ←x
16      else right[w] ←x
17  p[x] ←w
18  key[z] ←key[y]
19  if color[y] = BLACK then rb-delete-fixup(T,x)
20  return y
```

- poisto-operaatiossa on muutama ero aiemmin (sivulla 61) esitettyyn normaalista hakupuusta tehtävään poistoon
- tapaukset missä poistettavalla on nolla tai yksi lapsi on nyt sulautettu yhteen, riveille 1-10
- NIL esiintymät korvattu nil[T]:llä
- jos poistettava *solmu* on musta, kutsutaan lopussa korjausapurutiinia poistettavan lapselle x
- huom: vaikka poistettavan solmun lapsi x olisi NIL, asetetaan lapsen x parent-kentäksi joka tapauksessa poistettavan solmun vanhempi w:

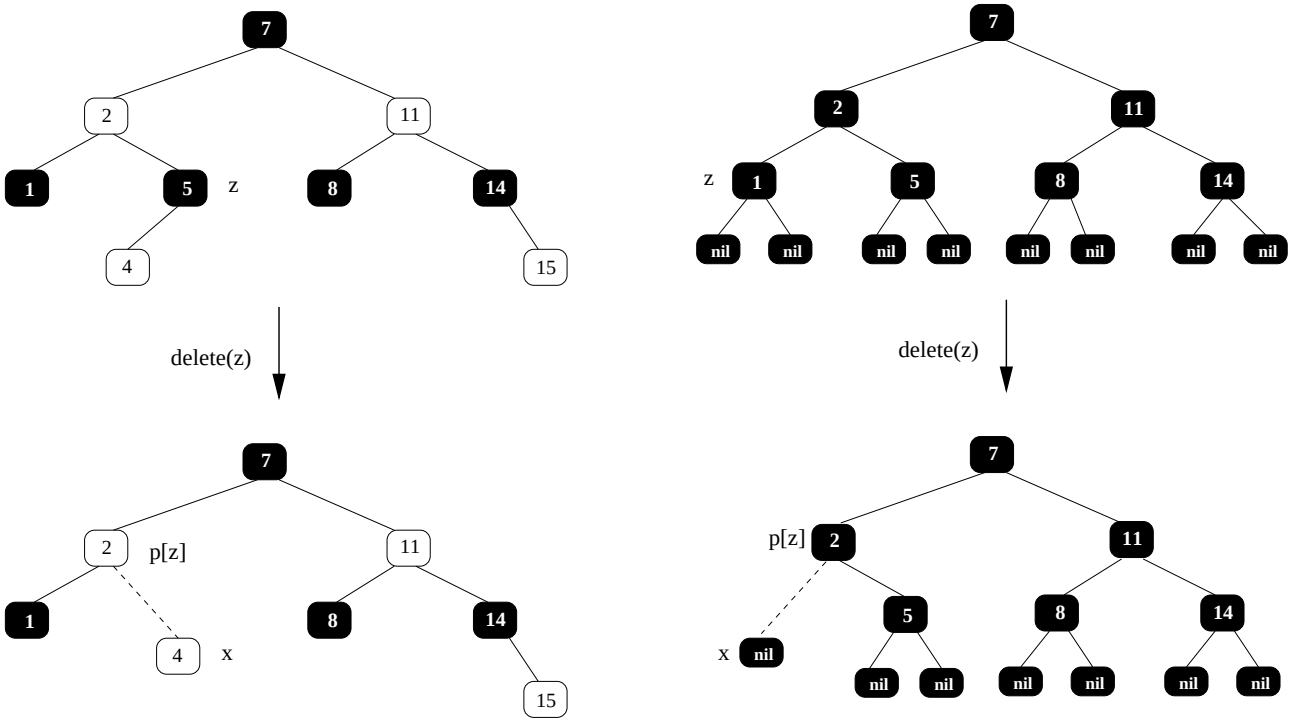


- näin korjausoperaatio yksinkertaistuu hieman, ja tämä onkin perimmäinen syy sille miksi käytämme eksplisiittistä NIL solmua punamustan puun toteutuksessa

- korjausapurutiini

```
rb-delete-fixup(T, x)
1   while x ≠ root[T] and color[x] = BLACK do
2       if x = left[p[x]] then
3           w ←right[p[x]]
4           if color[w] = RED then
5               color[w] ←BLACK              ▷tapaus 1
6               color[p[x]] ←RED             ▷tapaus 1
7               left-rotate(T,p[x])          ▷tapaus 1
8               w ←right[p[x]]               ▷tapaus 1
9           if color[left[w]] = BLACK and
                    color[right[w]]= BLACK then
10              color[w] ←READ               ▷tapaus 2
11              x ←p[x]                      ▷tapaus 2
12          else if color[right[w]]= BLACK then
13                  color[left[w]] ←BLACK    ▷tapaus 3
