hyväksymispäivä arvosana

arvostelija

Binääriset hakupuut

Jasu Viding

Helsinki 27.3.2016

Kandidaatintutkielma

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen	Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author
Jasu Viding

Työn nimi — Arbetets titel — Title

Binääriset hakupuut

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Kandidaatintutkielma	27.3.2016	29 sivua

Tiivistelmä — Referat — Abstract

Tämä tutkielma käsittelee binäärisen hakupuun eri variaatioita ja tarkastelee niiden kunkin hyviä ja huonoja puolia aikavaativuuden ja käytettävyyden näkökulmasta. Käsiteltäviin binäärisiin hakupuihin lukeutuvat tavallinen, tasapainotettu, optimoitu sekä ennakoiva binäärinen hakupuu. Pahimpien käyttötapausten aikavaativuuden määrittämisen lisäksi tutkielma esittelee myös pintapuolisesti tasoitetun analyysin aikavaativuudelle, mitä hyödyntäen se esittelee aiemmin mainittujen hakupuiden lisäksi vielä mukautuvan binäärisen hakupuun ja splay-puun.

ACM Computing Classification System (CCS): A.1 [Introductory and Survey], I.7.m [Document and Text Processing]: Miscellaneous

Avainsanat — Nyckelord — Keywords
binäärinen hakupuu, tasoitettu analyysi, splay-puu

Säilytyspaikka — Förvaringsställe — Where deposited
Tietojenkäsittelytieteen laitoksen kirjasto, sarjanumero C-2004-X

Muita tietoja — övriga uppgifter — Additional information

Sisältö

1 Johdanto

Binäärinen hakupuu on puumainen tietorakenne, joka järjestää tietoa niin kutsuttuihin solmuihin, joilla kullakin on aina kaksi lapsisolmua. Sen tehokkuus perustuu sen rakenteen helppoon muokkaamiseen ja nopeaan tiedonhakuun. Tiedonhaun aikavaativuus binääriselle hakupuulle on kuitenkin täysin riippuvainen siitä, miten se on rakentunut.

Helpoin tapa rakentaa tehokas binäärinen hakupuu on käyttää tasapainotusehtoa. Tasapainotusehto takaa, ettei hakupuusta voi muodostua toispuoleista. Tällöin yksittäisen haun aikavaatimus on pahimmassakin tapauksessa $O(\log\ n)$. Tasapainotusehto ei kuitenkaan huomioi puuhun kohdistuvaa käyttöä, jolloin kaikki suoritettavat haut saattavat kohdistua sen juuresta etäisimpiin solmuihin.

Jos tiedämme hakupuulle suoritettavien käyttötapausten toistuvuuden etukäteen, voimme rakentaa puun optimaaliseksi vastaamaan juuri näitä käyttötapauksia. Optimoitu hakupuu on tällöin kyseiselle sarjalle operaatioita nopein mahdollinen. Se rakennetaan järjestämällä sen useimmin haetut arvot lähemmäs juurta, kuin sen harvemmin haetut arvot. Optimoidun binäärisen hakupuun päivittämisen aikavaativuus on kuitenkin kelvottoman suuri, sillä jokainen lisäys- ja poisto-operaatio vaativat koko puun uudelleen järjestämisen.

Ennakoiva hakupuu pyrkii yhdistämään optimoidun hakupuun nopeat haut sekä tasapainotetun hakupuun nopeat lisäys- ja poisto-operaatiot. Sen uudelleen järjestävät algoritmit ovat kuitenkin vielä optimoitua hakupuutakin vaikeampia sekä siihen kohdistuvat rajoitukset voivat olla vielä paljon tasapainotettuja hakupuitakin monimutkaisempia.

Parhain tapa rakentaa tehokas hakupuu ei olekaan aina yrittää minimoida sen jokaisen yksittäisen käyttötapauksen aikavaativuutta.Yleensä hakupuille suoritetaan käytössä aina yhden operaation sijaa sarja useita perättäisiä operaatioita, jolloin

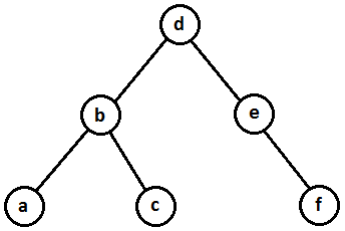
kunkin yksittäisen operaation aikavaativuutta olennaisempaa on kaikkien suoritettavien peraatioiden yhteenlaskettu aikavaativuus. Tälle oletukselle perustuukin niin kutsuttu tasoitettu analyysi, jota käyttämällä pyrimme minimoimaan sarjassa pahimpia mahdollisia käyttötapauksia niiden kunkin keskimääräisen aikavaativuuden absoluuttisen aikavaativuuden sijaan.

Mukautuva binäärinen hakupuu on aina kullakin sen käytön hetkellä ennalta määräämättömässä muodossa oleva puurakenne, joka jokaisella sille suoritettavalla operaatiolla pyrkii uudelleen järjestymään siten, että kaikki seuraavat sille suoritettavat operaatiot olisivat nopeampia. Sen toiminta rakentuu oletukselle, että tiedonhaku klusteroituu. Täten se pyrkii käyttönsä aikana aina järjestämään kaikki sen useimmin haetut arvot lähelle sen juurta olettaen, että niitä haetaan pian uudestaan. Mukautuvalle hakupuulle on kuitenkin olemassa sarjoja perättäisiä pahimpia käyttötapauksia, joilla jokaisen yksittäisen operaation aikavaativuus on $O(n)$.

Splay-puu on mukautuvasta hakupuusta jatkokehitetty hakupuun muoto, jolla sarjalle pahimpia perättäisiä operaatioita on aikavaatimus tavallista mukautuvaa hakupuuta pienempi $O(\log n)$. Sen ylivertaisuus perustuu sen poikkeavaan tapaan uudelleen järjestää rakennettaan jokaisella sille suoritettavalla haulla. Sen lisäksi, että se siirtää haetun solmun hakupuun uudeksi juureksi, se myös puolittaa kaikkien haun varrella olleiden solmujen syvyyden puoleen.

2 Binääriset hakupuut

Binäärinen hakupuu on hakurakenne, joka on toteutettu binääripuun avulla. Binääripuu on tietojenkäsittelytieteessä yleisesti käytetty puumainen järjestetty tietorakenne, jonka jokaiselle solmulle pätee, että sillä voi olla enintään kaksi lapsisolmua, joihin viitataan sen oikeana ja vasempana lapsena. Tässä tutkielmassa viittaamme solmun x vanhempaan solmuna $p(x)$ ja isovanhempaan solmuna $p^2(x)$.

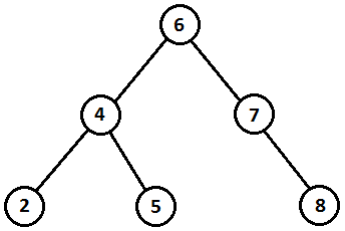


Kuva 1.1: Puun juuri on solmu d. Sen solmut a, c ja f ovat sen lehtisolmut. Solmun c vanhempi on solmu b ja isovanhempi solmu d. Solmut b ja e ovat sisaria, tasolla 1 ja syvyydellä 1. Koko puun korkeus on 2.

Binäärisessä hakupuussa solmuun, jolla ei ole vanhempaa, viitataan puun juurena, ja solmuihin, joilla ei ole kumpaakaan lapsisolmua, lehtisolmuina. Solmun x syvyys on matka puun juurisolmun ja sen itsensä välillä, ja vastaavasti korkeus, matka solmun x ja sen kaukaisimman lehtisolmun välillä. Joukko solmuja tietyllä syvyydellä on taso, ja solmuja, joilla on yhteinen vanhempi, kutsutaan sisaruksiksi. Jos on olemassa puun juurisolmusta alkava polku, joka kulkee solmun p^n kautta solmuun x, niin solmu p^n on solmun x esivanhempi ja solmu x on solmun p^n jälkeläinen.

Alapuuksi kutsutaan puuta, jonka juuri on solmu x, mutta joka ei kuitenkaan ole koko puun juuri, vaan solmulla x on olemassa vanhempi, jonka lapsi se on. Alapuu sisältää solmun x, sekä kaikki sen jälkeläiset.

Binäärisessä hakupuussa jokainen arvo on ainutkertainen, eikä siis ole kahta erillistä solmua, joihin molempiin olisi talletettu sama arvo. Hakupuun rakenne on myös järjestetty siten, että solmun vasemman lapsen arvo on aina solmun omaa arvoa pienempi, ja vastaavasti oikean lapsen arvo aina solmun omaa arvoa suurempi. Täten pätee, että solmun x vasen alapuu, jonka juuri on solmun x vasen lapsi, sisältää vain arvoja, jotka ovat pienempiä kuin solmun x sisältämä arvo, ja vastaavasti oikea alapuu, jonka juuri solmun x oikea lapsi on, sisältää vain suurempia arvoja kuin solmun x sisältämä arvo.[SlT85]



Kuva 1.2: Puun juuressa on talletettuna arvo 6. Sen vasempaan alapuuhun talletetut arvot 2, 4 ja 5 ovat kaikki pienempiä kuin juuressa oleva arvo 6, ja vastaavasti oikeaan alapuuhun talletetut arvot 7 ja 8 ovat molemmat sitä suurempia. Tämä järjestys pätee binäärisessä hakupuussa aina jokaiseen sen solmuun. Esimerkiksi juurisolmun 6 vasemman alapuun juuren arvo 4 on suurempi kuin edelleen sen oman vasemman lapsen arvo 2.

