Satunnaisuus tietorakenteissa

Iiro Ojala
iaojala@niksula.hut.fi

Seminaariesitelmä 30.4.2008
T-106.5800 Satunnaisalgoritmit
Tietotekniikan laitos
Teknillinen korkeakoulu

Tiivistelmä

Perinteinen tapa järjestää tietoa on tasapainotettu binäärinen hakupuu. Deterministisellä ratkaisulla on hyvä tasoitetun vaativuuden suorituskyky, mutta algoritmien toteuttaminen voi olla haastavaa. Vaihtoehdoksi jäävät satunnaiset menetelmät. Tässä esitelmässä esitellään kaksi keskeistä probabilistista hakurakennetta: Satunnainen treap ja skiplista. Lisäksi tarkastellaan lyhyesti pseudosatunnaiseen hajautusfunktioon perustuvia hajautustauluja. Käyttämällä satunnaisuutta hakupuu- ja listarakenteiden toteutusta voidaan yksinkertaistaa ja samalla saavuttaa parempi odotettu suorituskyky kuin vastaavalla deterministisellä ratkaisulla.

1 Johdanto

Yleinen tiedonjärjestämisongelman ratkaisu tarkoittaa tarkoittaa sellaisen joukon (tietorakenteen) ylläpitämistä, johon voidaan tehokkaasti kohdistaa alkioiden hakuja, päivityksiä, poistoja ja lisäyksiä. Näiden lisäksi voidaan tukea kokonaisiin joukkoihin kohdistuvia jako- ja yhdistämisoperaatioita. Tässä esitelmässä keskitytään keskeisimpien yksittäisen alkion haku-, lisäys- ja poisto-operaatioiden tarkasteluun.

Binääripuut ovat perinteinen ja tehokas tapa järjestää käsiteltävää tietoa. Deterministisillä algoritmeilla saavutetaan vertailuihin perustuvan haun tasoitetun vaativuuden alaraja $O(log(n))$. Käytännön suorituskyky voi kuitenkin suurien vakiokertoimien takia kärsiä, ja algoritmien implementointi on usein haastavaa. Kappaleessa 2. esitellään lyhyesti deterministinen ratkaisumalli.

Hakupuihin voidaan lisätä *satunnaisuutta* ja yksinkertaistaa näin algoritmien toimintaa. Esitelmässä osoitetaan, että satunnaisuuden käyttö itseasiassa tarjoaa saman $O(log(n))$ -suuruusluokan odotetun suorituskyvyn. Eräs yksinkertainen ja elegantti probabilistinen binääripuuhakurakenne on esitelty kappaleessa 3.1. Mahdollista on myös kuvata ei-binäärinen yleinen hakupuu monitasoisena listana, ja tätä ratkaisumallia käsitellään luvussa 3.2. Kappale 3.3. luo yleiskatsauksen toimintaperiaatteeltaan radikaalisti erilaiseen, pseudosatunnaista hajautusfunktiota hyödyntävään hajautustauluun. Hajautusfunktion avulla voidaan ylittää vertailuihin perustuvan haun teoreettinen alaraja $O(log(n))$ ja päästä vakioajassa $O(1)$ tapahtuvaan hakuun.

Esitelmän teksti perustuu alan keskeisiin julkaisuihin sekä Rajeev Motwanin ja Prabhakar Raghavanin kirjaan “Randomized Algorithms”[1], ja sen runko noudattelee pääpiirteissään kirjan Data Structures-luvun sisältöä. Hajautustauluja ja hajautusfunktioita on tutkittu erittäin paljon, ja niitä käsittelevä kappale perustuu alkuperäistutkimukset yhteenvetäviin teoksiin “Introduction to Algorithms”[2] ja “Algorithms in C”[3].

