6 Hakupuut

- Kalvoilla 5 tarkasteltiin syöteaineiston järjestämistä.
- Nyt tarkastellaan *valmiin järjestyksen ylläpitoa* kun aineistoon tehdään yhden alkion kokoisia muutoksia.
- Tavoitteena on tietorakenne T jonka nykyinen sisältö

$$a_1 < a_2 < a_3 < \cdots < a_n$$

pidetään

  - järjestyksessä
  - talletettuna niin, että seuraavat operaatiot ovat tehokkaita:

  Haku: "Onko sisällössä jo alkio b?"

  Lisäys: "Lisää sisältöön vielä alkio b (jos sellaista ei siellä vielä ole)!"

  Poisto: "Poista sisällöstä alkio b (jos sellainen siellä vielä on)!"

- Käyttöesimerkki:
  - Tarvitaan pieni *tietokanta* yliopistolaisten
    * henkilötunnus-
    * nimi-

    pareista.
  - Täytyy pystyä

    lisäämään uusi pari

    poistamaan annettua henkilötunnusta vastaava pari

    kysymään annettua henkilötunnusta vastaava nimi

    luettelemaan parit
    * ensin kasvavan iän
    * sitten sukupuolen
    * lopuksi väestörekisteröintijärjestyksen

    mukaisessa järjestyksessä.

- Yleinen tilanne:

  Haluaisimme jonkinlaisen taulukon, mutta
  - emme halua/voi varata omaa paikkaa jokaiselle mahdolliselle indeksiarvolle
  - joista vain murto-osa indeksiarvoista on kerrallaan käytössä.

- Tässä indeksityyppinä on "kaikki mahdolliset henkilötunnukset" joita on noin

$$\left(300\text{ v} \cdot 365\text{ vrk/v} + \frac{300}{4}\text{ karkausvrk}\right) \cdot 1\,000\text{ sarjanumeroa/vrk} = 109\,575\,000\text{ hengelle.}$$

  Yliopistolaisia on vain noin

$$38\,000\text{ opiskelijaa} + 7\,300\text{ henkilökuntaa} = 45\,300\text{ henkeä.}$$

- Tietorakenteen näkökulmasta indeksiä vastaava sisältö (tässä nimi) on passiivinen lisäkenttä.

- Tietorakenneratkaisuja kun pareja on n:

  Järjestämätön lista:

  lisäys $\mathcal{O}(1)$

  poisto $\mathcal{O}(n)$

  kysely $\mathcal{O}(n)$

  luettelu $\mathcal{O}(n \cdot \log(n))$

  Järjestetty lista:

  lisäys $\mathcal{O}(n)$

  poisto $\mathcal{O}(n)$

  kysely $\mathcal{O}(n)$

  luettelu $\mathcal{O}(n)$

  Taulukko järjestettynä ja tiivistettynä:

  lisäys $\mathcal{O}(n)$ — taulukkoa voi joutua kasvattamaan

  poisto $\mathcal{O}(n)$

  kysely $\mathcal{O}(\log(n))$

  luettelu $\mathcal{O}(n)$

- "Tehokkuus" tarkoittaa nyt:

  lisäys $\mathcal{O}(\log(n))$

  poisto $\mathcal{O}(\log(n))$

  kysely $\mathcal{O}(\log(n))$

  luettelu $\mathcal{O}(n)$

- Tiukat vaatimukset (jos saa vain vertailla $a_i \leq a_j$):

  Jos esimerkiksi tietorakenteella T olisi

  lisäys $o(\log(n))$

  luettelu $o(n \cdot \log(n))$

  niin voisimme *järjestää* n alkiota

  1. ensin lisäämällä ne kaikki aluksi tyhjään tietorakenteeseen T yksi kerrallaan
  2. sitten luettelemalla rakenteen T sisältö

  $$o(n \cdot \log(n))$$

  askeleessa, vastoin kalvoja 5.6.

6.1 Hakupuun idea

- Kalvojen 2.1 hajoita ja hallitse -periaate:

  Valitaan talletetusta aineistosta jokin *avain* a_i (key) (= indeksiarvo) ja hajotetaan muu aineisto 2 osaan:

  pienempiin $a_j < a_i$

  suurempiin $a_i < a_k$.

