Itä-Suomen yliopisto

Tietojenkäsittelytiede

Karelia, Savonia

Tietorakenteet ja algoritmit I

5 op, 3621414
syksy 2023

Simo.Juvaste@uef.fi
22.1.2024 13:35

elearn.uef.fi, kurssi: TRA I, avain: tira

Sisällys

1 Algoritmiikasta

⇒ Ohjelmointi ei ole vain ohjelmakoodin koneeseen syöttämistä.

- Ohjelmointiprosessi sisältää myös ohjelman suunnittelun usealla eri tasolla, sekä valmiin ohjelman arvioinnin.
- TRA -kurssit kertovat erään tärkeän osan/näkökulman tietojärjestelmien suunnitteluun ja ohjelmointiin.

1.1 Ongelmasta ohjelmaksi

⇒ Tehtävä on määriteltävä täsmällisesti, jotta tiedetään, mitä ratkaista.

- Esimerkiksi:
  - ”Laadi ohjelma, joka laskee lukujen summan.”
  - ”Laadi ohjelma, joka laskee ja tulostaa näppäimistöltä syötettävien kokonaislukujen summan. Syötteet loppuvat, kun näppäillään luku nolla.”
- Jälkimmäistä työn määrittelyä noudattaen on paljon todennäköisempää, että eri ohjelmoijat tekevät suunnilleen samanlaisen ohjelman.
- Tehtävän täsmällinen määrittely on paljolti muiden kurssien asia.

  - Tällä kurssilla syöte saadaan yleensä aliohjelman/metodin parametrina, eikä käyttäjän kanssa keskustelua tarvita.

⇒ Kun ongelma on määritelty, voidaan ryhtyä suunnittelemaan ratkaisuperiaatetta, eli algoritmia.

Ratkaisuperiaate eli algoritmi (*algorithm*):

⇒ Täsmällisesti kuvattu tapa ratkaista ongelma.

- äärellinen toimintojono
- suoritettavissa äärellisessä ajassa
- suoritettavissa äärellisellä työmäärällä
- deterministinen (yleensä)
- päättyvä

⇒ algoritmi ≠ ongelma

Algoritmin esittäminen

- Joko suoraan ohjelmointikielellä tai ns. pseudokielellä.
- Kuva havainnollistaa algoritmin toimintaa.

⇒ Pseudokielellä esitetty algoritmi voidaan toteuttaa (melko) suoraviivaisesti millä tahansa ohjelmointikielellä.

- Tällä kurssilla algoritmit esitetään pääosin Java-kielellä, kuitenkin käyttäen algoritminotaatiota (pseudokieleltä) apuna selkeyttämään algoritmeja.
- Algoritmin (tai sen osan) kuvaus pitäisi pystyä kullakin tasolla esittämään yhdellä pienellä sivulla (kalvolla), mieluummin n. 5-15 rivillä.
  - Yksinkertainen algoritmi mahtuu sivulle Javalla kirjoitettuna.
  - Monimutkainen algoritmi joudutaan esittämään yksinkertaistetulla notaatiolla (ja tarkentamaan kukin kohta erikseen).
- Kaikista algoritmeista ja tietorakenteista piirretään kuva...

Onko algoritmi valmis?

- Algoritmin oikeellisuus on varmistettava.
  - Hyvä suunnittelu teki oikean algoritmin laadinnasta helpompaa.
  - Invariantit (kohta 1.5, s. 20) on hyvä apuväline oikeellisuuden varmistamiseksi.
  - Kts. myös ohjelmointitekniikan ja testauksen kurssit.
- Algoritmin tehokkuus on varmistettava.
  - TRA I, TRA II, ASA
  - Jos algoritmi ei ole riittävän tehokas, niin:
    - Voidaanko tehostaa?
    - Kannattaako tehostaa?
  - Peräkkäisalgoritmilla tehokkuus ja nopeus ovat yhtä.
    - Voimme saada lisää nopeutta rinnakkaistamalla (mutta emme tehokkuutta).
    - Kts. kurssi Rinnakkaislaskenta.

Algoritmin toteutus

- Ohjelmoinnin kurssit (& TRA I).

- Asteittainen tarkentaminen (*gradual refinement*) algoritmin pseudokielestä
  - Pseudokieltä muokataan oikeaksi ohjelmointikieleksi.
  - Abstraktit tietotyypit vaihdetaan oikeiksi tietorakenteiksi.
- Tehokkuusnäkökohtiin täytyy paneutua vielä toteutusvaiheessakin.
  - Huolimaton toteutus voi romahduttaa tehokkuuden.
    - Hyvin suunniteltu ja esitetty algoritmi ohjaa toteutuksen tehokkaaksi.
  - Huolimaton valmiskirjastojen käyttö voi romahduttaa tehokkuuden, tai jopa oikeellisuuden.
    - On syytä tarkistaa kaikkien valmiskirjastojen operaatioiden tarkka toiminta jo ennen toteutukseen ryhtymistä.
    - Puutteellisesti dokumentoitua (kuten jossain määrin myös Java API) valmiskirjastoa käyttäen tehokkuuden varmistaminen voi olla vaikeaa muuten kuin testaamalla! (kts. TRA II).

Toteutetun algoritmin testaus

- Virheitä on voinut sattua niin algoritmin suunnittelussa kuin ohjelman toteutuksessakin.
- Täydellinen testaus ei varsinaisesti ole tämän kurssin asia.
- Hyvin suunniteltu ja toteutettu algoritmi on helpompi testata.
- Ohjelman oikeellisuus seuraa algoritmin oikeellisuudesta ja huolellisesta asteittain tarkentavasta algoritmin toteutuksesta.
- Aikavaativuuden varmistus (TRA II).

Valmiin ohjelman kunnossapito

- Ei ole tämän kurssin asia.

Onko tehokkuus tärkeää?

- Mitä tarkoittaa tehokkuus? (suunnittelun/toteutuksen, vai suorituksen?)
- Suorituksen tehokkuus ei aina ole oleellista (pienet syötteet, PCt).
- Pahan tehottomuuden välttäminen on tärkeää.
- Joskus (usein?) paras mahdollinen ohjelman suorituskyky (siis vähemmän suoritusaikaa) on hyödyllistä:
  - Palvelimet
    - Yksi palvelin voi palvella sitä useampaa asiakasta, mitä tehokkaampi palvelimessa suoritettava ohjelmisto on.
    - Sähköä säästyy $\rightarrow CO_2$ päästöt vähenevät!
    - Ostettua suoritinaikaa säästyy / pärjätään halvemmalla laitteistolla.
  - Akkukäyttöiset laitteet
    - Jokainen operaation suoritus vie energiaa, lepotila ei juuri vie energiaa.
    - Tehokkaampi (vähemmän suoritusaikaa) algoritmi vie vähemmän akun varausta $\rightarrow$ akku kestää pidempään.
  - Usein suoritettavat operaatiot.
    - Pienikin säästö kertautuu jos suorituskertoja ja/tai laitteita on paljon.
  - Suurella syötteellä tehokkuus on aina tärkeää (jotta laskenta päättyy järkevässä ajassa).

TRA kurssien motivaatio & sisältö

- Algoritmien (ja ohjelmien) sekä tiedon esittämisen abstraktiotason nostaminen.

- Ohjelmoijien ajattelun abstraktiotason nostaminen.
- Kyky suunnitella ja toteuttaa vaativampia algoritmeja (siis ratkaista vaativampia ongelmia).
- Modulaarisuuden lisääminen.
- Oikean tietorakenteen valitseminen kaikilla tasoilla.
- Valitun tietorakenteen oikea ja tehokas käyttäminen.
- Valmiiden operaatioiden ja algoritmien hyödyntäminen.
- Tehokkuuteen vaikuttavien asioiden ymmärtäminen.
- Suorituskyvyn ennakointi.

Miksi abstraktiota tarvitaan?

⇒ Ohjelmointi on vaikeaa!

- Abstraktio auttaa selkeyttämään ohjelmointityötä.
  - ”Ylös näpräämisestä.”
  - Syötteet, toiminnot, tulosteet näkyvät selkeämpinä, ymmärrettävämpinä kokonaisuuksina.
  - Ohjelmasta tulee selkeämpi ja muokattavampi.
  - Mitä riippumattomammista toiminnallisista osista (moduuleista) ohjelma koostuu, sitä helpompi kukin niistä on suunnitella, toteuttaa ja testata.

Jokainen ohjelmoija kuvittelee (joskus) tuottavansa selkeitä, helposti ymmärrettäviä ohjelmia.

- Silti ohjelmoijan on usein vaikea ymmärtää toisten ohjelmoijien tuottamia ohjelmia.
- Tai omiaan muutaman vuoden kuluttua...

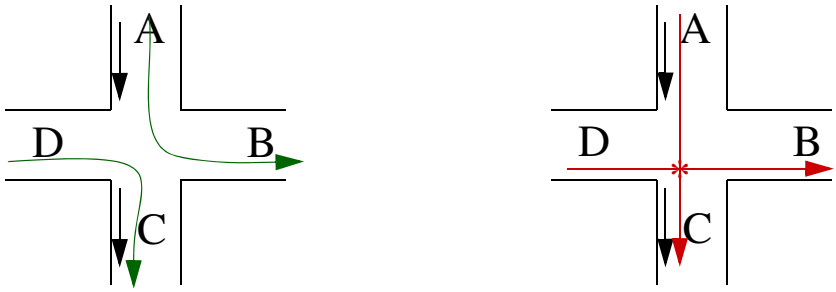
Abstraktio mahdollistaa ongelmien ratkaisemisen asteittain tarkentamalla!

- Haastavan ongelman ratkaiseminen kerralla voi olla liian vaikeaa.
- Tehdään se mieluummin useassa yksinkertaisemmassa vaiheessa!

Esimerkki: liikennevalojen vaiheistuksen suunnittelu

- Syöte: risteyksen mahdolliset ajoreitit, keskenään törmäävät reitit.
- Tulos: reittien optimaalinen ryhmittely siten, ettei yhteentörmäyksiä tule.

Havainnollistetaan ongelmaa kuvan 1.1 esittämällä risteyksellä, jossa kadut A ja C ovat yksisuuntaisia, kadut B ja D puolestaan kaksisuuntaisia. Mahdollisia ajoreittejä on kaikkiaan seitsemän erilaista. Niistä vaikkapa reitit AB ja DC voidaan ajaa samanaikaisesti (oletetaan nyt siis kaistoja olevan riittävästi), mutta reittien AC ja DB yhtäaikainen käyttäminen aiheuttaa yhteentörmäyksen vaaran.

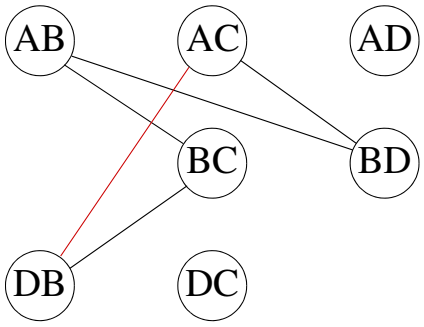


Kuva 1.1: Katujen risteys

	AB	AC	AD	BC	BD	DB	DC
AB				1	1		
AC					1	1	
AD							
BC	1					1	
BD	1	1					
DB		1		1			
DC							

Kuva 1.3: Verkon matriisiesitys

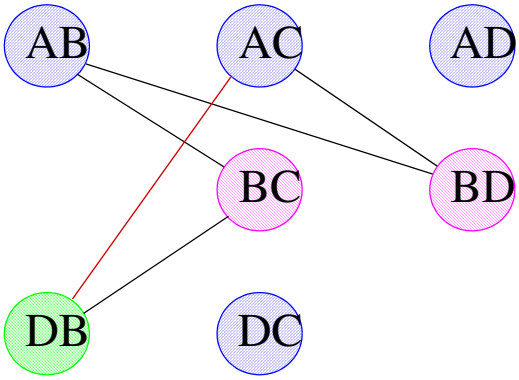
Kuvataan ongelma verkkona eli graafina, joka koostuu joukosta solmuja ja joukosta näitä solmuja yhdistäviä kaaria. Verkkojen käsitteistö esitellään tarkemmin luvussa 5 ja TRA II -kurssilla.



Kuva 1.2: Yhteentörmäävien reittien verkko

Esittäköötsolmut ajoreittejä ja olkoon verkossa kaari kahden solmun välillä vain siinä tapauksessa, ettei näitä kahta reittiä voida ajaa samanaikaisesti. Kuvan 1.1 risteystä vastaava verkko nähdään kuvassa 1.2. Taulukko 1.3 esittää saman verkon toisessa muodossa, taulukkona, jossa ykköset ilmaisevat kaaren olemassaolon ja tyhjät alkiot kaaren puuttumisen. Näistä esitysmuodoista kuva 1.1 on ilman muuta ihmiselle ymmärrettävin ja taulukko 1.3 puolestaan tietokoneelle ymmärrettävin. Kuvan 1.2 esitys ei liene paras mahdollinen kummallekaan, mutta ongelmaa verkkona tarkasteltaessa se antaa tyhjentävän kuvan tilanteesta.

Väritetään nyt verkon solmut niin, ettei minkään kaaren molemmissa päissä käytetä samaa väriä.



Kuva 1.4: Eräs mahdollinen ryhmittely

- Kuvan 1.4 kolmevärinen ratkaisu on jopa optimaalinen (minimaalinen) värien määrän suhteen.
- Tätä verkkoa ei voi värittää kahdella värillä siten, että naapurit olisivat aina eri väriset:
  - Kehässä *AB*, *BC*, *DB*, *AC*, *BD*, *AB* on pariton määrä solmuja.
  - Väritettäessä aina joka toinen solmu toisella värillä, tulee lopulta tilanne jossa värittämättömän solmun naapurit ovat eri väriset.
- Tarvitaan siis >2 väriä ja kun 3-värinen on löytynyt $\Rightarrow$ väritys on optimaalinen!

- Paluu liikennevaloihin?
  - Kukin väri (solmuryhmä) muodostaa valojen vaiheen.
  - Samalla värillä väritetyt ajosuunnat eivät törmää (koska ne eivät ole naapureita).

Miten löytää ratkaisu algoritmisesti?

- Väritysongelma on ”vaikea” (tarkemmin: vaatii paljon laskentaa löytää optimaalinen väritys).
  - Erilaisia mahdollisia kelvollisia värityksiä (naapurit eri värisiä) on eksponentiaalinen määrä.

  - Optimaalisen ratkaisun (mahdollisimman vähän värejä) varmistamiseen yleisessä tapauksessa ei nykyään tunneta muuta mahdollisuutta kuin käydä läpi (lähes) kaikki väritykset.
    * Kyse on niinsanotusta NP-vaikeasta (*NP-hard*) ongelmasta.
    * Yleisesti otaksutaan, ettei ongelmaan koskaan löydy tehokasta (ei-eksponentiaalista) ratkaisua!
- Jos luovutaan värityksen ehdottomasta optimaalisuudesta, voidaan luoda algoritmi, joka tuottaa nopeasti kohtuullisen hyvän tuloksen, muttei välttämättä aina ja/tai aivan parasta ratkaisua.
  - Heuristinen algoritmi, satunnaistettu algoritmi (harjoitustehtävänä Javalla TRA II -kurssilla!).

Suoraviivainen algoritmi:

0. Väritysehto: minkään kaaren molemmissa päissä ei saa käyttää samaa väriä.
1. Väritä yhdellä värillä niin monta solmua kuin väritysehto sallii.
2. Väritä jäljelle jääneistä solmuista mahdollisimman monta seuraavalla värillä.
3. Toista vaihetta 2 kunnes kaikki solmut on väritetty.

⇒ Tämä on ns. ahne algoritmi.

- Aloitetaan jostain, käsitellään kaikki mitä voidaan, jatketaan.
- Ei uudelleenyrityksiä, heuristiikkaa tms.
- Löytää joskus optimaalisen tuloksen, mutta ei aina.
- Tulokseen vaikuttavat mm.
  - aloitussolmu
  - solmujen tarkastelujärjestys.

Ylläoleva kuvaa algoritmin hyvin, mutta ei ole läheskään suorituskelpoinen tietokoneella joten joudumme vielä tarkentamaan sitä.

Kirjoitetaan algoritmi tarkempaan muotoon ja vaihdetaan samalla kieltä. Edellisen version vaiheet 0-2 tehdään seuraavalla metodilla *greedyColor* ja käynnistäminen, uuden värin käyttöönotto ja lopetus (vaihe 3) metodilla *greedyColorStart*.

```
public static int greedyColorStart(Graph G) {                          1
    mark all vertices non-colored;                                      2
    int numOfColors = 0;                                                3
    while (not all vertices colored)                                    4
        greedyColor(G, ++numOfColors);                                  5
    return numOfColors;                                                 6
}                                                                       7
                                                                        8
public static void greedyColor(Graph G, Color newColor) {              9
    for each uncolored vertex v of G                                    10
        if (v not adjacent to any vertex with color newColor)           11
            v.setColor(newColor);                                       12
}                                                                       13
```

⇒ Tämä on Java-pohjaista algoritminotaatiota – näyttää Javalta, edelleen ihmiselle täsmällinen, mutta ei vielä kelpaa kääntäjälle

Tätä on kuitenkin helppo tarkentaa eteenpäin kunhan tunnemme verkon operaatiot (jotka esitellään TRA II -kurssilla). Seuraavassa sama algoritmi, mutta rivit 10 ja 11 tarkennettuna:

```
public static void greedyColor(Graph G, Color newColor) {
    for (Vertex v : G.vertices()) {
        if (v.getColor() != noColor)
            continue;
        boolean found = false;
        for (Vertex w : v.neighbors())
            if (w.getColor() == newColor)
                found = true;
        if (! found)
            v.setColor(newColor);
} }
```

⇒ Tämä kelpaa Java -kääntäjälle ja antaa suoritettuna kelvollisen värityksen (tosin ei aina optimaalista)! Ei toki ole ihan yhtä luettavaa, mutta ei mahdotontakaan.

- Tarvittaessa voimme lisätä tarkennusvaiheita, esim. suomenkielisen ja ensimmäisen ”Java” -version väliin täsmällisempi (ohjelmointikielimäisempi) suomenkielinen versio.

1.2 Abstraktit tietotyypit

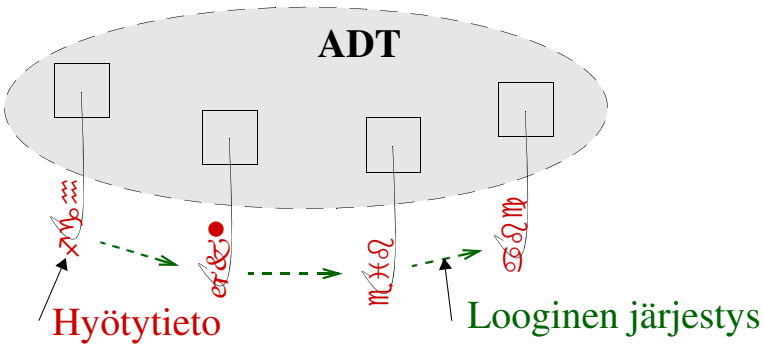
⇒ Abstraktit tietotyypit (ATT, ADT (*abstract data type*)) (joskus abstrakti tietorakenne) ovat malleja tiedon esittämiseksi ja käsittelemiseksi.

Abstrakti tietotyyppi määritellään kuvaamalla ne operaatiot, joilla tietotyypin esittämää tietoa käsitellään.

⇒ Operaatiot (yms.) = abstraktin tietotyypin liittymä (”käyttöliittymä”).

Vaihtoehtoinen havainnollistus

- ADT on (hyötytieto)kokoelmien hallinnan apuväline.
- Ripustetaan hyötytiedot (elementit, alkiot) ADT:n hallittavaksi.



Kuva 1.5: ADT kokoelman ylläpidon apuvälineenä

⇒ Keskitytään itse elementteihin ja niiden keskinäisiin suhteisiin!

- ADT:n valinta määrää talletettujen elementtien loogisen järjestyksen (jos sellaista tarvitaan).
  - Joskus vain osa elementeistä on ”näkyvissä” (kun emme muita tarvitse).

Esimerkki 1.1. Joukko (abstraktina tietotyyppinä) on kokoelma keskenään samantyyppisiä alkioita, vaikkapa henkilötietoja, kuvia, merkkijonoja tai verkon solmuja. Joukkomallille tyypillinen piirre on se, että sama alkio voi sisältyä joukkoon vain yhtenä esiintymänä kerrallaan. Joukkoja käsitellään

esimerkiksi muodostamalla kahden joukon yhdiste tai tutkimalla, kuuluuko jokin alkio joukkoon vai ei. Joukkotyypin liittymä voi sisältää vaikkapa seuraavankaltaisen osan:

```
// returns union of this Set<E> and Set<E> B
public Set<E> union(Set<E> B);

// returns whether element x is a member of this Set or not
public boolean member(E x);
```

Liittymä (*interface*) antaa ADT:n käyttäjälle kaiken tarpeellisen tiedon sen käyttämiseksi:

- tyypit
- sallitut operaatiot (funktiot/metodit)
- operaatioiden parametrit
- operaatioiden tuloksen tyypin
- operaatioiden käyttöä rajoittavat ehdot
- operaatioiden aikavaativuudet

⇒ ADT toteutetaan siten, että liittymässä kuvatut operaatiot toimivat kuvatulla tavalla!

- Toteutus sisältää operaatioiden ohjelmakoodin ja mallia vastaavan todellisen toteutustietorakenteen määrittelyn.
  - Toteuttamisen helpottamiseksi toteuttaja voi laatia lisää operaatioita omaan käyttöönsä.
    - Nämä operaatiot eivät näy käyttäjälle (kuten ei mikään muukaan toteutukseen liittyvä asia).

Määritelmä 1.2. (Ohjelmointikielen) Muuttujan tietotyyppi (*data type*) on kyseisen muuttujan sallittujen arvojen joukko.

Esimerkki 1.3. Kokonaislukujen tyyppi sisältää periaatteessa äärettömän monta arvoa: 0, 1, –1, 2, –2, 3, –3, ... Käytännössä tietokoneen sananpituus rajaa mahdollisten arvojen joukon aina äärelliseksi. Esimerkiksi 32 bitillä voidaan esittää 4 294 967 296 eri lukua.

Pascal-kielen tyypin *set of 0..9* arvoja ovat joukot { }, {0}, {1}, {2}, {3}, ..., {9}, {0,1}, {0,2}, ..., {8,9}, {0,1,2}, ..., {0,1,2,3,4,5,6,7,8,9}. Kaikkiaan näitä arvoja on 2^{10} eli 1024 erilaista.

Totuusarvotyypissä (*boolean*) erilaisia arvoja on vain kaksi, *false* ja *true*.

Määritelmä 1.4. Tietorakenne (*data structure*) on kokoelma toisiinsa kytkettyjä muuttujia. Kyseessä on siis ohjelmointikielen rakenne josta kaikki bitit voidaan periaatteessa piirtää näkyviin. Joskin Javan tapauksessa bittien täsmällinen piirtäminen vaatisi hieman enemmän paneutumista kääntäjän ja virtuaalikoneen toteutukseen. Pelkkä kielen dokumentaatio ei riitä.

- taulukko
- objektit
- viittein kytketyt objektit
- tiedosto

⇒ Abstrakti tietotyyppi: kokoelma-abstraktio ((tarkka) toteutus tuntematon).

⇒ Tietorakenne: ohjelmointikielen rakenne (bitit piirrettävissä).

- Tällä kurssilla opiskellaan nämä molemmat ja niiden yhteys.
- Käytännössä nämä menevät usein sekaisin kielenkäytössä.

1.3 Suorituksen vaativuus

Vaatimukset algoritmille:

0. Algoritmin pitää antaa oikea tulos.
1. Algoritmin tulee olla (riittävän) helppo ymmärtää ja toteuttaa.
2. Algoritmin tulee olla muistitilan ja ajankäytön suhteen (riittävän) tehokas.

Tehokkuus?

- aikavaativuus
  - paljonko aikaa kuluu?
- tilavaativuus
  - paljonko muistia kuluu?
- laitteistovaativuus
  - minkälainen laskentatarkkuus tms. tarvitaan?

Tällä kurssilla aikavaativuus on keskeinen, tilavaativuutta käsitellään jonkin verran, laitteistovaativuus ei yleensä ole merkityksellinen pseudokielisissä peräkkäisohjelmissa.

Miten suoritusaikaa (aikavaativuutta) mitataan?

- Absoluuttinen seinäkelloaika tai kulutettu prosessoriaika (sekuntia tms.)?
  - Loppukäyttäjälle tärkeä, tai ainakin joskus.
  - Huono, sillä kaikki vaikuttaa:
    - syöte
    - käytetty laitteisto
    - kuorma
    - välimuistit
    - mittavälineetkin voivat olla epätarkkoja
  - Huono, sillä vain jo toteutettua algoritmia voidaan tarkastella
  - Toki valmiille ohjelmalle seinäkelloaika on kiinnostava, jopa etukäteen arvioituna.
    - Palataan tähän TRA II -kurssilla.

⇒ Jotain abstraktimpaa kuin sekunti tms. siis tarvitaan.

- Kellojakso (tai konekielioperaatio) on parempi, mutta edelleen em. asiat vaikuttavat.
  - Edelleen vain jo toteutettua algoritmia voitaisiin tarkastella (järkevästi).

⇒ Käytetään sensijaan yksikköä ”askel” (*step*).

```
a = 1;                          1
b = 3;                          2
```

(n.) kaksi askelta

```
for (i = 0; i < 1000; i++)      1
    a = a + 1;                  2
```

(n.) 1000 askelta

Kuva 1.6: Suoritusaskel

- Askel on yksittäinen toimenpide algoritmissa. (Tarkempi määritelmä alempana)
  - Ylläolevan kaksiaskelisen ohjelmanpätkän konekielinen versio on 2 tai 4 konekielikäskyä, riippuen muuttujien toteutuksesta ja kääntäjästä.

  - 1000-askelisen toiston konekäskyjen määrä riippuu vahvasti kääntäjän optimoinnin asteesta, joka tapauksessa se on jokin 1000:n monikerta + muutama käsky.
  - Näiden konekielikäskyjen suoritusaika voi olla 1-100 ns, riippuen muuttujien talletuspaikasta...
- Hetken kuluttua nähdään, ettei laskemisen tarkkuudesta kannata stressata.

Suoritusaika suhteutetaan käsiteltävän syötteen kokoon.

- Syötteen koko vaihtelee eri suorituskerroilla, eikä sitä tiedetä etukäteen!
- Järjestäminen (eli lajittelu) 1000 askeleessa (tai 0,1 sekunnissa). Hyvä vai huono?
  - 10, 1.000, 1.000.000, 1.000.000.000 alkiota?
- Syötteen koko: alkioiden määrää, tai joskus joku muu vastaava suoritusajan määräävä suure.
- Siirräntään kuluvaa aikaa ei yleensä oteta huomioon.
  - Syötteen oletetaan yleensä olevan valmiina saatavilla (viitattavissa).

⇒ Esitetään suoritusaika syötteen koon (tms.) funktiona!

```
for (i = 0; i < n; i++)      1
    a = a + 1;               2
```

(n.) n askelta

⇒ Aikavaativuusfunktion mittayksikköä ei kiinnitetä. Sanomme sitä askeleeksi (tai jätämme yksikön mainitsematta).

- Todellinen yhden askeleen aikavaativuus riippuu mm. askeleessa suoritettavasta toimenpiteestä ja käytettävästä laitteistosta.
- Yhden askeleen aika ei kuitenkaan riipu syötteen koosta (alkioiden määrästä).
- Edelleen askeleita voi olla n, $3\times n$, $3\times n+4$, tms. kappaletta, laskutavasta riippuen, mutta tällä ei ole merkitystä.

Esimerkki 1.5. Suoritusaikafunktio $T(n)=cn^2+b$, missä b ja c ovat vakioita, ilmaisee suoritusajan olevan neliöllinen suhteessa syötteen kokoon n. Tämä merkitsee, että syötteen koon kymmenkertaistuessa, suoritusaika suurin piirtein satakertaistuu.

⇒ Aikavaativuus on siis syötteen koon funktio (tai jokin muun aikavaativuuden määräävän suureen funktio).

- $T(n)=f(n)$ luetaan "Algoritmin aikavaativuus (T = Time) alkiomäärällä n ilmaistaan funktiolla $f(n)$."
- Usein käytämme muuttujaa n kuvaamaan syötteen kokoa.
  - Tämä on kuitenkin aina muistettava mainita ellei se ole ilmeistä.
  - Yleensähän syöte on jokin lista, joukko, tms. Tällöin mainitaan esimerkiksi, että $n=|Laskut|$, missä *Laskut* on parametrina saatu laskujen joukko.
- Joskus myös muita muuttujia tarvitaan, jos n on varattu toiseen tarkoitukseen.
  - Erityisesti, jos kuvattavia syötteitä on useita erikokoisia.

Esimerkki 1.6. Esimerkiksi merkkijonon esiintymien etsimisen toisesta merkkijonosta aikavaativuus riippuu sekä etsittävästä avaimesta, että läpikäytävästä kohdetekstistä.

Kuvataan avaimen pituutta m:llä ja kohdetekstin pituutta n:llä. Erään yksinkertaisen etsintäalgoritmin suoritusaikafunktio on $T(n,m)=cnm$, missä c on vakio.

Palataan suoritusajan laskentaan annetusta algoritmista kohdassa 1.4 (s. 15).

Suoritusaika ei aina riipu pelkästään syötteen koosta, vaan myös syötteen laadusta.

Kun tämä otetaan huomioon, voidaan tarkastelu eriyttää seuraaviin kolmeen tapaukseen:

1. $T(n)$ tarkoittaa pahimman tapauksen suoritusaikaa eli pisintä mahdollista n:n kokoisen syötteen vaatimaa suoritusaikaa.
2. $T_{avg}(n)$ tarkoittaa keskimääräistä suoritusaikaa eli kaikkien n:n kokoisten syötteiden vaatimien suoritusaikojen keskiarvoa.
3. $T_{best}(n)$ tarkoittaa parhaan tapauksen suoritusaikaa eli lyhintä mahdollista n:n kokoisen syötteen vaatimaa suoritusaikaa.

⇒ Yleensä tarkastellaan pahinta tapausta.

- Paras tapaus ei yleensä ole edes mielenkiintoinen.
  - Esimerkiksi järjestämisessä usein paras tapaus on valmiiksi järjestetty taulukko.
    - Valmiiksi järjestetyn taulukon ”järjestämisen” aikavaativuus taas ei juuri kerro itse järjestämisalgoritmista mitään.
- Keskimääräisen suoritusajan arviointi saattaa puolestaan osoittautua erittäin hankalaksi tehtäväksi, koska kaikki samankokoiset syötteet voidaan vain harvoin olettaa keskenään yhtä todennäköisiksi.
  - Erilaisia syötteitä on yleensä eksponentiaalinen määrä syötteen koon suhteen.
  - Jos keskimääräinen tapaus eroaa olennaisesti pahimmasta tapauksesta, se toki on kiinnostava.
- Pahimman tapauksen yläraja (ja parhaan alaraja) on yleensä helpoin laskettava.
- Pahimman tapauksen yläraja on usein suunnitteluvaiheessa kiinnostavin.

Funktioiden yksinkertaistaminen kertaluokkatarkastelulla

- Koska algoritmista on mahdotonta ja/tai työlästä laskea täsmällistä aikavaativuusfunkiota täsmällisillä vakiokertoimilla, tyydymme tarkastelemaan aikavaativuusfunktioiden kertaluokkia.
  - Saman kertaluokan funktiot käyttäytyvät samalla tavalla kun syöte kasvaa suureksi.
    - Samalla tavalla: eroavat vain vakiokertoimella.
    - Kertaluokkatarkastelun täsmälliseen määritelmään palataan TRA II -kurssilla.
  - Kun yleensä analysoimme pahinta tapausta (suurinta suoritusaikaa), niin on helpointa hakea myös kertaluokkatarkastelussa aikavaativuuden yläraja.
    - Funktio jonka arvo on aina vähintään yhtä suuri kuin aikavaativuusfunktion (vakiokertoimella täydennettynä), kun syöte kasvaa suureksi.
    - Siis:
      1. jätämme vakiokertoimet huomiotta ja
      2. tarkastelemme vain ”suuria” syötteen kokoja.

Määritelmä 1.7. Kertaluokkamerkintä O:

$T(n) = O(f(n))$, jos on olemassa positiiviset vakiot c ja n_0 siten, että $T(n) \leq c \times f(n)$, kun $n \geq n_0$. [Lue: $T(n)$ on kertaluokkaa $f(n)$, ”iso-O”, ”ordo”; ”rajoittaa ylhäältä”]

⇒ Siis: kunhan valitaan sopiva vakio c ja annetaan syötteen kasvaa riittävästi, niin $T(n) \leq c \times f(n)$ pätee kaikilla suuremmilla syötteen ko’oilla (n:n arvoilla).

$\Rightarrow$ Siis: eroaa korkeintaan vakiokertoimella c kun syöte on riittävän suuri.

Käytössä on myös merkinnät:

- Ω (rajoittaa alhaalta),
- Θ (rajoittaa ylhäältä ja alhaalta),
- o (rajoittaa aidosti ylhäältä) ja
- ω (rajoittaa aidosti alhaalta).

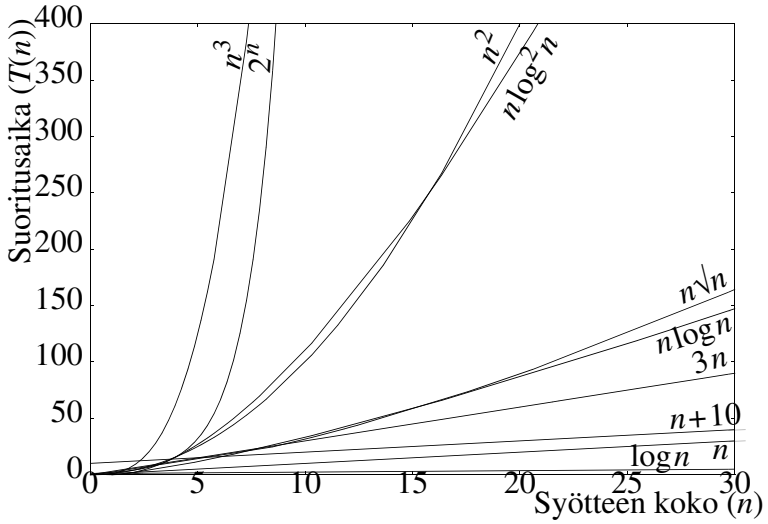
Palaamme näihin sekä O-merkinnän täsmälliseen määritelmään ja kertaluokkatodistuksiin TRA II -kurssilla.

Aikavaativuusfunktioiden luokittelu

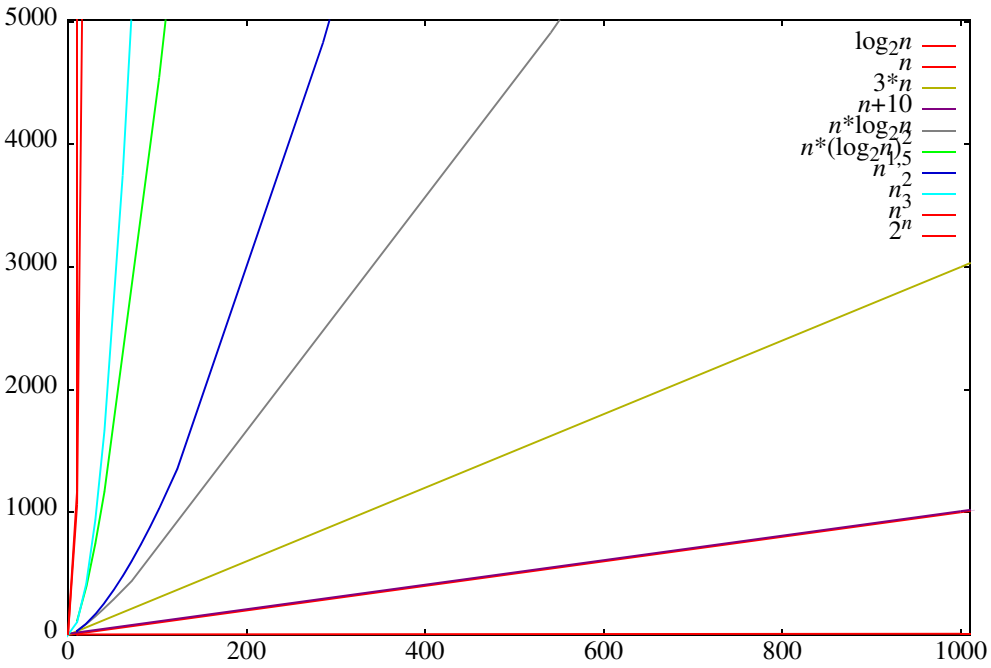
- Eksponentiaaliset aikavaativuudet (kasvavat "räjähdysmäisesti")
  - esim. 2^n, 3^n, $2^n/n$
  - käyttökelpoisia vain hyvin pienille syötteille
- Polynomiset aikavaativuudet (kasvavat syötteen mukana)
  - esim. $\sqrt{n}$, n, n^2, n^3, n^5, n^{12345}, $n\sqrt[3]{n}$, $n\log n$, jne
  - käytännössä yleisimpiä
  - tehokkaita .. kohtuullisen tehokkaita
  - n lineaarinen
  - n^2 neliöllinen
- Logaritmiset aikavaativuudet (kasvavat vain hyvin vähän)
  - esim. $\log n$, $\log\log n$, jne
  - huom $\log n$ on tällä kurssilla (ja tkt:ssä muutenkin melkein aina) $\log_2 n$
  - kaikki funktiot jotka ovat vähemmän kuin $O(n)$ on alilineaarista aikavaativuutta ($o(n)$)
    - kaikkia syötteen alkioita ei siis edes voida käsitellä.
- Vakioaikainen: $O(1)$ (aikavaativuus ei kasva vaikka n kasvaa).

$\Rightarrow$ Muistetaan huomioida merkitsevin osa aikavaativuusfunktiota!

Esimerkki 1.8. Kuva 1.7 esittää kymmenen aikavaativuudeltaan eri kertaluokkaa olevan ohjelman suoritusajat syötteen koon (n) funktioina. Kuva voi kuitenkin hämätä, kuten kuvista 1.8 ja 1.9 näemme.



Kuva 1.7: Suoritusaikafunktioita kuvaajana.



Kuva 1.8: Suoritusaikafunktioita kuvaajana, isompi mittakaava.

Aikavaativuus $T(n)$, ms	Ohjelman käsittelemän syötteen koko n annetussa ajassa:				
	Sekunnissa	Minuutissa	Tunnissa	Päivässä	Vuodessa
$\log n$	10^{301}	$10^{18\,061}$	$10^{1\,083\,707}$	$10^{26\,008\,991}$	$10^{9\,493\,281\,943}$
n	1 000	60 000	3 600 000	86 400 000	31 536 000 000
$n+10$	990	59 990	3 599 990	86 399 990	31 535 999 990
$3n$	333	20 000	1 200 000	28 800 000	10 512 000 000
$n\log n$	140	4 895	204 094	3 943 234	1 052 224 334
$n\log^2 n$	36	678	18 013	266 037	48 391 041
$n\sqrt{n}$	100	1 532	23 489	195 438	9 981 697
n^2	31	244	1 897	9 295	177 583
n^3	10	39	153	442	3 159
2^n	9	15	21	26	34

Kuva 1.9: Suoritusaikafunktioita taulukkona.

Kertaluokkasäännöt:

- Kaikille vakioille $k>0$, $b>0$ pätee $\log^k n$ on aidosti pienempää kertaluokkaa kuin n^b. (merkintä $\log^k n$ tarkoittaa $(\log n)^k$)
  - Siis kaikki logaritmiset funktiot kasvavat hitaammin kuin kaikki polynomiset funktiot.
- Kaikille vakioille $a>1$, $b>0$ pätee n^b on aidosti pienempää kertaluokkaa kuin a^n.
  - Siis kaikki exponentiaaliset funktiot kasvavat nopeammin kuin kaikki polynomiset funktiot.
- Muista tarkastella merkittävintä (suurinta) komponenttia funktiosta.
  - n kasvaa hitaammin kuin $n\log n$ (vrt. $1000<1000\times 2$)

Kasvunopeuksien vertailu

- Funktioiden kasvunopeuksien välinen vertailu on yleensä helppoa kun muistetaan edellinen luokittelu.
- Täsmällisesti vertailu voidaan tehdä tarkastelemalla funktioiden osamäärän raja-arvoa n:n (syötteen koon) kasvaessa rajatta. Tähän palataan TRA II -kurssilla.

Esimerkki 1.9. $n\log n$ vs. $n^{1.5}$

Funktioita voidaan vertailla myös epämuodollisesti päättelemällä, ikäänkuin epäyhtälöä ratkaisten, mutta aluksi tietämättä mihin suuntaan vertailu on:

$$n\log n \quad ?? \quad n^{3/2} \qquad | \quad :n \tag{1.1}$$

$$\log n \quad ?? \quad n^{1/2} \qquad | \quad ()^2 \tag{1.2}$$

$$\log^2 n \quad ?? \quad n \tag{1.3}$$

- Muistetaan, että logaritminen $log^k n$ kasvaa hitaammin kuin polynominen n.
- $n^{3/2}$ kasvaa siis nopeammin.