Binäärinen hakupuu tietorakenteena tukee monia dynaamiselle tietorakenteelle tyypillisiä operaatioita, joita ovat esimerkiksi *hakeminen*, *lisääminen* ja *poistaminen*. Näistä hakeminen tapahtuu vertaamalla haettavaa arvoa i hakupuun juuressa olevaan arvoon, jolloin kohtaamme yhden seuraavista tilanteista:

1. Ei ole juurta(binäärinen hakupuu on tyhjä): Haettu arvo i ei ole puussa, ja haku loppuu epäonnistuneena.

2. Haettu arvo i on sama kuin juureen tallennettu arvo: Haku päättyy onnistuneena.

3. Haettu arvo i on pienempi kuin juureen tallennettu arvo: Haku jatkuu siirtymällä juuren vasempaan alapuuhun ja toistamalla samat askeleet alusta.

4. Haettu arvo i on suurempi kuin juureen tallennettu arvo: Haku jatkuu siirtymällä juuren oikeaan alapuuhun ja toistamalla samat askeleet alusta.

Uuden arvon i, olettaen ettei sitä vielä ole hakupuussa, lisääminen tapahtuu suorittamalla samoja askelia kuin haussa, kunnes saavumme yhteen hakupuun lehtisol-

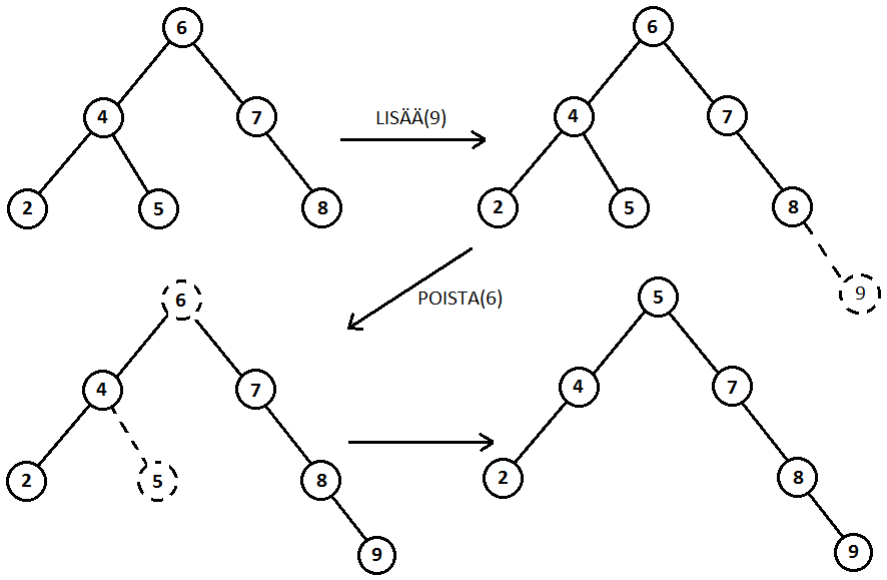
muista. Tämän jälkeen löytämällemme lehtisolmulle lisätään uusi lapsisolmu, jonka arvoksi i talletetaan, ja päätämme lisäyksen onnistuneena.

Vastaavasti myös arvon i poistaminen hakupuusta tapahtuu suorittamalla aluksi samoja haun askelia, kunnes saavumme solmuun x, joka sisältää poistettavan arvon i. Tämän jälkeen kohtaamme jonkin seuraavista vaihtoehdoista:

1. Poistettava solmu x on lehtisolmu: Solmu voidaan vain poistaa.

2. Poistettavalla solmulla x on yksi lapsisolmu: Kopioidaan solmuun x sen lapsisolmussa oleva arvo ja suoritetaan kohta 1 sen lapsisolmulle.

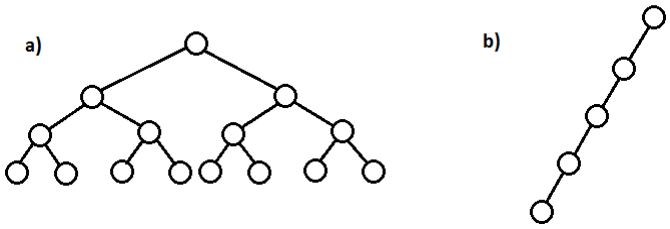
3. Poistettavalla solmulla x on kaksi lapsisolmua: Etsitään suurin arvo solmun x vasemmasta alapuusta ja kopioidaan se poistettavaan solmuun x. Nyt jos solmuun x kopioidun arvon sisältänyt solmu oli lehtisolmu, niin suoritetaan sille kohta 1, tai jos se oli lapsellinen solmu, niin suoritetaan sille kohta 2.



Kuva 1.3: Esimerkki hakupuusta, johon lisätään aluksi uusi arvo 9 ja sen jälkeen poistetaan arvo 6. Huomaa, kuinka poistettava solmu korvataan lehtisolmulla.

Näistä kustakin operaatiosta jokainen on aikavaativuudeltaan verrannollinen hakupuun korkeuteen, joka taas on riippuvainen siitä, miten puu on muodostunut. Parhaassa tapauksessa muodostuneen binäärisen hakupuun korkeus on mahdollisimman pieni, eli noin $\log_2 n$, missä n on solmujen lukumäärä. Tällöin yksittäisen haun, joka alkaa juuresta ja päättyy kuljettuaan koko puun korkeuden aina sen kaukaisimpaan lehtisolmuun, aikavaatimus on $O(\log\ n)$. Tämä vastaa kaikista mahdollisista puulle suoritettavista hauista aikavaativuudeltaan pahinta mahdollista.

Täysin sattumanvaraisesti rakennetun binäärisen hakupuun lopullista muotoa on kuitenkin mahdotonta ennustaa ja siten onkin mahdollista, että puusta on muodostunut rakenteeltaan täysin toispuoleinen, jolloin pahimman mahdollisen haun aikavaatimus voi olla jopa $O(n)$, joka ei ole enää ollenkaan nopeampi kuin vaikkapa peräkkäishaku linkitetystä listasta. [CLR09a]



Kuva 1.4: Esimerkit a) tasapainoisesta binäärisestä hakupuusta ja b) toispuoleisesti rakentuneesta binäärisestä hakupuusta.

2.1 Tasapainotettu binäärinen hakupuu

Yksittäisen arvon hakeminen binäärisestä hakupuusta on sitä nopeampaa, mitä lähempänä juurta haettu arvo on. Koska sattumanvaraisesti haettu arvo voi olla mikä tahansa hakupuun arvoista, on haun aikavaativuuden kannalta oleellisinta, että jokainen puun solmu on mahdollisimman lähellä sen juurta, jolloin aika kulkea niihin on mahdollisimman lyhyt. Esimerkiksi miljoonasolmuisella hakupuulla, joka on täysin toispuoleisesti rakentunut, voi pahimmassa tapauksessa haettu solmu

olla lehtisolmu, jolloin haun aikavaativuudeksi saadaan $O(n) = 1000000$. Kun taas täydelliseen tasapainoon rakentuneella hakupuulla pahin mahdollinen haku sen kaukaisimpaan lehtisolmuun on aikavaativuudeltaan vain $O(\log_2\ n) = 19$.

Rakenteeltaan tasapainoiselle binääriselle hakupuulle suoritettu yksittäinen haku, jollain satunnaisesti valitulla arvolla i, on siis aikavaativuudeltaan ylivertaisesti pienempi kuin täysin toispuoleisesti rakentuneelle suoritettu. Siispä välttääksemme tapaukset, joissa hakupuusta muodostuu toispuoleinen, otamme käyttöön tasapainotusehdon, joka yrittää pitää puun syvyyden mahdollisimman lähellä arvoa $\log_2 n$ taaten pahimman mahdollisen haun aikavaativuuden ylärajaksi $O(\log\ n)$. Tasapainotusehdon toteutuminen vaatii kuitenkin koko hakupuun uudelleen tasapainottamisen jokaisen tietorakenteeseen tehtävän muutoksen jälkeen, joten aivan ilmaista tasapainon ylläpito ei käytön aikavaativuudeltaan kuitenkaan ole.

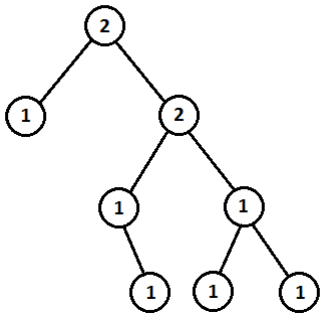
Ensimmäinen tietojenkäsittelytieteessä esitetty itsestään tasapainottuva binäärinen hakupuu oli AVL-puu. Jos AVL-puussa kaikkien solmujen alipuiden korkeusero on enemmän kuin yksi, suoritetaan puun uudelleen tasapainottaminen yhdellä tai useammalla kierrolla, joista kukin kierto vaihtaa aina kahden eri solmun paikkaa. AVL-puun hyvin tiukka tasapainotusehto tekee siitä erinomaisen, jos sen pääasiallinen käyttötarkoitus on vain tiedon hakeminen, sillä sen korkeus on aina pienin mahdollinen.[AdL62] Binäärisille hakupuille suoritetaan tavallisessa käytössä kuitenkin usein niin merkittävä määrä poisto- sekä lisäysoperaatioita, että AVL-puulle löytyy parempiakin vaihtoehtoja.