- Esitetään kalvojen 5.1.3.1 binääripuilla nyt *konkreettisena* tietorakenteena muistissa:

  juureen a_i *viitaksi* käännytäänkö

  vasempaan alipuuhun joka on hakupuu alkioille a_j

  oikeaan alipuuhun alkioille a_k.

  Kaaret toteutetaan kalvojen 4.1 osoittimilla.

  Tyhjät alipuut (= osoittimet) sallitaan, eli puu ei ole aito.

- Puu koostuu siis solmutietueista s joissa on seuraavat kentät:
  - $s.$key = solmun viitta
  - $s.$item = solmun viittaan liittyvä sisältö
  - $s.$left = osoitin vasemman alipuun juuritietueeseen, tai NULL jos vasen alipuu on tyhjä
  - $s.$right = oikea alipuu vastaavasti.

6.1.1 Avaimen haku puusta

- Viittoja seuraamalla on helppo löytää annettua avainta a vastaava solmu hakupuun T juuresta alkaen:

```
function
treeSearch(s: solmuosoitin,
           a: avain) : solmuosoitin
  if s = NULL or s.key = a then
    return s
  else if s.key < a then
    return treeSearch(s.left, a)
  else
    return treeSearch(s.right, a)
  end if.
```

- Vie aikaa

  $$\mathcal{O}(\text{koko puun korkeus})$$

  askelta.

- Takarekursiivisena tehtävissä toistorakenteella

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

  tilassa.

6.1.2 Aineiston luettelu järjestyksessä

- Hakupuun T sisällön tulostus järjestyksessä juuresta alkaen käy rekursiivisesti:

  procedure
  treeInorderWalk(s: **solmuosoitin**)
  1: **if** $s \neq$ NULL **then**
  2: treeInorderWalk(s.left);
  3: Tulosta kentät s.key ja s.item;
  4: treeInorderWalk(s.right)
  5: **end if.**

- Esimerkki binääripuun *läpikäynnistä* (traversal) *sisäjärjestyksessä* (inorder):

  Juurisolmu käsitellään alipuiden käsittelyn *välissä*.

  Esijärjestyksessä (*pre*order) juuri käsitellään *ennen* alipuita.
  (Rivit 2 ja 3 vaihtavat paikkaa.)

  Jälkijärjestyksessä (*post*order) juuri käsitellään alipuiden *jälkeen*.
  (Rivit 3 ja 4 vaihtavat paikkaa.)

- Vie aikaa

  $$\mathcal{O}(n)$$

  askelta, koska
  - koko puussa on solmuja n kappaletta
  - jokaisessa solmussa s käydään 3 kertaa:
    1. tullaan isäsolmusta rekursiolla
    2. palataan vasemman alipuun rekursiosta
    3. palataan oikean alipuun rekursiosta.

- Vie tilaa rekursiopinosta

  $$\Theta(\text{koko puun korkeus})$$

  paikkaa.

- Jos poistetaan rekursio kalvojen 4.6 tapaan, niin havaitaan:

  Solmun s kohdalla rekursiopino on polku puun T juuresta solmuun s takaperin.

  Havainto pätee *kaikille* rekursiivisille puualgoritmeille!

- Tämä polku solmusta s takaisin puun T juureen on hyödyllinen:
  - Voidaan korvata rekursio toistolla ilman pinoa.
  - Voidaan kirjoittaa sellaisia algoritmeja, jotka
    * aloittavat toimintansa keskellä puuta olevasta solmusta p puun T juuren r sijasta
    * toimivat silti kuin p olisi löydetty hakemalla juuresta r alkaen kalvojen 6.1.1 tapaan.
  - Sellaisilla algoritmeilla voidaan toteuttaa kalvojen 4.7 tyylisiä *puu*iteraattoreita.

- Siksi usein lisätäänkin solmutietueisiin s vielä kenttä
  - s.parent = osoitin tietueen s isäsolmun tietueeseen, tai NULL jos s on koko puun T juuren tietue.

6.1.2.1 Puuiteraattorit

- Kalvojen 6.1.2 tulostus puuiteraattoreilla:

  I := first(T);
  while valid(I) **do**
  Tulosta kentät retrieve(I).key
  ja retrieve(I).item;
  I := next(I)
  end while.

