

---

**Tietorakenteet ja algoritmit**

# **Hajautus**

**Ari Korhonen, Lauri Malmi**

---

# 10 Hajautus

## 10.1 Yleistä

## 10.2 Hajautusfunktio

## 10.3 Erillinen ketjutus

## 10.4 Avoin osoitus

### 10.4.1 Lineaarinen kokeilu

### 10.4.2 Neliöllinen kokeilu

### 10.4.3 Kaksoishajautus

## 10.5 Erityiskysymyksiä

### 10.5.1 Uudelleenhajautus

### 10.5.2 Ulkoinen etsintä

## 10.6 Hakurakenteet valinnasta

## 10.1 Yleistä

- **Jos avaimet ovat kokonaislukuja välillä 1-N, voidaan niitä käyttää suoraan hakutaulukon (koko N) indeksointiin. Hakuaika on  $O(1)$ .**
- **Useimmiten avainavaruus on liian laaja**
  - **pitkät kokonaisluvut**
  - **merkkijonot**
- **Hajautus pyrkii muuntamaan tällaiset avaimet kokonaisluvuksi välille 1-N, jolloin tieto voidaan tallentaa taulukkoon ja hakea sieltä ajassa  $O(1)$**

- **Hajautus on kompromissi aika-tila-vaatimusten suhteen:**
  - **rajaton tila:**
    - ✓ avain indeksinä
    - ✓ nopea haku, huono pakkaustiheys
  - **rajaton aika:**
    - ✓ tietueet peräkkäin, ei tietorakennetta
    - ✓ hidas haku, hyvä pakkaustiheys

**Hajautuksen toteuttamiseen tarvitaan kaksi asiaa:  
*hajautusfunktio ja yhteentörmäysten käsittelymekanismi***

**1) *Hajautusfunktio (hash function)* muuntaa kaikki avainavaruuden arvot hajautustaulukon osoiteavaruuteen  $1 \dots N$  (tai  $0 \dots N$  tai  $0 \dots (N-1)$ )**

- **Toteutettu yleensä aritmeettisilla laskutoimituksilla**
- **Ideaalinen hajautusfunktio palauttaisi eri osoitearvon kullekin eri avainarvolle. Käytännössä näin ei ole, vaan tarvitaan:**

## 2) *Yhteentörmäysten käsittelymekanismi (collision resolution)*

- **Kun hajautusfunktio tuottaa eri avaimille saman arvon, nämä tilanteet tulee erottaa toisistaan. Tähän on useita eri menetelmiä, mm.**
  - ✓ *erillinen ketjutus (separate chaining)*
  - ✓ *lineaarinen kokeilu (linear probing)*
  - ✓ *neliöllinen kokeilu (quadratic probing)*
  - ✓ *kaksoishajautus (double hashing)*

## 10.2 Hajautusfunktio (hash function)

- **Hyvä hajautusfunktio hajauttaa kaikki avainarvot tasaisesti osoiteavaruuteen**
- **Olennainen asia on se, että kaikkien avaimien kuuluvien numeroiden tai merkkien tulisi vaikuttaa hajautusarvoon**
- **Seuraavassa esitetään joitakin yksinkertaisia hajautusfunktioita (ja niiden sudenkuoppia):**

## Kokonaislukuavaimet:

**1) Jos avain on kokonaisluku ( $k \gg N$ ), varsin hyvä funktio on**

$$h(k) = k \% N$$

- **N:n tulee olla alkuluku. Tämä on helpoin tapa varmistaa, että kokonaisluvun  $k$  jokainen bitti vaikuttaa tulokseen.**

**2) Vielä parempi idea on, että korotetaan avain neliöön ja otetaan keskeltä muutama numero:**

$$h(k) = \text{trunc}(k^2/C) \% N, \text{ jossa } CN^2 \approx k^2 \text{ (avaimet välillä } 1-k)$$

- **N:ksi kannattaa valita kahden potenssi**

## Merkkijonoavaimet:

- **Merkkijono tulkitaan suurena kokonaislukuna ja käsitellään yleensä sopivina osina, esim. merkki kerrallaan**
- 3) Yksinkertainen ajatus on tulkita merkit numeroina, laskea ne yhteen ja jakaa summa sopivalla luvulla**

```
char k[100];  
int temp = 0;  
for (i = 0; i < key_length; i++)  
    temp = temp + k[i];  
h = temp % N;
```

