
Tietorakenteet ja algoritmit

Hakurakenteet

Ari Korhonen, Lauri Malmi

9. HAKURAKENTEET (dictionaries)

9.1 Etsintä

9.2 Listat tallennusrakenteina

9.3 Taulukko tallennusrakenteena

9.4 Binäärinen hakupuu

9.5 Tasapainotetut puut (balanced trees)

9.6 Digitaaliset avaimet

9.1 Etsintä

Etsintä on tietojenkäsittelyn perusongelmia, joka voidaan määritellä seuraavasti:

- **Tiedot on esitetty *tietorakenteessa*, joka koostuu tietueista**
- **Kussakin tietueessa on *avain (key)*, jonka avulla se voidaan tunnistaa**
- **Etsinnässä pyritään löytämään tietorakenteesta kaikki tietueet, joilla on annettu avain**
- **Luonnollisesti asiaan liittyy lisäysten, poistojen ja päivitysten tekeminen tietorakenteeseen**
- **Ongelmaan tunnetaan ratkaisuina erilaisia hakurakenteita**

Esimerkkejä käytännön etsintäongelmista:

- **Hae taulukosta annettu alkio**
- **Hae elektronisen sanakirjan sana**
- **Hae pankkitilin saldo**
- **Hae opintosuoritusrekisteristä opiskelijan suoritustiedot**
- **Hae opintosuoritusrekisteristä tietyn kurssin suorittaneiden opiskelijoiden nimet**
- **Etsi ne artikkelit, joissa esiintyy tämä hakusana**

Etsintää varten on hyödyllistä määritellä abstrakti tietotyyppi *hakurakenne (dictionary)*, johon liittyy ainakin seuraavia toimintoja:

- **Alusta hakurakenne**
- **Hae avainta vastaava(t) tietue(et)**
- **Lisää uusi tietue**
- **Poista avainta vastaava(t) tietue(et)**
- **Yhdistä kaksi hakurakennetta**
- **Tulosta hakurakenteessa olevat tietueet avainten perusteella lajiteltuna**
- **Lisäys ja poisto sisältävät yleensä osanaan hakutoiminnon**

Muita toimintoja, joita voidaan tarvita:

- **Hae tietue, jolla on lähinnä seuraava / edellinen avainarvo (*naapurikysely, neighbour query*)**
- **Hae tietueet, joiden avainarvo on tietyllä välillä (*aluekysely, range query*)**

Rakenteessa olevat *duplikaatit* (*duplicates*, tietueet, joilla on sama avainarvo) voidaan käsitellä eri tavoin:

- **kukin erillisenä tietueena**
- **yksi tietue ja siitä lähtevä osoitin toiseen rakenteeseen**
- **yksi tietue ja siinä laskuri**

Hakurakenne voidaan implementoida hyvin monella tavalla (eri käsitteellisten tietotyypin avulla). Asiaan liittyviä näkökulmia ovat ainakin:

- **tiedon määrä**
- **hakuihin, lisäyksiin ja poistoihin liittyvät aikavaatimukset**
- **talletetaanko rakenne keskusmuistiin vai massamuistiin?**
- **lisäysten ja poistojen määrä**
- **tarvitaanko erityyppisiä hakuavaimia?**
- **tarvitaanko operaatioita, joissa tietoa käsitellään järjestyksessä?**
- **menetelmän monimutkaisuus**

Joitakin sovellusalueita:

- **yksinkertaiset kortistot**
- **symbolitaulut**
- **tietokantojen hakemistot**
- **tiedostorakenteet**
- **CAD**
- **Tietokonegrafiikka**

Käsiteltävät hakurakenteet voidaan rinnastaa myös alkeellisiin *tietokantoihin*:

- **vain yksi hakuavain**
- **vain yksi käyttäjä**
- **ei suojaus- eikä toipumismekanismia**
- **ei yhtenäisyystarkistusta**

Käytännön *tiedonhallintajärjestelmissä* nyt käsiteltävät asiat liittyvät tietokannan indeksien toteutukseen.

9.2 Listat tallennusrakenteina

Joukko yksinkertaisia hakurakenteita, joille ovat yhteisiä seuraavat asiat:

Hakuoperaatio on lineaarinen, mistä syystä rakenne soveltuu vain pieniin sovelluksiin ($N < 100$)

Lisäykset ja poistot on helppo toteuttaa

Voidaan toteuttaa linkitettyinä listana tai listana taulukossa

Muunnelmia:

Järjestämätön lista

- **onnistunut haku:** $O(N)$, $N/2$
- **epäonnistunut haku:** $O(N)$
- **lisäys:** $O(1)$
- **poisto:** $\text{haku} + O(1)$

Järjestetty lista

- **onnistunut haku:** $O(N)$, $N/2$
- **epäonnistunut haku:** $O(N)$, $N/2$
- **lisäys:** $O(N)$, $N/2$
- **poisto:** $\text{haku} + O(1)$

Aktiviteetin mukaan järjestetty lista

- **Jos avainten hakufrekvenssit poikkeavat toisistaan paljon, lista voidaan järjestää hakutilaston mukaan**
=> Painotettu keskimääräinen haku aika lyhenee
- **Jos hakutilasto tunnetaan ennakoita, lista järjestetään alunperin sen mukaan**
- **Muutoin voidaan toteuttaa *itsejärjestävä lista (self-adjusting list)*:**
 - ✓ **Jokainen haku siirtää alkioita pykälän edemmäksi**
 - Tai***
 - ✓ **Haettu alkio siirretään listan alkuun**

