6.3.3 Avaimen poisto punamustasta puusta

- Täydennetään kalvojen 6.1.4 algoritmia palauttamaan kalvojen 6.3.1 punamustat ehdot 1–5 voimaan avaimen poiston jälkeen.

- Muutos kalvojen 6.1.4 algoritmiin:

  Lisätään loppuun riviksi $22\frac{1}{2}$:

  if $y.\text{color} = \text{BLACK}$ **then**
  RbDeleteFixup$(r, x, y.\text{parent})$
  end if;

  joka kutsuu tasapainotusvaihetta poiston jälkeen, jos ehto 5 saattoi rikkoutua.

- Tasapainotusvaiheen invariantti:

  – Puu jonka juuri on r toteuttaa muut ehdoista 1–5 kuin 5.

  – Myös ehto 5 olisi voimassa, jos solmun x kohdalla saisi laskea mustien määrä $+1$.

  Eli jos solmussa x olisi *ylimääräistä **musta**a* väriä.

- Mielenkiintoinen kohta x

  – on tasapainotuksen alussa

  joko poistuneen solmun y ainoa aito lapsi ($\neq$ NULL)

  tai NULL jos sellaista ei ollut.

  – omistaa isän p joka on

  joko $x.$parent jos $x \neq$ NULL

  tai $y.$parent jos $x =$ NULL.

  – liikkuu puussa, kunnes on

  joko juuri, josta ylimääräinen musta voidaan unohtaa

  tai punainen, johon ylimääräinen musta voidaan sijoittaa.

- Mielenkiintoinen kohta x liikkuu puussa

  – vaihtamalla solmujen värejä

  – kalvojen 6.2.1.1 kierroilla.

  – ehdot säilyttäen.

- Jos mielenkiintoinen kohta x on

  – yhä musta

  – mutta ei vielä juuri

  (eli tasapainotusvaihe on yhä kesken), niin

  – kohdalla x on aito velisolmu w

  – veljellä w on aidot lapsisolmut

  ehdon 5 ja ylimääräisen mustan nojalla.

- Tasapainotusvaihe jakautuu 4 tapaukseen veljen w ja sen lasten värien mukaan.

- Algoritmi etenee

  joko nostamalla kohtaa x ylöspäin (tapauksessa 2)

  tai palauttamalla aikaisempi tapaus pysähtyvään myöhäisempään tapaukseen.

1. Veli w on punainen (jolloin isä p on musta):

   - Vaihdetaan veljeä siten, että uusi veli onkin musta:

   (a) Isä p ja vanha veli w vaihtavat värejään keskenään.

   (b) Kierretään isää p vasemmalle.

   (c) Kohdan x uudeksi veljeksi tulee vanhan veljen w vasen lapsi.

   Se on musta ehdon 4 nojalla, koska sen vanha isä w oli punainen.

   Isä p säilyi samana solmuna mutta *punastui!*

   - Tuloksena on jokin tapauksista 2–4:

   – Tapaus 2 osoitetaan *pysähtyväksi silloin kun isä p on punainen!*

   – Tapaukset 3 ja 4 osoitetaan pysähtyviksi joka tapauksessa.

   Siis tämä tapaus 1 on *pysähtyvä vaikka kohta x siirtyy alaspäin puussa!*

Tämä tehdään riveillä 5–8.

```
procedure RbDeleteFixup(r: juuriosoitinviite,
x: solmuviite, p: solmuviite)
 1: while p ≠ NULL and (x = NULL or
    x.color = BLACK) do
 2:    if x = p.left then
 3:       w := p.right;
 4:       if w.color = RED then
 5:          w.color := BLACK;
 6:          p.color := RED;
 7:          LeftRotate(r, p) kalvoilta 6.2.1.1;
 8:          w := p.right
 9:       end if;
10:       if w.left.color = BLACK and
          w.right.color = BLACK then
11:          w.color := RED;
12:          x := p; p := x.parent
13:       else
14:          if w.right.color = BLACK then
15:             w.left.color := BLACK;
16:             w.color := RED;
17:             RightRotate(r, w)
                vasen/oikea-symmetrisesti kalvoilta 6.2.1.1;
18:             w := p.right
19:          end if;
20:          w.color := p.color;
21:          p.color := BLACK;
22:          w right.color := BLACK;
23:          LeftRotate(r, p) kalvoilta 6.2.1.1;
24:          x := r; p := x.parent
25:       end if
26:    else
27:       kuten rivit 3–27 vasen/oikea-symmetrisesti
28:    end if
29: end while;
30: if x ≠ NULL then
31:    x.color := BLACK
32: end if.
```


2. Veli w on musta, ja sen lapset ovat:

   vasen musta

   oikea musta.