Yksi tapa määritellä binääriselle hakupuulle tasapainotusehto, on asettaa jokaiselle solmulle x arvoaste $rank(x)$, joka antaa sille seuraavanlaiset ominaisuudet:

i) Jos x on mikä tahansa solmu, jolla on vanhempi, niin $rank(x) \leq rank(p(x)) \leq rank(x)+1$.

ii) Jos x on mikä tahansa solmu, jolla on isovanhempi, niin $rank(x) < rank(p^2(x))$.

iii) Jos x on solmu, jolla ei ole kumpaakaan lapsisolmua, niin $rank(x) = 0$ ja $rank(p(x)) = 1$, tapauksessa, jossa solmulla x on vanhempi.



Kuva 2.1: Esimerkki tasapainotusehdosta. Solmujen numerot kuvaavat niiden arvoastetta.

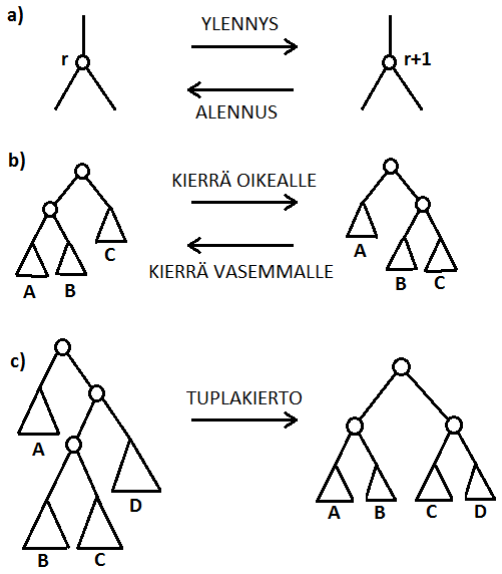
Kutsumme solmua x mustaksi, jos $rank(p(x)) = rank(x)+1$ tai jos $p(x)$ on määrittämätön, ja punaiseksi, jos $rank(p(x)) = rank(x)$. Ominaisuuden *i* mukaan jokainen solmu hakupuussa voi olla joko musta tai punainen, ominaisuuden *ii* mukaan jokaisella punaisella solmulla on musta vanhempi ja ominaisuuden *iii* mukaan jokainen lapseton solmu on musta. Määräämme myös, että jokaiseen puuttuvaan lapseen viitataan nollasolmuna, jonka arvoaste on nolla ja toteuttaa ehdon *iii* ollen musta. Näillä mustilla ja punaisilla solmuilla voimme luoda tasapainotetulle binääriselle hakupuulle määritelmän, jota vastaavaa puuta kutsutaan punamustaksi hakupuuksi.

LEMMA 1. *Solmulla, jonka arvoaste on k, on korkeus enimmillään* $2k$ *ja on ainakin* $2^{(k+1)}$-1 *jälkeläistä. Täten tasapainotusehtoa noudattavalla binäärisellä hakupuulla, jolla on n lapsellista solmua, on syvyys enintään* $2\lfloor \lg(n+1) \rfloor$.

Todistus: Lemman ensimmäinen osa seuraa suoraan sen toisesta osasta k:n induktiolla.□

Lemma 1 antaa ymmärtää, että haun aikavaativuus tasapainoehdolla tasapainotetulle binääriselle hakupuulle on $O(\log\ n)$. Myös uudelleen tasapainottaminen lisäyksen tai poiston jälkeen saadaan tehtyä aikarajassa $O(\log\ n)$. Olennaiset toiminnot

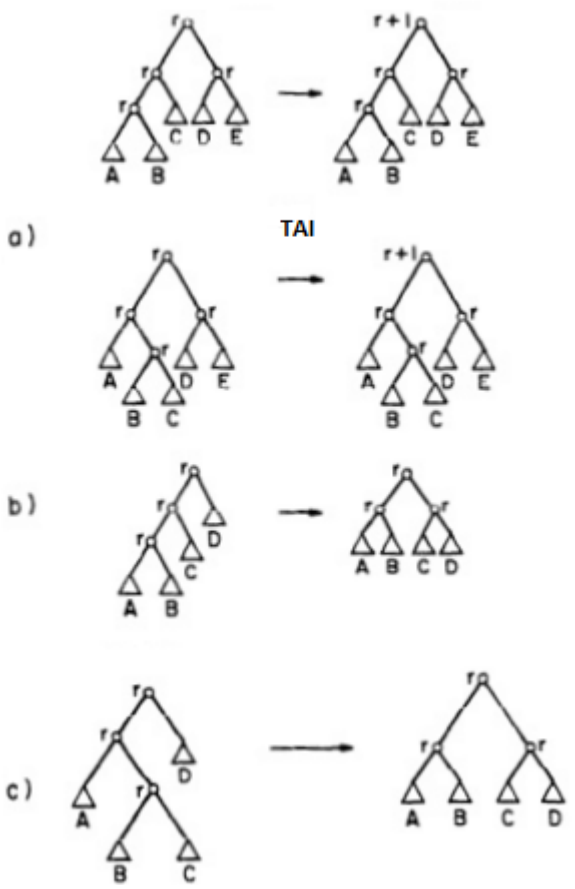
hakupuuta uudelleen tasapainotettaessa uuden solmun lisäämisen jälkeen ovat *ylennys*, joka korottaa solmun arvoastetta yhdellä, *yksinkertainen kierto* ja *tuplakierto*(joka vastaa kahta yksinkertaista kiertoa). Kukin näistä operaatioista vie ajan $O(1)$ onnistuessaan.



Kuva 2.2: Lisäys- ja poisto-operaation ydintoiminnot. Kolmiot kuvaavat alapuita. a) ylennys/alennus. b) Yksinkertainen kierto. c) Tuplakierto, jolle löytyy myös symmetrinen vastine.

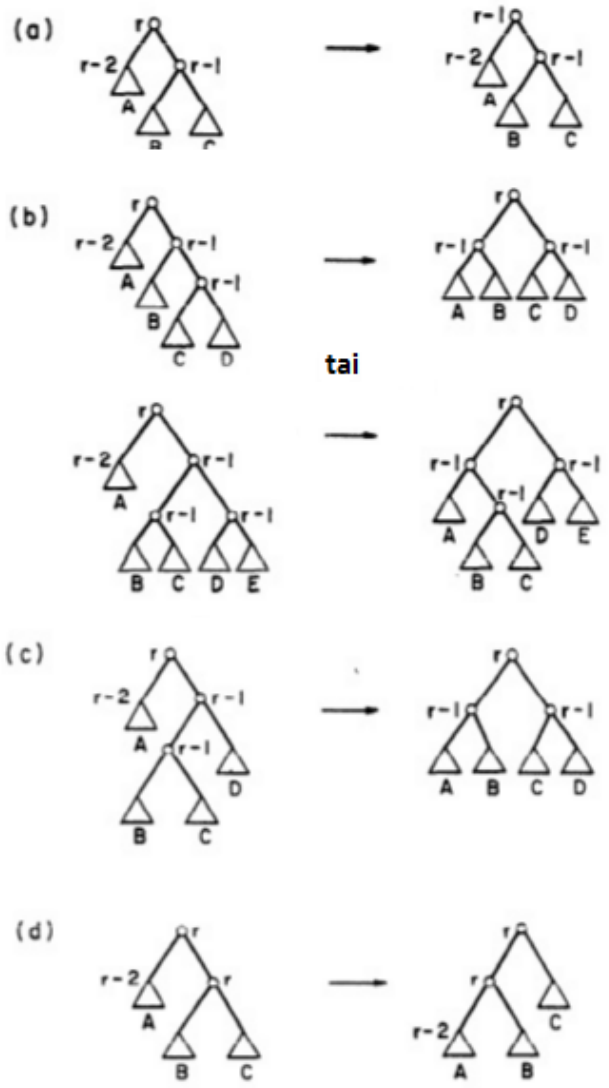
Uuden solmun lisääminen hakupuuhun tapahtuu korvaamalla jo puussa olevan nollasolmun, jonka arvoaste on nolla, hakupuuhun lisättävällä solmulla, jonka arvoaste on yksi ja jonka molemmat lapset ovat nollasolmuja. Tämä saattaa olla ristiriidassa ominaisuuden *ii* kanssa, koska mahdollinen lopputulos on punainen solmu, jonka vanhempi olisi myös punainen. Uudelleen järjestääksemme hakupuun, tutkimme lisätyn solmun x mustan isovanhemman $p^2(x)$ lapsisolmuja. Jos molemmat lapsisolmut ovat punaisia, niin korotamme isovanhemman $p^2(x)$ numeroarvoa yhdellä, jolloin sen lapsisolmut muuttuvat mustiksi, ja siirrymme lisätystä solmusta x sen isovanhempaan $p^2(x)$ kokeilemaan toteutuuko nyt sille ominaisuus *ii*. Tai jos isovanhemmalla $p^2(x)$ on musta lapsisolmu, niin suoritamme tarvittavan yksinkertaisen tai tuplakier-

ron, mikä viimeistelee hakupuun tasapainotuksen. Tällä metodilla hakupuun uudelleen tasapainottaminen lisäyksen jälkeen on aikavaativuudeltaan $O(\log\ n)$, koostuen sarjasta ylennyksiä, joita seuraa enintään kaksi yksinkertaista kiertoa.