14                  color[w] ←RED            ▷tapaus 3
15                  right-rotate(T,w)        ▷tapaus 3
16                  w ←right[p[x]]           ▷tapaus 3
17              color[w] ←color[p[x]]        ▷tapaus 4
18              color[p[x]] ←BLACK           ▷tapaus 4
19              color[right[w]] ←BLACK       ▷tapaus 4
20              left-rotate(T,p[x])          ▷tapaus 4
21              x ←root[T]                   ▷tapaus 4
22-41   else ▷tapaukset missä x=right[p[x]] sivulla 93
42  color[x] ←BLACK
```

- apurutiinin koodi näyttää jopa monimutkaisemmalta kuin insertin yhteydessä suoritetun apurutiinin koodi, käydään rutiini läpi kohta kohdalta
- jos poistettava solmu on punainen, ei mikään punamustaehdoista rikkoudu, joten rutiinia ei siinä tapauksessa kutsuta
- jos poistettava on musta syntyy ongelmia:

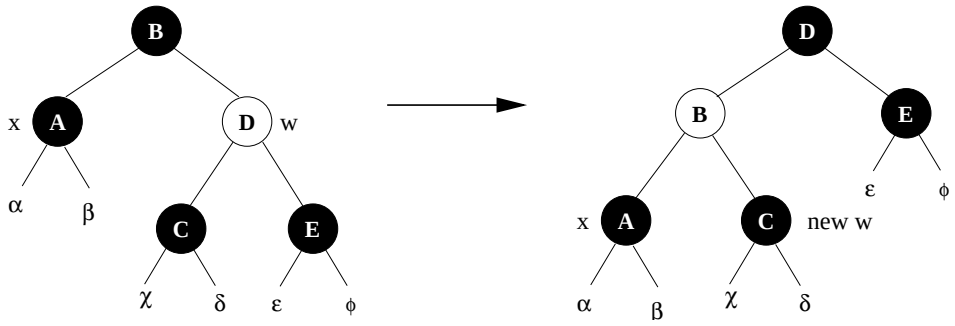


- koska yksi musta poistuu, niin poistetun solmun esi-isien osalta punamustaehto 5 *jokainen polku tietystä solmusta sen lehtiin sisältää saman määrän mustia solmuja* ei ole enää voimassa
- punamustaehdon 5 rikkoontuminen voidaan paikata sopimalla että poistettavan lapsi x sisältää ylimääräisen mustan
- sovitaan siis että poistettavan lapsi x on
  - puna-musta (kuvan vasemman puolen tapaus), tai

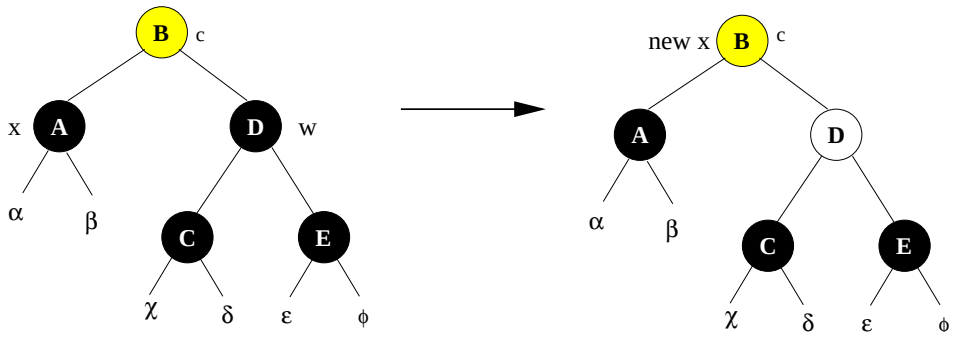
  - tupla-musta (kuvan oikean puolen tapaus)

- näin punamustaehto 5 on kunnossa, mutta ehto 1 *jokainen solmu on joko punainen tai musta* rikkoontuu!
- korjausapurutiinissa x osoittaa solmuun millä on ylimääräinen musta