2 Deterministinen ratkaisu: Binääriset hakupuut

Hakupuu on yleinen deterministinen vakioratkaisu joukon S esittämiseksi tietorakenteena. Hakupuussa avaimet (ja niihin assosioitu data) talletetaan binääripuun solmuihin. Ollakseen hakupuu, puun tulee täyttää yleinen *hakupuuehto*: Avaimen k sisältävän solmun vasen alipuu sisältää vain avaimia, joiden arvo on pienempi kuin k, ja oikea alipuu sisältää vain avaimia, joiden arvo on suurempi kuin k. Mikäli hakupuuehto täyttyy, sanotaan puun avaiminen olevan *symmetrisessä* järjestyksessä. Hakupuulle voidaan suhteellisen helposti toteuttaa edellisessä kappaleessa esitetyt operaatiot. Tässä paperissa ei kuitenkaan käydä läpi operaatioiden toteutusten yksityiskohtia.

Ideaalitapauksessa n alkiota sisältävän puun korkeus on $log(n)$. Tällaista puuta kutsutaan tasapainoiseksi hakupuuksi. Yksinkertainen hakupuu voi kuitenkin helposti päätyä epätasapainoon. Triviaali esimerkki epätasapainoisen puun konstruoimiseksi on lisätä alkiot puuhun järjestyksessä pienimmästä suurimpaan, jolloin puun korkeudeksi tulee n, ja se olennaisesti redusoituu linkitetyksi listaksi.

Hakupuun tasapainottamiseksi on esitetty useita erilaisia strategioita. Yleisimmin käytetty menetelmä on *itsetasapainottuva AVL-puu*[4]: Jokaisen alkion lisäyksen ja poiston yhteydessä suoritetaan tarpeellinen määrä rotaatioita puun saattamiseksi tasapainoon. Autorootatiot takaavat hakuoperaatiolle tasapainoisen puun optimaalisen $O(log(n))$ suorituskyvyn, mutta samalla lisäyksille ja poistoille voidaan taata ainoastan tasoitettu vaativus luokassa $O(log(n)$.

Pelkän itsetasapainottumisen lisäksi hakupuuta voidaan rotatoida myös hakuoperaation yhteydessä. Tällöin haettu alkio rotatoidaan aina puun juureksi, tarkoituksena taata nopea uusi pääsy alkioihin, joiden käytöstä on kulunut lyhyin aika. Tällaista hakupuuta kutsutaan *splay*-puuksi, ja se tarjoaa optimaalisen tasoitetun vaativuuden suorituskyvyn käytettäessä mitä tahansa mielivaltaista alkioiden hakutaajuutta.[5]

Autorotaatioista johtuen splay-puun hakuoperaation kustannukseksi voidaan taata ainoastaan tasoitettu vaativuus luokassa $O(log(n))$. Tasoitetusta vaativuudesta seuraa määritelmänsä mukaisesti perustavaa laatua oleva ongelma: Rakenne ei pysty takaamaan jokaisen yksittäisen operaation nopeaa suoritusta, ainostaan tasoitetun kustannuksen. Seuraavassa kappaleessa esitellään vaihtoehtoisia satunnaisuutta hyödyntäviä lähestymistapoja, joiden avulla voidaan taata keskimääräinen odotettu $O(log(n))$ suorituskyky.

3 Satunnaisuutta hyödyntävät mallit

3.1 Satunnainen treap

Keoksi kutsutaan binääripuuta, jonka solmujen avainten arvot ovat aidosti vähenevässä järjestyksessä kaikilla mahdollisilla juuri-lehti -poluilla, ja vastaavasti tätä järjestystä kutsutaan kekoehdoksi. Binääripuuta, jonka solmuihin on yhden arvon sijasta sijoitettu kaksi arvoa, nimittäin avain k ja prioriteetti p, ja joka on samanaikaisesti avainten arvojen suhteen hakupuu sekä prioriteettien suhteen keko, kutsutaan nimellä *treap* (engl. tree + heap).

Formaalisti treap T on joukko

$$S = \{(k_1, p_1), \ldots, (k_n, p_n)\}$$

siten että alkion i avaimen arvo on k_i ja prioriteetti p_i.