**Ongelma on se, että merkkien numeroarvot ovat epätasaisesti jakautuneet välillä 0-127 tai 0-255, ja näistäkin vain pieni osa on aktiivisesti käytössä**

**4) Otetaan kolme ensimmäistä kirjainta, tulkitaan ne 27-järjestelmän lukuna (27 erilaista kirjainta) ja muunnetaan ne 10-järjestelmään:**

```
temp = 729 * value(k[1]) + 27 * value(k[2]) +  
      value(k[3]);  
h = temp % N;
```

**Oletus: Value palauttaa luvun välillä 0-26**

- **Jos kirjaimet ovat satunnaisia, tässä on hyvä funktio. Käytännössä näin ei tietenkään ole.**

## 5) Tulkitaan koko merkkijono kokonaislukuna ja lasketaan siitä % N

- **N ei tietenkään saa olla 128 tai 256!**
- **Asia lasketaan inkrementaalisesti Hornerin säännöllä**

```
temp = 0;
for (i = 0; i < key_length; i++)
    temp = ( (32*temp) + value(k[i]) ) % N;
h = temp;
```

**Oletus: Funktio `value` palauttaa luvun välillä 0-31**

## 6) Hajautusfunktioiden toteutuksia

[Kenneth Oksanen, 2002]

Käytännössä käytetään usein bittioperaatioita, esim.

```
unsigned int hashf(const char *buf, size_t len) {
    unsigned int h = 0;    int i;
    for (i = 0; i < len; i++) {
        h += buf[i];
        h += h << 10;
        h ^= h >> 7;
    }
    h += h << 3;
    h ^= h >> 11;
    h += h << 15;
    return h;
}
```

3/18/05 }

```
unsigned int hashf(unsigned int key) {  
    unsigned int h = 0xDEADBEEF;  
    h = key & 0xFFFF;  
    key >>= 16;  
    h += h << 10;  
    h ^= h >> 7;  
    h += key & 0xFF;  
    key >>= 8;  
    h += h << 10;  
    h ^= h >> 7;  
    h += key;  
    key >>= 8;  
    h += h << 10;  
    h ^= h >> 7;  
    h += h << 3;  
    h ^= h >> 11;  
    h += h << 15;  
    return h;  
}
```

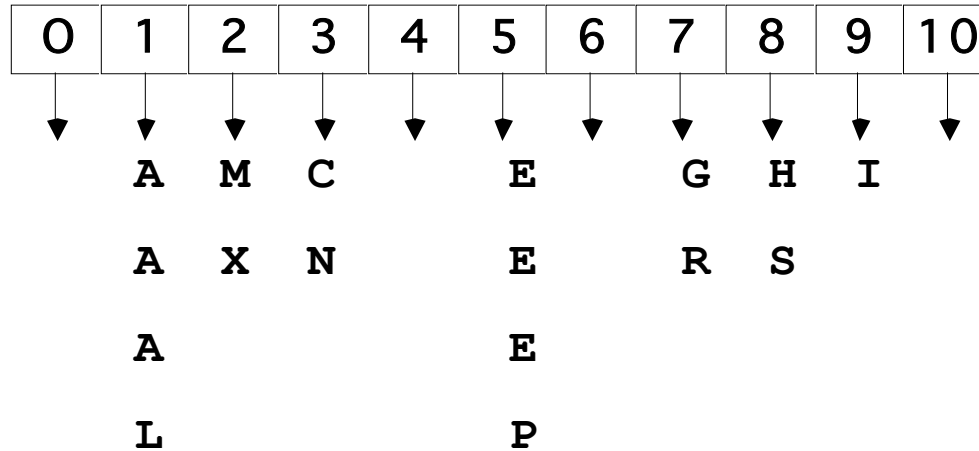
### 10.3 Erillinen ketjutus - separate chaining, (open hashing)