  - jos x on puna-musta solmu (color[x] = RED), värjätään x mustaksi ja ongelma korjattu (kuvan vasen tapaus)
  - jos x on juuri, voidaan ylimääräinen musta poistaa
  - jos x osoittaa tuplamustaan (kuvan oikea tapaus) mennään while-lauseeseen suorittamaan joitain kiertoja ja uudelleenvärjäyksiä kunnes ongelma korjaantuu
- while-lause koostuu kahdeksasta tapauksesta joista 1-4 ovat symmetrisiä tapauksille 5-8, erottavana tekijänä se onko x vasen vaiko oikea lapsi
- w osoittaa aina ongelmasolmun x veljeen, w ei voi olla nil[T] sillä muuten x:n vanhemman punamustaehto 5 ei toteutuisi
- tapausten 1-8 transformaatioiden idea on pitää mustien solmujen lukumäärä polulla juuresta alipuiden $\alpha, \beta, \ldots$ vakiona, eli punamustaominaisuus 5 pidetään koko ajan voimassa
- tarkastellaan ensin tapauksia 1-4
- tapaus 1 on voimassa kun sisar w on punainen, tapauksissa 2-4 w on musta ja tapaukset eroavat toisistaan w:n lasten värityksen perusteella

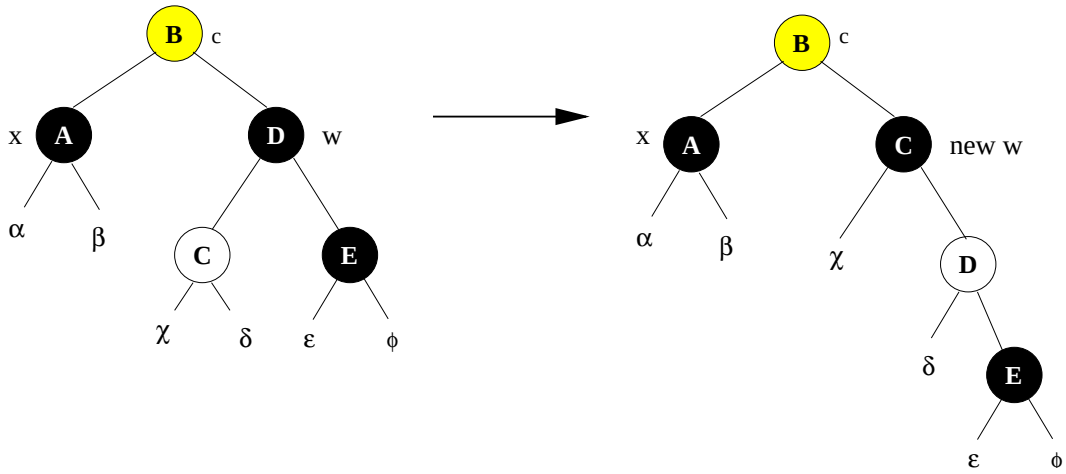
- tapaus 1: *sisar punainen*
  - w:n lasten täytyy olla mustia, joten värjätään w mustaksi ja x:n isä eli $p[x]$ punaiseksi
  - suoritetaan vasen kierto solmun $p[x]$ suhteen
  - punamusta-ominaisuudet säilyvät, ja uusi sisarsolmu on musta, joten tapaus 1 muuttuu joksikin tapauksista 2-4



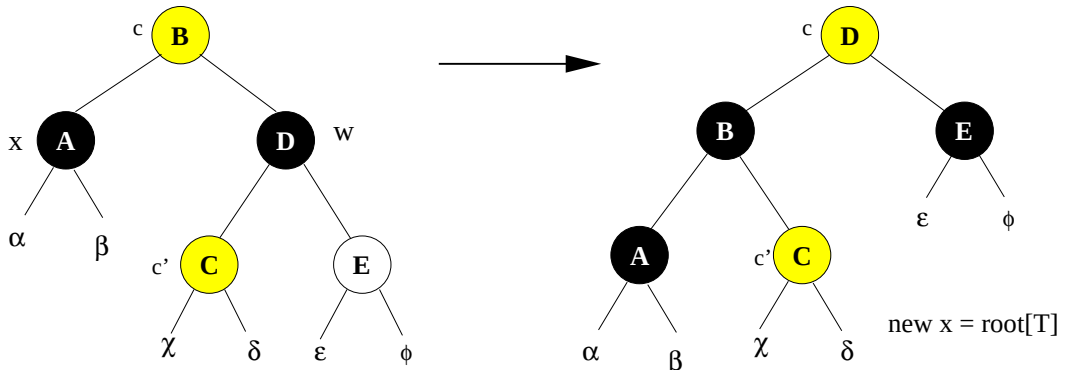
- tapaus 2: *sisaren molemmat lapset ovat mustia*
  - otamme yhden mustan pois sekä x:stä että w:stä ja lisäämme yhden mustan isäsolmuun
