12 Binäärihakupuu

http://imgur.com/L77FY5X

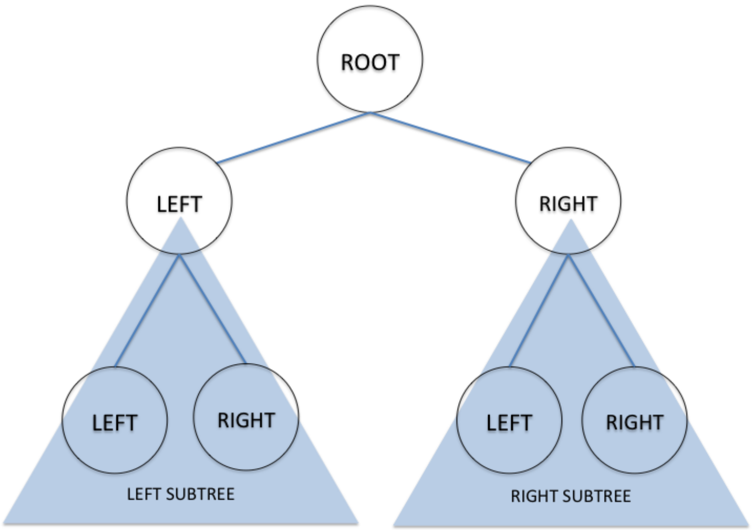
Tässä luvussa käsitellään erilaisia yleisiä puurakenteita.

- Ensin opitaan, millainen rakenne on binäärihakupuu,
- ja tasapainotetaan binäärihakupuu muuttamalla se puna-mustaksi puuksi.
- Sitten tutustutaan monihaaraisiin puihin: merkkijonopuu Trie ja B-puu.
- Lopuksi vilkaistaan splay- ja AVL-puita.

12.1 Tavallinen binäärihakupuu

Kertauksena: *Binääripuu* (*binary tree*) on äärellinen solmuista (*node*) koostuva rakenne, joka on joko

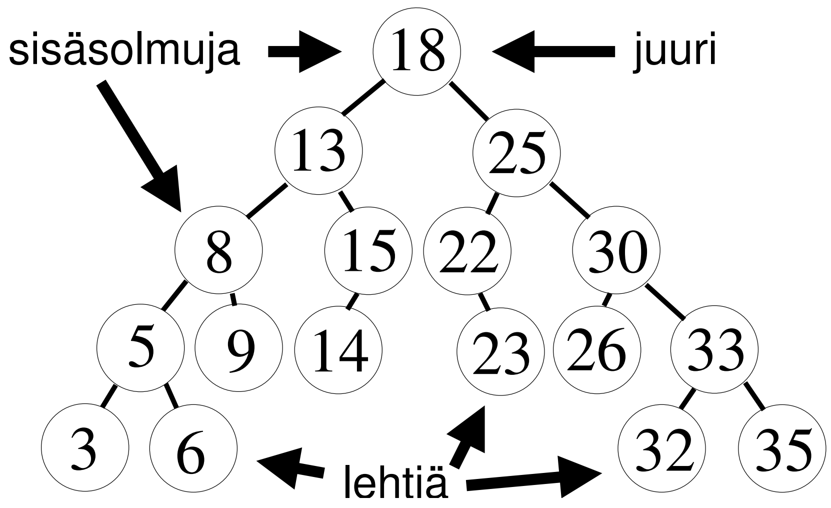
- tyhjä, tai
- sisältää yhden solmun nimeltä *juuri* (*root*), sekä kaksi binääripuuta nimeltä *vasen alipuu* (*left subtree*) ja *oikea alipuu* (*right subtree*).



Kuva 14: Kertaus: Binääripuu

Lisäksi määritellään:

- Lapseton solmu: *lehti* (*leaf*).
- Muut solmut *sisäsolmuja*.
- Solmu on lastensa *isä* (*parent*) ja solmun *esi-isiä* (*ancestor*) ovat solmu itse, solmun isä, tämän isä jne.
- *Jälkeläinen* (*descendant*) vastaavasti.

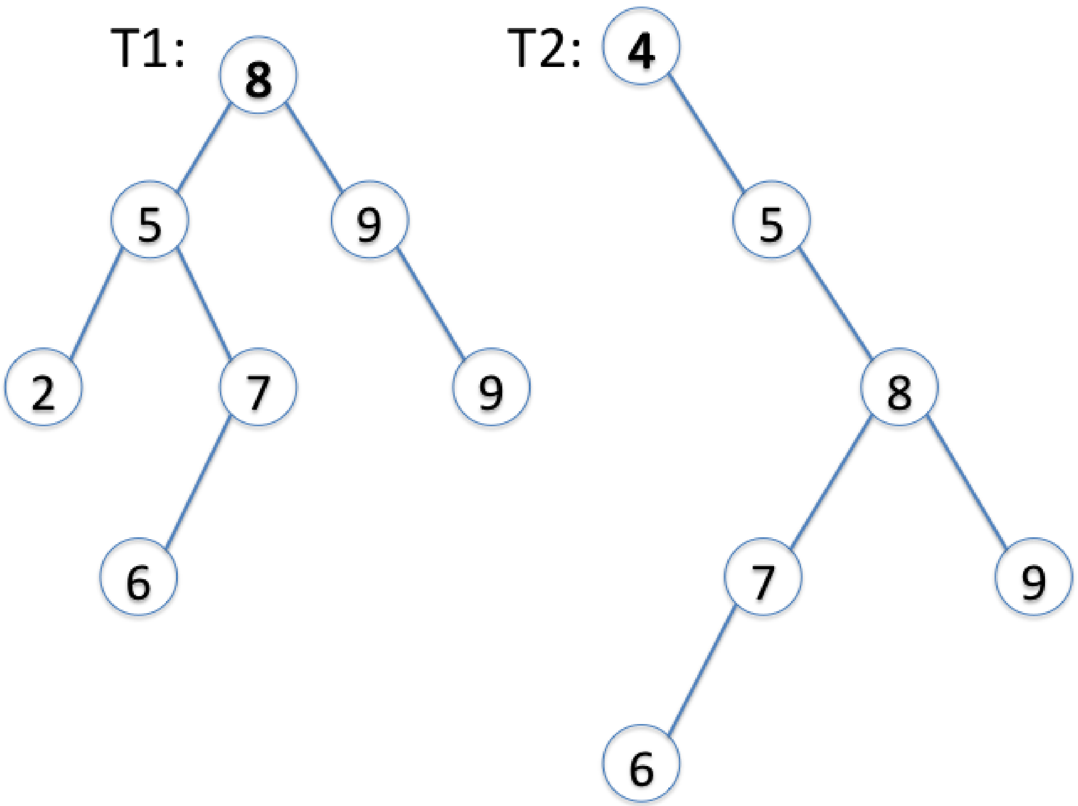


Binäärihakupuu (*binary search tree*) on binääripuu, jonka kaikille solmuille x pätee:

> *Jos l on mikä tahansa solmu x:n vasemmassa alipuussa ja r mikä tahansa solmu x:n oikeassa alipuussa, niin*
>
> $$l.key \leq x.key \leq r.key$$

- Edellisen sivun binääripuu on binäärihakupuu
- Luvussa 8.2 esitelty kekorakenne on binääripuu muttei binääri**haku**puu

Useinmiten binäärihakupuu esitetään linkitettynä rakenteena, jossa jokaisessa alkiossa on kentät avain (key), vasen lapsi ($left$), oikea lapsi ($right$) ja vanhempi (p (parent)). Lisäksi alkiolla on oheisdataa.



Kuva 15: Hakupuita

Avaimen haku binäärihakupuusta :

- koko puusta haku R-TREE-SEARCH($T.root, k$)
- palauttaa osoittimen x solmuun, jolle $x{\rightarrow}key = k$, tai NIL, jos tällaista solmua ei ole

R-TREE-SEARCH(x, k)
if x = NIL or $k = x{\rightarrow}key$ **then**
    return x *(etsitty avain löytyi)*
if $k < x{\rightarrow}key$ **then** *(jos etsitty on pienempi kuin avain...)*
    return R-TREE-SEARCH($x{\rightarrow}left, k$) *(...etsitään vasemmasta alipuusta)*
else *(muuten...)*
    return R-TREE-SEARCH($x{\rightarrow}right, k$) *(...etsitään oikeasta alipuusta)*

Algoritmi suunnistaa juuresta alaspäin huonoimmassa tapauksessa pisimmän polun päässä olevaan lehteen asti.

- suoritusaika $O(h)$, missä h on puun korkeus
- lisämuistin tarve $O(h)$, rekursion vuoksi

Saman voi tehdä myös ilman rekursiota, mikä on suositeltavaa.

- tällöin lisämuistin tarve on vain $\Theta(1)$
- ajoaika on yhä $O(h)$

TREE-SEARCH(x, k)

while $x \neq$ NIL and $k \neq x \rightarrow key$ **do**	*(kunnes avain on löytynyt tai ollaan lehdessä)*
if $k < x \rightarrow key$ **then**	*(jos etsitty on pienempi kuin avain...)*
$x := x \rightarrow left$	*(...siirrytään vasemmalle)*
else	*(muuten...)*
$x := x \rightarrow right$	*(...siirrytään oikealle)*
return x	*(palautetaan tulos)*

Minimi ja *maksimi*:

- minimi löydetään menemällä vasemmalle niin kauan kun se on mahdollista

TREE-MINIMUM(x)
while $x \rightarrow left \neq$ NIL **do**
    x := $x \rightarrow left$
return x

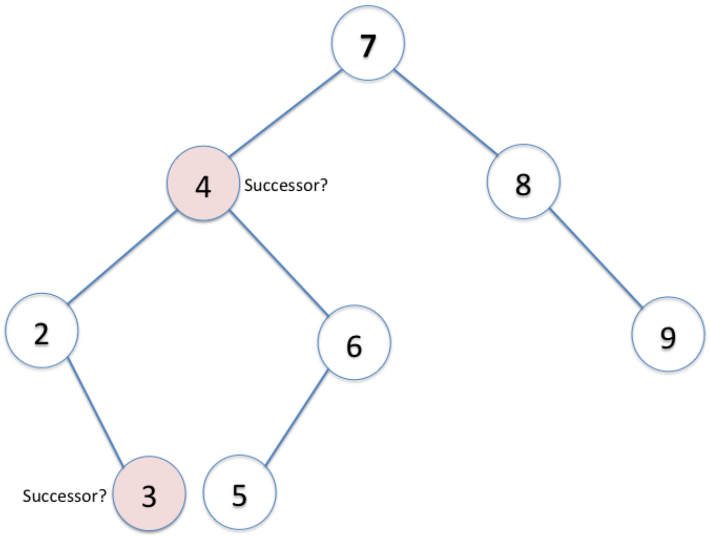
- maksimi löydetään vastaavasti menemällä oikealle niin kauan kun se on mahdollista

TREE-MAXIMUM(x)
while $x \rightarrow right \neq$ NIL **do**
    x := $x \rightarrow right$
return x

- molempien ajoaika on $O(h)$ ja lisämuistin tarve $\Theta(1)$

Solmun *seuraajaa* ja *edeltäjää* kannattaa etsiä binäärihakupuusta puun rakenteen avulla mieluummin kuin avainten arvojen perusteella.

- tällöin kaikki alkiot saadaan käytyä niiden avulla läpi, vaikka puussa olisi yhtä suuria avaimia
  ⇒ tarvitaan siis algoritmi, joka etsii annettua solmua välijärjestyksessä seuraavan solmun
- sellainen voidaan rakentaa algoritmin TREE-MINIMUM avulla

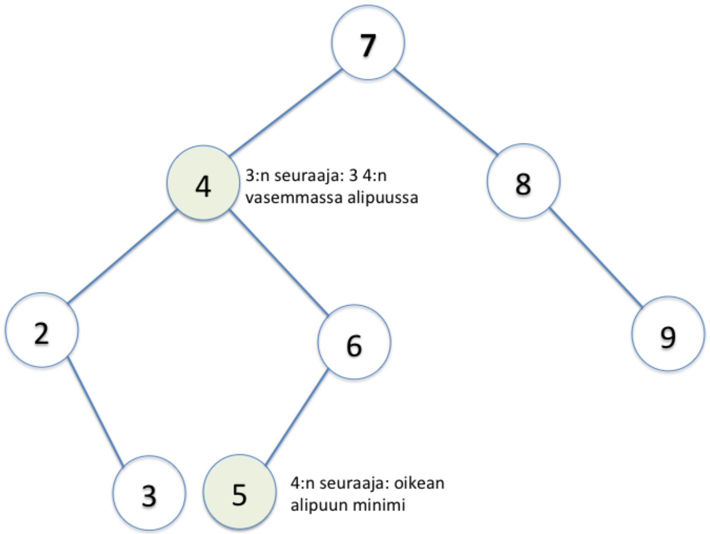


Kuva 16: Solmun seuraaja?