   - *Nostetaan* ylimääräinen **musta** ylöspäin puussa:
     - värjätään veli w punaiseksi.
     - siirretään kohta x isään p.
   - Jos ylimääräinen **musta** päätyi

     juureen tai punaiseen solmuun, niin rivin 1 silmukkaehto lopettaa tasapainotuksen.

     Riveillä 30–32 **musta** sijoitetaan sellaiseen solmuun, jossa se ei enää olekaan ylimääräistä.

     muuhun mustaan solmuun, niin rivien 1–29 tasapainotussilmukka jatkuu.

   Tämä tehdään riveillä 11–12.

3. Veli w on musta, ja sen lapset ovat:

   vasen punainen

   oikea musta.

   - Vaihdetaan veljeä siten, että
     - uusikin veli on yhä musta
     - mutta sen oikea lapsi onkin nyt punainen

     eli palautetaan tämä tapaus tapaukseen 4.
   - Tämä onnistuu kun
     - vanha veli w punastuu
     - sen vanha vasen lapsi mustuu
     - vanhaa veljeä w kierretään oikealle.

   Tämä tehdään riveillä 15–18.

4. Veli w on musta, ja sen lapset ovat:

   vasen kumpaa väriä tahansa

   oikea punainen.

   - Ylimääräisestä **musta**sta päästään eroon tuomalla solmun x yläpuolelle uusi musta solmu (sotkematta muita ehtoja):
     - Veljen w oikea lapsi mustuu.
     - Isä p ja veli w vaihtavat keskenään värejään.
     - Isää p kierretään vasemmalle.
     - Silloin veljen w vanha vasen lapsi siirtyy mustuneen isän p uudeksi oikeaksi lapseksi, eli sen(kään) musta korkeus ei muutu.
   - Rivien 1–29 tasapainotussilmukasta voidaan nyt poistua.

   Tämä tehdään riveillä 20–24.

Askelten lukumäärä:

- Algoritmi toistaa mustassa tapauksessa 2 kohdan x nostoa
- kunnes x päätyy

  joko juureen

  tai johonkin punaiseen pysähtyvään tapaukseen 1, 3 tai 4.
- Siis yhtälö (17) pätee jälleen.

Kiertojen lukumäärä:

Pahin mahdollisuus on

tapaus 2
tapaus 2
tapaus 2
⋮
tapaus 2
tapaus 1 $\xrightarrow{\text{kierto}}$ tapaus 3
$\xrightarrow{\text{kierto}}$ tapaus 4
$\xrightarrow{\text{kierto}}$ lopetus

eli yhteensä 3 kiertoa.

6.3.4 Punamusta puu prioriteettijonon talletusrakenteena

- Punamustien puiden perusoperaatiot eli solmun

  lisäys kalvoilta 6.3.2

  poisto kalvoilta 6.3.3

  – käyttävät tasapainotukseen vain puun rakennetta, ei avainten järjestystä

  – tekevät vain vakiomäärän kiertoja.
- Sen vuoksi punamustat puut soveltuvat kalvojen 5.4.3 prioriteettijonojen *talletusrakenteeksi*.

  – Prioriteettijonon maksimikokoa ei enää tarvitse tietää etukäteen.

  – Operaatioiden $\mathcal{O}$-aikavaatimukset säilyvät.

  (Mutta osoittimien käsittely lienee vakiotekijän verran hitaampaa kuin taulukkoindeksien.)

- Perusidea on yksinkertainen:

  – Puu on kalvojen 5.4.1.1 mukainen kekopuu.

  – Kalvojen 5.4.1.2 mukainen puussa kulkeminen tehdään

  * ei enää indeksiaritmetiikalla vaan
  * seuraamalla vastaavia osoittimia.

  – Prioriteettijono-operaatiot toteutetaan vastaavilla puun operaatioilla.

  Prioriteetin lisäys on vastaavan solmun lisäys.

  * Se NULL johon solmu lisätään voidaan valita mielivaltaisesti.
  * Esimerkiksi vasemman- tai oikeanpuoleisin kalvojen 6.1.2.1 tapaan.

  Prioriteetin poisto on vastaavan puusolmun poisto.

- Kekojärjestys on vaarassa, kun puuta muokataan:

  kierroissa joita tehdään kalvojen 6.3.2 ja 6.3.3 tasapainotusvaiheissa.

  poiston alussa kun vaihdetaan 2-lapsinen solmu seuraajansa kanssa:

  – Kalvojen 6.1.4 riveillä 4–6.

  – Muunnetaan vaihto

  loogisesta tietokenttien kopioimisesta

  fyysiseksi itse solmujen vaihdoksi.

  lisäyksessä kun uusi avain tuottaa uuden lehden.