9.3 Taulukko tallennusrakenteena

Yksinkertainen hakurakenne, jolla on seuraavia ominaisuuksia:

- **Järjestetyssä taulukossa haku voidaan toteuttaa erittäin tehokkaasti suurillakin tietomäärillä**
- **Lisäykset ja poistot ovat raskaita operaatioita, $O(N)$**
=> Jos paljon muutoksia, ei taulukkoa kannata käyttää

Ongelmaa voidaan hieman kiertää *lazy deletion* menetelmällä, (ei varsinaisesti poisteta alkiota, vaan merkitään se mitättömäksi)

- **Jos taulukko ei ole järjestyksessä, sitä ei kannata käyttää hakurakenteena**

Hakumenetelmät järjestetyssä taulukossa:

Lineaarinen haku (linear search)

- **Haetaan taulukon alusta lähtien, kunnes alkio löytyy, tai sen paikka on ohitettu**
- **Tehokkuus $O(N)$**

Binäärihaku (binary search):

- **Tutkittava alue jaetaan kahtia, katsotaan kummassa puoliskossa tieto on ja jatketaan hakua tässä puoliskossa. Vrt. yhtälön numeerinen ratkaisu puolitusmenetelmällä**
- **Tehokkuus $O(\log N)$**
- **1 000 000 alkioista tieto löytyy 21 haulla**

```
#define N 1000
int a[N];
...
int binary_search(int key)
{
    int left, right, probe;
    left = 0;
    right = N-1;
    do {
        probe = (left + right)/2;
        if (key < a[probe])
            right = probe - 1;
        else
            left = probe + 1;
    } while ((key != a[probe]) && (left <= right) );

    if (key == a[probe])
        return probe;
    else
        return = -1;
}
```

Interpolaatiohaku

- **Kehittyneempi menetelmä, jossa käytetään hyväksi tietoa tiedon jakaumasta.**
- **Puolitushaussa uusi hakupaikka lasketaan kaavalla:**

$$\text{probe} = \text{left} + \frac{\text{right} - \text{left}}{2}$$

- **Interpolaatiohaussa pyritään laskemaan, mistä alkiota kannattaa hakea (jakolasku katkaiseva DIV):**

$$\text{probe} = \text{left} + \frac{(\text{what} - \text{a}[\text{left}].\text{key}) * (\text{right} - \text{left})}{\text{a}[\text{right}].\text{key} - [\text{left}].\text{key}}$$

Interpolaatiohaku....

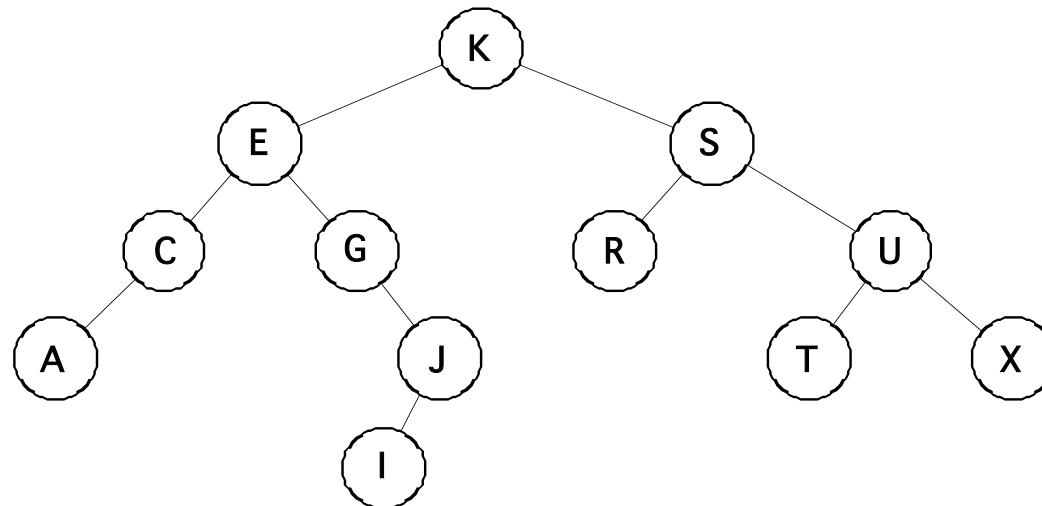
Huom.