Kaikki avaimien ja prioriteettien arvot ovat keskenään erisuuria, eikä niiden tarvitse olla samasta kaikkien mahdollisten arvojen joukosta. Joukosta S muodostettu treap takaa, että avainten arvot k_i ovat symmetrisessä järjestyksessä, ja samanaikaisesti prioriteettien arvot p_i ovat kekojärjestyksessä.

Aragon ja Seidel[6] näyttivät treap-rakenteen esitelleessä tutkimuksessaan, että kaikille mahdollisille erillisistä avaimista ja prioriteeteista $S = \{(k_1, p_1), \ldots, (k_i, p_i)\}$ koostuville joukoille on olemassa yksikäsitteinen treap $T(S)$. Toisin sanoen, treapin muoto ei ole riippuvainen lisäysten järjestyksestä, vaan mielivaltaisesta joukosta muodostuvan treap-tyyppisen puun muodon määrittelevät ainoastaan avainten ja prioriteettien suhteelliset arvot. Tällöin mikä tahansa haluttu puun muoto voidaan saavuttaa valitsemalla solmujen prioriteetit sopivasti.

Hakuoperaatiot suoritetaan treapissa täsmälleen samoin kuin binäärisessä hakupuussa. Lisäyksessä alkio lisätään rakenteeseen ensin kuten hakupuussa, jolloin rakenteen symmetrisyysehto täyttyy. Tämän jälkeen suoritetaan tarvittava määrä rotaatioita, jotta myös mahdollisesti rikottu kekoehto saadaan tyydytettyä. Poisto-operaatio toimii täsmälleen kuten käänteinen lisäys: Ensin alkio rotatoidaan lehtitasolle, ja sitten se yksinkertaisesti tuhotaan. Rotaation suunta valitaan poistettavan alkion vasemman ja oikean lapsen välisen suhteellisen prioriteetin mukaan.

Kun tarkoituksena on muodostaa *satunnunainen treap*, valitaan prioriteettien arvot satunnaisesti mistä tahansa mielivaltaisesta todennäköisyysjakaumasta. Valittavat arvot ovat muuten täysin toisistaan riippumattomia, mutta toteutuksen tulee taata että rakenteessa jo käytettyä prioriteettia ei hyväksytä uudelleen, eli prioriteettien on pysyttävä keskenään erillisinä. Alkion prioriteetti valitaan sen rakenteeseen lisäyshetkellä, eikä prioriteetti muutu suorituksen aikana. Lisäksi on huomionarvoista, että mikäli sama alkio (avain) poiston jälkeen lisätään rakenteeseen uudelleen, sille myös valitaan uudella lisäyshetkellä uusi prioriteetti täysin itsenäisesti.

Satunnaisesti valituista prioriteeteista konstruoidun treapin *odotettu* (expected) korkeus $O(log(n))$. Toisin sanoen satunnaisen treap-tyyppisen puun muodon odotusarvo on tasapainoinen hakupuu. Aragon ja Seidel näyttivät, että satunnaisella treapilla on (mm.) seuraavat kolme tärkeää ominaisuutta:

(i) Haun odotettu aikavaatimus on O(log(n)).

(ii) Lisäyksen odotettu aikavaatimus on O(log(n)).

(iii) Lisäyksessä tai poistossa tarvittavien rotaatioiden odotettu lukumäärä on vähemmän kuin kaksi.

Lisäksi helposti nähdään, että poiston aikavaatimus on sama kuin lisäyksen vaativuus, sillä operaatio on pohjimmiltaan käännetty lisäys.