  Siis solmut vastaavat taulukkopaikkoja.
- Kun taulukkopaikkojen $A[i]$ ja $A[j]$ sisällöt pitää vaihtaa keskenään (kuten kalvojen 5.4.1.3 ja 5.4.3 algoritmeissa), niin *vaihdetaan vastaavat fyysiset solmut* i ja j keskenään.

procedure swapNodes(r: **juuriviite**,
i: **solmuosoitin**,
j: **solmuosoitin**)
Vaihda kenttien i.color ja j.color sisällöt keskenään;
Vaihda kenttien i.left ja j.left sisällöt keskenään;
Vaihda kenttien i.right ja j.right sisällöt keskenään;
Vaihda kenttien i.parent ja j.parent sisällöt keskenään;
if $r = i$ **then**
r := j
else if $r = j$ **then**
r := i
end if.

- Samalla saadaan yksinkertainen toteutus kalvoilla 5.4.3 kaivatuille kahvoille: osoitin vastaavaan fyysiseen solmuun.

6.3.4.1 Kekojärjestyksen palauttaminen

Kierron jälkeen riittää kutsua

- kalvojen 5.4.1.3 algoritmia heapify (puuosoitinversiona)
- kiertosolmun z uudessa isässä z.parent (eli solmun z vanhassa lapsessa kiertosuuntaa vastaan).

Poiston alussa riittää kutsua

- kalvojen 5.4.1.3 algoritmia heapify
- puuhun jäävässä solmussa (eli siinä seuraajasolmussa joka korvasi poistettavan solmun).
- ennen tasapainotusvaihetta.

Lisäyksessä riittää kutsua

- kalvojen 5.4.3 algoritmia shiftup
- ennen tasapainotusvaihetta.

- Jokainen kekojärjestyksen palauttaminen vie

askeleita $= \mathcal{O}$(punamustan puun korkeus)
$= \mathcal{O}$(log(prioriteettijonon alkioiden lukumäärä))
lauseesta 6.3.1.

- Kekojärjestys täytyy palauttaa enintään

≤ 4 kertaa

yhden prioriteettijono-operaation aikana.

- Yksi prioriteettijono-operaatio vie siis

$\mathcal{O}$(log(prioriteettijonon alkioiden lukumäärä))

askelta toteutettuna punamustilla puilla.

Siis saman kuin toteutettuna taulukolla kalvojen 5.4.3 tapaan.

6.3.4.2 Sopivan puun luonti

- Kalvoilla 5.4.1.4 annettiin

$\mathcal{O}(n)$

algoritmi luoda taulukkoon talletettu (keko eli) prioriteettijono annetusta n alkion syöteaineistosta.

- Muunnetaan se toimimaan kalvojen 6.3.4 punamustaan puuhun talletetulle prioriteettijonolle.

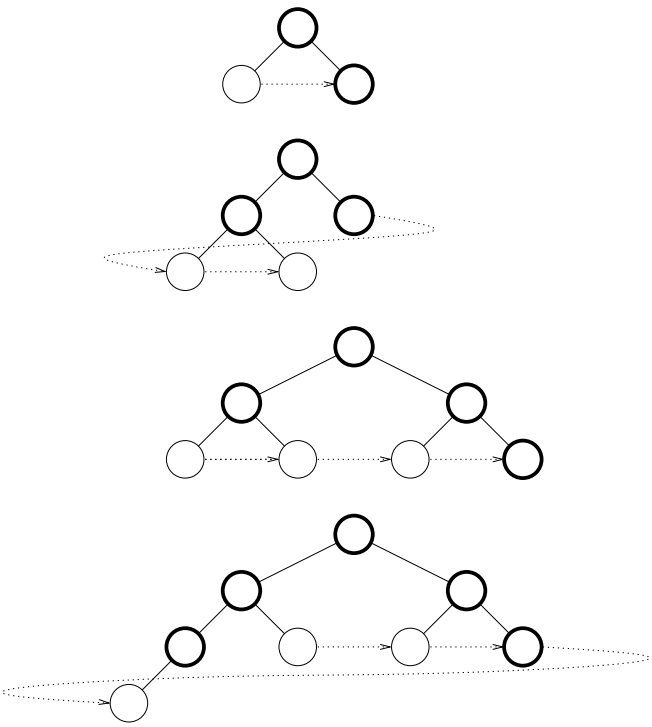
- 3 vaihetta:

1. Luodaan n-alkioinen punamusta puu ilman avaimia (ja sisältöjä).
2. Kopioidaan syötteen n avainta (ja sisältöä) vaiheessa 1 luotuun puuhun kekoehdosta välittämättä.
3. Asetetaan kekoehto voimaan vaiheessa 2 täytetyssä puussa.