⇒ Onko mahdollista laatia aina vain entistä tehokkaampia algoritmeja kuhunkin ongelmaan?

- Joihinkin ongelmiin tunnetaan vain tehottomia (ts. hitaita, yleensä eksponentiaalisen aikavaativuuden) ratkaisuja:
  - esim. verkon väritys, kauppamatkustajan ongelma
  - tehokas ratkaisu saattaa olla olemassa, mutta kukaan ei ole sitä vielä keksinyt (tai sitten sitä ei ole olemassa)
  - useimpiin vaikeisiin ongelmiin tunnetaan nopeampia heuristisia ratkaisuja (antavat suhteellisen hyvän ratkaisun nopeammin).
- Monille ongelmille on todistettu aikavaativuuden alaraja:
  - esim yleinen (vertailuun perustuva) järjestäminen on $\Omega(n\log n)$ (TRA II)
  - nämä todistukset voivat olla vaikeita
  - alarajan saavuttaminen voi olla vaikeaa, alarajan saavuttavaa algoritmia ei välttämättä ole keksittykään (ts. ei ehkä tiedetä onko alaraja tiukka, eli voidaanko se saavuttaa)
- On myös ongelmia, joita ei voi ratkaista algoritmisesti.
  - esim. pysähtymisongelma

⇒ Laskennan perusmallit, ASA -kurssit ...

1.4 Suoritusajan (kertaluokan) laskeminen algoritmista

⇒ Annetun algoritmin suoritusajan laskenta voi olla hyvin helppoa tai hyvin vaikeaa.

Yleensä riittää osata peruslaskentoa:

- yhteen- ja kertolaskua
- sarjojen summien sieventämistä
  - kertaluokkalaskenta auttaa huomattavasti, ei tarvita tarkkoja tuloksia
- logaritmi- ja eksponenttilaskentaa (joissakin algoritmeissa)
- rekursioyhtälöitä (rekursiivisissa algoritmeissa)

Kertaluokkalaskennan perussäännöt

- Olkoon ohjelmanosan P_1 suoritusaika $T_1(n)=O(f(n))$

- Olkoon ohjelmanosan P_2 suoritusaika $T_2(n)=O(g(n))$

Summasääntö

- $T_1(n)+T_2(n)=O(\max\{f(n),g(n)\})$

⇒ Tulkinta: peräkkäisten suoritusten aikavaativuuden kertaluokan määrää osasuorituksista hitaampi.

- Siis aikavaativuusfunktioista se jolla on suurempi kasvunopeus.
- Jos $T(n)=a_1n^k+a_2n^{k-1}+\cdots+a_{k-1}n^2+a_kn+a_{k+1}$, niin $T(n)=O(n^k)$.
- Summasäännön ansiosta voidaan kertaluokkatarkastelussa jättää vähempimerkityksiset termit kokonaan huomiotta.
- Todistus johdettavissa kertaluokkalaskennan määritelmästä 1.7 (s. 12).

Tulosääntö

- $T_1(n)\times T_2(n) = O(f(n)\times g(n))$.
- Seuraus: jos $T(n)=a\times f(n)$, (a on nollaa suurempi vakio), niin $T(n)=O(f(n))$.

⇒ Eli: kertaluokkatarkastelussa voidaan positiiviset vakiokertoimet jättää huomiotta.

Esimerkki 1.10.

$$\begin{aligned} T(n) &= 3n^2+5n+8, \\ &= O(\max(3n^2,5n,8)) && \text{(summasääntö)} \\ &= O(3n^2) && \text{(tulosääntö)} \\ &= O(n^2). \end{aligned}$$

Aikavaativuuden laskeminen algoritmista

⇒ Lasketaan sisältä ulos!

Esimerkki 1.11. Kuplajärjestämisalgoritmi voidaan esittää muodossa:

```
public static void bubbleSort(Comparable A[]) {                    1
   for (int i = 0; i < A.length−1 ; i++) {                         2
      for (int j = A.length−1; j > i ; j−−) {                      3
         if (A[j−1].compareTo(A[j]) > 0) {                         4
            Comparable tmp = A[j−1];                               5
            A[j−1] = A[j];                                         6
            A[j] = tmp;                                            7
         }                                                         8
      }                                                            9
}  }                                                               10
```

- Syöte on parametrina saatava järjestettävä taulukko.
- Tässä syötteen koko on järjestettävien alkioiden lukumäärä $A.length$, merkitään sitä n:llä.

Aikavaativuus:

- Rivit 5 - 7 kukin vakioajassa (suoritusaika ei riipu syötteen A alkiomäärästä n):
  - $O(\max\{1,1,1\}) = O(1)$
- Rivin 4 ehto vakioajassa (jos alkiot yksinkertaisia):
  - kaksi tapausta: rivit 5 - 7 joko suoritetaan, tai ei
  - joka tapauksessa rivit 4 - 8 vakioajassa
- Toiston askelmuuttujan hallinta $O(1)$
- Rivin 3 toistoja $(n-1)-(i+1)+1 = n-i-1 = O(n-i)$ kappaletta
- Rivin 3 oma aikavaativuus $O(1)$
- Tulosäännöllä rivit 3 - 9 : $O((n-i)\times 1) = O(n-i)$
  - huomaa, ettei lauseketta $n-i$ vielä "osata" sieventää!
- Ulompi toisto $n-1$ kertaa, askelmuuttujan käsittely $O(1)$.
  - Toistetun osan 3-9 suoritus on $O(n-i)$ ja i muuttuu toiston edistyessä!
- Tuloksena summa

$$(n-1)+(n-2)+\cdots+(n-i)+\cdots+(n-(n-1)) \tag{1.4}$$

$$=\sum_{i=1}^{n-1}(n-i) = (n-1)\frac{(n-1)+1}{2} = \frac{1}{2}n^2 - \frac{1}{2}n = O(n^2) \tag{1.5}$$

- Koko algoritmin suoritusaika siis $O(n^2)$.
- Summasarjan laskeminen/sieventäminen/arvioiminen on tarpeettoman työlästä useimpien algoritmien analyysissä.
  - Huomataan sensijaan, että sisempi silmukka (rivi 3) pyörii ensimmäisellä kerralla noin n kierrosta, sitten lyhenee tasaisesti ja lopuksi vain yhden kierroksen.
  - Näinollen sisempi silmukka pyörii keskimäärin $\frac{n}{2}$ kierrosta!
  - Koko aikavaativuus siten n (ulompi silmukka) $\times$ $\frac{n}{2}$ (sisempi silmukka) $\times$ $O(1)$ (sisäosat), eli $n \times \frac{n}{2} \times O(1) = O(n^2)$.

Tilavaativuus

- syöte: järjestettävä taulukko $O(n)$
- järjestämisen aikana käytetään kokonaislukumuuttujia i, j, *tmp*.
  - tilantarve 1+1+1 sanaa $= O(1)$
- tulos: järjestettty taulukko
  - ei vaadi omaa tilaa, järjestäminen tapahtuu "paikallaan"
- Aputilan tarve $O(1)$.

$\Rightarrow$ Huomaa: tilavaativuus analysoidaan vain aputilan osalta, koska syötteen (ja tuloksen) täytyy joka tapauksessa olla mukana!

Yleisiä ohjeita suoritusajan laskemiseksi

- Lasketaan sisältä ulos (mikäli mahdollista).
  - Joskus joudutaan tarkastelemaan kokonaisuutta (mitä operaatioita tehdään koko ohjelman(osan) suorituksen aikana).
- Sijoitus-, luku- ja tulostustoiminnot ovat yleensä $O(1)$.
  - poikkeus: suurten rakenteiden käsittely
  - huomaa parametrien tyyppi
    - taulukko arvoparametrina: kopiointi $O(n)$
    - taulukko muuttuja/viiteparametrina: viite $O(1)$
- Lausekkeen arvottaminen $O(1)$
  - Paitsi tietysti, jos esim. kutsutaan aliohjelmaa!

- Taulukkoviittaukset $O(1)$
- Toimintojono (= peräkkäiset käskyt)
  - Summasäännöllä
- Ehdolliset toiminnot
  - T(ehdon arvottaminen)+T(valittava osa) [yleensä pahin tapaus]
- Toisto
  - Tulosäännöllä
  - $\sum_i$(i:s toistokerta)
  - Usein riittää laskea
    - Toistojen_lkm$\times T$(toistettava_osa) [pahin tapaus].
    - Jos keskimääräinen tapaus on selkeä, niin sitä voidaan käyttää.

⇒ Jos pahin tapaus on hyvin harvinainen, tilanne on analysoitava tarkemmin (laskettava pahat ja hyvät tapaukset erikseen).

```
for (i = 0; i < n; i++)                                                    1
  if (i == n−1)                                                            2
    for (j = 0; j < n; j++)     // tämä suoritetaan vain kerran, O(n)      3
      a = a + 1;                                                           4
  else                                                                     5
    x = x + 1;          // tämä vakioaikainen suoritetaan useasti          6
```

- Mitään oleellista ei saa jättää huomiotta.

```
while (! etsi_väritys(k))                                                  1
  k = k + 1;                                                               2
```

- aliohjelma/metodikutsut
  - T(parametrien välitys)+T(aliohjelman suoritus)
- aliohjelmat/metodit
  - erillisenä lohkona
  - rekursiiviset rekursioyhtälöllä

Esimerkki 1.12. Useimmiten ohjelmissa käytetään jotain muuta toistoa kuin suoraa *for*-laskuria eivätkä kaikki operaatiot ole vakioaikaisia.

```
/* Lisää kokoelmaan A kaikki kokoelmassa B esiintyvät alkiot jolleivat ne siellä jo ole */   1
public static void lisaaKaikkiUudet(Collection A, Collection B) {                            2
  for (Object x : B)                                                                         3
    if (! A.contains(x))                                                                     4
      A.add(x);                                                                              5
}                                                                                            6
```

- Näppärä ja toimiva algoritmi, mutta onko se tehokas?
- Siis: mikä on tämän aikavaativuus?
  - Voidaanko sitä edes tietää?
  - Mitkä kaikki kohdat vaikuttavat aikavaativuuteen?
- Kurssin lopussa pitäisi osata vastata tähän kaikissa eri tilanteissa!

Esimerkki 1.13. Usein käytettävä kokoelma (tietorakenne) vaikuttaa operaatioiden aikavaativuuteen.

```
// Lisää  listaan A kaikki listassa B esiintyvät alkiot jolleivat jo ole siellä,          1
public static void lisaa KaikkiUudet(List A, List B) {     // HIDAS ALGORITMI, ÄLÄ KÄY-2
TÄ
  for (int ib = 0; ib < B.size(); ib++) {                                                  3
```

```
            boolean loytyi = false;                                    4
            for (int ia = 0; ia < A.size(); ia++)                      5
                if (A.get(ia).equals(B.get(ib))) {                     6
                    loytyi = true; break;                              7
                }                                                      8
            if (! loytyi)                                              9
                A.add(B.get(ib));                                      10
    }   }                                                              11
```

- Toimiva algoritmi, mutta onko se tehokas?
- Siis: mikä on tämän aikavaativuus?
  - Voidaanko sitä edes tietää?
  - Mitkä kaikki kohdat vaikuttavat aikavaativuuteen?
- Jos A ja B ovat *ArrayList*, niin *get*() ja *add*() ovat vakioaikaisia.
  - Niillä tiedoilla koko aikavaativuus osataan laskea.
- Kurssin lopussa pitäisi osata vastata tähän kaikissa eri tilanteissa!

Esimerkki 1.14. Kertoma voidaan laskea seuraavalla rekursiivisella algoritmilla:

```
public static int factorial(int n) {                                   1
    if (n <= 1)                                                        2
        return 1;                                                      3
    else                                                               4
        return n * factorial(n-1);                                     5
}                                                                      6
```

- Rivien 1-4 suoritusaika on selvästi $O(1)$.
- Rivillä 5 kertolasku ja sijoitus ovat $O(1)$, mutta:

⇒ Myös funktiokutsun arvottaminen vie aikaa!

- Merkitköön $T(n)$ koko algoritmin suoritusaikaa syötteellä n:

$$T(n) = \begin{cases} d, \text{ kun } n \leq 1 \\ c + T(n-1), \text{ kun } n > 1 \end{cases} \qquad (c,d \text{ ovat vakioita}) \tag{1.6}$$

- Puretaan äskeistä rekursiokaavaa:

$$\begin{aligned} &T(n) = c + T(n-1) \\ &\quad T(n-1) = c + T(n-2), \text{ ja} \\ &\quad\quad T(n-2) = c + T(n-3) \\ &\quad\quad\quad \ldots, \text{ joten} \\ &\quad T(n-1) = c + (c + T(n-3)) = 2c + T(n-3), \text{ joten} \\ &T(n) = c + T(n-1) = c + (2c + T(n-3)) = 3c + T(n-3) \end{aligned} \tag{1.7}$$

- Jatkamalla syvemmälle tähän tapaan huomaisimme, että

$$T(n) = ic + T(n-i), \text{ kun } i < n \tag{1.8}$$

- Asetetaan $i = n-1$ (mennään rekursiossa pohjalle):

$$\begin{aligned} &T(n) = c(n-1) + T(1) = c(n-1) + d \\ &\Rightarrow T(n) = O(n). \end{aligned} \tag{1.9}$$

Päättely voitaisiin helposti todistaa induktiolla, mutta sivuutetaan se nyt.

Rekursiivisten aliohjelmien aikavaativuuteen palataan TRA II-kurssilla.

1.5 Hieman algoritmien oikeellisuudesta

[Matti Nykänen, tarkempi läpikäynti 2018 tallenteessa luento 4, 42:20 alkaen].

Keskeinen väline oikein toimivien algoritmien kehittämisessä ovat

invariantit eli sellaiset väitteet, joka pysyvät tosina, vaikka algoritmin suorituksen edetessä moni asia muuttuukin.

Tällaisen invariantin voi liittää

muuttujaan jolloin se kertoo mitä tämä muuttuja tarkoittaa

koodirivien väliin jolloin se kertoo millainen tilanne on aina silloin kun suoritus kulkee tästä ohi.

Tarkastellaan esimerkkinä tuttua (?) *binäärihakua* – jota on yllättävän hankala saada ohjelmoitua täysin oikein!

Syötteenä saadaan epätyhjä taulukko A joka on kasvavassa järjestyksessä sekä kohde x.

Tuloksena halutaan tietää esiintyykö x taulukossa A.

Valitaan muuttujat ja niille invariantit:

i**:** "Taulukon alkuosa $A[0...i-1]$ on kokonaan $<x$."

j**:** "Taulukon loppuosa $A[j+1...A.length-1]$ on kokonaan $>x$."

osui: "Taulukkopaikka $A[m]$ on tutkittu ja siellä on x."

Koodin pitää siis päivittää näitä muuttujia vain siten, että nämä invariantit säilyvät tosina.

Vihje: Piirrä taulukko A ja siihen nämä invariantit!

```
1   i = 0;
2   j = A.length−1;
3   osui = FALSE;
    // Tarkista, että: onhan silmukkainvariantti (†) totta jo tässä välissä?
4   while i<j and not osui (‡)
        // Saat olettaa, että: silmukkainvariantti (†) on totta tässä välissä.
5       m = (i+j)div2;
6       if A[m]<x
7           i = m+1
8       elseif A[m]>x
9           j = m−1
10      else osui = TRUE;
        // Osoita oletusta ja silmukkaehtoa (‡) käyttäen, että:
        // silmukkainvariantti (†) on totta tässäkin välissä.
    // Silloin tiedät, että: tässä silmukan jälkeenkin silmukkainvariantti (†) on yhä totta
    // vaikka silmukkaehto (‡) ei enää olekaan.
11  return (osui or A[i]=x).
```

Muuttujien invariantit säilyvät tosina niitä päivitettäessä, kuten pitääkin.

Jos lopussa *osui* on...

TRUE niin tulos on oikein muuttujan *osui* oman invariantin nojalla

FALSE niin muuttujien i ja j invarianttien sekä silmukkaehdon $i < j$ epätotuuden nojalla vain taulukkopaikka $A[i]$ on enää mahdollinen.

Oikea toiminta vaatii myös, että emme koskaan vahingossa yritä indeksoida taulukon A ulkopuolelle.

Se voidaan varmistaa osoittamalla, että

$$0 \leq i \leq j < A.length$$

on eräs tämän *silmukan invariantti.*

Silmukkainvariantti (†) on sellainen koodissa olevaan silmukkaan liittyvä väite, joka on totta...

1. juuri ennen silmukkaa
   (silmukan käyttämien muuttujien saamien alkuarvojen perusteella)
2. myös silmukan rungon jälkeen, olettaen että se itse ja silmukkaehto olivat totta juuri ennen runkoa
   (eli runko säilyttää sen totena, vaikka muokkaakin muuttujien sisältöjä).

Silloin juuri silmukan jälkeen pätee se itse sekä silmukkaehdon vastakohta.

Silmukkainvariantti (†) on yksinkertaisempi tapa tehdä *induktiotodistus:*

Tarkista perustapaus: (†) on totta kun silmukka on pyörähtänyt $k=0$ kierrosta.

Osoita induktioaskel: Jos silmukka on pyörähtänyt k kierrosta ja se aloittaa vielä yhden lisäkierroksen $k+1$, niin (†) on totta myös lisäkierroksen lopussa.

Tiedät: Siis (†) on yhä totta kun silmukka on pyörinyt loppuun.

Binäärihakumme siis toimii *jos se pysähtyy.*

Silmukan pysähtyminen voidaan osoittaa "munakellolla" eli sellaisella lausekkeella

- jonka arvot ovat tyyppiä luonnolliset luvut $\mathbb{N}$
  (eli se "tikittää")
- että jos sen arvo on 0 niin silmukkaehto on epätosi
  (eli silmukka päättyy viimeistään silloin kun se "soi")
- jonka arvo silmukkarungon lopussa on aidosti pienempi kuin alussa
  (eli silmukkarungon suorittaminen kuluttaa "tikityksiä").

Tälle silmukalle munakelloksi käy lauseke $j-i$.

Silmukoita voi jopa *suunnitella lopusta alkuun* invariantti- ja munakelloajattelulla:

- Silmukan *jälkeen* haluamme, että

  kohde x on jo löytynyt **tai** sillä on enää yksi mahdollinen piilopaikka.

- Muokataan tavoitteemme muotoon

  ... **tai** (sen mahdolliset piilopaikat ovat $A[i...j]$ **ja** $i=j$)

lisäämällä muuttujat i ja j.

- Käännetään mahdollisuus mahdottomuudeksi

  ... **tai** ($A[0...i-1]$ on mahdoton **ja** $A[j+1...A.length-1]$ on mahdoton **ja** $i=j$).

- Invarianttiajattelun mukaan silmukan jälkeen pätee sen

  invariantti **ja ei** silmukkaehto.

- Valitkaamme invariantiksi

  $A[0...i-1]$ on mahdoton **ja** $A[j+1...A.length-1]$ on mahdoton.

- Siten silmukkaehdoksi jää

  kohdetta x ei vielä ole löytynyt **ja** $i<j$.

- Saamme invarianttimme voimaan ennen silmukan alkua alustamalla $i=0$ ja $j=A.length-1$.
- Tarvitsemme enää sellaisen silmukkarungon, joka muokkaa muuttujia i ja j siten, että

  munakello $j-i$ tikittää

  invariantit säilyvät voimassa.

  Ne voi tulkita vihjeiksi siitä, mitä kannattaa tehdä:
  - Jos löytyy $A[m]<x$ niin koko alkuosa $A[0...m]$ on mahdoton.
  - Jos löytyy $A[m]>x$ niin koko loppuosa $A[m...A.length-1]$ on mahdoton.
  - On monia tapoja valita m. Mikä niistä antaisi eniten informaatiota siitä, mikä on mahdotonta?

2 Abstraktit listatyypit

⇒ Aloitetaan yksinkertaisesta ja yleiskäyttöisestä kokoelmasta, listasta (*list*).

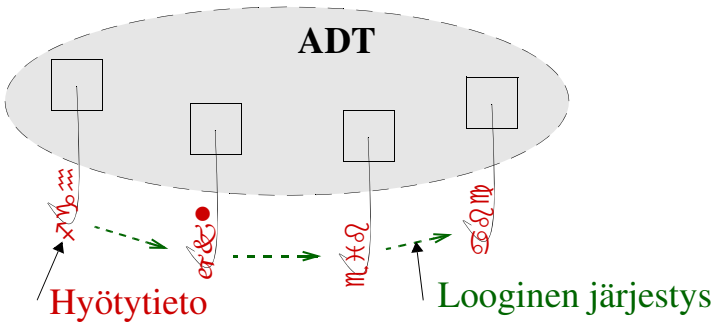
- Listalla on runsaasti variaatioita.
- Opitaan samalla yleisiä ADT periaatteita.

Abstraktin tietotyypin tarkoitus on hallita alkiokokoelmaa niin, että kokoelmaa voidaan käyttää

- käyttäjälle (siis ohjelmoijalle) tarkoituksenmukaisella tavalla
- tehokkaasti

Alkio (-tyyppi) voi olla mikä olio (luokka).

- Sovelluksen mukaan.
- Tällä kurssilla käytämme lähinnä dummy-alkioita...



Alkiokokoelmien hallinta ja käyttö edellyttää mahdollisuutta:

- lisätä kokoelmaan uusi alkio
- hakea alkioita kokoelmasta
- poistaa alkioita kokoelmasta
- muuttaa kokoelman alkioita
- käydä läpi kokoelman alkioita

⇒ Aina ei tosin tarvita näitä kaikkia mahdollisuuksia.

Mitä kokoelmaan (esim. listaan) talletetaan?

Tarkoitus on pystyä samalla ADT:llä hallitsemaan erilaisia alkiokokoelmia.

- Abstraktin tietotyypin tulee olla mahdollisimman riippumaton hallittavien alkioiden tyypistä.

⇒ Parametroidaan abstrakti tietotyyppi alkioiden tyypillä!

- Oliokielissä (kuten Java) tyyppiparametri on luokka ja lista ”sisältää” olioita
  - Perustyyppejä (*int*, *float*, jne) ei sellaisenaan voi laittaa Javan kokoelmiin vaan niiden olioksi paketoituja versioita (*Integer* *Float*, jne).
- Olioita käytetään viittauksilla, joten todellinen kokoelma sisältää viittauksia! vrt: kuva 1.5 (s. 8).
  - Aina välillä on hyvä piirtää täsmällinen kuva kokoelmasta ja olioista jotta rakenne hahmottuu mielessä oikein.
  - Usea kokoelma voi sisältää viittauksia samaan olioon.
  - Samaan kokoelmaan voi olla useita viittauksia.
  - Alkion poistaminen kokoelmasta tarkoittaa viittauksen poistamista kokoelmasta.
    - Jos samaan olioon on viittaus muualla, se säilyy ennallaan, samoin kuin olio itse.

Kertaus viittauksista

- Olioviittauksia X ja Y käytettäessä on muistettava mitä verrataan/käytetään:
  - $X == Y$ vertaa viittausten X ja Y samuutta (onko kyseessä sama olio).
    - $X = Y$ sijoittaa viittauksen samaan olioon.
    - $X = Y.clone()$ kopioi olion ja antaa siten uuden viittauksen (mutta **vain jos** Y:n luokka toteuttaa rajapinnan *Cloneable*).
    - Joillakin luokilla (esim. kokoelmilla) on myös konstruktori joka ottaa parametrinaan olion/kokoelman ja muodostaa uuden samansisältöisen olion/kokoelman).
  - $X.equals(Y)$ vertaa (viitattujen) olioiden samanlaisuutta.
  - Muuttamattomat (*immutable*) perustyyppien oliot (*Integer*, *String*, jne) ja niiden optimoinnit (*Integer* 0..255) hämäävät, ole huolellinen.
    - (new Integer(300)) != (new Integer(300)) (eri olio(viittaus))
    - (new Integer(300)).equals(new Integer(300)) (sisältö samanlainen, siis sama luku)
    - (new Integer(42)) == (new Integer(42)) (sama valmis olio virtuaalikoneessa)
    - (new String("abc")) != (new String("abc")) (eri merkkijono-olio)
    - (new String("abc")).equals(new String("abc")) (samanlaiset merkkijonot)

Esimerkki 2.1. Listatyypin liittymä voisi alkaa tekstillä:

```
public class EsimList<E> {                                          1
```

- Tässä E kuvaa alkioiden tyyppiä, jota liittymä ei siis kiinnitä.
  - Muodollinen tyyppiparametri (*formal type parameter*), kuten muodollinen parametri aliohjelmissa.
- Listoja luotaessa parametrityyppi sitten kiinnitetään:

```
EsimList<Integer> lukuLista = new EsimList<Integer>();              1
EsimList<EsimList<Integer>> listaLista = new EsimList<EsimList<Integer>>();  2
EsimList<Henkilo> hloLista = new EsimList<Henkilo>();               3
```

- Tarkkaan ottaen **new**:n jälkeisen luokan tyyppiparametrointi (yllä *harmaalla*) ei ole tarpeellinen sillä Javassa kokoelma ei sisäisesti ole varsinaisesti parametroitu, se on vain viittaus käännösaikana.

  - Mutta tätä ei tarvitse vielä huomioida.
  - Tyyppiparametrointi molemmissa ehkä kuitekin selkeyttää koodia (tai sitten ei).

Jollemme ole lainkaan kiinnostuneita alkioiden tyypistä, voimme luoda myös "tyypittömän" kokoelman.

- Tällöin alkiotyyppi on *Object* (eli viittaus mielivaltaiseen olioon):

```
EsimList olioLista = new EsimList();
```

- Tämä sitten rajoittaa käyttöä sikäli, että kun alkioista tiedetään vain se, että ne ovat *Object*-luokan ilmentymiä, niin voimme käyttää niille vain *Object*-luokan metodeja, eli lähinnä sijoitusta ja *.equals*() -metodia. Toki nekin joskus riittävät.

Luokkien lisäksi Javassa voidaan tyyppiparametroida myös yksittäisiä metodeja.

```
static <E> void teeJotain(List<E> L) {
    E apu = L.get(0);
    ...
```

- Tyypiparametroidussa metodissa voimme antaa kääntäjälle mahdollisuuden tehdä kunnolliset tyyppitarkastukset sen sijaan, että käyttäisimme pelkkiä *Object*-olioita.
- Lisäksi jos/kun määrittelemme/rajoitamme mitä luokkaa käsittelemme, voimme hyödyntää kyseisen luokan (yleensä rajapinnan) metodeja:
- Tyyppiparametroinnissa voimme rajoittaa tarvittaessa mahdollisten alkioiden tyyppiä:

```
static void vertailua(List<Comparable> L) { ...
static void vertailua(List<? extends Comparable> L) { ...
```

Ylläolevat esittelyt näyttävät loogisilta, mutta kääntäjä ei niistä pidä sillä myös *Comparable* on myös parametroitu tyyppi jolle pitää kertoa mitä vastaan vertailut tehdään.

- Virheistä ja varoituksista päästään eroon muodolla:

```
static <E extends Comparable<? super E>> void vertailua(List<E> L) { ...
```

- Suomeksi: *Comparable* on myös parametroitu tyyppi, joten se on parametroitava jollakin E:n yläluokalla jota vastaan vertailut voidaan tehdä.
- Tähän palaamme kurssin lopussa. Toistaiseksi voimme käyttää vakiomuotoisia listojakin vaikka kääntäjä niistä hieman varoittaakin.

Lista abstraktina tietotyyppinä: s. 24.

2.1 Lista

Lista abstraktina tietotyyppinä

Esimerkki 2.2. Lista voidaan esittää vaikkapa luettelemalla listaan sisältyvät alkiot peräkkäin. Viisialkioinen lista L kuvataan esimerkiksi seuraavalla tavalla:

$$L = a_1, a_2, a_3, a_4, a_5. \tag{2.1}$$

Tyhjä lista esitetään tällöin muodossa

$$L = . \tag{2.2}$$

Listan esitysmuoto valitaan aina tilanteen mukaisesti. Tyhjä lista voidaan esittää myös käyttäen jotakin erikoissymbolia, kuten λ tai $\perp$.

Alkioiden peräkkäisyyssuhde.

- Alkioiden välillä vallitsee yksikäsitteinen edeltäjä-seuraaja -suhde.
  - Alkio a_2 edeltää alkiota a_3.
  - Alkio a_3 seuraa alkiota a_2.
  - Listan ensimmäisellä alkiolla ei ole edeltäjää.
  - Listan viimeisellä alkiolla ei ole seuraajaa.

⇒ Käsitellessämme listassa olevia alkioita, meidän on pystyttävä yksilöimään kukin alkio ja sen sijainti listassa tavalla tai toisella.

Sijainti eli asema

- Listan i:nnen alkion etäisyys listan alusta on $i-1$.
  - Jos käytetään tavallista ohjelmointikielen taulukkoa, niin sekä etäisyys alusta, että kunkin alkion sijainti ovat yksikäsitteisiä.
  - Mutta: listaan on voitava mm. lisätä keskelle alkioita, jolloin uuden alkion seuraajien etäisyys alusta muuttuu!

⇒ Tarvitaan hieman abstraktimpi sijainnin käsite.

- Tai hyväksyttävä alkioiden aseman muuttuminen!

Kullakin listan alkiolla on oma asemansa (*position*) listassa.

- Listan ensimmäiseen käytössä olevaan aseman ja listan viimeisen käytössä olevan aseman jälkeiseen asemaan pystytään viittaamaan.
- Aseman tyyppi ja käytös alkioita lisättäessä ja poistettaessa riippuu listan toteutustavasta.

Esimerkki 2.3. Toteutettaessa lista suoraviivaisesti Java-taulukossa, asemana käytetään luonnollisesti taulukon indeksiä. Tällöin poistettaessa alkio asemasta i, siirretään kaikki poistettua alkiota seuranneet alkiot yhtä asemaa lähemmäs listan alkua. Vastaavasti lisättäessä alkio asemaan i, siirretään aiemmin asemassa i ollut alkio asemaan $i+1$ ja niin edelleen.

Esimerkki 2.4. Toteutettaessa lista linkitettynä listana, asemana käytetään yleensä viittausta (osoitetta, osoitinta) listasolmuun. Tällöin poistettaessa alkio asemasta i, häviää kyseinen asema i kokonaan olemasta, ja i:tä seuraavan aseman alkio vain linkitetään i:tä edeltävän aseman alkion kanssa. Vastaavasti lisättäessä uusi alkio, luodaan myös uusi asema (asema on siis viittaus eli muistiosoite).

⇒ Siis: aseman käyttäytyminen on toteutuksesta riippuvainen (valitettavasti).

java.util.Vector/ArrayList

- Taulukolla toteutettu lista, `https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html`.
- *Vector* tukee samanaikaista käsittelyä useasta säikeestä, *ArrayList* ei.
- Käytössä olevat asemat indeksoitu 0..*size*()–1.
- Toteuttaa listan operaatiot taulukon aikavaativuuksin.
  - *get*(i) vakioaikainen.
  - *set*(i, x) vakioaikainen.
  - *add*(x) ja *add*(i, x) listan loppuun vakioaikainen (samoin *remove* lopusta).
    - Tarkkaan ottaen keskimäärin vakioaikainen.

  - *add*(i, x) alkuun tai keskelle: lineaarinen aikavaativuus (samoin *remove* alusta).
- *Vector.firstIndex* == 0.
- *Vector.size*() kertoo alkioiden määrän.
- *V.add*(*V.size*()+1, x) tai *V.set*(*V.size*(), x) antaa *ArrayIndexOutOfBoundsException*.
- *setSize*() asettaa alkioiden määrän eksplisiittisesti, jos koko on enemmän kuin aiemmin, lisätään *null*:ja, jos vähemmän, niin poistetaan alkiot lopusta.
  - Tätä käytetään kun tarvitaan harva taulukko, muuten ei kovin suositeltavaa.
  - Kapasiteettia lisätään *ensureCapacity*():llä.

Kapasiteetti vs. alkioiden määrä.

- *Vector*/*ArrayList*illa on aluksi vakiokapasiteetti, esim. 10 alkiota.
  - Tyhjällä vektorilla nämä ovat kaikki laajennusvaraa.
  - Alkukapasiteetti voidaan antaa konstruktorin parametrina.
- Alkioita lisättäessä voidaan tarvita lisäkapasiteettiä.
  - Vektori varaa tilaa aluksi esim. 10:n alkion verran tilaa.
  - Lisätilanvaraus vie aikaa $O(size())$.
  - Oletusarvoisesti tilan loppuessa kapasiteetti kaksinkertaistetaan.
    * Miten vaikuttaa aikavaativuuteen?
  - Vaihtoehtoisesti käyttäjä voi määritellä lisäystilan vakiona (*capacityIncrement*).
    * Tätä ei kuitenkaan kannata käyttää: n alkion lisäys (*.add*()) vie aikaa $O(n^2/c)$, missä c on lisävarausyksikkö (*capacityIncrement*).
    * Kts. *VectorAikaEsimerkki.java*.
- Toistuvan lisävarauksen välttämiseksi voi käyttää *ensureCapacity*() -operaatiota tai antaa alkukoko konstruktorissa jos lopullinen koko on tiedossa tai siitä on hyvä arvio.

Vertaa *java.util.LinkedList* 26.

java.util.LinkedList

- *List* -rajapinta: https://docs.oracle.com/javase/8/docs/api/java/util/List.html.
- Liittymä on pääosin periytynyt *AbstractCollection* ja *AbstractList* -luokista (kuten mm. *Vector*/*ArrayList*:llakin), mutta aikavaativuudet eri toteutuksilla ovat erilaiset.
- Osa Javan vakiokirjastoa, kts. API:n dokumentaatio.
- https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html.
- Asemaa (listasolmua) ei paljasteta käyttäjälle.
- Suorat *get*(*index*), *add*(*index*, *element*), jne operaatiot ottavat parametrinaan indeksin (järjestysnumeron) listan alusta.
  - Näiden aikavaativuus on $O(|L|)$ jos/kun joutuvat etsimään indeksin!!
  - Jollei, niin esim. lisäys ja poisto ovat vakioaikaisia.
  - Juuri päinvastoin kuin *Vector*/*ArrayList* -luokalla (s. 25).

Esimerkki 2.5. Esimerkki oikeista ja vääristä tavoista käyttää linkitettyä listaa.

```
LinkedList<Integer> L = new LinkedList<Integer>();                    1
for (int i = 0; i < n; i++)          // yhteensä O(n)                 2
  L.add(x);                          // lisäys loppuun O(1)           3
                                                                       4
for (int i = 0; i < n; i++)          // yhteensä O(n)                 5
  L.add(0, x);                       // lisäys alkuun O(1)            6
                                                                       7
for (int i = 0; i < n; i++)          // yhteensä O(n^2)               8
  L.add(i/2, x);                     // lisäys kesk. (indeksin haku) O(n)  9
```

```
for (int i = 0; i < n; i++)          // yhteensä O(n^2)
    s = s + L.get(i);                // indeksin haku O(n)
```

⇒Läpikäynti $O(n^2)$???

⇒ Tehokkaaseen läpikäyntiin on käytettävä iteraattoria!

- Iteraattori toimii ”osoittimena” kuten myöhemmin käyttämämme asemamuuttuja, mutta se ”viittaa” (näennäisesti) alkioiden (listasolmujen) väliin, eikä asemaan (listasolmuun)!

Määritelmä 2.6. *java.util.ListIterator* operaatiot (tärkeimmät). (*LinkedList L*), tarkemmin https://docs.oracle.com/javase/8/docs/api/java/util/ListIterator.html.

1. ListIterator i = L.listIterator()
   Luo ja alustaa uuden iteraattorin.
2. boolean i.hasNext()
   kertoo onko seuraavaa alkiota olemassa vai ei.
3. E i.next()
   siirtää iteraattoria seuraavaan väliin ja palauttaa yli hyppäämänsä alkion.
4. E i.previous()
   siirtää iteraattoria edelliseen väliin ja palauttaa yli hyppäämänsä alkion (käytä *i.hasPrevious*() takaperin läpikäyntiin).
5. void i.add(x) lisää elementin x iteraattorin osoittamaan väliin. Uusi elementti on seuraava *previous*() jos sitä kutsutaan, ei *next*().
6. void i.remove() poistaa elementin joka viimeksi hypättiin yli *next*():llä tai *previous*():lla.

Esimerkki 2.7. Listan läpikäynti iteraattorilla ja alkuehtoisella toistolla.

```
LinkedList L;
...
ListIterator i = L.listIterator();
while (i.hasNext())                  // yhteensä O(|L|)
    ... = i.next();
```

- Kokoelmaa voi muuttaa kesken läpikäynnin vain tämän iteraattorin *add*/*remove* -operaatioilla.
  - Iteroinnin aikana ei siis saa käyttää **listan** .*add*() tai .*remove*() -operaatioita.
  - Kaksi erillistä samanaikaista (esim. sisäkkäistä) läpikäyntiä ei saa muuttaa listaa lainkaan.
- Yksinkertaisiin läpikäynteihin tämä on suhteellisen näppärä.
- *ListIterator* tukee takaperin läpikäyntiä, mahdollistaa aloituskohdan valitsemisen ja alkion vaihtamisen, kaikille kokoelmille määritelty *Iterator* ei.

Esimerkki 2.8. Alkion x kaikkien esiintymien poisto listasta L:

- (x ei saa olla *null*).

```
ListIterator i = L.listIterator();
while (i.hasNext())                  // O(n)
    if (x.equals(i.next()))
        i.remove();
```

- Jos *null* halutaan käsitellä kuten muutkin alkiot, se on erikseen huomioitava:

```
ListIterator i = L.listIterator();
while (i.hasNext()) {
    Object a = i.next();
    if ((x == null && a == null) || (x != null && x.equals(a)))
        i.remove();
}
```

- Tässä tarvitaan apumuuttuja *a* koska tarvitsemme viittausta alkioon kahdesti ja koska *.next()* etenee listassa joka kutsukerralla.

foreach -toisto

- Jos listaa ei aiota muuttaa kesken läpikäynnin, iteraattorikin on turha.
- Meille riittää saada viittaus kuhunkin alkioon vuorollaan.
- Java ≥1.5 tukee "tee kaikille alkioille" -toistoa (*foreach*):

```
for(E x : L)
    x.foo();
```

- *E* on listan *L* alkioiden tyyppi (luokka).
- Tämä on toteutettu iteraattorilla.
- Kokoelmaa ei saa (eikä voi) muuttaa kesken *foreach*-läpikäynnin!
- Riittää, että *L* toteuttaa rajapinnan *Iterable*, sen ei edes tarvitse olla kokoelma.

Iteraattorin parametrointi

- *Iterator* ja *ListIterator* ovat itseasiassa myös parametroituja luokkia (*Iterator*<*E*>) (vrt s. 23, s. 83).
  - Erityisesti *next*() ja *previous*() palauttavat luokan *E* olioita (tai *Object*:ja jollei parametrointia tehty).
  - Niinpä tyypintarkastusten vuoksi on syytä parametroida *Iterator*-tyypin muuttujat joko tunnetulla tyypillä tai metodin parametrina saadulla tyypillä.

```
static <E> void teeJotain(LinkedList<E> L) {
    ListIterator<E> li = L.listIterator();
    while (li.hasNext()) {
        E x = li.next();
        ...
} }
```

java.util.LinkedList:n huonot puolet

- Iteraattorit luontuvat vain suhteellisen yksinkertaiseen peräkkäiseen läpikäyntiin.
- Monimutkaisemmassa tapauksessa joudutaan turvautumaan tehottomampiin indekseihin.
  - Useaa iterointia käytettäessä listaa ei voi muuttaa.
- *ListIterator* tarjoaa enemmän mahdollisuuksia kuin *Iterator* (edelleen vain yhden iteraattorin puitteissa, eikä helpota samanaikaisten muutosten kieltoa).
  - Kts. esim. *LinkedListPurge.java* vrt *TraLinkedListPurge.java* ja *TraListMerge.java* vrt. *JavaListMerge.java*.

- *next*() siirtää iteraatiota aina eteenpäin, se ei tue nykyisen tai seuraavan alkion "kurkistamista" ilman etenemistä.
  - Jollemme halua edetä jokaisella askeleella, meidän on ylläpidettävä alkioita erillisissä muuttujissa ja päivitettävä niitä vain tarvittaessa.
  - Tällöin silmukan loppuminen on havaittava ja käsiteltävä erilailla.
  - Meidän on varmistettava, että myös viimeiset alkiot tulevat käsitellyksi.
  - Meidän on varauduttava tyhjään listaan.

Yhteenveto *ArrayList* ja *LinkedList* operaatioiden aikavaativuuseroista

metodi	*ArrayList*	*LinkedList*
$L.get(i)$	$O(1)$	$O(n)$
$L.set(i,\ o)$	$O(1)$	$O(n)$
$L.add(o)$	$O(1)$	$O(1)$
$L.add(0,\ o)$	$O(n)$	$O(1)$
$L.remove(i)$	$O(n)$	$O(n)$
$L.remove(0)$	$O(n)$	$O(1)$
$L.remove(L.size()–1)$	$O(1)$	$O(1)$
$L.remove(obj)$	$O(n)$	$O(n)$
$L.contains(obj)$	$O(n)$	$O(n)$
$iter.remove()$	$O(n)$	$O(1)$
$iter.next()$	$O(1)$	$O(1)$
$iter.add()$	$O(n)$	$O(1)$
$iter.set(o)$	$O(1)$	$O(1)$

List L; Object o, obj; int i; ListIterator iter; $n = L.size()$

ArrayList s. 25, *LinkedList* s. 26.

