8. Yhteenvetoa

Tietorakenteet ja abstraktit tietotyypit:

abstrakti tietotyyppi: mitä datalle halutaan tehdä
tietorakenne: miten se tehdään (talletusrakenne, operaatioden toteutus)

Tietorakenteet ja algoritmit:

- tietorakenteiden toteuttamisessa voidaan tarvita ei-triviaaleja algoritmeja (esim. AVL-puiden tasapainotus)
- algoritmien tehokkaassa toteuttamisessa voidaan tarvita ei-triviaaleja tietorakenteita (esim. Kruskalin algoritmi: keko ja Union-Find).

Millaisia asioita olemme oppineet

Toisin sanoen mitä kurssista olisi hyvä muistaa vielä ensi vuonna.

Perustietorakenteita ja -algoritmeja, jotka on hyvä osata jopa koodata melko sujuvasti:

- pino, jono, linkitetty lista, puu
- syvyyssuuntainen ja leveyssuuntainen läpikäynti

Yleisesti tarvittavia tietorakenteita ja algoritmeja, joita ei yleensä tarvitse (tai kannata) itse koodata, mutta keskeiset periaatteet pitää tuntea:

- hakemistorakenteet: puu vs. hajautus; keskusmuisti vs. levy
- järjestäminen: $O(n)$ vs. $O(n \log n)$ vs. $O(n^2)$; pikajärjestämisen vaikeat tapaukset.

Mallinnustekniikoita: etenkin puiden ja verkkojen käyttäminen ongelmanratkaisun mallina

- tekoäly: pelipuut, abstraktin puun tai verkon läpikäynti, graafiset mallit (verkkopohjaisia todennäköisyysmalleja)
- tietoliikenne: esim. reitittäminen Dijkstran algoritmilla
- ohjelmointi ja ohjelmointikielet: jäsennyspuu, verkkopohjaiset kaavioesitykset
- tietojenkäsittelyteoria: puut ja verkot abstraktien laskennan mallien esittämisessä.

Algoritmien suunnittelutekniikoita:

invariantit: perustekniikka, josta on iloa myös perusohjelmoinnissa turhien virheiden välttämiseksi

hajoita ja hallitse: keskeinen tekniikka edistyneemmässä algoritmisuunnittelussa; analyysi perustuu rekursioyhtälöihin; esim. lomitusjärjestäminen

ahne algoritmi: idea on usein luonteva, mutta oikeellisuus vaikea perustella; esim. Kruskal

taulukointi (dynaaminen ohjelmointi): monipuolinen tekniikka, itse asiassa varsin intuitiivinen kun siihen tottuu; esim. Floyd-Warshall.

Etenkin hajoita ja hallitse, ahneus ja taulukointi voivat olla vaikeita soveltaa. Niitä voi harjoitella lisää kurssilla *Algoritmien suunnittelu ja analyysi*. Tällä kurssilla päätavoite on ymmärtää esitettyjen esimerkkialgoritmien ideat ja osata soveltaa niitä hieman muuntaen.

Algoritmit, tietorakenteet ja muu maailma

Käytännössä algoritmit

- toteutetaan tietokonelaitteistolla ja
- liittyvät osaksi sovellusohjelmaa.

Tällä kurssilla on lähinnä tarkasteltu asymptoottista pahimman tapauksen aikavaativuutta ("O-notaatio"), joka kertoo yksinkertaisessa muodossa algoritmin skaalautuvuuden. Sovelluksessa tarvitaan muutakin:

- Kuinka isoja syötteet todella ovat? Helppoja vai vaikeita?
- Mikä on riittävä suorituskyky? Mikä oikeasti on järjestelmän pullonkaula?
- Jos algoritmia pitää tosissaan ruveta viilaamaan, miten se suhtautuu käytettävissä olevaan suoritusympäristöön (rinnakkaistuminen, välimuistin käyttö jne.)?
- Käytännön tekijät: ylläpidettävyys, virheestä toipuminen jne.

Kertaustehtäviä

Käydään kertauksena läpi kevään 2005 toisen kurssikokeen tehtävät.

Tehtävä 1: Erääseen sovellukseen tarvitaan seuraavanlainen tietorakenne R:

- Sen mahdolliset hakuavaimet k ovat 64-bittisiä etumerkittömiä kokonaislukuja (eli väliltä $0 \ldots 2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615$) ja niiden jakauma on tasainen.
- Hakuoperaatio lookup(R, k) vastaa, löytyykö avain k tällä hetkellä rakenteesta R vaiko ei.
- Lisäysoperaatio insert(R, k) lisää avaimen k rakenteeseen R, ellei se jo ole siinä.