- Puuiteraattorein perusoperaatiot:
  - first(T) palauttaa iteraattorin, joka osoittaa puun T avainjärjestyksessä *ensimmäiseen* solmuun.
  - retrieve(I) palauttaa sen solmun, johon iteraattori I nyt osoittaa.
  - next(I) palauttaa iteraattorin, joka osoittaa avainjärjestyksessä *seuraavaan* solmuun puussa kuin I.
  - valid(I) on epätotta, jos iteraattori I osoittaa puun (avainjärjestyksessä viimeisen solmun) *ohi*.

- Vie saman ajan

$$\mathcal{O}(n)$$

samalla "kosketa kolmesti" -argumentilla.

- Vie enää tilan

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

kunhan myös iteraattori I tarvitsee vain vakiotilan.

- Iteraattoriksi I riittää yksi **solmuosoitin**.

6.1.2.2 Puuiteraattorin operaatiot

- Voimassaolotesti:

function valid(I: **solmuosoitin**): **boolean**
  return $I \neq$ NULL.

- Iteraattori koko puun T alkuun:
  - Koko puun T avainjärjestyksessä ensimmäinen solmu on *vasemmanpuoleisin* solmu.
  - Kuljetaan siis puun T juuresta r vasemmalle niin pitkään kuin mahdollista.

function
first(r: **solmuosoitin**): **solmuosoitin**
  if $r \neq$ NULL **then**
    while r.left $\neq$ NULL **do**
      r := r.left
    end while;
  end if;
  return r.

- Iteraattorin I osoittamaa solmua seuraava suurempi solmu:
  - Jos solmun I oikea alipuu on epätyhjä, niin sen *pienin* solmu.
  - Jos tyhjä, niin solmun I lähin *esi-isä* (ancestor)

    I itse, I:n isä, I:n isoisä, I:n isoisoisä,...

    johon kiivettiin *vasemmalta*.

function
next(I: **solmuosoitin**): **solmuosoitin**
  if I.right $\neq$ NULL **then**
    return first(I.right)
  else
    P := I.parent;
    while $P \neq$ NULL **and** $I = P$.right **do**
      I := P;
      P := I.parent
    end while
    return P
  end if.

- Vasen-oikea-symmetrisesti saadaan myös
  - operaation first(T) vastine last(T), joka palauttaa koko puun *viimeisen* solmun
  - operaation next(I) vastine prev(I), joka palauttaa *edellisen* solmun.

- Myös kalvojen 6.1.1 funktion treeSearch(T, a) voi lukea puuiteraattorioperaationa:
  - Se palauttaa iteraattorina koko puusta T sen solmu, jonka avain on a.
  - Voi siis lähteä iteroimaan *keskeltä* järjestettyä aineistoa.

    Kalvojen 6 esimerkissä "luettele kaikki 30-50-vuotiaat yliopistolaiset".
  - Jos a puuttuu, niin iteraattorin kannattaa jäädä osoittamaan *viimeiseen solmuun josta ei päästy eteenpäin.*

6.1.3 Avaimen lisäys puuhun

- Lisätään pari

  avain a

  sisältö b

  hakupuuhun T jonka juuri on r.

- Jos puussa T on jo solmu x avaimella a, niin päivitetään sen sisällöksi b rivillä 15.

  Toinen mahdollisuus olisi tulkita se virhetilanteeksi.

- Juuri r muuttuu tyhjästä puusta NULL yksisolmuiseksi rivillä 2.

- Riveillä 4–13 etsitään se isäsolmu y jonka lapsessa x tai t on avaimen a oikea paikka.