  - näin sisaresta tulee punainen, x:stä musta ja isäsolmusta uusi ylimääräisen mustan omaava solmu, iteraatio jatkuu while-lauseen alusta
  - huom: isäsolmu (kuvassa harmaa) voi olla joko musta tai punainen, merkitään sen väriä c:llä



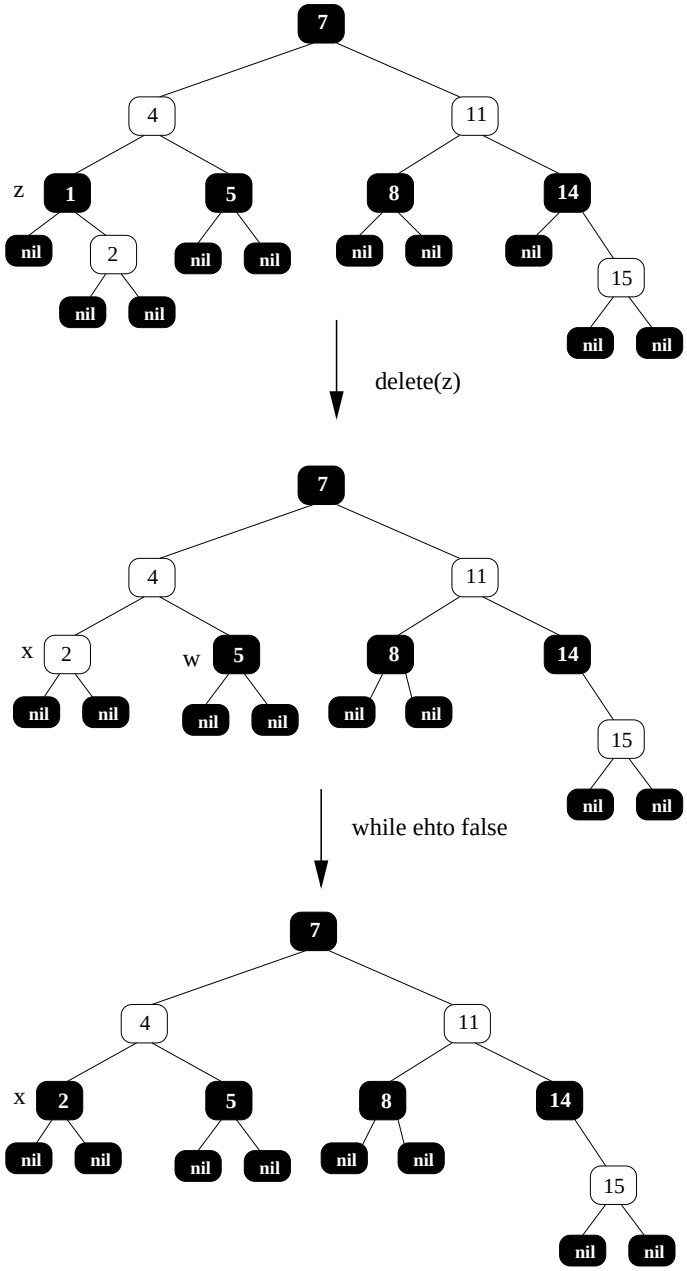
- tapaus 3: *sisaren vasen lapsi punainen ja oikea lapsi musta*
  - vaihdetaan sisaren ja sen vasemman lapsen värejä
  - kierretään sisaren suhteen oikealle
  - näin tapaus 3 on muutettu tapaukseksi 4



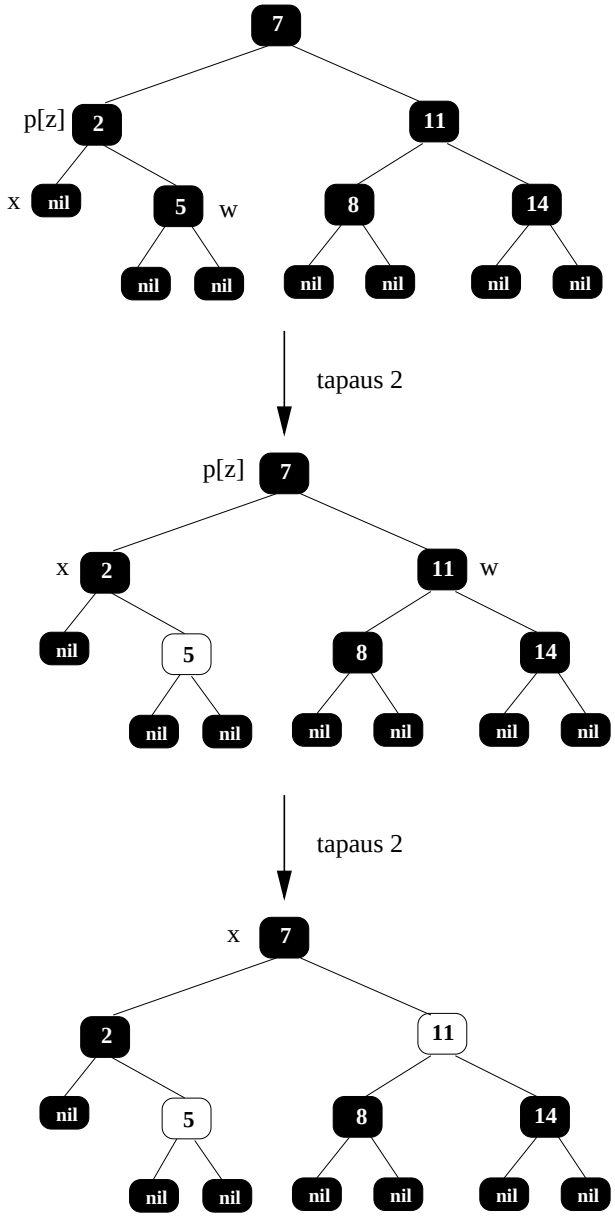
- tapaus 4: *sisaren oikea lapsi punainen*
  - tekemällä muutamia värien vaihtoja ja vasen kierto isäsolmun suhteen saadaan ylimääräinen musta poistettua rikkomatta punamustaehtoja
  - asetetaan x = root[T] joten while-silmukkaa ei enää toisteta



- esimerkki: poistetaan musta mutta while-toistoa ei tarvita

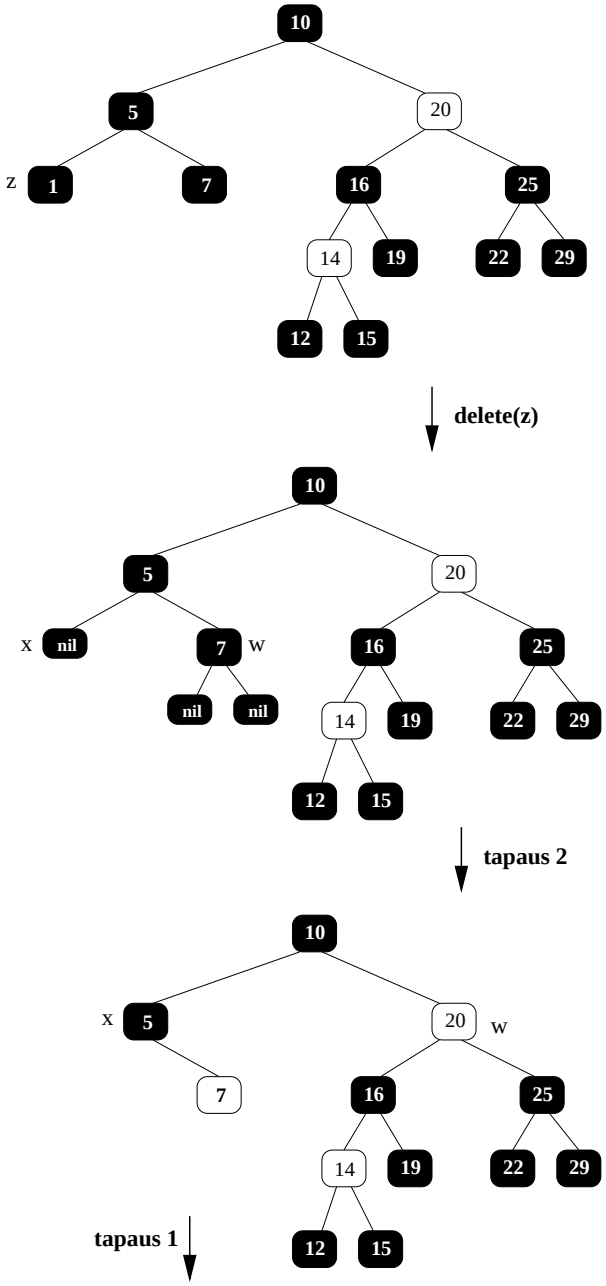


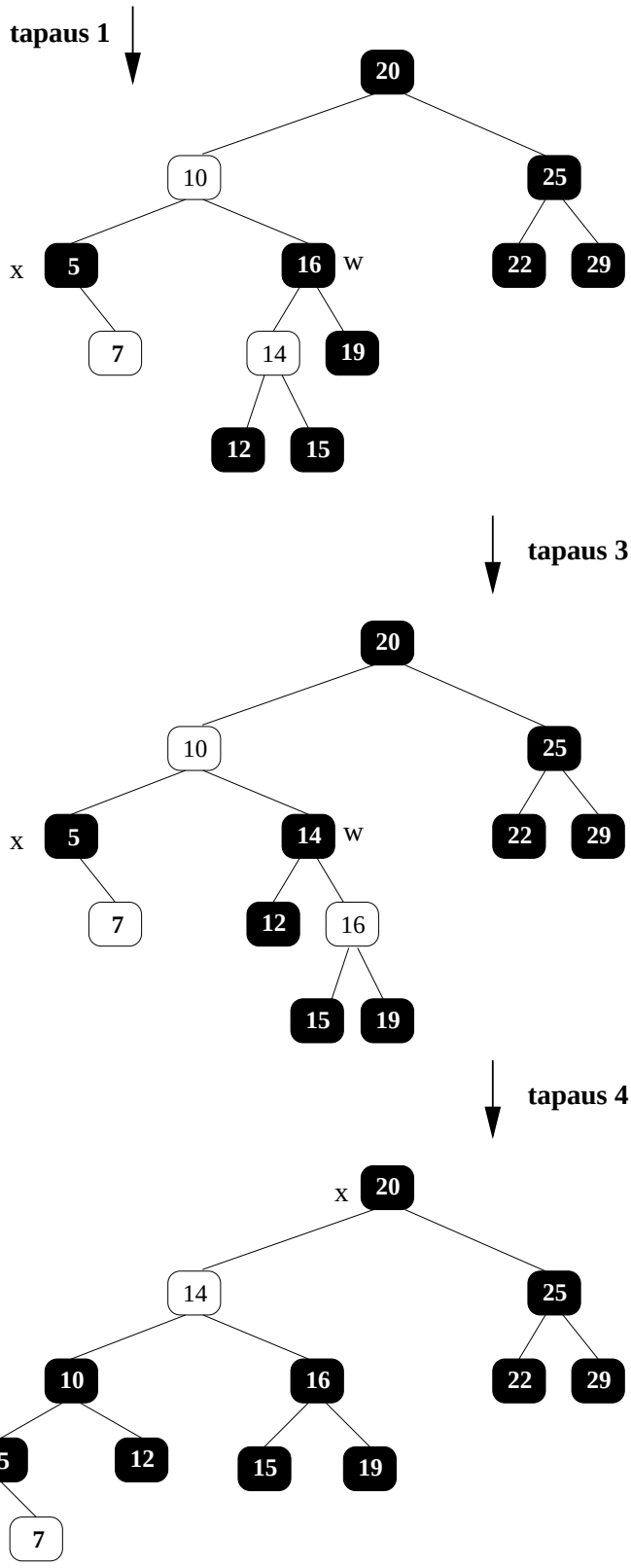
- esimerkki 2: sivun 85 kuvan oikea tapaus



- tapaus 2 toistuu kahdesti, tämän jälkeen x osoittaa juurta ylimääräinen musta voidaan poistaa

- esimerkki 3: poisto aiheuttaa kaikkien neljän korjaustoimenpiteen suorittamisen

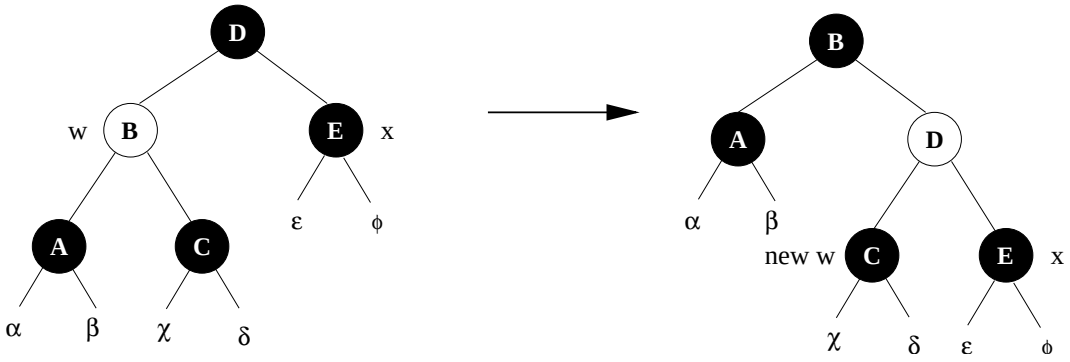



tapaus 1
20
10
25
x
5
16
w
22
29
7
14
19
12
15
tapaus 3
20
10
25
x
5
14
w
22
29
7
12
16
15
19
tapaus 4
x
20
14
25
10
16
22
29
5
12
15
19
7

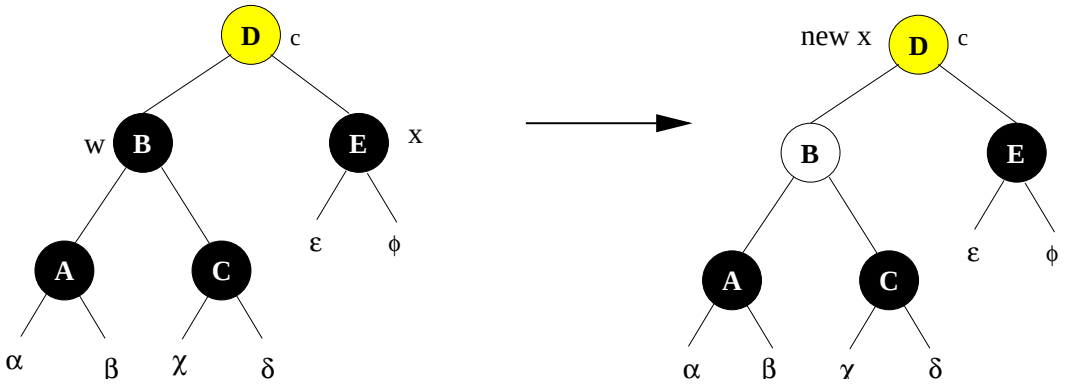
- else-haaran koodi

```
rb-delete-fixup(T, x)
1   while x ≠ root[T] and color[x] = BLACK do
2-21    if x = left[p[x]] then ▷sivulla 84
22      else