Koska sovelluksessa täytyy säästää muistia, niin rakenteessa R saa olla yhtä aikaa korkeintaan $2^{20} = 1\,048\,576$ avainta. Jos rakenne R on jo täynnä, niin uusi avain k korvaa sen vanhan avaimen k', jonka edellisestä käsittelystä (eli lisäyksestä insert(R, k') tai hausta lookup(R, k')) on mahdollisimman kauan.

Sovelluksessa on tärkeintä, että onnistuvat haut (eli ne, joilla lookup(R, k) vastaa `true`) olisivat keskimäärin hyvin nopeita. Muut operaatiot eivät ole niin kriittisiä.

(a) Ehdota tälle rakenteelle R sopiva toteutus. Perustele valintasi.

(b) Anna kohdan (a) toteutuksessasi pseudokoodi rakenteen R hakuoperaatiolle lookup(R, k).

(c) Anna kohdan (a) toteutuksessasi pseudokoodi rakenteen R lisäysoperaatiolle insert(R, k).

Tehtävän 1 ratkaisuidea: Tärkeimmäksi kriteeriksi on annettu onnistuneen haun keskimääräinen aikavaativuus. Ensimmäisenä mieleen tulee hajautus. Koska tehtävässä lisäksi avaimet ovat kokonaislukuja ja jakautuneet tasaisesti, esim. jakolaskumenetelmä toimisi varsin hyvin.

Valitaan siis perusratkaisuksi hajautus jakolaskumenetelmällä. Koska joudutaan suorittamaan myös poistoja, yhteentörmäykset on kätevämpää hoitaa ketjuttamalla (eikä avoimella hajautuksella).

Valitaan talletusalueen kooksi p alkuluku läheltä lukua $(2^{18} + 2^{19})/2 = (3/4) \cdot 2^{19} = (3/8) \cdot 2^{20}$. Tällöin se on kaukana kakkosen potensseista, ja täyttöasteeksi tulee korkeintaan $8/3$ eli varsin kohtuullinen. (Muutkin valinnat ovat mahdollisia.)

Mietitään nyt poistojen toteuttamista.

Koska poistettavaksi valitaan kauimmin joutilaana ollut avain, ensimmäisenä tulee mieleen tallentaa avaimet jonoon. Ongelmana on, että lookup-operaatiot ”nuorentavat” avaimia, joten jonon ”first in, first out” -järjestystä pitää päästä muokkaamaan.

Tähän ongelmaan sopii suoraan ratkaisuksi prioriteettijono (maksimiversio) varustettuna DecreaseKey-operaatiolla. Ikävä kyllä DecreaseKey vaatii logaritmisen ajan, ja tässä sovelluksessa se jouduttaisiin aina suorittamaan lookup-operaation yhteydessä. Menettäisimme siis lookup-operaation (keskimääräisen) vakioaikaisuuden, mikä oli hajautuksen keskeinen hyöty.

Tarkemmin ajatellen emme kuitenkaan tarvitse tässä yleistä DecreaseKey-operaatiota, sillä lookup vie löydetyn avaimen aina poistojärjestyksessä viimeiseksi.

Siis talletetaan avaimet listaan siten, että

- lisäykset tehdään listan loppuun
- poistot tehdään listan alusta
- lookup-operaation yhteydessä alkio siirretään listan viimeiseksi.

Jokainen lisättävä avain tulee varsinaisessa hajautustaulussa talletetuksi ylivuotolistaan, ja lisäksi tarvitaan lista avainten poistojärjestyksestä. Koska listoista tehdään myös poistoja, toteutetaan ne kahteen suuntaan linkitettyinä.

Eräs mahdollinen ratkaisu olisi luoda jokaiselle avaimelle kaksi listasolmua, toinen hajautustaulun ylivuotolistaan ja toinen poistolistaan. Nämä pitäisi kuitenkin vielä linkittää toinen toisiinsa. Tehokkaampaa lienee tallentaa kukin avain vain yhteen solmuun, ja liittää tähän kahdet linkitykset. Solmuun r tulee siis seuraavat kentät:

$key[r]$: avain
$next[r]$: seuraaja ylivuotolistassa
$prev[r]$: edeltäjä ylivuotolistassa
$newer[r]$: seuraaja poistolistassa
$older[r]$: edeltäjä poistolistassa.

Olkoon hajautustaulun talletusalue $A[0 \mathinner{..} p-1]$. Toteutetaan poistolista L tunnussolmullisena rengaslistana. Siis poistolistan alku- ja loppusolmut ovat $newer[L]$ ja ja $older[L]$; kenttiä $next[L]$ ja $prev[L]$ ei käytetä. Laskuri n pitää kirjaa avainten lukumäärästä.