Binäärihakupuun solmun seuraaja on joko:

- oikean alipuun pienin alkio
- tai solmusta juureen vievällä polulla ensimmäinen kohdattu solmu, jonka vasempaan alipuuhun solmu kuuluu

jos edellä mainittuja solmuja ei löydy, on kysymyksessä puun suurin solmu



Kuva 17: Seuraajat

```
TREE-SUCCESSOR(x)
if x→right ≠ NIL then                       (jos oikea alipuu löytyy...)
    return TREE-MINIMUM(x→right)            (...etsitään sen minimi)
y := x→p                                    (muuten lähdetään kulkemaan kohti juurta)
while y ≠ NIL and x = y→right do            (kunnes ollaan tultu vasemmasta lapsesta)
    x := y
    y := y→p
return y                                    (palautetaan löydetty solmu)
```

- huomaa, että avainten arvoja ei edes katsota!
- vrt. seuraajan löytäminen järjestetystä listasta
- ajoaika $O(h)$, lisämuistin tarve $\Theta(1)$
- TREE-PREDECESSOR voidaan toteuttaa vastaavalla tavalla

TREE-SUCCESSORIN ja TREE-MINIMUMIN avulla voidaan rakentaa toinen tapa selata puu läpi välijärjestyksessä.

```
TREE-SCAN-ALL(T)
if T.root ≠ NIL then
    x := TREE-MINIMUM(T.root)        (aloitetaan selaus puun minimistä)
else
    x := NIL
while x ≠ NIL do                     (selataan niin kauan kun seuraajia löytyy)
    käsittele alkio x
    x := TREE-SUCCESSOR(x)
```

- jokainen kaari kuljetaan kerran molempiin suuntiin
  $\Rightarrow$ TREE-SCAN-ALL selviää ajassa $\Theta(n)$, vaikka kutsuukin TREE-SUCCESSORia n kertaa

- lisämuistin tarve $\Theta(1)$
  $\Rightarrow$ TREE-SCAN-ALL on asymptoottisesti yhtä nopea, ja muistinkulutukseltaan asymptoottisesti parempi kuin INORDER-TREE-WALK
  - vakiokertoimissa ei suurta eroa

  $\Rightarrow$ kannattaa valita TREE-SCAN-ALL, jos tietueissa on p-kentät
- TREE-SCAN-ALL sallii useat yhtäaikaiset selaukset, INORDER-TREE-WALK ei

Lisäys binäärihakupuuhun:

```
TREE-INSERT(T, z)                     (z osoittaa käyttäjän varaamaa alustettua tietuetta)
y := NIL; x := T.root                 (aloitetaan juuresta)
while x ≠ NIL do                      (laskeudutaan kunnes kohdataan tyhjä paikka)
    y := x                            (otetaan potentiaalinen isä-solmu talteen)
    if z→key < x→key then             (siirrytään oikealle tai vasemmalle)
        x := x→left
    else
        x := x→right
z→p := y                              (sijoitetaan löydetty solmu uuden solmun isäksi)
if y = NIL then
    T.root := z                       (puun ainoa solmu on juuri)
else if z→key < y→key then            (sijoitetaan uusi solmu isänsä vasemmaksi . . . )
    y→left := z
else                                  (. . . tai oikeaksi lapseksi)
    y→right := z
z→left := NIL; z→right := NIL
```

Algoritmi suunnistaa juuresta lehteen; uusi solmu sijoitetaan aina lehdeksi.
$\Rightarrow$ ajoaika $O(h)$, lisämuistin tarve $\Theta(1)$

Poisto on monimutkaisempaa, koska se voi kohdistua sisäsolmuun:

```
TREE-DELETE(T, z)                               (z osoittaa poistettavaa solmua)
if z→left = NIL or z→right = NIL then           (jos z:lla on vain yksi lapsi . . . )
    y := z                                      (. . . asetetaan z poistettavaksi tietueeksi)
else
    y := TREE-SUCCESSOR(z)                      (muuten poistetaan z:n seuraaja)
if y→left ≠ NIL then                            (otetaan talteen poistettavan ainoa lapsi)
    x := y→left
else
    x := y→right
if x ≠ NIL then                                 (jos lapsi on olemassa . . . )
    x→p := y→p                                  (. . . linkitetään se poistettavan tilalle)
if y→p = NIL then                               (jos poistettava oli juuri . . . )
    T.root := x                                 (. . . merkitään x uudeksi juureksi)
else if y = y→p→left then                       (sijoitetaan x poistettavan tilalle . . . )
    y→p→left := x                               (. . . sen isän vasemmaksi . . . )
else
    y→p→right := x                              (. . . tai oikeaksi lapseksi)
if y ≠ z then                                   (jos poistettiin joku muu kuin z . . . )
    z→key := y→key                              (. . . vaihdetaan poistetun ja z:n datat)
    z→satellitedata := y→satellitedata
return y                                        (palautetaan osoitin poistettuun solmuun)
```

Huom! Rivillä 5 todellakin tiedetään, että y:llä on korkeintaan yksi lapsi.