Kuva 2.3: Lisäysoperaation askeleet. Jokaiselle on olemassa symmetrinen vastine. Kolmiot kuvaavat alapuita, joilla on mustat juuret. Nyt solmu x on alapuun B juurisolmun vanhempi. a) Ylennys. Huomaa kuinka ylennettävän solmun lapsisolmut vaihtavat väriä. b) Yksinkertainen kierto. c) Tuplakierto.

Solmun poistaminen tapahtuu kuten tavallisella binäärisellä hakupuulla. Poisto-operaation jälkeen joudumme kuitenkin tasapainottamaan hakupuun uudelleen, mihin käytämme ylennyksen sijasta *alentamista*, joka alentaa solmun arvoastetta yhdellä. Poisto-operaation jälkeen voi nimittäin olla, että poistetun solmun(Huomaa, että poistettava solmu ei välttämättä ole sama kuin solmu, joka sisältää poistetta-



Kuva 2.4: Poisto-operaation askeleet. Jokaiselle on olemassa symmetrinen vastine. Nyt solmu x on alapuun A juurisolmu ja kaikkien alapuiden, paitsi A:n, juurisolmut ovat mustia. a) Alatapaus 1a. Alennus. b) Alatapaus 1b. Yksinkertainen kierto. c) Alatapaus 1c. Tuplakierto. d) Tapaus 2. Yksinkertainen kierto, joka johtaa poisto-operaation päättävään tapauksen 1 alatapaukseen.

van arvon.) tilalle luotu uusi musta nollasolmu x on nyt vanhempaansa $p(x)$ arvoasteeltaan kaksi pienempi, mikä rikkoo ominaisuutta i. Saavuttaaksemme uuden tasapainon puussa, noudatamme seuraavanlaisia ehtoja:

Tapaus 1. Solmun x sisar y on musta.

Alatapaus 1a. Solmun y molemmat lapset ovat mustia. Alenna $p(x)$, siirry solmusta x solmuun $p(x)$, ja kokeile ominaisuuden i toteutuminen.

Alatapaus 1b. Solmun y lapsi, joka on kauimpana solmusta x, on punainen. Suorita yksinkertainen kierto solmulle $p(x)$ ja lopeta.

Alatapaus 1c. Solmun y lapsi, joka on lähimpänä solmua x, on punainen, ja sen sisar on musta. Suorita tuplakierto solmulle $p(x)$ ja lopeta.

Tapaus 2. Solmun x sisar y on punainen ja solmun y molemmat lapsisolmut ovat mustia ominaisuuden i mukaisesti. Suorita yksinkertainen kierto solmuille x ja $p(x)$ ja jatka kuten tapauksessa 1. (Kierron jälkeen solmun x uusi sisar on musta.) Nyt alatapaus 1a ei voi aiheuttaa ominaisuuden i kanssa ristiriitaa, sillä kaikki tapauksen 1 alatapaukset päättävät tasapainotuksen. Siten hakupuun uudelleen tasapainottamisen aikavaativuus poiston jälkeen on $O(\log\ n)$, koostuen sarjasta alennuksia, joita seuraa enintään kolme yksinkertaista kiertoa.

Tasapainotetulla binäärisellä hakupuulla yksittäisen satunnaisen haun pahimman tapauksen aikavaativuus on siis O(log n), joka on binääriselle hakupuulle pienin mahdollinen. Täten sarjalle perättäisiä täysin satunnaisia hakuja on paras mahdollinen binäärinen hakupuu tasapainotettu binäärinen hakupuu. Käytännössä sarjassa perättäisiä hakuja haut eivät ole kuitenkaan lähes koskaan täysin satunnaisia, vaan jotkut niistä esiintyvät toisia useammin, jolloin hakupuun kokonaisaikavaativuuden kannalta yksittäisen pahimman tapauksen aikavaativuutta olennaisempaa onkin se, kuinka usein niitä joudutaan kohtaamaan. Eli toisin sanoen, kuinka usein suoritettavan sarjan haut joudutaan ulottamaan puun juuresta aina sen kaukaisimpiin solmuihin.[Tar83]

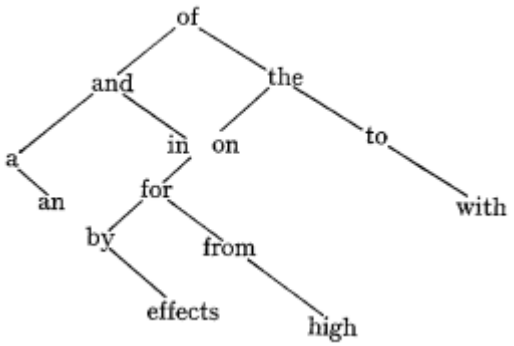
2.2 Optimoitu binäärinen hakupuu

Jos hakupuulle suoritettavassa sarjassa hakuja kunkin yksittäisen suoritettavan haun todennäköisyys toistua on ennalta tiedettävissä, voidaan käytön kokonaisaikavaativuutta yrittää minimoida sijoittamalla kaikki hakupuun useimmin haetut solmut mahdollisimman lähelle sen juurta.

Otetaan esimerkkinä seuraavanlainen lista sanoja ja sarjoja:

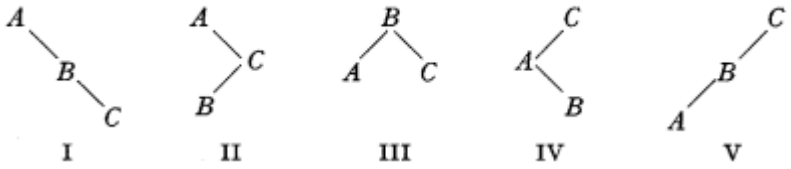
a	32	for	15	on	22
an	7	from	10	the	79
and	69	high	8	to	18
by	13	in	64	with	9
effects	6	of	142		

Näille paras mahdollinen binäärinen hakupuu on seuraavanlainen:

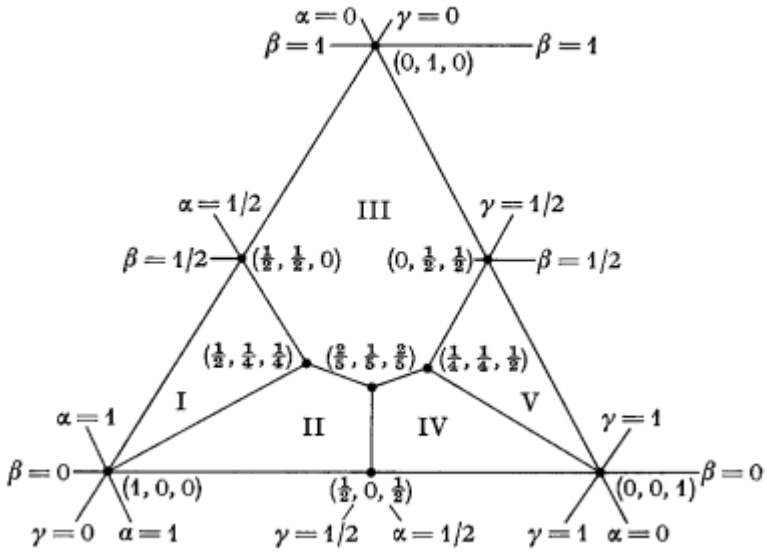


Kuten voimme kuvasta huomata, niin ei paras mahdollinen binäärinen hakupuu ole rakenteeltaan välttämättä aina tasapainotettu. Itseasiassa optimoidun, eli tietylle sarjalle hakuja parhaan mahdollisen, hakupuun ongelma on huomattavasti monimutkaisempi kuin perinteinen *Huffmanin koodaus*, jossa useimmin haetut sanat laitetaan vain raa'asti aina suoraan puun juureen [Huf52].

Oletetaan, että sanamme ovat A, B, C ja niiden toistuvuudet ovat α, β ja γ. Nyt voimme muodostaa näille viisi rakenteeltaan erilaista binääristä hakupuuta, joissa jokaisessa on kolme solmua.



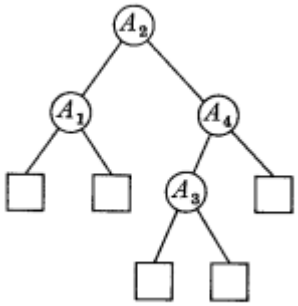
Seuraava diagrammi näyttää α:n, β:n ja γ:n arvot, joilla kukin näistä puista on optimaalinen, olettaen, että $\alpha+\beta+\gamma = 1$.