23          w ←left[p[x]]
24          if color[w] = RED then
25              color[w] ←BLACK              ▷tapaus 5
26              color[p[x]] ←RED             ▷tapaus 5
27              right-rotate(T,p[x])         ▷tapaus 5
28              w ←left[p[x]]                ▷tapaus 5
29          if color[left[w]] = BLACK and
                    color[right[w]]= BLACK then
30              color[w] ←READ               ▷tapaus 6
31              x ←p[x]                      ▷tapaus 6
32          else if color[left[w]]= BLACK then
33                  color[right[w]] ←BLACK▷tapaus 7
34                  color[w] ←RED            ▷tapaus 7
35                  left-rotate(T,w)         ▷tapaus 7
36                  w ←left[p[x]]            ▷tapaus 7
37              color[w] ←color[p[x]]        ▷tapaus 8
38              color[p[x]] ←BLACK           ▷tapaus 8
39              color[left[w]] ←BLACK        ▷tapaus 8
40              right-rotate(T,p[x])         ▷tapaus 8
41              x ←root[T]                   ▷tapaus 8
42  color[x] ←BLACK
```

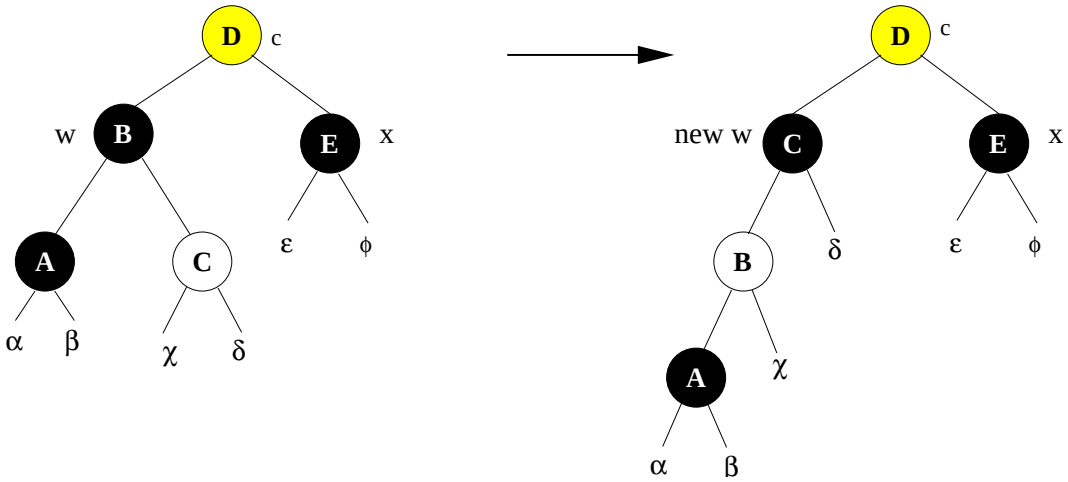
- tapaus 5: *sisar punainen*
  - w on punainen, eli w:n lasten täytyy olla mustia
  - värjätään w mustaksi ja x:n isä $p[x]$ punaiseksi
  - suoritetaan oikea kierto solmun $p[x]$ suhteen
  - punamusta-ominaisuudet säilyvät, ja uusi sisarsolmu on musta, joten tapaus 1 muuttuu joksikin tapauksista 6-8



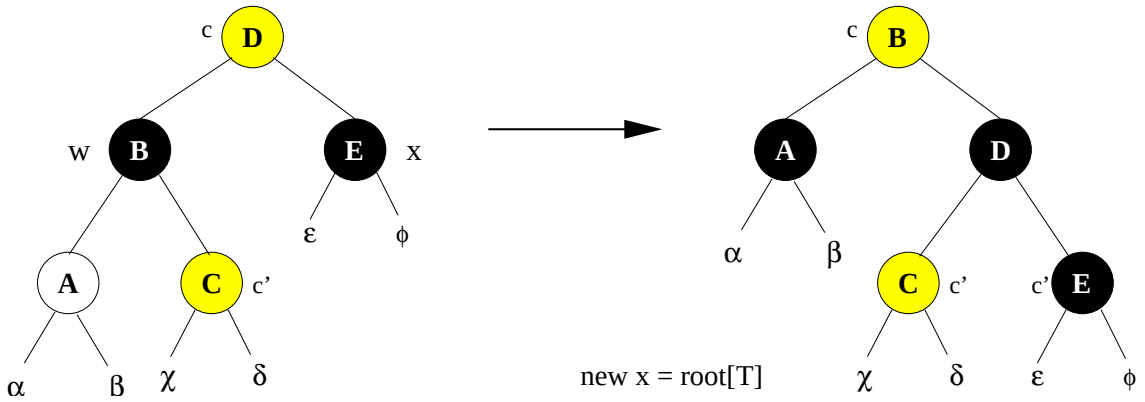
- tapaus 6: *sisaren molemmat lapset ovat mustia*