- jos z:lla on vain yksi lapsi, y on z
- jos rivillä 4 kutsutaan TREE-SUCCESSORIA tiedetään, että z:lla on oikea alipuu, jonka minimi on y
  - minimillä ei voi olla vasenta lasta

Algoritmi näyttää monimutkaiselta, mutta rivin 4 TREE-SUCCESSORIA lukuunottamatta kaikki operaatiot ovat vakioaikaisia.
$\Rightarrow$ ajoaika on siis $O(h)$ ja lisämuistin tarve $\Theta(1)$

Siis kaikki dynaamisen joukon perusoperaatiot saadaan binäärihakupuulla toimimaan ajassa $O(h)$ ja lisämuistilla $\Theta(1)$:

SEARCH, INSERT, DELETE, MINIMUM, MAXIMUM, SUCCESSOR ja PREDECESSOR

Binäärihakupuita - kuten muitakin tietorakenteita - voi sovittaa uusiin tehtäviin lisäämällä uusia tehtävän kannalta olennaista tietoa sisältäviä kenttiä.

- tällöin perusoperaatiot tulee muokata ylläpitämään myös uusien kenttien sisältöä
- esimerkiksi lisäämällä solmuihin kenttä, joka kertoo solmun virittämän alipuun koon
  - saadaan toteutettua algoritmi, joka palauttaa puun korkeuteen nähden lineaarisessa ajassa i:nnen alkion
  - saadaan toteutettua algoritmi, joka puun korkeuteen nähden lineaarisessa ajassa kertoo, monesko kysytty alkio on suuruusjärjestyksessä
  - ilman ylimääräistä kenttää algoritmit täytyisi toteuttaa huomattavasti tehottomammin puun alkioiden määrään nähden lineaarisessa ajassa

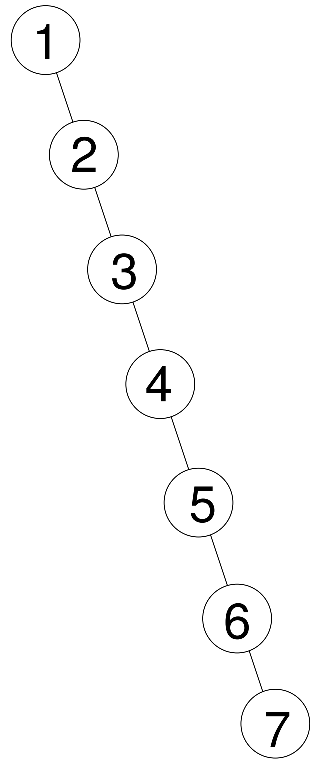
12.2 Kuinka korkeita binäärihakupuut yleensä ovat?

Kaikki dynaamisen joukon perusoperaatiot saatiin binäärihakupuulla toimimaan ajassa $O(h)$. $\Rightarrow$ puun korkeus on tehokkuuden kannalta keskeinen tekijä.

Jos oletetaan, että alkiot on syötetty satunnaisessa järjestyksessä, ja jokainen järjestys on yhtä todennäköinen, suoraan INSERTillä rakennetun binäärihakupuun korkeus on keskimäärin $\Theta(\lg n)$.
$\Rightarrow$ kaikki operaatiot keskimäärin $\Theta(\lg n)$

Valitettavasti lopputulos on surkeampi, jos avaimet syötetään (lähes) suuruusjärjestyksessä, kuten viereisestä kuvasta voi havaita.

- korkeus on n -1, surkeaa!



Ongelmaa ei pystytä ratkaisemaan järkevästi esimerkiksi satunnaistamalla, jos halutaan säilyttää kaikki dynaamisen joukon operaatiot.

Puu pitää siis tasapainottaa. Siihen palataan myöhemmin.

15 Tasapainotetut puurakenteet

Binäärihakupuu toteuttaa kaikki dynaamisen joukon operaatiot $O(h)$ ajassa

Kääntöpuolena on, että puu voi joskus litistyä listaksi, jolloin tehokkuus menetetään ($O(n)$)

Tässä luvussa käsitellään tapoja pitää huolta siitä, ettei litistymistä käy

Ensin opitaan tasapainoitus puna-mustan puun invarianttia ylläpitämällä

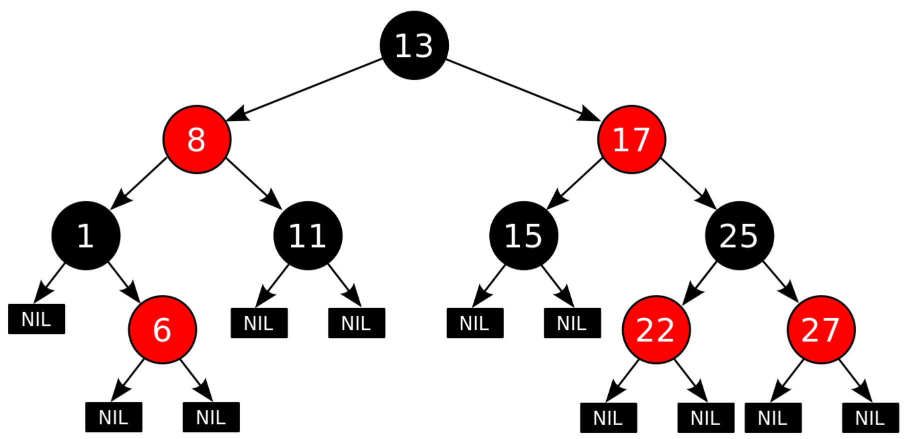
Lopuksi vilkaistaan muista tasapainotetuista binäärihakupuista Splay- ja AVL-puita

15.1 Puna-musta binäärihakupuu

Puna-mustat puut ovat tasapainotettuja binäärihakupuita.

Ne tekevät lisäysten ja poistojen yhteydessä tasapainotustoimenpiteitä, jotka takaavat, ettei haku ole koskaan tehoton vaikka alkiot olisikin lisätty puuhun epäsuotuisassa järjestyksessä.

- puna-musta puu ei voi koskaan litistyä listaksi, kuten perusbinäärihakupuu



Kuva 23: Punamustapuu (via Wikipedia, ©Colin M.L. Burnett (CC BY-SA 3.0))

Puna-mustien puiden perusidea:

- jokaisessa solmussa on yksi lisäbitti: *väri* (*colour*)
  - arvot *punainen* ja *musta*
- muut kentät ovat vanhat tutut key, $left$, $right$ ja p
  - jätämme oheisdatan näyttämättä, jotta pääideat eivät hukkuisi yksityiskohtien taakse
- värikenttien avulla ylläpidetään *puna-mustan puun invarianttia*, joka takaa, että puun korkeus on aina kertaluokassa $\Theta(\lg n)$

Puna-mustien puiden **invariantti**:

1. Jos solmu on punainen, niin sillä joko
   - ei ole lapsia, tai
   - on kaksi lasta, ja ne molemmat ovat mustia.
2. Jokaiselle solmulle pätee: jokainen solmusta alas 1- tai 0-lapsiseen solmuun vievä polku sisältää saman määrän mustia solmuja.
3. Juuri on musta.