Kuten voimme kuvasta nyt nähdä, on olemassa myös tapauksia, joissa optimaalisuuden kannalta onkin parasta laittaa B juureen, vaikka molemmat A ja C toistuisivat hauissa sitä useammin. Ei ole myöskään riittävää valita juurta siten, että se tasapuolistaisi vain oikeaan ja vasempaan alapuuhun kohdistuvien hakujen todennäköisyyksiä niin paljon kuin mahdollista. Näistä johtuen optimoidun hakupuun uudelleen optimoiminen jokaisen lisäys- ja poisto-operaation jälkeen onkin hyvin haasteellista, sillä n-solmuiselle binääriselle hakupuulle on olemassa $\binom{2n}{n}\frac{1}{n+1}$ erilaista rakennetta, mikä tekee niiden kaikkien läpikäymisen parasta optimaalista binääristä hakupuun muotoa etsittäessä mahdottomaksi. Siispä optimoidun binäärisen hakupuun löytämiseen onkin kehitetty erilaisia algoritmeja.

Käytännössä haluamme yleistää ongelman, ei vain koskemaan hakuja joilla saamme onnistuneen haun, vaan myös koskemaan hakuja jotka ovat epäonnistuneita. Siten saamme n nimeä A_1, A_2, ..., A_n ja $2n{+}1$ toistuvuutta α_0, α_1, ..., α_n; β_1, β_2, ..., β_n. Tässä β_i on toistuvuus, jolla kohtaamme A_i, ja α_i on toistuvuus jolla kohtaamme nimen, joka on A_i ja A_{i+1} välillä; huomaa myös arvojen α_0 ja α_n tulkinnat. Avainasia, joka tekee ongelmasta suotuisan dynaamiselle ohjelmoinnille, on että kaikki optimaalisen alapuun alapuut ovat myös optimaalisia. Jos A_i on juuressa, niin silloin sen vasen alapuu on optimi ratkaisu sarjalle α_0, ..., α_{i-1} ja β_1, ..., β_{i-1} ja oikea alapuu optimaalinen ratkaisu sarjalle α_i, ..., α_n ja β_{i+1}, ..., β_n. Siispä voimme rakentaa optimaalisen hakupuun kaikille eri sarjojen intervalleille α_i, ..., α_j ja β_{i+1}, ..., β_j, kun $i \leq j$, aloittaen pienimmistä intervalleista edeten kohti suurinta. Koska tapaukselle $0 \leq i \leq j \leq n$ on vain $\frac{(n+2)(n+1)}{2}$ vaihtoehtoa, niin kokonaismäärä laskennalle ei ole liiallinen.

Otetaan seuraavanlainen binäärinen hakupuu, jossa puun neliöt kuvaavat sen lopetussolmuja, joihin ei ole talletettuna nimeä:



Haun painotettu pituus binääriselle hakupuulle P on sen kaikkien solmujen tason ja todennäköisyyden tulon yhteenlaskettu kokonaissumma. Näin haun painotetuksi pituudeksi esitetylle hakupuulle saadaan

$$3\alpha_0 + 2\beta_1 + 3\alpha_1 + \beta_2 + 4\alpha_2 + 3\beta_3 + 4\alpha_3 + 2\beta_4 + 3\alpha_4.$$

Yleisesti ottaen, voimme nähdä, että haun painotettu pituus täyttää kaavan

$$P{=}P_L{+}P_R{+}W$$

ehdot, missä P_L ja P_R ovat oikean ja vasemman alapuun hakujen painotetut pituudet ja $W{=}\alpha_0{+}\alpha_1{+}...{+}\alpha_n{+}\beta_1{+}...{+}\beta_n$ on puun "paino", eli kaikkien toistuvuuskertoimien summa. Haun painotettu pituus kuvaa relatiivista työmäärää hauille, jotka suoritetaan kaikille valituille arvoille α ja β. Täten optimaalisen hakupuun löytämisen ongelma on ongelma löytää binääripuu, jolla on mahdollisimman pieni yhteenlaskettu hakujen painotettu pituus, kun painot huomioidaan hakupuussa vasemmalta oikealle.

Yllä tehdyt huomiot johtavat suoraan seuraavanlaiseen laskutapaan selvittää optimoitu binäärinen hakupuu. Olkoot P_{ij} ja W_{ij} haun painotettu pituus ja optimipuun kokonaispaino kaikille sanoille, jotka ovat välillä A_i ja A_{i+1}, kun $i{<}j$; ja olkoon R_{ij} tämän puun juuren indeksi, kun $i{<}j$. Seuraavanlaiset kaavat muodostavat nyt haluamamme algoritmin:

$$P_{ii} = W_{ii} = \alpha_i, \text{ kun } 0 \leq i \leq n,$$

$$W_{ij} = W_{i,j-1} + \beta_j + \alpha_j,$$

$$P_{i,R_{i,j-1}} + P_{R_{ij},j} = \min_{i<k\leq j}(P_{i,k-1} + P_{kj}) = P_{ij} \text{ - } W_{ij}, \text{ kun } 0 \leq i < j \leq n.$$

Tarkastelemalla esitettyä algoritmia huomaamme, että optimaalisen binäärisen hakupuun löytämiselle on laskettavissa aikavaativuuden yläraja $O(n^3)$. Tästä on mahdollista vielä jalostaa tehokkaampi $O(n^2)$ laskutapa optimaaliselle binääriselle hakupuulle, kuten Knuth on esittänyt[Knu70].

Vaikka optimoitu binäärinen hakupuu on ennalta tunnetulle sarjalle hakuja kaikista binääripuista paras mahdollinen, tekee kuitenkin sen jokaisen lisäys- ja poistooperaation tarvitsema koko puun jälleenrakentaminen siitä erittäin raskaan käytettävän. Oikean elämän sovelluksissa on myös hyvin haasteellista määrittää ennalta

kaikkia hakupuun optimointiin tarvittavia todennäköisyyksiä, sillä usein hakupuun eri käyttötapaukset korreloivat keskenään, jolloin suoritettavat sarjat hakuja saattavat poiketa hyvinkin paljon toisistaan. Muutokseen sopeutumiskyvyltään surkealle optimoidulle hakupuulle onkin olemassa muita vaihtoehtoisia binäärisiä hakupuita, joista monet pyrkivät samankaltaiseen optimaalisuuteen, mutta kuitenkin säilyttäen kykynsä mahdollisimman nopeisiin rakenteen muutoksiin.

2.3 Ennakoiva binäärinen hakupuu

Ennakoiva binäärinen hakupuu on puurakenne, joka pyrkii yhdistämään sekä tasapainotetun että optimoidun binäärisen hakupuun hyvät puolet. Se muistuttaa rakenteellisesti hyvin paljon tasapainotettua binääristä hakupuuta, yrittäen pitää korkeutensa aina mahdollisimman pienenä. Mutta sitä päivittävät algoritmit ovat monimutkaisempia kuin tasapainotetun binäärisen hakupuun, sillä se pyrkii tasapainotuksen lisäksi myös optimoidun hakupuun kaltaisesti pitämään useimmin haetut arvonsa nopeammin saatavilla kuin harvemmin haetut, järjestämällä ne lähemmäksi juurta. Olennainen ero optimoituihin hakupuihin on kuitenkin ennakoivan binäärisen hakupuun kyky sopeutua muutoksiin, sillä sen päivitysnopeus on tasapainotetun binäärisen hakupuun kaltaisesti vain $O(\log\ n)$, siinä missä optimoitu binäärinen hakupuu joutuu jälleenrakentamaan itsensä aina kokonaan uudelleen jokaisella tehdyllä muutoksella ajassa $O(n^3)$ (tai ajassa $O(n^2)$ Knuthin esittämällä algoritmilla).

Hyvänä esimerkkinä havainnollistaaksemme ennakoivaa binääristä hakupuuta on ottaa tässä tutkielmassa aiemmin käsitelty punamusta hakupuu ja lisätä sille optimoivaa ominaisuutta. Punamusta hakupuu on itsessään tasapainotettu binäärinen hakupuu, jolle on määritelty tietyt ehdot uudelleen tasapainottumiselle, joka suoritetaan aina jokaisen rakenteeseen tehdyn muutoksen jälkeen. Nämä ehdot järjestävät puun siten, että sen pahin mahdollinen aikavaatimus yksittäiselle haulle on sen koko

korkeus $O(\log\ n)$. Punamusta hakupuu ei kuitenkaan ota huomioon rakenteen uudelleen järjestämisessä millään tavalla tulevia suoritettavia hakuja, vaan on täysin sokea omalle käytölleen tahtoen vain pitää hakupuun rakenteen tasapainossa.