- **Algoritmi toimii vain, jos haettava avain on tutkittavan alueen sisällä (pitää ottaa huomioon koodissa)**
- **Algoritmi soveltuu käytännössä vain kokonaisluku-avaimille. Merkkijonoilla interpolaation laskeminen on raskasta.**
- **Tehokkuus $O(\log(\log N))$, jos avainten jakauma on kohtalaisen tasainen**

9.4 Binäärinen hakupuu (binary search tree)

Binääripuu, joka on järjestetty sopivasti:

- **Kunkin solmun vasemmassa lapsessa olevat alkiot pienempiä kuin solmu itse**
- **Oikealla puolella olevat lapset suurempia kuin solmu itse**
- **Jos puussa on duplikaatteja, ne ovat aina vain toisessa haarassa, esim. oikealla puolella**



- **Lehtien lapsina voi olla NULL tai z-solmu**
 - **Voidaan käyttää myös Head-solmua, josta on osoitin juureen ja jonka avainarvo on pienempi kuin millään alkiolla puussa**
- => Haun ja lisäyksen koodi yksinkertaistuu**

9.4.1 Alkion haku:

- **Edetään juuresta käsin alaspäin kääntyen aina vasempaan tai oikeaan haaraan, kunnes avain löytyy**
- **Palautetaan osoitin tähän solmuun. Jos avainta ei löydy, palautetaan NULL tai z-solmu**
- **Käytettäessä z-solmua haettava alkio talletetaan ensin siihen, jolloin haku päättyy aina avaimen löytymiseen. Nyt täytyy vain tarkistaa, löytyikö alkio puusta vai z:sta.**
 - **rinnakkaisuus?**
- **Jos duplikaatit on talletettu puuhun, ne haetaan jatkamalla siitä, mihin edellinen haku päättyi:**
 - **item = find(data, head);**
 - **next_item = find(data, item->right);**

9.4.2 Alkion lisäys:

- 1) Uusi alkio lisätään aina lehdeksi**
- 2) Etsitään rekursiivisesti oikea paikka uudelle alkiole**
- 3) Haun aikana täytyy säilyttää tieto siitä, minkä lehden alle haku päättyi.**

9.4.3 Alkion poisto:

- **Monimutkaisempi toimenpide, jossa useita eri tapauksia:**
 - a) **Lehti voidaan poistaa suoraan**
 - b) **Jos solmulla on yksi lapsi, solmun isä asetetaan osoittamaan tähän lapseen**
 - c) **Jos solmulla on kaksi lasta:**
 - **Etsitään lähinnä seuraavan avainarvon sisältävä solmu S**
 - **Kopioidaan tiedot tästä tuhottavaan solmuun**
 - **Poistetaan solmu S rekursiivisesti. Tällä on korkeintaan oikea lapsi**
- **Toinen ratkaisu: *Lazy deletion* ja ajoittain puun uudelleen rakentaminen**

9.4.4 Alkioiden haku järjestyksessä:

- **Binäärisen hakupuun sisältö saadaan lajitellussa järjestyksessä käymällä puu läpi sisäjärjestyksessä**

9.4.5 Tehokkuusominaisuuksia:

- **Ideaalisessa binäärisessä hakupuussa kaikki operaatiot tehdään ajassa $O(\log N)$ paitsi tyhjän puun luonti $O(1)$**
- **Käytännössä puun kaikki haarat eivät ole yhtä pitkiä, jolloin haku-aika riippuu polun pituudesta**
- **Pahimmassa tilanteessa puu degeneroituu listaksi, jolloin haku-aika on $O(N)$**

9.4.6 Analyysi:

Teoreema: keskeisten operaatioiden (haku, lisäys, poisto, minimi, maksimi, seuraaja, edeltäjä) aikavaatimus binäärisessä hakupuussa on $O(h)$, jossa h on puun korkeus.

Esim. haku:

- **Pahin tapaus: $h = N$, jolloin pahimmassa tapauksessa hakuaika $O(N)$**
- **Paras tapaus: $h = 1$, jolloin parhaassa tapauksessa hakuaika $O(1)$**
- **Ideaalisessa tapauksessa puu tasapainossa:**
 - **$h = \log(N)$ ja hakuaika $O(\log N)$**
- **Keskimääräinen polun pituus:**
 - **Oletus: kaikki puut yhtä todennäköisiä**
 - **\Rightarrow keskimääräinen hakuaika $O(\log N)$**

MUTTA...

- **Delete-operaatioista johtuen kaikki puut eivät olekaan yhtä todennäköisiä \Rightarrow Lisäysten ja poistojen vuorottelu johtaa pitkän päälle haku aikaan $O(\sqrt{N})$**
- **Pahimmassa tapauksessa haku aika on $O(N)$.**
 - **alkiot järjestyksessä**
 - **alkiot käänteisessä järjestyksessä**
 - **vuorottelevat avainarvot, esim. 1 19 2 18 3 17...**
 - **sekvenssit, esim. 1 5 9 2 6 10 3 7 11 4 8 12...**
- **Nämä tapaukset voivat hyvin esiintyä käytännössä \Rightarrow Hakuajat voivat venyä liian pitkiksi**
- **Ratkaisu: Hakupuuta muokataan siten, että se pysyy tasapainossa**

9.5 Tasapainotetut puut (balanced trees)

- **Puun käsittelyyn liittyy sopivia ehtoja ja toimintoja, jotka takaavat lähes täydellisen tasapainon (eri haarojen pituus sama).**
- **Useita erilaisia rakenteita, mm.:**
 - **AVL-puu**
 - **Punamusta puu**
 - **B-puu**
- **Näitä tarkastellaan seuraavassa lähemmin**
- **Lisäksi on olemassa ns. Splay-puu, jossa**
 - **Yksittäinen operaatio voi ottaa ajan $O(N)$**
 - **M peräkkäistä operaatiota voidaan taata ottavan aikaa $O(M \log N)$**

9.5.1 Korkeustasapainotettu puu eli AVL-puu (height balanced tree, AVL tree)

Määrittely:

- Tyhjä puu on korkeustasapainotettu
- Jos T on ei-tyhjä binääripuu, jolla on lapsina T_L ja T_R , niin T on korkeustasapainotettu, jos ja vain jos
 - 1) T_L ja T_R ovat korkeus-tasapainotettuja
 - 2) $|h_L - h_R| \leq 1$, jossa h_L ja h_R ovat alipuiden korkeudet

Puun toteutuksessa jokaiseen solmuun liittyy *tasapainotekijä* (*balance factor*), jonka arvo on $h_L - h_R$ eli -1, 0 tai 1.