Solmun x *musta-korkeus* (*black-height*) $bh(x)$ on siitä alas 1- tai 0-lapsiseen solmuun vievällä polulla olevien mustien solmujen määrä.

- invariantin osan 3 mukaisesti jokaisen solmun mustakorkeus on yksikäsitteinen
- jokaisella vaihtoehtoisella polulla on sama määrä mustia solmuja
- koko puun mustakorkeus on sen juuren mustakorkeus

Puna-mustan puun maksimikorkeus

- merkitään korkeus = h ja solmujen määrä = n
- kunkin juuresta lehteen vievän polun solmuista vähintään puolet ($\lfloor \frac{h}{2} \rfloor + 1$) ovat mustia (invariantin osat 1 ja 3)
- jokaisella juuresta lehteen vievällä polulla on saman verran mustia solmuja (invariantin osa 2)
  $\Rightarrow$ ainakin $\lfloor \frac{h}{2} \rfloor + 1$ ylintä tasoa täysiä
  $\Rightarrow n \geq 2^{\frac{h}{2}}$
  $\Rightarrow h \leq 2 \lg n$

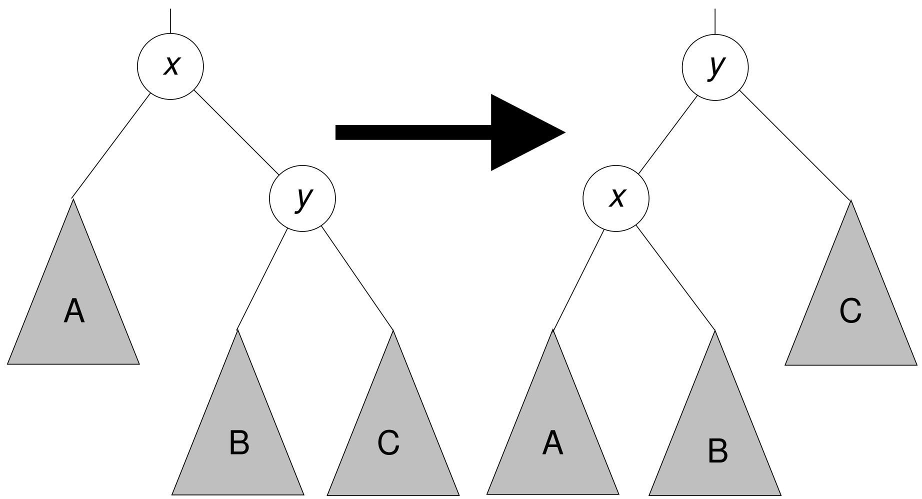
Invariantti siis todellakin takaa puun korkeuden pysymisen logaritmisena puun alkioiden määrään nähden.
$\Rightarrow$ Dynaamisen joukon operaatiot SEARCH, MINIMUM, MAXIMUM, SUCCESSOR ja PREDECESSOR saadaan toimimaan puna-mustille puille ajassa $O(\lg n)$.

- binäärihakupuulle operaatiot toimivat ajassa $O(h)$, ja puna-musta puu on binäärihakupuu, jolle $h = \Theta(\lg n)$

Puna-mustien puiden ylläpitämiseen ei kuitenkaan voida käyttää samoja lisäys- ja poistoalgoritmeja kuin tavallisilla binäärihakupuilla, koska ne saattavat rikkoa invariantin.

Niiden sijaan käytetään algoritmeja RB-INSERT ja RB-DELETE.

- operaatiot RB-INSERT ja RB-DELETE perustuvat *kiertoihin* (*rotation*)
- kiertoja on kaksi: vasemmalle ja oikealle
- ne muuttavat puun rakennetta, mutta säilyttävät binäärihakupuiden perusominaisuuden kaikille solmuille



- kierto vasemmalle
  - olettaa, että solmut x ja y ovat olemassa
- kierto oikealle vastaavasti
  - $left$ ja $right$ vaihtaneet paikkaa

```
LEFT-ROTATE(T, x)
    y := x→right; x→right := y→left
    if y→left ≠ NIL then
        y→left→p := x
    y→p := x→p
    if x→p = NIL then
        T.root := y
    else if x = x→p→left then
        x→p→left := y
    else
        x→p→right := y
    y→left := x; x→p := y
```

- molempien kiertojen ajoaika on $\Theta(1)$
- ainoastaan osoittimia muutetaan

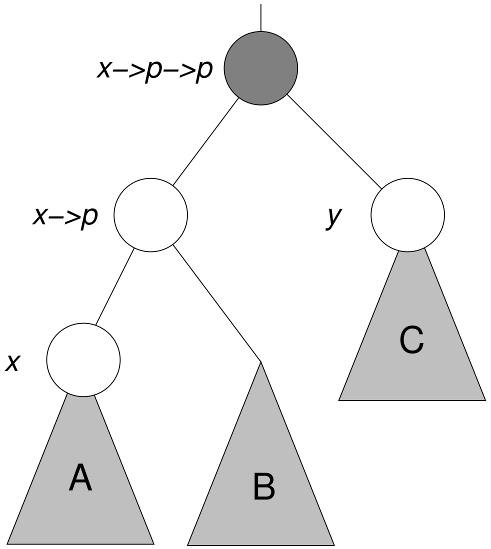
Lisäyksen perusidea

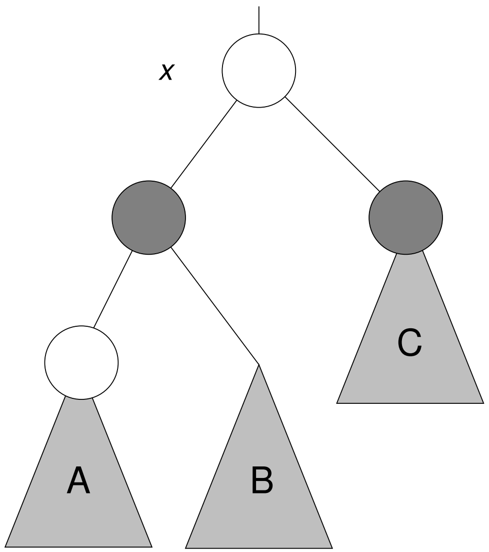
- ensin uusi solmu lisätään kuten tavalliseen binäärihakupuuhun
- sitten lisätty väritetään punaiseksi
- mitä puna-mustien puiden perusominaisuuksia näin tehty lisäys voi rikkoa?