Kirjoitetaan ensin apuproseduuri search, joka etsii avainta k ja jos löytää, niin siirtää vastaavan tietueen poistolistan loppuun:

```
search(R, k)
    r ← A[k mod p]
    while r ≠ Nil and key[r] ≠ k
          do r ← next[r]
    if r ≠ Nil
       then newer[older[r]] ← newer[r]
            older[newer[r]] ← older[r]
            older[r] ← [older[L]]
            newer[r] ← L
            newer[older[r]] ← r
            older[newer[r]] ← r
    return r
```

Operaation lookup saadaan suoraan apuproseduurista search:

```
lookup(R, k)
    return search[R, k] ≠ Nil
```

Operaatio insert hoituu sekin pelkällä search-kutsulla, mikäli avain löytyy. Muuten se täytyy lisätä, mikä jakautuu kahteen päähaaraan.

Jos avainten lukumäärä on suurin sallittu, niin poistolistasta pitäisi poistaa ensimmäinen solmu ja lisätä uusi avain viimeiseksi solmuksi. Teemme tämän "kiertämällä" kehälistaa niin, että ensimmäisestä solmusta tulee uusi tunnussolmu ja uusi avain talletetaan vanhaan tunnussolmuun. Tällöin poistuva avain pitää poistaa myös itse hajautustaulusta.

Jos lukumäärä ei ole suurin sallittu, luodaan tavalliseen tapaan uusi solmu.

Kummassakin tapauksessa uusi avain lisätään vastaavan ylivuotoketjun alkuun.

```
insert(R, k)
    if search[R, k] = Nil
       then if n = 2^20
               then r ← L
                    key[r] ← k
                    L ← newer[L]
                    if prev[L] = Nil
                       then A[key[L] mod p] ← next[L]
                       else  next[prev[L]] ← next[L]
                    if next[L] ≠ Nil
                       then prev[next[L]] ← prev[L]
               else r ← new listasolmu
                    n ← n + 1
                    key[r] ← k
                    newer[r] ← L
                    older[r] ← newer[L]
                    older[newer[r]] ← newer[older[r]] ← r

            next[r] ← A[k mod p]
            prev[r] ← Nil
            A[k mod p] ← r
            if next[r] ≠ Nil
               then prev[next[r]] ← r
```

Tehtävä 2: Mikä keko-, pika- ja lomitusjärjestämisalgoritmeista on se, joka tarvitsee vähiten muistia?

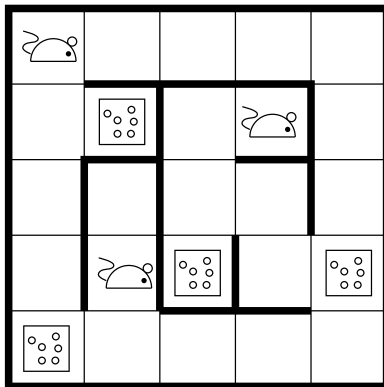
Perustele vastauksesi selittämällä, mihin ne kaksi muuta tarvitsevat enemmän muistia kuin se.

Tehtävän 2 ratkaisu: Kekojärjestäminen toimii vakiotyötilassa ja vie näin ollen algoritmeista vähiten muistia.

Pikajärjestäminen tarvitsee rekursiopinoa varten keskimäärin $\Theta(\log n)$ ja pahimmassa tapauksessa $\Theta(n)$ työmuistia.

Lomitusjärjestämisessä lomittamiseen tarvitaan alkuperäisen taulukon kokoinen aputaulukko, joten työtilaa kuluu $\Theta(n)$.

Tehtävä 3: Labyrinttiin on asetettu sinne tänne hiiriä ja juustoja oheisen kuvan tapaan. (Voit olettaa, että aluksi mitkään kaksi hiirtä eivät ole samassa labyrintin ruudussa.) Jokainen hiiri voi siirtyä yhdellä askeleella viereiseen ruutuun, ei kuitenkaan viistosti eikä seinien läpi. Tehtävänäsi on selvittää, millä hiiristä on lyhyin matka sitä lähimmän juuston luokse.



Kehitä tähän ongelmaan sellainen algoritmi, jonka suoritusaika on lineaarinen labyrintin pinta-alan suhteen. Anna algoritmistasi

(a) lyhyt sanallinen selitys sen toimintaperiaatteesta
(b) kohdan (a) selitystäsi vastaava pseudokoodi
(c) perustelu kohdan (b) koodisi suoritusajalle.

Tehtävän 3 ratkaisuidea: Ensimmäisenä mieleen tuleva ratkaisu on laskea Floydin-Warshallin algoritmilla täydellinen etäisyystaulukko ja valita kaikista hiiri-juustopareista etäisyydeltään pienin. Tämä on kuitenkin hyvin kaukana lineaarisesta ajasta.

Seuraava idea on laskea erikseen kullekin juustolle etäisyys lähimpään hiireen. Koska verkossa ei ole painoja, ei tarvita Dijkstran algoritmia, vaan leveyssuuntainen läpikäynti riittää. Tämä vaatii ajan $O(|V| + |E|)$ per juusto, missä (V, E) on labyrinttia esittävä verkko.