Treapista voidaan luoda myös *painotettu satunnainen treap*. Tällöin solmuihin assosioidaan avaimen ja prioriteetin lisäksi kokonaislukuarvoinen paino w. Lisäyksessä solmun prioriteetiksi valitaan suurin w:stä erillisestä satunnaisesta prioriteetistä p. Lisäksi Aragon ja Seidel esittivät, että painotettuun satunnaiseen treapiin voidaan liittää splay-puuta vastaava ominaisuus, jolloin jokaisella hakukerralla solmun prioriteetiksi asetetaan maksimi sen hetkisestä prioriteetista ja uudesta satunnaisesta prioriteetista. Satunnaisuuden käyttäminen ei sulje siis pois painottamisen mahdollisuutta, painotusta voidaan haluttaessa käyttää myös treapin yhdeydessä.

3.2 Skiplista

Skiplista (engl. skip list) on tietorakenne, joka niin ikään käyttää probabilistista tasapainottamista tiukan deterministisen tasapainotuksen sijaan. Tämän seurauksena lisäysten ja poistojen toteuttaminen on merkittävästi yksinkertaisempaa kuin tasapainotettujen hakupuiden vastaavien operaatioden toteutus. Yksinkertaisemmin toteutettujen operaatioiden käytännön suorituskyky nousee tällöin jopa paremmaksi kuin hakupuiden tasoitettu vaativuus. Pugh antoi rakenteelle nimeksi skiplista, sillä hakuoperaatio ikäänkuin hyppii (enlg. skip) osan listan alkioiden yli lähestyessään haettavaa alkiota.[7]

Skiplistaa kuvaa monitasoinen järjestetty linkitetty lista, jossa alimmalla tasolla L_1 on linkki kaikkien alkioiden välillä, seuraavalla tasolla L_2 keskimäärin joka toisen alkion välillä, sitä seuraavalla keskimäärin joka kolmannen välillä ja niin edelleen. Kullakin tasolla olevien alkioiden lukumäärä noudattaa siis teoriassa geometrista todennäköisyysjakaumaa, mutta sallittu tasojen maksimimäärä jää implementaation määriteltäväksi. Jokaisen alkion taso (linkkien lukumäärä) valitaan satunnaisesti ennakkoon määritellystä satunnaisjakaumasta alkion listaanlisäyshetkellä, ja alkion taso säilyy samana sen koko listassaoloajan. Poistettaessa alkio samalla poistetaan myös kaikki siihen liittyvät linkit. Haku etenee listassa aina ylintä mahdollista tasoa pitkin, peruuttaa edelliseen alkioon ja jatkaa yhtä alemmalla tasolla mikäli ohitettiin etsittävä alkio, ja päättyy kun alkio on löydetty tai kun hakua ei voida enää jatkaa ja alkiota ei listasta löydy.

Formaalisti skiplista on järjestetty joukko

$$S = \{x_1 < x_2 < \cdots < x_n\}$$

ja joukon S *tasottaminen* (leveling) r:llä tasolla on sisäkkäisten osajoukkojen sarja

$$L_r \subseteq L_{r-1} \subseteq \cdots \subseteq L_2 \subseteq L_1$$

siten että $L_r = \emptyset$ ja $L_1 = S$.

Kun on annettu järjestetty joukko S ja tasot sille, minkä tahansa elementin $x \in S$ taso on määritelty siten, että

$$l(x) = \max\{i \mid x \in L_i\}$$

.

Teoriassa geometriseen todennäköisyysjakaumaan perustuva tasojen lukumäärän valinta voi generoida erittäin huonon listan, tai jopa redusoida sen pelkäksi yksitasoiseksi linkitettyksi listaksi. Tämä on kuitenkin erittäin epätodennäköistä, ja itseasiassa probabilistinen analyysi osoittaa, että hyvin suurella todennäköisyydellä satunnaista skiplistaa vastaava hakupuu on tasapainossa. Tällöin suurella todennäköisyydellä listan tasojen lukumäärä $r = O(log(n))$. Seuraavaksi tarkastellaan probabilistisen analyysin todistusta.