- Hajautustaulukko sisältää osoittimet listoihin, joihin yhteentörmänneet tietueet talletetaan
- **Esimerkki:** Hajautetaan alla oleva merkkijono 11 alkiota sisältävään taulukkoon käyttäen hajautusfunktiota  $k\%11$
- Kirjainten numeroarvot ovat:  
 $A = 1, B = 2\dots$

K:        A S E A R C H I N G E X A M P L E  
h(k):    1 8 5 1 7 3 8 9 3 7 5 2 1 2 5 1 5

	<b>k</b>	<b>k % 11</b>			
<b>A</b>	= 1	1	<b>P</b>	= 16	5
<b>B</b>	= 2	2	<b>Q</b>	= 17	6
<b>C</b>	= 3	3	<b>R</b>	= 18	7
<b>D</b>	= 4	4	<b>S</b>	= 19	8
<b>E</b>	= 5	5	<b>T</b>	= 20	9
<b>F</b>	= 6	6	<b>U</b>	= 21	10
<b>G</b>	= 7	7	<b>V</b>	= 22	0
<b>H</b>	= 8	8	<b>W</b>	= 23	1
<b>I</b>	= 9	9	<b>X</b>	= 24	2
<b>J</b>	= 10	10	<b>Y</b>	= 25	3
<b>K</b>	= 11	0	<b>Z</b>	= 26	4
<b>L</b>	= 12	1			
<b>M</b>	= 13	2			
<b>N</b>	= 14	3			
<b>O</b>	= 15	4			

- **Hajautusrakenteen sisältö:**



- **Listat kannattaa pitää järjestyksessä**
- **Listojen pituus riippuu hajautusarvojen jakaumasta ja duplikaattien määrästä**
- **Keskimäärin se on  $\approx M / N$  hyvällä hajautusfunktiolla (M hajautettua alkioita)**
- **N kannattaa valita esimerkiksi siten, että  $M / N \approx 10$**

### **Haku:**

- **Laske hajautusfunktion arvo**
- **Etsi alkion paikka listasta**
- **Palauta alkio tai tyhjä, jos haku epäonnistui**

### **Lisäys ja poisto:**

- **Suoritetaan haku. Tehdään lisäys/poisto listaan.**
- **Rakenne ei rajoita talletettavan tiedon määrää. Jos se kasvaa runsaasti, toiminnot vähitellen hidastuvat.**

## 10.4 Avoin osoitus - open addressing, (closed hashing)

- **Tietueet talletetaan suoraan hajautustaulukkoon hajautusfunktion osoittamaan paikkaan**
- **Vältetään dynaaminen muistinkäsittely**
- **Taulukon koko  $N$  suurempi kuin hajautettavien avainten määrä  $M$**
- **Yhteentörmäysten käsittelyyn on useita eri menetelmiä, joissa tietueiden haku-/ sijoituspaikka lasketaan seuraavasti**

$$h_i(k) = (\text{hash}(k) + f(i)) \% N \quad i = 0, 1, 2, \dots \quad f(0) = 0$$

- **Haku yleensä erittäin nopeaa, ellei taulukko tule liian täyteen**
- **Jos taulukko täyttyy, uudelleen järjestely raskas toimenpide**

## 10.4.1 Lineaarinen kokeilu (linear probing)

- **Yhteentörmäysten sattuessa etsitään taulukosta järjestyksessä seuraavaa paikkaa, kunnes operaatio onnistuu tai epäonnistuu**

$$h_i(k) = (\text{hash}(k) + i) \% N$$

$$i = 0, 1, 2, \dots$$

- **Tyhjät paikat merkitään jollain sopivalla erikoisarvolla**

### Haku:

- **Laske hajautusfunktion arvo**
- **Jos paikka on jo käytössä, etsi taulukosta järjestyksessä eteenpäin, kunnes tietue löytyy tai tulee vastaan tyhjä paikka**

**Lisäys:**

- **Laske hajautusfunktion arvo**
- **Jos paikka on jo käytössä, etsi taulukosta järjestyksessä eteenpäin seuraava tyhjä paikka. Lisää alkio tähän**