Koska $|V|$ on sama kuin labyrintin pinta-ala (sopivalla yksiköllä mitattuna) ja $|E| \leq 4|E|$, aika on pinta-alan suhteen lineaarinen yhdelle juustolle. Jos juustoja voi olla mielivaltaisen monta, tämä ei vielä kelpaa.

Edellinen ratkaisuyritys tekee paljon päällekkäistä laskentaa, koska eri juustoista lähtevät polut voivat yhtyä, mutta tätä ei mitenkään oteta huomioon.

Tästä saadaan taulukointi-idea: muodostetaan labyrintin kokoinen taulukko A, missä $A[i,j]$ on ruudun (i,j) etäisyys lähimmästä juustosta.

Aluksi asetetaan $A[i,j] \leftarrow 0$, jos ruudussa (i,j) on juusto, ja $A[i,j] \leftarrow \infty$ muuten.

Vaiheessa k asetetaan $A[i,j] \leftarrow k$ kaikilla ruuduilla (i,j), joilla $A[i,j]$ on toistaiseksi ∞ mutta joilla on naapuriruutu (r,s), jolla $A[r,s] = k-1$.

Tämäkään ei vielä toimi lineaarisessa ajassa, jos joka vaiheessa joudutaan käymään koko taulukko läpi ja etsimään muuttuvia kohtia.

Nyt kuitenkin havaitaan, että vaiheessa k tarvitsee tarkastella vain vaiheessa $k-1$ muuttuneiden ruutujen naapureita. Laittamalla nämä jonoon päästään tilanteeseen, jossa jokainen taulukon ruutu käsitellään vain kerran.

Nyt kun on päädytty jonoon, havaitaan itse asiassa, että ongelmalla on seuraava yksinkertainen ratkaisu:

1. Yhdistä verkossa (V, E) kaikki juuston sisältävät solmut yhdeksi maalisolmuksi.
2. Tee leveyssuuntainen läpikäynti maalisolmusta lähtien. Lopeta, kun ensimmäinen hiiri löytyy.

(Tämän olisi tietysti voinut keksiä suoraankin.)

Kuten edellä todettiin, leveyssuuntainen läpikäynti sujuu labyrintin pinta-alan suhteen lineaarisessa ajassa. Ratkaisua voidaan vielä yksinkertaistaa jättämällä etäisyyksien laskenta pois, jos riittää löytää yksi lähin hiiri.

Seuraavassa $F[i,j] = 1$, jos ruutuun (i,j) on jo löydetty polku jostain juustosta, ja $F[i,j] = 0$ muuten.

```
Hiiret-ja-juustot(Labyrintti)
    Q ← tyhjä jono
    for kaikilla labyrintin ruuduilla (i,j)
        do if ruudussa (i,j) on juusto
            then F[i,j] ← 1
                 Enqueue(Q,(i,j))
            else F[i,j] ← 0
    while not Empty(Q)
        do (i,j) ← Dequeue(Q)
           for kaikille ruudun (i,j) naapureille (r,s)
               do if ruudussa (r,s) on hiiri
                     then return (r,s)
                  elseif F[r,s] = 0
                     then F[r,s] ← 1
                          Enqueue(Q,(r,s))
    return "ei ratkaisua"
```

Algoritmin rivi

For kaikille ruudun (i, j) naapureille (r, s)

tarkoittaa, että toistetaan seuraaville korkeintaan neljälle solmulle:

1. $(r, s) = (i + 1, j)$, *paitsi* jos $i + 1$ on suurempi kuin rivien lukumäärä tai välillä $(i, j) \leftrightarrow (i + 1, j)$ on seinä
2. $(r, s) = (i, j + 1)$, *paitsi* jos $j + 1$ on suurempi kuin sarakkeiden lukumäärä tai välillä $(i, j) \leftrightarrow (i, j + 1)$ on seinä
3. $(r, s) = (i - 1, j)$, *paitsi* jos $i - 1 \leq 0$ tai välillä $(i, j) \leftrightarrow (i - 1, j)$ on seinä
4. $(r, s) = (i, j - 1)$, *paitsi* jos $j - 1 \leq 0$ tai välillä $(i, j) \leftrightarrow (i, j - 1)$ on seinä.

Alustukset menevät selvästi labyrintin koon suhteen lineaarisessa ajassa.

Tämän jälkeen tehdään vakiomäärä työtä jokaista jonosta poimittavaa ruutua (i, j) kohti. Koska jonoon viedään vain ruutuja (r, s), joilla $F[r, s] = 0$, ja samalla asetetaan $F[r, s] \leftarrow 1$, mikään ruutu ei käy jonossa kuin korkeintaan kerran.

Siis kokonaisaika on lineaarinen labyrintin ruutujen lukumäärän suhteen.