  - otamme yhden mustan pois sekä x:stä että w:stä ja lisäämme yhden mustan isäsolmuun
  - näin sisaresta tulee punainen, x:stä musta ja isäsolmusta uusi ylimääräisen mustan omaava solmu, iteraatio jatkuu while-lauseen alusta



- tapaus 7: *sisaren vasen lapsi musta ja oikea lapsi punainen*
  - vaihdetaan sisaren ja sen oikean lapsen värejä
  - kierretään sisaren suhteen vasemmalle
  - näin tapaus 3 on muutettu tapaukseksi 4



- tapaus 8: *sisaren oikea lapsi punainen*
  - tekemällä muutamia värien vaihtoja ja oikea kierto isäsolmun suhteen saadaan ylimääräinen musta poistettua rikkomatta punamustaehtoja
  - asetetaan x = root[T] joten while-silmukkaa ei enää toisteta



- tapauksissa 1, 3, 4, 5 ja 8 suoritetaan ainoastaan vakiomäärä kiertoja sekä uudelleen värityksiä
- while-silmukka toistuu vain tapauksien 2 ja 6 kohdalla, tällöin seuraavalla kerralla aloitetaan lähempänä juurta
- rb-delete-fixup operaation aikavaativuus siis pahimmassa tapauksessa on sama kuin puun korkeus, eli $\mathcal{O}(log\ n)$, tilaa operaatio ei vie kuin vakiomäärän
- korjausoperaatio ei siis huononna poisto-operaation vaativuutta, eli delete-operaation aikavaativuus punamustassa puussa on $\mathcal{O}(log\ n)$ ja tilavaativuus $\mathcal{O}(1)$

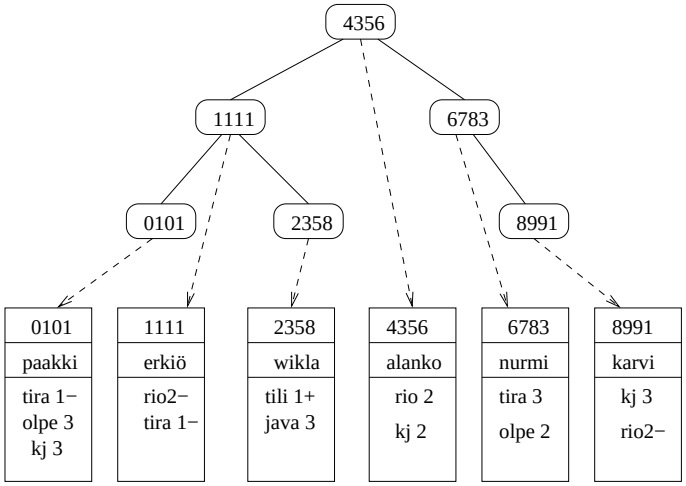
- sivun 22 joukko-tietotyypin kaikki operaatiot voidaan siis punamustassa puussa suorittaa ajassa $\mathcal{O}(log\ n)$ ja tilassa $\mathcal{O}(1)$, missä n on joukkoon talletettujen alkioiden lukumäärä

3.3 Monta indeksiä samaan dataan

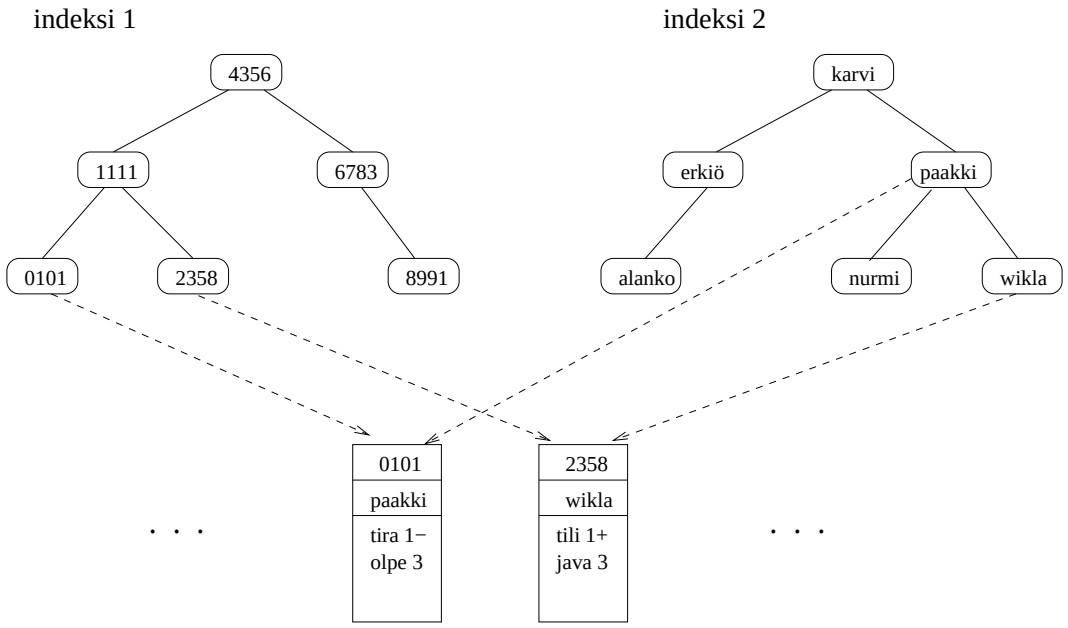