- Invariantin osa
  - 1 rikkoutuu lisätyn solmun osalta, jos sen isä on punainen; muuten se ei voi rikkoutua.
  - 2 ei rikkoudu, koska minkään solmun alla olevien mustien solmujen määrät ja sijainnit eivät muutu, ja lisätyn alla ei ole solmuja.
  - 3 rikkoutuu, jos puu oli alun perin tyhjä.

- korjataan puu seuraavasti:
  - ominaisuutta 2 pilaamatta siirretään 1:n rike ylöspäin kunnes se katoaa
  - lopuksi 3 korjataan värittämällä juuri mustaksi (ei voi pilata ominaisuuksia 1 ja 2)
- 1:n rike = sekä solmu että sen isä ovat punaisia
- siirto tapahtuu värittämällä solmuja ja tekemällä kiertoja

```
RB-INSERT(T, x)
1   TREE-INSERT(T, x)
2   x→colour := RED
    (suoritetaan silmukkaa kunnes rike on hävinnyt tai ollaan saavutettu juuri)
3   while x ≠ T.root and x→p→colour = RED do
4       if x→p = x→p→p→left then
5           y := x→p→p→right
6           if y ≠ NIL and y→colour = RED then  (siirretään rikettä ylöspäin)
7               x→p→colour := BLACK
8               y→colour := BLACK
9               x→p→p→colour := RED
10              x := x→p→p
11          else                                 (siirto ei onnistu → korjataan rike)
12              if x = x→p→right then
13                  x := x→p; LEFT-ROTATE(T, x)
14              x→p→colour := BLACK
15              x→p→p→colour := RED
16              RIGHT-ROTATE(T, x→p→p)
17      else
. . .       ▷ sama kuin rivit 5. . . 16 paitsi "left" ja "right" vaihtaneet paikkaa
30  T.root→colour := BLACK                       (väritetään juuri mustaksi)
```





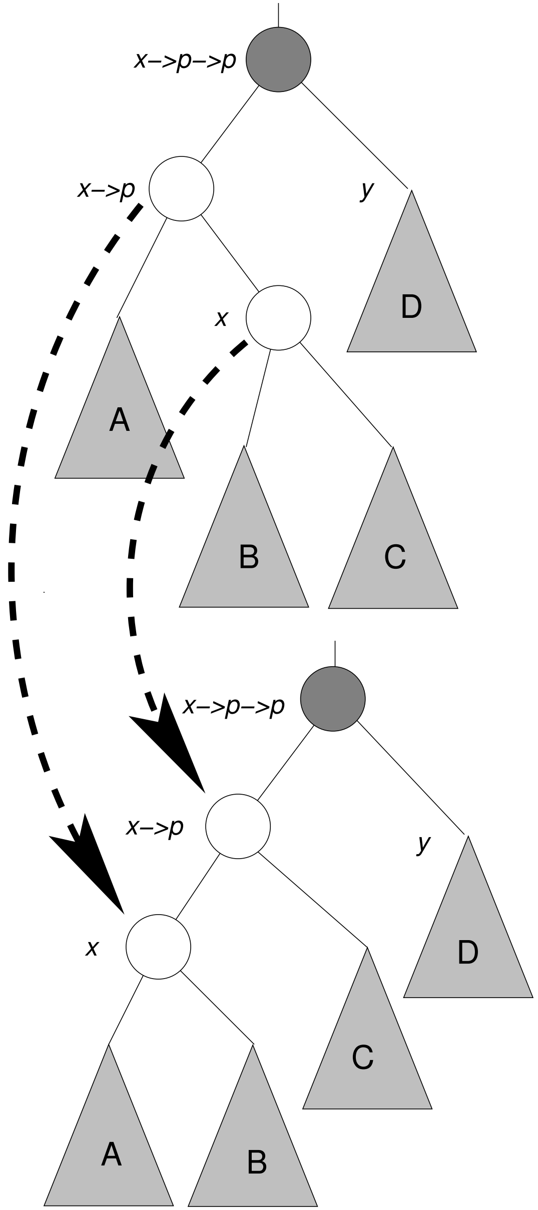
Ominaisuuden 1 rikkeen siirto ylöspäin:

- solmu x ja sen isä ovat molemmat punaisia.
- myös solmun x setä on punainen ja isoisä musta.

$\Rightarrow$ rike siirretään ylöspäin värittämällä sekä x:n setä että isä mustiksi ja isoisä punaiseksi.