- Vaihe 1:
  - Luodaan n (sisä)solmun kalvojen 5.4.1.2 mukainen melkein täydellinen binääripuu.
  - Se on punamusta kun
    * vajaan (alimman) syvyystason (aidot) solmut ovat punaisia
    * täysien (ylempien) syvyystasojen (aidot) solmut ovat mustia.
  - Puun solmut luodaan kalvojen 5.4.1.2 kasvavassa numerojärjestyksessä:
    * Pidetään kalvojen 4.3 jonossa niitä solmuja, joilla ei vielä ole molempia (aitoja) lapsia.
    * Jonossa olevat solmut ovat
      · muuten punaisia
      · mutta syvyystason viimeinen (oikeanpuoleinen) solmu onkin musta

      jotta tiedetään, milloin taso vaihtuu.

if $n = 0$ **then**
  r := NULL
else
  r := **new** juurisolmu s jolla
  s.left := NULL, s.right := NULL,
  s.parent := NULL ja s.color := BLACK;
  Q := makeEmptyQueue( );
  enqueue(Q, r);
  for $n - 1$ **times do**
    t := **new** solmu s jolla s.left := NULL,
    s.right := NULL ja
    s.parent := front(Q);
    if front(Q).left = NULL **then**
      front(Q).left := t;
      t.color := RED
    else
      front(Q).right := t;
      t.color := front(Q).color;
      front(Q).color := BLACK;
      dequeue(Q)
    end if;
    enqueue(Q, t)
  end for;
  while front(Q).color = RED **do**
    front(Q).color := BLACK;
    dequeue(Q)
  end while
end if;
return r.



- Vaihe 2:
  - Jos syötetaulukko $A[1 \ldots n]$ on kalvojen 5.4.1.2 keko
    * niin puun täyttäminen voidaan tehdä samassa järjestyksessä kuin sen luonti vaiheessa 1.
    * jolloin prioriteettijono valmistuu ilman vaihetta 3.
  - Esimerkki puun *leveys*suuntaisesta läpikäynnistä (*breadth* first traversal):

    Seuraavaksi käsitellään (lähtösolmua eli) *juurta lähin* vielä käsittelemätön solmut.
  - Kalvojen 6.1.2 esi-, sisä- ja jälkijärjestykset ovat *syvyys*suuntaisia läpikäyntejä (*depth* first traversal):

    Seuraavaksi käsitellään nykyisen solmun lapsisolmut.
  - Läpikäyntiperiaatteiden suhde:

suunta	toteutus
syvyys	kalvojen 4.2 tai rekursio-*pino*lla
leveys	kalvojen 4.3 *jono*lla

- Jos syötteen n avainta (ja sisältöä) ovat *kasvavassa* järjestyksessä, niin vastaava *hakupuu* täyttyy kalvojen 6.1.2 *sisä*järjestyksessä:

  Aseta syötteen käsittelykohta sen alkuun;
  fillTree(juuri)

  missä
  procedure fillTree(s: **solmuviite**)
  if $s \neq$ NULL **then**
  fillTree(s.left);
  Kopioi syötteen seuraava
  avain/sisältö-pari solmuun s;
  fillTree(s.right)
  end if.

- Vaihe 3:
  - Kalvojen 5.4.1.4 algoritmi muunsi syötetaulukon $A[1 \ldots n]$ kekojärjestykseen.

    Sovitetaan se punamustaan puuhun talletetulle keolle.
  - Idea: kutsutaan solmussa kalvojen 5.4.1.3 aliohjelmaa, mutta vasta sitten kun solmun alipuut ovat jo kekoja.

    Käytetään aliohjelman puuversiota.
  - Sama voidaan tehdä käymällä puu läpi kalvojen 6.1.2 jälkijärjestyksessä: ensin solmun alipuut rekursiivisesti, vasta lopuksi solmu itse.

    procedure buildHeapTree(s: **solmuviite**)
    if $s \neq$ NULL **then**
    buildHeapTree(s.left);
    buildHeapTree(s.right);
    heapifyTree(s)
    end if.
  - Aikavaatimus on sama yhtälö (11) koska vaiheessa 1 luotiin taulukon A muotoinen puu.

7 Hajautus

- Kalvoilla 5.6.2 saatiin tehokkaampia järjestämisalgoritmeja kuin kalvojen 5.6 alaraja, kun oletettiin lisäksi että *avaimilla saa suorittaa laskutoimituksia*.