Siispä yhdistämme punamustaan hakupuuhun optimaalisesta hakupuusta tutun laskutavan, jossa jokaiselle solmulle on olemassa kerroin, joka vastaa sen haun todennäköisyyttä. Nyt voimme laskea punamustan hakupuun solmuille uudenlaiset arvoasteet ja muuttaa sen uudelleen järjestävää algoritmia siten, että tasapainotus huomioi myös kuhunkin solmuun kohdistuvan haun todennäköisyyden. Yksinkertaisimmillaan siis yritämme vain hieman muuttaa punamustan hakupuun omaa toimintalogiikkaa siten, että tasapainotusalgoritmi siirtäisi useimmin haettuja solmuja lähemmäs puun juurta.[BST85]

Ennakoivien binääristen hakupuiden suuri ongelma on kuitenkin niiden vielä optimoituja hakupuitakin paljon hankalammat algoritmit ja vielä tasapainotettuja hakupuitakin monimutkaisemmat ja vaikeammin ylläpidettävät ehdot. Ennakoiville binäärisille hakupuille onkin olemassa vaihtoehtoisia hakupuurakenteita, jotka lähestyvät hakupuun optimointiongelmaa aivan uudella tavalla.

2.4 Tasoitettu analyysi

Usein binäärisen hakupuun aikavaativuutta tarkasteltaessa sille pyritään esittämään yläraja pahimpien mahdollisten käyttötapauksien summana. Näin laskettu aikavaativuuden yläraja on monesti kuitenkin jo liioitellun pessimistinen, eikä se usein huomioi ollenkaan niitä positiivisia ominaisuuksia, joita yksittäiset pyynnöt voivat saada aikaan hakupuun rakenteessa. Aikavaativuuden määrittämistä ei kannata myöskään yrittää lähestyä määrittämällä jokaiselle yksittäiselle pyynnölle keskimääräistä aikavaativuutta lähinnä sen haastavuuden ja hyvin helpon oletuksissa erehtymisen takia.

Sen sijaan binääristen hakupuiden aikavaativuuden analysoimiseen on kehitetty ihan oma metodinsa; niin kutsuttu *tasoitettu analyysi*. Tasoitetussa analyysissa jokaiselle yksittäiselle operaatiolle lasketun keskimääräisen suoritusajan sijaan muodostamme yhden yleisen keskiarvon, jolla kuvaamme jokaista yksittäistä hakupuulle suoritettavaa pyyntöä perättäisessä sarjassa aikavaativuudeltaan raskaimpia mahdollisia operaatioita. Näin saadut laskelmat antavat meille sekä vahvoja että realistisia tuloksia.

Havainnollistaaksemme tasoitettua analyysia, oletamme että meillä on keko, jolle voidaan suorittaa kahta erilaista operaatiota: *push* ja *pop*. Push lisää keon päälle aina yhden uuden arvon, ja vastaavasti pop poistaa keosta aina siihen viimeisimpänä lisätyn arvon. Pop-operaatiota ei voida suorittaa, jos keko on tyhjä. Analysoimme keolle suoritettavan sarjan pyyntöjä aikavaativuutta, kun jokaista push-operaatiota voi seurata aina nolla tai useampi pop. Oletetaan, että keko on aloittaessamme tyhjä, ja että suoritamme sille määrän m operaatioita. Nyt yhdelle yksittäiselle operaatiolle voidaan määrittää aikavaativuuden yläraja $O(m)$, mikä seuraa jos kaikki m-1 ensimmäistä operaatiota ovat olleet pelkästään push, ja näiden jälkeen viimeisenä suoritettava operaatio on push ja m kertaa pop, poistaen keosta myös kaikki aiempien push-operaatioiden siihen lisäämät arvot. Näin saatua yksittäisen operaation pahimman tapauksen aikavaativuuden ylärajaa hyödyntäen, voidaan edelleen laskea koko suoritettavalle sarjalle operaatioita aikavaativuuden yläraja $O(m^2)$.

Näin laskettu aikavaativuuden yläraja on kuitenkin selkeästi liioitellun pessimistinen, sillä se jättää täysin huomiotta sen keon rakenteellisen ominaisuuden, että jokaista pop-operaatiota on täytynyt vastata aina vähintään yksi aiempi push-operaatio, mikä rajaa suoritettavien operaatioiden yhteenlasketuksi lukumääräksi aina enimmilläänkin $2m$. Siispä saadaksemme tarkempia tuloksia analysoidessamme binääristen hakupuiden aikavaativuuksia, tulee meidän huomioida sarjassa suoritettavien operaatioiden vaikutukset toisiinsa. Näin ollen analysoimme tästedes binääristen hakupuiden ominaisuuksia tasoitetulla analyysillä.[Tar85]

2.5 Mukautuva binäärinen hakupuu

Mukautuva binäärinen hakupuu on hakurakenne, joka jokaisen yksittäisen operaation aikavaativuuden minimoimisen sijaan pyrkii aina uudelleen järjestämään rakennettaan siten, että kaikki kutakin sille suoritettavaa operaatiota seuraavat operaatiot olisivat nopeampia. Sillä voi siis olla yksittäisiä aikavaativuudeltaan hyvinkin kalliita operaatioita, mutta tasoitetulla analyysilla tarkasteltaessa sen keskimääräinen aikavaativuus kullekin operaatiolle on kuitenkin pieni.

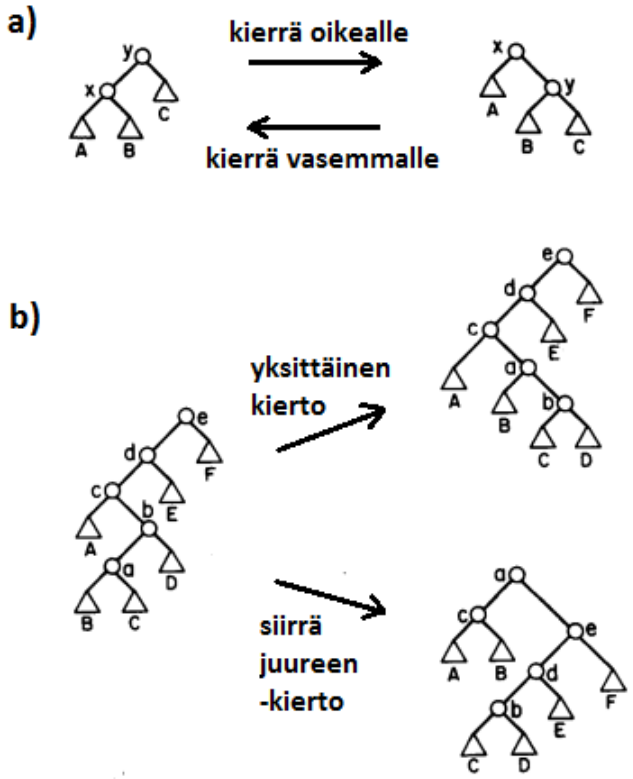
Toisin kuin ennakoivat ja optimoidut hakupuut, mukautuva binäärinen hakupuu ei myöskään tarvitse sille suoritettavista operaatioista ollenkaan mitään ennakkotietoja, vaan sen tehokkuus perustuu olettamukseen tiedonhaun *klusteroitumisesta*, eli siitä, että jokaista usein haettua arvoa tullaan todennäköisesti hakemaan pian uudestaan. Käytännössä tämä tarkoittaa sitä, että mukautuva binäärinen hakupuu on rakenteeltaan aina ennalta määräämättömässä muodossa, mutta kuitenkin siten, että jokaisella sille suoritettavalla haulla se uudelleenjärjestää haetun arvon sen uudeksi juureksi olettaen, että samaa arvoa haetaan pian uudestaan.

Mukautuvan binäärisen hakupuun uudelleen järjestävät algoritmit perustuvat erilaisiin kiertoihin, jotka muokkaavat sen rakennetta jokaisella operaatiolla:

- Yksittäinen kierto: Haku arvoon i solmussa x. Kierrä solmua x sen vanhemman kanssa. (Paitsi jos solmu x on hakupuun juuri.)
- Siirrä juureen -kierto: Haku arvoon i solmussa x. Kierrä solmua x sen vanhemman kanssa, ja toista kunnes solmu x on hakupuun uusi juuri.

Mukautuvan binäärisen hakupuun etuja *staattisiin*, rakenteeltaan muuttumattomiin, hakupuihin nähden on erityisesti sen kyky sopeutua sille suoritettaviin sarjoihin operaatioita. Tasoitetulla analyysilla tarkasteltaessa mukautuva hakupuu ei olekaan ikinä paljon staattisia hakupuita huonompi, vaan joillain optimaalisilla syöt-

teillä se saattaa olla jopa huomattavasti niitä tehokkaampi. Myös sen laitteelta vaatima muistitila toimiakseen on pienempi, sillä esimerkiksi tasapainotukseen tarvittavaa muistitilaa ei tarvita ja sen haku-, lisäys- ja poistoalgoritmit ovat yksinkertaisia ja helppoja implementoida.



Kuva 6.1: Mukautuvan binäärisen hakupuun kierrot. Kolmiot kuvaavat alapuita. Kuvien puut voivat olla myös suuremman puun alapuita. Jokainen yksittäinen kierto on aikavaativuudeltaan O(1) ja säilyttää solmujen symmetrisen järjestyksen. a) Kierretään solmuja x ja y keskenään. b) Mukautuvan hakupuun kierto solmussa a.