Abstraktin listan operaatiot (TRA-kirjasto)

- Tällä kurssilla esittelemme Javan vakiolistan lisäksi listan jossa asemat (listasolmut) ovat konkreettisemmin ohjelmoijan käytettävissä.
- Sisäinen toteutus on aivan sama kuin *java.util.LinkedList*:llä, mutta liittymä on erilainen.

⇒TRA-kirjaston paketti (.jar) ja käännösohjeet Moodlessa.

Määritelmä 2.9. Asemaperustaisen listan operaatiot. Asemana käytetään viitettä listasolmuun (*ListNode*). (Parametrien tyypit: *E x*, *ListNode p*, *TraLinkedList L*)

1. TraLinkedList<E> TraLinkedList<E>()
   Muodostaa ja palauttaa uuden tyhjän listan. Alkiotyyppi E voi olla mikä tahansa luokka.

2. ListNode L.first()
   Palauttaa listan L ensimmäisen alkion aseman. Jos L on tyhjä lista, palauttaa aseman $L.EOL$.

3. ListNode L.last()
   Palauttaa listan L viimeisen alkion aseman. Jos L on tyhjä lista, palauttaa aseman $L.EOL$.

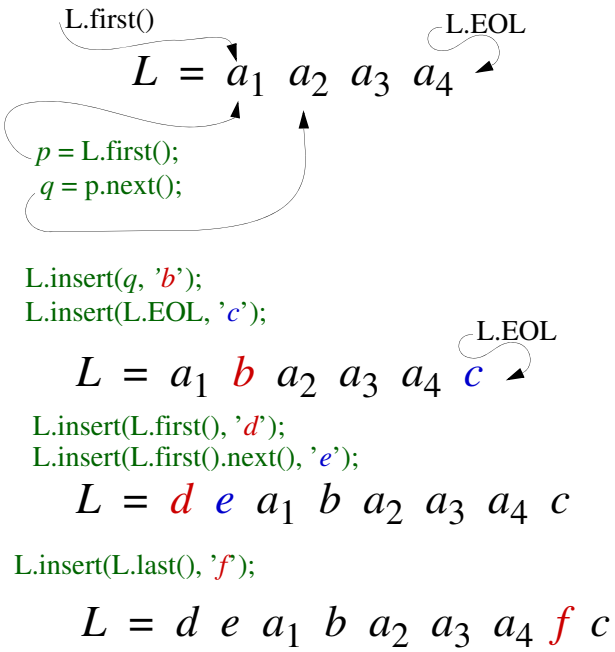
4. void L.insert(p, x)
   Lisää alkion x listaan L aseman p eteen. Jos $p = L.EOL$, kohdistuu lisäys listan loppuun. Jos taas asemaa p ei listassa L lainkaan ole, on vaikutus määrittelemätön.

5. void L.remove(p)
   Poistaa listasta L asemassa p olevan alkion. Jos $p = L.EOL$ tai asemaa p ei listassa L lainkaan ole, on vaikutus määrittelemätön.

6. ListNode p.next()
   Palauttaa asemaa p seuraavan aseman listassa. Jos p on listan viimeisen alkion asema, palauttaa aseman $L.EOL$ ja jos $p = L.EOL$ (tai listassa ei ole asemaa p) on $p.getNext$ määrittelemätön.

7. ListNode p.previous()
   Palauttaa asemaa p edeltävän aseman listassa. Jos p on listan ensimmäisen alkion asema (tai listassa ei ole asemaa p), on $p.getPrevious$ määrittelemätön.

8. E p.getElement()
   Palauttaa asemassa p olevan alkion. Jos $p = L.EOL$ on tulos määrittelemätön.



Kuva 2.1: Listan operaatiot

Esimerkki 2.10. Asemaperusteisen listan läpikäynti.

```
// TraLinkedList<E> L                                                    1
...                                                                      2
ListNode<E> p = L.first();     // ensimmäinen asema                      3
while (p != L.EOL) {           // kunnes asemaviittaus osoittaa listan loppuun  4
   E x = p.getElement();       // asemassa p oleva alkio                 5
   // operaatio alkiolle x                                               6
   p = p.next();               // asemaviittaus seuraavaan asemaan       7
}                                                                        8
```

Esimerkki 2.11. Olkoon L järjestämätön lista, jonka alkioiden tyyppiä ei tunneta. Tehtävänä on laatia algoritmi, joka poistaa listasta L kaikki saman alkion toistuvat esiintymät.

```
public static void purge(TraLinkedList L) {                             1
   ListNode p = L.first();                                               2
   while (p != L.EOL) {        // kaikkien asemien (alkioiden) läpikäynti  3
```

```
        ListNode q = p.next();      // käydään läpi asemat asemasta p eteenpäin
        while (q != L.EOL) {
            if (q.getElement().equals(p.getElement())) {      // jos löytyy sama alkio kuin asemassa p
                ListNode r = q.next();                        // niin siirretään läpikäyntikohta seuraavaan asemaan
                L.remove(q);                                  // ja poistetaan samaksi osoittautunut alkio (asema)
                q = r;
            } else
                q = q.next();
        }
        p = p.next();
}   }
```

Listaoperaatioiden aikavaativuus

- Operaatioiden *first*, *next* ja *getElement* sekä listan luonnin tulisi aina olla $O(1)$ (vakioaikaisia) jotta läpikäynti olisi tehokasta.
- *insert*, *remove*, *previous*, *last* ja *EOL* on mahdollista (jopa helppoa) tehdä vakioaikaisiksi, joten jos/kun niitä tarvitaan, niin ne on syytä tehdä vakioaikaisiksi.

Esimerkki 2.12. Esimerkin 2.11 algoritmin aikavaativuuden arvioimiseksi täytyy tuntea paitsi listaoperaatioiden aikavaativuus, myös samuusfunktion aikavaativuus.

Oletetaan, että alkioiden vertailuoperaatio *equals*() on $O(1)$ ja että esimerkissä 2.11 käytetyt listaoperaatiot ovat myös $O(1)$. Silloin rivit 6 - 11 vaativat aikaa $O(1)$. Molemmissa toistoissa käydään listaa läpi alkio kerrallaan. Sisemmässä toistossa käsiteltävän listanosan pituus tosin lyhenee kierros kierrokselta, koska listan alkuosaa ei enää tarkastella, ja lisäksi lista lyhenee aina toistuvia alkioita havaittaessa. Pahimmassa tapauksessa ei löydetä yhtään toistuvaa esiintymää, joten ulompi toisto suoritetaan $O(listan\_pituus)$ kertaa ja koko toiston aikavaativuus on $O(listan\_pituus^2)$. Jos *equals*() ei ole vakioaikainen, aikavaativuus voidaan kirjoittaa muodossa $O(T_{equals} \times listan\_pituus^2)$.

Esimerkki 2.13. Esimerkki: listojen samuuden vertailu. Listat ovat samat, jos ne ovat yhtä pitkät ja niissä on samanlaiset alkiot samassa järjestyksessä.

```
public static boolean compareLists(TraLinkedList L1, TraLinkedList L2) {
    ListNode p1 = L1.first();
    ListNode p2 = L2.first();
    while ((p1 != L1.EOL) && (p2 != L2.EOL)) {
        if (! p1.getElement().equals(p2.getElement()))
            return false;
        p1 = p1.next();
        p2 = p2.next();
    }

    if (p1 == L1.EOL && p2 == L2.EOL)
        return true;
    else
        return false;
}
```

Aikavaativuus?

(Linkitetyn) listan käyttökohteet:

- Mikä tahansa, missä kokoelmaa käydään lävitse peräkkäisjärjestyksessä.

- Erityisesti, jos kokoelmalla on jokin järjestys.
- Erityisesti, jos kokoelman kokoa ei tunneta etukäteen.
- Erityisesti, jos keskelle kokoelmaa kohdistetaan lisäys- tai poisto -operaatioita.
  - (jotka taulukossa ovat raskaita!)
- Esim: ”laske kokoelman alkoiden summa”.
  - Peräkkäisrakenne ok
- Esim: ”etsi annetun puhelinnumeron haltijan nimi”.
  - peräkkäisrakenne liian hidas etsimiseen
  - lista on huono vaihtoehto (paitsi jos etsiminen on hyvin harvinaista)

Lista on yleiskäyttöinen tietorakenne.

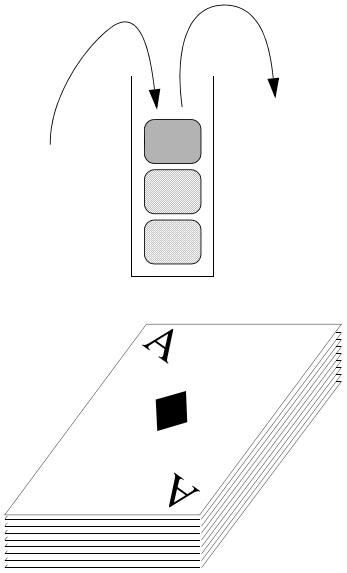
⇒ Läheskään aina ei tarvita kaikkea listan toiminnallisuutta.

- Erityisesti apurakenteina tiedon tilapäiseen tallentamiseen algoritmissa.
- Näissä tapauksissa riittää rajoittaa operaatiojoukkoa sopivaksi, yleensä saantipaikan suhteen
- Hyöty: saadaan huomattavasti yksinkertaisemmat operaatiot.
  - helppolukuisempi ohjelma.
  - vähemmän virhemahdollisuuksia.
  - helpompi toteuttaa.
  - yksikäsitteisempi.

Seuraavassa listan rajoitettuja muotoja.

2.2 Pino (*stack*)

- Lisäys- ja poisto-operaatiot sallitaan vain kokoelman toiseen päähän.
  - Aktiivinen pää == pinon pinta (*top*).
  - Toinen pää == pinon pohja (*bottom*).
  - Pinon alkioista vain päällimmäinen on saatavilla.
  - Pinon sisällöstä voidaan havainnoida vain päällimmäinen.
    - Edes pinon alkioiden lukumäärää ei yleensä tiedetä (paitsi tyhjä — epätyhjä), ellei sitä lukumäärää itse ylläpidetä pinoa käytettäessä.
- Viimeinen lisätty on ensimmäisenä vuorossa poistettaessa: (LIFO, Last In First Out).
- Pino sopii erityisesti apurakenteeksi tilanteisiin jossa osatehtäviä/syötettä/tms laitetaan ”syrjään” hetkeksi kun tehdään jotain muuta.
  - Myöhemmin sitten otetaan käsittelyyn näitä ”syrjään” laitettuja.



- Koska saantia rajoitetaan, eivät läheskään kaikki listan operaatiot ole mielekkäitä (eivätkä tarpeellisia, käteviä, jne.).

⇒ *insert*, *getElement*, *remove* ja pinon luonti riittävät.

- Näissäkin esim. lisäyskohta on aina ilmeinen.
  - Ei tarvita aseman käsitettä lainkaan (eikä vastaavia parametreja operaatioissa).
- Lisäksi tarvitaan keino selvittää onko pino tyhjä.

- Käsittelyn selkeyttämiseksi vaihdetaan vielä operaatioiden nimet.

Määritelmä 2.14. Pino-operaatiot määritellään perinteisesti seuraavien, yleisen listan operaatioiden nimiin nähden pinon luonteeseen paremmin sopivien nimien mukaisiksi: (*LinkedStack* S, E x)

1. LinkedStack<E> LinkedStack<E>()
   muodostaa tyhjän pinon.
2. void S.push(x)
   vie pinoon S päällimmäiseksi alkion x. [insert]
3. E S.peek()
   palauttaa pinon S päällimmäisen alkion arvon. Jos pino on tyhjä, antaa poikkeuksen. [getElement]
4. E S.pop()
   poistaa pinosta S päällimmäisen alkion. Jos pino on tyhjä, antaa poikkeuksen. [remove]
5. boolean S.isEmpty()
   palauttaa boolen arvon *true*, mikäli pino S on tyhjä, muuten palauttaa arvon *false*.

$\Rightarrow$ Kaikkien operaatioiden aikavaativuuden tulisi olla $O(1)$.

- koneenläheisessä toteutuksessa vain 1-3 kellojaksoa, pinokoneessa jopa 0-1 kellojaksoa.

Esimerkki 2.15. Sulkujen tarkastus:

- Merkkijono sisältää merkkejä, joiden joukossa on sulkeita ( ) [ ] { } .
- On selvitettävä ovatko sulkeet oikeassa järjestyksessä
  - Vastinsulut ovat samaa tyyppiä (ei ”( ]”).
  - Sulut eivät ole limittäin (ei ”([ )]”).
  - Sulkua ei suljeta ennen alkamista (ei ”) (”).
  - Kaikki sulut suljetaan ennen listan loppumista (ei ”([ ]”).
- Täydellisempi koodi Moodlessa.

```
public static boolean sulkuParit(String m) {                                  1
   LinkedStack<String> S = new LinkedStack<String>();                          2
   for (int i = 0; i < m.length(); i++) {                                      3
      char c = m.charAt(i);                                                    4
      switch (c) {                                                             5
         case '(' : S.push(")"); break;                                        6
         case '[' : S.push("]"); break;                                        7
         case '{' : S.push("}"); break;                                        8
         case ')' :                                                            9
         case ']' :                                                            10
         case '}' :  if (S.isEmpty() || ! m.substring(i, i+1).equals(S.pop()))  11
                              return false;                                    12
   }  }                                                                        13
   return S.isEmpty();                                                         14
}                                                                              15
```

Kun merkkijono- ja pino-operaatiot ovat tehokkaita, algoritmin aikavaativuus on O(merkkijonon_pituus) ja tilavaativuus O(sulkujen_syvyys).

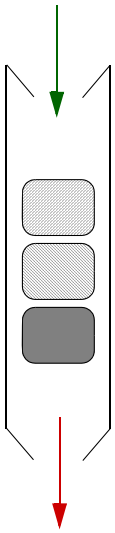
java.util.LinkedList, *java.util.Vector*, jne.

- *LinkedList* kummin päin vain, *Vector*/*ArrayList* lopusta käytettynä.
  - *add*(), *getLast*(), *removeLast*(), *isEmpty*().
- *Vector*:n laajennus *java.util.Stack* (https://docs.oracle.com/javase/8/docs/api/java/util/Stack.html).
  - *push*(), *peek*(), *pop*(), *isEmpty*().
- *java.util.ArrayDeque* (https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html).
  - Pakan, pinon ja jonon operaatiot yhdessä kokoelmassa, kts. s. 37.

Avopino

- Avopino on muuten kuten pino, mutta pinon aivan päällimmäisen alkion lisäksi (muutamia) seuraavia voidaan "kaivaa" esiin operaatiolla
  - $S.top(k)$
    Palauttaa avopinon S pinnalta lukien k:nnen alkion arvon.
    - Aikavaativuus toteutuksesta riippuen $O(k)$ tai $O(1)$.
- Huomaa, ettei muita kuin päällimmäistä alkiota kuitenkaan voi muuttaa tai ainakaan poistaa (ellei näin välttämättä haluta määritellä).
- *java.util.Vector* toimii avopinona mainiosti (pinon pinta on taulukon lopussa joten kurkkaus on mallia $S.get(S.size()–k)$.

2.3 Jono (*queue*)



- Lisäys-operaatio sallitaan vain kokoelman yhteen päähän ja poisto-operaatio vain siihen toiseen päähän.
  - Lisäyspää == jonon häntä.
  - Poistopää == jonon keula.
  - Jonon alkioista vain etummainen voidaan poistaa.
  - Jonon sisällöstä voidaan havainnoida vain etummainen.
    - Edes jonon alkioiden lukumäärää ei yleensä tiedetä (paitsi tyhjä — epätyhjä), ellei tietoa jonoa käytettäessä itse ylläpidetä.
- Ensimmäisenä lisätty on ensimmäisenä vuorossa poistettaessa: FIFO (First In First Out).
- Koska saantia rajoitetaan, eivät läheskään kaikki listan operaatiot ole mielekkäitä:
  - *insert*, *getElement*, *remove* ja jonon luonti riittävät.
    - Näissäkin käsittelykohta on aina ilmeinen eli sitä ei tarvita parametrina (lisäys/poisto).
    - Ei tarvita aseman käsitettä lainkaan (eikä vastaavia parametreja operaatioissa).
  - Lisäksi tarvitaan keino selvittää onko jono tyhjä.
  - Luettavuuden parantamiseksi vaihdamme myös operaatioiden nimet.

Esimerkki 2.16. Kaupan kassajono on tyypillinen jono: uudet jonottajat asettuvat aina jonon loppuun ja jonottajia palvellaan saapumisjärjestyksessä.

Tietojärjestelmissä jonorakenteita käytetään tyypillisesti esimerkiksi tilapäistalletuksen apurakenteina ja eri komponenttien väliseen kommunikaatioon (puskureina).

- Jonon avulla komponenttien välinen kommunikaatio voidaan tehdä synkroniseksi (kohtaaminen) tai asynkroniseksi (puskurointi).
- Kommunikoivia komponentteja voi olla monella tasolla, eri tietojärjestelmissä (hajautettu järjestelmä) tai ohjelman metodien tai säikeiden välillä.

Määritelmä 2.17. Jono-operaatiot TR-kirjastossa (suluissa perinteinen nimeäminen): (*LinkedQueue* Q, $E\ x$)

1. LinkedQueue<E> LinkedQueue<E>()
   muodostaa tyhjän jonon Q.
2. void Q.offer(x)
   vie jonon Q loppuun alkion x. [enqueue]
3. E Q.peek()
   palauttaa jonon Q ensimmäisen alkion arvon. Jos jono on tyhjä, antaa poikkeuksen. [front]
4. E Q.poll()
   poistaa ja palauttaa jonosta Q ensimmäisen alkion. Tyhjään jonoon kohdistettaessa antaa poikkeuksen. [dequeue]
5. boolean Q.isEmpty()
   palauttaa arvon *true*, mikäli jono Q on tyhjä, muuten palauttaa arvon *false*.

$\Rightarrow$ Kaikkien operaatioiden aikavaativuus tulisi olla $O(1)$.
$\Rightarrow$ *java.util.LinkedList* toimii hyvin jonona.
$\Rightarrow$ *ArrayList*/*Vector* eivät toimi sellaisenaan (lisäys tai poisto alusta on $O(n)$)!

- Taulukoilla voidaan toki toteuttaa jono tehokkaasti ns. rengaspuskurilla, kts. ”Jono ja pakka taulukossa” s. 92.
- Java API:ssa *java.util.Queue*:n toteutukset (Java SE1.6-: *ArrayDeque*).
  - https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html
  - Lue johdantoteksti huolella ja valitse tarpeeseesi oikea toteutus ja sen oikeat operaatiot!
    - Operaatioista on erilaiset versiot sen mukaan mitä tapahtuu erikoistilanteissa kun jono on tyhjä tai täysi.
    - Rajapinta sallii myös muun kuin FIFO-järjestyksen!
    - Eri toteutuksilla on erilaiset rajoitteet (ja jopa hieman erilaiset käytökset).

Esimerkki 2.18. Pinon sisältö voidaan kääntää jonon avulla ja päinvastoin:

```
public static void reverse(LinkedStack S) {
    LinkedQueue Q = new LinkedQueue();
    while (! S.isEmpty())
        Q.offer(S.pop());
    while (! Q.isEmpty())
        S.push(Q.poll());
}

public static void reverse(LinkedQueue Q) {
    LinkedStack S = new LinkedStack();
    while (! Q.isEmpty())
        S.push(Q.poll());
    while (! S.isEmpty())
        Q.offer(S.pop());
}
```

- Aika- ja tilavaativuus $O(|S|)$.
- Pinon kääntäminen ja käännetyn pinon takaisin alkupinoon saattaminen ei onnistu yhtä apupinoa käyttäen, tarvitaan jono tai kaksi pinoa.

Esimerkki: rekursion korvaaminen

⇒ Rekursiivinen: Algoritmi, joka kutsuu itseään sopivalla syötteen osalla / osilla yksinkertaistaakseen koko tehtävän ratkaisua.

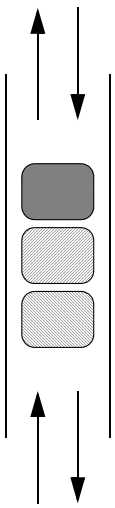
- Rekursio voi olla hieman (vakiokertoimella) hitaampi ratkaisu tai käytettävä ohjelmointikieli ei tue rekursiota.
- Joskus ongelman vaatima käsittelyjärjestys ei salli suoraa rekursiota tai se ei ole paras mahdollinen.
  - Ohjelmointikielten rekursio toimii aina pinolla, eli on LIFO.
  - Jos tarvitaan jokin muu käsittelyjärjestys, on rekursio korvattava omalla rakenteella.
- Hyvin syvä rekursio kuluttaa paljon muistia (pinoa).
- Yhden rekursiivisen kutsun voi yleensä korvata suoraviivaisella toistolla, mutta usean rekursiivisen haaran ylläpitäminen vaatii jonkin tietorakenteen säilyttämään kaikki odottavat ”rekursiokutsut” (tekemättömät osatehtävät).

Kuinka korvata rekursio?

- Ylläpidetään tekemättömien osatehtävien pinoa tai jonoa.
  - Suorituksen alussa laitetaan alkuperäinen tehtävä pinoon/jonoon.
  - Toistetaan algoritmin runkoa kunnes pino/jono on tyhjä.
    - Rekursiivisten kutsujen sijaan työnnetään kunkin uuden osatehtävän tiedot pinoon/-jonoon.
- Osatehtävien lopullinen läpikäyntijärjestys riippuu siitä, käytetäänkö pinoa, jonoa, prioriteettijonoa vai vaikkapa joukkoa.
  - Jono: ”leveyssuuntainen” läpikäynti.
  - Pino: ”syvyyssuuntainen” läpikäynti.
  - Prioriteettijono: ”tärkein/paras ensin” läpikäynti.
- Samaa tekniikkaa voi käyttää kaikissa tilanteissa, joissa ongelman ratkaisu haarautuu useampaan osaan.
- Palataan tähän puun (luku 3) ja verkon (TRA II) käsittelyn yhteydessä.

2.4 Pakka (*deque*)

- Sekä lisäys- että poisto-operaatio sallitaan kokoelman molempiin päihin, mutta vain päihin.
  - Päät on voitava erottaa toisistaan:
    - pinta (päällimmäinen, etummainen, ensimmäinen)
    - pohja (alimmainen, takimmainen, viimeinen)
  - Pakan alkioista päällimmäinen ja alimmainen ovat poistettavissa.
  - Pakan sisällöstä voidaan havainnoida päällimmäinen ja alimmainen.
    - Edes pakan alkioiden lukumäärää ei yleensä tiedetä (paitsi tyhjä — epätyhjä), ellei sitä pakkaa käytettäessä itse lasketa.
  - Jos pakassa on vain yksi alkio, se on sekä pinnalla, että pohjalla.



- Koska saantia rajoitetaan, eivät läheskään kaikki listan operaatiot ole mielekkäitä.
  - *insert*, *getElement*, *remove* ja pakan luonti riittävät.
  - Lisäksi tarvitaan keino selvittää onko pakka tyhjä.

⇒ Pinosta ja jonosta poiketen tarvitaan keino ilmaista käsittelykohta.

- Aseman käsite on tarpeettoman hankala kun mahdollisia paikkoja on vain kaksi.
- Operaatio kohdistuu joko pinnalle tai pohjalle.
- Voimme joko käyttää erillistä kaksiarvoista parametria, tai käyttää eri operaatioita eri päihin.
- Luettavuuden parantamiseksi, nimetään operaatiot kuvaavasti.

Esimerkki 2.19. Pakkatyypin luonnollinen ilmentymä on kädessä pidettävä korttipakka, josta sekä päällimmäinen, että alimmainen kortti voidaan poistaa. Pakkaan vietävä kortti voidaan myös asettaa joko päällimmäiseksi tai alimmaiseksi.

Määritelmä 2.20. Pakan operaatiot (samat kuin *java.util.Deque*, *java.util.LinkedList*): (*LinkedDeque D*, *E x*)

1. LinkedDeque<E> LinkedDeque<E>()
   muodostaa tyhjän pakan.

2. void D.addFirst(x)
   void D.addLast(x)
   lisää pakan *D* alkuun / loppuun alkion *x*.

3. E D.removeFirst()
   E D.removeLast()
   poistaa ja palauttaa pakasta *D* ensimmäisen / viimeisen alkion. Tyhjään pakkaan kohdistettaessa antaa poikkeuksen.

4. E D.getFirst()
   E D.getLast()
   palauttaa pakan *D* ensimmäisen / viimeisen alkion arvon. Jos pakka on tyhjä, antaa poikkeuksen.

5. boolean D.isEmpty()
   palauttaa boolen arvon *true*, mikäli pakka *D* on tyhjä, muuten palauttaa arvon *false*.

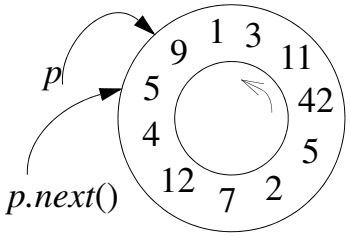
Usein operaatiot nimetään kuten jonon operaatiot. Erillisten ... *First* ja ... *Last* -operaatioiden sijaan voidaan yhtä hyvin esitellä vain yhdet operaatiot ja käyttää lisäparametria joka osoittaa kumpaan päähän operaation halutaan kohdistuvan.

⇒ Kaikkien operaatioiden aikavaativuuden tulisi olla $O(1)$.

java.util.Deque

- Rajapinta https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html
- Taulukkototeutus https://docs.oracle.com/javase/8/docs/api/java/util/ArrayDeque.html
- Sekä linkitetty toteutus, että taulukkototeutus ("kehäpuskuri", kts. s. 92).
  - Huomaa, että *ArrayList*/*Vector* eivät näy toteutuksina, eivätkä ole lainkaan tehokkaita!
- Lisäksi spesiaaliversioita samanaikaisohjelmoinnin käyttöön.

2.5 Rengas (*ring*)



- Joskus listan päillä ei ole merkitystä, vaan käymme läpi kokoelman alkioita jossakin järjestyksessä toistuvasti.
  - *last*:sta *first*:een hyppääminen joutaa jäämään ADT:n tehtäväksi.
- Lisäysten ja poistojen pitää sujua vaivattomasti.
  - Edes lisäysasemalla ei välttämättä ole merkitystä jos järjestykseen ei haluta vaikuttaa, tällöin riittää $R.next()$ ja $R.add()$.
  - Rengas voi olla yksi- tai kaksisuuntainen.
- Asetetaan
  - $R.last().next() = R.first()$ ja $R.first().previous() = R.last()$,
- Saadaan *rengas*, päättymätön lista.
- Listan operaatioista *first*:in sijaan *any* joka palauttaa jonkin aseman.
- *EOL*:n sijaan *isEmpty* (kuten esim. pakassa).
- Operaatiokokoelma voi olla hyvin yksinkertainen jos järjestyksellä ei ole väliä.

⇒ Asema on oltava jossain muodossa käytössä läpikäyntien mahdollistamiseksi (mahdollisesti iteraattoriin piilotettuna).

2.6 Taulukko

⇒ Lista ei aina tue alkion muuttamista, eikä hakemista suoraan (vakioajassa) sijainnin (indeksin) perusteella.

- Alkion muuttaminen listassa vaatii yleensä poiston ja lisäyksen.
- Alkioiden käsittely mielivaltaisessa järjestyksessä (hajasaanti) on mahdotonta (tai raskasta).

Jos sekä peräkkäiskäsittely, että mielivaltaisessa järjestyksessä käsittely ovat tärkeitä, tarvitaan (abstraktia) taulukkotyyppiä.

- Taulukon alkiot ovat keskenään peräkkäissuhteessa (≈lista).
- Taulukon alkion asema selviää indeksoimalla (ts. (suhteellinen) etäisyys taulukon alusta).
  - Indeksit ovat kokonaislukutyyppiä (tai lueteltua tyyppiä), niillä voidaan jopa laskea.

Taulukon operaatiot kuten listalla:

- Asemana käytetään indeksiä taulukon alusta.
- Taulukon luonnissa tosin on syytä antaa taulukon alkumitat.
- Koska lisääminen on taulukossa raskasta, taulukon peruspäivitysoperaatio onkin indeksiin päälle vienti:
  - $A.set(i,\ x)$
- Joskus tarvitaan myös
  - $A.force(i,\ x)$
- joka toimii kuten $set()$, mutta tarvittaessa laajentaa taulukkoa siten, että uusikin indeksi i on olemassa.
- Jotta voisimme viitata taulukon rajoihin, tarvitaan
  - $A.first()$ ja $A.last()$

Operaatioiden $set()$, $first()$, $last()$, $getElement()$ ja taulukon luonti aikavaativuus on $O(1)$. $next()$ ja $previous()$ ovat vain indekseillä laskemista.

- Jos kuitenkin taulukon luonnissa taulukko alustetaan (kuten Javassa) se vie lineaarisen ajan.

Operaatioiden *insert*(), *remove*() ja *force*() aikavaativuus on O(taulukon_koko), tai O(taulukon_loppuosan_pituus).

- Jos lisäystä tai poistoa keskelle/keskeltä tarvitaan, ei taulukko yleensä ole hyvä vaihtoehto.

Avoin/kiinteänkokoinen taulukko

- Avoimessa taulukossa lisäykset/poistot laajentavat/pienentävät taulukkoa yhden alkion verran, force laajentaa tarpeen mukaan.
- Taulukko voi olla avoin joko molemmista päistä tai vain toisesta päästä.
- Kiinteäkokoisessa taulukossa *insert* ja *remove* on yleensä jätetty pois.
  - Jos em. operaatiot sallitaan kiinteäkokoisessa taulukossa, kyseessä on hylly.
    - Hyllyssä lisäys keskelle tiputtaa alkion laidalta pois, poisto keskeltä lisää tyhjän paikan laidalle.

```
Object[ ]      A1;        // täysin avoin          1
Object[1..]    A2;        // ylhäältä avoin        2
Object[..10]   A3;        // alhaalta avoin        3
Object[1..10]  A4;        // kiinteä koko          4
```

  - Huomaa *first* ja *last* operaatioiden tarpeellisuus. Javassa alhaalta avoin taulukko ei suoraan onnistu. Rajoituksen voi kiertää kuvauksella (*Map*) tai käyttämällä omaa indeksointia, esim:

```
A[abs(i)*2 + ((i < 0) ? 1 : 0)]                   1
```

java.util.Vector/ArrayList

- Taulukolla toteutettu lista, https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html.
- *Vector* tukee samanaikaista käsittelyä useasta säikeestä, *ArrayList* ei.
- Käytössä olevat asemat indeksoitu 0..*size*()–1.
- Toteuttaa listan operaatiot taulukon aikavaativuuksin.
  - *get*(i) vakioaikainen.
  - *set*(i, x) vakioaikainen.
  - *add*(x) ja *add*(i, x) listan loppuun vakioaikainen (samoin *remove* lopusta).
    - Tarkkaan ottaen keskimäärin vakioaikainen.
  - *add*(i, x) alkuun tai keskelle: lineaarinen aikavaativuus (samoin *remove* alusta).
- *Vector.firstIndex* == 0.
- *Vector.size*() kertoo alkioiden määrän.
- *V.add*(*V.size*()+1, x) tai *V.set*(*V.size*(), x) antaa *ArrayIndexOutOfBoundsException*.
- *setSize*() asettaa alkioiden määrän eksplisiittisesti, jos koko on enemmän kuin aiemmin, lisätään *null*:ja, jos vähemmän, niin poistetaan alkiot lopusta.
  - Tätä käytetään kun tarvitaan harva taulukko, muuten ei kovin suositeltavaa.
  - Kapasiteettia lisätään *ensureCapacity*():llä.

Kapasiteetti vs. alkioiden määrä.

- *Vector/ArrayList*illa on aluksi vakiokapasiteetti, esim. 10 alkiota.
  - Tyhjällä vektorilla nämä ovat kaikki laajennusvaraa.
  - Alkukapasiteetti voidaan antaa konstruktorin parametrina.
- Alkioita lisättäessä voidaan tarvita lisäkapasiteettiä.
  - Vektori varaa tilaa aluksi esim. 10:n alkion verran tilaa.

  - Lisätilanvaraus vie aikaa $O(size())$.
  - Oletusarvoisesti tilan loppuessa kapasiteetti kaksinkertaistetaan.
    * Miten vaikuttaa aikavaativuuteen?
  - Vaihtoehtoisesti käyttäjä voi määritellä lisäystilan vakiona (*capacityIncrement*).
    * Tätä ei kuitenkaan kannata käyttää: n alkion lisäys ($.add()$) vie aikaa $O(n^2/c)$, missä c on lisävarausyksikkö (*capacityIncrement*).
    * Kts. *VectorAikaEsimerkki.java*.
- Toistuvan lisävarauksen välttämiseksi voi käyttää *ensureCapacity*() -operaatiota tai antaa alkukoko konstruktorissa jos lopullinen koko on tiedossa tai siitä on hyvä arvio.

Vertaa *java.util.LinkedList* 26.

Ohjelmointikielten taulukot

- Kaikissa lausekielissä on taulukkotyyppi, useimmissa myös moniulotteinen taulukko.
- Vanhemmat kielet eivät tue dynaamista taulukon kokoa (Pascal, C83)
  - Aliohjelman paikallisen taulukon koko parametrin tai muun lasketun arvon mukaan.
- Olemassaolevan taulukon koon muuttamista eivät tue juuri mitkään kielet.
- Yleisemmillä dynaamisilla tilanvarausoperaatioilla nämä ongelmat voidaan kiertää niillä kielillä joissa on dynaaminen tilanvaraus tavuperusteisesti.
- Luontia, sijoitusta ja lukemista lukuunottamatta, muut abstraktin taulukon operaatiot joudutaan tekemään itse jos niitä tarvitaan.
  - Operaatioita *insert*, *remove* ja *force* ei yleensä tarvita/tehdä (käytetään mieluummin listoja, puita tai hajautusta jos tarvitaan).
- Käytännössä ohjelmointikielten taulukot ovat meille yleensä aivan riittäviä ja erityisesti tehokkaita.
  - Niinpä mm. tietorakennekirjastomme ei sisällä erillistä abstraktia taulukkotyyppiä.

2.7 Yhteenveto peräkkäisrakenteista

- Taulukkoa lukuunottamatta kaikki tässä luvussa esitetyt tyypit soveltuvat parhaiten peräkkäiskäsittelyyn.
  - Linkitettyä listaa käytetään iteraattoreilla (tai listasolmuilla tai pino/jono/pakka operaatioilla) – ei indekseillä.
  - Taulukkorakenteita käytetään indekseillä – ei lisäyksiä tai poistoja muualle kuin loppuun.
- Näiden kaikkien kokoelmien käyttö on helppoa ja tehokasta kun niitä käytetään oikein.

⇒ Muuttuvasta kokoelmasta hakemiseen tarvitaan parempia kokoelmia.

- ”muuttuva” : ei taulukko
- ”hakeminen” : ei lista

Esimerkki 2.21. Listatyypit (taulukkoa lukuunottamatta) ovat keskenään hyvin lähisukulaisia. Yleisen listan päälle voidaan toteuttaa muut varsin helposti ja suoraviivaisesti.

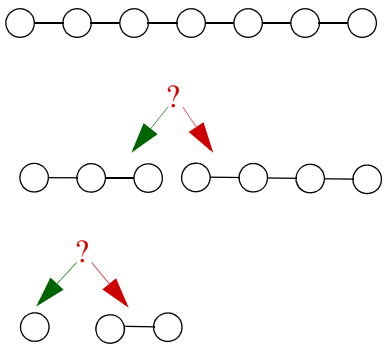
Seuraavassa esimerkkinä pino-operaatioiden toteutus listaoperaatioita käyttäen. Operaatioista voitaisiin kokonaan poistaa yläluokalta sellaisenaan perityt *Stack* -konstruktori ja *isEmpty*. Samoin *super* -yläluokkaviittaukset ovat tarpeettomia, mutta ne ehkä parantavat koodin luettavuutta.

```
public class Stack<E> extends java.util.LinkedList<E> {
  public Stack()
    super();
  }
```

```
    public void push(E x) {
        super.addFirst(x);
    }
    public E pop() {
        return super.removeFirst();
    }
    public E top() {
        return super.getFirst();
    }
    public boolean isEmpty()
        return super.isEmpty();
    }
}
```

3 Puut

Listan/taulukon ongelmat talletusrakenteena:

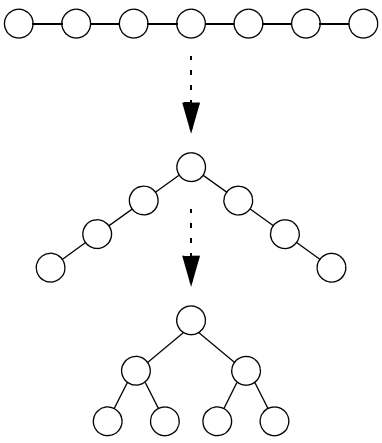


- Lista on pitkä, joten siitä etsiminen on hidasta.
- Järjestetystä taulukosta etsiminen on nopeaa, mutta lisääminen on hidasta.

⇒ Idea: pistetään ”järjestetty lista puoliksi”!

- Oikeasti linkitettyä listaa ei niin vain pistetä puoliksi, mutta ajatellaan aluksi näin.
- Ensin selvitetään kummassa puoliskossa alkio on ja haetaan vain sieltä → puolet vähemmän läpikäytävää.
- Pistetään nämäkin puoliskot puoliksi → taas puolet vähemmän läpikäytävää.
- log$n$ kertaa toistettuna vain nämä log$n$ päätöstä, ei lainkaan peräkkäistä selattavaa.
- Tämä oli siis puolitushaku (eli binäärihaku) (joka toimii järjestetyllä taulukolla, mutta ei onnistu listalla vaikka se olisi järjestetty).
  - Järjestettyyn taulukkoon lisääminen taas on hidasta!

Miten tämä on toteutettavissa kokoelmana?



⇒ Haaroitetaan lista:

⇒ Listan edeltäjä — seuraaja -käsitteen sijaan tulee puun edeltäjä — 2 seuraajaa.

⇒ Puu antaa kokoelman alkioille hierarkkisen järjestyksen (vrt. listan peräkkäinen järjestys).

- Puurakenne soveltuu paitsi hakemiseen, myös hierarkkisen tiedon esittämiseen.
- Hierarkiassa on voitava esittää myös malli yksi edeltäjä, monta seuraajaa.

⇒ Erotetaan kaksi vaihtoehtoa: kaksi seuraajaa (binääripuu (*binary tree*)) ja nolla tai useampi (monta) seuraajaa (yleinen puu).

- Binääripuu sopii hakurakenteeksi.
- Yleinen puu sopii hierarkkisen tiedon esittämiseen.

3.1 Puiden peruskäsitteistö

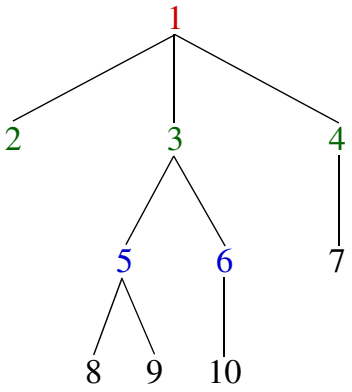
- Puu koostuu solmuista (*node*) joiden kesken vallitsee hierarkia.
  - Isä (vanhempi) (*parent*) -relaatio määrää hierarkisen suhteen.
    - Sukupuolineutraali ”vanhempi” -nimitys on ehkä hieman huonompi, sillä puussa yläpuolella hierarkiassa oleva solmu on yksikäsitteinen.
  - Epätyhjässä puussa on yksi solmu, ns. juuri (*root*), jolla ei ole isää.
  - Puun kaikilla muilla solmuilla on yksiymmärteinen isä.
- Puu voidaan määritellä myös rekursiivisesti:

Määritelmä 3.1. Yksittäinen solmu muodostaa puun, jonka juuri on puun ainoa solmu.
Jos n on solmu ja T_1,T_2,...,T_k ovat puita, joiden juuret ovat vastaavasti n_1,n_2,...,n_k, voidaan muodostaa uusi puu asettamalla solmu n solmujen n_1,n_2,...,n_k isäksi. Uuden puun juuri on n ja T_1,T_2,...,T_k ovat n:n alipuut. Solmuja n_1,n_2,...,n_k sanotaan solmun n lapsiksi.

Tyhjässä puussa ei ole yhtään solmua, ei edes juurta. Tyhjää puuta merkitään esimerkiksi Λ -merkillä). Joskus tyhjä puu rinnastetaan tyhjään solmuun, jota myös merkitään Λ tai $\perp$ -merkillä.

- Huomaa kuitenkin, että käytännöllisessä toteutuksessa tyhjän puun pitää olla kelvollinen puu eikä esimerkiksi *null* (jotta siihen esimerkiksi voidaan lisätä solmuja). Pelkkä *null* voisi sitten olla puu jota ei ole olemassa (jota siten ei voida käyttää).