- Samasta ideasta saa myös *hajautuksen* (engl. hashing, viroksi paiskonta) vaihtoehdoksi kalvojen 6 hakupuille:
  - Yleensä luovutaan mahdollisuudesta luetella tietorakenteeseen talletettu sisältä avainjärjestyksessä kalvojen 6.1.2 tapaan.

    Eli keskitytään vain *hakuun*, *lisäykseen* ja *poistoon*.
  - Tavoitteena on (logaritmisiakin nopeammat) *vakio*aikaiset operaatiot.

    Mutta vain *keskimääräisessä* (eikä myös pahimmassakin) tapauksessa.

- Hajautuksen perusidea:
  - Talletusrakenne on pieni taulukko $T[0 \ldots m-1]$, hajautus*taulu*.
  - Mahdollisten avainten joukko U on suuri.
  - Hajautus*funktio*

    $$h\colon U \to \{0, 1, 2, \ldots, m-1\}$$

    kertoo, että alkio $x \in U$ kuuluu paikkaan $T[h(x)]$.

    Tämä h on se laskutoimitus avaimilla.
  - h valitaan laskettavaksi vakioajassa $\mathcal{O}(1)$.

- Ongelmana *yhteentörmäykset* (collision):

  Koska $|U| > m$, on avaimia $x \neq y \in U$, jotka kuuluisivat samaan paikkaan.

  Nämä yhteen törmänneet avaimet x ja y voidaan tallettaa

  ylivuotoketjuun

  hajautustauluun

  ja saadaan 2 erilaista hajautustapaa.

7.1 Yhteentörmäykset ylivuotoketjuun

- Tässä tavassa hajautustaulun T paikka $T[j]$ sisältää
  - kalvojen 4 mukaisen listan kaikista niistä myöhemmin talletetuista avaimista $y_1, y_2, y_3, \ldots, y_{m_j}$
  - jotka törmäsivät yhteen paikkaan $T[j]$ ensimmäiseksi tulleen avaimen x kanssa
  - eli joilla $h(y_1) = h(y_2) = h(y_3) = \cdots = h(y_{m_j}) = h(x) = j$.
- Tämä ketjutus voi olla

  suora jolloin paikassa $T[j]$ on x ja ylivuotoketju $y_1, y_2, y_3, \ldots, y_{m_j}$

  erillinen jolloin paikassa $T[j]$ on vain kaikkien siihen kuuluvien avainten ketju $x, y_1, y_2, y_3, \ldots, y_{m_j}$.

  Erillinen on yksinkertaisempi toteuttaa ja analysoida.

- Operaatiot ovat nyt:

  Haku: Etsi annettua avainta a paikassa $T[h(a)]$ olevasta ketjusta.

  Lisäys: Etsi annettua avainta a paikassa $T[h(a)]$ olevasta ketjusta.

  Jos sellaista listasolmua ei löytynyt, niin lisää se tähän ketjuun.

  Poisto: Etsi annettua avainta a paikassa $T[h(a)]$ olevasta ketjusta.

  Jos sellainen listasolmu löytyi, niin poista se tästä ketjusta.

  Solmun lisäys ja poisto voidaan tehdä vakioajassa (esimerkiksi) katsomalla edeltäjään kuten kalvoilla 4.7.

- On siis selvitettävä *haku*un kuluva aika.
- Muistipaikkoja kuluu

$$\mathcal{O}(m + n). \tag{18}$$

- Tässä $n =$ hajautustaulussa T tällä hetkellä olevien avainten lukumäärä.

- Oletetaan yksinkertaisuuden vuoksi (simple uniform hashing) että:

  Avaimiston U todennäköisyysjakauma on sellainen, että
  - kun valitaan satunnaisesti avain $z \in U$
  - niin jokainen arvo $h(z)$ on yhtä todennäköinen

  eli h jakaa avaimet U tasaisesti paikkoihin $0, 1, 2, \ldots, m-1$.

  Hajautusfunktio h on sellainen, että seuraavan avaimen z paikka $h(z)$ ei riipu aikaisemmin käsitellyistä avaimista.

  Eli että h *ei mukaudu*
  - hajautustaulun T nykyiseen tilaan
  - avaimiston U todennäköisyysjakaumaan.

- Hajautustaulun T tämänhetkinen *täyttösuhde*

$$\alpha = \frac{n}{m} \tag{19}$$
$$= \text{ketjun pituus keskimäärin}$$

  edellisten oletusten nojalla.

- Silloin saadaan, että

  tuloksekas

  tulokseton

  haku vie keskimäärin

$$\mathcal{O}(1 + \alpha)$$

  askelta.

  (Todennäköisyyslaskelmat sivuutetaan.)