Vaihtoehtoisesti talletaan kaikkiin solmuihin niiden korkeus.

Huomaa, että tasapainovaatimus koskee puun kaikkia solmuja.

Haku:

- **Kuten binäärisessä hakupuussa**

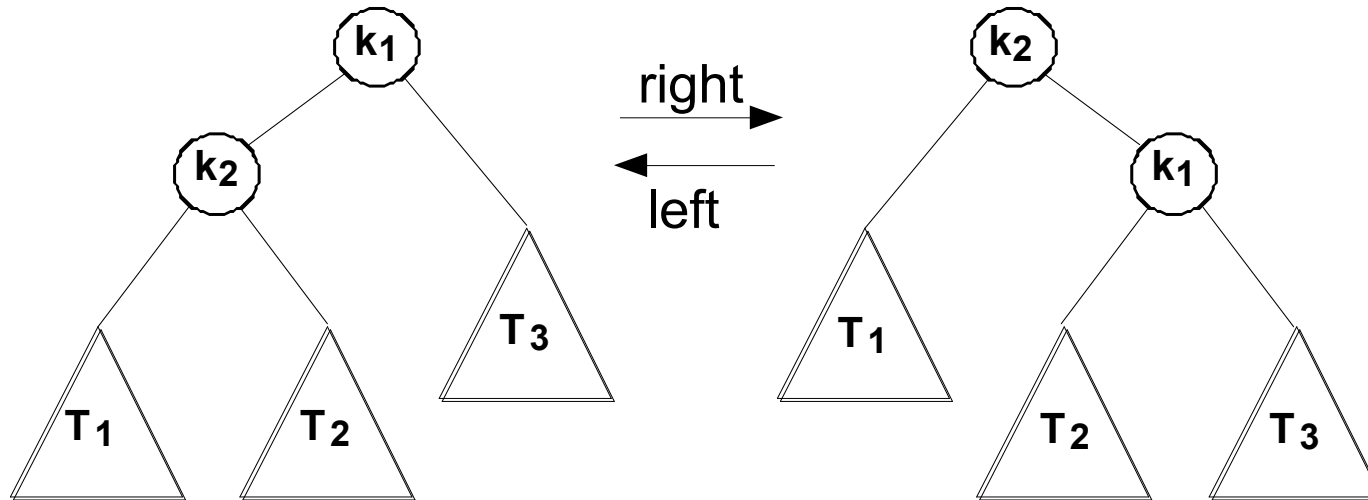
Lisäys ja poisto:

- **Kuten binäärisessä hakupuussa, mutta sitä seuraa tasapainotekijöiden päivitys ja tarvittaessa puun tasapainotus**

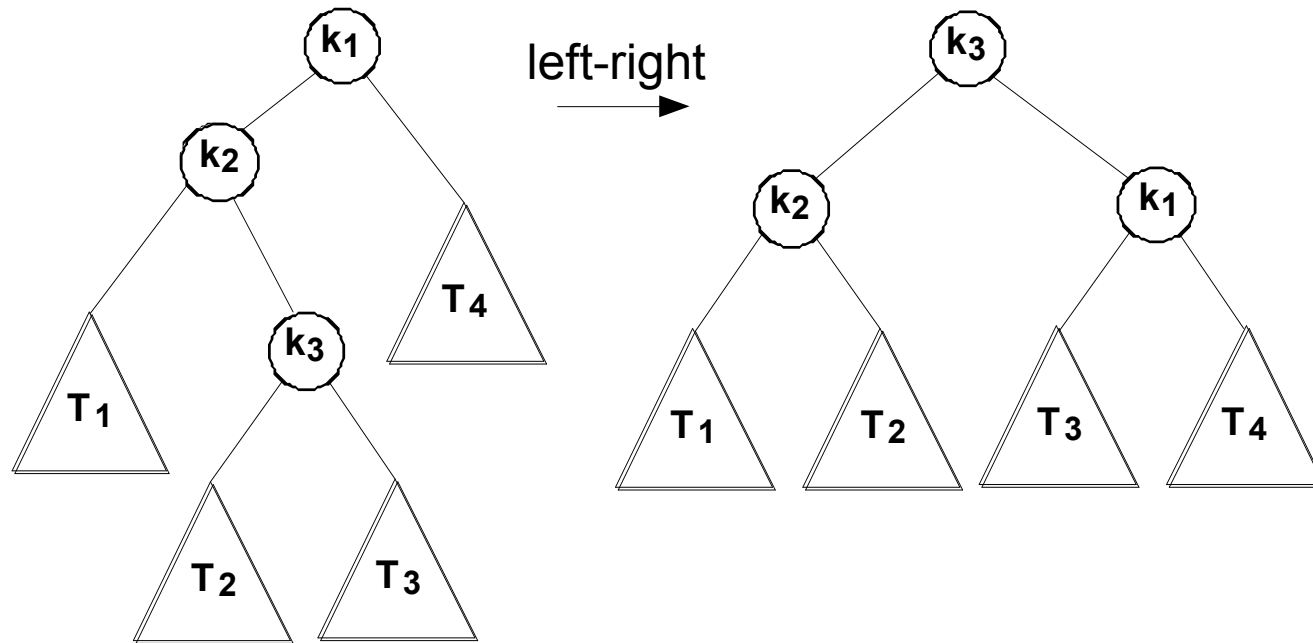
Tasapainotus lisäyksen yhteydessä:

- **Kun jossakin solmussa tasapainotekijän arvoksi tulee 2 tai -2, puun muotoa muutetaan tässä kohdassa *rotaatioilla***
- **Rotaatioita on kahdenlaisia:**
 - yksinkertaisia rotaatioita
 - kaksoisrotaatioita
 - Molempia voidaan tehdä sekä vasemmalle että oikealle

Yksinkertainen rotaatio (single rotation)



Kaksoisrotaatio (double rotation)



Kaksoisrotaatio voidaan ymmärtää myös kahtena yksinkertaisena rotaationa, jotka tehdään peräkkäin

Rotaatiot säilyttävät puun avainten järjestyksen.

- **Se, kumpi rotaatio tehdään, määräytyy siitä, mihin haaraan epätasapainon aiheuttava lisäys on tullut**
- **Rotaatio tehdään alimmassa solmussa, jossa on epätasapainoa**
- **Jos lisäys on tullut ko. solmun vasemman- tai oikeanpuoleisimman lapsenlapsen alle, tehdään yksinkertainen rotaatio päinvastaiseen suuntaan**
- **Jos lisäys on tullut ko. solmun sisempään alipuuhun, tehdään kaksoisrotaatio, joka nostaa ko. alipuuta ylemmäs puussa**
- **Lisäyksen jälkeen tarvitaan aina korkeintaan 1 rotaatio. Poiston jälkeen voidaan tarvita $O(\log N)$ rotaatiota.**
- **AVL-puussa haku, lisäys ja poisto tehdään ajassa $O(\log N)$. Tasapainotus ja tasapainotekijöiden asetus hidastavat puun käsittelyä vakiotekijällä (vähäinen merkitys)**

Luentotehtävä

- **Lisää seuraavat avainarvot tässä järjestyksessä tyhjiin AVL-puuhun:**

A V L D O N E F I R S T

9.5.2 Punamusta puu

- **Punamusta puu on binäärinen hakupuun, jossa on voimassa seuraavat säännöt**
 - 1) **Jokainen solmu on joko punainen tai musta**
 - 2) **Juuri on musta**
 - 3) **Jos solmu on punainen, sen lapset ovat mustia**
 - 4) **Jokainen polku juuresta NULL-pointteriin / Z-solmuun sisältää saman määrän mustia solmuja**

- **Usein käytetään määritelmää, jossa värit liitetään kaariin, esim.**
 - 1) **Jokainen kaari on joko punainen tai musta**
 - 2) **NULLiin / Z-solmuun johtavat kaaret ovat mustia**
 - 3) **Millään polulla juuresta lehteen ei saa olla kahta peräkkäistä punaista kaarta**
 - 4) **Jokainen polku juuresta NULLiin / Z-solmuun sisältää saman määrän mustia kaaria**

Haku:

- **Tapahtuu kuin binäärisessä hakupuussa**

Lisäys: (*bottom-up menetelmä*)

- ***Isäkaari* tarkoittaa solmun yläpuolella olevaan kaarta (isästä solmuun)**
- **Haetaan alkion paikka puussa ja lisätään se uudeksi lehdeksi p . Lisätyn solmun isäkaari merkitään punaiseksi.**
- **Jos $p:n$ isän $t:n$ isäkaari on musta, ei tarvita muuta**
- **Jos $t:n$ isäkaari on punainen, tarvitaan rotaatio, joka poistaa peräkkäiset punaiset kaaret. Rotaation jälkeen punaiset kaaret eivät ole enää peräkkäin.**

- **Jos ennen lisäystä sekä $t:n$ että sen sisaruksen $v:n$ isäkaaret ovat punaisia, rotaatio ei auta. Silloin tehdään värien vaihto, jossa $t:n$ isän $u:n$ isäkaari muuttuu punaiseksi ja $t:n$ ja $v:n$ isäkaaret mustiksi.**
- **Tarvittaessa tasapainotusta jatketaan ylempänä puussa, jos $u:n$ isän isäkaari on punainen.**

Lisäys: (top-down menetelmä)

- **Tasapainotus tehdään jo matkalla kohti lehteä, jonka alle uusi alkio lisätään**
- **Jos kohdataan solmu t , jonka lapsikaaret ovat punaisia, ne vaihdetaan mustiksi ja $t:n$ isäkaari muuttuu punaiseksi**
- **Jos tällöin $t:n$ isän $u:n$ isäkaari on punainen, tehdään rotaatio**
- **Tilanne, jossa rotaation jälkeen olisi kaksi peräkkäistä punaista kaarta, on mahdoton, koska $t:n$ isän $u:n$ sisaren isäkaari ei voi olla punainen. Se on ehkäisty ennakoita.**