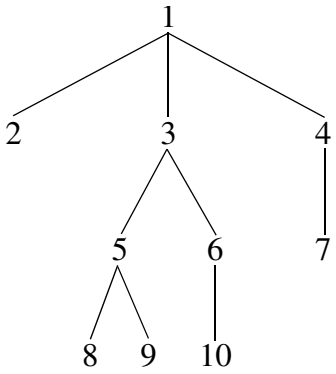
Esimerkki 3.2. Oheisen yleisen puun juuri on solmu (jossa on alkio) 1. Juuren alipuiden juurisolmut ovat 2, 3 ja 4, solmun 3 alipuiden juuret ovat 5 ja 6 ja niin edelleen. Solmu 5 on solmun 9 isä ja solmu 9 solmun 5 lapsi. Solmulla 9 ei ole lasta lainkaan, toisin sanoen solmu 9 ei ole minkään solmun isä.



Puun sisältö

- Puu ”sisältää” (ylläpitää) solmuja.
- Solmu sisältää alkion (hyötytiedon).
- Kuten listojenkin alkiot, myös puiden solmuissa säilytettävä tieto voi olla minkätyyppistä tahansa.
- Esimerkkipuissa käytämme yleensä numeroita tai kirjaimia.
- Kuten linkitetyssä listassakin, toteutusrakenteessa on erikseen kukin puusolmu ja sen sisältämä (viittaama) hyötytieto.
- Selkeyden vuoksi yleensä piirrämme vain hyötytiedon.

Polku



- Puun solmujen n_1,n_2,...,n_k jono on polku (*path*), jos solmu n_i on solmun n_{i+1} isä kaikille $i=1,...,k-1$.
  - Polun pituus on $k-1$, kaarten (isä-lapsi -suhteiden) lukumäärä.
- Polku voidaan määritellä tarvittaessa myös toiseen suuntaan alhaalta ylös.

Lehtisolmu (*leaf node*)

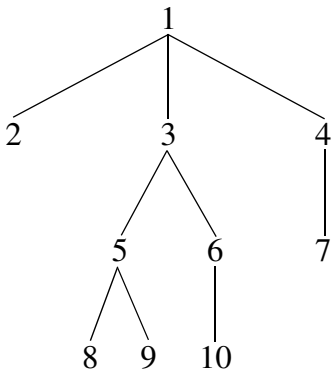
- Solmu, jolla ei ole lapsia.

Haarautumissolmu (*branching node*)

- Solmu, jolla on lapsi(a).

⇒ Jokainen solmu on joko lehtisolmu tai haarautumissolmu.

Korkeus (*height*)



- Solmun korkeus on pisimmän solmusta lehtisolmuun johtavan polun pituus.
- Puun korkeus on puun juurisolmun korkeus.

Syvyys (*depth*)

- Solmun syvyys on juuresta kyseiseen solmuun johtavan polun pituus.

Esimerkki 3.3. Oheisen puun solmun 3 korkeus on kaksi, solmun 4 korkeus on yksi ja solmun 2 korkeus on nolla. Koko puun korkeus on kolme. Solmun 6 syvyys puolestaan on kaksi.

Sisarukset

- Yhden solmun kaikkia lapsia kutsutaan yhdessä sisaruksiksi (*sibling*) (tai veljeksiksi).
- Sisarukset järjestetään yleensä vasemmalta oikealle (järjestetty puu).
  - Sisarusten järjestyksellä on merkitystä (yleensä).
  - Jos järjestystä ei määrätä, kyseessä on järjestämätön puu.
- Yleisessä puussa vasemmanpuoleisinta sisarusta sanotaan myös vanhimmaksi, vastaavasti oikeanpuoleisinta myös nuorimmaksi.
- Jollei solmuilla ole yhteistä isää, ne eivät ole sisaruksia.
  - Jos solmuilla on yhteinen isoisä, voidaan puhua serkuksista.
- Yleisessä puussa rajoittamattoman kokoinen sisarusparvi voidaan ilmaista joko solmun lasten listana tai kunkin solmun vanhimpana lapsena ja kunkin solmun seuraavana veljenä.
  - Lasten lista voisi olla selkeämpi, mutta lapsi-veli -järjestelmällä liittymä ja toteutus ovat

yksinkertaisempia.
– Palataan tähän yleisen puun käsittelyn yhteydessä (s. 50).

Puun solmujen järjestykset

Kuva 3.1: Kaksi sisällöltään erilaista järjestettyä puuta (mutta rakenteeltaan samanlaista).

- Solmut voidaan käsitellä tai luetella monessa eri järjestyksessä, tarpeen mukaan.
- Yleisimmät ovat esi-, sisä-, jälki- ja tasoittainen järjestys.
- Myöhemmin TRA II -kurssilla sovellamme samoja järjestyksiä verkoille.

Määritelmä 3.4. Esi-, sisä-, jälki- ja tasoittainen järjestys.

a) Tyhjän puun esi-, sisä-, jälki- ja tasoittainen järjestys ovat tyhjiä.

b) Yksisolmuisen puun esi-, sisä-, jälki- ja tasoittainen järjestys ovat puun ainoa solmu.

c) Jos puun juuri on n ja juuren alipuut ovat vasemmalta lukien T_1,T_2,...,T_k (binääripuulla vain T_1 ja T_2), on

   i) solmujen esijärjestys (*preorder*):
      1. n,
      2. T_1:n solmut esijärjestyksessä,
      3. T_2:n solmut esijärjestyksessä,
      4. ...,
      5. T_k:n solmut esijärjestyksessä;

   ii) solmujen sisäjärjestys (*inorder*): (joka sopii erityisesti binääripuulle jossa jokaisella on solmulla kaksi lasta (paikkaa lapselle), T_1 tai T_2 voi siis olla tyhjä):
      1. T_1:n solmut sisäjärjestyksessä,
      2. n,
      3. T_2:n solmut sisäjärjestyksessä,
      4. (...,)
      5. (T_k:n solmut sisäjärjestyksessä;)

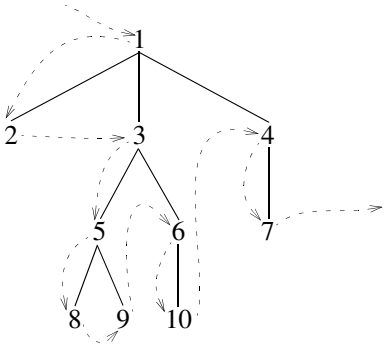
   iii) solmujen jälkijärjestys (*postorder*):
      1. T_1:n solmut jälkijärjestyksessä,
      2. T_2:n solmut jälkijärjestyksessä,
      3. ...,
      4. T_k:n solmut jälkijärjestyksessä,
      5. n.

   iv) Solmujen tasoittainen järjestys (*order by level*):

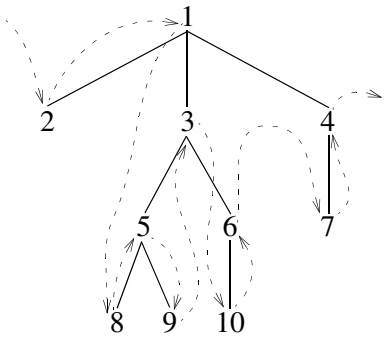
1. n,
2. syvyydellä 1 olevat solmut (n:n lapset) vasemmalta oikealle,
3. syvyydellä 2 olevat solmut (n:n lapsenlapset) vasemmalta oikealle,
4. ...,
5. puussa syvimmällä olevat solmut vasemmalta oikealle.

Esimerkki 3.5. Kuvan puun solmut ovat

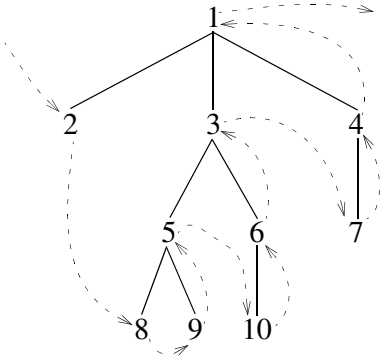
1. esijärjestyksessä 1, 2, 3, 5, 8, 9, 6, 10, 4, 7;



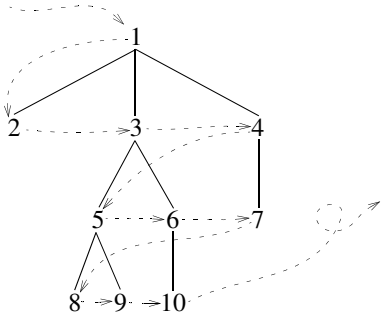
2. sisäjärjestyksessä 2, 1, 8, 5, 9, 3, 10, 6, 7, 4;



3. jälkijärjestyksessä 2, 8, 9, 5, 10, 6, 3, 7, 4, 1;

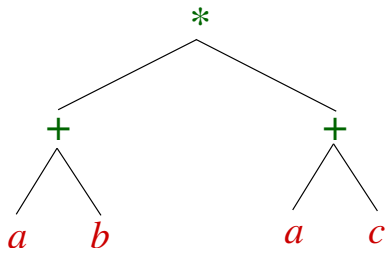


4. tasoittain järjestettynä 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.



Esimerkki 3.6. Kuvan 3.2 nimiöity puu kuvaa aritmeettista lauseketta $(a+b)*(a+c)$. Puussa on seitsemän solmua. Kustakin solmusta puuhun on merkitty vain sen nimiö. Solmut ja solmujen nimiöt voidaan siis piirrettäessä aivan hyvin samaistaa.

1. Jokainen lehti nimiöidään operandilla.



Kuva 3.2: Aritmeettisen lausekkeen jäsennyspuu.

2. Jokainen haarautumissolmu nimiöidään operaattorilla siten, että haarautumissolmun vasen alipuu kuvaa operaattorin vasemmanpuoleista operandia ja vastaavasti oikea alipuu oikeanpuoleista operandia.

Esimerkki 3.7. Esimerkin lauseke on esimerkintäisenä $*+ab+ac$, sisämerkintäisenä $a+b*a+c$, ja jälkimerkintäisenä $ab+ac+*$. Näistä kolmesta esitysmuodosta esi- ja jälkimerkintäisen laskentajärjestys on yksiymmärteinen. Vain sisämerkintäiseen muotoon tarvitaan sulkuja laskentajärjestyksen osoittamiseksi.

Puun sisältö

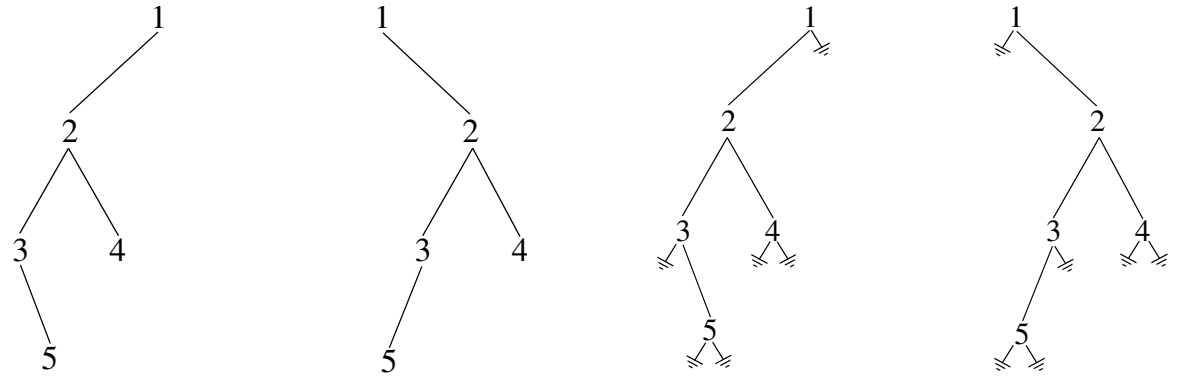
⇒ Kuten muissakin kokoelmissa, puissakin halutaan säilyttää hyötytietoa varsinaisen sovelluksen tarpeisiin.

- Usein puun solmussa olevaa alkiota sanotaan nimiöksi (*label*).
- Käyttämämme tietorakennekirjaston puu tyyppiparametroidaan samoin kuin listakin.
- Puu sisältää puusolmuja, kukin puusolmu ylläpitää viittausta yhteen hyötytietoon (*Object*:n aliluokan ilmentymään), aivan vastaavasti kuten *TraLinkedList*.
- Puu ja puusolmut ovat tyyppiparametroituja, molempien parametrityyppien ($<E>$) on oltava samat.
- Puusolmu on nyt pakollinen koska tarvitsemme kullekin solmulle monipuolisemman operaatiojoukon kuin vain seuraaja (vrt. *List*).
- Koska alkio on sovelluksen tyyppi, sillä ei voi olla esimerkiksi vasenta lasta.

3.2 Binääripuu

- Yleisen puun solmulla voi olla mielivaltainen määrä lapsia.
  - Tämä hankaloittaa toteutusta (tarvitaan lasten lista), hahmottamista ja hakemista.
  - Aikavaativuudet $O(\text{puun\_korkeus} \times \text{sisarussarjojen\_koko})$.
- Binääripuussa rajoitetaan lasten määrä enintään/tasan kahteen.
  - Solmulla on/ei ole vasen lapsi.
  - Solmulla on/ei ole oikea lapsi.
  - Jos solmulla on vain yksi lapsi, se on joko oikea tai vasen (jolloin toinen ”lapsi” on ”tyhjä”).
- Hyödyt:
  - Toteutus suoraviivaista, suora viittaus molempiin lapsiin ja (tarvittaessa) vanhempaan.
  - Eteneminen syvemmälle puussa yhden vertailun perusteella suoraan jompaan kumpaan lapseen.
    - Moni algoritmi käy läpi puusta vain yhden haaran ylhäältä alas (tai päinvastoin), jolloin aikavaativuudeksi tulee $O(\text{puun\_korkeus})$.

Esimerkki 3.8. Binääripuiden piirtäminen:



(a) Tyhjät lapset näkymättömissä.

(b) Tyhjät lapset piirretty näkyviin.

Kuva 3.3: Kaksi erilaista viisisolmuista binääripuuta.

- Kuvan 3.3a puiden tyhjiä lapsia ei ole piirretty (suunta ilmaistu vinon viivan suunnalla).
- Kuvassa 3.3b tyhjät lapset on piirretty (⏚). Tämä kuva (ehkä) selkeyttää toteutusperiaatetta ja korostaa vasemman ja oikean lapsen eroa, mutta ei ole kätevä jatkuvasti piirrettäessä.
- Algoritmeissa on useimmiten kätevä ottaa huomioon, että aliohjelmaa voidaan kutsua myös tyhjällä solmulla.
  - Riittää tehdä *null*-tarkastus yhdessä pisteessä.
- Molempien kuvan puiden solmut ovat esijärjestyksessä:
  - 1, 2, 3, 4, 5
- Sensijaan sisäjärjestyksessä solmut ovat
  - vasemmanpuoleisessa puussa:
    - 3, 5, 2, 4, 1
  - Kun taas oikeanpuoleisessa:
    - 1, 5, 3, 2, 4
- Vastaavasti on toki mahdollista löytää eri puita, joiden solmut ovat samat sisäjärjestyksessä, mutta eri esijärjestyksessä.

Binääripuun operaatiot TRA-kirjastossa

Määritelmä 3.9. Tavallisimmin tarvitaan seuraavanlaisia binääripuuoperaatioita:
(*BTree* T, *BTreeNode* n, m, E x)

1. BTree<E> BTree<E>()
   muodostaa tyhjän binääripuun.
2. BTreeNode<E> BTreeNode<E>(x)
   luo uuden kytkemättömän puusolmun jonka alkiona (hyötytietona) on x.
3. BTreeNode T.getRoot()
   palauttaa puun T juurisolmun. Jos T on tyhjä puu, palauttaa tyhjän solmun (*null*).
4. BTreeNode T.setRoot(n)
   asettaa puun T juureksi solmun n. Puussa aiemmin olleet solmut menetetään.
5. BTreeNode n.getParent()
   palauttaa solmun n isän. Jos n on juuri, palauttaa tyhjän solmun (*null*).
6. BTreeNode n.getLeftChild()
   palauttaa solmun n vasemman lapsen tai tyhjän solmun (*null*) jollei vasenta lasta ole.

7. BTreeNode n.getRightChild()
   palauttaa solmun n oikean lapsen tai tyhjän solmun (*null*) jollei vasenta lasta ole.

8. E n.getElement()
   palauttaa solmun n alkion (hyötytiedon, nimiön).

9. void n.setLeftChild(m)
   asettaa solmun n vasemmaksi lapseksi solmun m. Solmun n aiempi vasen lapsi (ja sen jälkeläiset!) menetetään.

10. void n.setRightChild(m)
    asettaa solmun n oikeaksi lapseksi solmun m. Solmun n aiempi oikea lapsi menetetään.

11. void T.killNode(n)
    tuhoaa ja vapauttaa puusta T solmun n ja kaikki sen lapset ja jälkeläiset.

⇒ Kaikkien puuoperaatioiden aikavaativuus tehokkaassa toteutuksessa on $O(1)$.

Hakupuut

- Miksi sisäjärjestetty binääripuu on niin hyvä (kätevä, tärkeä, jne)?

⇒ Hakeminen, lisääminen ja poisto: $O(\text{puun\_korkeus})$

- Puun korkeus on korkeintaan $n-1$ ($O(n)$) (triviaalisti, ei hyödytä hakemista)
- Puun korkeus vähintään $\log n$ ($\Omega(\log n)$).
- Voidaanko varmistaa, että puun korkeus on $O(\log n)$ (siis $\Theta(\log n)$)?
  - Pienellä lisävirittämisellä kyllä, kuten myöhemmin (kohta 8.4, s. 104) nähdään!
  - Hakeminen, lisääminen ja poisto: $O(\log n)$ ajassa eli varsin nopeaa vaikka syöte olisi hyvin suuri.
- Läpikäynti järjestyksessä $O(n)$ ajassa.

Esimerkki 3.10. Hakupuu: Ylläpidetään sisäjärjestettyä binääripuuta, ts. kaikki solmun vasemman alipuun alkiot edeltävät solmun alkiota ja kaikki oikean alipuun alkiot seuraavat solmun alkiota.

Hakeminen: jollei haettavaa alkiota x vielä löydetty solmusta n, niin jos haettava edeltää solmun n alkiota, haetaan solmun n vasemmasta alipuusta, muuten haetaan solmun n oikeasta alipuusta.

```
public static boolean inorderMember(BTree T, Comparable x) {
    BTreeNode n = T.getRoot();
    while (n != null) {
        if (x.compareTo(n.getElement()) == 0)
            return true;
        else if (x.compareTo(n.getElement()) < 0)
            n = n.getLeftChild();
        else
            n = n.getRightChild();
    }
    return false;
}
```

Aikavaativuus: kullakin tasolla: 2 vertailua ja linkin seuraaminen ($O(1)$). Yhteensä $O(\text{puun\_korkeus})$. Puun korkeus voi vaihdella välillä $\log n..n$, missä n = puun solmujen määrä.

Samat alkiot sisältäviä sisäjärjestettyjä binääripuita voi olla erilaisia!

- Antaa ”joustavuutta” lisäämiseen (toki lisäyskohta on aina yksikäsitteinen).
- Aina voidaan lisätä lehtisolmuksi (korvata jonkin solmun lapsena oleva ”tyhjä solmu” uudella lapsettomalla solmulla).
- Jos sisäjärjestetyssä puussa on n solmua, uudelle alkiolle on $n+1$ mahdollista paikkaa (kunkin n alkion eteen sekä lisäksi viimeiseksi).
  - n-solmuisessa binääripuussa on sopivasti $n+1$ ”tyhjää lasta”!
- Koska alkiot on talletettu sisäjärjestettyyn puuhun järjestykseen ja koska uudella alkiolla on sijainti järjestyksessä, on lisäyskohta aina yksikäsitteinen.
  - Puun rakenne siis määräytyy lisäysjärjestyksen mukaan.
- Jotta puu ei kasva turhan korkeaksi (hitaaksi), puuta voidaan tarvittaessa oikaista siten, että sisäjärjestys säilyy (s. 104).

Esimerkki 3.11. Lisäys sisäjärjestettyyn binääripuuhun s.e. sisäjärjestys säilyy:
Suoritetaan etsintä (kuten esimerkki 3.10) puussa kunnes törmätään tyhjään solmuun jossa ko. uusi alkio olisi jos olisi.
Lisätään alkio uudeksi solmuksi ko. tyhjän solmun paikalle (vanhempansa vasemmaksi/oikeaksi lapseksi). [HT]

Esimerkki 3.12. Annetun lähtö**solmun** n seuraaja sisäjärjestyksessä. Jos solmulla n on oikea lapsi, seuraaja on ko. oikean lapsen vasemmanpuoleisin jälkeläinen. Muuten, seuraaja on se (esi)vanhempi jonka vasemmassa alipuussa lähtösolmu n on. [HT]

- Jos solmulla n on oikea lapsi (ei ole *null*), niin:
  - $s \leftarrow$ n:n oikea lapsi
  - niin kauan kuin s:n vasen lapsi ei ole *null*, toista:
    - $s \leftarrow$ s:n vasen lapsi
  - palauta s [seuraaja löytyi]
- muuten [n oikea lapsi on *null*]:
  - $s \leftarrow n$
  - $p \leftarrow$ n:n vanhempi
  - niin kauan kuin p ei ole *null*, toista:
    - jos s on p:n vasen lapsi, palauta p [seuraaja löytyi]
    - muuten:
      - $s \leftarrow p$
      - $p \leftarrow$ p:n vanhempi
  - palauta *null* [seuraajaa ei ole]

Esimerkki 3.13. Annetun lähtösolmun n edeltäjä sisäjärjestyksessä. Jos solmulla n on vasen lapsi, edeltäjä on ko. vasemman lapsen oikeanpuoleisin jälkeläinen. Muuten, edeltäjä on se (esi)vanhempi jonka oikeassa alipuussa lähtösolmu n on. [HT]

Esimerkki 3.14. Puun alkio joka on seuraavaksi suurempi **alkioon** x nähden. Huomaa ero esimerkkiin 3.12, nyt ei aloiteta solmusta jossa x on, eikä alkiota x välttämättä edes ole puussa.

Tehdään haku normaalisti juuresta alkaen, mutta jos löydetään alkio x, jatketaan oikealle (siis oikealle jos $n.getElement().compareTo(x) \leq 0$, muuten vasemmalle). Alaspäin edetessä pidetään yllä pienintä x:ää suurempaa alkiota johon on törmätty. Kun päästään umpikujaan, kyseinen havaittu pienin alkio oli seuraava x:ää suurempi.

Edellinen pienempi alkio löytyy vastaavasti, algoritmissa pienempi $\leftrightarrow$ suurempi ja oikea $\leftrightarrow$ vasen.

Esimerkki 3.15. Poisto sisäjärjestetystä binääripuusta. Poistaminen on helppoa jos poistettava on lehtisolmu tai poistettavalla on vain yksi lapsi (jolloin ko. lapsi sijoitetaan poistettavan solmun

tilalle). Muuten joudutaan hakemaan solmun alkion tilalle edeltäjäsolmun alkio (tai seuraajasolmun) ja poistamaan ko. edeltäjäsolmu. [*BinPuuEsim.java*]

Esimerkki 3.16. Binääripuun korkeus on juurisolmun korkeus. Solmun korkeus on korkeamman lapsen korkeus +1. Lehtisolmun korkeus on 0. Tyhjän solmun "korkeus" on siten −1.

3.3 Yleinen puu abstraktina tietotyyppinä

Binääripuu on hyvä hakurakenne, mutta sillä ei voi esittää yleistä hierarkiaa jos/kun solmulle tarvitaan enemmän kuin kaksi seuraajaa (jälkeläistä).

- Organisaatiokaavio, prosessin jakautuminen osatehtäviin, luokkien perimys, pelitilanteet, jne.
- Yleisessä puussa rajoittamattoman kokoinen sisarussarja voidaan ilmaista joko solmun lasten listana tai kunkin solmun vanhimpana lapsena ja kunkin solmun seuraavana veljenä.
- Lasten lista voisi olla selkeämpi ajatuksena, mutta lapsi-veli -järjestelmällä sekä liittymä, että toteutus ovat yksinkertaisempia, käytetään tässä jälkimmäistä.
- Samoin lapsi-veli -järjestelmän toteutuksessa kullekin puun tasolle tulee 1-2 viittaustasoa vähemmän.

Määritelmä 3.17. Tavallisimmin tarvitaan seuraavanlaisia puuoperaatioita: (*Tree T*; *TreeNode n*, *m*; *E x*)

1. Tree<E> Tree<E>()
   muodostaa tyhjän puun.

2. TreeNode<E> TreeNode<E>(x)
   luo uuden kytkemättömän puusolmun jonka alkiona (hyötytietona) on x.

3. TreeNode T.getRoot()
   palauttaa puun T juurisolmun. Jos T on tyhjä puu, palauttaa tyhjän solmun (*null*).

4. TreeNode T.setRoot(n)
   asettaa puun T juureksi solmun n. Puussa aiemmin olleet solmut menetetään.

5. TreeNode n.getParent()
   palauttaa solmun n isän. Jos n on juuri, palauttaa tyhjän solmun (*null*).

6. TreeNode n.getLeftChild()
   palauttaa solmun n vanhimman lapsen. Jos n on lehtisolmu, palauttaa tyhjän solmun (*null*).

7. TreeNode n.getRightSibling()
   palauttaa solmun n lähinnä nuoremman sisaruksen. Jos n on juuri tai isänsä nuorin (oikeanpuoleisin) sisarus, palauttaa tyhjän solmun (*null*).

8. E n.getElement()
   palauttaa solmun n alkion (nimiön, hyötytiedon).

9. void n.setLeftChild(m)
   asettaa solmun n vasemmanpuoleiseksi lapseksi solmun m. Solmun n aiempi vasemmanpuoleinen lapsi (ja sen sisarukset!) menetetään.

10. void n.setRightSibling(m)
    asettaa solmun n lähinnä oikeanpuoleiseksi sisareksi solmun m. Solmun n aiempi lähinnä oikeanpuoleinen sisarus menetetään.

11. void T.killNode(n)
    tuhoaa ja vapauttaa puusta T solmun n ja kaikki sen lapset ja oikeanpuoleiset sisarukset.

⇒ Kaikkien puuoperaatioiden aikavaativuus tehokkaassa toteutuksessa on $O(1)$.

Esimerkki 3.18. Puuoperaatioiden käytöstä: Algoritmi, joka tuottaa parametriensa yksilöimän (ali)puun solmujen nimiöiden listauksen käymällä puun solmut läpi esijärjestyksessä. Rekursiivisena algoritmi on seuraavanlainen:

```
public static void preorderPrint(Tree T) {                              1
    if (T.getRoot() != null)                                             2
        preorderPrintBranch(T.getRoot());                                3
    System.out.println();                                                4
}                                                                        5
public static void preorderPrintBranch(TreeNode n) {                    6
    System.out.print(n.getElement() + "␣");      // tms hyödyllistä      7
    TreeNode child = n.getLeftChild();                                   8
    while (child != null) {                                              9
        preorderPrintBranch(child);                                      10
        child = child.getRightSibling();                                 11
}   }                                                                    12
```

Itse *preorderPrint* on vain kuorrutus joka kutsuu varsinaista rekursiivista läpikäyntiä puun juurella. Algoritmin rivillä 7 voidaan luonnollisesti suorittaa muitakin operaatioita kuin tulostamista.

Puiden rakentamisesta

⇒ Puun rakentuminen riippuu puuta käyttävästä ohjelmasta (hyötytiedon rakenteesta).

- *set{Root | LeftChild | RightSibling}* sekä *new TreeNode*(x) soveltuvat solmu kerrallaan lisäämiseen (miten muuten puita rakennetaan?).
- Jos rakennamme koko puun kerralla (esim. olemme saaneet syötteenä kasan alkioita joista muodostamme kokoelman tavalla tai toisella):
  - Pyritään aloittamaan juuresta ja etenemään syvyyssuunnassa alaspäin.
    - Näin puun irrallisia oksia ei tarvitse säilyttää väliaikaiskokoelmissa,
    - eikä tarvitse jatkuvasti hakea lisäyskohtaa juuresta käsin.
  - Alhaalta ylös rakennettaessa oksia voidaan pitää tallessa esimerkiksi jonossa tai pakassa (tai rekursiossa).
- Joskus on määritelty myös *construct*(n,T_1,T_2,...,T_k).
  - Luo puun palikoista kerralla.
  - Muuttaminen, erityisesti uudelleenjärjestely on lähes mahdotonta.

Solmuista ja puista

Solmu ?=? Puu

- Voisimmeko viitata yhteen solmuun kuten yhteen puuhun ja päinvastoin.
- Esimerkiksi ”Hae puusta T” toimisi myös vaikka T olisi mielivaltainen alipuu.
- Operaatioille riittäisi yksi parametri.
- Erityisesti puiden rakentelussa olisi mukavaa, jos kokonaisen aiemmin rakennetun puun voisi lisätä toiseen puuhun oksaksi.

- Sopivin järjestelyin tämä onnistuu, TR-kirjastossa tämä on toteutettu *set* -operaatioille. Algoritmien kirjoittamisessa rajoituksena (esteenä) on tyypintarkastuksen haasteet.
- Palataan tähän puiden toteuttamisen yhteydessä.

Esimerkki 3.19. Polku puun solmuun jossa on alkio x. Algoritmi etsii polun järjestämättömän puun T juuresta solmuun, jonka nimiö on x. Algoritmi myös tulostaa löytämänsä polun solmujen nimiöt juuresta alkaen. Algoritmi etsii aluksi oikean solmun puusta. Etsinnän aikana algoritmi pitää yllä pakkaa, jossa on tallessa polku juuresta siihen solmuun, jota ollaan juuri tutkimassa. *PuuEsim.searchPath*().

Esimerkki 3.20. Haku esijärjestetystä puusta. Aloitetaan juuresta, edetään lapsiin, oikeisiin veljiin kunnes löytyy, tai löytymättömyysehto täyttyy, tai törmätään tyhjään solmuun. Jos haettava alkio on pienempi kuin käsittelyssä oleva solmu, ei sitä puusta löydy, jos pienempi kuin oikea veli, niin vaihdetaan käsittelykohta vasempaan lapseen, muuten vaihdetaan käsittelykohta ko oikeaan veljeen. HT.

Esimerkki 3.21. Läpikäynti tasoittain (leveyssuuntaisesti). Viedään juuri jonoon. Toistetaan kunnes jono on tyhjä. Otetaan alkio jonosta, käsitellään alkio, viedään kaikki lapset jonoon.

```
public static void printByLevel(Tree T) {                                    1
    LinkedQueue<TreeNode> Q = new LinkedQueue<TreeNode>();                    2
    if (T.getRoot() != null)                                                  3
        Q.offer(T.getRoot());                                                 4
    while (! Q.isEmpty()) {                                                   5
        TreeNode n = Q.poll();                                                6
        System.out.print(n.getElement() + "␣");     // tms hyödyllistä        7
        n = n.getLeftChild();                                                 8
        while (n != null) {                                                   9
            Q.offer(n);                                                       10
            n = n.getRightSibling();                                          11
        }                                                                     12
    }                                                                         13
    System.out.println();                                                     14
}                                                                             15
```

Muita puita

- Paitsi vakioon 2, voidaan lapsien määrä rajoittaa mihin tahansa muuhunkin vakioon.
- Kätevä on esimerkiksi aakkoston koko:
  - Kussakin puun haarassa on taulukkona lapset (esim) [a..ö]
  - Varsinainen tieto tallennetaan (viittauksena) syvyydelle *avaimen_pituus* puussa
  - Hakeminen aina O(avaimen_pituus)
  - Kullakin avaimella on yksikäsitteinen paikka puussa.
  - Jos avaimia on paljon (suhteessa niiden pituuteen), ei lisätilaa mene paljoakaan.
- B-puu
  - Sisäjärjestetty hakupuurakenne joka on optimoitu massamuistissa toteutettavaksi.
  - Tavoitteena leveä, hyvin matala puu (massamuistissa 2-3 tasoa).
  - Puun solmussa on $t..2t-1$ avainta, ja solmulla on lapsia yksi enemmän kuin avaimia.
  - Solmu ladataan kerralla keskusmuistiin, solmussa binäärihaku.
  - Juurisolmun kopiota ylläpidetään keskusmuistissa.
  - Käsitellään tarkemmin TRA II -kurssilla.

4 Joukot

⇒ Listat ja puut olivat melko konkreettisia kokoelmia – niissä yksilöitiin alkioiden sijainti.

Joukko (*set*) on vielä jonkin verran abstraktimpi.

⇒ Joukko on vain ”musta laatikko” jossa alkiot ovat.

Kokoelmana yksi käytetyimpiä:

- tietokannat (relaatiotietokannoissa relaatio on tai ei ole)
- solmujen joukko
- avainsanojen joukko ja tunnusten joukko ohjelmointikielessä
- jne

4.1 Määritelmiä

Joukko (*set*) on kokoelma mielivaltaisen tyyppisiä alkioita.

- Alkion tyyppiä ei rajoiteta (kuten ei ole aiemminkaan rajoitettu).

Joukon alkiot ovat keskenään eri alkioita.

- Alkiot ovat kuitenkin samaa tyyppiä (tai ainakin vertailukelpoisia).
  - Yksi tyhjä alkio (*null*) sallitaan/ei sallita (tilanteen mukaan).
- Sama alkio ei siis voi koskaan olla joukossa kahteen kertaan!
- Kaksi alkiota jotka ovat sovelluksen kannalta samoja, eivät voi olla samanaikaisesti samassa joukossa.
  - Samuuden käsitteen tulee olla yksikäsitteinen.
  - Monimutkaisia alkiotyyppejä käsiteltäessä samuusfunktio jää käyttäjän tehtäväksi.
- Joukossa jo olevaa alkiota ei saa muuttaa siten, että sen samuus (verrattuna johonkin muuhun) muuttuu.

Usein (mutta ei aina) kokoelman alkiolle oletetaan lineaarinen järjestys (”järjestetty joukko”) (*ordered/sorted set*):

1. Jos a ja b ovat järjestetyn joukon S alkioita, vain yksi väittämistä

   (a) $a < b$
   (b) $a == b$
   (c) $b < a$

   on tosi. (ts. järjestys on yksiselitteinen)

2. Jos a, b ja c ovat joukon S alkioita siten, että $a < b$ ja $b < c$, niin $a < c$. (järjestys on transitiivinen).

Yksinkertaisille tyypeille on helppo määritellä järjestys.

- Monimutkaisille tyypeille, esim. joukoille on vaikeampaa.
- Esimerkiksi kokonaislukujoukkojen joukko:

  - pienin alkio
  - suurin alkio
  - alkioiden keskiarvo | summa | mediaani
  - alkioiden määrä
- Järjestys voi olla keinotekoinen jollei itse sovellus sitä tarjoa/tarvitse.

Joukko-operaatiot

- $\emptyset$: tyhjä joukko (*empty*)
- $=$: sama joukko (*equal*)
- $\in$: kuuluu joukkoon (*member*, *contains*)
- $\subseteq$: osajoukko (subset)
- $\subset$: aito osajoukko (*proper subset*)
- $\cup$: yhdiste (*union*)
- $\cap$: leikkaus (*intersection*)
- $\setminus$: erotus (*difference*)

- joukon luonti
- joukon kopiointi
- alkion lisäys
- alkion poisto
- alkioiden läpikäynti

Määritelmä 4.1. Abstraktin joukkotyypin tavanomaisimmat operaatiot merkityksineen ovat seuraavat (kurssin tietorakennekirjasto, Java API: katso s. 57): (*TraSet* A, B; E x)

1. TraSet<E> TraSet()
   muodostaa tyhjän joukon.

2. TraSet<E> TraSet(java.util.Collection<? extends E> C)
   muodostaa uuden joukon kopioiden siihen kokoelman C alkiot.

3. boolean A.isEmpty()
   palauttaa arvon *true*, jos joukossa A ei ole yhtään alkiota ($A == \emptyset$), muuten arvon *false*.

4. boolean A.equals(B)
   palauttaa arvon *true*, jos joukkojen A ja B sisällöt ovat samat, muuten arvon *false*.

5. TraSet A.union(B)
   palauttaa uuden joukon $A \cup B$ (yhdiste).

6. TraSet A.intersection(B)
   palauttaa uuden joukon $A \cap B$ (leikkaus).

7. TraSet A.difference(B)
   palauttaa uuden joukon $A \setminus B$ (erotus).

8. boolean A.contains(x)
   palauttaa arvon *true*, jos $x \in A$, muuten arvon *false*.

9. boolean A.add(x)
   vie joukkoon A alkion x säilyttäen joukon A muilta osin ennallaan, palauttaa *true* jos lisäys onnistui. Jos x oli jo joukossa A, lisäystä ei tehdä, ja palauttaa *false*.

10. boolean A.remove(x)
    poistaa joukosta A alkion x säilyttäen joukon A muilta osin ennallaan, palauttaa *true* jos alkio löydettiin ja poistettiin, *false* jollei alkiota joukosta löydetty.

11. TraSet A.clone()
    palauttaa kopion joukosta A. Uusi joukko sisältää samat alkiot (viittaukset) kuin A, mutta itse alkioita ei kopioida.

12. E A.first()
    palauttaa joukon A pienimmän alkion arvon. Operaation tulos on määrittelemätön, jos $A == \emptyset$ tai jos joukon A alkioille ei ole määritelty lineaarista järjestystä.

13. E A.last()
    palauttaa joukon A suurimman alkion arvon. Operaation tulos on määrittelemätön, jos $A == \emptyset$ tai jos joukon A alkioille ei ole määritelty lineaarista järjestystä.

Operaatioiden aikavaativuudet riippuvat toteutuksesta, niihin palataan tarkemmin myöhemmin.

- Perusoperaatiot $O(1)$ tai $O(\log n)$, usean joukon operaatiot on mahdollista tehdä $O(n)$ ajassa (n = joukkojen alkiomäärien summa).

⇒ Useimpiin joukkosovelluksiin riittää vain osa tässä mainituista operaatioista.

Joukon alkioiden läpikäynti

Ylläolevat operaatiot eivät tue joukon alkioiden tehokasta läpikäyntiä joukkoa muuttamatta.

- Aina tätä ei tarvita, mutta usein kyllä.
- Yleensä riittää: ”kaikille joukon S alkioille, tee seuraavaa”.

Algoritmiteksteissä joukon kaikkien alkioiden läpikäynti esitetään yleensä seuraavaan tapaan:

```
for each x in S do                    1
    toimenpide alkiolle x             2
```

- Javassa (versio $\geq$ 1.5) tämä onnistuu kaikille *Iterable* -rajapinnan toteuttaville kokoelmille:

```
for (x : S)                           1
    toimenpide alkiolle x             2
```

- missä x on viittaus (muuttuja) kokoelman S:n alkiotyyppiä.
- Tämä on selkeä, mutta ei salli kokoelman muuttamista, eikä mahdollista toistoon vaikuttamista.
- Vaihtoehtoisesti:
  - ohjelmointikielen oma toistolause (*while*)
  - sopiva lopetusehto
  - sopiva tapa saada esille ensimmäinen ja seuraavat käsiteltävät alkiot
  - iterointimuuttuja (ja sen tyyppi).

Määritelmä 4.2. Joukon läpikäyntioperaatiot (*TraSet* S; *Iterator* i; E x)

1. Iterator<E> S.iterator()
   Alustaa joukon S läpikäynnin iterointimuuttujaan i.

2. boolean i.hasNext()
   palauttaa arvon *true*, jos joukon S läpikäynti i on vielä kesken, ts. $i.next()$ -operaatiolla saataisiin uusi alkio, muuten palauttaa arvon *false*.

3. E i.next()
   palauttaa joukon S jonkin läpikäynnissä i vielä käsittelemättömän alkion. Jos joukon S kaikki alkiot on jo käsitelty läpikäynnissä i (eli *i.hasNext*() olisi palauttanut epätoden), aiheuttaa ajonaikaisen poikkeuksen.

4. void i.remove()
   Poistaa kokoelmasta S edellisen *i.next*() -operaation antaman alkion.

- Läpikäytävää joukkoa ei saa muuttaa läpikäynnin aikana (paitsi tämän iteraattorin *i.remove*() -operaatiolla), ei esim. saa lisätä tai poistaa alkioita.
- Läpikäyntimuuttujalla ei voi laskea (koska se on viite)
- Läpikäyntimuuttujia ei voi verrata (tai ainakaan siinä ei ole mitään järkeä).
- Läpikäyntijärjestystä ei oletusarvoisesti taata.
  - *HashSet*: läpikäyntijärjestys on ”satunnainen” ja voi vaihdella kerrasta toiseen.
  - *TreeSet* ja *TraSet*: läpikäynti alkioiden ”suuruus”järjestyksessä (luonnollisessa järjestyksessä).
  - *LinkedHashSet*: läpikäynti lisäysjärjestyksessä.
- Sisäkkäiset läpikäynnit (tai eri joukkojen läpikäynnit) eri iterointimuuttujilla.
- https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html

Esimerkki 4.3. Kaikkien alkioiden läpikäynti käyttäen määritelmän 4.2 operaatioita:

```
import java.util.Iterator;                                                    1
...                                                                           2
Iterator<E> i = S.iterator();                                                 3
while (i.hasNext()) {                                                         4
    E x = i.next();              // tai joku tyyppi suoraan E:n tilalla       5
    toimenpide alkiolle x;                                                    6
}                                                                             7
```

⇒ Alkuehtoinen toistolause on ehdoton, sillä myös ensimmäinen *next* voi epäonnistua jos joukko on tyhjä!

iterator, *hasNext* ja *next* tulee toteuttaa toimimaan $O(1)$ ajassa (keskimäärin).