- Jos siis hajautustaulun T koko on

$$m = \Theta(n) \tag{20}$$

  niin
  - operaatiot vievät luvatun vakioajan
  - tilantarve (18) pysyy lineaarisena.

7.2 Hajautusfunktioita

- Hyvä hajautusfunktio
  - on nopea laskea
  - aiheuttaa vain vähän yhteentörmäyksiä.
- Yhteentörmäyksien yleisyys *riippuu myös siitä jakaumasta josta syöte tulee:*
  - Kuinka todennäköistä on, että yhteen törmäävät alkiot esiintyvät yhdessä?
  - Kuinka hyvin hajautusfunktio onnistuu erottamaan toisistaan todennäköisesti yhdessä esiintyvät avaimet?
- Hyvä hajautusfunktio
  - tiivistää saamaansa avainta
  - hävittää tiivistyksessä mahdollisimman vähän avaimen informaatiosta
  - käyttää jäljelle jäävää informaatiota avainten erottamiseen toisistaan.

Jakojäännösmenetelmä (division method)

$$h(x) = x \bmod m$$

missä hajautustaulun pituus m

- on alkuluku
- joka ei ole lähellä mitään luvun 2 potenssia.

Kertolaskumenetelmä (multiplication method)

$$h(x) = \lfloor m \cdot (x \cdot A - \lfloor x \cdot A \rfloor) \rfloor$$

missä $0 < A < 1$ on jokin liukulukuvakio.

- Taulun pituuden m saa valita vapaasti.

  Tyypillisesti $m = 2^p$.
- Hyvä A riippuu syötteestä.
- Kultainen leikkaus

$$A = \frac{\sqrt{5} - 1}{2} = 0.618033988749894\ldots$$

on todennäköisesti riittävän hyvä.

Universaali hajautus (universal hashing) ehkäisee syötteen jakauman vaikutusta *satunnaisuutta* hyödyntäen:

- Ohjelman käynnistyessä *arvotaan hajautusfunktio* jota käytetään tällä ajokerralla.

  Käytännössä arvotaan hajautusfunktion parametrit satunnaislukugeneraattorilla.
- Hajautus toimii eri tavoin eri ajokerroilla:
  - joku arvottu funktio voi olla huono syötteen jakaumalle

    jolloin tämä ajokerta on hidas
  - mutta seuraava ajokerta tehdäänkin toisin

    jolloin seuraava ajokerta ei kompuroi samasta syystä
  - kun syöte on samasta jakaumasta molemmilla ajokerroilla.

- Hajautusfunktio arvotaan *universaalista* kokoelmasta $\mathcal{H}$:
  - $\mathcal{H}$ on äärellinen kokoelma hajautusfunktioita

$$h\colon U \to \{0, 1, 2, \ldots, m-1\}.$$

  - Kiinnitetään mielivaltainen avainpari $x \neq y \in U$.
  - On $\leq \frac{|\mathcal{H}|}{m}$ kappaletta sellaisia funktioita $h \in \mathcal{H}$ joilla $h(x) = h(y)$.
  - Intuitio: koko $\mathcal{H}$ ei törmäytä yhteen mitään avainparia x, y enempää kuin mitään muuta avainparia x', y'.
  - Todennäköisyys yhteentörmäykselle $h(x) = h(y)$ on $\frac{1}{m}$ kun $h \in \mathcal{H}$ valitaan satunnaisesti.

    (Kun jokainen $h \in \mathcal{H}$ on yhtä todennäköinen valinta.)

- Yksinkertainen universaali $\mathcal{H}$ on seuraava:
  - Olkoon avaimisto $U = \{0, 1, 2, \ldots, p\}$ missä p on alkuluku.
  - Kokoelma on
    $$\mathcal{H} = \{h_{ab} \colon 1 \leq a < p, 0 \leq b < p\}$$
    missä
    $$h_{ab}(x) = ((a \cdot x + b) \bmod p) \bmod m.$$
- Siis algoritmina:
  1. Valitse alkuluku $p \geq$talletettavien avainten lukumäärä.
     (Tai arvio sille, jos käytät kalvojen 7.1 ylivuotoketjutusta.)
  2. Generoi satunnaisluvut $1 \leq a < p$ ja $0 \leq b < p$.
     (Siten, että jokainen satunnaisluku on yhtä todennäköinen. Käytännössä vakiokirjaston satunnaislukugeneraattori riittänee.)
  3. Käytä hajautusfunktiota h_{ab} tällä ajokerralla.