Tarkasti ottaen tasojen lukumäärä $r = 1 + \max_{x \in S} l(x)$, ja tasot $l(x)$ noudattavat toisistaan riippumatta geometrista jakaumaa parametrilla $p = 1/2$. Näin ollen S:n alkioiden tasot voidaan nähdä riippumattomasti geometrisesti jakautuneina satunnaismuuttujina $X_1, \ldots, X_n$. On helppoa näyttää, että $\mathbf{Pr}[X_i > t] \leq (1-p)^t$, joten

$$\mathbf{Pr}[\max_i X_i > t] \leq n(1-p)^t = \frac{n}{2^t},$$

koska tässä tapauksessa $p = 1/2$. Sijoitettaessa $t = \alpha \log n$ ja $r = 1 + \max_i X_i$ saadaan haluttu tulos

$$\mathbf{Pr}[r > \alpha \log n] \leq \frac{1}{n^{\alpha - 1}}$$

mille tahansa $\alpha > 1$.

Pugh esittää intuitioon ja empiirisiin kokeisiin perustuen, että satunnaisesti luotu enemmän kuin yhden tason yli sen hetkisestä maksimista oleva taso korvataan rakenteessa tasolla maksimi + 1. Näin modifioitu algoritmi toimii käytännössä erittäin hyvin, mutta se tekee algoritmin tehokkuuden analysoinnista merkittävästi vaikeampaa, ellei jopa mahdotonta, ja sen analyysi sivuutetaan tässä paperissa.

3.3 Hajautustaulu

3.3.1 Yleistä

Hajautustaulu (engl. hash table) ja hajautus (hashing) poikkevat perusteiltaan täysin edellä kuvatuista puu- ja listarakenteista. Siinä missä puut ja listat perustuvat avainten vertailuihin ja tarvittavien vertailujen minimoimiseen hyvin organisoidun rakenteen avulla, hajautus perustuu *avainten muuntamiseen aritmeettisesti* alkiot sisältävän taulun indekseiksi [3]. Muunnokseen käytettävää funktiota kutsutaan *hajautusfunktioksi* ja käsiteltävät alkiot sisältävää taulua *hajautustauluksi*.

Yksinkertaisin mahdollinen hajautus, suora osoittaminen, ei sisällä ollenkaan hajautusfunktion käyttöä vaan avaimia käytetään suoraan hajautustaulun indekseinä. Tämä on käytännölistä vain, jos mahdollisten avaimien lukumäärä eli universumi U tunnetaan ja se on riittävän pieni. Tässäkin tapauksessa on mahdollista, että todellisten käytettyjen avainten lukumäärä K jää niin pieneksi, että hukatun tilan suhde hyödynnettyyn tilaan kasvaa kestämättömän suureksi. Onkin kyseenalaista voidaanko menettelyä kutsua ollenkaan hajauttamiseksi.

Tehokas hajautusfunktio h kuvaa avaimet mahdollisimman tasaisesti välille $0 \ldots M-1$, jossa M on valitun hajautustaulukon koko. Tyypillisesti $M \ll U$. Ihannetapauksessa jokainen avain k tuottaa eri hajautusfunktion arvon $h(k)$. Käytännössä tämä ei ole useinkaan mahdollista, ja viimeistään kun hajautettujen arvojen lukumäärä n ylittää valitun hajautustaulukon koon M, saavat kaksi erisuurta avainta saman hajatusarvon, eli $\exists\, x, y \in U, x \neq y\ s.e.\ h(x) = h(y)$ [2]. Syntyvää tilannetta kutsutaan *törmäykseksi*.