  Eli kalvojen 6.1.1
  funktiosta treeSearch(T, a) versio, joka pysähtyy viimeiseen solmuun josta ei enää voi edetä.

procedure treeInsert(r : **juuriosoitinviite**,
a : **avainarvo**,
b : **sisältö**)

```
 1: if r = NULL then
 2:    r := new juurisolmu s jossa s.key := a,
       s.item := b, s.left := NULL, s.right := NULL ja
       s.parent := NULL
 3: else
 4:    x := r;
 5:    y := x.parent (eli aluksi NULL);
 6:    while x ≠ NULL and a ≠ x.key do
 7:       y := x;
 8:       if a < x.key then
 9:          x := x.left
10:       else
11:          x := x.right
12:       end if
13:    end while;
14:    if x ≠ NULL then
15:       x.item := b
16:    else
17:       t := new lehtisolmu s jossa s.key := a,
          s.item := b, s.left := NULL, s.right := NULL
          ja s.parent := y;
18:       if a < y.key then
19:          y.left := t
20:       else
21:          y.right := t
22:       end if
23:    end if
24: end if.
```


- Aikaa kuluu

  $$\mathcal{O}(\text{koko hakupuun } T \text{ korkeus})$$

  askelta.

- Tilaa kuluu

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

  muistipaikkaa.

- Pikku optimointi:

  Rivillä 18 tehtävä vertailu tekee uudelleen viimeisen rivillä 8 tehdyn vertailun.

6.1.4 Avaimen poisto puusta

- Poistetaan hakupuusta T siinä oleva solmu z.

- Solmu z on saatu (esimerkiksi) kalvojen 6.1.1 avainhaulla.

- Tämä poistomenetelmä käyttää kalvojen 6.1.2 osoittimia isäsolmuun.

- Jakaudutaan eri tapauksiin solmun z lapsimäärän mukaan:

  0: z on lehti, se voidaan poistaa

  1: z voidaan korvata ainoalla lapsellaan

  2: z voidaan vaihtaa kalvojen 6.1.2.1 mukaisen seuraavan solmunsa y kanssa.

  Joko solmu y on lehti

  tai sillä on vain 1 lapsi x.

  (Solmuksi y kävisi myös solmun z edeltäjä.)

procedure treeDelete(r : **juuriosoitinviite**, z : **solmuviite**)

```
 1: if z.left = NULL or z.right = NULL then
 2:     y := z
 3: else
 4:     y := next(z);
 5:     z.key := y.key;
 6:     z.item := y.item
 7: end if;
 8: if y.left ≠ NULL then
 9:     x := y.left
10: else
11:     x := y.right
12: end if;
13: if x ≠ NULL then
14:     x.parent := y.parent
15: end if;
16: if y.parent = NULL then
17:     r := x
18: else if y = y.parent.left then
19:     y.parent.left := x
20: else
21:     y.parent.right := x
22: end if;
23: delete y.
```


- Rivien merkitys:

  1–7: Aseta solmuosoitin y sellaiseen solmuun, joka voidaan poistaa.

  8–12: Aseta solmuosoitin x solmun y ainoaan lapsisolmuun, jos sellainen on olemassa.

  Muuten x on NULL.

  13–15: Ohjaa isäsolmuosoitin lapsisolmusta x ohi poistuvan solmun y sen isäsolmuun y.parent.

  16-22: Ohjaa isäsolmun y.parent vastaava osoitin ohi poistuvan solmun y sen tilalle nousevaan lapsisolmuun x.

  Jos y oli juuri, niin sillä ei ole isäsolmua y.parent. Silloin uusi juuri on lapsisolmu x.

- Aikaa ja tilaa kuluu jälleen

  $\mathcal{O}$(koko hakupuun T korkeus) ja $\mathcal{O}(1)$.

6.1.5 Lehtihakupuut

- Edellä selostettiin *sisä*hakupuita:
  - Pari $\langle$avain, sisältö$\rangle$ on *ainoassa avainta vastaavassa solmussa*.
  - Tämä solmu voi olla sisä- tai lehtisolmu.
  - Avain toimii tässä solmussa

    viittana solmun alipuihin

    nimenä solmun sisällölle.

- Toinen mahdollisuus ovat *lehti*hakupuut:

  lehtisolmussa on tällainen pari

  sisäsolmussa on avain *viittana sisällöttä*.

- Sisähakupuita *hieman* suurempia:

  korkeus $+1$: uusi lehtitaso

  solmujen määrä $+(n-1)$: n-lehtisessä binääripuussa on $(n-1)$ sisäsolmua.

- Muutokset kalvojen 6.1.1 puusta hakuun:
  - Algoritmi etenee lehtisolmuun saakka.

    Koska vasta siellä on avainta vastaava sisältö — ei vielä sisäsolmuissa joissa avain on viittana.
  - Haarautumisvertailuna on '$\leq$' eikä '$<$'.