Esimerkki 4.4. Taulukon V alkioiden kaikki permutaatiot eli erilaiset järjestykset, joita on kaikkiaan $|V|!$ kappaletta, saadaan kerätyksi listaan:

```
public static <E> LinkedList<Vector<E>> permutations(Vector<E> V) {           1
    TreeSet S = new TreeSet(V);                                               2
    LinkedList<Vector<E>> L = new LinkedList<Vector<E>>();                    3
    permute(V, 0, S, L);                                                      4
    return L;                                                                 5
}                                                                             6
                                                                              7
public static <E> void permute(Vector<E> V, int i, TreeSet<E> S, LinkedList<Vector<E>> L) {  8
    if (S.isEmpty())                                                          9
        L.add(new Vector(V));                                                 10
    else {                                                                    11
        for (E x : S) {                                                       12
            V.set(i, x);                                                      13
            TreeSet<E> R = new TreeSet<E>(S);                                 14
            R.remove(x);                                                      15
            permute(V, i+1, R, L);                                            16
}   }   }                                                                     17
```

java.util.Set

- https://docs.oracle.com/javase/8/docs/api/java/util/Set.html
- Liittymä joka määrittelee joukko-operaatiot (ei tosin yhdisteitä, jne).
- JavaAPI:ssa niin *HashSet* kuin *TreeSet* on itseasiassa toteutettu käyttäen vastaavia kuvaustoteutuksia (*HashMap*/*TreeMap*, s. 60) (tarkemmin kuvauksen avaimia), mutta sen ei pidä antaa häiritä.
- Joukossa olevia alkioita ei saa muuttaa siten, että *.equals*() -operaation tulos muuttuu!
- Käytännössä kuten ylläoleva *TraSet*, mutta liittymä ei tarjoa sellaisia usean joukon operaatioita jotka muodostavat uuden tulosjoukon.
  - Niiden sijaan liittymä tarjoaa sisältöä muuttavat operaatiot *addAll*(), *containsAll*(), *removeAll*(), *retainAll*()
  - Näillä voidaan toki saada aikaan aivan sama lopputulos, esim.

```
Set leikkaus = new TreeSet(A);                                          1
leikkaus.retainAll(B);                                                  2
Set leikkaus2 = new TreeSet() {{ addAll(A); retainAll(B); }};           3
```

  - Joukko-operaatioiden aikavaativuus on $O(n\log n)$ (*TreeSet*:llä) tai $O(n)$ (*HashSet*:llä).
    - Käytännössä nopeusero on useimmiten vain 2-4 -kertainen.
    - Uudempien Java versioiden *TreeSet.addAll*(*SortedSet*)) -operaatio on lineaarinen kunhan molemmat ovat järjestettyjä joukkoja.
- *java.util.HashSet*<*E*>
  - *java.util.Set*:n toteutus avoimella hajatuksella (kts "Hajautus" s. 98).
  - *add*, *remove* ja *contains* ovat aikavaativuudeltaan yleensä/keskimäärin $O(1)$.
    - Tilan loppuessa uudelleenhajautus vie lineaarisen ajan.
    - Mutta koska se tapahtuu vain n operaation välein, niin se ei vaikuta keskimääräiseen aikavaativuuteen. (vrt. *ArrayList* tilan kasvatus)
  - Yksi *null* arvo joukossa on sallittu (mutta ei suositeltava).
  - Läpikäynti vain "satunnaisessa" järjestyksessä, läpikäyntijärjestys saattaa jopa muuttua läpikäynnistä toiseen.
  - *java.util.LinkedHashSet*<*E*>
    - Kuten *HashSet*, mutta toteutukseen on lisätty alkioiden (itseasiassa solmujen) linkitetty lista jota ylläpidetään automaattisesti.
    - Listaa käytetään joukon alkioiden läpikäyntiin jolloin alkiot saadaan läpikäytyä aina lisäysjärjestyksessä.

java.util.NavigableSet (ja *SortedSet*)

- https://docs.oracle.com/javase/8/docs/api/java/util/NavigableSet.html
- Liittymä joka laajentaa *Set*-liittymää.
  - Alkioiden on toteutettava *Comparable*-rajapinta (tai kokoelman on käytettävä *Comparator* -apuluokkaa).
    - *Comparable*:a käyttäen ei sallita *null* -arvoja, *Comparator* voidaan tehdä siten, että *null*:lla on yksikäsitteinen sijainti.
  - Joukossa olevia alkioita ei saa muuttaa siten, että vertailun tulos muuttuu!
    - Alkion poisto ja muutoksen jälkeen lisäys takaisin on toki luvallista.
  - *first*(), *last*(), *higher*(), *headSet*(), *subSet*(), läpikäynti kasvavassa järjestyksessä, ...
  - Osajoukko-operaatiot tarjoavat uuden näkymän joukkoon, eivät muodosta uutta joukkoa.
- *java.util.TreeSet*<*E*>
  - *java.util.NavigableSet*:n toteutus tasapainotetulla binääripuulla (puna-musta puu).

  - Kuten ylläoleva *TraSet*, mutta ei usean joukon operaatioita (paitsi *AbstractCollection*:lta perityt *addAll*(), *containsAll*(), *removeAll*(), *retainAll*())
  - *add*, *remove* ja *contains* aikavaativuus $O(\log n)$.
- Paremmin nimetty *SortedSet* oli alkuperäinen versio joka haluttiin säilyttää yhteensopivuussyistä, siksi uusi (JSE6-) *NavigableSet*.
  - Lisäyksiä mm. hienojakoisemmin osajoukkoja, läpikäynti takaperin, jne.

java.util.Collection:n *xxxAll*() -operaatiot

- *addAll*(), *containsAll*(), *removeAll*() ja *retainAll*() on toteutettu jo runkoluokassa *AbstractCollection*. https://docs.oracle.com/javase/8/docs/api/java/util/AbstractCollection.html
- Nämä operaatiot ottavat parametrina toisen kokoelman.
- Nämä runkototetukset käyttävät operaatioita jotka löytyvät kaikilta kokoelmilta (sekä muokattava kokoelma, että parametrikokoelma).
  - *add*(), *remove*(), *contains*(), *iterator*().
- Osa operaatioista on toteutettu uudelleen joissakin luokissa joissa toteutustapa mahdollistaa tehokkaammat operaatiot.
- Operaatiot ovat riippuvaisia myös parametrikokoelman operaatioiden aikavaativuuksista!
  - Esimerkiksi *removeAll*() ja *retainAll*() käyvät iteraattorilla läpi tätä kokoelmaa ja kutakin alkiota kohti kutsuvat parametrikokoelman *contains*() -operaatiota (ja sen mukaan poistavat tai säilyttävät).
    - Niinpä *removeAll*() ja *retainAll*():n aikavaativuus riippuu myös parametrikokoelmasta (sen *contains*():n aikavaativuudesta).
      - *HashSetA.retainAll*(*HashSetB*): $O(|A|)$
      - *HashSetA.retainAll*(*ArrayListB*): $O(|A| \times |B|)$
    - Tämäkin on kerrottu JavaAPI:n dokumentaatiossa!

4.2 Sanakirja

⇒ Sovelluskohtaisesti voidaan käytettävien joukko-operaatioiden määrää usein rajata huomattavasti.

- Yksittäisen kokoelman hallintaan riittävät *insert*, *remove*, *contains*.
- Eikä *remove*:kaan aina ole välttämätön.
  - Kokoelmaan vain lisätään alkioita ja niitä välillä haetaan.
    - Esim. kääntäjän symbolitaulu.
  - Joskus jopa koko kokoelma luodaan kerralla, ja sitten siitä vain haetaan.
    - Esim. (CD) puhelinluettelo.

Tälläistä vain yhtä joukkoa hallitsevaa kokoelmaa sanotaan sanakirjaksi (*dictionary*).

- *contains* ja *remove* -operaatioiden x-parametria sanotaan yleensä avaimeksi.
- Avain ei välttämättä ole koko alkio, avainta käytetään varsinaiseen hyötytietotietueeseen viittaamiseen.
  - Tällöin *insert* ja *get* tarvitsevat (palauttavat) varsinaisen hyötytiedon, muut tulevat toimeen avaimella.
  - Tai sitten *contains* palauttaa totuusarvon sijaan hyötytiedon.
- JavaAPI:ssa alkion osan käyttö avaimena rakentuu *.equals*() ja *.compareTo*() -metodien kautta.
  - Alkio annetaan parametrina kokonaan, mutta vertailut käsittelevät vain avain-osaa.
  - Tai sitten käytetään yleisemmin kuvausta (*Map*), kts. seuraava.

4.3 Relaatio ja kuvaus

⇒ Jos joukon alkioilla on erityinen rakenne, voidaan joukko-operaatiot määritellä uudestaan, esimerkiksi:

Määritelmä 4.5. Relaatio.

A ja B joukkoja, $A \times B$ näiden karteesinen tulo

- $A \times B$:n alkiot ovat järjestettyjä pareja (a, b), $a \in A$ ja $b \in B$.
- A:n ja B:n alkiotyypit voivat olla keskenään erilaisia.

$M \subseteq A \times B$ on relaatio

- merkintöjä:
  - $(a, b) \in M \Leftrightarrow a\ M\ b$,
    - eli a ja b ovat keskenään relaatiossa M
  - A on lähtöjoukko, B on maalijoukko

⇒ Relaatio toteutetaan kuten joukko

- Monikkojen (*tuple*) joukko (mutta kuitenkin olioiden joukko).
- Relaatiossa voi olla useampikin kuin kaksi osaa.

Määritelmä 4.6. Kuvaus

Jos kaikille relaation M alkioille (a_1,b_1), (a_2,b_2) pätee

$$a_1 \neq a_2 \text{ tai } b_1 = b_2 \tag{4.1}$$

on kyseinen relaatio kuvaus (*mapping*, *function*).

- Relaatio on kuvaus, jollei siihen voi sisältyä kahta paria, joiden ensimmäiset jäsenet olisivat samat.
- Tällöin b on a:n kuva.
- Merkitään $M(a) = b$.
- Toisin sanoen: jos kuva on määritelty, se on yksikäsitteinen.
- Kuvauksen määritelmä on sama kuin funktion määritelmä, joten jossain yhteyksissä siitä käytetään nimitystä funktio.

Esimerkki 4.7. Kuvauksia.

- neliöintikuvaus $\text{sqr}(a) = a^2$
  - kuvaus voidaan laskea (funktio)
- opiskelijanumeron ja henkilötietojen välinen yhteys
  - henkilötiedot täytyy tallettaa; tiedot haetaan esiin opiskelijanumeron avulla
- relaatio $<$ (kokonaislukujen joukossa) ei ole kuvaus:
  - $1 < 2$ ja $1 < 3$

Kuvaus abstraktina tietotyyppinä

- Koska kuvaus on yleisen joukon erikoistapaus, voitaisiin kuvauksia käsitellä yleisen joukon operaatioilla.
  - Nämä ovat kuitenkin tarpeettoman monipuolisia, sanakirjaoperaatiot yleensä riittävät.
  - Lisäksi kun alkiot ovat pareja, on niitä kätevämpi käsitellä kahdella parametrilla esim. lisäysoperaatiossa.

– Parametroidaan kuvaukseen erikseen avaintyyppi (*Key*, K) ja alkiotyyppi (*Value*, E).

Määritelmä 4.8. Kuvaus abstraktina tietotyyppinä (*Map* M, K k, V x)

1. Map<K, V> Map<K, V>()
   muodostaa tyhjän kuvauksen, avaimet tyyppiä K ja arvot tyyppiä V.
2. V M.put(k, x)
   määrittelee avaimen k kuvaksi kuvauksessa M arvon x. Saman alkion kuvan uudelleen määritteleminen kumoaa aiemman määrittelyn (ja palauttaa sen operaation arvona). [vertaa taulukon *set*]
3. boolean M.containsKey(k)
   palauttaa arvon *true* jos avaimen k kuva on määritelty, muuten palauttaa arvon *false*.
4. V M.get(k) palauttaa avaimen k kuvan jos se on määritelty, muuten *null*.
5. V M.remove(k)
   poistaa avaimen k kuvan kuvauksesta ja palauttaa sen jos se on määritelty, muuten *null*.

- Äärettömissä (erityisesti jatkuvissa) kuvauksissa *set* ei ole mielekäs, joten se jätetään pois ja *compute* (*get*) toteutetaan funktiolla (laskemalla).
- Kuvaus (ja relaatio) toteutetaan kuten joukkokin.
- Aikavaativuudet $O(\log n)$ puutoteutuksessa, $O(1)$ onnistuneessa hajautustoteutuksessa.

java.util.Map, *HashMap*, *NavigableMap*, *TreeMap*

- Kts. https://docs.oracle.com/javase/8/docs/api/java/util/Map.html
- Perusoperaatiot (*put*, *get*, *remove*), järjestetyn kuvauksen operaatiot, avainten joukko, kuvat, jne.
- Aikavaativuudet vaihtelevat toteutuksen (*Tree*/*Hash*) mukaan.
- Osa operaatioista on tehokkaita (esim. *containsKey*()), osa ei (esim. *containsValue*()).
- Esim. avainten läpikäynti:

```
for (K key : M.keySet()) {
    ...
```

- Avain-arvo -pareja voi käsitellä nippuna *Map.Entry* -luokalla:

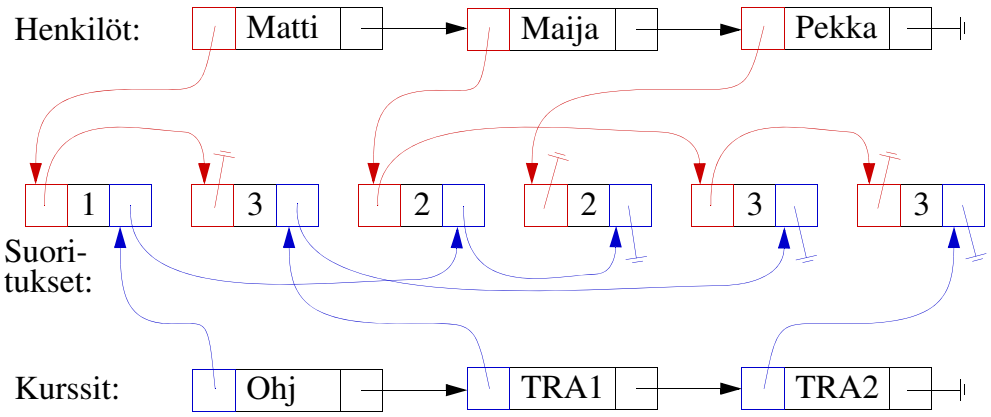
```
for (Map.Entry<K,V> e : M.entrySet()) {
    K avain = e.getKey();
    V arvo = e.getValue();
    ...
}
```

- Java8-: mahdollisuus käsitellä ja muuttaa alkioita lambda-lausekkeilla:
  – *Map.compute*()
  – *Map.forEach*((*k*, *v*) -> ...)

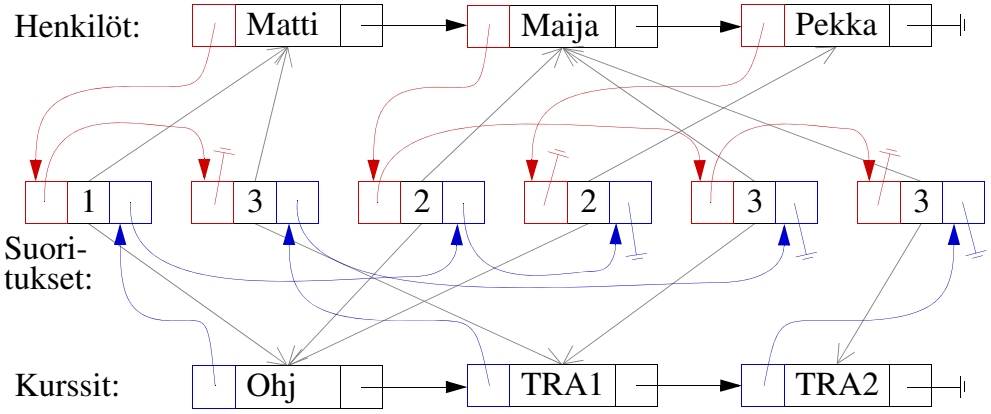
4.4 Monilista (*multilist*)

⇒ Tietorakenne relaation toteuttamiseen.

- Vastaa nopeasti esim. kysymyksiin ”mitä Matti on suorittanut”, tai ”ketkä ovat suorittaneet kurssin TRA1”. Vaatii linkin (viitteen) suorituksista henkilöihin ja kursseihin.



Kuva 4.1: Monilista, selkeyden vuoksi paluulinkit on jätetty piirtämättä



Kuva 4.2: Monilista, paluulinkit näkyvissä

- Lisäys $O(1)$, poisto $O(1)$ (jos 2-suuntaiset listat), hakukin on nopea jos lähtö- ja maalijoukot on toteutettu esim. puulla listan asemasta.
- Jos listat ovat 2-suuntaisia, niin relaatioalkiossa on 6 viitettä + hyötytieto, toteuttajan on siis syytä olla huolellinen.
- Voidaan yleistää useammankin joukon relaatioon.
  - Esim. suorituksen antanut opettaja.
    - Uutena opettajien lista.
    - Kolmas seuraaja-linkki (”seuraava tämän opettajan antama suoritus”) kuhunkin suoritusalkioon.
  - Jollei tarvita nopeaa hakua ”kaikki annetun opettajan antamat suoritukset”, riittää sijoittaa osoitin opettajaan 2-listaisen monilistan hyötytietoon.

4.5 Prioriteettijono (*priority queue*)

- Kuten listaoperaatioiden, voidaan myös joukko-operaatioiden käsittelykohtaa ja käsittelyjärjestystä rajoittaa / säätää.

Määritellään joukon alkioille prioriteetti (tärkeysjärjestys, kiireellisyysjärjestys):

- Prioriteetiltaan ”parempi”, ”tärkeämpi” otetaan käsittelyyn (poistetaan jonosta) aikaisemmin kuin prioriteetiltaan ”huonompi”.
- Prioriteetiltaan samanarvoiset poistetaan jossakin järjestyksessä, ennen kaikkia prioriteetiltaan huonompia.
- Usein (oletusarvoisesti) tärkeämmälle annetaan lukuarvoltaan pienempi prioriteetti.

Esimerkki 4.9. Ensiapupoliklinikan potilaat muodostavat prioriteettijonon: sydänkohtauksen saanut potilas ohittaa jonossa potilaan jonka sormi on poikki – riippumatta siitä, kuinka kauan sormipotilas on jo joutunut odottamaan.

Esimerkki 4.10. Prioriteettijonon suhde jonoon ja pinoon.

- prioriteettina saapumisjärjestys ⇒ JONO
- prioriteettina käänteinen saapumisjärjestys ⇒ PINO

Prioriteetin määrääminen

- Jotta poistamisjärjestys voitaisiin yksiselitteisesti määrätä, on kullekin jonossa olevalle alkiolle voitava joko laskea lukuarvoinen (vertailtava) prioriteettiarvo, tai alkioita on voitava vertailla toisiinsa järjestyksen selville saamiseksi.

Vaihtoehtoja:

1. Prioriteetti annetaan reaali- tai kokonaislukuna kullekin prioriteettijonoon vietävälle alkiolle erikseen.
2. Prioriteetti lasketaan kullekin jonoon vietävälle alkiolle erillisellä elementin tyypin huomioivalla *priority*(x) -funktiolla (joka palauttaa reaali- tai kokonaislukuarvon).
3. Prioriteetti annetaan prioriteettifunktiona *compare*(E $x1$, E $x2$) tai *x1.compare*(E $x2$) jota käytetään alkioiden järjestyksen määräämiseen.

- Näistä 1 on toteuttajalle selkein ja helpoin, mutta ehkä vähiten joustava. Tätä käytämme myös TR-kirjaston *AssignablePriorityQueue*:ssa.
- Javan *PriorityQueue* -luokka tukee vaihtoehtoa 3.
  - Käytännössä Javan *PriorityQueue* edellyttää joko alkiotyypin toteuttavan *Comparable* -rajapinnan tai erillisen vertailijan (*Comparator*) antamisen prioriteettia luotaessa.

Muuttuva prioriteetti?

- Joskus on tarpeen pystyä muuttamaan jonossa jo olevan alkion prioriteettia (nostamaan lähemmäs huippua tms).
- Tällöin on myös oltava keino yksilöidä jo jonossa olevia alkioita (mihin perusoperaatiot eivät taivu).
  - On itseasiassa yksilöitävä alkion tallettavia solmuja jotta niiden järjestystä prioriteettijonon talletusrakenteessa voidaan muuttaa prioriteetin muuttuessa.
- Javan *PriorityQueue*:n sisältämiä alkioita ei saa muuttaa siten, että prioriteetti muuttuu.
- Kts. Tra-kirjaston *AdjustablePriorityQueue*.
  - Palataan tähän prioriteettijonon toteutuksen yhteydessä (s. 102).

Prioriteettijonon operaatiot

- Kuten jonossakin, tarvitaan lähinnä lisäys jonoon (”jonnekkin sinne”) ja tärkeimmän haku&poisto (”kiireellisyysjonon keulilta”).
- Lisäyksessä annetaan elementin lisäksi ko. elementin prioriteetti tavalla tai toisella.
- Käsittelyssä oleva (jonon keulilla oleva) alkio on prioriteetiltaan (lukuarvo) pienin, niinpä operaatiot kohdistuvat prioriteetiltaan pienimpään alkioon.
- Prioriteettijonossa voi (joukosta poiketen) olla sama alkio useasti, tai ainakin usea samapriori-teettinen alkio.

– Nämä samanarvoiset saadaan jonosta mieluiten lisäysjärjestyksessä (tai sitten satunnaisessa järjestyksessä).

Määritelmä 4.11. Prioriteettijonon operaatiot *java.util.PriorityQueue*:ssa: (*PriorityQueue*<*E*> *P*, *E x*. *E* toteuttaa liittymän *Comparable*)

1. PriorityQueue<E> PriorityQueue<E>()
   Luo uuden tyhjän prioriteettijonon.
2. boolean P.isEmpty()
   Palauttaa *true* jos prioriteettijonossa *P* ei ole yhtään alkiota, muuten *false*.
3. boolean P.add(x)
   Lisää prioriteettijonoon *P* alkion *x*.
4. E P.poll()
   Poistaa ja palauttaa prioriteettijonosta *P* prioriteettijärjestyksessä ensimmäisen alkion.
5. E P.peek()
   Palauttaa prioriteettijonon *P* prioriteettijärjestyksessä ensimmäisen alkion.

Prioriteettijono voidaan toteuttaa monella eri tavalla. Tehokkaasti keolla (eli kasalla) toteutettuna *add* ja *poll* ovat $O(\log n)$ (missä n on jonossa kulloinkin olevien alkioiden määrä) ja muut $O(1)$ (kts. "Prioriteettijonon toteutus" s. 102). Huomaa, että tasapainotetulla sisäjärjestetyllä binääripuulla päästään samaan kertaluokkaan, mutta vakiokerroin on selvästi suurempi.

Esimerkki 4.12. Niin sanottu kekojärjestäminen (*heapsort*) toteutetaan prioriteettijonon avulla. Jos järjestettävät alkiot ovat aluksi listassa *L*, jonka halutaan järjestämisen päätyttyä sisältävän alkiot järjestyksessä, on järjestämisalgoritmi suoraviivainen: alkiot prioriteettijonoon ja takaisin. [*HeapSortEsim.java*]

```
public static void heapSort0(Collection A) {
   Iterator i = A.iterator();
   PriorityQueue P = new PriorityQueue();
   // alkiot prioriteettijonoon
   while (i.hasNext()) {
      P.add(i.next());
      i.remove();          // tai lopuksi kerralla .clear()
   }
   // alkiot takaisin kokoelmaan
   while (! P.isEmpty())
      A.add(P.poll());
}
```

Tämä on toimiva järjestämisalgoritmi, mutta varsinaisessa kekojärjestämisessä (s. 104) hyödynnetään hieman tarkemmin prioriteettijonon kekototeutusta eikä tarvita aputilaa.

Mikäli kokoelman *A* operaatiot, erityisesti *remove* ja *add* lopusta/loppuun, ovat tehokkaita, on algoritmin aikavaativuus $O(n \log n)$, kun järjestettävänä on n alkiota. Ellei kokoelman loppuun lisääminen onnistu tehokkaasti, voidaan prioriteettifunktio määritellä niin, että jonosta poistetaankin suurin alkio, jolloin alkioita takaisin listaan vietäessä lisäyskohta on aina kokoelman alussa. Kekojärjestämisen tehokkaaseen (vakioaputila) toteutukseen palaamme myöhemmin (s. 104).

Esimerkki 4.13. (Listan) *k*:nneksi suurin alkio. Ryhdytään viemään alkioita prioriteettijonoon. Kun jonossa on $k+1$ alkiota, niistä pienin ei voi olla *k*:nneksi suurin, joten se poistetaan (*poll*). Jatketaan *add*+*poll* operaatioita kunnes lista läpikäyty. Nyt *k*:nneksi suurin alkio on prioriteettijonossa keulilla (*peek*). Aikavaativuus? [HT]

Prioriteettijonon algoritmisia käyttökohteita

- Optimointi (uusia vaihtoehtoja, paras(t) senhetkisistä jatkokäsittelyyn).
- Painotettu haku.
- Osittainen järjestäminen, valinta.

4.6 Laukku

⇒ Sama alkio useasti kokoelmassa.

- Yleinen joukkomalli (sanakirja, kuvaus) sallii saman alkion esiintyvän joukossa vain yhtenä ilmentymänä kerrallaan.
- Prioriteettijonoissa on usein sallittua viedä sama alkio (tai saman prioriteetin omaava alkio) jonoon useita kertoja.

Jos saman alkion moninkertainen esiintyminen on tarpeellista, se voidaan sallia!

- Tällöin kysessä on laukku eli monijoukko (*bag*, *multiset*).

Laukku abstraktina tietotyyppinä:

- Kuten joukko (operaatiovalikoima tarpeen mukaan).
- *add*(x) vie alkion x laukkuun riippumatta siitä onko alkiota laukussa ennestään vai ei.
- *remove*(x) poistaa kaikki alkion x esiintymät.
- *remove*(x, i) poistaa enintään i esiintymää.

⇒ Java API ei sisällä laukun toteutusta.

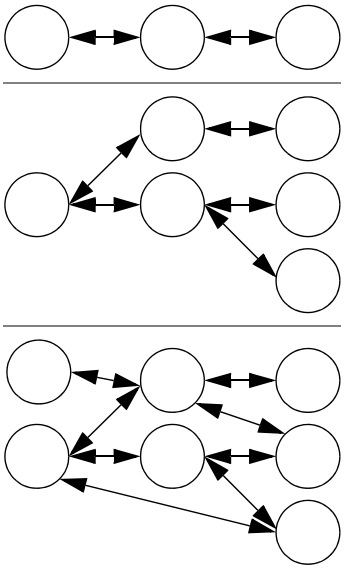
- Sen voi toteuttaa itse, mutta täysi toteutus on jo melko työläs.
- Valmiita kolmannen osapuolen kirjastoja löytyy netistä, esim *org.apache.commons.collections.Bag*, `https://commons.apache.org/proper/commons-collections/javadocs/api-4.4/org/apache/commons/collections4/Bag.html`.
- Jos alkiot ovat täysin identtisiä (ei tarvitse oikeasti tallettaa useita olioita), kuvauksella (*Map*<*E*, *Integer*>) pärjää mainiosti kun kokonaislukuarvoa käyttää ko. elementtien lukumäärän ylläpitämiseen.
- Vaihtoehtoisesti voidaan käyttää joukkoa ja sopivasti rakennettua alkiotyypin *compareTo*() metodia.
  - Samanlaiset alkiot tulevat peräkkäin johonkin yksiselitteiseen järjestykseen.
  - ... *All*() operaatiot eivät kuitenkaan toimi sellaisenaan.

5 Verkot

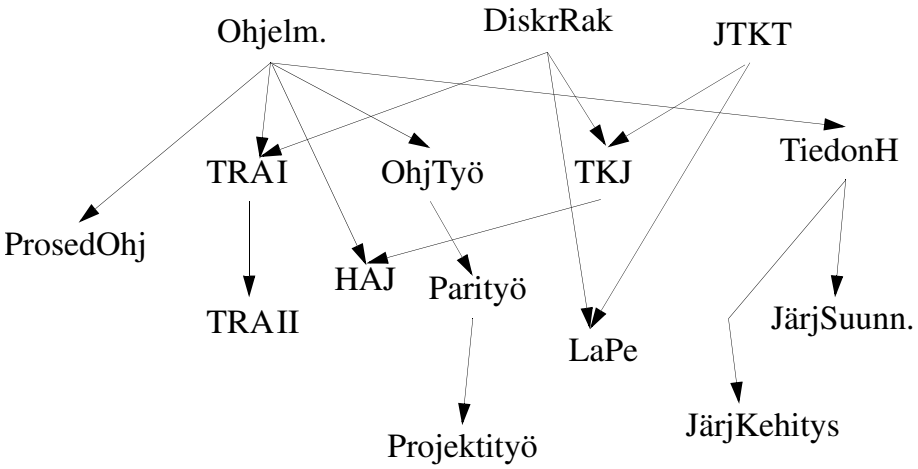
- Lista:
  - alkiolla yksi seuraaja
  - alkiolla yksi edeltäjä
- Puu:
  - alkiolla yksi edeltäjä (vanhempi)
  - alkiolla monta seuraajaa (lapset)
- Entä kun tarvitaan
  - monta edeltäjää ja
  - monta seuraajaa?

⇒ (Suunnattu) verkko ((*directed*) *graph*)

- Jollei edeltäjä/seuraaja -suunnalla ole merkitystä, niin käytetään suuntaamatonta verkkoa (kts. s. 66) (*undirected*).



Esimerkki 5.1. Tkt:n LuK:n kurssijärjestys (vanha):



Käyttökohteita

- (Tieto/henkilö/materiaali/sähkö/tms)liikenteen mallinnus.
  - Tietoliikenneverkko, tieverkko, virtapiirit, prosessikaaviot.
  - Reititys, liikennesuunnittelu, virtasuunnittelu.
- Relaatiokaaviot, luokkakaaviot, olio-suhdekaaviot, käyttötapauskaaviot, organisaatiokaaviot, prosessi/projektikaaviot.
- Hypertekstit, viittaukset.
- Kemialliset, fysikaaliset, tekniset, biologiset, sosiologiset, taloudelliset verkostot sekä niiden suhteet ja vuorovaikutukset.

Algoritmit osin sovellusalakohtaisia

- Tällä kurssilla vain peruskäsitteet.
- TRA II -kurssilla käsitellään tarkemmin abstrakti tietotyyppi verkko, sen toteutus ja joitakin verkkoalgoritmeja.
- Datatiedekursseilla sovelluksia.

Käsitteitä

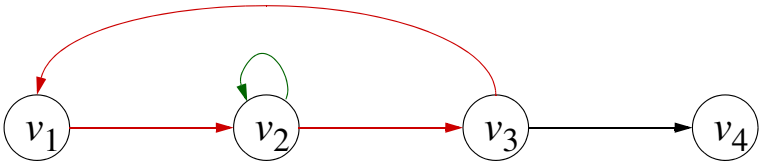
Suunnattu verkko $G=(V,E)$

- V on solmujen (*vertex*, *node*) joukko.
  - verkon perusalkio
- E on (suunnattujen) kaarten (*edge*) joukko ($\approx$ relaatio)
  - Kahden solmun välinen suhde (naapuruus)
  - Suunnattu kaari merkitään (u,v) tai $u \to v$
    - u on lähtösolmu
    - v on päätesolmu (tai maalisolmu)
  - Päätesolmu v on lähtösolmun u naapuri,
    - mutta ei päinvastoin, ellei myös $(v,u) \in E$.
    - $(u,v) \neq (v,u)$
- V ja E aina äärellisiä $\Rightarrow$ G äärellinen.
  - Voi olla $V \neq \emptyset$ ja $E = \emptyset$, mutta
  - jos $V = \emptyset$, niin myös $E = \emptyset$.
- Solmulla voi olla mielivaltainen määrä naapureita.
  - Solmu voi olla itsensä naapuri (kaari (u,u), silmukka (*loop*))
  - Kaaret voivat (tarvittaessa) olla moninkertaisia, eli voi olla useampi samansuuntainen kaari saman solmuparin välillä. Myös useampi silmukka.
    - Näitä voidaan joskus/usein rajoittaa, riippuen sovelluksesta.

Määritelmä 5.2. Polku (*path*)

- Suunnatun verkon solmujen jono $(v_1,v_2,\ldots,v_n)$ on *polku*), jos (v_i,v_{i+1}) on verkon kaari kaikille $i=1,\ldots,n-1$.
  - Polun pituus on tällöin $n-1$ (kaarten määrä).
- Polku on *yksinkertainen*, jos $v_i \neq v_j$ kun $i \neq j$ tai ($i=1$ ja $j=n$).
- Yksinkertainen polku, jossa $v_1 = v_n$ on *kehä* (*cycle*).

Esimerkki 5.3. Nelisolmuinen verkko

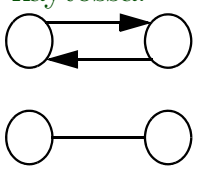


- $V = \{v_1,v_2,v_3,v_4\}$.
- $E = \{(v_1,v_2),(v_2,v_2),(v_2,v_3),(v_3,v_1),(v_3,v_4)\}$.
- Kehät (v_1,v_2,v_3,v_1) ja (v_2,v_2).
- Polkuja mm. $(v_2,v_2,v_3,v_1,v_2,v_3,v_4)$ ja (v_3,v_1).

Suuntaamattomat verkot

$\Rightarrow$ Suuntaamattomassa verkossa kaaren molemmat suunnat ovat aina käytössä.

- Soveltuu vain sovelluksiin joissa yhteys on aina symmetrinen.
- Hieman helpompi hahmottaa ja piirtää.
- Hieman hankalampi toteuttaa.
- Matematiikassa (verkkoteoriassa) se perinteisempi vaihtoehto (oletus).



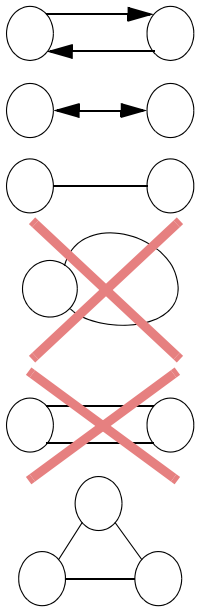
- Pääosa suunnatun verkon algoritmeista soveltuu (jokseenkin) sellaisenaan suuntaamattomallekin verkolle.

  - Esim. lyhyimmät polut.
  - Kaarta seurattuamme meidän on kuitenkin varmistuttava, ettemme vahingossa palaa samaa kaarta takaisin.

Suuntaamattoman verkon määritelmiä

⇒ Kuten suunnattu verkko paitsi:

- $(u, v) = (v, u)$
  - Kaarella ei ole suuntaa.
  - Naapuruussuhde on siis aina molemminpuolinen.
  - Kaari (ja siten polku) voidaan kulkea molempiin suuntiin.
- Solmusta itseensä ei (yleensä) saa olla kaarta (ts. silmukkaa).
- Solmujen välillä ei (yleensä) saa olla useampaa kuin yksi kaari.
- Koska kehä ei saa sisältää samaa kaarta kahdesti (ja jos kaksi edellistä kohtaa ovat voimassa), niin kehässä on vähintään kolme solmua.



Hyötytieto verkossa

- Sekä solmut, että kaaret voidaan varustaa hyötytiedolla (nimiöllä, viittauksella), tarpeen (sovelluksen) mukaan.
  - Esim. kaupungin, tien, lennon, reitittimen, kaapelin, jne. tunnus.
  - Solmuissa ja kaarissa on usein eri tyyppinen data.
- Lisäksi solmuille ja kaarille voidaan antaa paino (hinta) jonka mukaan tuloksia lasketaan.
  - Esim. (kaaren) etäisyys, aika, hinta, läpäisykapasiteetti, todennäköisyys, virta, jne
  - Joskus myös solmuilla voi olla lukuarvo, esim. (solmun) hinta, asukasluku, tilavuus, lämpötila, aika, jne.
- Tarvittaessa voidaan myös käyttää merkintöjä mm. jo käsitellyistä solmuista/kaarista. Näitä nimitetään yleensä väreiksi.
- Käyttämässämme tietorakennekirjastossa ”hyötytieto” on merkkijono, paino on reaaliluku ja väri on lueteltu tyyppi (kokonaisluku).
  - Lisäksi on käytössä kokonaislukutyyppinen indeksi jota voidaan käyttää esimerkiksi solmujen numerointiin.
  - Jos/kun sovelluksessa tarvitaan muuta tietoa (yleensä viite), voidaan periyttämällä lisätä jäseniä.

Verkko abstraktina tietotyyppinä

Verkko muodostuu solmujen ja kaarten joukoista.

- Käytetään (sovellettuja) joukko-operaatioita niiden käsittelyyn.
  - Solmut ovat selkeästi joukko.
  - Kaaret ovat (solmuparien) relaatio.

Mitä operaatioita tarvitaan?

- verkon luonti
- lisäys, poisto (solmun ja kaaren erikseen)
- olemassaolon tutkiminen, nimiön haku (muuttaminen)
  - painojen ja värien asetus ja tutkinta
  - solmun ja kaaren erikseen
- usein myös läpikäyntejä
  - kaikki solmut
  - kaikki kaaret
  - solmun naapurit (ja solmusta lähtevät kaaret)

Kaikkien kaarten joukkoa ei tarvitse toteuttaa erikseen koska kaaret voidaan selvittää naapuruussuhteita (naapurien joukkoja) tarkastelemalla.

Verkko-ongelmia/algoritmeja

⇒ Eräs monipuolisimmista algoritmiikan alueista – ongelmia ja algoritmeja on ”loputtomasti”.

- Polkujen pituudet, olemassaolot, maksimi virta, erilliset polut
- Verkkojen numerointi, isomorfisuus (vertailu)
- Verkon yhtenäisyys, vahva yhtenäisyys, (vahvasti)yhtenäiset komponentit, k-yhtenäisyys, leikkauskohdat, jne
- Syklien/klikkien etsintä | poisto | muodostaminen
- Virittävät puut, peitteet, väritykset
- Eulerin-, Hamiltonin, jne läpikäynnit (kauppamatkustaja yms.)
- Verkon täydentäminen halutun ominaisuuden saavuttamiseksi
- Verkon ositukset / ryhmittelyt (klustereiden minimi/tasa-{paino | TS | välikaaret | jne}
- Verkon piirto/sijoittelu tasoon | tilaan | pallopinnalle | torukselle jne
- Topologinen järjestäminen
- k-jakoisuus, sovitukset
- Eri tyyppisillä verkoilla (solmut d-ulottuvuudessa, erilaiset etäisyysmetriikat, negatiivipainoiset, epäyhtenäiset, syklittömät, jne).
- Kts. verkkoteorian oppikirjat, algoritmikirjat

⇒ Jatkuu kurssilla TRA II

6 Järjestäminen eli lajittelu

⇒ Järjestämisessä (*sort*) järjestetään (peräkkäisen) kokoelman alkiot lineaariseen järjestykseen.

- Järjestäminen on parempi termi kuin myös yleisesti käytetty lajittelu.
  - Sanan ”lajittelu” voi sekoittaa luokitteluun eli jaotteluun (joka on taas eri ongelma).
- Useimmiten järjestetyt alkiot jätetään samaan kokoelmaan kuin mikä oli syöttenä.
- Sisäisessä järjestämisessä kaikki alkiot mahtuvat kerralla keskusmuistiin.
  - Ulkoisessa järjestämisessä eivät, tarvitaan massamuistia apuna (TRA II).

Kuinka tärkeää järjestäminen on?

- Usein suuria tietomääriä säilytetään ”järjestettynä” (indeksoituna); koko datan uudelleenjärjestämistä tarvitaan harvoin.
- (Oikein tehty) järjestäminen on suhteellisen nopeaa (ainakin keskusmuistissa).
- Asiantuntijajärjestelmissä arvotetaan suuri joukko mahdollisia ratkaisuja ja valitaan niistä parhaat (jatkokäsittelyyn).
  - Tietomäärät suuria.
- Jotkin reititysprotokollat järjestävät reititettävät paketit kohteen mukaan (järjestämisen on oltava nopeaa (rinnakkaista)).
- Jotkin lineaarialgebran algoritmit tarkentuvat kun käytetään sopivasti järjestettyä syötettä.
- Joskus muistinkäytön optimointi hyötyy pyyntöjen järjestämisestä (hidasta (massa)muistia käytettäessä).
- Järjestetty taulukko/lista toimii prioriteettijonona (kun kaikki alkiot lisätään kerralla).
- Tehokas järjestämisalgoritmi on tarjolla (liki) kaikissa API-kirjastoissa.

⇒ Ei kovin tärkeää, mutta mielenkiintoista ja opettavaista.

- Tässä tarkastellaan erilaisia algoritmeja samaan ongelmaan, opitaan erilaisia ratkaisutapoja (algoritmistrategioita), teema jatkuu TRA II ja ASA -kursseilla.