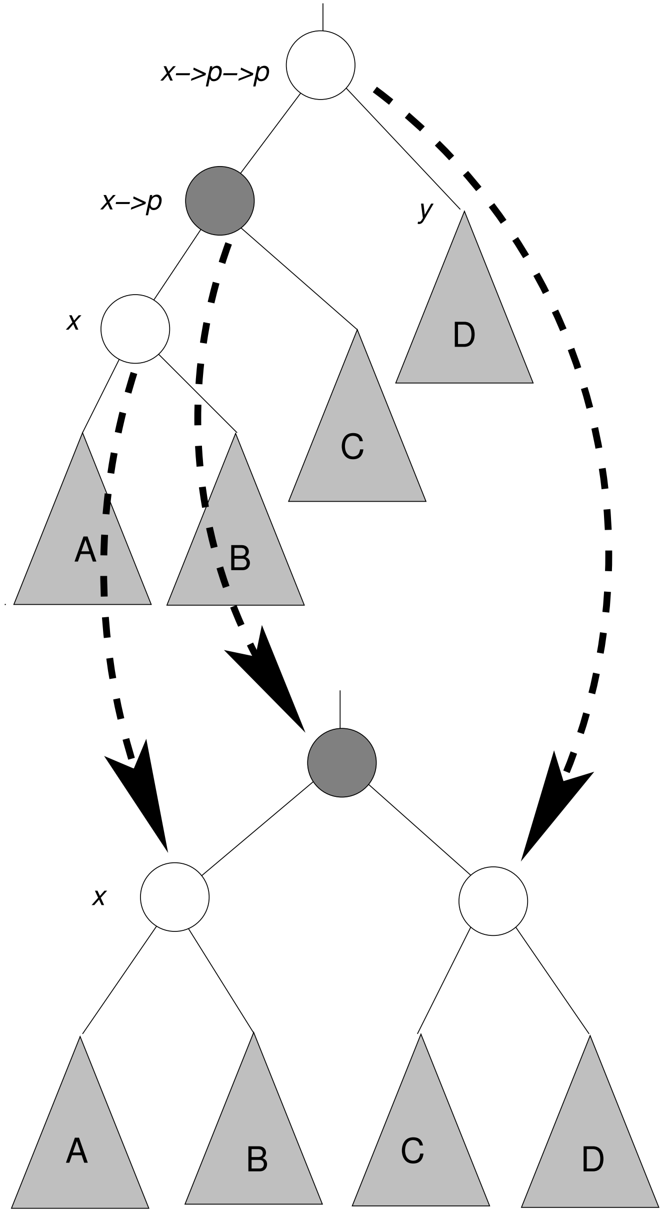
Korjauksen jälkeen:

- ominaisuus 1 saattaa olla edelleen rikki
  - solmu x ja sen isä saattavat molemmat olla punaisia
- ominaisuus 2 ei rikkoudu
  - kaikkien polkujen mustien solmujen määrä pysyy samana
- ominaisuus 3 saattaa rikkoutua
  - jos ollaan noustu juureen asti, se on saatettu värittää punaiseksi



Mikäli punaista setää ei ole olemassa, rikettä ei voi siirtää ylöspäin vaan se täytyy poistaa käyttäen monimutkaisempaa menetelmää:

- Varmistetaan ensin, että x on isänsä vasen lapsi tekemällä tarvittaessa kierto vasemmalle.



- tämän jälkeen väritetään x:n isä mustaksi ja isoisä punaiseksi, ja suoritetaan kierto oikealle
  - isoisä on varmasti musta, koska muuten puussa olisi ollut kaksi punaista solmua päällekkäin jo ennen lisäystä

Korjauksen jälkeen:

- puussa ei enää ole päällekkäisiä punaisia solmuja
- korjausoperaatiot yhdessä eivät riko 2. ominaisuutta

  $\Rightarrow$ puu on ehjä ja korjausalgoritmin suorittaminen voidaan lopettaa

Poistoalgoritmin yleispiirteet

- ensin solmu poistetaan kuten tavallisesta binäärihakupuusta
  - w osoittaa poistettua solmua
- jos w oli punainen tai puu tyhjeni kokonaan, puna-musta-ominaisuudet säilyvät voimassa
  $\Rightarrow$ ei tarvitse tehdä muuta
- muussa tapauksessa korjataan puu RB-DELETE-FIXUPin avulla aloittaen w:n (mahdollisesta) lapsesta x ja sen isästä $w \rightarrow p$
  - TREE-DELETE takaa, että w:llä oli enintään yksi lapsi

```
RB-DELETE(T, z)
    w := TREE-DELETE(T, z)
    if w→colour = BLACK and T.root ≠ NIL then
        if w→left ≠ NIL then
            x := w→left
        else
            x := w→right
        RB-DELETE-FIXUP(T, x, w→p)
    return w
```

```
RB-DELETE-FIXUP(T, x, y)
1   while x ≠ T.root and (x = NIL or x→colour = BLACK) do
2       if x = y→left then
3           w := y→right
4           if w→colour = RED then
5               w→colour := BLACK; y→colour := RED
6               LEFT-ROTATE(T, y); w := y→right
7           if (w→left = NIL or w→left→colour = BLACK) and
            (w→right = NIL or w→right→colour = BLACK)
            then
8               w→colour := RED; x := y
9           else
10              if w→right = NIL or w→right→colour = BLACK then
11                  w→left→colour := BLACK
12                  w→colour := RED
13                  RIGHT-ROTATE(T, w); w := y→right
14              w→colour := y→colour; y→colour := BLACK
15              w→right→colour := BLACK; LEFT-ROTATE(T, y)
16              x := T.root
17      else
...         ▷ sama kuin rivit 3...16 paitsi "left" ja "right" vaihtaneet paikkaa
32      y := y→p
33  x→colour := BLACK
```