    Koska viittaa vastaava lehtisolmu on itsekin (vasemmassa) alipuussa.

- Muutokset kalvojen 6.1.3 puuhun lisäykseen:
  - Tehdään uudelle parille uusi lehtisolmu u.
  - Haku päätyy lehtisolmuun v jossa onkin väärä avain.
  - Korvataan v uudella sisäsolmulla s jonka
    - lapsina ovat u ja v avainten mukaan järjestyksessä
    - viittana on pienempi näistä avaimista.

  Puu pysyy *aito*na!

- Muutokset kalvojen 6.1.4 puusta poistoon:
  - Poistettava solmu u on lehti.
    Vaihtoa seuraajan kanssa ei tarvita.
  - Aitouden nojalla sillä on velisolmu v.
    (Tai u oli koko puun ainoa solmu.)
  - Korvataan solmujen u ja v välinen sisäsolmu solmulla v.

  Siis *perutaan* solmun u lisäys.

  (Paitsi että v voi nyt olla myös sisä- eikä vain lehtisolmu.)

  \+ Lehtihakupuusta poisto on helpompaa kuin sisähakupuusta.

  − Hakupolun muissa sisäsolmuissa voi olla viittoina kopioita poistettavasta avaimesta.

- Tällä kurssilla keskitytään sisähakupuihin.

6.1.6 Suuremman haarautumisasteen hakupuut

- Edellä selostettiin *binääri*hakupuita:
  - Solmulla on *enintään 2 alipuuta*.
  - Alipuiden *välillä* on avainjärjestys.
  - Alipuiden *välissä* on viitta.
- Kalvoilla 5.1.3.1 mainittiin yleiset puut, joiden *haarautumisaste* (branching factor) eli suurin lasten lukumäärä voi olla > 2.
- Hakupuuksi käy myös sellainen puu, jonka haarautumisaste on *vakio* $b > 2$*:*
  - Solmulla on $0, 1, 2, \ldots, b$ alipuuta.
  - Alipuut ovat *peräkkäin:* 1., 2., 3., ...
  - Peräkkäisten alipuiden
    * välillä on avainjärjestys
    * välissä on viitta.

- Kalvojen 6.1.1 haku puusta suunnistaa solmusta alipuuhun käymällä läpi viitat 1. vs 2., 2. vs 3., 3. vs 4., ... kalvojen 2.4.1 peräkkäishaulla.
- Solmussa tehdään siis
  $$\mathcal{O}(b)$$
  askelta, mutta b on ohjelmoijan käännösaikana kiinnittämä vakio.
- Yksi solmu voi osittaa sisältämänsä aineiston b osaväliin.

  Silloin hyvän puun korkeus on
  $$\mathcal{O}(\log_b(n)).$$
- Siis puista voi tulla *hyvin matalia*.

  Eli nopeita käsitellä.
- Tällä kurssilla keskitytään binäärisiin hakupuihin.

- Hyödyllinen varsinkin ylläpidettäessä hakupuuta *levyllä:*

  Jokainen muistioperaatio (luku/kirjoitus)
  - on hidas
  - tekee kokonaisen lohkon kerrallaan.
- Lohkon koon määrää laite.

  Valitaan sellainen b jolla yhdestä lohkosta saadaan eniten hyötyä.
- Paljon käytettyjä juuri *tietokannanhallintajärjestelmissä*.

  Tietorakenteiden näkökulmasta ne
  - ylläpitävät erilaisia hakurakenteita joista hakupuut ovat yksi tärkeä perhe
  - optimoiden allaolevan
    * hitaan
    * kiinteissä lohkoissa olevan

    muistin käsittelyä.

6.2 Puun tasapainoisuus

- Kalvojen 6 algoritmeissa kuljettiin solmusta toiseen viittojen ohjaamana.

- Pisimmänkin polun pitäisi olla lyhyt, jotta suoritusaika olisi pahimmillaankin pieni.

- Vapaasti kasvaessaan n solmun binäärinen hakupuu T voi muistuttaa muodoltaan

  listaa jolloin pisin polku on

  $$\mathcal{O}(n)$$

  — *pahin* tapaus

  täydellistä binääripuuta jolloin pisin polku on

  $$\mathcal{O}(\log_2(n))$$

  — *paras* tapaus.