Tavallisen mukautuvan binäärisen hakupuun heikkoutena on kuitenkin sen paikalliset muokkaukset, joita on paljon erityisesti hakuoperaation aikana, kun taas staattiset hakupuut vaativat muokkausta vain uuden solmun lisäyksen tai poiston aikana. Mukautuvalle hakupuulle voi myös olla joskus yksittäisiä hakuja, joilla rakenteita

muokkaavat kierrot ovat hyvinkin raskaita, mikä saattaa näkyä reaaliaikaisissa sovelluksissa huomattavana heikkoutena. Tasoitetulla analyysilla tarkasteltaessa esittämäämme hakupuuta muokkaavat kierrot eivät olekaan erityisen tehokkaita, sillä on olemassa satunnaisen pitkiä sarjoja pyyntöjä, joilla aikavaatimus jokaista yksittäistä pyyntöä kohden on $O(n)$.[Slt85]

2.6 Splay-puu

Splay-puu on Daniel Sleatorin ja Robert Tarjanin vuonna 1985 kehittämä tavallista mukautuvaa binääristä hakupuuta hyvin paljon muistuttava hakupuu, mutta jonka poikkeukselliset uudelleen järjestävät kierrot takaavat sille kuitenkin paremman tehokkuuden kuin tavalliselle mukautuvalle hakupuulle. Tasoitetulla analyysilla tarkasteltaessa splay-puun haku-, lisäys- ja poisto-operaatiot ovat pahimmillaankin sarjassa perättäisiä operaatioita aikavaativuudeltaan kukin keskimäärin vain $O(\log\ n)$ ja täten splay-puu on hakupuuna ainakin yhtä tehokas kuin dynaamisesti tasapainotetut ja staattiset optimoidut hakupuut. Voidaan myös olettaa, että millä tahansa riittävän pitkällä sarjalla suoritettavia operaatioita, splay-puu on aina ainakin yhtä tehokas kuin mikä tahansa dynaamisesti päivittyvä binäärinen hakupuu.

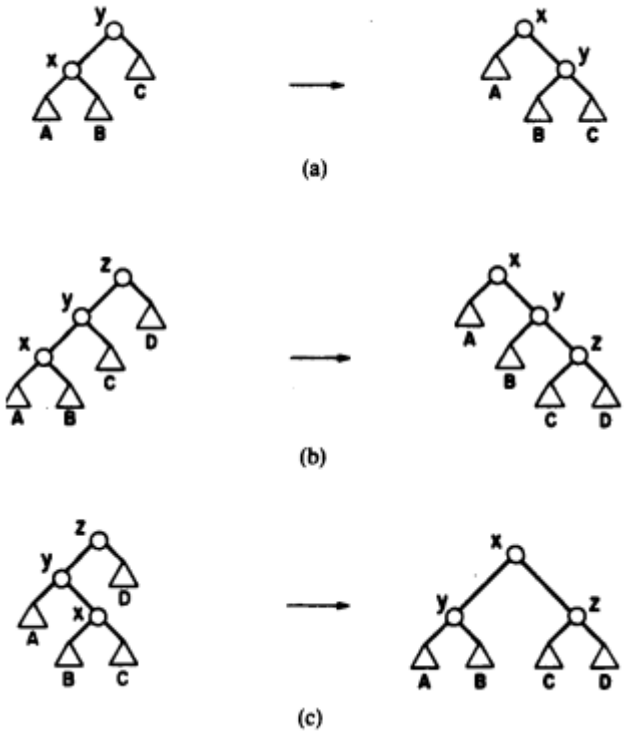
Splay-puuta uudelleen järjestävät kierrot, niin kutsutut *splay-kierrot*, poikkeavat tavallisen mukautuvan binäärisen hakupuun kierroista siinä, että ne tehdään pareittain järjestyksessä, joka riippuu haun reitistä haettuun solmuun. Epäonnistuneen haun tapauksessa splay-kierrot suoritetaan hakureitin viimeiselle solmulle. Suorittaaksemme splay-kierron solmussa x, toistamme seuraavanlaisia askelia, kunnes x on hakupuun uusi juurisolmu:

1. Zig: Jos $p(x)$, solmun x vanhempi, on hakupuun juuri, kierrä solmuja x ja $p(x)$ keskenään.

2. Zig-zig: Jos $p(x)$ ei ole hakupuun juuri, ja x ja $p(x)$ ovat molemmat vasempia

tai oikeita lapsia, kierrä solmua $p(x)$ solmun x isovanhemman $g(x)$ kanssa ja sen jälkeen vielä kierrä solmuja x ja $p(x)$ keskenään.

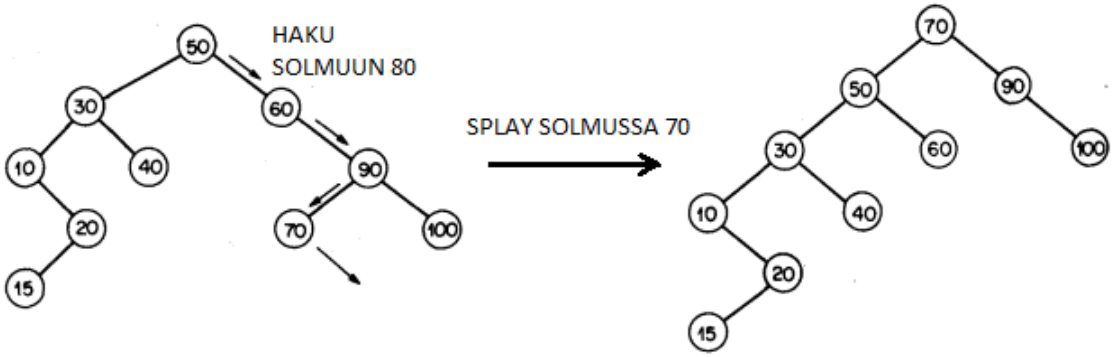
3. Zig-zag: Jos $p(x)$ ei ole hakupuun juuri, ja x on vasen lapsi ja $p(x)$ oikea lapsi, tai päinvastoin, niin kierrä solmuja x ja $p(x)$ keskenään ja sen jälkeen kierrä solmua x vielä sen uuden vanhemman kanssa.



Kuva 7.1: Splay-kierto solmussa x. Jokaisella tapauksista on symmetrinen vastine. a) Zig: splay-kierrot päättävä yksinkertainen kierto, kun solmu y on puun juuri. b) Zig-zig: Kaksi yksinkertaista kiertoa. c) Zig-zag: Tuplakierto.

Splay-kierto solmussa x syvyydessä d vie ajan $\Theta(d)$, joka vastaa hakuoperaation aikaa kulkea hakupuun juuresta sen solmuun x. Splay-kierto ei pelkästään tee solmusta x uutta hakupuun juurta, vaan karkeasti määriteltynä puolittaa jokaisen hakuoperaation reitillä olevan solmun syvyyden. Juuri tämä syvyyksien puolittaminen

zig-zig ja zig-zag -tapauksissa toisistaan eroavin kierroin on se, mikä tekee splay-kierrosta niin tehokkaan ja ainutlaatuisen, sillä mikään muu yksinkertaisempi algoritmi, kuten vaikkapa tavallisen mukautuvan hakupuun *siirrä juureen* -kierto, ei jaa sen ominaisuuksia.



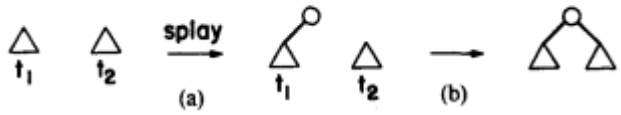
Kuva 7.2: Epäonnistunut haku arvolla 80. Kun hakuoperaatio päättyy epäonnistuneena nollasolmuun, suoritetaan splay-kierrot viimeiselle lehtisolmulle, jonka kautta haku on kulkenut.

Myös splay-puun lisäys- ja poisto-operaatiot ovat aiemmin esitellyistä hakupuista poikkeavat. Koska splay-kierroilla on ominaisuus siirtää nimetty solmu juureen, on uuden solmun lisäys ja poistaminen helpointa tehdä *liitoksella* ja *halkaisulla*:

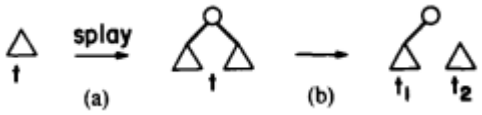
> Liitos(T_1,T_2): Yhdistää hakupuut T_1 ja T_2 toisiinsa muodostaen yhden suuremman hakupuun, joka sisältää molempen T_1 ja T_2 kaikki arvot. Ehtona liitokselle on, että kaikki hakupuun T_1 arvot ovat pienempiä kuin hakupuun T_2 arvot.
>
> Halkaisu(i, T): Halkaisee hakupuun T kahteen hakupuuhun T_1 ja T_2, joista T_1 sisältää kaikki hakupuun T arvot, jotka ovat pienempiä tai yhtäsuuria kuin i, ja joista T_2 sisältää kaikki hakupuun T arvot, jotka ovat suurempia kuin i.