7.3 Hajautus ja järjestys

- Kalvojen 5.6.2.3 lokerikkojärjestäminen muistuttaa kalvojen 7.1 erillistä ketjutusta:
  - Hajautustauluna T on lokerikko B.
  - Taulun koko on valittu
    $$m = n$$
    yhtälön (20) mukaisesti.
  - Hajautusfunktiona on
    $$h(a) = \lfloor n \cdot a \rfloor . \qquad (21)$$
- Siis lokerikkojärjestämisalgoritmi on tästä näkökulmasta:
  1. Hajauta.
  2. Järjestä kukin ylivuotoketju erikseen.
  3. Yhdistä järjestetyt ketjut peräkkäin.

- Algoritmi toimii, koska hajautusfunktio (21) on *heikosti järjestyksen säilyttävä:*

  Jos $h(x) = h(z)$ ja $x < y < z$, niin myös $h(x) = h(y) = h(z)$.

- Toisin sanoen, tämä hajautusfunktio *ei riko osavälejä:*

  Jos x ja z viedään samaan ketjuun/lokeroon, niin myös kaikki niiden välissä olevat y viedään sinne.

- Jos hajautusfunktio on tällainen, niin hajautustaulu T voi *säilyttää avaintensa järjestyksenkin:*
  - Pidetään jokaisen ylivuotoketjun sisältö avainjärjestyksessä.
  - Ketjujen keskinäinen järjestys määräytyy hajautusfunktiosta.
- Toisaalta tämä lisävaatimus *vaikeuttaa sopivien hajautusfunktioiden rakentamista.*

- Eräs yksinkertainen heikosti järjestyksen säilyttävä hajautusfunktio on seuraava:
  - Hajautustaulu olkoon $T[0 \ldots 2^b - 1]$ (muistiin) sopivalla vakiolla b.
  - Tulkitaan (ei-negatiiviset kokonaisluku)avaimet
    - kiinteän mittaisiksi *bittijono*iksi
    - joissa on vähintään b bittiä
    - lisäämällä alkunollia jos tarpeen.
  - Otetaan hajautusarvoksi b *ylintä* bittiä.
  - Esimerkiksi kun $b = 3$ ja avaimet 6-bittisiä:

x	bitit	$\overleftarrow{h(x)}$	$\overrightarrow{h(x)}$
5	000101	0	0
12	001100	1	4
19	010011	2	2
53	110101	6	3

  Mukana myös tulkinta *nurin päin*, koska silloin uuden bitin mukaan ottaminen on helppoa.

7.4 Avoin hajautus

- Toinen tapa käsitellä yhteentörmäykset hajautuksessa on *tallettaa ylivuotoketju itse hajautustauluun*.

  Avoin hajautus (open addressing).

- Ketjutus voidaan tehdä (kalvojen 4.1 osoittimien sijasta) hajautusfunktiota yleistämällä:

  $h\colon U \times \{0,1,2,\ldots,m-1\} \to \{0,1,2,\ldots,m-1\}$

  missä $h(x,i) =$

  - se paikka hajautustaulussa $T[0 \ldots m-1]$
  - josta pitää seuraavaksi katsoa avainta x jos kaikki i edellistä katsomista ovat epäonnistuneet.

- Avaimen $x \in U$ *kokeilujono* (probe sequence)

  $$h(x,0), h(x,1), h(x,2), \ldots, h(x,m-1).$$

  korvaa nyt ylivuotoketjun.

- Kokeilujonon pitäisi käydä läpi (jossakin järjestyksessä) kaikki indeksit $0, 1, 2, \ldots, m-1$ (eli olla niiden permutaatio).

  Muuten hajautustaulun T paikkoja ei käytetä tarkkaan.

- Taulukkopaikan $T[j]$.status =

  VAPAA: Tähän paikkaan *ei vielä* ole kertaakaan talletettu avainta.

  VARATTU: Tässä paikassa on *parhaillaan* tallessa avain (ja sitä vastaava sisältö).

  POISTETTU: Tässä paikassa on *aikaisemmin* ollut tallessa avain, mutta se on sittemmin poistettu.

  Muuten emme tietäisi tarkoittaako $T[j]$.status = VAPAA ylivuotoketjun korvikkeen

  - loppua
  - keskellä olevaa aukkoa.

- Silmukan

```
j := m;
i := 0;
while i < m do
   k := h(x,i);
   if (T[k].status = VARATTU and T[k].key = x)
   or (T[k].status ≠ VARATTU and j = m) then
      j := k
   end if;
   if (T[k].status = VARATTU and T[k].key = x)
   or (T[k].status = VAPAA) then
      i := m
   else
      i := i + 1
   end if
end while
```

  päätteeksi

  joko $j = m$ jolloin hajautustaulu T

  - on jo täynnä
  - eikä sisällä avainta x

  tai avain x

  - on jo talletettu
  - tai voidaan tallettaa

  paikkaan $T[j]$.