6.1 Sisäinen järjestäminen

- Kun alkiot mahtuvat keskusmuistiin, alkioita voidaan käsitellä mielivaltaisessa järjestyksessä.
  - Taulukossa järjestäminen on kaikkein helpointa:
    - suora pääsy indeksiin ($get(i)$ ja $set(i, x)$)
    - lisäys keskelle on työlästä, eli sitä ei kannata tehdä
    - useimmiten järjestetään vaihtelemalla alkioita keskenään (tai kopioimalla alkioita tyhjään kohtaan)
    - tai käytetään sopivaa apurakennetta (yleensä toinen taulukko) väliaikaisena talletuspaikkana osalle/kaikille alkioille (tällöin aputilan tarve yleensä lineaarinen)
  - Listankin järjestäminen onnistuu nopeasti:
    - vain peräkkäinen käsittely (seuraaja/edeltäjä) on tehokasta
    - lisäys keskelle ja listan jakaminen ovat helppoja (kunhan meillä on viittaus listasolmuun)
    - aputilaa ei yleensä tarvita, kukin alkio on kerrallaan vain yhdessä listassa
    - Jopa jonon järjestäminen onnistuu nopeasti pelkällä peräkkäisläpikäynnillä käyttäen kahta apujonoa/pakkaa.

Järjestämisavain (*key*)

- Kasvava järjestys päätetään järjestämisavaimen perusteella.
  - Avaimen ei tarvitse olla koko alkio.
  - Joko kokonaiset alkiot järjestetään avainten mukaiseen järjestykseen, tai
  - luodaan erillinen indeksi avainten mukaan josta on linkki kokonaisiin alkioihin.
  - Joskus tarvitaan useita järjestämisavaimia, esim:
    - ensisijaisesti sukunimen perusteella
    - toissijaisesti etunimen perusteella
    - lopulta puhelinnumeron perusteella
  - Aivan täydellistä järjestystä ei välttämättä tarvita.

    - Jos kahdella alkiolla ei ole keskenään luonnollista järjestystä, kyseiset alkiot yleensä pyritään jättämään siihen järjestykseen kuin missä ne olivat ennen järjestämistä, ns. vakaa järjestäminen (*stable sort*).
    - 5 3 3 6 → 3 3 5 6
    - Näin voidaan samaa järjestämisalgoritmia kutsua toistuvasti aluksi vähiten merkitsevällä (osa)avaimella ja sitten merkitsevämmillä (kts. s. 76).
  - Vertailuun perustuvassa järjestämisessä riittää, että meillä on vertailuoperaatio $e1.compareTo(e2)$ joka palauttaa tiedon siitä, edeltääkö alkio $e1$ alkiota $e2$.
    - Tai vertailija (*Comparator*) jolla on metodi $compare(e1, e2)$.

Sisäisen järjestämisen aikavaativuus

- Lasketaan askeleet kuten aikavaativuusanalyysissä yleensä.
- Jos vertailu on hyvin raskasta (pitkä avain tms), huomioidaan se.
- Vertailujen määrä on hyvä mittari, jollei tehdä jotain muuta erikoista.
- Siirtojen määrä, jos alkiot ovat suuria ja jos niitä siirrellään.
  - Jos alkiot ovat isoja, niitä käsitellään Javassa (ja yleensä muutenkin) viittauksilla jotka ovat vakiokokoisia.

⇒ Yleensä järjestämme siis viittaustaulukon!

6.2 Yksinkertaisia järjestämisalgoritmeja

Kuplajärjestäminen (*bubble sort*)

- Vaihdetaan keskenään väärässä järjestyksessä olevia naapureita päikseen toistuvasti kokoelmaa läpikäyden.
- Toimii listallekin. Kts. *LajitteluEsim.java*.
- $O(n^2)$

Upotusjärjestäminen (*insertion sort*)

- Viedään kukin alkio paikalleen taulukkoon.
  - Vaihtoehtoisesti uuteen aluksi tyhjään taulukkoon (lisäysjärjestäminen).
- Taulukolla tilan tekeminen uuteen taulukkoon on hidasta, listalla oikean paikan etsiminen.
- $O(n^2)$

Valintajärjestäminen (*selection sort*)

```
for (int i = 0; i < n−1; i++)                    1
  swap(A[i], min(A[i], ... , A[n−1]));           2
```

- Jäljelläolevista pienin paikalleen.
- $O(n^2)$

Useimpien $O(n^2)$ järjestämisalgoritmien aikavaativuus on sama parhaassa ja pahimmassa tapauksessa

- Valitsemalla läpikäynti sopivasti, voidaan parhaassa tapauksessa (valmiiksi järjestetty tms.) päästä $O(n)$ aikaan.
- Jos tiedetään syötteen olevan lähes järjestyksessä, voidaan valita jokin sopiva erikoisjärjestäminen (HT).
  - Vain k alkiota lopussa/alussa epäjärjestyksessä.
  - Kukin alkio vain k sijainnin päässä oikeasta paikasta.
  - jne

  - Timsort: hyödyntää valmiita järjestettyjä jaksoja, lisäysjärjestämisellä tekee (lyhyitä) uusia järjestettyjä jaksoja, lomitusjärjestäminen lopuksi.

Yleisessä tapauksessa aikavaativuudeltaan $O(n^2)$ oleva järjestämisalgoritmit ovat tehottomia jos syötettä on vähänkään enemmän, käytetään mieluummin $O(n\log n)$ algoritmeja:

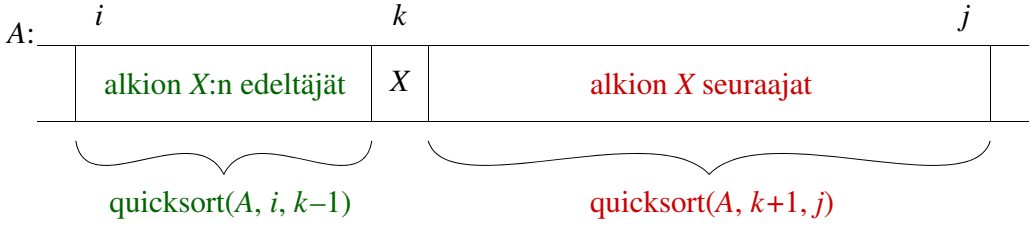
Kuva 6.1: Yleisimmät järjestämisaikavaativuudet

n	$n\log n$	n^2
10	40	100
1 000	10 000	1 000 000
1 000 000	20 000 000	**1 000** 000 000 000
1 000 000 000	**30 0**00 000 000	**1 000 000 000 0**00 000 000

- On olemassa myös järjestämisalgoritmeja, joiden (keskimääräinen) aikavaativuus on $O(n^k)$, missä k on murtoluku väliltä $(1, 2]$, esim. Shellsort.
- Muistetaan, että $n\log n = o(n^{1+\epsilon})$
  - Shellsort:sta on myös $O(n\log^2 n)$ versio [Pratt].

6.3 Pikajärjestäminen (*quick sort*) (C.A.R. Hoare)

- Kussakin vaiheessa: aluksi jaotellaan taulukon alkiot kahteen osaan taulukosta valitun jakoalkion (*pivot element*) mukaan.
  - Alkuosaan siirretään jakoalkiota pienemmät alkiot.
  - Loppuosaan siirretään jakoalkiota suuremmat alkiot.
- Jaottelun jälkeen jätetään jakoalkio paikalleen ja järjestetään rekursiivisesti alkuosa ja loppuosa.



Kuva 6.2: Pikajärjestämisen rekursio *partition*-aliohjelman **jälkeen** (k = *jakokohta*).

⇒ Hajoita-ja-hallitse.

Algoritmi 6.1. Pikajärjestäminen.

```
public static <E extends Comparable<? super E>> void quicksort(E[ ] A, int i, int j) {
    if (i < j) {
        int jako = partition(A, i, j);          // jaottelu
        quicksort(A, i, jako−1);                // alkuosa
        quicksort(A, jako+1, j);                // loppuosa
    }
}
```

Koko taulukko järjestetään kutsumalla *quicksort*$(A, 0, A.length-1)$.

Jaottelualgoritmi

- Ensin valitaan jakoalkio (s. 72), sitten siirrellään alkioita sen mukaan.

- Aliohjelmassa *partition* otetaan aluksi jakoalkio talteen jotta vaihdoille saadaan tyhjä paikka taulukkoon.
  - Jos/kun jakoalkio valittiin muualta kuin alusta, niin vaihdetaan vielä ensimmäinen alkio jakoalkion entiselle paikalle jotta tyhjä paikka saadaan taulukon alkuun.
- Tarkastellaan osataulukkoa $A[i..j]$ vuorotellen eri päistä (indekseistä i ja j) kunnes jako on tehty.
  - Kun loppupäästä löytyy jakoalkiota pienempi alkio, siirretään se taulukon alkuosaan tyhjälle paikalle. Loppuun jää ”tyhjä” paikka.
  - Kun alkupäästä löytyy jakoalkiota suurempi alkio, siirretään se taulukon loppuosaan tyhjälle paikalle. Alkuun jää ”tyhjä” paikka.
- Lopuksi jakoalkio viedään viimeiseksi jääneeseen tyhjään paikkaan lopulliselle paikalleen.
  - Sitä pienemmät on siirretty alkuun, suuremmat loppuun.
  - Osataulukko on siis jaoteltu pieniin ja isoihin, nämä voidaan järjestellä rekursiivisesti toisistaan riippumatta.
  - Jaottelualgoritmi palauttaa jakoalkion lopulliseksi muodostuneen sijainnin itse pikajärjestämiselle.
- Tästä on myös muita variantteja, mutta periaate ja lopputulos on sama.
  - Sensijaan, että jakoalkiota ylläpidetään erillisessä apumuuttujassa, voidaan muistaa vain jakoalkion kulloinenkin sijanti ja vaihdella alkioita keskenään sen mukaan kumpaan päähän ne kuuluvat.

Algoritmi 6.2. Pikajärjestämisen jaottelu:

```
public static <E extends Comparable<? super E>> int partition(E[ ] A, int i, int j) {
    E jakoalkio = A[arvottuJakoalkionIndeksi];          // jakoalkio talteen
    A[arvottuJakoalkionIndeksi] = A[i];                 // tyhjä tila alkuun
    while (i < j) {                 // toistetaan kunnes i ja j törmäävät
        // etsitään lopusta jakoalkiota pienempi
        while ((i < j) && (jakoalkio.compareTo(A[j]) < 0))
            j--;
        A[i] = A[j];
        // etsitään alusta jakoalkiota suurempi tai yhtäsuuri
        while ((i < j) && (jakoalkio.compareTo(A[i]) >= 0))
            i++;
        A[j] = A[i];
    }
    // jakoalkio paikalleen ja palautetaan sijainti
    A[i] = jakoalkio;
    return i;
}
```

Jakoalkion valinta

- Jakoalkion valinta ei vaikuta pikajärjestämisen oikeellisuuteen, mutta se vaikuttaa aikavaativuuteen (s. 73).
- Aikavaativuuden kannalta olisi parasta, jos osataulukot olisivat keskenään samankokoiset.
  - Tällöin jakoalkio (keskusalkio) olisi oltava osataulukon mediaani.
  - Valitettavasti mediaanin haku on melko työlästä, kts. alempana.
- Helpointa on arpoa jokin alkio jakoalkioksi.
  - Arvottu alkio, keskimmäinen alkio, (ensimmäinen alkio).
- Jakoalkion valintaan voidaan jopa käyttää hieman aikaa.
  - Jakoalkio valitaan kerran kullakin rekursiotasolla, *partition*() vie aina $O(n)$.

  - (mediaani | keskiarvo) k:sta (satunnaisesta | tasavälein) valitusta alkiosta.
    * Keskiarvo on mahdollista laskea vaikka kaikista alkioista (jos ne ovat lukuarvoisia).
  - Parantavat keskimääräistä toimintaa [reaalimaailman syötteillä].
  - Aina on mahdollista löytää harvinainen erikoistapaus, jolla ko. menetelmä ei toimi hyvin.

Jakoarvo (esim. keskiarvo) jakoalkion sijaan

- Jaotellaan taulukko kahtia jakoarvon kahdelle puolelle.
- Jakoalkiota ei jää jäljelle, rekursio käynnistetään osataulukoille $i..jako$ ja $jako{+}1..j$.

Aikavaativuus

- *partition* on selvästi $O(n)$, missä $n{=}j{-}i{+}1$.
  - Joka askeleella joko j vähenee tai i kasvaa.
- Jos jako osuisi aina tasan puoleen väliin, olisi koko algoritmi

$$T_{best}(n) = \begin{cases} O(n)+2\times T\left(\frac{n}{2}\right) & \text{, kun } n>1 \\ O(1) & \text{, kun } n\leq 1 \end{cases}$$
$$= \underbrace{n+2\left(\tfrac{n}{2}\right)+4\left(\tfrac{n}{4}\right)+\cdots+n\times 1}_{\log n \text{ kpl}} \qquad = O(n\log n) \tag{6.1}$$

- Jos taas jako osuisi aina huonosti (pienin alkio), olisi

$$T(n) = \begin{cases} O(n)+T(n-1)+O(1) & \text{, kun } n>1 \\ O(1) & \text{, kun } n\leq 1 \end{cases}$$
$$= \underbrace{n+(n-1)+(n-2)+\cdots+1}_{n \text{ kpl}} \qquad = O(n^2) \tag{6.2}$$

- Keskimääräisen tapauksen (satunnaisella syötteellä) analysointi on vaikeampaa, todetaan sen olevan: $O(n\log n)$.
- Aputilan tarve tulee rekursiosta (rekursiotasojen määrä tai osatehtävien pinosta/jonosta), parhaimmillaan (ja keskimääräisesti) $O(\log n)$, pahimmillaan $O(n)$.

Lisävirityksiä

- Pienet tehtävät (esim. n $\leq$ 9) rekursion edetessä järjestetään vaikka kuplajärjestämisellä (HT)
- Usea jakokohta: hieman vähemmän rekursiotasoja ja siirtoja.

Jakoalkion valinta käytännössä

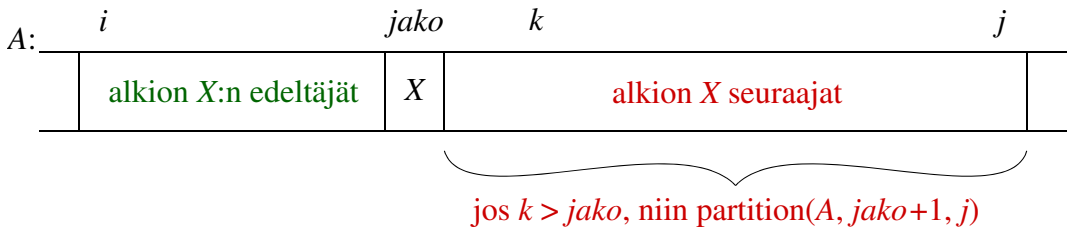
- 1. alkion saa valita jakoalkioksi vain opetustarkoituksissa, ei koskaan todellisille syötteille.
  - Oikeat syötteet ovat usein valmiiksi (melkein) järjestyksessä ja aikavaativuus olisi neliöllinen!
- Mediaani kolmesta (tai 9) satunnaisesta (tai tasavälein valitusta), tai edes satunnainen alkio.
- Kts. esim Java API *java.util.Arrays.sort*(*int*[]) lähdekoodi ($\leq$ JavaSE6).
  - Java SE7 vaihtoi Dual-Pivot Quicksort -versioon.
  - Olioiden järjestäminen tehdään viritetyllä lomitusjärjestämisellä (Timsort) (hyödyntää valmiita järjestyksiä).

Valinta (mediaani) (*selection, median*)

$\Rightarrow$ k:nneksi pienin (tai suurin) alkio

- Järjestämisen jälkeen löytyisi indeksistä k, mutta vie ajan $O(n\log n)$ (tai $O(kn)$).
- Prioriteettijonolla: $O(n\log k)$ (kts. esimerkki 4.13, s. 63)

Pikavalinta (*quickselect*): hyödynnetään pikajärjestämisen *partition* -aliohjelmaa:



Kuva 6.3: *partition* valinnan apuna.

- *partition* -aliohjelman suorituksen jälkeen tiedämme kummalla puolella jakokohtaa (*jako*) valinta (k:nneksi pienin alkio) on.
- Toistetaan jaottelua rekursiivisesti **vain** sille puolelle jossa k on.
- Haettava alue lyhenee jokaisella kerralla parhaassa tapauksessa puoleen, mutta vähintään yhdellä.
- Jos jako osuisi aina puoliväliin, niin:

$$T_{best}(n) = \begin{cases} O(n)+T\left(\frac{n}{2}\right) & , \text{kun } n>1 \\ O(1) & , \text{kun } n\leq 1 \end{cases}$$
$$= \underbrace{n+\tfrac{n}{2}+\tfrac{n}{4}+\tfrac{n}{8}+\cdots+2+1}_{\log n \text{ kpl}} \quad = 2n-1 \;=\; O(n) \tag{6.3}$$

- Jos taas jako osuu huonosti, aikavaativuus on $O(n^2)$ kuten pikajärjestämiselläkin.
- Jakoalkio kannattaa ehdottomasti valita esim. 3:n mediaani -menetelmällä, varsinkin syötteen ollessa vielä suuri.
- Sopivasti mediaanit valitsemalla (mediaani 5-nippujen mediaaneista) voimme varmistaa lineaarisen aikavaativuuden (suurehkolla vakiokertoimella) (kts. esim. Weiss s. 374).
- Huomaa, että tämä algoritmi sotkee syötteensä alkioiden järjestyksen, joten ota ensin kopio jos/kun se ei ole hyväksyttävää.

6.4 Kekojärjestäminen (*heap sort*)

- Alkiot prioriteettijonoon ja sieltä pois.
- Kts. esimerkki 4.12 (s. 63).
- $O(n\log n)$ sekä parhaassa, että pahimmassa tapauksessa.
- Pienellä muutoksella k pienintä alkiota ajassa $O(n\log k)$.
- Pienellä muutoksella järjestäminen syötteestä jossa alkiot ovat korkeintaan k aseman päässä oikeasta kohdasta $O(n\log k)$ (HT).
- Vain k viimeistä alkiota väärässä paikassa: $O(n+k\log k)$.
  - Jos $k\leq n/\log n$, niin $O(n)$. (HT)
- Varsinainen tehokas, vakioaputilainen kekojärjestäminen perustuu järjestettävän taulukon hyödyntämiseen prioriteettijonon kekototeutuksessa. Kts. ”Prioriteettijonon toteutus” s. 102.

6.5 Lomitusjärjestäminen (*merge sort*)

Hajoita-ja-hallitse, mutta nyt eri järjestyksessä kuin pikajärjestämisessä.

```
public static void mergesort(Comparable A[], int alku, int loppu) {
   if (alku < loppu) {
      int k = (alku + loppu) / 2;          // jako tasan puolivälistä
      mergesort(A, alku, k);               // järjestetään puoliskot rekursiivisesti
      mergesort(A, k+1, loppu);
      merge(A, alku, k, loppu);            // lomitetaan järjestetyt puoliskot
}  }
```

Aikavaativuus

- *merge*(): $O(n)$, missä $n = \text{loppu} - \text{alku} + 1$ (HT).
- *mergesort*():

$$T(n) = \begin{cases} O(n) + 2 \times T\left(\frac{n}{2}\right) & , \text{kun } n > 1 \\ O(1) & , \text{kun } n \leq 1 \end{cases} \tag{6.4}$$
$$= O(n \log n) \quad \text{[Parhaassa ja pahimmassa tapauksessa!]}$$

- Aputila $O(n)$ suoraviivaisessa ratkaisussa, mahdollista pärjätä myös $O(\log n)$ aputilalla.
- Toimii myös listoille, sillä käy syötettä läpi vain peräkkäisessä järjestyksessä.
  - Toimii myös peräkkäistiedostoille.
  - Palataan tähän ulkoisen muistin käsittelyssä (TRA II).

6.6 Kaukalojärjestäminen ja kantalukujärjestäminen

Kaukalojärjestäminen (*bin sort*):

Avaimet (tai alkiot) ovat kokonaislukuja $0..m$ (tai sellaiseksi kuvattavia), missä m kohtuullisen kokoinen (korkeintaan samaa kertaluokkaa kuin n).

- Muodostetaan $m+1$ kaukaloa (taulukko $0..m$), $O(m)$ aika.
  - Laskentajärjestäminen (*counting sort*): Jos koko alkio on (ilmaistavissa) luku(na) $0..m$, riittää laskea kuhunkin kaukaloon tulevien alkioiden määrä ja muodostaa lukumääristä järjestetty taulukko.
  - Jos $0..m$ on vain avain, mutta lisäksi on talletettava muukin alkio, niin kaukalotaulukon alkiot ovat elementtilistoja.
- Viedään järjestettävän kokoelman kukin alkio suoraan oikeaan kaukaloon avaimen perusteella.
  - Kunhan toteutus on tehokas, kukin lisäys onnistuu vakioajassa, ja kaikkien alkioiden vienti $O(n)$ ajassa.
- Joko ketjutetaan m listaa yhdeksi listaksi ($O(m)$ aika) tai kopioidaan alkiot listoista takaisin alkuperäiseen kokoelmaan ($O(m+n)$ aika).
- Koko algoritmi $O(m+n)$ aika ja aputila.

⇒ Huomaa: ei yhtään vertailua!

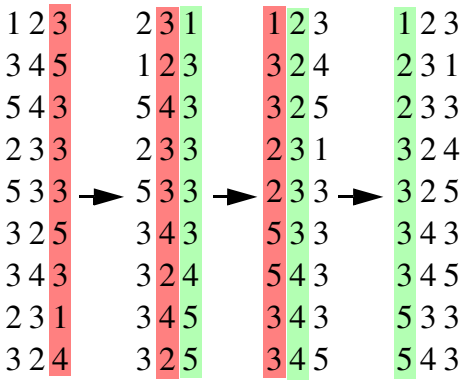
Algoritmi 6.3. Kaukalojärjestäminen.

```
public static binsort(E A[], int maxkey) {
   ArrayList<List<E>> kaukalot = new ArrayList<List<E>>(maxkey+1);
   for (int i = 0; i <= maxkey; i++)
      kaukalot[i] = new LinkedList<E>();
   for (int i = 0; i < A.length; i++)
      (kaukalot.get(A[i].getKey())).add(A[i]);
```

```
    A.clear();                                                    7
    for (int i = 0; i <= maxkey; i++)                             8
        A.addAll(kaukalot.get(i));                                9
}                                                                 10
```

Kantalukujärjestäminen (*radix sort*)

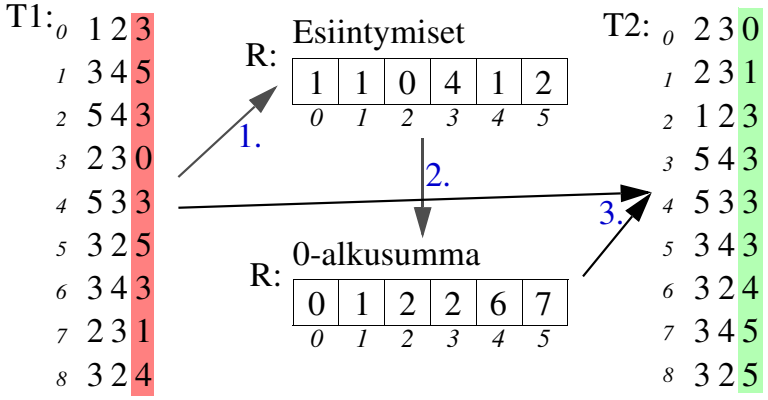
- Kaukalojärjestäminen ei ole hyvä jos avainten arvoalue (m) ja siten vaadittavien kaukaloiden määrä on hyvin suuri.
  - Avaimina on esimerkiksi liukulukuja tai merkkijonoja.
  - Kaukalolistojen ylläpito ei ole helppoa/kaikkein tehokkainta.
- Tehdään järjestäminen vaiheittain avaimenosien mukaan.
  - Jaetaan avaimet sopivan kokoisiin osiin, järjestetään ensin vähiten merkitsevän avaimenosan mukaan, sitten seuraavaksi merkitsevämmän avaimenosan mukaan, ..., lopuksi merkitsevimmän avaimenosan mukaan.
- Avainten pätkiminen kannattaa tehdä avaimen tyypin mukaan joko jakojäännösaritmetiikalla tai bittioperaatioilla.



- Järjestämisvaiheiden on oltava vakaita (*stable*), eli keskenään samanarvoisten alkioiden (kunkin vaiheen käyttämän avaimenosan mukaan) on pysyttävä alkuperäisessä järjestyksessä.

Järjestäminen lukumäärätaulukolla (histogrammilla)

- Muutetaan kaukalojärjestämistä vielä siten, ettei erillisiä kaukaloita varsinaisesti perusteta ja järjestetä, vaan syöte käydään kahdesti läpi, ensin lasketaan esiintymät (histogrammi) ja toisella kerralla sijoitetaan alkiot lukumäärätaulukon 0-alkusumman (alkuosasumma, *prefix sum*) osoittamaan paikkaan.



Kuva 6.4: Histogrammin käyttö kantalukujärjestämisessä.

- Taulukon 0-alkusumman kukin alkio on syötetaulukon edeltävien alkioiden summa: $R'[i] = \sum_{k=0}^{i-1} R[k]$, eli $R'[i] = R'[i-2] + R[i-1]$.
- Alkusummat osoittavat suoraan, mihin kohtaan kunkin ”kaukalon” alkiot tulevat. Saman ”kaukalon” alkiot pidetään alkuperäisessä järjestyksessä.
  - avain(T1[0]) = 3 ⇒ T2[R[3]] = T[0]; R[3]++;

  - avain(T1[1]) = 5 ⇒ T2[R[5]] = T[1]; R[5]++;
  - avain(T1[2]) = 3 ⇒ T2[R[3]] = T[2]; R[3]++;
  - T2[R[T1[i].getSubKey(vaihe)]] = T[i]; R[T1[i].getSubKey(vaihe)]++;
  - tai jopa T2[R[T1[i].getSubKey(vaihe)]++] = T[i];
- Avaimenosan mukaan samanarvoiset alkiot saadaan suoraan oikeaan (entiseen, vakaaseen) järjestykseen kun kasvatetaan 0-alkusumma -taulukon arvoja sitä mukaa kun taulukon alkion osoittama paikka käytetään.
- Jos/kun avaimet eivät ole kokonaislukuja, käytetään yleensä avaimen bittiesitystä r bittiä kerrallaan (ja kokonaisluvuillakin jotta säästytään jakolaskulta).
- Kaukaloita on siten 2^r.
  - Tyypillinen arvo r:lle on 16 (8...20) (joten m=65536 (256...1M)), riippuen syötteestä ja koneen muistiarkkitehtuurista (R-taulukon on hyvä mahtua välimuistiin).
  - Pitkä osa-avain eli enemmän kaukaloita vähentää vaiheiden määrää, mutta kasvattaa R-taulukkoa.
  - Käytettäessä yhtä aputaulukkoa, kannattaa $\lceil \frac{m}{r} \rceil$:n olla parillinen.
  - Aikavaativuus on $O\big(\frac{m}{r}(n+2^r)\big)$, missä m on avaimen pituus (bitteinä).
  - Tämä toki edellyttää, että avainten bittiesitys antaa suoraan oikean järjestyksen.
- Välimuisteille hieman haastava (jos r on iso), tehokkaampi jos keskusmuistiin kirjoitus viivästetty (ja nopea).

⇒ Kantalukujärjestäminen voi olla nopein peräkkäisjärjestäminen kun avaimet ovat lyhyitä ja syöte on suuri.

7 Abstraktien tietotyyppien toteuttaminen

⇒ Jotta abstraktia tietotyyppiä voitaisiin todella käyttää, on se toteutettava.

- Valmis toteutus on kierrätyksen (*reuse*) nimissä ensisijainen, mutta joskus sopivaa ei ole valmiina.
- Sopiva (käyttökelpoinen, riittävän tehokas):
  - tarvitsemamme operaatiot
    - riittävän tehokkaina
  - alkiotyyppi sovitettavissa tarpeisiimme
  - sopivalla ohjelmointikielellä toteutettuna
    - tällä kurssilla Java
  - soveltuvalla ohjelmistolisenssillä
- Jos soveltuvaa toteutusta ei ole saatavilla, meidän pitää tehdä se itse.
- Toteutustapojen kunnollinen ymmärtäminen myös auttaa meitä käyttämään tietotyyppejä oikein.

Vaikka itse toteutamme operaatiot, pidetään silti liittymä ja toteutus erillään, niin pääsemme joskus kierrättämään sitä samaa toteutusta.

- Oikeasti erillään!
  - Ei yllätyksiä tietorakenteen, eikä sovelluksen puolelle.
  - Mieluiten jo suunnitteluvaiheessa.
  - Myös dokumentointivaiheessa.
- Lisäksi niin algoritmistamme kuin ADT:n toteutuksesta tulee selkeämpi sekä helpommin luettava, tarkastettava, ylläpidettävä ja siirrettävä.

7.1 Kotelointi ja parametrointi

⇒ Toteutetaan annetun/tarvitsemamme abstraktin tietotyypin mukainen kokoelma.

- Liittymä määrää toteutuksen (ei päinvastoin).
  - Joskin toteutusta suunniteltaessa voimme ohjata/tarkentaa liittymässä ilmoitettuja asioita (mikäli tarpeen toteutuksen tehokkuuden kannalta).
  - Liittymän dokumentointiin tulee joitakin asioita vasta toteutuksesta riippuen (esim. aikavaativuudet, rajoitukset, jne).
- Todellisen toteutustavan voimme valita vapaasti (ideaalitapauksessa).
  - Käytännössä joskus toteutustapa vaikuttaa ainakin jonkinverran myös konkreettisten kokoelmien liittymäänkin.
    - Esim listan aseman käytös.
    - Järjestetyn joukon lisäoperaatiot (vrt. hajautustaululla toteutettu järjestämätön joukko).
  - Esim. pinon ja jonon toteutustapa sensijaan vaikuttaa korkeintaan aikavaativuuksiin.

Kotelointi (*encapsulation*)

- Käyttäjälle annetaan liittymä, toteutusta ei välttämättä lainkaan.
- Käyttäjällä ei pitäisi olla keinoa käyttää toteutusriippuvaisia ominaisuuksia (muuten kuin liittymän kautta).
- Kokoelman toteutuksen ei pidä olettaa mitään ylimääräistä sovelluksen toiminnasta.
  - Näin ei synny ylimääräisiä riippuvuuksia sovelluksen ja kokoelmatoteutuksen välille.
    - Kokoelmatoteutus voidaan vaihtaa muuttamatta sovellusta.
    - Samaa kokoelmatoteutusta voidaan käyttää useassa sovelluksessa.

Liittymä (*interface*)

- tyypit
  - rajoitukset
- operaatiot
  - parametrit, palautusarvot
  - rajoitukset
  - aikavaativuudet
  - käytös virhetilanteissa (poikkeukset)
- dokumentaatio

Toteutus (*implementation*)

- piilossa käyttäjältä
- sisäiset tyypit/luokat
- Sisäiset aliohjelmat/metodit
- operaatioiden toteutukset (aliohjelmien/metodien rungot)

⇒ Toteutuksen on oltava täsmälleen liittymän (ja sen dokumentoinnin) mukainen.

⇒ Eri ohjelmointikielet tukevat kotelointia eri tavoin.

Java

- Tarvittaessa voidaan liittymä (*interface*) ja sen toteuttava luokka esitellä erikseen, kts. esim *java.util.Set*, *java.util.HashSet*.
  - Käyttäjä voi käyttää liittymässä esiteltyjä operaatioita, mutta uuden ilmentymän luominen vaatii oikean luokan (tai tehdasmetodin).
  - Kun liittymä on kirjoitettu ja dokumentoitu, IDE generoi toteutettavan luokan rungon automaattisesti ja kopioi javadoc-kommentit uuteen luokkaan.
  - Hyöty: voidaan (myöhemmin) tehdä uusi toteutus joka toteuttaa saman liittymän ilman, että sovellusohjelmia muutetaan.
- Yksinkertaisemmin, kirjoitetaan ”liittymä” ja toteutus suoraan samaan tiedostoon.
  - Liittymää itseasiassa ei kirjoiteta erikseen lainkaan.
  - Generoidaan ”liittymän” kuvaus javadoc:lla.
  - Käyttäjä ei tarvitse/saa liittymän lähdekoodia, vaan kaikki riittävä informaatio kokoelman käyttämiseen löytyy generoidusta dokumentaatiosta.
  - Hyvää:
    - helppo tehdä (kunhan muistaa javadoc:n kommentointivaatimukset) `https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html`
    - IDE auttaa
    - voidaan valita mitkä osat dokumentoidaan, tarvittaessa voidaan editoida jälkikäteen.
    - tukee luokkahierarkioita (erityisesti perityt metodit)
  - Epävarmaa: dokumentaation päivitys muistettava
  - Luokka tai rajapinta on oltava olemassa jotta dokumentaatio käyttäjälle saadaan generoitua.
    - Toki sen ei vielä tarvitse toimia.

Pascal:

TRA_BTree.p:

```
module TRA_BTREE interface;
export TRA_BTREE = (BTREE, BTREE_ASSIGN_CHILD, ...)
type BTREE = ...
procedure BTREE_ASSIGN_CHILD(T:BTREE;
    ND:BTREE_NODE; dir:BTREE_DIR; NS:BTREE_NODE);
procedure ...
end. (* interface *)

module TRA_BTREE implementation;
procedure BTREE_ASSIGN_CHILD(T:BTREE;
    ND:BTREE_NODE; dir:BTREE_DIR; NS:BTREE_NODE);
begin
    ...
end;
...
end. (* implementation *)
```

Huonohkoa:

- Liittymä ja toteutus samassa tiedostossa.
- Liittymän käännetty .gpi (tai .TPU) tiedosto ei ole luettava.
- Toistoa (tunnukset jopa kolmesti).

C/C++

TRA_BTree.h:

```
typedef struct BTREE_str * BTREE;
void BTREE_ASSIGN_CHILD (BTREE T,
        BTREE_NODE ND, BTREE_DIR dir, BTREE_NODE NS);
```

TRA_BTree.c:

```
#include "TRA_BTree.h"
void BTREE_ASSIGN_CHILD (BTREE T,
        BTREE_NODE ND, BTREE_DIR dir,
        BTREE_NODE NS)
{
    ...
}
```

- Aivan kaikkea ei C:ssäkään saada piiloon (muuten tyyppien tarkastus ei onnistuisi).
- Ei nimettyjä moduuleja, vain tiedostoja (ANSI C:ssä).

Python

- Luokan/moduulin funktioiden ja jäsenten kotelointi perustuu osin käytäntöihin eikä ole aivan pakotettua.
  - Pakettisuojatut luokan metodit ja jäsenet (*protected* Javassa): jäsenen nimen eteen alaviiva (_).
  - Luokan sisäiset metodit ja jäsenet (*private* Javassa): jäsenen nimen eteen kaksi alaviivaa (_ _).
- " *from moduuli import ** " ei ota mukaan _ -alkuisia funktioita.

Parametrointi (*parameterization*)

- Haluamme kerran toteutetun listan (tms) toimivan erilaisille elementtityypeille sujuvasti.
  - Mieluiten vieläpä perustamatta erillistä toteutusta.
- Haluamme (mieluiten) kääntäjän/IDEn pystyvän tekemään tyyppitarkastukset jo lähdekoodista.

Reunaehtoja:

- Elementtityypin operaatiot (vertailut, luonnit, tulostukset, muut käsittelyt, kopioinnit ja vapautukset) pitää olla toteutettuna elementtityypissä.
- (Ainakin) jos elementtityyppi on iso (rakenteinen), niin kokoelmaan talletettava "elementti" on osoitin/viite rakenteeseen/olioon.
  - Tällöin talletusrakenteen luonnissa ja erityisesti tilan vapautuksessa tulee olla huolellinen (jos vapautus ei ole automaattista).
- ADT:n tyypit ja operaatiot on syytä nimetä siten, ettei tule konflikteja käytettäessä eri ADT:tä samassa ohjelmassa (sikäli kun kieli sellaisesta välittää).
- Sisäiset esittelyt ja toteutus on koteloitava pois loppukäyttäjältä.

Parametroinnin toteutusvaihtoehtoja:

Lähdekoodin tekstuaalinen parametrointi

- ADT:n esittelyssä ja toteutuksessa käytetään pseudotyyppiä (muodollinen tyyppiparametri) (nimeltään vaikkapa *E* tai *element*).
- Otettaessa ADT käyttöön, tuodaan todellinen elementtityyppi ADT:n esittelyyn ja toteutukseen.
  - *type element = real /* typedef */*
  - #define *element ...*
  - #include "*int_element*.h" /* "*element.h*" */
  - *template*:t (suomeksi mallit/rungot??) niissä kielissä, joissa ne ovat olemassa (C++, Modula-3).
  - C#: syntaksi (lähes) kuten Java (mutta erilaiset rajoitukset tyyppiturvallisuuden nimissä), ajonaikana (lähes) kuten C++ (erillinen käännös kullekin alkiotyypille).
- Elementtityypin esittely sisältää vertailut, uuden alkion luonnit, tilan vapautuksen, jne.

⇒ Tekstuaalisessa parametroinnissa myös ADT:n toteutus käännetään ohjelman mukana ((jokaisella) uudella elementtityypillä).

- Käännetyssä ohjelmassa on oma kopio parametroiduista kokoelmista/aliohjelmista kullekin alkiotyypille.
- ADT käyttöönoton työläys ja ADT:n koteloinnin aste vaihtelevat kielestä toiseen.
- Hyvää: kääntäjä tarkastaa tyyppien yhteensopivuudet joka vaiheessa.
- Alkiotyyppikohtainen toteutuksen käännös helpottaa hieman käännösaikaista optimointia.
- Huonoa: kankeahko useissa kielissä, toteutuksen oltava julkinen käyttöönottajalle, käännetty ohjelma kasvaa monistettaessa jokainen kokoelman toteutus kullekin alkiotyypille.

Tyypitön elementti / joustava tyypitys ajonaikana

⇒ Elementin tyyppi on tyypitön osoitin/viittaus (*void**, *pointer*, *Object*).

- ADT:n toteutus voidaan koteloida ja kääntää valmiiksi.
- Jos tyyppitarkastukset halutaan jo käännösaikana, niin ADT:n käyttöönottaja joutuu joko
  - muuntamaan (*cast*) tyypittömät osoittimet aina oikeaan tyyppiin, tai
  - tekemään (muokkaamaan) oikeaan tyyppiin muuntavat (makro)kuorrutukset operaatioiden ympärille.
- Vaihtoehtoisesti tyypintarkastukset/muunnokset voidaan tehdä vasta ajonaikaisesti kuten useimmissa skriptikielissä (Python, Javascript, jne).
  - Tyyppien oikeellisuutta ei voida tarkastaa ilman kattavaa testausta.
  - Tämä myös hieman hidastaa toimintaa.
- Vertailujen, lisäysten, jne. parametrien tyyppi on aina viittaus/osoitin, yksinkertaisten tyyppien paketointi osoittimeksi jää ohjelmoijalle, samoin tilanvaraus/vapautus.
- Ohjelmoijan on varmistuttava ADT:ien oikeasta käytöstä (pinosta otettavat alkiot tulkitaan (castataan) samaksi tyypiksi, kuin mitä sinne on viety).
- Hyvää: ei tarvita eri operaatioita eri pinoille tms.
  - Tosin vertailuoperaatiot annettava parametrina.
- Huonoa: kääntäjä ei voi tarkistaa tyyppien yhteensopivuuksia.

Tyypitön elementti ilman erillisiä osoittimia: elementille varataan sopivan kokoinen tavutaulukko (tavumäärä) ADT:n toteutusrakenteesta.

- Operaatiot palauttavat osoittimen tähän.

- Etu: tilanvaraus ja vapautus hoituvat automaattisesti.
- Tehokas: vain yksi lohko/alkio, vähemmän muistiviittauksia.
- Haitta: vaikeampi toteuttaa siirrettäväksi.
- Vaatii kuria/huolellisuutta.
- Ei ole mahdollista toteuttaa kaikilla kielillä (esim. Javalla).
- C/C++:llä onnistuu.

Välimuoto

⇒ Elementti, joka sisältää kohtuullisen kokoelman mahdollisia alkiotyyppejä.

- variable record / union jos kieli sitä tukee
- mukaan lukien tyypitön osoitin jollei valikoimasta löydy valmista
- ei kulu ylimääräistä tilaa
- Käyttäjältä edellytetään kuria (tai toteutukselta virhetarkastuksia).
- Valmiiksi rakennetuille tyypeille vertailut yms. toteutetaan valmiiksi.
- Operaatiot toteutetaan sekä tyypittämättöminä, että tyypitettyinä.
- Kokoelma tuntee tyyppinsä.
- Alkiotyyppi (kuvaaja) tuntee operaationsa.
- Alkio (solmu) tuntee kokoelmansa tai tyyppinsä.
- Kts. TR-kirjaston Pascal- tai C versio.