Suorittaaksemme liitoksen, aloitamme nostamalla pienempiä arvoja sisältävän puun T_1 suurimman arvon i sen uudeksi juureksi suorittamalla siihen haun. Tämän jäl-

keen asetamme vain puun T_1 puun T_2 pienimmän solmun vasemmaksi lapseksi. Vastaavasti halkaisu arvolla i hakupuussa T aloitetaan haulla arvolle i, jonka jälkeen puu T_1 on puun T juurisolmu ja sen vasen alapuu ja puu T_2 on puun T juurisolmun oikea alapuu.



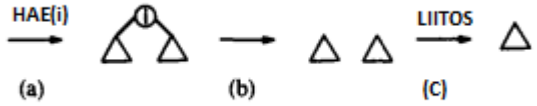
Kuva 7.3: Liitos(T_1,T_2). a) Haku puun T_1 suurimpaan arvoon. b) Aseta puu T_1 puun T_2 pienimmän solmun vasemmaksi lapseksi.



Kuva 7.4: Halkaisu(i, T). a) Haku arvolla i. b) Nyt puu T_1 on juurisolmu ja sen oikea alapuu ja puu T_2 on juurisolmun oikea alapuu.

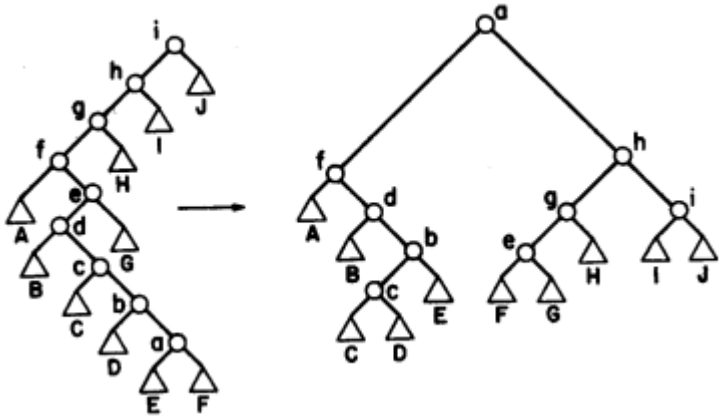
Näin saamme edelleen määriteltyä lisäyksen ja poiston seuraavanlaisesti. Uuden yksittäisen solmun lisääminen tehdään suorittamalla halkaisu lisättävällä arvolla i, jonka jälkeen lisäämme lisättävälle solmulle arvolla i näin saadut puut T_1 ja T_2 sen oikeaksi ja vasemmaksi alapuuksi. Vastaavast jo olemassa olevan solmun poistaminen tehdään suorittamalla ensin haku poistettavalla arvolla i, jonka jälkeen suoritetaan juureen nousseen arvon i sisältävän solmun oikealle ja vasemmalle alapuulle liitos.

Kuva 7.5: Lisäys arvolla i. a) Halkaisu arvolla i. b) Aseta puut T_1 ja T_2 arvon i sisältävän solmun oikeaksi ja vasemmaksi lapseksi.

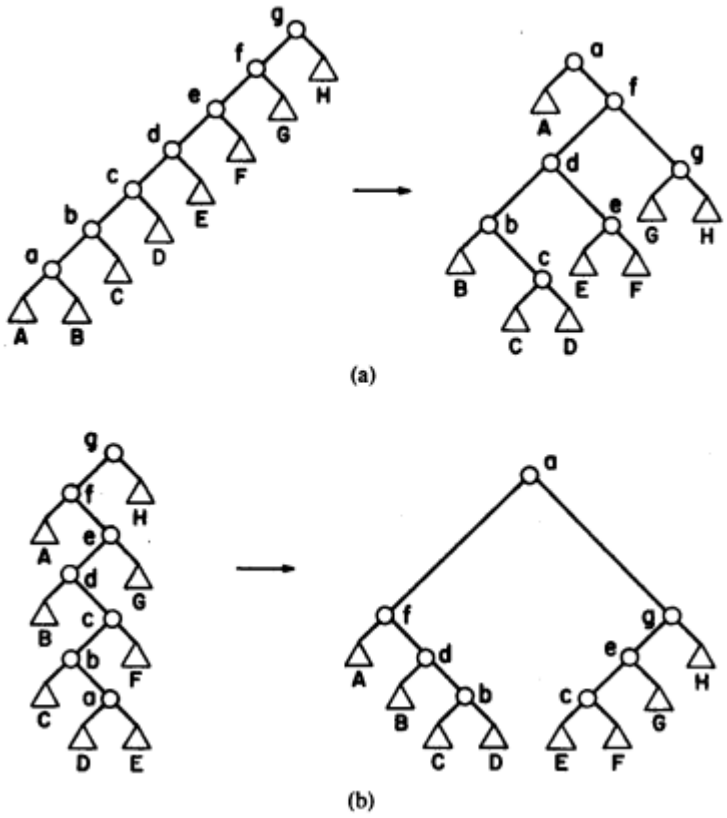


Kuva 7.6: Poisto arvolla i. a) Haku arvolla i. b) Puut T_1 ja T_2 ovat juurisolmun oikea ja vasen alapuu. c) Liitos puille T_1 ja T_2.

Kuten tavallisella mukautuvalla binäärisellä hakupuulla, myös splay-puulla on kuitenkin olemassa yksittäisiä hakuja, joista seuraa suuri määrä perättäisiä uudelleen järjestäviä kiertoja, mikä saattaa näkyä huomattavanakin heikkoutena sen reaaliaikaisessa käytössä. Yksi tähän kehitelty ratkaisu on suorittaa splay-askel vain osittain kullakin suoritettavalla haulla. Tämä pienentää pahimpien yksittäisten tapausten aikavaativuutta, mutta samalla tasoitetulla analyysilla tarkasteltaessa koko puun käytön aikavaativuus kuitenkin hieman kasvaa.



Kuva 7.7: Tavalliset splay-kierrot suoritettaessa haku solmuun a.



Kuva 7.8: Ääriesimerkkejä splay-kierroista suoritettaessa haku solmuun a. a) Kaikki zig-zig-kiertoja. b) Kaikki zig-zag-kiertoja.

3 Yhteenveto

Binääriset hakupuut ovat pääsääntöisesti helposti muokattavia tietorakenteita, jotka pystyvät takaamaan kohtuullisen aikavaativuuden ylärajan niiden pahimmillekin mahdollisille käyttötapauksille. Niihin tallennettu tieto on myös aina järjestetyssä muodossa, jolloin suurimpien tai pienimpien arvojen tai tiettyjen arvovälien hakeminen niistä on hyvin yksinkertaista ja tehokasta. Ne ovat myös helppoja ylläpidettäviä muistinhallinnan näkökulmasta, sillä niiden vaatima muistitila ei ole yhtenäinen, vaan solmut on mahdollista hajauttaa pitkin kiintolevyä. Tämän lisäksi myös vaikka hakupuun itsensä rakenne muuttuisi, niin ei kiintolevyltä varattuja muistilohko-

ja välttämättä tarvitse muuttaa, sillä riittää vain päivittää kunkin solmun muihin solmuihin osoittavat osoittimet.

Jos tietorakenteen pääasiallinen käyttö koostuu kuitenkin enimmäkseen vain haku-, lisäys- ja poisto-operaatioista, on binääriselle hakupuulle kuitenkin olemassa parempiakin vaihtoehtoja. Esimerkiksi *hajautustaulu* (engl. *hash table*) pystyy tarjoamaan näille käyttötapauksille keskimääräisen aikavaativuuden $O(1)$.[CLR09b]

4 Lähteet

[AdL62] Georgy Adelson-Velsky ja Evgenii Landis, An algorithm for the organization of information, Teoksessa *Soviet Mathematics Doklady*, 3, sivut 1259-1263.

[BST85] Samuel W. Bent, Daniel D. Sleator ja Robert E. Tarjan, Biased Search Trees, Julkaisussa *SIAM Journal on Computing*, Vol. 14, Nro. 3, Elokuu 1985, sivut 545-568.

[CLR09a] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest ja Clifford Stein, 12: Binary search trees. Teoksessa *Introduction to Algorithms*, 3. painos, The MIT Press, 2009, sivut 286-298.

[CLR09b] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest ja Clifford Stein, 11: Hash Tables. Teoksessa *Introduction to Algorithms*, 3. painos, The MIT Press, 2009, sivut 256.

[Huf52] D. A. Huffman, A Method for the Construction of Minimum-Redundancy Codes, Julkaisussa *Proceedings of the I.R.E.*, September 1952, sivut 1098-1102.

[Knu70] D. E. Knuth, Optimum Binary Search Trees, Julkaisussa *Acta Informatica*, 1, 1971, sivut 14-25

[SlT85] Daniel D. Sleator ja Robert E. Tarjan, Self-Adjusting Binary Search Trees, Julkaisussa *Journal of the Association Machinery*, Vol. 32, Nro. 3, 1985, s. 685.

[Tar83] Robert Endre Tarjan, 4.2. Balanced binary trees, Teoksessa *Data Structures and Network Algorithms*, Bell Laboratories, New Jersey, 1983, sivut 48-53.

[Tar85] Robert Endre Tarjan, Amortized Computational Complexity, Julkaisussa *Journal of the ACM*, Bell Laboratories, New Jersey, 1985, s. 306.