- Avoin hajautus ei siis ole hyvä, jos poistoja on paljon:

  Hajautustaulun T paikkoja kokeillaan ikään kuin poistoja ei olisikaan tehty.

- Avoin hajautus ei siis laajene automaattisesti:

  - Tauluun T yhtä aikaa mahtuvien avainten lukumäärällä on yläraja $n \leq m$.
  - Eli yhtälön (19) täyttösuhde $\alpha \leq 1$.
  - Täyttä taulua T voidaan *kasvattaa* tarvittaessa:
    * Varataan uusi, suurempi hajautustaulu U.
    * Talletetaan nykyisen taulun T avaimet a (ja sisällöt) tauluun U.
    * On olemassa hajautusmenetelmiä, joilla avaimia a ei tarvitse hajauttaa uudelleen.

      Esimerkiksi kalvojen 7.3 nurin päin luettu bittijono $\overleftarrow{h(x)}$.

7.4.1 Dynaamisesti pidentyvä (ja lyhentyvä) taulukko

- Olkoon avoin hajautustaulu $T[0 \ldots m-1]$ jo täynnä.

  Siihen täytyisi lisätä vielä avaimia.

- Taulun T pidentäminen tauluksi $U[0 \ldots m-1+\delta]$ (missä $\delta \geq 1$ on pidennys) vaatii

  $$\Theta(m+\delta)$$

  askelta

  – uusien paikkojen alustukseen

  – vanhan sisällön kopioimiseen.

  (Jätetään käyttöjärjestelmän/virtuaalikoneen muistinhallinnan tekemä työ huomiotta.)

- Ohjelmointikielen Java vakiokirjastossa on tällainen luokka `Vector`.

- *Ei* kannata pidentää vain $\delta = 1$ paikka kerrallaan.

- Silloin voisi pahimmillaan käydä näin:

1. lisäys	$\Theta(m+1)$
2. lisäys	$\Theta(m+2)$
3. lisäys	$\Theta(m+3)$
⋮	⋮
m. lisäys	$\Theta(m+m)$
m lisäystä	$\Theta(m^2)$ askelta

  – Eli tehdään $\Theta(m)$ askelta pidennystyötä *jokaisella* lisäyksellä!

  – Eli luovutaan hajautuksen tavoitteesta saada keskimäärin vakioaikaiset operaatiot!

- Sama ongelma on kaikilla *vakioilla* δ.

- Java-konstruktori `Vector(`m`,`δ`)` seuraa tätä pidennysstrategiaa.

- Sen sijaan *kaksinkertaistetaan* taulun pituus aina kun se osoittautuu liian pieneksi:

  $$\delta = \text{taulun nykyinen pituus.}$$

- Silloin pahimmillaan käykin näin:

1. lisäys	$\Theta(2 \cdot m)$
2. lisäys	mahtuu, ei pidennetä
3. lisäys	mahtuu, ei pidennetä
⋮	⋮
m. lisäys	mahtuu, ei pidennetä
$m+1$. lisäys	$\Theta(4 \cdot m)$
$m+2$. lisäys	mahtuu, ei pidennetä
$m+3$. lisäys	mahtuu, ei pidennetä
⋮	⋮
$3 \cdot m$. lisäys	mahtuu, ei pidennetä
$3 \cdot m+1$. lisäys	$\Theta(8 \cdot m)$
$3 \cdot m+1$ lisäystä	$\Theta((\underbrace{2+4+8}_{14}) \cdot m)$ ask.

- Java-konstruktori `Vector(`m`)` seuraa tätä pidennysstrategiaa.

- Java-konstruktori `Vector()` seuraa tätä pidennysstrategiaa siten, että ensimmäisen taulukon pituus on oletusarvoinen $m = 10$.

- Kun näin jatketaan p pidennystä, niin lopulta

  – saadaan $2^p \cdot m$-paikkainen taulu

  – $(2^{p-1}-1) \cdot m + 1$ lisäyksellä

  – yhteensä

  $$(\underbrace{2+4+8+\cdots}_{p \text{ kpl.}}) \cdot m$$

  askeleella pidennykseen kuluvaa työtä.

- Yhtä *lisäystä kohti* pidennystyötä tehdään

  $$\frac{(2^{p+1}-2) \cdot m}{(2^{p-1}-1) \cdot m + 1} < 4$$

  askelta.

- Näin saavutetaan jälleen hajautuksen tavoite keskimäärin vakioaikaisista operaatioista:

  silloin tällöin lisäys käyttääkin pidennykseen lineaarisen määrän lisätyötä

  mutta niin harvoin että pitkällä aikavälillä pidennyksiin kuluu vain vakiomäärä lisätyötä operaatiota kohden.