- Olemme käsitelleet yksinkertaistettua tilannetta missä puun solmuihin ei ole talletettu muuta kuin avaimen arvo
- jos organisoitavaa dataa on paljon, paras ratkaisu on tallettaa puusolmuihin avaimen lisäksi viite muut datakentät sisältävään muistialueeseen
- eli puusolmu muodostuu tällöin kentistä:

key	talletettu avain
data	viite avaimeen liittyvään tietoon
left	viite vasempaan lapseen
right	viite oikeaan lapseen
p	viite vanhempaan

- näin puu toimii *indeksi-rakenteena* minkä avulla talletettuun tietoon on mahdollista tehdä nopeita hakuja avaimen perusteella
- opiskelijarekisteri jossa indeksirakenne opiskelijanumeron suhteen:



- nyt siis opiskelijan tietojen haku opiskelijanumeron perusteella on nopeaa
- entä jos haluamme nopeat haut myös nimen perusteella?
- lisätään samalle datalle toinen indeksirakenne joka mahdollistaa nopeat haut nimeen perustuen



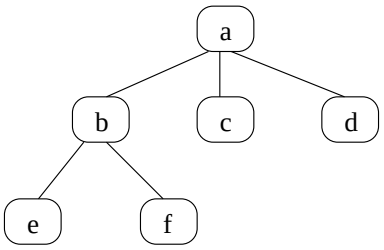
- jos talletettavaa tietoa on paljon voi se olla talletettuna myös massamuistiin

3.4 Yleisen puun talletus ja läpikäynti

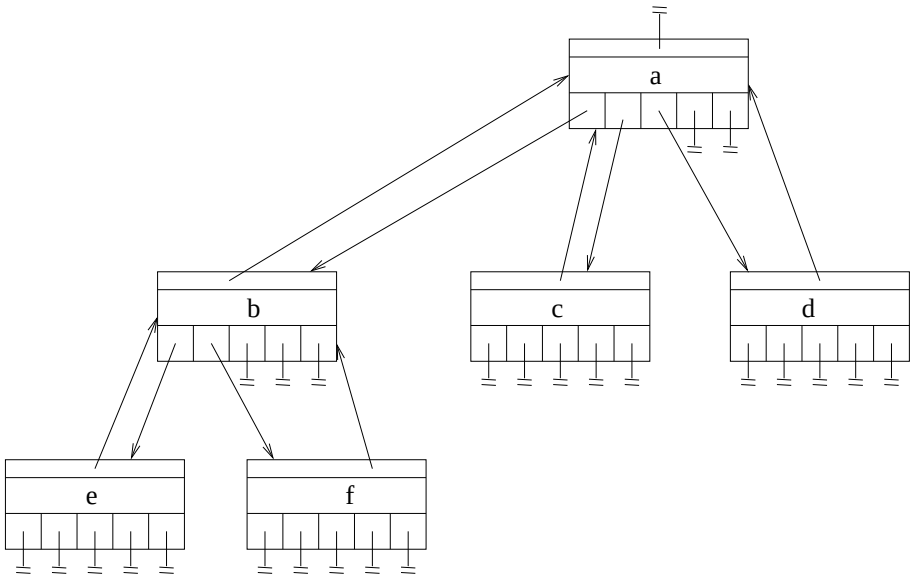
- kuten jo puu-luvun alussa mainittiin, on puille monenlaista käyttöä tietojenkäsittelyssä
- eivätkä kaikki puut suinkaan ole binääripuita
- miten siis voimme tallettaa yleisen puun
- jos tiedämme mikä on solmun maksimi haarautumisaste, voimme tallettaa solmuun viitteet kaikkiin mahdollisiin lapsiin
- eli puusolmu muodostuu tällöin kentistä:

key	talletettu avain
c1	viite 1. lapseen
c2	viite 2. lapseen
...	