Poisto:

- **Voidaan toteuttaa samantyyppisten periaatteiden avulla sekä bottom-up että top-down -algoritmeina**
- **Punamustan puun korkeus on korkeintaan $n \cdot 2\log N$**
- **Haku, lisäys ja poisto voidaan toteuttaa ajassa $O(\log N)$. Kustannus 1 bitti / solmu**

Luentotehtävä

- **Lisää seuraavat avainarvot tässä järjestyksessä tyhjiin punamustaan puuhun:**

S H O W R E D B L A C K

- **Duplikaattiarvot talletetaan oikealle.**

9.5.3 B-puu

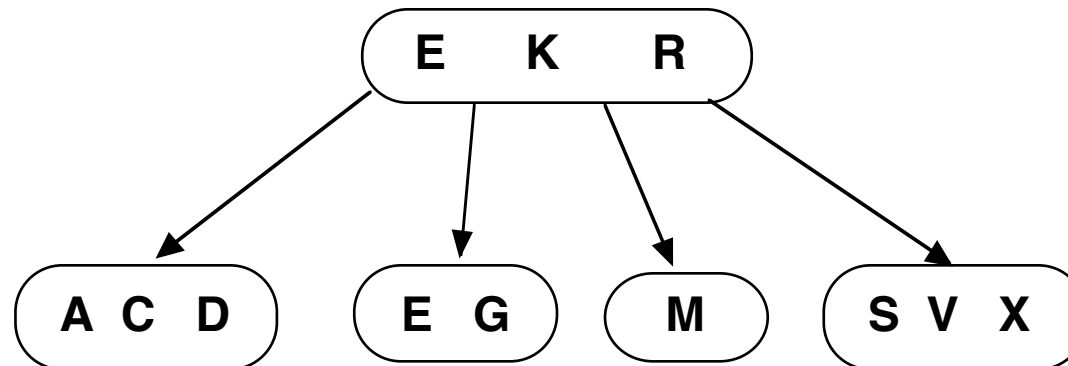
B-puu on tasapainotettu hakupuu, jossa yhdellä solmulla voi olla useita lapsia eli sen haarautumisaste (m) voi vaihdella muutamasta aina tuhansiin.

Määritelmä:

- **Juuri on joko lehti tai sillä on $2 \dots m$ lasta**
- **Kaikilla sisäsolmuilla on $(m + 1) \text{ div } 2 \dots m$ lasta (eli $\lfloor m/2 \rfloor \dots m$ lasta)**
- **Kaikki lehdet ovat samalla syvyydellä**
- **Solmuissa on osoittimia varsinaisiin tietueisiin**
- **Esimerkkitapaus: 2-3-4-puu ($m=4$)**

Käsittelysäännöt:**Haku:**

- **Sama idea kuin binäärisessä hakupuussa, mutta lavennettuna useampihaaraiseen puuhun**



Lisäys:

- 1) Haetaan uuden alkion paikka**
- 2) Jos solmussa on tilaa, avain lisätään siihen**
- 3) Jos solmu on täynnä (haun aikana), se jaetaan (*split*) kahdeksi solmuksi, joihin talletetaan:
 - Toiseen pienet avaimet ja toiseen suuret**
 - Pienet ja suuret erottava arvo talletetaan isäsolmuun (kohta 2)****
- 4) Jos juurikin joudutaan halkaisemaan, luodaan uusi juuri, jonka lapsiksi tulevat vanhan puolikkaat**

VAIN NÄIN PUUN KORKEUS VOI KASVAA!

=> Puu pysyy tasapainossa

Luentotehtävä

- **Lisää seuraavat avainarvot tässä järjestyksessä tyhjiin 2-3-4-puuhun**

T I E T O R A K E N N E

- **Duplikaattiarvot talletetaan oikealle.**

Poisto:

- **Melko monimutkainen toimenpide, jossa voidaan joutua myös siirtämään alkioita solmusta toiseen**
- **Jos solmuun jää liian vähän tietueita, täytyy etsiä solmu, jonka kanssa k.o. solmu voi sulautua**
- **Tällöin isäsolmusta häviää yksi osoitin, jne. Voi johtaa juuren häviämiseen, jolloin puun korkeus pienenee**

Ominaisuudet:

- **haku ajassa $O(\log N)$**
- **lisäys ja poisto: $O(M \log N)$**
- **Todellinen hyöty B-puista saadaan levymuistitoteutuksissa**
- **M valitaan niin, että solmujen koko vastaa levyblokin kokoa**
- **Levyhakuja tarvitaan puun korkeuden verran**

Eri versioita

- ***B⁺ -puu***: avaimia talletetaan vain lehtisolmuihin
 - Sisäsolmut sisältävät vain ns. *viittoja (router)*
 - Splitin jälkeen sisäsolmuun luodaan uusi viitta.
 - Käytännössä yleisin versio

- ***B^{*}-puu***: sisäsolmut vähintään 2/3-täynnä

- ***B^{link}-puu***, jossa samalla tasolla olevien lehtien ja sisäsolmujen välillä vaakasuorat linkit (*level link*)
 - Hyödyllinen sovelluksissa, joissa vaaditaan rinnakkaisuutta.