**Poisto:**

- **Hankalampi tilanne, koska syntyvä tyhjä paikka ei saa sotkea muuta rakennetta**
- **Ratkaisuna poistetun alkion paikalle asetetaan erikoisarvo (place holder), joka tulkitaan tyhjäksi lisäysoperaatiossa, mutta ei hakuoperaatiossa**

## Luentotehtävä: lineaarinen kokeilu

- Seuraavassa on aloitettu alkioiden hajauttaminen 19 paikkaiseen taulukkoon käyttäen hajautusfunktiona:

$$h_i(k) = (k + i) \% 19, \quad i = 0, 1, 2, \dots; \quad f(0) = 0$$

K:        A S E A R C H I N G E X A M P L E

h(k):    1 0 5 2

i:        0 0 0 1

**Tehtävä: Jatka hajauttamalla loput avaimet**

S	A	A			E													
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

---

	<b>k</b>	<b>k % 19</b>			
<b>A</b>	= 1	1	<b>P</b>	= 16	16
<b>B</b>	= 2	2	<b>Q</b>	= 17	17
<b>C</b>	= 3	3	<b>R</b>	= 18	18
<b>D</b>	= 4	4	<b>S</b>	= 19	0
<b>E</b>	= 5	5	<b>T</b>	= 20	1
<b>F</b>	= 6	6	<b>U</b>	= 21	2
<b>G</b>	= 7	7	<b>V</b>	= 22	3
<b>H</b>	= 8	8	<b>W</b>	= 23	4
<b>I</b>	= 9	9	<b>X</b>	= 24	5
<b>J</b>	= 10	10	<b>Y</b>	= 25	6
<b>K</b>	= 11	11	<b>Z</b>	= 26	7
<b>L</b>	= 12	12			
<b>M</b>	= 13	13			
<b>N</b>	= 14	14			
<b>O</b>	= 15	15			

- **Lineaarisen kokeilun ongelmana on se, että taulukon täyttyessä syntyy *kasaumia (cluster)*, jotka pidentävät hakuketjua merkittävästi**
- **Esimerkin kohdassa ...E X A syntyy kasauma 5-10 talletettaessa merkki X**
- **Kasaumien todennäköisyyttä pyritään pienentämään muissa menetelmissä**

## 10.4.2 Neliöllinen kokeilu (quadratic probing)

- Sama pääperiaate kuin lineaarisessa kokeilussa, mutta yhteentörmäyksen sattuessa seuraavat kokeilut tehdään alkioissa:

$$h_i(k) = (\text{hash}(k) + i^2) \% N, \quad i = 0, 1, 2, \dots$$

- Estää em. primäärisen kasautumisen
- Periaatteessa esiintyy *sekundääristä kasautumista*
  - kaikilla avaimilla, joilla on sama hajautusarvo, testataan samoja paikkoja
  - vähäinen merkitys
- Taulukon koon täytyy olla alkuluku, muuten kokeilujen määrä voi jäädä aivan liian pieneksi
- Uusia alkioita voidaan varmuudella lisätä vain, jos taulukon täyttöaste on alle puolet

### 10.4.3 Kaksoishajautus (double hashing)

- Sama pääperiaate kuin edellisissä, mutta yhteentörmäyksen sattuessa seuraavat kokeilut tehdään alkioissa:

$$h_i(k) = (\text{hash}(k) + i * \text{hash}_2(k)) \% N, \quad i = 0, 1, 2, \dots$$

- $\text{hash}_2$  ei saa koskaan saada arvoa 0
- yksi hyvä valinta on:

$$\text{hash}_2(k) = R - (k \% R), \quad \text{jossa } R \text{ on alkuluku, } < N$$

- Kaksoishajautus ratkaisee primäärisen kasautumisen ongelman.