3.3.2 Törmäysten käsittely

Toimiva hajautus vaatii aina toimivan *törmäysten käsittelyn*. Törmäysten käsittelymenetelmät jakaantuvat kahteen päätyyppiin: Erilliseen ketjutukseen (engl. separate chaining) ja avoimeen osoittamiseen (open addressing). Erillisessä ketjuttamisessa törmänneet alkiot talletetaan hajautustaulun ulkopuoliseen sekundaariseen tietorakenteeseen, tyypillisesti listaan tai binääripuuhun. Avoimessa osoittamisessa kaikki alkiot talletetaan itse hajautustauluun. Mikäli alkiota avaimella k lisättäessä taulun kohdassa $h(k, 0)$ on jo alkio, yritetään lisäystä kohtaan $h(k, 1)$ ja niin edelleen kunnes vapaa kohta löytyy. Vastaavasti haut suoritetaan kohdassa $h(k)$ olevaan apurakenteeseen tai kohdasta $h(k, 0)$ alkaen hajatustaulun alkioihin. Avoimessa osoittamisessa käytetty hajautusfunktio $h(k, i)$ toteuttaa normaalin hajautusen $h(k)$ ja menetelmän taulun kaikkien alkioiden läpikäymiseksi jossakin indeksin i osoittamassa järjestyksessä. Yksinkertaisin mahdollinen toteutus on lineaarinen tutkiminen (linear probing) $h(k, i) = h(k) + i$.

Törmäysten todennäköisyys kasvaa yhdessä taulun kuormituksen kanssa. Kuormitusta kuvaa rakenteessa olevien alkioiden lukumäärän n suhde taulukon kokoon M, jolloin kuormituskerroin $\alpha = \frac{n}{M}$.

Erillistä ketjuttamista käyttävässä hashtoteutuksessa $\alpha \geq 0$. Rakenne ei siis voi koskaan tulla täyteen. Keskimääränen hakuaika onnistuneelle haulle on $O(1+\frac{\alpha}{2})$ ja epäonnistuneelle haulle $O(1+\alpha)$. Worst-case-tehokkuus on katastrofaalisen huono $\theta(n)$. Siihen päädytään kaikkien alkioiden saadessa saman hajautusfunktion arvon rakenteen siten redusuituessa linkitetyksi listaksi. Harkittu hajautusfunktion tai funktiojoukon valinta tekee kuitenkin tällaisen käyttäytymisen äärimmäisen epätodennäköiseksi.

Koska ulkoisia rakenteita ei tarvita, avoin osoittaminen kuluttaa yleisesti vähemmän muistia kuin ketjutettu hajauttaminen. Tästä seuraa kuitenkin ilmeinen ongelma: Kuormituskerroin ei voi koskaan nousta yli yhden, ts. $0 \leq \alpha \leq 1$. Taulu voi siis tulla täyteen eikä enempiä lisäyksiä ole mahdollista tehdä. Käytännössä tämä voidaan estää varaamalla tarpeeksi suuri taulu, mikäli avainten lukumäärä K tunnetaan, tai rakentamalla taulu uudestaan suuremmalla M:n arvolla α:n ylittäessä jonkin rajan. Järkevä raja riippuu hajautusfunktion $h(k, n)$ ominaisuuksista. Onnistunut haku vaatii keskimäärin viisi kyselyä lineaarisen tutkimisen menetelmällä kun $\alpha < 0.90$ ja ***kaksinkertaisen hajauttamisen*** menetelmällä kun $\alpha < 0.99$ [3].

Kaksinkertaisessa hajauttamisessa nimensä mukaisesti käytetään kahta eri hajautusfunktiota. Toisen funktion tehtävä on varmistaa hakusekvenssin riippuvuus kummastakin parametrista k ja i kaikissa tilanteissa, ja se tarjoaa m^2 eri hakusekvenssiä. Funktiona käytetään tyypillisesti yksinkertaista ja nopeaa jakomenetelmää. Funktion jakajalla ja taulun koolla ei saa olla 1:tä suurempia yhteisiä tekijöitä, jotta kaikki sijainnit käydään läpi [2] [3].

Keskimääräinen hakuaika kaksinkertaista hajauttamista käytettäessä on onnistuneelle haulle $O(\ln(1-\alpha)/\alpha)$ ja epäonnistuneelle haulle $O(1/(1-\alpha))$ [3]. Worst-case-tehokkuus on sama kuin erillisessä ketjuttamisessa $\theta(n)$.