Tyyppiparametrointi skriptikielissä

- Useimpien skriptikielten muuttujat, parametrit ja lausekkeet ovat enemmän tai vähemmän ”tyypittämättömiä”, ts. kunkin lausekkeen tyyppi määräytyy ajonaikana jonkin säännöstön mukaan (merkkijono, kokonaisluku, lukuluku, taulukko, lista, jne).
- Niinpä tarvetta erilliselle tyyppiparametroinnille ei ole.
- Sensijaan kokoelman käyttäjä on itse vastuussa siitä mitä kokoelmaan talletetaan ja mitä sieltä otetaan pois.
  - Muuttujien kuvaava nimeäminen on hyvä apu.
  - Uudemmissa Pythonin versioissa on mahdollista merkitä (*type hints*) muuttujan tyyppi tai kokoelman parametrityyppi, mutta se on lähinnä ohje kehitysympäristölle (?)

Oliototeutus (kielille joissa ei ole generics-järjestelmää)

- Kokoelma toteutaan luokaksi, operaatiot metodeiksi.
- Sukulais-ADT:t toteutetaan periyttämällä (vain uudet operaatiot toteutettava).
  - Erilaisillakin kokoelmilla on jotain yhteistä, joten ne voivat olla kaikki saman abstraktin kokoelman alaluokkia.
- Alkioksi kelpaa mikä tahansa olio.
  - Alkiotyyppi kiinnitetään kokoelmaa luotaessa (konstruktorin parametri) tai ensimmäistä alkiota vietäessä.
  - Alkiotyyppien tyypintarkastus käännösaikana (jos kieli tukee) tai ajonaikana (pakotetut tyypinmuunnokset).
- Vertailuja, tulostuksia, jne varten joko rajoitetaan jonkin sopivan luokan aliluokkiin, tai edellytetään sopivan rajapinnan (vrt. *Comparable*) toteuttamista.
- Pakotetut tyypinmuunnokset (*cast*) hieman hankalia jos niitä joutuu käyttämään.
- ADT:n käyttö sopii sinänsä erinomaisesti olio-ohjelmointiin (ja päinvastoin).
- Vaihtoehtoisesti parametrointi tehdään luomalla aliluokka.

Javan geneeriset kokoelmat

⇒ Käännösvaiheessa sukua tekstuaaliselle parametroinnille, mutta ajonaikana virtuaalikoneessa toteutus on mallia tyypitön osoitin ja pakotetut tyypinmuunnokset.

- Geneerinen kokoelma parametroidaan yhdellä tai useammalla luokalla.

```
Kokoelma<Alkiotyyppi>
Kokoelma<Alkiotyyppi1, Alkiotyyppi2>
```

- Esimerkiksi kuvauksessa annetaan avaimen luokka ja kuvan luokka.
- Alkiotyypin on oltava luokka, yksinkertaista tyyppiä ei voi käyttää.
- Kokoelmaa esiteltäessä ja toteutettaessa käytetään muodollista tyyppiparametria, (mikä tahansa tunnus, mutta usein käytetään tunnusta "*E*").

```
public class BTreeNode<E> {
    private E element;
    ...
    public E setElement(E element) {
        ...
```

- *E* on siis alkiotyyppi (luokka) koko luokan esittelevässä lohkossa (muttei tietenkään sen ulkopuolella).
- Rajoituksena: ei voida suoraan luoda (*new*) *E* -tyypin ilmentymiä, ei myöskään taulukoita joiden alkiotyyppi olisi *E* (siis viittaus *E*:hen).
  - Kierto: luodaan *Object*-taulukko ja pakotetaan (cast) se *E*-taulukoksi, kts esimerkit.
    - Kyseessähän on joka tapauksessa vain viittaustaulukko.
  - Viittaus ja viittaustaulukko(viittaus) voivat olla tyyppiä *E* tai *E*[].
  - Jos tarvitsee (yleensä ei) luoda *E* alkioita, on käytettävä luontimetodia (*factory*) (*Class.newInstance*() tai *Constructor.newInstance*()).
    - Muistetaan, että kokoelmat ovat apuvälineitä joilla näppärästi hallitaan *E*:n olemassaolevia ilmentymiä (viittauksilla).
    - Uusien alkioiden luominen ei (yleensä) ole kokoelman tehtävä!
- Jos alkiotyyppiä halutaan jotenkin rajoittaa (yleensä tarvittaessa jotain ominaisuuksia), voidaan käyttää esittelyä *<E extends HaluttuYläluokka>*, esim:

```
public class JarjestyvatuvaLista<E extends Comparable> {
```

- Näin kokoelman alkioilta voidaan olettaa (ilman pakotettua tyypinmuunnosta) *compareTo* -metodin toteuttaminen
- Tarkkaan ottaen tämäkään ei juuri *Comparable*:n tapauksessa vielä riitä, sillä *Comparable* pitää parametroida, jotta vertailuiden "toisen" parametrin tyyppi tulisi varmistettua.
  - https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html
- Siis

```
public class JarjestyvaLista<E extends Comparable<E>> {
```

- Suomeksi: *E* toteuttaa *Comparable*:n parametroituna *E*:llä (vertailut luokkaa *E* vasten).
- Jos halutaan mahdollistaa *E*:n aliluokkienkin käyttö samassa listassa, niin:

```
public class LajLista<E extends Comparable<? super E>> {
```

- *Comparable*:n vaihtoehto on toteuttaa ja käyttää erillistä *Comparator* -rajapinnan toteuttavaa luokkaa, joten esim. JavaAPI:n järjestetyn joukon luokka on esitelty vain *TreeSet<E>*.
  - https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html
  - Esimerkiksi *TreeSet<E>* ei vaadi, että *E* toteuttaa *Comparable*:n.
  - Tällöin kelvollisen alkioluokan tarkastus tapahtuu vasta suorituksen aikana.
    - Katso esim *TreeMap*:n toteutus.

- Kokoelmaa käyttöönotettaessa annetaan todellinen tyyppiparametri.

```
Kokoelma<Alkio> oma = new Kokoelma<Alkio>();
```

- Tarkkaan ottaen jälkimmäinen parametri ei ole pakollinen sillä kokoelma on joka tapauksessa vain viittauskokoelma ja viittausten käytön tarkastuksiin viittauksen parametrointi riittää:

```
Kokoelma<Alkio> oma1 = new Kokoelma<Alkio>();
Kokoelma<Alkio> oma2 = new Kokoelma<>();
Kokoelma<Alkio> oma3 = new Kokoelma();
```

- Esimerkiksi:

```
LinkedList<String> mjonoLista = new LinkedList<String>();
LinkedList<Set<Integer>> lukuJoukkojenLista = new LinkedList<Set<Integer>>();
```

- Jos kokoelmalla on (toteutuksessa) komponentteina muita luokkia, molemmille määritellään erikseen (samat) alkiotyypit.

```
List<Integer> lista = ...;
...
ListNode<Integer> p = lista.getFirst();
Integer x = p.getElement();
```

Geneeriset metodit

- Myös yksittäisiä metodeja voidaan parametroida.
- Erityisesti kun kyse on ”apumetodista”.

```
public class Esim {
    public static <E> boolean samoja(Collection<E> kokoelma) {
        for (E x : kokoelma) {
            ... x.metodi() ...
        }
    }
...
LinkedList<String> L = new LinkedList<String>( ... );
boolean onko = samoja(L);
```

Lisää Java Generics-järjestelmästä: https://docs.oracle.com/javase/tutorial/java/generics/index.html.

Javan parametrointi ajonaikana

- Virtuaalikone (JVM) ei (yhteensopivuussyistä vanhoihin versioihin) ymmärrä mitään tyyppiparametreista.
  - Ne ovatkin enemmän kääntäjän ja ohjelmoijan väline.
- Kääntäjä korvaa kaikki tyyppiparametrit *Object*:lla (tai rajoitetulla luokalla/rajapinnalla (esim. *Comparable*)) (”*Type Erasure*”).
- Alkioiden käsittelyvaiheeseen sitten kääntäjä generoi tarvittavat pakotetut tyypinmuunnokset (*cast*) ja mahdollisesti apumetodeja (”*Bridge Methods*”).

Javan kokoelmajärjestelmä (*Collection framework*)

⇒ Sisältää liittymiä, runkototeutuksia ja varsinaisia toteutuksia, kaikki geneerisinä.

- Liittymä (*interface*) määrittelee mitä jonkin kokoelman on osattava (toteutettava).
  - Esim. *Collection*, *Set*, *Map*, *List*.
- Runkototeutus (*abstract implementation*) sisältää mahdollisuuksien mukaan pääosan monimutkaisemmasta toiminnallisuudesta (yksinkertaisina, geneerisinä versioina).
  - Esim. *AbstractCollection*, *AbstractList*
  - Perustoiminnallisuus (esim. lisäys, poisto, läpikäynnit) on toki näihin vielä rakennettava varsinaisessa toteutuksessa.
- Varsinaiset toteutukset ovat valmiita paketteja. Näitä voi olla useita samaan liittymään.
  - Esim. *TreeSet*, *HashSet*, *LinkedList*, *ArrayList*.
- https://docs.oracle.com/javase/8/docs/technotes/guides/collections/index.html

default metodit rajapinnoissa (Java 8-)

- Java8:sta alkaen rajapinnan (*interface*) määrittelemälle metodille on voinut tehdä myös rungon kunhan metodin merkitsee avainsanalla ***default***.
- Tälläistä metodia ei ole pakko toteuttaa (tai edes mainita) rajapinnan toteuttavassa luokassa. Tällöin luokka käyttää rajapinnan *default*-metodia.
- Näin voidaan myös lisätä olemassaolevaan rajapintaan metodi ilman, että olemassaolevia rajapinnan toteuttavia luokkia muutetaan lainkaan!
- Mekanismilla voidaan yksinkertaisessa tilanteessa helpottaa rajapinnan toteuttamista käyttämättä erillistä abstraktia luokkaa rajapinnan ja toteutuksen välissä.
- Rajapinnan oletusmetodi ei luonnollisestikaan voi tehdä mitään kovin monimutkaista koska sillä ei esimerkiksi ole jäseniä käytettävissä lainkaan.
- Oletusmetodi voi esimerkiksi heittää poikkeuksen *UnsupportedOperationException* tai kutsua toista saman rajapinnan metodia, kts. esimerkit.

7.2 Listan toteuttaminen

⇒ Toteutusvaihtoehtoja on lukuisia...

- Kullakin hyvät ja huonot puolensa.
- Aivan toteutusriippumattomuuteen ei valitettavasti päästä koska aseman käytös vaihtelee hieman toteutustavasta toiseen.

Käytetään geneerisiä parametroituja luokkia.

Listan ja muiden peräkkäisrakenteiden taulukkopohjainen toteutus (*array list*)

⇒ Talletetaan alkiot taulukkoon tiiviisti.

- Varattava riittävän/sopivan suuri taulukko?
  - Tilan loputtua joko
    - virheilmoitus (ei hyvä) tai
    - tilaa on laajennettava (hidasta, mutta harvinaista, eli keskimäärin vakioaikaista) tai
    - käytettävä dynaamista taulukkototeutusta (jolloin tila ei siis lopu) (myös keskimäärin vakioaikaista).
  - Tilaa kuluu hukkaan jos varataan tarpeettoman suuri tila.
- Muistetaan viimeisen alkion indeksi.
- Asemana taulukon indeksi.

- Lisäys ja poisto muualle kuin loppuun $O(n)$.
- Voidaan myös tehostaa poistoa alusta ylläpitämällä erikseen ensimmäisen alkion indeksiä, mutta tällöin alkuun jää tyhjää tilaa. Kts. pakan toteutus s. 92.

⇒ Toteutus suoraviivaista.

Esimerkki 7.1. Lista taulukossa (*ArrayList2*.java).

```
public class ArrayList2<E> {
   private E[] data;                    // viittaukset alkioihin
   private int n;                       // alkioiden määrä

   @SuppressWarnings({"unchecked"})
   public ArrayList2() {
      // E[] taulukkoa ei voi luoda, on luotava Object[] ja castattava
      data = (E[]) (new Object[20]);
      n = 0;
   }
   public int EOL() {
      return n;
   }
   public int first() {
      return 0;
   }
   public E getElement(int p) {
      if (p >= n || p < 0)
         throw new ListException("getElement: invalid position " + p);
      return data[p];
   }
   public void insert(int p, E x) {
      if (p > n || p < 0)
         throw new ListException("Insert: invalid position " + p);
      if (n == data.length)                // jos taulukko täynnä, niin varataan uuden
         doubleCapacity();                 // isompi tila ja kopioidaan vanhat alkiot
      for (int i = n; i > p; i--)
         data[i+1] = data[i];
      data[p] = x;
      n++;
   }
   public int next(int p) {
      if (p >= n || p < 0)
         throw new ListException("Invalid position " + p);
      return p+1;
   }
   // ...
}
```

Esimerkki 7.2. Listan kopiointi vaikuttaa päällisin puolin hyvin yksinkertaiselta tehtävältä, mutta asiaa lähemmin tarkasteltaessa osoittautuu, ettei kopionti ole aivan yksiselitteinen. Jos muuttujat $L2$ ja $L1$ ovat äsken esitetyllä tavalla toteutettuja listoja (*ArrayList*2 $L1$, $L2$;), aiheuttaa sijoitus $L2 = L1$ pelkän objektiviitteen kopioitumisen. Listan talletusrakenne ei toisin sanoen kopioidu, vaan sijoituksen jälkeen molemmat muuttujat tarkoittavat samaa varsinaista listaa. Jos sijoituksen jälkeen esimerkiksi poistetaan alkio jommastakummasta listasta, häviää sama alkio toisestakin listasta, mikä

ei todennäköisesti ole tarkoitus! Java kieltä käytettäessä käyttäjän toki pitäisi tämä arvata, sillä Javassa kaikki ovat viittauksia.

Jotta kopiointi onnistuisi, tulisi kopioida koko talletusrakenne (taulukko tai kukin listasolmu erikseen). Turvallisin kopiointikeino on muodostaa tyhjä $L2$-lista, johon listan $L1$ sisältö kopioidaan alkio alkiolta listaoperaatioiden avulla. Tämäkään ei vielä kopioi itse alkioita, vaan listat sisältävät viitteet samoihin alkio-objekteihin.

Listan ketjutettu taulukkototeutus (*linked list using array of nodes*)

⇒ Yksinkertaisessa taulukkototeutuksessa poisto ja lisäys olivat tarpeettoman raskaita.

- Miksi käyttää listaa jos siinä on kaikki taulukon huonot puolet?
- Lisäys ja poisto pitäisi voida tehdä siirtämättä alkioita!
- Tämä mahdollistuu ketjuttamalla alkiot, jolloin ”linkkejä” seuraamalla voidaan hypätä poistetun alkion yli, ja vastaavasti löytää mihin tahansa kohtaa lisätty uusi alkio helposti.
  - Ketjutus tehdään ylläpitämällä kunkin alkion seuraajan indeksiä erillisessä taulukossa.
- Vapaat paikat ketjutetaan ”toiseksi listaksi”.
  - Aluksi pitää linkittää kaikki vapaat paikat (O(talletusalueen_koko) aika).
  - Lisättäessä poistetaan linkitys vapaiden listasta ja lisätään oikeaan listaan.
  - Poistettaessa poistetaan oikeasta listasta ja lisätään vapaiden listaan. [*ArrayLinkedList.java*, *ArrayLinkedList2.java*]

```
public class ArrayLinkedList<E> {
   private E[] data;
   private int[] next;
   private int first, last, firstFree;
   public static final int EOL = -1;
   public ArrayLinkedList(int size) {
      data = (E[]) (new Object[size]);;
      next = new int[size];
      for (int i = 0; i < size-1; i++)
         next[i] = i+1;
      next[size-1] = EOL;
      first = EOL;
      last = EOL;
      firstFree = 0;
   }
   public void insert(int p, E x) {
      if (p >= data.length || p < 0)
         throw new ListException("Invalid position");
      if (firstFree == -1)
         doubleCapacity();                    // tuplaa myös next[] -taulukon
      int i = firstFree;
      firstFree = next[firstFree];
      if (p != EOL) {                  // lisäys muualle kuin loppuun
         data[i] = data[p];                  // siirretään aiempi alkio
         next[i] = next[p];                  // vapaaseen paikkaan
         data[p] = x;                        // uusi tilalle
         next[p] = i;                        // seuraajaksi siirretty alkio
      } else {                         // lisäys listan loppuun
         data[i] = x;
         next[i] = EOL;
         if (first == EOL)             // tyhjään listaan myös ensimmäiseksi
```

```
            first = i;
        else
            next[last] = i;                    // viimeisen seuraajaksi
        last = i;
      }
   }
   public int first() {
      return first;
   }
   public int next(int p) {
      if (p >= data.length || p < 0)
         throw new ListException("Invalid position");
      return next[p];
   }
}
```

⇒ Tässäkin versiossa *remove* ja *previous* ovat $O(n)$, mikä ei aina ole hyväksyttävää.

- *deleteNext* olisi vakioaikainen.
- Samoin voidaan siirtää poistettavan seuraaja poistettavan paikalle, näin poistettavan edeltäjän *next*-viittausta ei tarvitse päivittää.
- Viimeisen alkion poisto on kuitenkin $O(n)$.
- Linkitetään lista kahteen suuntaan.

```
public class ArrayDoublyLinkedList<E> {
   private E[] data;
   private int[] next;
   private int[] prev;
   private int first, last, firstFree;
   public static final int EOL = -1;
   @SuppressWarnings({"unchecked"})
   public ArrayDoublyLinkedList(int size) {
      data = (E[]) (new Object[size]);
      next = new int[size];
      prev = new int[size];
      for (int i = 0; i < size-1; i++)
         next[i] = i+1;
      next[size-1] = EOL;
      for (int i = 1; i < size; i++)
         prev[i] = i-1;
      prev[0] = EOL;
      first = EOL;        last = EOL;
      firstFree = 0;
}  }
```

Taulukkototeutuksen edut:

- Tehokas (ainakin yksinkertaisena versiona).
  - ei linkkitilaa
  - peräkkäiskäsittely tehokasta
- Ei tarvita dynaamista tilanvarausta (ja vapautusta) jokaisen muutoksen yhteydessä.

Haitat

- Rajoitettu tila / runsaasti hukkatilaa.
- Monimutkainen (kaksisuuntaiseksi linkitettynä).

3-alkioinen tietue 3:n taulukon sijaan ei välttämättä ole kätevä.

Listan dynaaminen linkitetty toteutus (*linked list*)

Taulukkototeutuksen haitat (lisääminen/poisto keskelle/ltä ja tilan rajallisuus/tilan kasvatus) vältetään dynaamisella toteutuksella.

- Varataan tilaa yhdelle alkioviittauksen sisältävälle listasolmulle kerrallaan.
- Vapautetaan solmu kun alkio poistetaan listasta (Javassa vain irroitetaan viittaukset, vapautuksen hoitaa roskienkeruu).
- Ylläpidetään kunkin listasolmun seuraajatietoa (ja edeltäjää, jos tarpeen) olioviitteillä listasolmuihin.
- Toiminta varsin samanlaista kuin linkitetyssä taulukkototeutuksessakin.
- Tarpeen mukaan erilaisia variaatioita aivan kuten taulukkototeutuksessakin.
- Ongelmat osittain samoja.

Yksisuuntaisesti linkitetty lista (*singly linked list*)

- Lista on viittaus ensimmäiseen solmuun.
- Kussakin solmussa on viittaus (vain) seuraajaan.
- Hyvää: yksinkertainen, *next*(), *insertAfter*() ovat $O(1)$.
  - *insert*() voidaan myös tehdä $O(1)$ ajassa siirtämällä olemassaoleva alkio uuteen solmuun ja sijoittamalla uusi alkio olemassaolevaan solmuun (kuten aiemmin linkitetyssä taulukkototeutuksessa).
- Puutteita:
  - *previous*(), *remove*() ovat $O(n)$.
    - *removeNext*() ja *removeFirst*() olisivat vakioaikaisia.
    - *remove*() voi ottaa seuraavan alkion tilalle tähän solmuun, mutta silloinkin viimeisen alkio poisto on $O(n)$.
    - Voidaan välttää kaksisuuntaisuudella.
  - Tyhjä lista on *null*.
    - Käyttäjän on joko osattava itse lisätä ensimmäinen alkio *new*:llä, tai operaatioiden on oltava luokkametodeja (*static*).
  - Listan ensimmäisen alkion muuttuminen muuttaa koko listamuuttujan arvoa; listaa on käsiteltävä varoen.
    - Ongelma voidaan välttää käyttämällä tunnussolmua.

Esimerkki 7.3. Yksisuuntaisesti linkitetty lista ilman tunnussolmua.

```
public class SingleLinkedList<E> {
   private E element;
   private SingleLinkedList<E> next;
   public SingleLinkedList(E x) {
      this.element = x;       // this -viittaukset eivät ole pakollisia
      this.next = null;
   }
   public void insertAfter(E x) {
      SingleLinkedList<E> n = new SingleLinkedList<E>(x);
      n.next = this.next;
      this.next = n;
   }
   public E deleteNext() {
      if (this.next == null)
         throw new NullPointerException("Cannot delete null node");
```

```
        E x = next.getElement();                                    16
        this.next = this.next.next;                                17
        return x;                                                  18
    }                                                              19
    // ...                                                         20
}                                                                  21
```

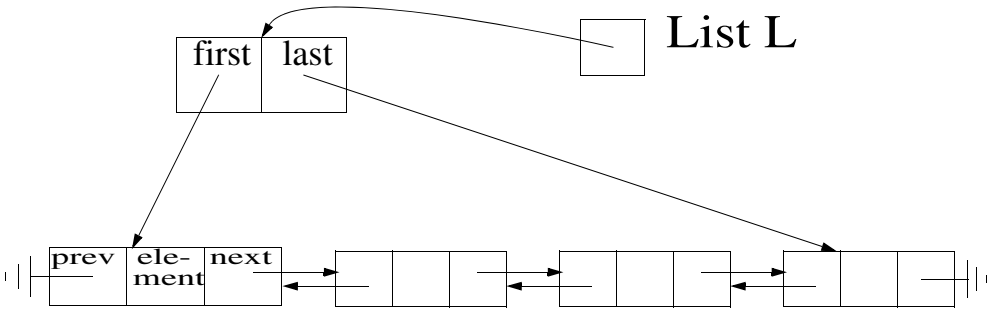
Tunnussolmulla varustettu lista

- Listan ”ensimmäisenä solmuna” on ylimääräinen tunnussolmu josta on viittaus varsinaiseen ensimmäiseen solmuun (mikäli se on olemassa).
- Lista on viittaus tunnussolmuun.
- Tyhjässä listassa on vain tunnussolmu (josta viittaus ”ensimmäiseen” on *null*).
- Listaviittaus muuttuu vain luotaessa, muuttuvia parametreja ei tarvita muualla.
- Joka tilanteessa tarkistetaan käsitelläänkö tunnussolmua vai oikeaa listan solmua.
- Luokat *List* ja asema (”*ListNode*”) voivat olla samat.
  - Kääntäjä ei voi tarkastaa oikeaa käyttöä.
    - Esim. parametrien järjestys *remove*:ssa.
- Kts. esim. *DoubleLinkedList2.java*
- Palaamme samantyyppisen tunnussolmun käyttöön puun toteutuksen yhteydessä kohdassa 7.4 (s. 94).
- Joskus käytetään myös ylimääräistä loppusolmua (*sentinel node*) jos ei käytetä erillistä tunnusolmua.

Kaksisuuntaisesti linkitetty lista (*doubly linked list*)

- Kaikissa yksisuuntaisissa toteutuksissa *remove*() ja *previous*() vievät $O(n)$ ajan (ainakin joissakin tapauksissa).
- Jos/kun näitä operaatioita tarvitaan, kannattaa lista linkittää kaksisuuntaisesti jolloin jokaisesta solmusta on viittaus myös edeltäjään.

Erillinen tyyppi tunnussolmua varten



Kuva 7.1: Kaksisuuntaisesti linkitetyn tunnussolmulla varustetun listan toteutus.

- Selkeyden vuoksi (sekä itselle, että kääntäjälle) esittelemme eri tyypin (luokan) tunnussolmua varten.
- Saman pakkauksen (*package*) käyttö tai solmuluokan sijoittaminen listaluokan sisään (tiedostoon) helpottaa jäseniin viittauksia.

  - Täysin erillisten luokkien tapauksessa *setX* ja *getX* operaatiot haittaavat luettavuutta, kts. *DoubleLinkedList.java*.
- Jos listasolmun luokkaa ei näytetä käyttäjälle, se voi olla listan sisäluokka.

Algoritmi 7.4. Kaksisuuntaisesti linkitetty dynaaminen lista erillisellä tunnussolmulla samassa pakkauksessa. [*DoubleLinkedList3.java*, *DoubleLinkedList3Node.java*]

```
public class DoubleLinkedList3<E> {
   private DoubleLinkedList3Node<E> first;
   private DoubleLinkedList3Node<E> last;
   public static final DoubleLinkedList3Node EOL = null;
   public DoubleLinkedList3() {
      first = EOL;
      last = EOL;
   }

   public void insert(DoubleLinkedList3Node<E> p, E x) {
      DoubleLinkedList3Node<E> n = new DoubleLinkedList3Node<E>(x);
      if (p != EOL) {
         n.next = p;                    // muualle kuin loppuun
         n.prev = p.prev;
         p.prev = n;
         if (p == this.first)
            this.first = n;              // alkuun
         else
            n.prev.next = n;             // väliin
      } else {
         n.next = EOL;                  // listan loppuun
         n.prev = this.last;
         if (this.last == EOL)
            this.first = n;              // tyhjään listaan
         else
            n.prev.next = n;             // epätyhjään listaan
         this.last = n;
   }  }

   public E delete(DoubleLinkedList3Node<E> p) {
      if (p == EOL)
         throw new NullPointerException(''Nonexisting node'');
      E x = p.element;
      if (p.next != EOL)
         p.next.prev = p.prev;          // muu kuin viimeinen
      else
         this.last = p.prev;            // poistetaan viimeinen
      if (p.prev != EOL)
         p.prev.next = p.next;          // muu kuin ensimmäinen
      else
         this.first = p.next;           // ensimmäinen
      p.next = EOL;                     // irroitetaan poistettava solmu
      p.prev = EOL;                     // listasta
      return x;
   }
}

public class DoubleLinkedList3Node<E> {
```

```
    protected DoubleLinkedList3Node<E> prev;
    protected DoubleLinkedList3Node<E> next;
    protected E element;
    public static final DoubleLinkedList3Node EOL = null;
    protected DoubleLinkedList3Node(E x) {
        prev = EOL;
        next = EOL;
        element = x;
    }

    public DoubleLinkedList3Node<E> prev() {
        return prev;
    }
    public DoubleLinkedList3Node<E> next() {
        return next;
    }
    public E getElement() {
        return element;
    }
}
```

Hyvää:

- Rajaton, ei hukkatilaa (paitsi viittauksille).
- Kaikki operaatiot ovat vakioaikaisia.

Huonoa

- Paljon *new* -kutsuja ja vastaavia vapautuksia (jotka tosin ovat nopeita).
- Aseman käsite hieman epämääräinen (mutta parempi kuin taulukkototeutuksessa).

Läpikäyntien toteutus s. 106.

7.3 Listan erikoistapaukset

⇒ Pinon, jonon, pakan ja renkaan toteuttamiseen voidaan soveltaa listan toteutusmalleja.

Usein yksinkertainen toteutus riittää.

- Pino, jono: linkitetyssä dynaamisessa toteutuksessa yksi suunta riittää.
- Pino, jono, pakka: ei lisäystä keskelle
  ⇒ taulukko toimii kunhan tilan kasvatuksesta huolehditaan tarvittaessa.

Esimerkki 7.5. Pinon alkiot voidaan tallettaa taulukkoon siten, että pohjimmainen alkio on indeksiä 0 (tai 1) vastaavassa talletuspaikassa ja pino kasvaa suurempia indeksin arvoja kohden. Taulukon lisäksi tarvitaan silloin myös pinon pinnan ilmaiseva tieto, mutta alkioita ei koskaan tarvitse siirrellä talletusalueella, koska pinon päällimmäisen alkion ”yläpuolella” on aina tilaa — siihen saakka, kunnes koko talletusalue täyttyy. Talletusalueen täyttyessä joudutaan varaamaan uusi suurempi alue ja kopioimaan vanhat alkiot sinne. Tai ylläpitämään talletustaulukoiden listaa (pinoa).

Jono ja pakka taulukossa

⇒ Jono ryömii pois taulukon alusta.
⇒ Pakka saattaa ryömiä kumpaan suuntaan tahansa.

- Emme voi jatkuvasti siirrellä alkioita sillä se veisi lineaarisen ajan.
- Tulkitaan taulukko ”renkaaksi” (ns. rengaspuskuri (*ring buffer*)):
  - Kun häntä (se pää johon lisätään) saavuttaa taulukon lopun, tehdään lisäys taulukon alkuun (**jos** se on vapautunut poistossa).
  - Kun keula (se pää josta poistetaan) saavuttaa taulukon lopun, siirrytään poistamaan taulukon alusta (**jos** sinne on jo lisätty).
- Pidetään muistissa sekä keulan, että hännän indeksi.
- Kun keula == häntä+1, jono/pakka on tyhjä
  - tai täysi; nämä pitää pystyä erottamaan toisistaan.
  - Indeksoinnin voi hoitaa monella tavalla
    - Kaksi seuraavista kolmesta riittää: ensimmäinen, viimeinen, lukumäärä.
    - Ensimmäinen alkio, seuraava lisäyskohta, jne.
- Jos jonon/pakan talletustaulukko täyttyy, on varattava uusi (kaksi kertaa) isompi taulukko ja kopioitava vanhat alkiot sinne ennen uuden alkion lisäämistä.
- HT

Pino, jono, pakka ja rengas dynaamisesti

- Kuten lista, kutakin alkiota varten on oma solmu.
- Koska käyttäjä ei voi/saa käyttää solmua suoraan, solmu voi/tulee olla sisäluokka (siis privaatti luokka samassa tiedostossa).
- Kussakin solmussa on viittaus seuraajasolmuun (ja tarvittaessa edeltäjäsolmuun) (ja alkio).
- Kaksisuuntaisuutta tarvitaan vain pakassa ja renkaassa (renkaassa tosin voidaan pärjätä ilmankin jos rajoitetaan operaatioita).
- Pinossa ja jonossa piirrä kuva miten päin viittaukset tarvitaan jotta lisäys ja poisto onnistuvat vakioajassa.
- HT

Läpikäyntien toteutus s. 106.

”Rajattoman” kokoinen/harva taulukko

- Varsinainen talletusalue palasina.

⇒ Indeksitaulukko joka sisältää viitteitä varsinaisen talletusalkueen palasiin.

taulukkoTaulukko[i/m][i%m] 1

- Palaset varataan vasta kun niitä tarvitaan.
  - Sijoitusoperaation on tarkistettava onko ko. palasta varattu.

Jos indeksi kasvaa liian suureksi, paloitellaan se ja perustetaan ylemmän tason indeksi.

- 10 000 tietueen palaset.
- 10 000 osoittimen väli-indeksi.
- 10 000 osoittimen pääindeksi.
- 1 000 000 000 000 tietuetta.
- Hukkatila min 10 000 sanaa, max 100 010 000 sanaa.
- Alin (alimmat) tasot voivat olla massamuistissa, kts. B-puu (TRA II).

Harva taulukko vai kuvaus?

- Joskus alkioita on hyvin vähän suhteessa avainten arvoalueeseen.

  - Vähän alkioita verrattujan käytettyjen indeksien määrään.
  - ”Indeksit” ovatkin todella suuria lukuja, reaalilukuja, merkkijonoja, tms.
- Käytetään hajautustaululla tai puulla totetutettua kuvausta: *Hash/TreeMap*<*indeksi*,*arvo*>.
  - Vain vakiokertoimella hitaampi kuin oikea taulukko.
  - Paitsi jos tarvitaan järjestyksessä läpikäyntiä, niin *TreeMap* ja $O(\log n)$ lisäkerroin.
  - Muistinkulutus lineaarinen alkioiden määrän suhteen.
  - *HashMap*<*Long*, *V*>, *HashMap*<*BigInteger*, *V*>, *HashMap*<*Double*, *V*>, *HashMap*<*String*, *V*>, jne.
  - Monissa dynaamisesti tyypitetyissä (skripti)kielissä taulukolta näyttävä rakenne onkin toteutukseltaan kuvaus.

7.4 Puiden toteuttaminen

⇒ Kehitellään listan toteutustavoista.

- Taulukkoon (kun puun maksimikoko (korkeus) ja muoto tunnetaan, eikä puuta muokata (esim. tehdä kiertoja)).
- Dynaaminen (jos puuta muokataan tai kokoa (korkeutta) ei tunneta).

Yleinen puu vs. binääripuu:

- Molemmilla on kaksi ”nuorempaa” sukulaista joihin tarvitaan viittaus (tai muu keino löytää ne).
- Kumpi tahansa voidaan toteuttaa käyttäen toisen operaatioita.
- Käsitellään binääripuuta.

Dynaaminen puun toteutus

- Jokaisella alkiolla on oma puusolmu.
- Samat valinnat kuin listallakin:
  - tunnussolmu vai ei?
  - saman- vai erityyppinen tunnussolmu?
  - onko solmun ja puun tyyppi sama?

Puu ja solmu yhteensopivina tyyppeinä

- Erityisesti rekursiivisissa algoritmeissa olisi mukavaa, jos puun mitä tahansa solmua voisi kutsua kuten puuta ja päinvastoin.
  - Käynnistysaliohjelma jää usein tarpeettomaksi kun riittää yksi parametri.
- Suurin käytännön hyöty olisi puita rakennettaessa (yhdistettäessä).
- Huonona puolena tyyppitarkastusten huononeminen ja sitä kautta virhemahdollisuudet.
  - Jos yksi puu liitetään toisen puun solmun lapseksi, jääkö liitetty puu edelleen olemaan? Miten viitteet toimivat, jne?
- Kolme vaihtoehtoa yhteensopivuuteen:

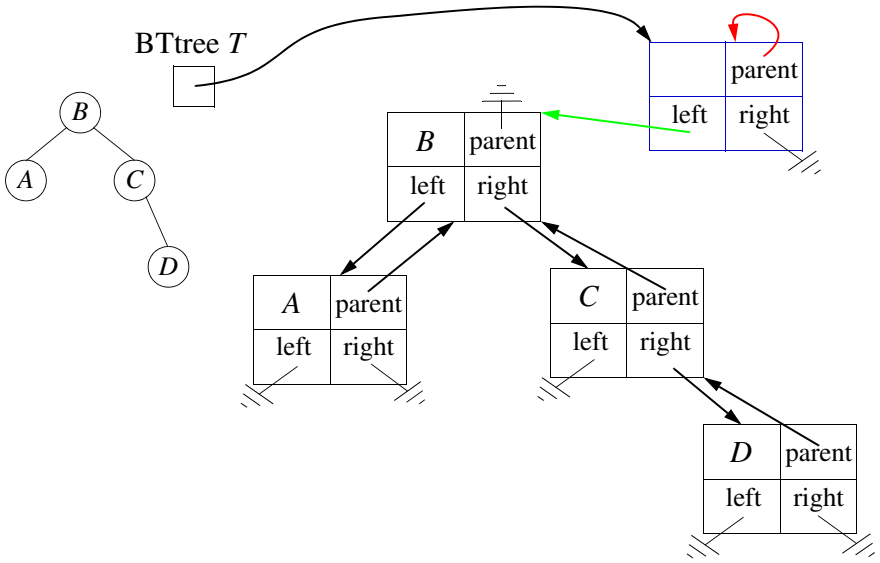
  1. Toteutetaan puu ilman tunnussolmua, jolloin puun juurisolmu edustaa koko puuta.
     - Hyvää
       * yksinkertainen, ei poikkeuksia

- Huonoa
  - ei ylimääräisiä tietoja puusta (tai jos on, niin tilanhukkaa)
  - juurisolmu voi muuttua ⇒ kokoelmamuuttujan (viittauksen) arvo voi muuttua

2. Käytetään tunnussolmun esittämiseen samaa tietuetyyppiä (luokkaa) kuin puun varsinaisten solmujen esittämiseen.
   - Hyvää
     - kokoelmamuuttuja ei muutu (paitsi luotaessa)
   - Huonoa
     - toteutuksessa otettava aina huomioon tunnisteen ja solmun ero
     - vähemmän selkeä, vaatii ohjelmoijalta kuria
3. Määritellään yläluokka *puukomponentti* jota puu ja solmu laajentavat
   - Hyvää: Selkeämpi tyyppien tarkastus.
   - Huonoa: Toteutuksen kaikissa metodeissa on tarkastettava kumpi on kyseessä.

Esimerkki 7.6. Binääripuu samantyyppisellä tunnussolmulla.

- Merkitään tunnussolmua $T.parent = T$, ja laitetaan viite varsinaiseen juureen $T.left$:iin. $T.right$:iin ja $T.elem$:iin voidaan laittaa jotain jos halutaan.



Algoritmi 7.7. Binääripuu samantyyppisellä tunnussolmulla.

```
public class BTree1<E> {                                                    1
   private BTree1<E> left;                                                  2
   private BTree1<E> right;                                                 3
   private BTree1<E> parent;                                                4
   private E element;                                                       5
   public BTree1() {                        // uuden puun luonti           6
      left = null;                           // juuri                      7
      parent = this;                         // tunnussolmun merkki        8
      right = null;                                                         9
      element = null;                                                       10
   }                                                                        11
   public BTree1(E x) {                 // uusi vielä kytkemätön puusolmu (alkiolle x)   12
      left = null;                                                          13
      parent = null;                                                        14
      right = null;                                                         15
```

```
        element = x;                                                    16
    }                                                                   17
                                                                        18
    public BTree1<E> getRoot() {                                        19
        if (parent != this)                                             20
            throw new TreeException(''Not a BTree1'');                  21
        return left;                                                    22
    }                                                                   23
    public void setRoot(BTree1<E> n) {                                  24
        if (parent != this)                                             25
            throw new TreeException(''Not a BTree1'');                  26
        left = n;                                                       27
        n.parent = null;                                                28
    }                                                                   29
    public BTree1<E> getLeftChild() {                                   30
        if (parent == this)                                             31
            throw new TreeException(''Not a BTree1 node'');             32
        return left;                                                    33
    }                                                                   34
        ...                                                             35
}                                                                       36
```

Binääripuu taulukossa

- Staattinen toteutus: varataan riittävän suuri taulukko (ja/tai ”kasvatetaan” taulukkoa tarpeen mukaan):
  - Oletetaan puu täydeksi (ts. varataan puun kaikille mahdollisille solmuille tilaa).
  - Tarvitaan tilaa 2^{k+1} solmulle (missä k on puun maksimikorkeus).
    - Sukulaiset löytyvät laskemalla (ei tarvita viittauksia):
    - juuri on taulukon indeksissä 1
    - juuren lapset ovat indekseissä 2 ja 3
    - 2:n lapset ovat indekseissä 4 ja 5
    - 3:n lapset ovat indekseissä 6 ja 7
    - i:n vanhempi on indeksissä $i/2$.
    - i:n vasen lapsi on indeksissä $2\times i$ ja oikea lapsi on indeksissä $2\times i+1$
    - Indeksit voidaan tarvittaessa muuttaa 0-alkuisiksi (∓ 1).
  - Kaikki operaatiot ovat vakioaikaisia (paitsi ehkä taulukon alustus).
    - Solmun olemassaolon tarkistamiseksi tyhjät alkiot on pystyttävä erottamaan (jos olemassaoloa on tarpeen tarkastaa).
    - Vaihtoehtoisesti: täysin tasapainoisessa puussa riittää laskea solmujen määrä (kts. prioriteettijonon toteutus ja kekojärjestäminen s. 102).
  - Tehokas ja taloudellinen tallennustapa kun puu on jokseenkin tasapainoinen (haarat ”täysiä”).
    - Täysi katastrofi jos puu on kovin epätasapainoinen.
  - Ei lainkaan viittauksia puusolmuihin tms. ohjaustietoa!
  - Yleinen puu toteutettaan ”vinona binääripuuna”.
- Linkitetty taulukkototeutus:
  - Kuten linkitettu listan taulukkototeutus.
  - Erillisissä taulukoissa tiedot ko. solmun lapsien sijainnista.
  - Toimii myös yleiselle puulle suoraan.

8 Joukkojen toteuttaminen

⇒ Joukko toteutetaan ”tarpeen mukaan”.

- Mitä operaatioita tarvitaan?
  - Joukko, kuvaus, prioriteettijono, laukku?
- Mitä ei tarvita?
  - Esim. usean joukon operaatiot?
- Minkälaisia läpikäyntejä tarvitaan?
- Mitä ei tarvita (kovin usein)?
- Mitkä operaatiot oltava (erityisen) tehokkaita?
- Mitä ominaisuuksia alkiotyypillä on? (tai mitä siltä voidaan olettaa)
- Tunnetaanko perusjoukon (arvoalueen) tai talletettavan joukon koko?
- Joukko-operaatiot s. 54.
- Kuvaus toteutetaan kuten joukko (avain-arvo -pari alkion sijaan).

⇒ Eri toteutustapojen operaatioiden aikavaativuudet ja varsinkin rajoitukset vaihtelevat!

Joukkojen toteuttaminen vai käyttäminen?