- **Sekundääristä kasautumista esiintyy, mutta sen merkitys on vähäinen**
- **Kahden hajautusfunktion laskeminen kuormittaa laskemista enemmän kuin neliöllisessä kokeilussa**

## Luentotehtävä: kaksoishajautus

- Seuraavassa on aloitettu alkioiden hajauttaminen 19 paikkaiseen taulukkoon käyttäen hajautusfunktioina:

$$h_1(k) = k \% 19, \quad h_2(k) = 1 + (k \% 11)$$

K:        A S E A R C H I N G E X A M P L E

$h_1(k)$ : 1 0 5 1

$h_2(k)$ : 2 9 6 2

**Tehtävä: Jatka hajauttamalla loput avaimet**

S	A		A		E													
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

<b>k</b>	<b>k%19</b>	<b>1+(k MOD 11)</b>			
<b>A = 1</b>	<b>1</b>	<b>2</b>	<b>O = 15</b>	<b>15</b>	<b>5</b>
<b>B = 2</b>	<b>2</b>	<b>3</b>	<b>P = 16</b>	<b>16</b>	<b>6</b>
<b>C = 3</b>	<b>3</b>	<b>4</b>	<b>Q = 17</b>	<b>17</b>	<b>7</b>
<b>D = 4</b>	<b>4</b>	<b>5</b>	<b>R = 18</b>	<b>18</b>	<b>8</b>
<b>E = 5</b>	<b>5</b>	<b>6</b>	<b>S = 19</b>	<b>0</b>	<b>9</b>
<b>F = 6</b>	<b>6</b>	<b>7</b>	<b>T = 20</b>	<b>1</b>	<b>10</b>
<b>G = 7</b>	<b>7</b>	<b>8</b>	<b>U = 21</b>	<b>2</b>	<b>11</b>
<b>H = 8</b>	<b>8</b>	<b>9</b>	<b>V = 22</b>	<b>3</b>	<b>1</b>
<b>I = 9</b>	<b>9</b>	<b>10</b>	<b>W = 23</b>	<b>4</b>	<b>2</b>
<b>J = 10</b>	<b>10</b>	<b>11</b>	<b>X = 24</b>	<b>5</b>	<b>3</b>
<b>K = 11</b>	<b>11</b>	<b>1</b>	<b>Y = 25</b>	<b>6</b>	<b>4</b>
<b>L = 12</b>	<b>12</b>	<b>2</b>	<b>Z = 26</b>	<b>7</b>	<b>5</b>
<b>M = 13</b>	<b>13</b>	<b>3</b>			
<b>N = 14</b>	<b>14</b>	<b>4</b>			

## 10.5 Erityiskysymyksiä

- 1) **Mikä on taulukon oikea koko?**  
⇒ jos se arvioidaan väärin, joudutaan *hajauttamaan uudelleen (rehashing)*, mikä on raskas operaatio,  $O(N)$
- 2) **Jos tiedon määrä vaihtelee suuresti, joudutaan kenties tarpeettomasti hajauttamaan uudelleen tai täyttöaste voi jäädä liian alhaiseksi**  
⇒ ei sovellu näihin tilanteisiin
- 3) **Jos aineiston koko on suuri, rakenne ei mahdu muistiin**  
⇒ ulkoinen etsintä
- 4) **Jos täyttöaste  $\alpha$  kasvaa lähelle arvoa 1, operaatioiden kesto kasvaa nopeasti:**

$$\alpha = M / N$$

**▪ Pienillä arvoilla:**

- **onnistunut haku:**             $1 + M / N$
- **epäonnistunut haku:**     $1 + M / 2N$

**▪ Suurilla täyttöasteilla operaatioiden vaatima aika on muotoa:**

$$\frac{1}{1 - \alpha}$$

## 10.5.1 Uudelleenhajautus (Rehashing)

- **Taulukon täytyessä luodaan uusi, kaksi kertaa isompi taulukko, johon vanhan taulukon tiedot hajautetaan**
- **Voidaan suorittaa myös toiseen suuntaan, jos taulukosta tulee poistojen jälkeen liian ”harva”**
- **Keskimääräinen kustannus  $O(1)$  yhtä operaatiota kohti, mutta saattaa viivästyä yksittäistä tapahtumaa huomattavasti**

## 10.5.2 Ulkoinen etsintä (external searching)

- **Ulkoisessa etsinnässä tieto on osin tai kokonaan talletettu oheismuistiin**

=> **I/O-aika dominoi**

=> **Laitteistotekniikka voi asettaa lisärajoituksia**

- **Ulkoisilla hakumenetelmillä on hyvin suuri käytännön merkitys, koska tietokannat yleensä talletetaan levyille.**