Alkioiden poistaminen avoimella osoittamisella rakennetusta taulusta ei ole aivan yhtä suoraviivaista kuin niiden poistaminen erillisestä apurakenteesta. Alkio paikassa $j = h(k)$ voi sijaita yhden tai useamman muun alkion tutkintasekvenssillä. Mikäli alkio nyt vain poistetaan tästä kohdasta ja paikka merkitään tyhjäksi, voi osa rakenteessa olevista alkioista jäädä saavuttamattomiin haun pysähtyessä taulukon tyhjään kohtaan. Yksinkertainen menetelmä korjata tämä tilanne on merkitä paikka tyhjän sijasta *poistetuksi*, jolloin paikka on vapaa uudelle alkiolle, mutta alkioita haettaessa hakusekvenssi ei pysähdy siihen kohtaan. Menettely aiheuttaa pidemmän päälle rakenteen ”rapautumista” ja hidastaa sen toimintaa. Käytännössä taulu pitää rakentaa uudestaan, kun siihen on tehty tarpeeksi paljon poistoja.

3.3.3 Hajautusfunktio

Hajautuksen suorituskyvyn kannalta keskeisimmässä osassa on hajautusfunktio. Hyvä hajautusfunktio jakaa avaimet tasaisesti välille $0 \ldots M - 1$. On olemassa kolme hyväksihavaittua perusmenetelmää toteuttaa hajautusfunktio [2]:

- Jakomenetelmä (division method)

  Metodi kuvaa avaimen halutulle välille laskemalla jakojäännöksen $h(k) = k \; mod \; M$. Menetelmän toimivuus riippuu pikälti M:n arvosta. Tutkimuksissa hyvin toimiviksi arvoiksi on havaittu esimerkiksi sellaiset alkuluvut, jotka eivät ole hyvin lähellä mitään 2:n potenssia. Tämän tarkempi analyysi sivuutetaan.

- Kertomismenelmä (multiplication method)

  Metodin mukaan avaimen arvo kerrotaan vakiolla C, tulosta otetaan desimaaliosa ja kerrotaan se taulun koolla M. Saatu tulos pyöristetään alas lähimpään kokonaislukuun $h(k) = \lfloor m * ((k * C) \; mod \; 1) \rfloor$. Menetelmä toimii kaikilla C:n arvoilla, mutta toisten arvojen on todettu tuottavan parempia tuloksia kuin toisten. Yksi yleinen hyvänä pidetty arvo on $(\sqrt{5} - 1)/2$. Todistus sivuutetaan.

- Yleinen hajautus (universal hashing)

  Yleinen eli universaali hajautusmetodi perustuu yhden kiinteän hajautusfunktion käytön sijasta joukkoon funktiota. Joukosta valitaan käyttöön ajonaikana satunnaisesti yksi funktio. Universaali hajautus ei siis välttämättä tuota samallakaan aineistolla eri ajokerroilla samaa tulosta, mutta se takaa hyvän keskimääräisen odotetun tuloksen mille tahansa aineistolle. Tällainen funktiojoukko on määriteltävissä helposti, kun avaimen pituus ja taulun koko tunnetaan.

Näistä menetelmistä mielenkiintoisin satunnaisuutta ja odotettua suorituskykyä hyödyntävä universaali hajauttaminen.

Formaalisti universaali hajauttaminen kuvauksena, jossa

Universumista $U = \{0, 1, \ldots, u - 1\}$ valitaan todelliset avaimet $M = \{0, 1, \ldots, m - 1\}$, $u \geq m$. Hajautusfunktioperhe H joukosta U joukkoon M on 2-universaali, mikäli $\forall x, y \in U$ siten että $x \neq y$ ja valittaessa h satunnaisesti joukosta H

$$\mathbf{Pr}[h(x) = h(y)] \leq \frac{1}{m}.$$

Täysin satunnaisen kuvauksen joukosta U joukkoon M todennököisyys törmäykselle millä tahansa arvoilla x, y on täsmälleen $1/m$. Näin ollen, satunnaisesti universaalista hajautusfunktioperheestä hajautusfunktion valitseva menetelmä on odotusarvoltaan täysin satunnaista funtiota vastaava.