ck	viite k:nteen lapseen
p	viite vanhempaan

- esim



- puu talletettuna käyttäen puusolmuja joissa haarautumisaste on 5

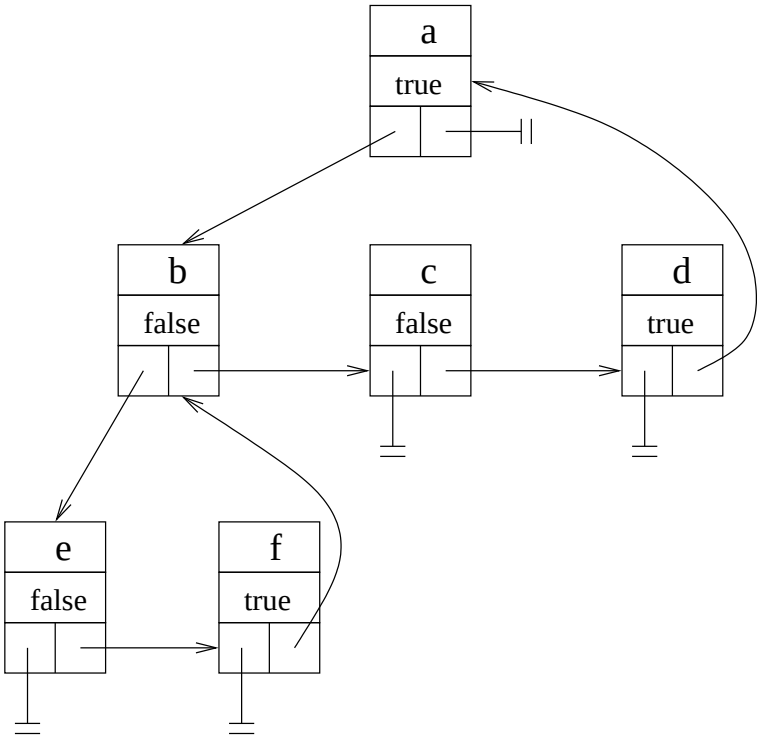


- huomaamme että rakenne tuhlaa paljon muistia tarpeettomiin linkkikenttiin
- toisaalta voi käydä myös niin että johonkin solmuun tulisikin enemmän lapsia kuin 5

- parempi ratkaisu yleisen puun tallettamiseen onkin seuraava
- puusolmun kentät

key	talletettu avain
last	bitti jonka arvo on true jos kyseessä on sisaruksista viimeinen
child	viite 1. lapseen
next	viite seuraavaan sisarukseen (jos last=false), tai vanhempaan (jos last=true)

- esimerkkipuumme talletettaisiin seuraavasti:



- muistia ei tuhlaudu turhiin linkkikenttiin ja toisaalta puun haarautumisaste ei ole rajoitettu
- puussa liikkuminen:
  - solmusta x päästään vanhempaan kulkemalla next-linkkejä kunnes on ohitettu sisarus jolla last=true

    parent(x)
    y ←next[x]
    while last[x]=false **do**
    x ←y
    y ←next[y]
    return y
  - seuraavan operaation avulla saadaan lueteltua solmun x lapset

nextchild(x,y)
    if y=NIL **then return** child[x]
    else return next[y]

– viite ensimmäiseen x:n lapseen saadaan kutsumalla nextchild(x,NIL)

– viite x:n lasta y seuraavaan lapseen saadaan kutsumalla nextchild(x,y)

– jos seuraavaa lasta ei ole, palauttaa operaatio NIL

- binäärihakupuun yhteydessä saimme tulostettua puun solmut suuruusjärjestyksessä käymällä puun läpi *sisäjärjestyksessä*, eli ensin vasen lapsi, sitten solmu itse ja lopulta oikea lapsi
- yleisten puiden kohdalla mielekkäät läpikäyntitavat ovat *esijärjestys* ja *jälkijärjestys*

- *esijärjestyksessä* käsittelemme ensin solmun ja tämän jälkeen lapset
- esimerkkipuumme solmut esijärjestyksessä lueteltuna: a, b, e, f, c, d

- algoritmina

```
preorder-tree-walk(x)
    print key[x]
    y ←nextchild(x,NIL)
    while ( y ≠ NIL )
        preorder-tree-walk(y)
        y ←nextchild(x,y)
```

- kutsu preorder-tree-walk(root[T]) tulostaa nyt puun sisällön esijärjestyksessä, huom operaatio ei toimi tyhjälle puulle!

- *jälkijärjestyksessä* käsittelemme ensin lapset ja tämän jälkeen solmun itsensä
- esimerkkipuumme solmut jälkijärjestyksessä lueteltuna: e, f, b, c, d, a
- algoritmina

```
postorder-tree-walk(x)
    y ←nextchild(x,NIL)
    while ( y ≠ NIL )
        postorder-tree-walk(y)
        y ←nextchild(x,y)
    print key[x]
```

- toki muitakin tapoja puun läpikäynnille on, esim. *leveyssuuntainen läpikäynti* missä puun alkiot käydään läpi taso kerrallaan, alkaen juuresta,

  esimerkkipuumme solmut leveyssuuntaisesti lueteltuna: a, b, c, d, e, f