- **Tavoitteena on minimoida levyhakujen määrä**

=> **Peräkkäishaku levyllä mieletöntä**

## **Tärkeitä menetelmiä ovat:**

- **Indeksoitu hierarkkinen rakenne**
  - **B-puut**
  - **Laajennettava hajautus**
- 
- **sisäisten menetelmien loogisia laajennuksia**
  
  - **eroavat suorituskyvyltään lähinnä lisäysten joustavuuden suhteen**

## **Laitteistotekniikan vaikutukset:**

### **1) Sarjasaantilaitteet (mg-nauha, kasettinauha)**

- **Ainoastaan peräkkäishaku on mahdollista, joten mitään erityismenetelmiä ei ole**

## 2) Suorasaantilaitteet (levy)

- **Oletetaan yksinkertainen toimintamalli:**
  - **Käytössä on useita levyjä**
  - **Levyllä tieto on jaettu sivuille, joihin voi viitata mielivaltaisessa järjestyksessä**
  - **Kullekin sivulle mahtuu useita tietueita**
  
- **Todellisuus on monimutkaisempi:**
  - **virtuaalimuisti**
  - **cache**
  - **rinnakkaisuuden hallinta**
  
- **Jatkossa tarkastellaan vain levyllä olevia tallennusrakenteita**

## Indeksoitu hierarkinen rakenne (indexed sequential access)

- **Peräkkäishakumenetelmä, jonka apuna on yksi tai useampia tasoja *hakemistoja (index)***
  - 1) **Tieto on järjestyksessä ja jaettu erillisille sivuille**
  - 2) **Kullakin levyllä on hakemistosivuja, joissa on tieto kaikista sivuista, mitä rakenteeseen kuuluu:**
    - **sivun sisältämät alkiot (alku, loppu)**
    - **sivun osoite levyllä**
  - 3) **Keskusmuistissa voi olla *master-hakemisto (master index)*, johon on talletettu tieto, mitä tietoja on kullakin levyllä**

## **Esimerkki:**

**Merkkijono EXTERNALSEARCHINGEXAMPLE  
tallennettuna tähän rakenteeseen:**

- **Jokaisella levyllä hakemistosivu on ensimmäisenä, sitten datisivut**
- **Hakemistossa on tieto käytössä olevien sivujen osoitteista ja sivuilla olevien avainten minimi- ja maksimiarvoista**
- **Samaa avainta voi olla usealla sivulla**

	0	1	2
Disk 1:	<u>*1C2E</u>	AAAC	EEEE
Disk 2:	<u>E1I2N</u>	EGHI	LLMN
Disk 3:	<u>N1R2X</u>	NPRR	STXX

**Master index: \*1E2N3X**

- **Tietueen hakuun tarvitaan tavallisesti vain kaksi levyhakua (3 tai 4 hakua, jos avain esiintyy hakemistoissa)**
- **Päivittäminen erittäin kankeaa, koska kaikki hakemistot voidaan joutua rakentamaan uudelleen. Siten soveltuu vain hyvin staattisiin sovelluksiin.**

## **Laajennettu hajautus (EXHASH, extendible hashing)**

- **Vain 2 hakua**
- **Tehokas lisääminen**
- **Tieto tallennettu sivuille, jotka täytyessään jakaantuvat (vrt. B-puu)**
- **Sivut löydetään indeksin avulla (vrt. hierarkinen indeksoitu haku)**
- **Apuna avainten digitaaliset ominaisuudet (bitit)**