- **2-3-4-puu** voidaan keskusmuistirakenteena toteuttaa punamustana puuna
 - 2-3-4-puun solmuja erottavat mustat kaaret
 - Solmun sisällä (binääripuun) solmuja yhdistävät punaiset kaaret.

Tietorakenteet ja ulkoiset muistit

- **keskusmuisti ([Kilo-], Mega- ja Gigatavuja)**
 - **kallis ja nopea (0.5-5€ / MB)**
 - **hierarkia: muisti, cache, rekisteri**
- **ulkoinen muisti (Giga- ja Teratavuja, [Peta-, Exa-])**
 - **hidas ja halpa (0,005-0,05€ / MB)**
 - **massamuistit**
 - **hierarkia: raita (track, vrt. CD-levy), sivu (page, sektori)**
- **tyypillinen B-puu: ei mahdu kokonaan keskusmuistiin**
 - **datatietueet voivat viedä paljon enemmän tilaa kuin itse puu**
- **sivuhakujen määrä dominoi**

9.5.4 B-puu ulkoisena hakurakenteena

- **Jokainen solmu tallennetaan omalle levysivulle**
=> **Uuden solmun varaus helppoa**
- **Sisäsolmuissa on avainten lisäksi linkkitieto, mistä sivuilta lapset löytyvät**
- **Lehdissä on vain avaimia ja dataa tai avaimia ja linkit datasivuille**
- **Kun dataa ei talleteta puun sisäsolmuihin (B⁺-puu), haarautumisastetta voidaan nostaa tuntuvasti:**
 - **Esim. sisäsolmuissa $M_{\text{int}} = 1024$, jolloin 2-tasolta on linkit jopa yli miljoonaan sivuun. Lehdissä $M_{\text{ext}} =$ tietueiden määrä / sivu**

- **Lehtiä ei tarvitse merkitä erikseen, koska puun korkeus tunnetaan**
- **Sisäsolmut kannattaa hajauttaa eri levyille, jolloin hakuvarsista ei tule pullonkaulaa.**
- **B-puu on yleisesti käytössä tietokantarakenteiden toteutuksessa**
- **B-puu sallii joustavat lisäykset ja nopean haun**

9.6 Digitaaliset avaimet

Etsintä perustuu suoraan avaimen esitysmuodon bitteihin, ei avainten keskinäiseen vertailuun

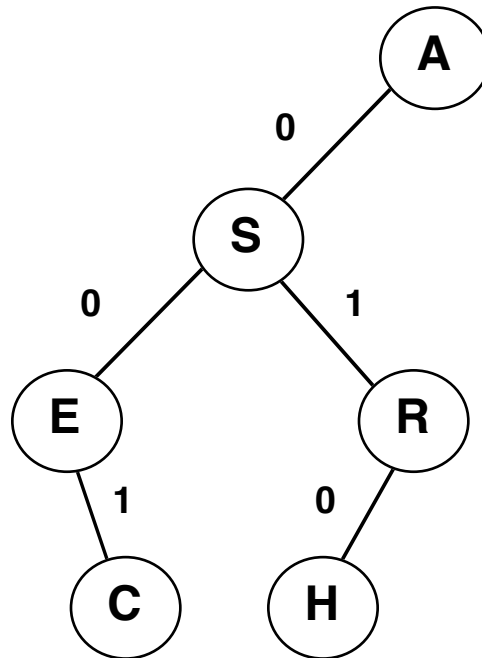
A = 00001
S = 10011
E = 00101
R = 10010
C = 00011
H = 01000
I = 01001
N = 01110

- Toteutus edellyttää tehokasta pääsyä bitti-informaatioon käsiksi

- **Jos avaimet tasaisesti jakautuneet, keskimääräinen tehokkuus $O(\log N)$**
- **Worst case on bittien määrä**
- **Merkkijonoavaimet eivät ole hyviä, koska ne ovat hyvin epätasaisesti jakautuneet**
- **Menetelmä voidaan yleistää muissakin kuin kantaluussa 2 esitettyihin avaimiin**

9.6.1 Digitaalinen hakupuu (digital search tree)

- **Hakupuu, jossa haarautuminen tapahtuu avaimen bittien perusteella**
 - **vasen = 0, oikea = 1 (tai päinvastoin)**
 - **Jokainen taso alaspäin tarkoittaa seuraavaa bittiä**
- **Muutoin toiminta muistuttaa binääristä hakupuuta**



A = 001011

C = 001101

E = 001111

H = 010010

R = 011100

S = 011101

Haku:

- 1. Aloitetaan juuresta ja eniten merkitsevistä bitistä**
- 2. Jos avain on käsiteltävässä solmussa haku päättyy**
- 3. Jos ei, edetään puussa nykyisen bitin perusteella**
- 4. Siirrytään seuraavaan bittiin, jatketaan kohdasta 2**

Lisäys:

- 1. Alkio lisätään aina puun lehdeksi**
- 2. Haetaan lisättävän alkion paikka**
- 3. Lisätään alkio lehtisolmun alle seuraavan bitin perusteella**