- On kehitetty erilaisia ehtoja joilla voi taata puun T pysyvän *tasapainoisena* (balanced) kun sitä muutetaan.

6.2.1 Tasapainotus korkeusrajoitteilla

- Yksi tapa taata tasapainoisuus on *rajoittaa alipuiden välistä korkeuseroa*.

  Puun korkeushan on määritelty pisimmän lehteen vievän polun pituutena.

- Esimerkiksi **AVL-puu** on

  – binäärinen sisähakupuu

  – jonka jokaisessa solmussa s pidetään yllä lisäkenttää $s.\text{height}$ = solmusta s alkavan alipuun korkeus

  (ajatellaan että $\text{NULL}.\text{height} = 0$)

  – ja vaaditaan aina

  $$|s.\text{left}.\text{height} - s.\text{right}.\text{height}| \leq 1 \quad (15)$$

  eli että *vasemman ja oikean alipuun korkeuksien ero saa olla enintään* 1 *taso*.

- Tämä tasapainoehto (15) on palautettava voimaan

  – puun muutosten eli solmun

  * lisäyksen

  * poiston

  jälkeen

  – paluumatkalla rekursiossa lisäyskohdasta takaisin juureen.

- Tasapainotus voidaan tehdä sopivin *kierroin* (rotation):

  Paikallisin muunnoksin nykyisen solmun r lähistöllä

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

  askeleessa osoittimia muokaten.

- Tarkastellaan lisäystä.

Yksinkertainen oikealle (single) kierto:

- Jos ehto (15) on rikki nykyisessä solmussa r, niin katsotaan sen lapsista korkeampaan lapseen $c = r.\text{left}$.
- Jos lapsen c omista lapsista korkeampi on saman suuntainen $c.\text{left}$, niin *vaihdetaan* c *isäksi ja* r *lapseksi*.
- Avainjärjestyksen täytyy säilyä:

  Solmujen c ja r välisen alipuun $c.\text{right}$ on siirryttävä alipuuksi $r.\text{left}$.

Kaksinkertainen oikealle (double) kierto:

- Jos eri suuntainen $g = c.\text{right}$, niin tehdään 2 yksinkertaista kiertoa:
- Ensin solmussa c vasemmalle.

  Silloin c ja g vaihtavat paikkaa.
- Sitten solmussa r oikealle.

  Silloin g ja r vaihtavat paikkaa.

Vasen/oikea-symmetrisesti muut 2 vaihtoehtoa.

6.2.1.1 Kierto

- Yksinkertainen kierto vasemmalle puun T solmussa x
  - olettaa että solmulla x on myös oikea lapsi $y = x.\text{right}$ — ei kutsuta ellei ole
  - nostaa lapsen y solmun x uudeksi isäksi
  - josta tulee solmun y uusi vasen lapsi
  - joka huolehtii solmun y vanhasta vasemmasta alipuusta uutena oikeana alipuunaan.
- Pidetään yllä kalvojen 6.1.2 isäosoittimia.
- *Ei* pidetä yllä kalvojen 6.2.1 korkeuskenttiä.

  Ne liittyvät vain AVL-puihin, kiertojen käsite puihin yleisemminkin.
- Yksinkertainen kierto vasemmalle saadaan samasta ohjelmakoodista vasen/oikea-symmetrisesti.

procedure LeftRotate(T: **juurisolmuviite**, x : **solmuosoitin**)

```
 1: y := x.right;
 2: x.right := y.left;
 3: y.left.parent := x;
 4: y.parent := x.parent;
 5: if x.parent = NULL then
 6:    T := y
 7: else if x = x.parent.left then
 8:    x.parent.left := y
 9: else
10:    x.parent.right := y
11: end if;
12: y.left := x;
13: x.parent := y.
```

Rivien merkitys:

2–3: Solmun y vanha vasen alipuu siirtyy solmun x uudeksi oikeaksi alipuuksi.

4–11: Solmu y siirtyy (ali)puun juureksi solmun x tilalle.

12–13: Solmu x siirtyy solmun y uudeksi vasemmaksi lapseksi.