## **Esimerkki:**

**Talletetaan alla oleva merkkijono tietorakenteeseen tällä menetelmällä:**

### **EXTERNALSEARCHINGEXAMPLE**

- **4 elementtiä / sivu**
- **kirjaimilla 5-bittinen koodi**

---

	<b>k</b>	<b>koodi</b>			
<b>A</b>	<b>= 1</b>	<b>00001</b>	<b>P</b>	<b>= 16</b>	<b>10000</b>
<b>B</b>	<b>= 2</b>	<b>00010</b>	<b>Q</b>	<b>= 17</b>	<b>10001</b>
<b>C</b>	<b>= 3</b>	<b>00011</b>	<b>R</b>	<b>= 18</b>	<b>10010</b>
<b>D</b>	<b>= 4</b>	<b>00100</b>	<b>S</b>	<b>= 19</b>	<b>10011</b>
<b>E</b>	<b>= 5</b>	<b>00101</b>	<b>T</b>	<b>= 20</b>	<b>10100</b>
<b>F</b>	<b>= 6</b>	<b>00110</b>	<b>U</b>	<b>= 21</b>	<b>10101</b>
<b>G</b>	<b>= 7</b>	<b>00111</b>	<b>V</b>	<b>= 22</b>	<b>10110</b>
<b>H</b>	<b>= 8</b>	<b>01000</b>	<b>W</b>	<b>= 23</b>	<b>10111</b>
<b>I</b>	<b>= 9</b>	<b>01001</b>	<b>X</b>	<b>= 24</b>	<b>11000</b>
<b>J</b>	<b>= 10</b>	<b>01010</b>	<b>Y</b>	<b>= 25</b>	<b>11001</b>
<b>K</b>	<b>= 11</b>	<b>01011</b>	<b>Z</b>	<b>= 26</b>	<b>11010</b>
<b>L</b>	<b>= 12</b>	<b>01100</b>			
<b>M</b>	<b>= 13</b>	<b>01101</b>			
<b>N</b>	<b>= 14</b>	<b>01110</b>			
<b>O</b>	<b>= 15</b>	<b>01111</b>			

**Alkion haku:**

- 1) Laske avaimen d:n ensimmäisen bitin perusteella oikea hakemistosivu**
- 2) Hae hakemistosta sivun osoite ja sen jälkeen sivu levyltä muistiin**
- 3) Muistissa hae tieto peräkkäishaulla, binäärihaulla tms.**

## **Uuden alkion lisääminen:**

### **3 eri tapausta:**

- 1) Lisäys sivulle, jolla on tilaa**
- 2) Lisäys sivulle, jolla ei ole tilaa ja hakemiston päivitys**
- 3) Lisäys sivulle, jolla ei ole tilaa ja hakemiston kahdennus (harvinaista)**

## Hakemisto

- **Sisältää vain avaimet ja osoittimet levyille, ei dataa**  
**=> Sivulle mahtuu entryjä enemmän kuin varsinaisia tietueita. Esimerkissä suhde = 8 / 4, käytännössä suurempi**
- **Kun hakemisto ei mahdu yhdelle sivulle, tarvitaan *master index*, joka toimii samalla periaatteella ja sijaitsee keskusmuistissa**  
**=> Tieto löytyy kahdella haulla**

## Ongelmia

### 1) Epäsymmetrinen avainjakauma

=> Hakemiston koko kasvaa tarpeettomasti

Esim. 32-bittiset avaimet:

- Joillakin sivuilla erotteluun tarvitaan 20 bittiä, joillakin riittää 5 bittiä

=> Hakemistossa  $2^{20}$  entryä ja paljon päällekkäisiä osoittimia

**Ratkaisu:**

- Hajautetaan avaimet ennen hakua
  - => Jakauma tasoittuu
  - => Yhteentörmäyksistä ei ongelmaa
  - => Nimitys: Extendible hashing

## **2) Liikaa samoja avaimia**

- **Perusalgoritmi ei toimi, jos samoja avaimia on enemmän, kuin sivulle mahtuu**

**=> Tarvitaan virittelyä**

## **YHTEENVETO**

**1) Tehokas haku, vain 2 levyhakua**

**2) Joustava päivitys**

## 10.6 Hakurakenteen valinnasta

- **Hakurakenne voidaan implementoida monella tavalla. Asiaan liittyviä näkökulmia ovat:**
  - **tiedon määrä**
  - **talletetaanko rakenne keskusmuistiin vai massamuistiin?**
  - **hakuihin, lisäyksiin ja poistoihin liittyvät aikavaatimukset**
  - **lisäysten ja poistojen määrä**
  - **tarvitaanko erityyppisiä hakuavaimia?**
  - **operaatiot, joissa avaimia käsitellään järjestyksessä?**
  - **menetelmän monimutkaisuus**