- Useimmissa moderneissa ohjelmointikielissä/alustoissa on valmiina tai lisäkirjastona vähintään joku joukkototeutus.
  - Esim. *java.util.*Set*, glib, .NET dictionary, perl, Python, sh, jne.
  - Relaatiotietokanta on joukko (relaatio / kuvaus).
  - Aikavaativuudet on usein valitettavan huonosti kuvattu.
- Useimmiten on järkevintä käyttää näitä valmiita toteutuksia.
  - Joskus toki tulee eteen jokin erikoinen vaatimusyhdistelmä johon ei löydy valmista ratkaisua.

⇒ Oleellista onkin osata valita oikea joukkototeutus tarpeen mukaan ja käyttää sitä tehokkaasti (oikein).

- Oikein == ei väärin!
- Jotta osaamme valita oikean joukkototeutuksen ja käyttää sitä tehokkaasti, meidän on käytännössä tunnettava niiden toiminta.
- Joukkototeutuksissa valinta on yleensä hajautuksen ja puun välillä.
  - Tietokannoissa indeksien toteutus on valittava näiden kesken.
- Sama toiminnan ymmärtämisen tärkeys pätee kuin valintaan listan ja taulukon välillä.
- Tarvitaanko järjestystä? Muuttuuko alkioiden määrä? Onko hakuja vai lisäyksiä/poistoja enemmän? jne

8.1 Yksinkertaiset joukkomallit

⇒ Joskus perusjoukko (kaikki mahdolliset alkiot) on jokin äärellinen (suppea) kokonaislukuväli (tai muuten numeroituva):

- Voidaan käyttää bittivektoria (totuusarvotaulukkoa).

  if (B[i]) *kuuluu_joukkoon* **else** *ei_kuulu_joukkoon* 1

- *contains*, *insert* ja *remove* aikavaativuudet ovat $O(1)$.

- Muut $O(|\text{perusjoukko}|)$.
- Myös epäyhtenäinen kokonaislukujen äärellinen osajoukko onnistuu, kunhan se huomioidaan operaatioissa (tai talletusrakenteessa).

⇒ Jollei bittivektoria voida muodostaa (alkiota ei voi esittää kokonaislukuna, arvoalue on liian suuri, tms), niin alkiot on todella talletettava johonkin konkreettiseen kokoelmaan.

- Linkitetty lista
  - Yksi alkio(viite) yhteen listasolmuun.
  - Järjestetty tai järjestämätön.
  - Perusoperaatiot *contains*, *insert*, *remove*: $O(n)$, missä n = joukon koko.
  - Muut $O(n)$ (jos järjestetty) tai $O(n\log n)$, jos järjestämättä.
- Taulukko
  - Yksi alkio(viite) yhteen taulukon soluun (indeksiin).
  - Joko säilytettävä alkiot tiiviisti järjestettynä, tai indeksoitava suoraan (hajautus).
  - Tiiviisti järjestettynä: *contains* $O(\log n)$, *insert*, *remove* (ja muut) $O(n)$.
  - Vanhat alkiot järjestettynä, uudet sekaisin: perusoperaatiot $O(\sqrt{n})$ (tasoitettu).
  - Hajautuksella *insert*, *remove* $O(1)$, joukko-operaatiot $O(n+m)$ (kun n ja m ovat osallisten joukkojen alkiomäärät).
- (Binääri)puu
  - Yksi alkio(viite) yhteen puusolmuun.
  - Tasapainotettuna perusoperaatiot $O(\log n)$, joukko-operaatiot $O(n+m)$.

8.2 Joukkojen erikoistapaukset

Sanakirja

⇒ Keskitytään hakemiseen.

- Mahdollisesti myös lisääminen, ehkä poistokin tarvitaan.
- Ei kuitenkaan yhdiste, erotus, jne.
- Jos lisäyksiä tai poistoja ei tehdä, järjestetty taulukko on melko hyvä: $O(\log n)$ binäärihaulla.
- Jos tehokkaita lisäyksiä (tai/ja poistoja) tarvitaan, niin käytetään hajautusta tai tasapainotettua puuta.

8.3 Hajautus (*hashing*)

- Talletusalueena m -alkioinen taulukko tai tiedosto, indeksit $0..m-1$.
- Kullekin (lisättävälle/haettavalle) alkiolle lasketaan osoite talletusalueella (ns. kotiosoite).
  - Nopeasti avaimesta laskettava kokonaislukuarvo väliltä $0..m-1$.
  - Hajautusfunktio h (*hash function*). $0 \leq h(\text{avain}) < m$, siis käytännössä $h(\text{avain}) \mod m$.
  - Jotta alkiot löytyisivät hajautustaulusta, hajautusfunktion on annettava joka kerta sama osoite kaikille alkioille x ja y joille $x.equals(y)$ on tosi.
- Peräkkäinen järjestys ei (yleensä) säily.
  - Järjestetyt läpikäynnit (tai *min*, *max*) eivät suoraan onnistu, varsinkaan massamuistissa. (Kts. *java.util.LinkedHashSet*)
  - ”Satunnais”järjestyksessä läpikäynti onnistuu ajassa $O(n+m)$.

⇒ Ihannetilanne olisi jos kukin joukon alkio (tai kuvauksen avain) kuvautuisi hajautusfunktiolla eri osoitteeseen.

- Käytännössä tämä on hyvin epätodennäköistä jos/kun avain on esim. merkkijono (oma osoite kullekin avaimelle on teoreettisestikin mahdotonta).
- Käytännössä siis aina kaksi tai useampia alkiota kuvautuu samaan osoitteeseen ("törmää").
  ⇒ Törmäyksiin (*collision*) on **aina** varauduttava ja niiden vaikutukset on minimoitava.
  - Kaikki alkiot on pystyttävä tallettamaan jonnekin josta ne löytyvät myöhemmin!

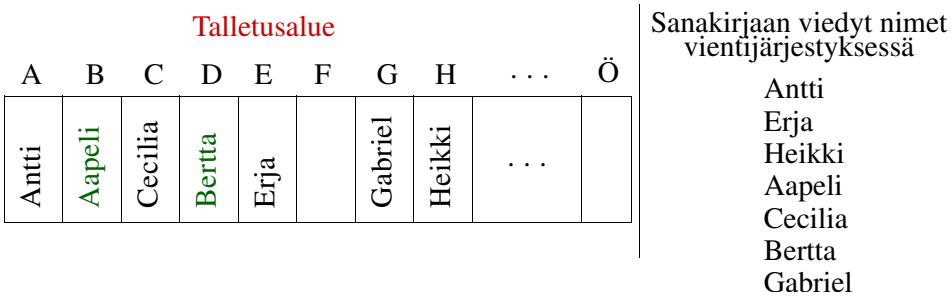
⇒ Törmäyksiä todella tulee:

- 23 arvoa satunnaisesti 365 lokeroon: törmäys todennäköisyydellä 50,73% (ns. syntymäpäiväparadoksi).
- n alkiota n lokeroon: johonkin lokeroon $\Theta(\log n)$ alkiota (suurella todennäköisyydellä).
- Hajautusfunktio yritetään valita siten, etteivät alkiot jakautuisi kovin epätasaisesti (todella montaa alkiota samassa (tai lähi-) osoitteessa samaan aikaan kun muualla runsaasti tyhjiä osoitteita).
  - Käytetään (lähes) koko avainta (tai sen eroavimpia osia).
  - Varmistetaan, että koko hajautusalue $(0..m-1)$ on käytössä.
  - Varotaan säännönmukaisuuksia aiheuttavia asioita (esim. jos m ja jokin hajautusfunktion kerroin ovat molemmat 2:n potensseja, niin ne saattavat "kumota" toisensa).
  - "Satunnaisuus" auttaa välttämään säännönmukaisuuden aiheuttamia katastrofeja.
- On kuitenkin teoreettisestikkin mahdotonta määritellä hajautusfunktiota josta tulisi satunnaisia arvoja epäsatunnaisesta syötteestä.

⇒ Törmäystapauksessakin alkio on sijoitettava jonnekkin (sellaiseen paikkaan josta se löytyy myöhemmin).

Suljettu hajautus (*closed hashing*, *open addressing*)

- Talletusalue (taulukko) $(0..m-1)$ sisältää varsinaiset alkiot/viitteet, ei käytetä muuta tilaa.
- Mikäli alkiota lisättäessä sen kotiosoite $h(\text{key})$ on jo varattu, alkio viedään lähinnä seuraavaan vapaaseen paikkaan.
  - Lähinnä seuraava: joko seuraava indeksi (*linear probing*), tai jokin muu yksiselitteinen sääntö, kts. kaksinkertainen hajautus alla.
- Jollei alkiota haettaessa löydy kotiosoitteestaan, alkio voi olla seuraavissa paikoissa yo. säännön mukaisesti.
  - Riittää tarkastella (joudutaan tarkastelemaan) seuraavaan tyhjään paikkaan asti (koska siihen alkio olisi sijoitettu jos olisi sijoitettu).

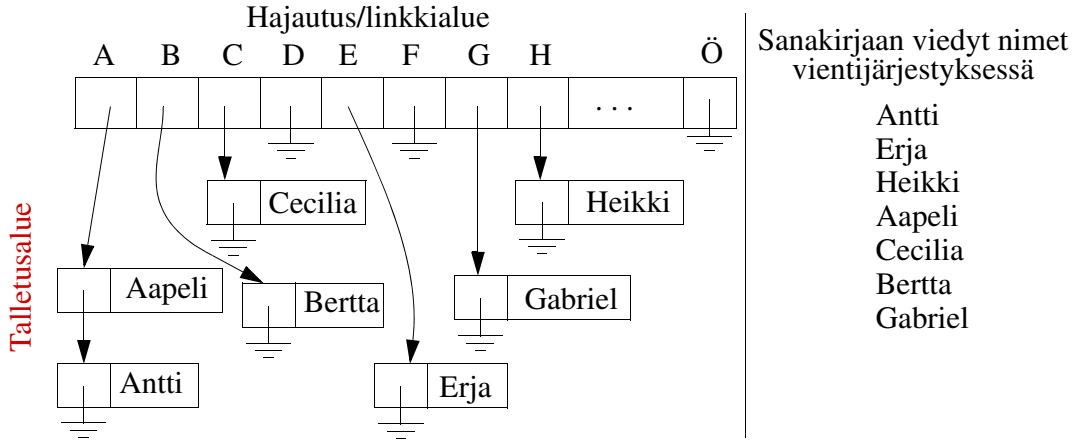


Kuva 8.1: Suljettu hajautus, hajautusfunktiona alkukirjain (hajautusfunktioksi huono).

- Poistaminen vaatii hieman huolellisuutta, joko:
  - Joutuisimme siirtelemään seuraavissa paikoissa olevia jos tarpeen (ei kovin kätevää).
  - Mieluummin jätetään poisto "tekemättä" ja vain merkitään alkio poistetuksi (ts. korvataan alkio poistettu-merkillä).
    * Haku ylittää poistetuksi merkityn paikan (ennen poistoa lisätty voi olla kauempana).

    * Lisäys tehdään ensimmäiselle vapaalle paikalle (poistetuksi merkitylle tai tyhjälle).
- Hyvää: yksinkertainen talletusrakenne (tiedostossakin).
  - Vaikka joudutaan selaamaan seuraavia lohkojakin, kyse on peräkkäiskäsittelystä, yhden levyhaun pitäisi riittää.
- Huonoa: jos talletusalue (liki) täyttyy, haut venyvät pahasti.
- Kun haut venyvät, hajautus kannattaa suorittaa uudelleen suuremmalle talletusalueelle ja/tai eri hajautusfunktiolla.
  - Hajautusaluetta ei saisi päästää lähellekkään täyteen
  - Yksinkertaisesti: uudelleenhajautetaan kun esim. $^1/_2$ alueesta on käytössä.
  - Lisäviritys: tarkkaillaan pisimpiä yhtenäisiä varattuja jaksoja.

Avoin hajautus (*open hashing*, *external hashing*, *chained hashing*)



Kuva 8.2: Avoin hajautus, ”hajautusfunktiona” alkukirjain.

- Kutakin kotiosoitetta kohti on lista johon kaikki ko. osoitteeseen kuuluvat alkiot sijoitetaan (järjestykseen).
  - Kukin alkion viedään aina omaan kotiosoitteeseen koska siinä on rajattomasti tilaa.
  - Hajautustaulun alkio on viite alkiolistan ensimmäiseen alkioon.
  - Tai: hajautustaulu sisältää yhden elementin, sekä viitteen listan toiseen solmuun (siis toiseen elementtiin). Taulukon alkio on siis samaa tyyppiä kuin listasolmu.
    * Tämä vie enemmän tilaa, mutta voimme välttää yhden muistiviittauksen kun alkio on suoraan taulukossa. Javalla tämä ei ole mahdollista, joten hajautustaulu sisältää vain viittauksia.
- Talletusalue on ”rajaton”: vaikka alkioita olisi enemmän kuin hajautusosoitteita, joskin haku luonnollisesti hidastuu kun listat pitenevät.
- Listat voi korvata vaikka puilla, mutta mieluummin käytetään suurempaa määrää kotiosoitteita listojen lyhentämiseksi.
  - Java8 *HashMap*: jos lista kasvaa pitkäksi (yli 8), lista muutetaan tasapainotetuksi binääripuuksi (jossa ehkä joudutaan tekemään alkioille keinotekoinen järjestys).
    * Hajautustaulun käyttöaste (*load factor*), eli keskimääräinen listan pituus pidetään kuitenkin alle 1:n, joten näitä pitkiä listoja jotka joudutaan muuttamaan puuksi tulee todella vähän (ellei hajautusfunktio ole huono).
- Muistihakuja 2 tai enemmän, riippuen toteutuksesta.

Hajautusfunktioiden suunnittelusta

- Koko avainta pitäisi käyttää funktion laskemiseen.
- Eri osat kerrotaan/bittisiirretään eri kertoimilla (jotta kaikki eivät summautuisi samaan kasaan).
- Vältä säännönmukaisuutta.
- Käytä alkulukuja kertoimina. Varmista, että kertoimet ja m ovat keskenään jaottomia.
- Varmista, että kaikki mahdolliset arvot $0..m-1$ ovat mahdollisia (ja suunnilleen yhtä todennäköisiä).
  - Myös kun m suurenee tai pienenee uudelleenhajautuksen yhteydessä.
- Kts. esim. *java.lang.String.hashCode*(), *java.util.HashMap.hash*(), *Arrays.hashCode*().

Uudelleenhajautus (*rehashing*)

- Hajautustaulun täyttyessä (tai ruuhkautuessa) joudutaan koko hajautustaulu rakentamaan uudelleen.
- Uusi talletusalue.
- Uusi hajautusfunkio (ainakin uusi m).
- Jokainen alkio joudutaan sijoittamaan erikseen uudelleen, vanhan hajautustaulun kopiointi ei riitä (koska m muuttui).
- Alkioiden järjestys muuttuu.

Kaksinkertainen hajautus

- Valitaan kaksi (tai useampia) täysin erilaista hajautusfunktiota.
- Jos ensimmäisen hajautusfunktion arvo antaa törmäyksen, lasketaan toisen funktion antama arvo ja talletetaan alkio sen osoittamaan paikkaan.
  - Jos toinenkin hajautusyritys antaa törmäyksen, käytetään seuraavia indeksejä (kuten edellä) tai jopa kolmatta hajautusfunktiota.
  - Voimme myös käyttää aluksi seuraavia osoitteita ja jonkin rajan jälkeen vaihtaa toissijaiseen hajautusfunktioon.
  - Kunhan funktioita käytetään aina samalla tavalla jotta alkio voidaan myös löytää.
- Yhden funktion mukaisen kasauman pitäisi hajota toisen funktion mukaan hajautettuna.
- Hakeminen vastaavasti.
- Suljetulla hajautuksella suoraviivaista (kuten seuraavan indeksin valinta).
- Avoimella (ketjutetulla) hajautuksella voidaan listan alussa pitää tietoa listan pituudesta, tietyn raja-arvon jälkeen käytetään toissijaista funktiota.

Satunnaistettu hajautus

- Käytetään satunnaisesti valittua hajautusfunktiota.
  - Jos se ei toimi hyvin, niin valitaan uusi satunnainen hajautusfunktio, uudelleenhajautetaan koko talletusalue.
- Suurella todennäköisyydellä uudelleenhajautusta tarvitaan niin harvoin, ettei se vaikuta keskimääräiseen aikavaativuuteen.
- Voidaan myös satunnaistaa kaksin- (tai usempi) kertainen hajautus.
- Sopii todistuksiin...

Fibonacci -hajautus

- Esimerkki kertolaskuhajautuksesta ja irrationaaliluvun käytöstä.

- Keino saada kokonaisluvut hajautumaan tasaisesti välille $0..m-1$.
- Luvut 0, 1, 2, ... hajautuvat välille [0..1) siten, että seuraava luku sijoittuu leveimmälle jäljelläolevalle tyhjälle välille (kultaisen leikkauksen suhteessa).
  - Kun luku kerrotaan m:llä saadaan jakauma välille $[0..m)$.

$$h(k) = \left\lfloor m \left(\left(\frac{\sqrt{5}-1}{2} k \right) \mod 1 \right) \right\rfloor \tag{8.1}$$

Kuvauksen toteutus

1. laskeminen
2. hajautus / puu (kuten joukko)
   - Joukon alkion sijaan kuhunkin talletuspaikkaan (puun solmu / suljetun hajautuksen taulukon indeksi / avoimen hajautuksen listasolmu) talletetaan avain-arvo -pari.
   - Muuten kuten joukon toteutus.

Laukun toteutus

- Kuten muutkin joukot, varmistaen, että kaikki samanarvoiset alkiot ovat löydettävissä.
- Pitääkö jokainen alkio tallettaa vai riittääkö samanlaisten alkioiden lukumäärän tallentaminen?
- Suljettu hajautus ei ole hyvä jos samaa alkiota voi olla (todella) monta.
  - Jos *equals*() on tosi, niin *hashCode*() on sama.
- Java8:n avoimen hajautuksen toteutuksen (jossa pitkät listat muutetaan binääripuuksi) tapainen toteutus sopisi hyvin.
- Poisto joko alkio kerrallaan, tai kaikki samat (huolellisesti).
  - Yleensä kaksi eri operaatiota.

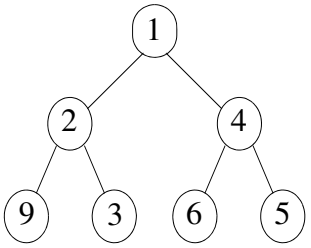
Prioriteettijonon toteutus

- Listalla/taulukolla: lisäys tai poisto $O(n)$.
- AVL-puulla (tms) lisäys, poisto ja minimi olisivat $O(\log n)$, mutta kiertoja tulisi tarpeettoman paljon.

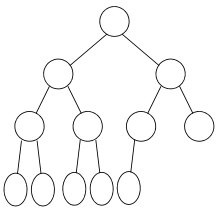
Näytetään ratkaisu joka on huomattavasti yksinkertaisempi ja tarjoaa pienemmän vakiokertoimen.

Tasapainoinen osittain järjestetty binääripuu (keko, kasa, *binary heap*)

- Osittain järjestetty:
  - Kunkin alkion prioriteetti on parempi (tärkeämpi) kuin lastensa.
    - Siten myös lastenlastensa.
  - Sisarusten järjestyksellä ei ole väliä.
  - Juuri on prioriteetiltaan paras.
  - Toiseksi paras on juuren jompikumpi lapsi, jne.
  - Huomaa, että tälläisestä puusta tietyn alkion etsiminen vie $O(n)$ ajan.



- Tasapainoinen:
  - Alinta tasoa lukuunottamatta kukin taso on täysi.
  - Alimman (vajaan) tason solmut ovat mahdollisimman vasemmalla.
  - Lisäyskohta on aina alimman tason oikea reuna (vasemmanpuoleisin vapaa kohta) (tosin siitä alkio saatetaan siirtää ylemmäs).
  - Poistokohta on aina alimman tason oikeanpuoleisin solmu.
    - Tosin sitä alkiota ei prioriteettijono poista vaan se siirretään poistetun juuren alkion paikalle.
  - Puun korkeus on $\lfloor \log_2 n \rfloor$.



Prioriteettijonon operaatiot kekototeutuksessa

removemin (*poll*)

1. Otetaan juuren alkio talteen (prioriteetiltaan paras alkio) ($O(1)$ aika).
2. Alimman tason oikeanpuoleinen alkio siirretään juuren paikalle (poistetaan solmu puusta) ($O(1)$, kts. alla).
3. Uusi juuri lasketaan alemmas: vaihdetaan aina prioriteetiltaan paremman lapsen kanssa kunnes vaihtoa ei enää tarvitse tehdä ($O(\log n)$). (*fixdown*)

add (*offer*)

1. Uusi alkio alimmalle tasolle uudeksi oikeanpuoleisimmaksi solmuksi ($O(1)$, kts. alla).
2. Uusi solmu nostetaan ylemmäs: vaihdetaan vanhemman kanssa jos vanhemman prioriteetti on huonompi kun uuden solmun, toistetaan kunnes ei enää tarvitse vaihtaa ($O(\log n)$). (*fixup*)

offerAndPoll() -operaatio (yhdistetty *removemin* ja *add*) on suunnilleen yhtä helppo kuin *removemin* (fixdown), joten jos sitä voidaan hyödyntää, se säästää hieman aikaa. Samoin toisin päin (*pollAndOffer*()).

Yksinkertaiset operaatiot, vakiokerroin on pieni.

Miten löydetään lisäys/poistokohta?

- Dynaamisessa toteutuksessa joko:
  - Ketjutetaan solmut tasoittain, tai
  - Pidetään yllä tietoa alimman tason oikeanpuoleisimmasta solmusta, tämän tiedon päivitys onnistuu $O(\log n)$ ajassa.

Taulukkototeutus

- Taulukko E $A[max+1]$ (indeksi 0 ei ole käytössä (tai jos on, niin kaikki indeksit alla: -1)).
- Juuri $A[1]$.
- Juuren lapset $A[2]$ ja $A[3]$.
- 2:n lapset $A[4]$ ja $A[5]$.
- 3:n lapset $A[6]$ ja $A[7]$.

- i:n vasen lapsi $A[2*i]$ ja oikea lapsi $A[2*i+1]$.
- i:n vanhempi $A[i/2]$ (kokonaislukujako).
- Alimmaisen tason viimeinen alkio on $A[n]$, kun puussa on n alkiota.

Taulukkototeutus on hyvin tehokas sekä tilan, että ajan suhteen.

- Vakiokertoimet pieniä.
- Puun korkeus ei aiheuta onglmia kun puu on täysin tasapainoinen.

Esimerkki 8.1. Kekojärjestäminen (*heap sort*) onnistuu $O(1)$ aputilalla. Taulukon alkioita prioriteettijonoon vietäessä taulukon kasvavaa alkuosaa käsitellään kekona (puuna) ja taulukon kutistuvaa loppuosaa jäljellä olevien alkioiden taulukkona, jonka koko pienenee sitä mukaa kun alkuosan koko kasvaa. Kun kaikki alkiot ovat prioritettijonossa, on koko taulukko keon talletusaluetta.

Vastaavasti alkioiden prioriteettijonosta poistamisen myötä keon talletusalue pienenee ja järjestyksessä oleva taulukon osa kasvaa. Alkiot kannattaa tällöin poistaa prioriteettijonosta suurimmasta pienimpään (miksi?).

Sopivasti alkiot sekä taulukosta-keoksi, että keosta-taulukoksi ovat oikeassa paikassa kunkin operaation alkaessa. Niinpä riittää tehdä vaihto keosta-taulukoksi ja sitten fixup/fixdown kullekin alkiolle.

8.4 Järjestetyn joukon toteutus hakupuulla

⇒ Keskitytään haku-, lisäys- ja poisto-operaatioihin.

- Kaikki $O(\log n)$ ajassa.
- Dynaaminen joukon koko.
- Mielivaltaisen tyyppinen alkio (*compareTo*() tai *Comparator* riittää).
- Järjestyksessä läpikäynti $O(n)$ kokonaisajassa ($O(1)$ keskimäärin), (yksittäinen *next*() on $O(\log n)$ ellei tehdä erikoisratkaisuja).
- Läpikäyntejä käyttäen joukko-operaatiot yhdiste, leikkaus, erotus $O(n)$ ajassa.

⇒ Sisäjärjestetty (jokseenkin) tasapainoinen (*balanced*) binääripuu.

- Hakeminen on helppoa (käsitelty aiemmin).
- Lisääminen sinänsä on helppoa, mutta puu voi vinoutua ja siten kasvaa liikaa korkeutta!
  - Tasapainon ylläpitäminen vaatii hieman lisää vaivaa, mutta onnistuu kuitenkin $O(\log n)$ ajassa.
  - Täydellisen tasapainon sijaan riittää kohtuullinen tasapaino, kunhan korkeus on $O(\log n)$ (esimerkiksi korkeintaan 1,5 tai 2-kertainen).
- Lähestymistapoja useita, useimmille yhteistä on pienen epätasapainon sietäminen, heti isomman epätasapainon ilmetessä tasapainotetaan.

AVL-puu [Adelson-Vel'skij & Landis]

⇒ Minkään solmun vasemman ja oikean alipuiden korkeusero (tasapaino) ei saa olla yhtä suurempi.

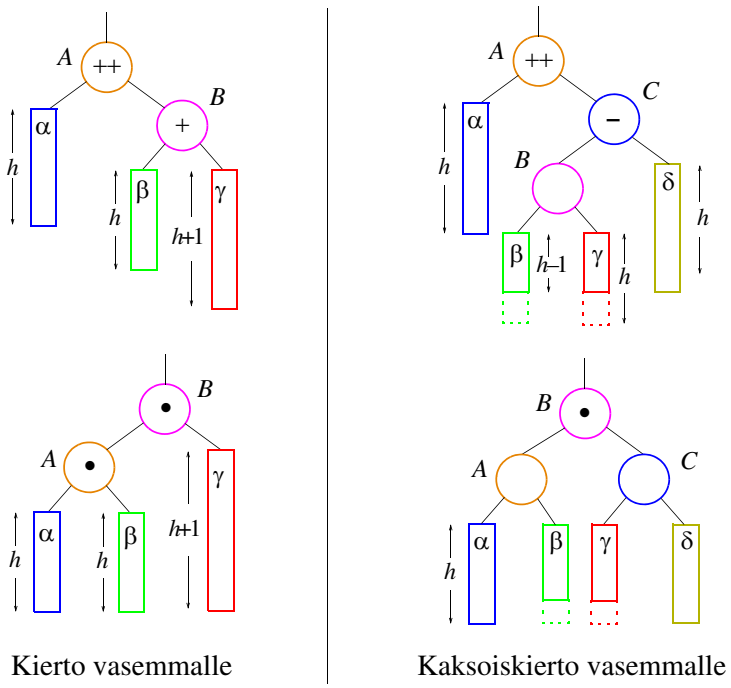
Solmun tasapaino:

- Kussakin solmussa pidetään yllä tasapainotietoa (**ei** korkeustietoa):
- Tasapainot päivitetään alhaalta ylös (vain!) polkua ylöspäin lisättäessä tai poistettaessa solmu:
  - Jos solmun tasapaino muuttuu lisäyksessä 0:sta +:ksi tai −:ksi, (siis jommankumman (muttei matalamman) alipuun korkeus kasvaa) muuttuu myös vanhemman tasapaino sen mukaan onko tämä solmu vasen vai oikea lapsi.

  - Jos tasapaino muuttuu +:sta tai −:sta 0:ksi, muutos pysähtyy (korkeus ei kasva).
  - Poistettaessa vastaavasti jommankumman alipuun korkeus saattaa vähentyä.
- Huomaa, että korkeuksia ei lasketa, vain korkeuseroja (tasapainoja).
  - Korkeuseroja päivitetään paikallisesti, vakioajassa, koskaan ei tarvitse laskea koko (ali)puuta uusiksi.

⇒ Tasapaino on rikkoutunut, jos se on −− tai ++ (−2 or +2).

- Jos solmun tasapaino lisäyksen tai poiston myötä menetetään, korjataan tasapainoa (puuta) vain kyseisen solmun kohdalta.
  - Suoritetaan kierto epätasapainoisen solmun suhteen.
  - Kiertoja on kahta erilaista (ja kumpaakin kahteen suuntaan):
    * Jos solmun tasapaino on −− ja vasemman lapsen tasapaino on −, suoritetaan kierto oikealle.
    * Jos solmun tasapaino on −− ja vasemman lapsen tasapaino on +, suoritetaan kaksoiskierto oikealle (eli vasen-oikea -kierto).
    * Jos solmun tasapaino on ++ ja oikean lapsen tasapaino on +, suoritetaan kierto vasemmalle.
    * Jos solmun tasapaino on ++ ja oikean lapsen tasapaino on −, suoritetaan kaksoiskierto vasemmalle (eli oikea-vasen -kierto).
- Kierto pysäyttää tasapainovirheen ”nousemisen”, yksi kierto riittää aina lisäyksessä.
- Muiden kuin kiertoon osallistuvien solmujen tasapainoon ei kosketa.
- Huomaa, että kaikki kierrot onnistuvat vakioajassa (4-8 viittauksen päivitystä)!
  - Puuta ei koskaan rakenneta uusiksi (vaikka piirrettäessä siltä näyttäisi...).
- Poisto vastaavasti $O(\log n)$ ajassa (tosin saatetaan tarvita kierto usealla tasolla).



Kuva 8.3: AVL-puun kierrot

Muita vastaavia

- Punamusta puu (katso CLR, JavaAPI).
- B-puu (erityisesti massamuistiin, katso TRA II).

- 2-3 puu (B-puun erikoistapaus keskusmuistiin).
- ”Itsetasapainottuva” puu, Splay -puu, jne.

Muita puita

- Paitsi vakioon 2, voidaan lapsien määrä rajoittaa mihin tahansa muuhunkin vakioon (tai jopa olla rajoittamatta).
- Kätevä on esimerkiksi aakkoston koko:
  - Kussakin puun haarassa on taulukkona lapset (esim) a..ö
  - Varsinainen tieto tallennetaan (viittauksena) syvyydelle *avaimen_pituus* puussa.
  - Hakeminen aina O(avaimen_pituus)
  - Kullakin avaimella on yksikäsitteinen paikka puussa.
  - Jos avaimia on paljon (suhteessa niiden pituuteen), ei lisätilaa mene paljoakaan.

50

8.5 Läpikäyntien toteutus

- Kts. iteraattorit s. 27, s. 55.
- Jotta luokka (yleensä kokoelma) toteuttaa *Iterable* -liittymän, sillä on oltava metodi

```
Iterator<E> iterator()
```
1

- joka palauttaa *Iterator* -liittymän toteuttavaa luokkaa olevan objektin jolla kokoelma voidaan käydä läpi.
  - *Iterator* -liittymä edellyttää operaatiot *hasNext*() ja *next*() (ja *remove*(), jonka tosin ei tarvitse toimia).
- *Iterator* -liittymän toteuttava luokka voidaan/kannattaa sijoittaa kokoelman sisäluokaksi, jolloin sen käyttöä voidaan kontrolloida.
  - Tällöin myös ko. iteraattoriluokka voi käyttää kokoelman metodeja ja jäseniä (joita se väistämättä tarvitsee toimiakseen).
  - Kokoelman metodi *iterator*() (vain) luo (*new*) ja palauttaa uuden ilmentymän tästä luokasta.
  - Tämä *Iterator*:n toteuttava luokka ei ole erikseen tyyppiparametroitu, vaan se käyttää isäntänsä tyyppiparametria.

iterator(), *hasNext*() ja *next*() käyttävät hyväksi talletusrakennetta:

- Iterointimuuttuja viittaa tavalla tai toisella kokoelman viimeksi *next*():llä palautettuun alkioon (tai/ja joskus sen seuraajaan).
  - Taulukon indeksi, puun/listan solmun viite, jne.
- *iterator*() hakee viitteen ensimmäiseen (tai mihin tahansa) solmuun.
- Käyttäjän kannalta iteraattori osoittaa alkioiden väliin, *next*() ”siirtää” iterointimuuttujaa eteenpäin seuraavaan väliin ja palauttaa ylihypätyn alkion.
  - Mutta kun väliin ei voi toteutuksessa viitata, niin viitataan jompaan kumpaan tai molempiin.
- Käyttäjän kannalta *hasNext*() tutkii, onko iterointimuuttujan osoittama alkio luvallinen (ts. onko seuraavassa asemassa oikeasti alkiota).

Listoilla, taulukoilla helppoa, kts. *ArrayList2.java*, *JLaukku.java*.

Hajautuksessa joudutaan huomioimaan tyhjät kohdat (ja ketjutetussa hajautuksessa läpikäymään listat) (HT).

- Jos halutaan vakioaikainen *next*() ja järjestyksen säilyvän uudelleenhajautuksessa, linkitetään alkiot listalla (kts. *java.util.LinkedHashSet*).

Puussa muistetaan viimeksi annettu/seuraavaksi annettava solmu ja "kiipeillään" puussa *sisäSeuraaja*() -operaatiolla (s. 49).

Toteutuksessa:

- Jos talletusrakenne on selkeä, riittää ylläpitää viittausta viimeksi *next*:llä palautettuun asemaan (solmu tai indeksi).
  - *hasNext*() pystyy siitä päättelemään onko seuraavaa asemaa (alkiota) olemassa.
  - *remove*():n oikeaa toimintaa varten tarvitaan yleensä lisäviittaus, kts. alla.
- Jos talletusrakenne ei ole selkeä, voi olla "vaikea" nähdä onko tämänhetkinen alkio viimeinen vai ei.
- Näinollen, joskus helpointa on jos (konstruktori ja) *hasNext*() hyppivät tyhjien kohtien yli ja etsivät seuraavan alkion.
  - *next*() vain palauttaa tämänhetkisen alkion.
    - Jos käyttäjä kutsuu *next*():iä ilman *hasNext*():iä, *next*() joutuu tekemään siirtymisen (kutsumaan itse *hasNext*():iä).
  - Poisto on helppoa, tosin joudutaan ylläpitämään tietoa viimeisestä ylihypätystä (koska käyttäjä saattaa kutsua *next*() – *hasNext*() – *remove*()).
  - Kts. *JLaukku.java*.
- Vaihtoehtoisesti ylläpidetään viittausta sekä edelliseen, että seuraavaan alkioon.
  - *next*() (ja konstruktori) ottaa seuraavan edelliseksi ja etsii uuden seuraavan (jos löytyy). *hasNext*() on triviaali.
  - Katso esim. *java.util.TreeMap.SubMapIterator*.
- *remove*() poistaa vain viimeksi *next*():llä ylihypätyn alkion. Kaikki muu heittää *IllegaStateException* (esim kaksi *remove*:a peräkkäin).
  - Helpointa on ylläpitää kahta viittausta (edellinen-seuraava tai poistettava-seuraava) jolloin *remove*() nollaa niistä toisen.

Puiden läpikäynti

- Suoraviivainen sisäjärjestys (tai esi-, jälki-, tasoittainen).
- *iterator* (ts. *Iterator*:n toteutuksen konstruktori) hakee puun ensimmäisen solmun.
- *hasNext*() hakee seuraajan puussa, *next*() palauttaa sen.
  - Koko läpikäynti $O(n)$ ajassa, (binääripuussa $2n$ vanhempi-lapsi -suhteen seuraamista).
  - Tasapainoisessa puussa kaksi *next*:tiä kestää n. log$n$ ajan, muut vähemmän, keskimäärin joka tapauksessa $O(1)$.

Tehostuksia: esim:

- Käydään läpi tasoittain, ylläpidetään jonoa iteraattorissa.
  - Ei enää (sisä)järjestyksessä.
- Jos järjestyksessä läpikäytäessä halutaan kaikkien *next*-operaatioiden olevan vakioaikaisia: Linkitetään puun solmut valmiiksi haluttuun järjestykseen (ylimääräinen *next*-viittaus kussakin solmussa).
  - Ylläpito vaikeutuu (mutta onnistuu edelleen logaritmisessa ajassa).

Kokoelman muuttumisen havaitseminen kesken läpikäynnin

- Javan kokoelmat käyttävät ns. *FailFast* -lähestymistapaa jos kokoelmaa muutetaan kesken läpikäynnin muuten kuin läpikäynnin omilla metodeilla.
  - Jos läpikäynti havaitsee läpikäyntiin vaikuttavan edes mahdollisen muutoksen, se heittää *ConcurrentModificationException*:n.
- Kullakin kokoelmalla on sisäinen jäsen *int modCount* jota kasvatetaan aina kun jokin (talletusrakenteeseen vaikuttava) muutos (*add*(), *remove*(), tms.) tehdään (muutoksia voidaan tehdä vain luokan omilla metodeilla).
  - Kukin iteraattori ylläpitää omaa kopiota alustuksessa havaitsemastaan *modCount*:n arvosta.
  - Jos arvo on muuttunut, tiedetään jossain muualla tehdyn muutoksen ja heitetään poikkeus.
  - Iteraattorin omat rakennetta muokkaavat metodit (*add*(), *remove*()) päivittävät iteraattorin oman *modCount*-kopion arvoa.
- Muutos toki havaitaan vasta (ja vain) jos ko. iteraattoria vielä käytetään muutoksen jälkeen.
  - Jollei iteraattoria enää käytetä, muutos on luvallinen.
- Talletusrakennetta muuttamattomissa operaatioissa (esim *set*()) *modCount*:ia ei tarvitse/pidä päivittää.

Kokoelman muuttaminen muualta kesken läpikäynnin

- Salliminen vaatii huolellisuutta ja tiukkoja ehtoja.
  - Ja/tai varustautumista talletusrakenteessa.
  - Ja/tai luopumista läpikäynnin täsmällisestä toiminnasta.
  - Kokoelman on pystyttävä päivittämään iterointimuuttujan arvoa kokoelmaa muutettaessa (*insert*, *remove*).
  - Käytännössä kaikki aktiiviset läpikäyntimuuttujat on talletettava (tai linkitettävä) itse kokoelmassa.
  - Usean samanaikaisen läpikäynnin mahdollistamiseksi tarvitaan kokoelmassa esim. läpikäyntien lista.

Joukkojen yhdistäminen jne

- *union*, *intersection*, *difference*, *removeAll*, *retainAll*, *addAll*, jne.
- Suoraviivainen *foreach* – *add*/*remove* toimii hajautustaulussa keskusmuistissa, puullakin kohtuullisesti, mutta puulle se ei ole tehokkain mahdollinen.
- Puutoteutuksessa kannattaa lukea lähdejoukot järjestyksessä listaan tai taulukkoon ($O(n)$ aika), ”lomittaa” (operaation mukaisesti) listat ($O(n)$) ja luoda uusi talletusrakenne optimaaliseksi (tasapainoiseksi) ($O(n)$).
  - Jos tietorakenne tukee järjestyksessä läpikäyntiä (esim puu), voidaan ”lomitus” tehdä jo läpikäyntivaiheessa.
  - Tasapainoinen puu on helppo muodostaa taulukosta lineaarisessa ajassa, mutta se on mahdollista myös listasta.
  - Joukko-operaatioiden kokonaisaika paranee $O(n\log n) \Rightarrow O(n)$, käytännössä tosin hyöty voi olla aika pieni.

8.6 Verkkojen toteuttaminen

⇒ Muodollisesti verkko koostuu solmujen ja kaarten joukoista, joten toteutuskin voisi käyttää joukko-operaatioita.

- Käytännössä kuitenkin vahvoja joukko-operaatioita (yhdiste jne) ei juuri tarvita verkkoja käsiteltäessä. Niinpä yksinkertaisempikin toteustapa riittää.
  - Solmut tallennetaan listaan tai taulukkoon (joskus ehkä puuhun).
  - Kaaret tallennetaan solmuittain listaan
- Vaihtoehtoisesti tallennetaan erikseen solmujen taulukko ja kaarien matriisi.
- Kts. TRA II.

9 Kirjallisuutta

1. Aho A. V., Hopcroft J. E., Ullman J.D.: Data Structures and Algorithms. Addison-Wesley, 1983.

2. Cormen T. H., Leiserson C. E., Rivest R. L.: Introduction to Algorithms. MIT Press 1990.

3. Hämäläinen A.: Tietorakennekirjasto Javalla. Joensuun Yliopisto, Tietojenkäsittelytieteen laitos, 2005. Ilmestyy pian.

4. Knuth D. E.: The Art of Computer Programming, Volumes 1-3, (2-3ed). Addison-Wesley, 1997-1998.

5. Sun Microsystems: JavaTM 2 Platform Standard Edition 5.0 API Specification. https://java.sun.com/j2se/1.5

6. Weiss M. A.: Data Structures and Algorithm Analysis in C. Addison-Wesley, 1997.

7. Weiss M. A.: Data Structures and Algorithm Analysis in Java. Addison-Wesley, 1999.

Olemassaolevia tietorakenne/algoritmikirjastoja

- https://www.cs.sunysb.edu/~algorith/
- https://www.mpi-sb.mpg.de/LEDA/
- C++
- 100+ luokkaa
- Kaupallistettu
- Eiffel, Oberon, Modula-3 -kirjastot tkt-laitoksella.
- Numerical Recipes in [Fortran [77|90] |C | Pascal(| Basic | Lisp | Modula-2 | IDL | Mathcad]
- https://www.nr.com/
- Animointeja: Jeliot, Polka, XTango, jne.
- TR&A oppi/käsikirjat, erityisesti Cormen, Leiserson, Rivest
- https://java.sun.com/j2se/1.5.0/docs/api/.