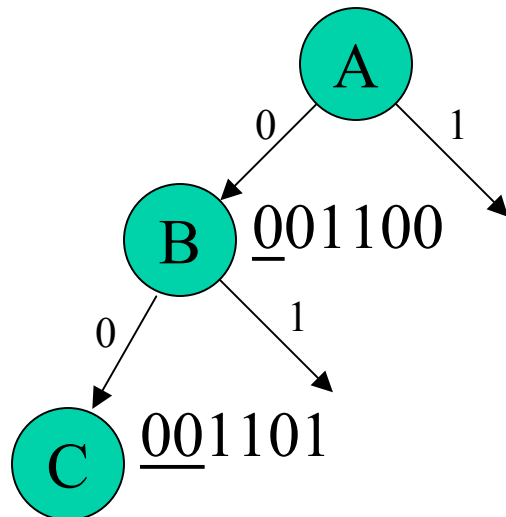
Poisto:

- Kuten binäärisessä hakupuussa**

Puussa ei voi olla duplikaatteja (käsitellään muutoin)

Luentotehtävä

- **Jatka seuraavassa alkioiden lisäämistä digitaaliseen hakupuuhun. Alkiot A,B ja C on jo lisätty puuhun. Lisää seuraavaksi alkiot D,E,F,G,H,I,J,K,L ja M tässä järjestyksessä:**



Alkioiden bittiavaimet:

A:	001011	H:	010010
B:	001100	I:	010011
C:	001101	J:	010100
D:	001110	K:	010101
E:	001111	L:	010110
F:	010000	M:	010111
G:	010001		

Ominaisuudet:

Average case: $O(\log N)$

Worst case: $O(b)$

Usein tasapainoisempi kuin vastaava binäärinen hakupuu

9.6.2 Radix search trie

Digitaalisen hakupuun yhtenä puutteena on se, että siinä joudutaan vertailemaan koko avainta jokaisella askelella, vaikka eteneminen tapahtuu vain bittien perusteella.

⇒ Idea: erotetaan etsintäpuu ja avaimet toisistaan

⇒ Radix search trie :

- **Hakupuussa on vain linkkejä bittien perusteella**
- **Tietueet ovat puun lehtinä**
- **Puun haarat ovat vain niin pitkiä, kuin on tarpeen avainten erottamiseksi toisistaan**
- **Puun muoto ei riipu avainten tallennusjärjestyksestä**

Haku:

- **Aloitetaan juuresta ja eniten merkitsevistä bitistä**
- **Edetään bitti kerrallaan puussa alaspäin kääntyen vasempaan tai oikeaan haaraan, kunnes kohdataan puun lehti**
- **Katsotaan, onko alkio lehdessä**

Lisäys:

- 1) Haetaan alkion paikka bittien perusteella**
- 2) Jos löydetty paikka oli tyhjä, lisätään alkio tähän kohtaan puun lehdeksi**
- 3) Jos paikassa oli lehti, tästä tulee puun sisäsolmu ja sen alle rakennetaan alipuu tai alipuut sen mukaan, että lehdessä ollut avain ja lisättävä avain eroavat toisistaan**
- 4) Luodaan uudet lehtisolmut, joihin entisen 3)-lehden avain ja lisättävä avain talletetaan**

Luentotehtävä

- **Lisää alkiot A, B, C, D, E, F, G, H, I, J, K, L ja M tässä järjestyksessä tyhjään radix search trie-rakenteeseen:**

Alkioiden bittiavaimet:

A: 001011	H: 010010
B: 001100	I: 010011
C: 001101	J: 010100
D: 001110	K: 010101
E: 001111	L: 010110
F: 010000	M: 010111
G: 010001	

Poisto:

- 1) Etsi poistettava avain**
- 2) Poista kyseinen lehtisolmu**
 - Jos lähin sisarus on myös lehtisolmu, lyhennä puuta sen osalta, kunnes ko. solmu eroaa vain yhden bitin mittaisen matkan jostakin puun muusta haarasta**

Tehokkuus:

- Average case: $O(\log N)$**
- Worst case: $O(b)$**

9.6.3 Variaatioita

- **Polun pituutta voidaan pienentää, jos verrataan kerralla M bittiä. Puu koostuu silloin solmuista, joissa on 2^M haaraa**
- **Jos M suuri, puu on matala, mutta sen hyötysuhde on huono (paljon käyttämättömiä linkkejä)**
- **Ratkaisuna *hybridirakenne*, jossa lähellä juurta M_1 on suuri ja lehdissä M_2 on pieni**
- **Ns. *level-compressed trie (LC-trie)* tiivistää puuta rekursiivisesti:**
 - **Juuren aste on 2^i , jossa i on pienin luku, jolla ainakin yksi juuren lapsista on lehtisolmu.**
 - **Jokainen lapsi on rakenteeltaan LC-trie**
- **LC-trie on tilankäytöltään optimaalinen**

- ***Patricia-puussa* on ratkaistu radix search trien ongelma, että puussa voi olla pitkiä tyhjien linkkien ketjuja**
- **Solmuissa on tieto siitä, minkä bitin mukaan haarautuminen tapahtuu**
- **Lisäksi puussa on vain yhdenlaisia solmuja**
- **Avaimia talletetaan, paitsi lehtiin, myös puun sisäsolmuihin. Niihin viitataan niistä puun lehdistä, joihin haku päättyy ko. avaimen bittien perusteella**
- **Toimii hyvin myös pitkillä avaimilla, koska itse avain testataan vain kerran ja biteistäkin vain osa**