Universaali hajautusfunktioperhe voidaan helposti konstruoida, kun tunnetaan U ja M. Valitaan alkuluku $p \geq u$ ja joukko $Z_p = \{0, 1, \ldots, p-1\}$. Määritellään funktiot $g : Z_p \rightarrow M$, $g(x) = x \mod m$. Samoin määritellään $\forall a, b \in Z_p$ lineaarinen funktio $f_{a,b} : Z_p \rightarrow Z_p$ sekä hajautusfunktio $h_{a,b} : Z_p \rightarrow M$ seuraavasti

$$f_{a,b}(x) = ax + b \mod p,$$
$$h_{a,b}(x) = g(f_{a,b}(x)).$$

4 Yhteenveto

Perustavaa laatua oleva tiedonjärjestämis- ja hakuongelma voidaan täydellisesti ratkaista deterministisillä menetelmillä, eikä probabilistiselle menetelmille siksi ole välttämätöntä tarvetta.

Kuitenkin käyttämällä hyväksi satunnaisuutta, voidaan useiden determinististen algoritmien toteutusta huomattavasti yksinkertaistaa. Käytännössä yksinkertaisempi toteutus usein tarkoittaa pienemmillä vakiokertoimilla varustettua suuruusluokan $O(log(n))$ kustannusfunktiota, ja näin ollen tarjoaa myös paremman reaalisen suorituskyvyn, kuten Pugh[7] skiplistan kohdalla empiirisillä kokeillaan osoitti.

Satunnainen treap tarjoaa saman odotetun haun suorituskyvyn kuin tasapainotettu puu, ja se voidaan tarvittaessa toteuttaa myös painotettuna satunnaisena treapina. Treap ei kuitenkaan tarvitse hankalia tasapainolaskelmia suorittaakseen keskimäärin tarvittavat rotaatiot lisäysten ja poistojen yhteydessä. Satunnaisen kekoehdon tyydyttävän algoritmin implementointi on helppoa, ja sen suorittaminen keskimäärin tuottaa tasapainoisen hakupuun.

Hajauttaminen eroaa radikaalisti järjestämiseen perustuvista menetelmistä. Se ei takaa absoluuttista hyvää worst-case-käyttäytymistä, mutta satunnaisuutta hyödyntäviä universaaleita hajautusfunktioperheitä käyttäen todennäköisyys sille on äärimmäisen pieni. Järkevästi valituilla parametreilla ja kohtuullisilla oletuksilla aineiston suhteen odotettu keskimääräinen hakuaika on $O(1)$. Hajautusfunktioiden tutkimus on hyvin aktiivista, ja niille onkin käyttöä myös hajautustaulujen ulkopuolella.

Viitteet

[1] R. Motwani and P. Raghavan, *Randomized Algorithms*, ch. 8. Cambridge University Press, 1995.

[2] L. C. Cormen, T. and R. R., *Introduction to Algorithms*, pp. 219–243. The MIT Press, 1990.

[3] R. Sedgewick, *Algorithms in C*, pp. 231–258. Addison-Wesley Publishing Company. Inc, 1990.

[4] G. Adelson-Velskii and E. M. Landis, "An algorithm for the organization of information," *Soviet Math. Doklady*, vol. 3, pp. 1259–1263, 1962.

[5] D. Sleator and R. Tarjan, "Self-adjusting binary search trees," *Journal of the ACM*, vol. 32, pp. 652–686, July 1985.

[6] C. Aragon and R. Seidel, "Randomized search trees," in *30th Annual Symposium on Foundations of Computer Science*, pp. 540–545, Oct./Nov. 1989.

[7] W. Pugh, "Skip lists: a probabilistic alternative to balanced trees," *Communications of the ACM*, vol. 33, no. 6, pp. 668–676, 1990.