- **Yhteenveto sisäisten menetelmien tehokkuusluvuista, kun haku onnistuu**

	<b>W(N)</b>	<b>A(N)</b>
<b>Linkitetty lista (järj.)</b>	<b>N</b>	<b>N/2</b>
<b>Binäärihaku taulukosta</b>	<b>log N</b>	<b>log N</b>
<b>Interp.haku taulukosta</b>		<b>log log N</b>
• <b>tasaisella jakaumalla</b>		
<b>Binäärinen hakupuu</b>	<b>N</b>	<b>1.38log N</b>
<b>2-3-4-puu</b>	<b>O(log N)</b>	<b>log N</b>
<b>AVL-puu</b>	<b>1.4logN</b>	<b>log N</b>
<b>Puna-musta puu</b>	<b>2log N</b>	<b>log N</b>
<b>Erillinen ketjutus</b>	<b>N</b>	<b>N / M</b>
<b>Lineaarinen kokeilu (<math>\alpha = 0.9</math>)</b>	<b>N</b>	<b>5</b>
<b>Kaksoishajautus (<math>\alpha = 0.99</math>)</b>	<b>N</b>	<b>5</b>
<b>Digitaalinen hakupuu</b>	<b>b</b>	<b>log N</b>
<b>Radix search trie</b>	<b>b</b>	<b>log N</b>

## **Muita näkökulmia sovellettavuuteen**

- **Linkitetty lista**
  - **Yksinkertainen toteuttaa**
  - **Pienille sovelluksille ( $N < 20$ )**
  - **Kannattaa pitää järjestyksessä**
  - **Voidaan järjestää myös aktiviteetin mukaan**
  
- **Binäärihaku / interpolaatiohaku taulukosta**
  - **Staattisille sovelluksille, joissa vähän lisäyksiä ja poistoja**
  - **Interpolointi vie aikaa, joten sitä ei kannata käyttää aivan pienillä aineistoilla**

- **Binäärinen hakupuu**
  - **Suhteellisen yksinkertainen**
  - **Sopii moniin sovelluksiin, joissa ei ehdottomia aikavaatimuksia**
  - **Voi käyttäytyä huonosti varsin tavallisilla aineistoilla**
  
- **Tasapainotetut puut**
  - **Sovelluksiin, joissa aikavaatimus on ehdoton**
  - **Melko monimutkaisia toteuttaa**
  - **Sopeutuvat hyvin dynaamisiin tilanteisiin**

- **Hajautus**
  - **Sopii moniin sovelluksiin ja on yleensä hyvin tehokas hakumenetelmä**
  
- **Hajautusta ei kannata käyttää:**
  - **aikakriittisissä sovelluksissa (ilman virittelyä)**
  - **jos tarvitaan mahdollisuus käsitellä tietoa järjestyksessä**
  - **jos tiedon määrää ei voida ennakoida ollenkaan**

## Hajautusmenetelmän valinnasta:

- **Erillinen ketjutus**
  - **Yksinkertaisin toteuttaa ja joustaa hyvin tiedon määrän vaihdellessa**
  - **Listat voi järjestää myös aktiviteetin mukaan itseorganisaation avulla**
  - **Listat voi korvata puilla, mikä parantaa worst casea.**
  
- **Avoin osoitus**
  - **Taulukon täyttöasteen tulisi olla alle 0.5**
  - **Lineaarista kokeilua ei kannata käyttää**
  - **Neliöllinen kokeilu on tehokkaampi kuin kaksoishajautus**

- **Digitaaliset hakupuut**
  - Hyvät worst case- ja average case- tehokkuusluvut
  - Toteutus edellyttää tehokasta pääsyä käsiksi bitti-informaatioon
  - Melko monimutkaisia toteuttaa
  - Pitkille avaimille kannattaa käyttää vain Patricia-puuta
  - Useita erilaisia muunnelmia, joihin kannattaa perehtyä, jos hakee todella tehokasta rakennetta.
  
- **Ulkoiset hakumenetelmät**
  - Käytännön valinta on B-puu tai laajennettu hajautus
  - B-puusta on erilaisia virityksiä, joihin kannattaa perehtyä etukäteen