15.2 AVL-puut ja Splay-puut

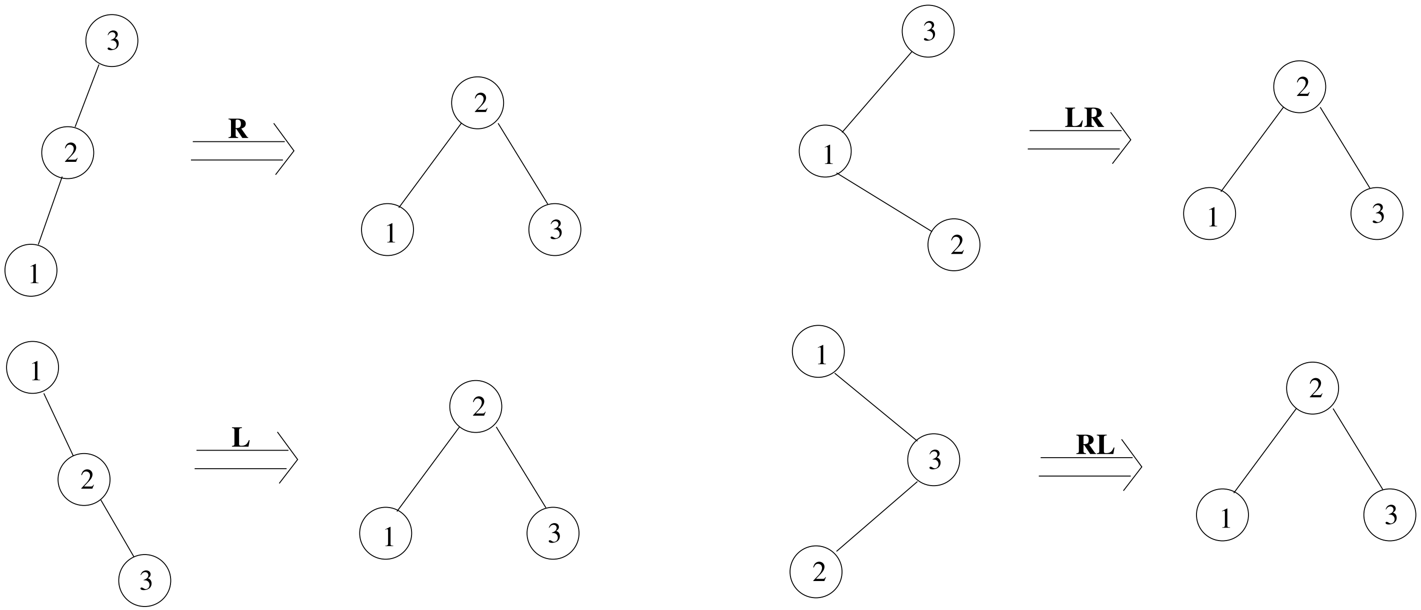
AVL puu (Adelson-Velsky, Landis mukaan) on binäärihakupuu, jossa jokaisella solmulla on **tasapainokerroin**: 0, +1, tai -1, kun tasapainossa.

- kerroin määräytyy solmun oikean ja vasemman alipuun korkeuksien erotuksesta.

Kun uuden solmun lisäys tekee AVL-puusta epätasapainoisen, puu palautetaan tasapainoiseksi tekemällä rotaatioita.

Mahdollisia rotaatioita on neljä:

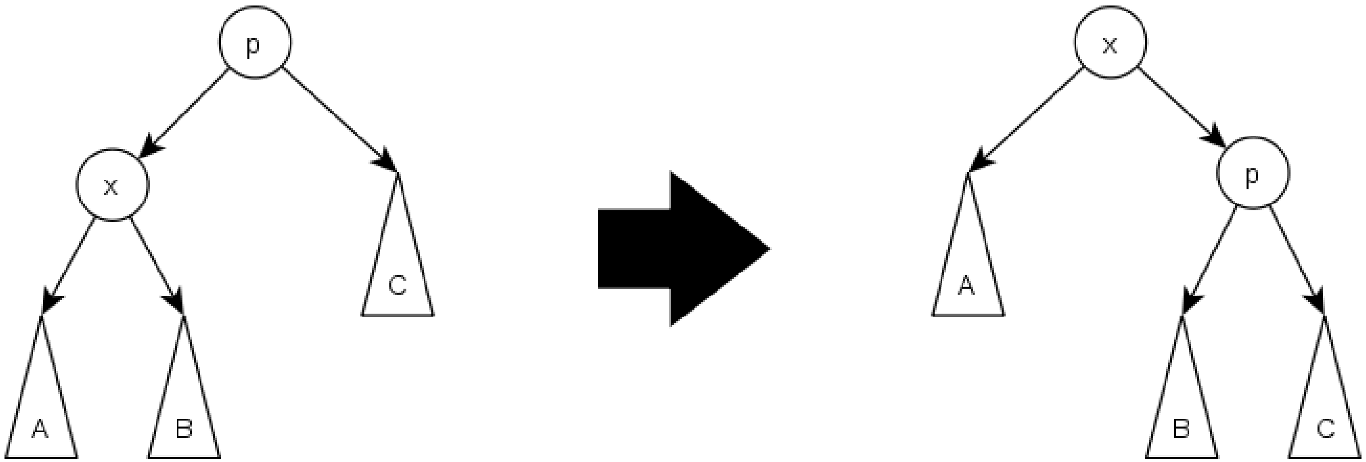
- Oikealle
- Vasemmalle
- Kaksois-rotaatio vasen-oikea
- Kaksois-rotaatio oikea-vasen



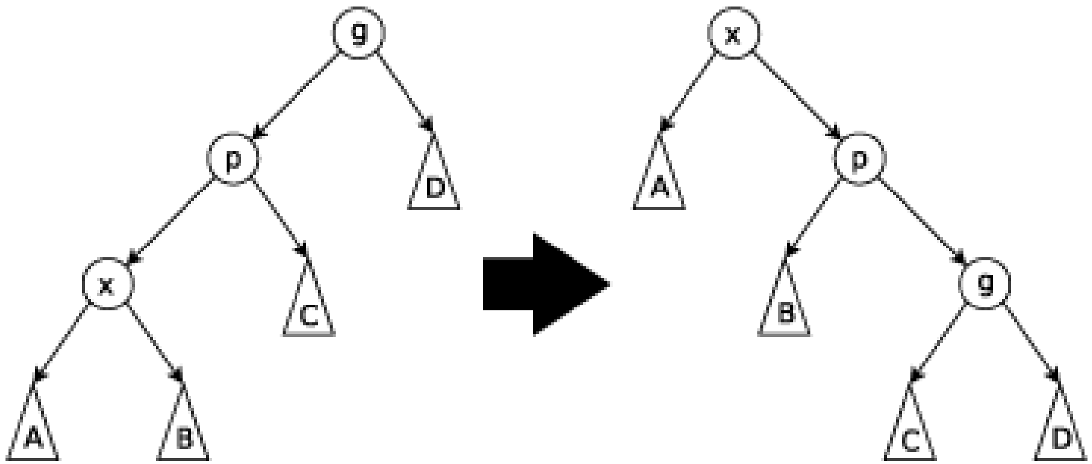
Splay puu on binäärihakupuu, jossa lisäominaisuutena viimeksi haetut alkiot ovat nopeita hakea uudelleen.

Splay-operaatio suoritetaan solmulle haun yhteydessä. Tämä ns. splay-askelien sekvenssi siirtää solmun askel askeleelta lähemmäksi juurta ja lopulta juureksi.

- Zig-askel:



- Zig-Zig-askel:



- Zig-Zag-askel: