Tietorakenteet ja Algoritmit luentomoniste

Kari Peisa (kari.peisa@kpeisa.fi)
Ramk/Tekniikka ja liikenne 2009

Oheinen moniste on luentojen oheislukemistoksi tarkoitettu kurssille Tietorakenteet ja Algoritmit Rovaniemen amk:ssa ennen v 2012, jolloin kyseinen kurssi oli samalla C-ohjelmoinnin jatkokurssi. Sen vuoksi tietorakenteiden koodiesimerkit on toteutettu C-kielellä. Monisteessa pyritään käsittelemään tietorakenteita ja niihin liittyviä algoritmeja niin, että sekä teoreettinen pohja että niiden implementointi C-kielellä tulisi ymmärretyksi.

Sisältö

Tietorakenteet ja algoritmit

Kari Peisa

Ramk/Tekniikka ja liikenne 2009

kari.peisa@ramk.fi

1 Algoritmit

Tällä kurssilla **algoritmilla** tarkoitetaan hyvin määriteltyä tietokoneen suorittamaa ohjelmaa, joka vastaanottaa syötteenä arvon tai joukon arvoja (input) ja suorituksen päätteeksi tulostaa arvon tai joukon arvoja (output). Hyvin määritelty tarkoittaa sitä, että algoritmit palauttavat tuloksen kaikille tietyt kriteerit täyttäville syötejoukoille.

Tehokas algoritmi on ratkaiseva työkalu monissa tietokoneen käsittelyä vaativissa tehtävissä. Tallennettaessa ja analysoitaessa tietokannassa sellaista informaatiota kuten esimerkiksi ihmisen DNA, jossa esiintyy yhteensä yli 3000 miljoonaa koodikirjainta, on tiedon tallentaminen ja käsittely tehtävä erittäin tehokkailla algoritmeilla – tavallisella peräkkäiskäsittelyllä tehokkainkaan tietokone ei olisi käyttökelpoinen.

Internetissä olemme tottuneet hakemaan tietoa hakukoneella useiden miljoonien web-dokumenttien joukosta muutaman sekunnin viiveellä. Esimerkiksi Googlella tämän mahdollistaa tehokas Pagerank –algoritmi, joka perustuu todennäköisyys-laskennan matematiikkaan.

Jokainen ohjelmoija tarvitsee jossakin vaiheessa lajittelualgoritmia. Tehokkaita lajittelualgoritmeja on kehitetty lukuisia. On vain ymmärrettävä niiden periaatteet ja ominaisuudet, jotta niitä voi parhaiten hyödyntää ohjelmointiongelmien ratkaisuissa. Lajittelussa annetaan joukko arvoja $\{a_1, a_2, \ldots, a_n\}$, joille määritellään jokin järjestysrelaatio. Algoritmi, jos se toimii oikein, palauttaa arvot järjestysrelaation mukaisesti lajiteltuna $\{a^*_i\}_{i=1}^{n}$, missä $a^*_1 \leq a^*_2 \leq \ldots \leq a^*_n$.

Algoritmin suunnittelussa joudutaan tarkastelemaan sellaisia kysymyksiä kuin

- Onko ongelma ylipäätään ratkaistavissa sellaisella algoritmilla, joka suoriutuu tehtävästään jossakin järkevässä äärellisessä ajassa?
- Mikä on algoritmin suorituskyky eli kuinka suuria aineistoja se voi käsitellä?

Monien ongelmien ohjelmallisessa ratkaisemisessa ei riitä pelkästään toteutuskielen perustietotyyppien ja ohjelmarakenteiden hallinta. On hyvin tärkeää tietää mm.

- erityyppisiin ongelmiin soveltuvista algoritmeista ja niiden kompleksisuudesta
- algoritmeihin hyvin soveltuvista abstrakteista tietorakenteista
- algoritmin suorituksen analyysistä ja mittaamisesta.

Abstraktit tietorakenteet määritelevät tietorakenteen perusidean eivät niiden yksityiskohtaista toteutusta. Yleensä abstraktin tietotypin yhteydessä määritellään käsiteltävän tiedon tyyppi sekä käsittelymetodien ulkoiset rajapinnat. Tässä kurssissa esitellään ja toteutetaan yleisimmistä tietorakenteista **taulukot, listat**, **pinot**, **jonot** ja **puut** sekä perehdytään hieman myös **verkkoihin**. Tietorakenteita toteutetaan sekä **staattisina ohjelmarakenteina** esimerkiksi C-kielen taulukon avulla kuin myös **dynaamisina linkitettyinä ohjelmarakenteina**, joiden fyysinen tila muotoutuu ohjelman ajon aikana käyttäjän antamien syötteiden perusteella.

Algoritmin analyysissä vertaillaan erityyppisten algoritmien suorituskykyä niiden vaatiman muistitilan ja/tai käsittelyajan suhteen.

1.1 Algoritmien analyysi

Algoritmien analyysissä vertaillaan erityyppisten algoritmien suorituskykyä niiden vaatiman muistitilan ja/tai käsittelyajan suhteen sekä esitellään yleiset **algoritmien kompleksisuusluokat**. Tässä kappaleessa tutkitaan lisäksi **iteratiivisen** (ei-rekursiivisen) ja **rekursiivisen toistoalgoritmin** vahvuuksia ja toisaalta ongelmallisuutta.

Algoritmin suorituskykyä voidaan arvioida analysoimalla algoritmin rakenteita. Suorituskykyä voidaan myös mitata laskemalla vastaavien ohjelmien vaatiman muistitilan tai käsittelyajan kasvua suhteessa syötteenä annetun joukon kokoon.

Algoritmin tilakompleksisuus (space complexity) tarkoittaa sitä ohjelman muistitilan määrää, jonka ohjelman suorittaminen loppuun vähintään vaatii.

Algoritmin aikakompleksisuus (time complexity) tarkoittaa ohjelman suorituksen vaatimaa laskenta-aikaa (computer time).

Usein kompleksisuus ilmoitetaan erikseen parhaimmassa, keskimääräisessä tai pahimmassa tapauksessa. Tällöin syötejoukon fyysinen tila vaikuttaa oleellisesti algoritmin suorituskykyyn. Esimerkiksi melkein järjestyksessä olevan lukujoukon {5, 1, 2, 3, 4} lajitteleminen suuruusjärjestykseen voi algoritmin toteutuksesta riippuen olla huomattavasti kompleksisempi suoritus kuin täysin epäjärjestyksessä olevan lukujoukon { 2, 1, 5, 4, 3} (esimerkkiä tarkastellaan pikalajittelun yhteydessä).

Ohjelmien aikakompleksisuutta voidaan mitata vaikka rannekellolla. Fiksumpi tapa on kuitenkin kirjoittaa ohjelmaan sopiviin kohtiin toteutuskieleen kuuluvia kirjastofunktioita, jotka palauttavat tietokoneen keskusyksikön ylläpitämän kellonajan. Voi myös valmistaa itse apufunktioita, jotka sopiviin kohtiin sijoitettuna palauttavat algoritmin suoritusvaiheesta erilaisia lokitietoja.

Ohjelmien tilakompleksisuutta mitataan laskemalla sen sisältämien **ohjelma-askeleiden** lukumäärä. *Ohjelma-askel on sellainen ohjelman osa, jonka suoritusaika ei riipu ohjelmalle annetuista syötteistä*. Esimerkiksi C-kielessä yleisiä vähintään yhden askeleen ohjelman osia ovat mm:

- lausekkeet (ei funktion kutsu) ja sijoituslause (1 askel)
- ohjausrakenteet, joissa askelluku määräytyy rakenteen ohjausosissa ja lohkoissa esiintyvien lausekkeiden ja sijoituslauseiden lukumäärän mukaan

- funktiokutsut (1 askel + parametrien luku (arvovälitys) + funktion sisältämät ohjelma-askeleet). Funktioiden suorituksessa on lisäksi huomioitava paikallisten muuttujien vaatiman tilanvaraus, jota myös pidetään yhtenä ohjelma-askeleena.

Ohjelma-askeleiden lukumäärä voidaan laskea esimerkiksi **globaalin laskurimuuttujan** avulla tai analysoimalla ohjelman koodia.

Algoritmi voidaan määrittää hyvin monella eri tavalla. Kaikkein vapaamuotoisin esitys on puhdas lausekieli. Yleensä käytetään kuitenkin ns. **pseudokoodia**, joka ei ole varsinaisesti minkään ohjelmointikielen mukaista koodia vaan ainoastaan muistuttaa yleisten ohjelmointikielten rakenteita. Algoritmin suoritusvaiheet voidaan myös esittää numeroiduin ja sisennetyin kappalein. Yksinkertaiset algoritmit voidaan kuvata myös suoraan jonkin ohjelmointikielen mukaisella syntaksilla tai erityppisillä lohkokaavioilla.

Esimerkki: Vaihtolajittelualgoritmi ja sen toteutus C-funktiona

Algoritmi:

1. Valitaan listan `luvut` ensimmäinen kokonaisluku ja merkitään se indeksillä *i*.
2. Jos indeksi *i* on listan viimeinen, lopetetaan. Muutoin merkitään ja valitaan indeksi $j = i+1$.
3. Jos `luvut[`*i*`] > luvut[`*j*`]`, vaihdetaan lukujen paikat listassa.
4. Jos indeksi *j* on listan viimeinen, kasvatetaan indeksiä *i* yhdellä ja mennään kohtaan 2. Muutoin kasvatetaan indeksiä *j* yhdellä ja mennään kohtaan 3.

Pseudokielinen toteutus:

```
VaihtoLajittele(taulu,koko)
      for i←1 to koko-1
            for j←i+1 to koko
                  if taulu[i]>taulu[j]
                  then
                        temp ← taulu[j]
                        taulu[j] ← taulu[i]
                        taulu[i]← temp
```

Yllä oleva algoritmi ei ole kovin tehokas lajittelualgoritmi ja toimii aina samalla tavalla (samalla kompleksisuudella) riippumatta syötetaulukon järjestyneisyyden tilasta. Myöhemmin tutustutaan kehittyneempiin lajittelualgoritmeihin ja tarkastellaan myös niiden kompleksisuutta.

C-ohjelmassa aikakompleksisuus voidaan mitata käyttämällä ohjelmistoympäristössä (**Microsoft Visual C++ .NET**) määriteltyjä kellofunktioita, jotka on esitetty esimerkiksi moduuleissa `<time.h>` tai `<sys/timeb.h>`. Ala on esitetty esimerkki jälkimmäisen moduulin sisältämien funktioiden ja tietorakenteiden käytöstä.

```
/* otetaan käyttöön kirjastofunktiot*/

#include <sys/timeb.h>

/* esitellään tietuemuuttujat, joiden rakenne on määritelty
kirjastossa <sys/timeb.h> */

_timeb t1,t2;

…

/* lisätään ohjelmaan funktion _ftime(struct _timeb *timeptr)
kutsut ennen ja jälkeen lajittelufunktion kutsua. Funktio _ftime
tallentaa systeemin kelloajan sekunneissa alkaen hetkestä
00:00:00 vuonna 1.1.1970 tietuerakenteisiin t1 ja t2*/

_ftime(&t1);

vaihto_lajittele(taulukko,koko);

_ftime(&t2);

/* lasketaan lajitteluun kulunut aika sekunneissa millisekunnin
tarkkuudella palauttamalla tietueista t1 ja t2 kenttien time ja
millitm sisällöt*/

double t = t2.time+0.001*t2.millitm-t1.time-0.001*t1.millitm;
```

1.2 Asymptoottinen kompleksisuus

Algoritmin suorituskyvyn merkitys tulee esille vasta suuremmilla syöteaineistoilla. Olemme siis kiinnostuneita algoritmin suoritusajan kasvunopeudesta suhteessa syöteaineiston kokoon. Tätä kasvunopeutta kuvataan **vertailufunktioiden g(N)** avulla, missä N on syöteaineiston koko. Tyypillisiä vertailufunktioita ovat tehokkuutensa mukaisessa järjestyksessä:

1. Vakiofunktio g(N) = 1
2. Logaritminen funktio g(N) = log N
3. Lineaarinen funktio g(N) = N
4. Funktio g(N) = N log N
5. Polynomimuotoinen funktio, g(N) = N^2, N^3, …
6. Eksponentiaalinen funktio g(N) = a^N, missä a > 1

Kahdesta vertailufunktiosta pienemmäksi katsotaan se, joka saa pienempiä arvoja suurilla N:n arvoilla. Tällöin

$$1 < \log N < N < N \log N \;\ll\; N^2 < N^3 \ldots \;\ll\; a^N,$$

missä << tarkoittaa, että kompleksisuus nimenomaan suurilla N:n arvoilla on merkittävästi pienempi.

Seuraavassa merkintä T(N) tarkoittaa algoritmin suorituksen vaatimaa aikaa syöteaineistolla N ja g on jokin edellisistä vertailufunktioista. Algoritmin **kompleksisuusluokka** on **O(g),** mikäli löytyy sellainen syöteaineiston koko N_0 ja vakio c, että algoritmin suoritusaika noudattaa ehtoa

$$\mathbf{T(N) \le c\, g(N)}$$

aina kun $N \geq N_0$. Vakio c kuvaa tietokoneen tehokkuutta eikä se vaikuta kompleksisuusluokan määräytymiseen.

Kompleksisuusluokkien vertailuesimerkki:

Esimerkki. Olkoon tietokone A 100 kertaa tehokkaampi kuin tietokone B. Esimerkiksi jos B:n tehokkuus on 10^9 konekielistä käskyä/s, niin A:n tehokkuus on 10^7 konekielistä käskyä/s.

Tietokoneessa A ajetaan lajittelualgoritmia, jonka kompleksisuusluokka on $O(N^2)$ ja B:ssä parasta mahdollista lajittelualgoritmia, jonka kompleksisuusluokka on O(N log N). Oletetaan, että A:lle c = 2 ja B:lle c=50. Lasketaan lajitteluaika, kun lajiteltavana on ensin 100 alkiota, 10 000 ja sitten miljoona alkiota ($N=10^6$).

A: $\frac{2\cdot(100)^2}{10^9\frac{1}{s}} = 0.00002\ s$ B: $\frac{50\cdot 100\cdot log 100}{10^7\frac{1}{s}} = 0.0023\ s$

A: $\frac{2\cdot(10000)^2}{10^9\frac{1}{s}} = 0.2\ s$ B: $\frac{50\cdot 10000\cdot log 10000}{10^7\frac{1}{s}} = 0.4605\ s$

A: $\frac{2\cdot(10^6)^2}{10^9\frac{1}{s}} = 2000\ s$ B: $\frac{50\cdot 10^6\cdot log 10^6}{10^7\frac{1}{s}} = 69.078\ s$

Voit kokeilla eri tietokoneen tehokkuutta kuvaavilla arvoilla c, jolloin huomaat, että suurilla aineistomäärillä sillä ei ole suurta merkitystä suoritusajan kasvussa.

Alla on laskettu eri vertailuaikoja, kun samankokoinen aineisto, käsitellään kompleksisuusluokkia vastaavilla algoritmeilla ja tietokone kykenee suorittamaan miljardi operaatiota sekunnissa.

O(g)	aika	
log N	n. 0,000012 sek.	Hyvät algoritmit!
N log N	1,2 sek.	
N^2	2,8 tuntia	
N^3	31,7 vuotta	Käyttökelvottomia!
2^N	$3.2\cdot 10^{30089} \cong \infty$	

Algoritmeissa, joiden suoritusaikaa kuvaa polynomi, kompleksisuusluokan määrittelee aina polynomin korkeimman termin asteluku. Tämä voidaan todeta esimerkiksi 2. asteen polynomin tapauksessa tutkimalla epäyhtälön

$$a_1N^2 + a_2N + a_3 \leq c\ N^2$$

toteutumista suurilla N:n arvoilla. Kun valitaan $c > a_1$, löytyy välttämättä arvo N_0 siten, että epäyhtälö on tosi, kun $N \geq N_0$, jolloin algoritmin kompleksisuusluokka on $O(N^2)$.

- **O(log N)** **Logaritminen kompleksisuusluokka** ilmaisee mm. puolitusten tai 1:n tuplauksien lukumäärän syöteaineistolle N. Esimerkiksi N=32, jolloin $\log_2 N = 5$ eli 5 puolitusta/tuplausta: 32 – 16 – 8 – 4 – 2 – 1 .
  Esimerkiksi, kun haetaan alkiota järjestyksessä olevasta aineistosta, on algoritmi logaritmista kompleksisuusluokkaa (puolitushaku).

- **O(N)** **Lineaarinen kompleksisuusluokka** syntyy esimerkiksi taulukon tai listan alkioiden peräkkäisessä läpikäynnissä. Ohjelmassa peräkkäinen läpikäynti toteutetaan tyypillisesti toistosilmukoilla (for, while, do … while)

- **O(NlogN)** Kompleksisuusluokka **O(NlogN)** syntyy, kun esimerkiksi kahdella **sisäkkäisellä silmukalla käydään aineisto N läpi siten, että ensimmäinen silmukkamuuttuja käy kaikki alkiot läpi ja toinen käy alkiot läpi puolittamalla tai tuplaamalla**. Tämä on tärkeä kompleksisuusluokka, sillä voidaan osoittaa, että parhaimmat lajittelualgoritmit ovat kompleksisuusluokkaa O(NlogN).

- **O(N^2)** **Neliöllinen kompleksisuusluokka** syntyy tyypillisesti kahdella sisäkkäisellä toistosilmukalla, jossa kummassakaan silmukassa ei tapahdu puolituksia tai tuplauksia.

  Esimerkiksi edellinen vaihtolajittelualgoritmi on neliöllistä kompleksisuusluokkaa O(N^2). Kun taulukon koko on N, suorittaa 1. silmukkamuuttuja vähintään N ohjelma-askelta (silmukkamuuttujan kasvatus) ja jälkimmäinen kullakin kierroksella vähintään N-1, N-2 jne. ohjelma-askelta (silmukkamuuttujan kasvatus + mahdollinen vaihto). Täten ohjelma-askeleiden lukumääräksi saadaan aritmeettisen sarjan kaavan perusteella vähintään

  $$\mathrm{N} + (\mathrm{N}\text{-}1) + (\mathrm{N}\text{-}2) + \ldots + 2 + 1 = \frac{1+N}{2} \cdot N = \frac{1}{2}N^2 + \frac{1}{2}N = O(N^2).$$

- **O(a^N)** **Eksponentiaalinen kompleksisuusluokka** voi syntyä helposti rekursiivisella algoritmilla esimerkiksi, kun rekursiot tuottavat saman kutsun toistoja puumaisesti haarautuvassa kutsurakenteessa.

  Algoritmin, joka noudattaa eksponentiaalista kompleksisuusluokkaa, ajoajat noudattavat periaatetta: kun syötejoukon suuruus kasvaa aina vakiomäärän, peräkkäisten ajoaikojen suhde pysyy vakiona.

  Tämä on seurausta eksponenttifunktion matemaattisesta ominaisuudesta

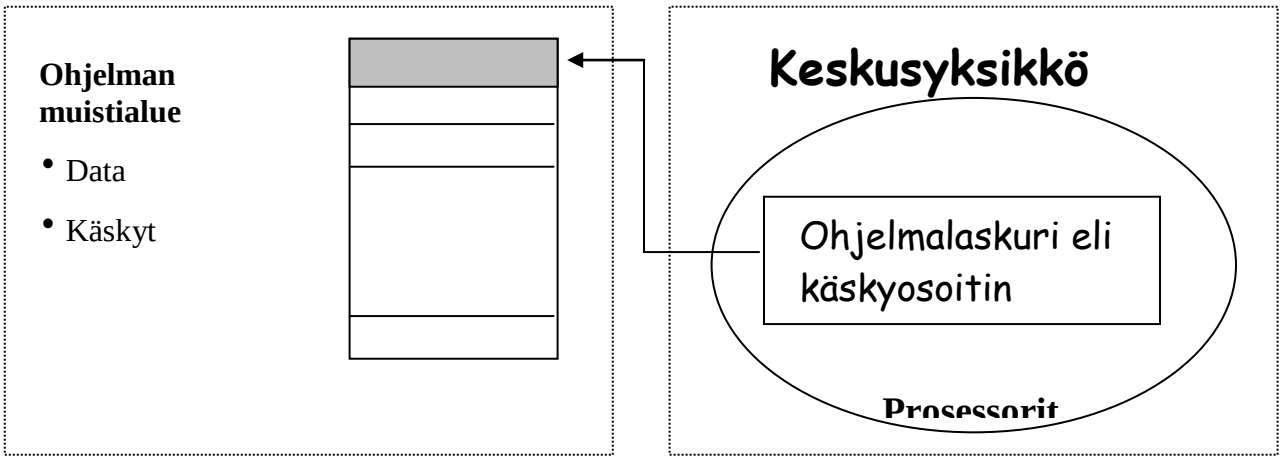
  $$\frac{a^{N+k}}{a^N} = \frac{a^N a^k}{a^N} = a^k = \text{vakio},$$

  kun k on syötejoukon vakiokasvu N, $N+k$, $N+2k$, … jne.

1.3 Iteraatio ja rekursio toistorakenteena

Nykyisen tietokonearkkitehtuurin mukaan ohjelman sisältämät käskyt ja data sijaitsevat eri muistialueella (levymuisti) kuin keskusyksikön prosessorien muistialue, jossa varsinainen suoritus tapahtuu. Tietojen välittäminen muistialueen ja keskusyksikön prosessorien välillä muodostaa nykyisen tietokoneen pullonkaulan suorituksen tehokkuutta ajatellen. Tehokkainta on suorittaa muistialueen peräkkäisissä paikoissa olevia käskyjä. Jos joudutaan varaamaan ohjelman muuttujille uusia muistipaikkoja, se hidastaa suoritusta jonkin verran. Kuitenkaan algoritmin kompleksisuusluokka ei muistipaikkojen varaamisesta muutu, vaan sen määrää algoritmin perusrakenne.

Nykyisen tietokoneen arkkitehtuuri:



Tietokoneohjelman luonnollinen suoritusperiaate on seuraava:

- Alusta ohjelmalaskuri

  Toistetaan

  - Hae ohjelmalaskurin osoittama käsky
  - Kasvata ohjelmalaskurin arvoa yhdellä
  - dekoodaa käsky
  - suorita käsky

Rekursio on algoritmi, joka kutsuu itseään joko suoraan tai välillisesti.

Tietokoneen rekursiivinen aliohjelma (metodi) on funktio, jonka rungossa (määrittelyosassa) esiintyy yksi tai useampi ”itsensä kutsu”.

Jokaisen aliohjelman kutsun yhteydessä luodaan ohjelman muistialueelle **aktivaatiotietue**, joka sisältää tilat mm. aliohjelman paikallisille muuttujille sekä mahdolliselle paluuarvolle. Peräkkäiset rekursiofunktion kutsut ovat toisistaan riippumattomia, joten jokaiselle kutsulle luodaan oma aktivaatiotietue. Tämä aiheuttaa sen, että **rekursioon liittyy aina tilakompleksisuus** (`STACK OVERFLOW`) **suurilla syöteaineistoilla**, jotka aiheuttavat paljon rekursiokutsuja.

Rekursio on teoreettisesti tarpeeton, sillä se voidaan aina (jos osataan) korvata ns. **iteratiivisella algoritmilla**. Ohjelmointikielissä iteratiivinen toisto toteutetaan **toistolauseilla** (**`for`**, **`while`** , **`do`** … **`while`**), joissa toistojen lukumäärää hallitaan silmukkalaskurilla. Iteratiivisen toistolauseen suoritus on tehokasta, koska keskusyksikön ei tarvitse luoda uusia muistipaikkoja (tai aktivaatiotietueita) ohjelman käyttämille muuttujille ja koska toistolauseen sisältämät ohjelmakäskyt sijaitsevat peräkkäisissä muistipaikoissa.

Tarkastellaan seuraavassa esimerkissä sitä miten rekursiivinen ja iteratiivinen algoritmi voivat pahimmillaan erota kompleksisuutensa puolesta toisistaan. Esimerkki kuvaa erästä erittäin vanhaa probleemaa.

Leonardo Pisalainen alias **Fibonacci**, pohti 1200-luvulla seuraavaa ongelmaa: Montako kaniparia syntyy vuodessa, jos alussa on 1 pari, ja jos jokainen kanipari tuottaa kuukaudessa uuden parin, joka tulee sukukypsäksi kahdessa kuukaudessa?

Kaniparien lukumäärää peräkkäisinä kuukausina kuvaa lukujono

1, 1, 2, 3, 5, 8, 13, …

Fibonaccin jonon luvut voidaan laskea helposti seuraavasta algoritmista:

Jos Fibonaccin luvun järjestysnumero N on 1 tai 2, palautetaan 1, muutoin lasketaan yhteen kaksi edellistä Fibonaccin lukua järjestysnumeroilla N-1 ja N-2.

Fibonaccin lukujen laskeminen on esimerkki ohjelmointiongelmasta, johon rekursiivinen algoritmi tuottaa ratkaisun helposti ja luonnollisesti. Alla laskenta-algoritmi on esitetty rekursiivisena pseudokoodina.

```
recFib(N)
      if N == 1 or  N ==2
      then return 1
      else return  recFib(N-1) + recFib(N-2)
```

Iteratiiviseen toistolauseeseen perustuva Fibonaccin lukujen laskenta-algoritmi on hieman vaikeampi muodostaa kuin rekursiivinen. Toistolauseessa tarvitaan laskurimuuttuja (i) sekä (helpossa ratkaisussa) kolme muistipaikkaa, merk. L1, L2 ja L3, joihin tallennetaan kullakin kierroksella peräkkäiset Fibonaccin luvut järjestysnumeroilla N, N-1 ja N-2. Alla on iteratiivisen algoritmin pseudokoodi.

```
iterFib(N)
      if N == 1 or N == 2
      return 1
      else
            i←3
            L1=L2=L3←1
            While i <= N
                  L1←L2
                  L2←L3
                  L3←L2+L1
                  i←i+1
      return  L3
```

Kun ohjelmoit edelliset algoritmit C-kielisiksi (tai joku muu kieli) funktioiksi, ja kokeilet niiden toimintaa käytännössä, huomaat, että rekursiivisen algoritmin käyttö on mahdotonta jo aika pienillä syötteen N (> 30) arvoilla. Sen sijaan iteratiiviseen toistoon perustuvalla algoritmilla voit huoleti antaa suuriakin syötearvoja, algoritmi suoriutuu salaman nopeasti. Kuitenkin ongelmaksi muodostuu hyvinkin pian se, että tavanomaiset kokonaislukujen suuruusrajoitukset tuottavat laskennassa ylivuodon ja lopputulos ei ole oikea.

Rekursiivinen Fibonacci-algoritmi on eksponentiaalista kompleksisuusluokkaa, mikä todetaan harjoituksissa mittaamalla ajoaikoja. On kuitenkin huomioitavaa, että rekursio ei aina merkitse pahaa aikakompleksisuutta, vaan se voi tuottaa joskus hyvinkin tehokkaan ratkaisun kuten esimerkiksi hajota ja hallitse periaate pikalajittelu algoritmissa (seur. kappale).

Tietorakenteet ja algoritmit

Kari Peisa

Ramk/Tekniikka ja liikenne 2009

Kari.Peisa@ramk.fi

2 Tietorakenteet

Tässä kappaleessa esitellään:

C-kielen rakenteelliset tietotyypit

- **Taulukko**
- **Tietue**

Taulukoiden yhteydessä esitellään kaksi lajittelualgoritmia, **lisäyslajittelu** ja **pikalajittelu**.

Abstraktit tietotyypit

- **Lista**
- **Pino ja jono**
- **Puut**

Abstrakteja tietotyyppejä toteutetaan **staattisina** ja **dynaamisina** C-rakenteina, joille määritellään käsittelyfunktiot mm. lisäys ja poisto. Pinon, jonon ja linkitetyn listan sovelluksena käsitellään hieman **verkkoja**.

Puut ja niiden sovellukset käsitellään pääasiassa vain abstrakteina rakenteina. Kuitenkin **binäärinen etsintäpuu** toteutetaan linkitettynä C-rakenteena.

2.1 Taulukot

Tässä kappaleessa tarkastellaan (C-kielen) rakenteelliset tietotyypit **taulukko** (`[]`)ja **tietue** (`struct`) ja esitellään joitakin hyödyllisiä haku- ja lajittelualgoritmeja, joita voidaan käyttää taulukoiden yhteydessä.

2.1.1 C-kielen taulukko

Esittely:

```
tyyppi nimi[koko]
tyyppi nimi[rivit][sarakkeet]
```

Esimerkiksi:

```
/*10 alkioinen merkkitaulukko */
char  merkit[10];
```

```
/*2-ulotteinen kokonaislukutaulukko, jossa 10 rivillä kullakin 5-alkion kokoinen 1-ulotteinen tauluko*/

int lukuMatriisi[10][5];
```

Taulukossa voi olla vain yhden tyyppisiä tietoalkioita, jotka voivat olla mitä tahansa tietotyyppiä. 1-ulotteinen taulukko on peräkkäisistä muistipaikoista koostuva indeksoitu lista, jonka alkioihin viitataan indeksillä, 2-ulotteisen taulukon alkioihin viitataan indeksiparilla, 3-ulotteiseen taulukon alkioihin indeksikolmikolla jne.

Taulukoiden alustaminen:

Taulukot voidaan alustaa koodissa.

Esimerkiksi kokonaislukutaulukot:

```
int luvut1[4]= {23,6,7,12};

int luvut2[4]={2};

int luvut3[2][3]={{1,2,3},{4,5,6}};
```

Taulukko `luvut1` on samaa muotoa kuin koodissa annettu lista. Taulukko `luvut2` sisältää listan {2, 0, 0, 0} ja kolmas voidaan esittää 2-ulotteisena matriisina

$$\begin{Bmatrix} \{1,2,3\} \\ \{4,5,6\} \end{Bmatrix}$$

Merkkitaulukot:

```
#define MAX_PITUS 32

char nimi[MAX_PITUUS+1] = ”Kalle”;
```

1. rivillä määritellään esikääntäjälle maksimipituus 32 merkkiä käsiteltäville merkkijonoille. Merkkijonotaulukko on samalla **C-kielen merkkijonotyppi**. Merkkijonon lopussa on oltava lopetusmerkki `'\0'`, jonka avulla voidaan tietää, mikä osa taulukosta kuuluu merkkijonoon. Lopetusmerkille varataan yleensä 1 merkki tilaa varsinaisen maksimikoon lisäksi. Kun merkkijono alustetaan koodissa, laittaa kääntäjä lopetusmerkin automaattisesti.

Arvon sijoitus ja saanti taulukossa:

```
luvut[0] = 15; /*Sijoitetaan arvo 15 ja liitetään se taulukon indeksiin 0 */

luvut[MAX_PITUUS] = '\0'; /*Sijoitetaan lopetusmerkki taulukon viimeiseen lokerroon */

int temp = luvut[0]; /*Taulukon indeksin 0 osoittamassa kohdassa oleva arvo sijoitetaan muuttujan temp arvoksi*/
```

2.1.2 Haku taulukoista

Peräkkäishaku

Jos taulukko ei ole järjestetty, joudutaan hakemisessa soveltamaan peräkkäishakua eli kaikki taulukon alkiot käydään läpi toistosilmukassa. Tällöin algoritmin on lineaarista kompleksisuusluokkaa **O(N)**.

Puolitushaku

Soveltuu vain **järjestyksessä oleviin taulukoihin**. Periaate käydään läpi tässä esimerkillä. Harjoituksissa perehdytään algoritmin ohjelmoimiseen.

Esimerkki: Taulukossa **`taulu`** on luvut **`{3, 5, 8, 8, 9, 12, 13, 15, 24, 29}`**

Haetaan lukua **`15`**.

Verrataan haettavaa lukua taulukon keskimmäisen indeksin osoittamaan arvoon.

Keskimmäinen indeksi = (0 + 9)/2 = 4 (Huom! C-kielen laskuperiaate kokonaisluvuilla)

```
taulu[4] < 15
```

Puolitetaan aineisto ja katsotaan seuraavaksi indeksien 5…9 sisältämiä arvoja samalla periaatteella. Keskimmäinen indeksi =(5+ 9)/2 = 7

```
taulu[7] == 15
```

Hakua ja sitä seuraavaa puolitusta toistetaan kunnes keskimmäinen alkio on yhtä suuri kuin haettava arvo tai kunnes on jäljellä vain yksi alkio, joka joko on tai ei ole yhtä suuri. Koska tässä tehdään peräkkäisiä puolituksia, on algoritmin kompleksisuus luokkaa **O(log N)** eli on erittäin hyvä. Jos esimerkiksi taulukossa on 10 000 järjestyksessä olevaa alkiota, tarvitaan tietyn alkion hakemiseen (tai tietoon alkion puuttumisesta) vain 14 hakua. Alla on puolitushaun pseudokoodi.

Pseudokoodi:

```
etsi(haettava,lista,koko)
      alku←1, loppu←koko
      While alku <= loppu
            keski←(alku+loppu)/2
            if lista[keski]==haettava
            then return true
            else  if lista[keski]> haettava
                  then alku←keski+1
                  else  loppu←keski-1
      return false
```

2.1.3 Lajittelualgoritmit taulukossa

Seuraavassa esiteltävät algoritmit ovat parempia kuin edellä esitetty vaihtolajittelualgoritmi useimmissa tapauksissa. Taulukon alkuperäinen järjestäytyneisyyden tila vaikuttaa kuitenkin voimakkaasti algoritmin tehokkuuteen.

Lisäyslajittelu (Insertion Sort)

Periaate: Lajitellaan vasemmalta (oikealta) lähtien siten, että alkiot [0, …, i], missä i läpikäy kaikki arvot `1, 2, … , koko-1`, pysyvät järjestettynä lajittelun edetessä.

Esimerkki:

{ 4, 6, 2, 8, 3, 1}

4 ← lisätään 6

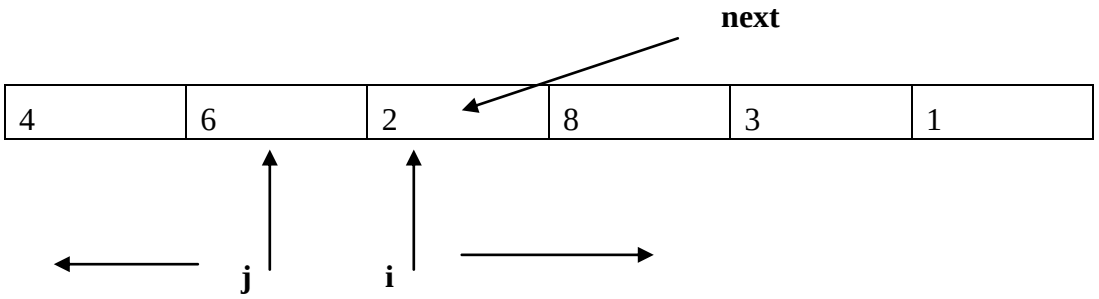
4, 6 ← lisätään 2

2, 4, 6 ← lisätään 8

2, 4, 6, 8 ← lisätään 3

2, 3, 4, 6, 8 ← lisätään 1

1, 2, 3, 4, 6, 8



Lisäyslajittelun C-kielinen koodi

```
void insertionSort(int list[], int size)
{
      int i,j,next;

      for(i=1;i<size;i++)
      {
            next=list[i];
            for(j=i-1;j>=0 && next < list[j];j--)
                  list[j+1]=list[j];
            list[j+1]=next;
      }

   }
```

Koodin analysointi: 1. silmukassa indeksi `i` läpikäy kaikki taulukon alkiot ensimmäistä lukuun ottamatta. Jälkimmäisessä silmukassa indeksi `j` käy läpi kaikki indeksin `i`

vasemmalla puolella olevat alkiot, jotka ovat suurempia kuin vertailtava alkio. Apumuuttujalla `next` on aina indeksissä `i` oleva solun arvo. Jos se on pienempi kuin indeksin `j` osoittama arvo, siirretään jälkimmäistä yhdellä oikealle. Näin jatketaan niin kauan kuin tullaan listan alkuun tai kohdataan `next`-arvoa pienempi tai yhtä suuri arvo, jolloin lopuksi sijoitetaan `next` -arvo viimeiseksi siirretyn alkion eteen.

Lisäyslajittelun kompleksisuudesta

- paras tilanne: lista valmiiksi järjestyksessä ”oikein päin”, jolloin vaihtoja ei tehdä → lineaarinen **O(N)** kompleksisuusluokka.
- pahin tilanne: lista väärin päin järjestyksessä, jolloin jokaisen alkion paikka vaihdettava → **$O(N^2)$**
- keskimääräinen kompleksisuusluokka on ikävä kyllä myös **$O(N^2)$**

Lisäyslajittelu sopii yleisesti pienille aineistoille tai melkein järjestyksessä olevan aineiston lajitteluun.

Pikalajittelu (**Quicksort**)

Idea: **”hajoita ja hallitse”**, perustuu osiin jakamiseen ja osien rekursiiviseen käsittelyyn samalla algoritmilla. Pikalajittelualgoritmissa valitaan ensin ns. **hajottaja-alkio** (**pivot**), joksi voidaan valita mikä tahansa lajiteltavista alkioista. Sen jälkeen lajiteltavat alkiot **jaetaan kahteen erilliseen osaan** siten, että hajottaja-alkion vasemmalle puolelle tulevat kaikki hajottajaa pienemmät alkiot ja oikealle puolelle kaikki sitä suuremmat. Tämän jälkeen edellistä **algoritmin osaa toistetaan molemmille puolille erikseen** kunnes puoliskon koko on <=1 alkio, jolloin lopetetaan.

Esimerkki: (Valitaan tässä hajottajaksi aina listan vasemmanpuolisin alkio)

{ **4**, 6, 2, 8, 3, 1, 5} →	{{ 2, 3, 1},**4**, {6, 8, 5}}
{ **2**, 3, 1} → {{1}, **2**,{3}}	{{{1},**2**,{3}}**4**, {6, 8, 5}}
{**6**, 8, 5} → { {5}, **6**, {8}}	{{{1},**2**,{3}}**4**, {{5}, **6**,{ 8}}}

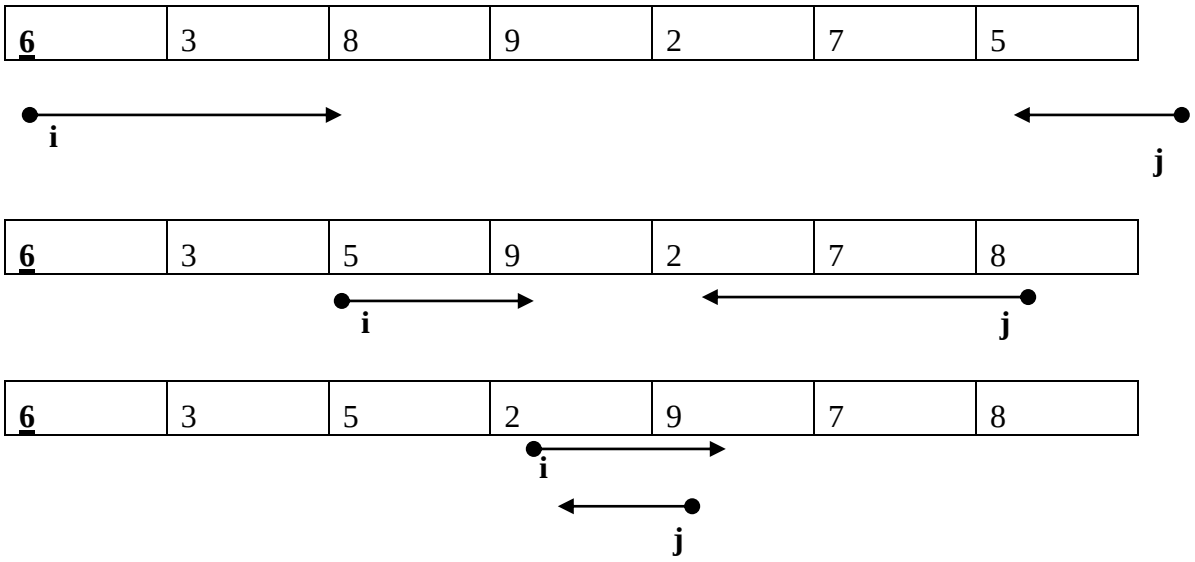
Pikalajittelualgoritmin C-kielinen toteutus (kokonaislukutaulukoille)

Funktiota kutsutaan muodossa `quickSort( list, left, right)`, missä `list` on se taulukon osa, joka on rajattu indeksein `left` ja `right`. Makro `SWAP` vaihtaa kahden parametrin arvot keskenään.

```
#define SWAP(x,y,t) ((t)=(x),(x)=(y),(y)=(t))

void quickSort(int list[],int left, int right)
{
      /* Tässä pivot valitaan aina listan alusta */
      int i,j,pivot,temp;
      if (left < right)
      {
            i=left;
            j=right+1;
            pivot=list[left];
            do {
                  do
                        i++;
                  while (list[i]</*kytkin*/pivot && i < right);
                  do
                        j--;
                  while (list[j]> /*kytkin*/pivot && j > left);
                  if(i<j) SWAP(list[i],list[j],temp);
            }while (i<j);
            SWAP(list[left],list[j],temp);
            quickSort(list,left,j-1);
            quickSort(list,j+1,right);
      }
}
```

Koodin analysointia esimerkin avulla:

Valitaan hajottajaksi vasemmanpuoleisin arvo. Indeksi `i` juoksee käsiteltävän taulukon vasemmasta alkiosta lähtien ja indeksi `j` juoksee vastaavasti oikeasta reunasta käsin. Kun `list[i] > list[j]`, vaihdetaan niiden paikkaa.



Kun `i` ja `j` ovat ohittaneet toisensa, vaihtoa ei tehdä, vaan siirrytään uloimmasta `do-while`- silmukasta ulos ja suoritetaan `pivot`-alkion ja `j`-alkion vaihto (**6** ←→ 2).

2	3	5	**6**	9	7	8

Nyt kutsutaan rekursiivisesti taulukkoja

2	3	5

ja

9	7	8

Pikalajittelun kompleksisuudesta:

Jos hajottaja-alkioksi valitaan listan ensimmäinen (kuten yllä) ja lista on valmiiksi järjestetty väärässä lajittelujärjestyksessä, on pikalajittelun kompleksisuus luokkaa **O(N^2)**. Tämä johtuu siitä, että hajottaja-alkioksi tulee valituksi listan pienin (tai suurin) luku ja kussakin rekursiivisessa kutsussa oleva taulukon osa (N_k alkiota) jaetaan kahteen osaan, joista toinen on tyhjä ja toinen sisältää loput N_k-1 alkiota. Tämä voi olla ongelma, sillä usein taulukot saattavat olla melkein järjestettyjä.

Jos taulukko on täysin epäjärjestyksessä, ei hajottaja-alkion valinnalla ole vaikutusta, sillä silloin se on pieni, keskisuuri tai suuri sattumanvaraisesti. Voidaan osoittaa, että kun hajottaja-alkio jakaa aineiston johonkin vakiosuhteeseen {k} – hajottaja – {N-k-1} millä tahansa arvolla k>0, noudattavat rekursiiviset kutsut aina logaritmista kompleksisuutta ja pikalajittelun kompleksisuus on tällöin luokkaa **O(N log N)**.

Jotta vältetään edellä tullut kompleksisuus, valitaan pikalajittelussa usein hajottaja-alkioksi keskimmäinen tai vielä mieluummin satunnaisesti väliltä `left`…`right`.

Rekursiivinen algoritmi ei tässä aiheuta samanlaista eksponentiaalista kompleksisuutta kuten Fibonaccin luvut -esimerkissä, koska peräkkäiset rekursiokutsut käsittelevät kukin taulukon eri osia.

Matemaattisesti voidaan osoittaa, että keskimäärin **pikalajittelun kompleksisuus on luokkaa O(N log N)**, joka on hyvä. (kts. Cormen & co.: Introduction to Algorithms 2nd Ed., ss. 154 – 158). Edelleen voidaan osoittaa, että tämä kompleksisuusluokka on paras mahdollinen (vertailuperiaatteella toimivalle) lajittelualgoritmille (kts. Cormen & co.: Introduction to Algorithms 2nd Ed., ss. 165 – 167).

2.1.4 Rakenteelliset tietotyypit taulukossa

C-kielen osoittimista ja tietueista

Tarkastellaan aluksi osoittimien käyttöesimerkkiä taulukoiden yhteydessä.

```
int *list_ptr;
int taulu[4]={1,2,3,4};
list_ptr=taulu+1;
*list_ptr=taulu[2];
```

Koodin alussa esitellään kokonaislukutyyppinen osoitinmuuttuja **`list_ptr`**, sekä alustetaan 4 kokonaislukuarvoa taulukkomuuttujaan **`taulu`**. Taulukon nimi `taulu` viittaa C-kielessä taulukon 1. alkioon (**`taulu[0]`**). Kun osoittimen osoitearvoon lisätään 1, merkitsee se sitä, että osoitin siirtyy osoittamaan seuraavaa muistilohkoa eli taulukossa seuraavaa alkioita. Kolmannella rivillä alustetaan siten osoitinmuuttuja **`list_ptr`** osoittamaan taulukon alkiota **`taulu[1]`** (eli arvoa 2). Sijoituslauseessa **`*list_ptr=taulu[2]`** sijoitetaan osoittimen **`list_ptr`** osoittamaan arvoon (eli **`taulu[1]`**) taulukon indeksissä 2 oleva arvo (eli arvo 3). Tällöin lopulta taulukko tulostuisi muodossa:

{1,3,3,4}

C-kielen tietueet (`struct`)

Taulukot voivat sisältää vain samaa tyyppiä olevia tietoalkioita. Tietue (**`struct`**) on rakenteellinen tietotyyppi, joka voi sisältää vaihtelevan tyyppisiä tietoalkioita sekä myös muita tietueita.

Tietueen syntaksi (esimerkki)

```
struct {
      int ika;
      char nimi[32+1];
      …
      float palkka;
      } henkilo;
```

Usein määritellään tietue tietuetyyppinä, jota voidaan sujuvasti käyttää muuttujien esittelyssä.

```
typedef struct {
      int ika;
      char nimi[32+1];
      …
      float palkka;
      } henkilo;
```

Tämän jälkeen voidaan **`henkilo`** -tietuetyypin muuttujat esitellä seuraavasti:

```
henkilo hlo1, hlo2;
```

Tietuemuuttujan kenttiin viitataan **pistenotaatiolla**

```
hlo1.ika=32;
strcpy(hlo1.nimi,"Kalle");//Huom! "incluudaa" <string.h>
hlo1.palkka=2300.00;
```

Tietueen kopiointi tehdään sijoituslauseella, jolloin kaikki kentät kopioituvat.

hlo2=hlo1;

Kahden tietueen samuutta ei voi verrata loogisten operaattorien (**==** ja **!=**) avulla kuten **hlo1==hlo2**, vaan se on tehtävä kenttäkohtaisesti.

Tietuetyyppinen osoitin

```
henkilo* hlo_p = &hlo1;//osoitin hlo_p viittaa tietueeseen hlo1
```

Tietueosoittimen avulla tietueen arvokenttiin viitataan **nuolinotaatiolla**

```
printf("Henkilö %s saa palkkaa %f",hlo_p->nimi, hlo_p->palkka);
```

Tietueosoittimet ovat tärkeitä apuvälineitä linkitettyjen tietorakenteiden käsittelyssä, mihin palataan myöhemmin.

Lajittelu taulukon rakenteellisilla alkioilla

Kun taulukkoon on sijoitettu tietuetyyppisiä tietoalkioita, on edellä esitettyjä algoritmeja hieman muutettava lajiteltaessa tietueet jonkin tietokentän arvojen perusteella.

Esimerkki:

```
/*varataan taulukko 100 henkilötietueelle*/
henkilo henkilosto[100];

/* lisäyslajittelualgoritmi henkilötietuetaulukon
lajittelemiseksi kentän ika mukaisesti  */

void insertionSort(henkilo list[], int size)
{
      int i,j;
      henkilo next;
      for(i=1;i<size;i++)
      {
            next=list[i];
            for(j=i-1;j>=0 && next.ika < list[j].ika;j--)
                  list[j+1]=list[j];
            list[j+1]=next;
      }
}
```

Vertaa yo. koodia edelliseen. Muutokset koodissa eivät ole kovin suuria. Suurilla taulukoilla, joissa olevat tietuerakenteet ovat suuria, lajittelualgoritmit ovat suhteellisen raskaita, koska niissä tietueet kopioidaan fyysisesti jokaisen vaihdon (tietueen sijoituksen) yhteydessä.

Tietorakenteet ja algoritmit

Kari Peisa

Ramk/Tekniikka ja liikenne 2006 (Päivitetty 2008)

Kari.peisa@ramk.fi

2.2 Abstraktit tietorakenteet

Abstraktilla tietorakenteella tarkoitetaan tietorakenteen yleistä mallia, jolle ei ole määritelty yksityiskohtaista toteutusta, vaan pelkästään

- tietorakenteen toimintaan liittyvät säännöt
- datan käsittelytoiminnot

Abstraktit tietorakenteet voidaan toteuttaa monella tavalla ja monilla eri ohjelmointikielillä. Tässä kurssissa toteutetaan abstrakteista tietorakenteista **pino**, **jono**, **lista** ja **puu** ja toteutuksissa käytetään niin **staattisia** kuin myös **dynaamisia** rakenteita.

Staattinen rakenne tarkoittaa sitä, että tietorakenteen fyysinen tila (varattu muistitila) on määritelty ennen ajoa ohjelman käännöksen yhteydessä. Tällainen syntyy esimerkiksi käyttämällä **C-kielen taulukkoa**.

Dynaamisessa tietorakenteessa fyysinen tila muuttuu dynaamisesti ohjelman ajon yhteydessä käyttäjän antamien syötteiden mukaisesti. Dynaamiset tietorakenteet toteutetaan **C-kielen tietueiden ja osoittimien** avulla.

2.2.1 Lista, pino ja jono abstrakteina tietorakenteina

Lista

Säännöt. Lista on tietorakenne, joka varastoi data-alkioita lineaariseen järjestykseen.

Operaatioita:

- Lisää (insert, add)
- Poista (delete)
- Hae (search)
- Pituus? (length)

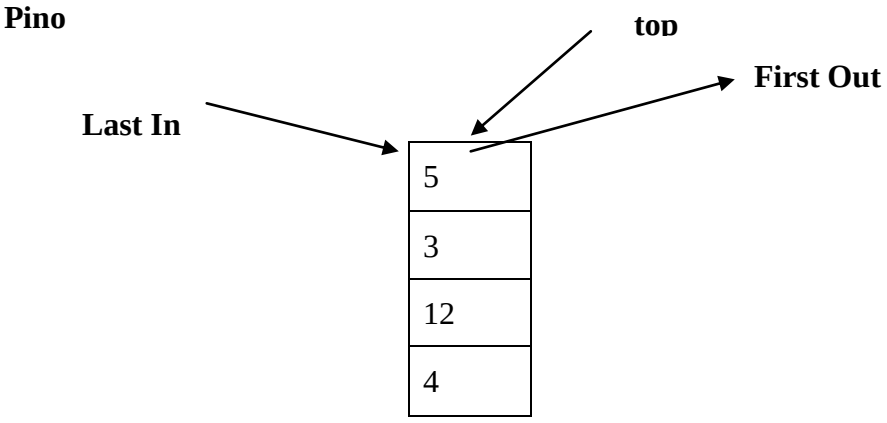
Lista voidaan toteuttaa järjestettynä jolloin operaatioina voisi olla

- Lajittele (sort)
- Käännä (invert)

Listoja voidaan myös yhdistää, jolloin operaationa olisi

- Yhdistä (concanate)

Lista on perusrakenne ja usein tietorakenteiden yhteydessä käytetäänkin jotain listan erikoistapausta kuten esimerkiksi **pino** ja **jono**, jotka esitellään seuraavassa.



Säännöt: Pino on tallennusrakenne, joka varastoi dataa järjestettyyn listaan toimintaperiaatteella: **LIFO** (**Last In First Out**].

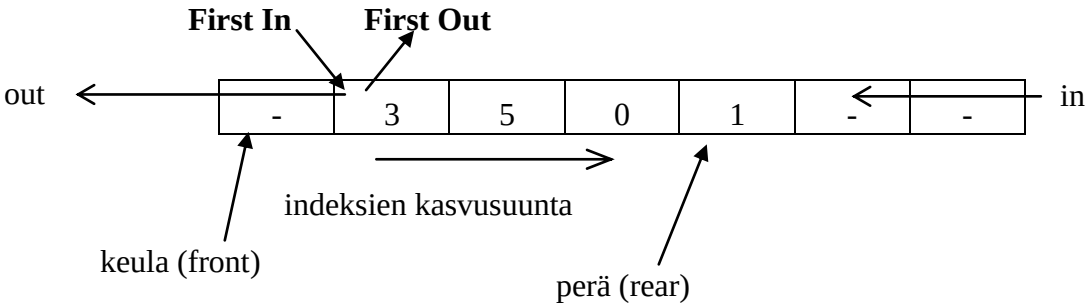
Operaatioita:

- Syöttö (push)
- Poisto (pop)
- Tyhjennä (makeEmpty)
- Onko tyhjä? (isEmpty)
- Onko täysi? (isFull) (jos staattinen rakenne)

Pinon sovelluksista

Pino on tallennusrakenne, jonka avulla voidaan toteuttaa **erittäin nopeat lisäys- ja poisto-operaatiot**. Esimerkiksi käyttöjärjestelmien ja kääntäjien toiminnan nopeus perustuu erittäin suuressa määrin pinon käyttöön. Pinon avulla voidaan helposti mm. kääntää alkioiden järjestys. Pinot ovat monien sellaisten algoritmien perusrakenne, joissa joudutaan toteuttamaan erilaisia ”paluita”. Tällaisia algoritmeja tarvitaan esimerkiksi verkkojen läpikäynneissä (kpl. 2.4.5) sekä erilaisissa älykkäissä päättelyongelmissa.

Jono



Säännöt: Jono on tallennusrakenne, joka varastoi dataa järjestettyyn listaan toimintaperiaatteella: **FIFO** (**First In First Out**].

Operaatiot:

- Syöttö (enqueue)
- Poisto (dequeue)
- Tyhjennä (makeEmpty)
- Onko tyhjä? (isEmpty)
- Onko täysi? (isFull) (jos staattinen rakenne)

Jonon sovelluksista

Jono kuten pinokin on tallennusrakenne, jonka avulla **lisäys- ja poisto-operaatiot** voidaan toteuttaa erittäin tehokkaasti. Jonolla voidaan ylläpitää suoritusjärjestyksiä. Esimerkiksi käyttöjärjestelmät käyttävät jonoa kontrolloidessaan rinnakkaisten tehtävien suoritusjärjestystä. Monet liikenteen ohjausjärjestelmät perustuvat jonojen käyttöön. Jonoja voidaan käyttää myös verkkojen läpikäyntialgoritmeissa (kpl. 2.4.5).

2.3 Staattiset toteutukset taulukossa

Pino ja jono voidaan toteuttaa taulukossa staattisena rakenteena. Jos tallennettavat tietoalkiot ovat rakenteeltaan monimutkaisia ja niiden kopiointi alkaa olla raskas operaatio, kannattaa pino ja jono toteuttaa dynaamisina linkitettyinä rakenteina, jotka esitellään seuraavassa kappaleessa.

2.3.1 Pino taulukossa (kokonaislukupino)

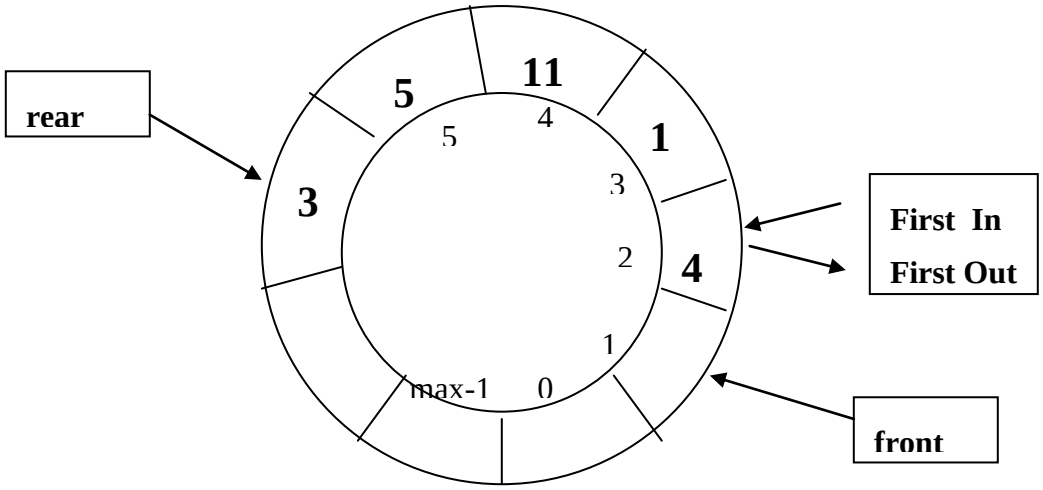
```
#define MAX_STACK_SIZE 50

/* Pinon luonti*/
int stack[MAX_STACK_SIZE];/*pinotaulukko*/
int top=-1;

/* Operaatioiden määrittely */

int isEmpty(){
      if (top<0) return 1;
      else return 0;
}
int isFull(){
      if (top==MAX_STACK_SIZE-1) return 1;
      else return 0;
      }
void push(int nmbr){
      if(isFull()) stackFull();/*Handle overflow*/
      else stack[++top]=nmb;
}
int pop(){
      if(isEmpty()) stackEmpty();/*Handle underflow*/
      else return stack[top--];
}
```

Funktio `stackFull()` käsittelee tilanteen, jossa pinoon yritetään lisätä, vaikka pino on täynnä. Vastaavasti funktio `stackEmpty()` käsittelee tilanteen, jossa pinosta yritetään ottaa, vaikka pino on tyhjä. Näiden funktioiden määritys riippuu siitä, miten syntynyt poikkeustilanne halutaan hoitaa. Yllä oleva pinon määrittely kannattaa sijoittaa erilliseen otsikkotiedostoon (header file), jolloin pinon rajapintafunktiot on helposti nähtävissä.

2.3.2 Rengasjono taulukossa

Tavallinen peräkkäisrakenteinen taulukko ei sovellu hyvin jonon toteutukseen, sillä lisättäessä alkioita jonoon sen perä siirtyy (tehokkuussyistä) kokoajan vain yhteen suuntaan ja taulukon kokorajoitukset estäisivät pian jonon käytön. Taulukossa voidaan kuitenkin toteuttaa **rengasjono** `modulo` -jakolaskuoperaattorin (`%`) avulla. Taulukon rajojen ylitystä ei tapahdu, kun indeksi lasketaan jakojäännöksenä `i%MAX_QUEUE_SIZE`. Tällöin taulukon alkiot tulevat aina kierroksen täytyttyä uudelleen käytetyiksi.

Operaatioiden toteutuksessa käytetään 4 muuttujaa: `keula` (`front`), `perä` (`rear`), alkioiden lukumäärä `n` ja maksimilukumäärä `MAX_QUEUE_SIZE`.




```
/* Rengasjonon luonti*/

int queue[MAX_QUEUE_SIZE];
int rear=-1,front=-1,n=0;

/* Operaatioiden määrittely */

int isEmpty(){
      if(n==0) return 1;
      else return 0;
}
int isFull(){
      if(n==MAX_QUEUE_SIZE) return 1;
      else return 0;
}
void enqueue(int nmb){
      if(isFull()) queueFull();/*Handle overflow*/
      else {
            rear=(rear+1)%MAX_QUEUE_SIZE;
            n++;
            queue[rear]=nmb;
      }
}
int dequeue(){
      if(isEmpty()) queueEmpty();/*Handle underflow*/
      else {
            front=(front+1)%MAX_QUEUE_SIZE;
            n--;
            return queue[front];
      }

}
```

Funktiot `queueFull()` ja `queueEmpty()` määritellään vastaavasti kuin pinon yhteydessä. Jonon keula ja perä alustetaan alkuarvolla -1, jolloin koodissa sekä jonoon lisättäessä että poistettaessa molempien uusi indeksi voidaan laskea samalla periaatteella ennen varsinaista taulukkosijoitusta.

2.4 Dynaamiset tietorakenteet

Dynaamisten tietorakenteiden fyysinen tila määräytyy ohjelman suorituksen aikana. Tällöin tietorakenteiden vaatimaa muistitilaa ei varata staattisesti ohjelman käännöksen yhteydessä, vaan se joudutaan varaaman (allokoimaan) dynaamisesti ajon aikana. Ohjelman laatijan on huolehdittava muistinhallinnasta. Ohjelmointikielestä riippuen joudutaan huolehtimaan myös muistin vapauttamisesta (C/C++) tietorakenteiden tuhoamisen yhteydessä. Joissakin kielissä kuten esimerkiksi Javassa on sisäänrakennettu **automaattinen roskienkeruujärjestelmä**, joka huolehtii tuhottujen olioiden muistitilan vapauttamisesta uudelleen ohjelman käyttöön.

Dynaamisia tietorakenteita käsitellään **osoitinmuuttujien** avulla. Tämä mahdollistaa tehokkaiden lisäys- ja poisto-operaatioiden toteuttamisen, sillä tietokenttien kopiointia ei tarvitse tehdä, vaan linkitys tapahtuu muistiosoitteiden sijoituksilla

2.4.1 C ja C++ -kielten dynaaminen muistinallokointi

C/C++ -kielissä ohjelman laatijan on huolehdittava dynaamisesti niin varatun muistitilan varaamisesta kuin myös vapauttamisesta, kun ohjelmassa sitä ei enää tarvita. C-kielessä varaaminen ja tuhoaminen tehdään funktioiden **`malloc`** ja **`free`** avulla, C++ -kielessä operaattoreiden **`new`** ja **`delete`** avulla.

C-kielessä muisti varataan funktion **`malloc`** avulla.

```
void *malloc(size_t size)
```

Parametri **size_t** on tyyppiä **`unsigned integer`**. Funktio palauttaa **`void`** –tyyppisen osoittimen allokoituun muistitilaan tai **`NULL`** -osoittimen, mikäli muistia ei ole riittävästi. Paluuarvon tyyppi määritellään tyyppimuunnoksella halutun tietotyypin osoittimeksi. Muistialueen koko määritellään **`sizeof`** –funktion avulla.

Seuraavassa esimerkissä määritellään tietuetyyppi ELEMENT. Esimerkkikoodilla esitetään miten dynaaminen muistinallokointi ja –vapautus tehdään ELEMENT tyypin tietuemuuttujille. Tietuetyyppi ELEMENT määritellään globaalisti ja se kirjoitetaan isoilla kirjaimilla selkeyden vuoksi. Usein se on nimetty vain isolla alkukirjaimella, jolloin sitä on helpompi kirjoittaa koodissa.

```
/* Tietuetyypin ELEMENT määrittely*/

typedef struct {
  int key;
  /* other fields */
} ELEMENT;
```

```
/* esitellään ELEMENT -tyyppinen osoitinmuuttuja newElemPtr ja
asetetaan se osoittamaan dynaamisesti allokoituun muistitilaan.
Tietueosoittimen kenttiä käsitellään nuolioperaation avulla*/

#include <stlib.h> /*funktiot malloc ja free*/

ELEMENT *newElemPtr = (ELEMENT*)malloc(sizeof(ELEMENT));
newElemPtr->key=1;

/* muistin vapauttaminen ajon aikana, kun tietuetta ei enää
käytetä ohjelmassa*/

free(newElemPtr);
```

C++ -kielessä muistin dynaaminen allokointi on hieman yksinkertaisempi suoritus kuin C –kielessä. Edellisen esimerkin muistin allokointi toteutetaan C++ -tyylillä esimerkiksi seuraavasti

```
ELEMENT *newElem = new ELEMENT;
```

ja muistin vapauttaminen seuraavasti

```
delete newElem;
```

Tietuetyypin määrittely ja sen kenttien käsittely osoitinmuuttujan ja nuolioperaattorin avulla tapahtuu samalla tavalla kuin edellisessä esimerkissä. Tällä kurssilla käytetään yksinkertaisuuden vuoksi muistin allokointiin C++ -tyyliä. Dynaamiset tietorakenteet toteutetaan C-kielen mukaisesti tietuerakenteina eikä oliopohjaisesti.

Solmu

Linkitettyjen tietorakenteiden perusosa on **solmu**. C-kielessä solmu on tietue, joka sisältää vähintään kaksi tietokenttää:

- **datakenttä** tallennettavalle datalle, sekä
- **linkkikenttä**, johon tallennetaan listassa seuraavana olevan solmun osoite.

Alustettaessa solmun linkkikentälle annetaan arvo NULL (0).

Solmu toteutetaan seuraavalla tietorakenteella:

```
typedef struct NODE * NODEPTR;

typedef struct {
      int key;
      /* other data fields */
      NODEPTR link;

} NODE;
```

Edellä määritellään tietuetyyppi NODEPTR, joka on osoitintyyppi tietuetyypille NODE. Tämä määritellään vasta seuraavalla rivillä ja sillä on puolestaan yhtenä tietokenttänä NODEPTR tyypin osoitinmuuttuja. Tällainen itseensä viittaus (self-referential structure) on tehty C-kielessä mahdolliseksi, koska muutoin oltaisiin paradoksaalisessa tilanteessa:

emme voi määritellä osoitintyyppiä vielä määrittelemättömään tietuetyyppiin, mutta emme voi myöskään määritellä tietuetyyppiä, koska se sisältää tämän osoitintyypin.

Uuden solmun luonti ja alustus:

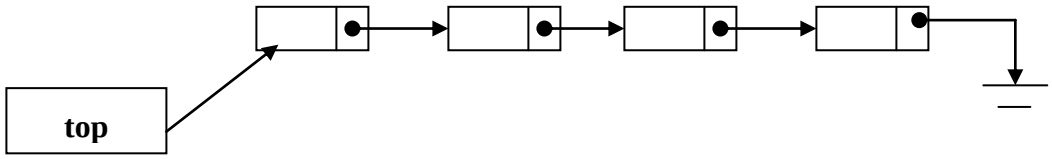
```
#define NULL 0 /* Tarvitaan C-kielen koodissa */
NODEPTR newNode = new(NODE);
newNode->key=/*...alustetaan datakentät*/
newNode->link=NULL;/*alustetaan linkkikenttä*/
```

Solmun linkkiä ei voi alustaa tyyppimäärityksen yhteydessä osoittamaan arvoa NULL, mikä olisi sinänsä ihan hyvä asia. Solmun varaama muisti vapautetaan seuraavasti:

```
delete(newNode);
```

Dynaamisia tietorakenteita suunniteltaessa on tarkkaan mietittävä, mille ohjelman osalle vastuu dynaamisen muistin varaamisesta ja vapauttamisesta jätetään. Seuraavissa esimerkeissä on noudatettu periaatetta, että **abstraktin tietotyypin käsittelyfunktioille välitetään vain datarakenteiden osoittimet, jolloin dynaamisen muistin varaamisen ja vapauttamisen vastuu jätetään käsittelyfunktioita kutsuvalle ohjelman osalle**. Näin abstraktien tietorakenteiden käsittelyfunktiot voidaan toteuttaa mahdollisimman yksinkertaisesti ja keskittyä olennaiseen eli itse tietorakenteen käsittelyyn. Lisäksi käsittelyoperaatioista tulee mahdollisimman nopeita, koska datakenttien kopiointia ei tapahdu.

2.4.2 Linkitetty pino



Linkitetty pino on linkitetty lista, jossa lisäykset ja poistot tapahtuvat aina listan samasta päästä. Linkitetyn pinon käsittely vaatii vain yhden **solmu -tyyppisen osoitinmuuttujan** `top`, jonka avulla ”pidetään kiinni” listan siitä päästä, johon uudet solmut lisätään.

```
/* Pinoon lisäys push-funktiolla. Parametreina ovat osoittimen
osoitin pinon huippuun ja osoitin lisättävään tietueeseen.*/

void push(NODEPTR *top, NODEPTR *newNode)
{
      (*newNode)->link = *top;
      *top = *newNode;
}
```

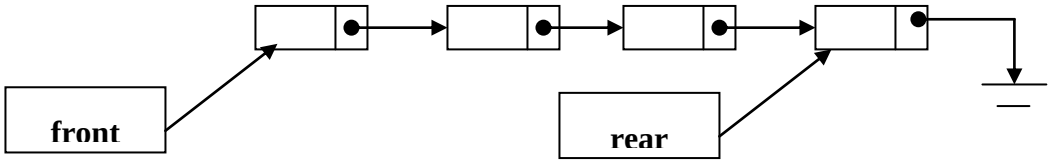
```
/* Pinosta poisto pop-funktiolla. Parametrina on osoittimen
osoitin pinon huippuun. Tässä palautetaan tietueosoitin pinon
huipulla olevaan tietueeseen. */

NODEPTR pop(NODEPTR *top)
{
NODEPTR ptr=*top;

      if(*top){
            *top=(*top)->link;
            ptr->link=NULL;
      }
      return ptr;
}
```

Edellä pinon käsittelyfunktioissa pinon huipun osoitinmuuttuja `top` ja lisättävän tietueen osoitin on annettava osoitinparametrina, koska muussa tapauksessa parametreja käsiteltäisiin paikallisina muuttujina (arvovälitys), ja pinon päivitysrutiinit eivät välittyisi kutsuvalle funktiolle, jossa pino on määritelty. Pinossa oleva tietueen varaamaa muistialuetta ei vapauteta, jolloin vastuu muistialueen vapauttamisesta siirtyy kutsuvalle ohjelman osalle. Jos pino on tyhjä, palautuu NULL osoitin.

Kutsuvassa funktiossa pinon käsittely tapahtuu seuraavasti:

```
/*Tyhjän pinon luonti*/
NODEPTR top=NULL;

/* uuden solmun luonti*/
NODEPTR ptr = new NODE;
ptr->key=1;/*...alustetaan datakentät*/
ptr->link=NULL;/*alustetaan linkkikenttä*/

/*lisätään pinoon */
push(&top,&ptr);

/*poistetaan pinosta*/
ptr=pop(&top);

/*kun tietuetta ei enää tarvita*/
delete ptr;
ptr=NULL;
```

2.4.3 Linkitetty jono

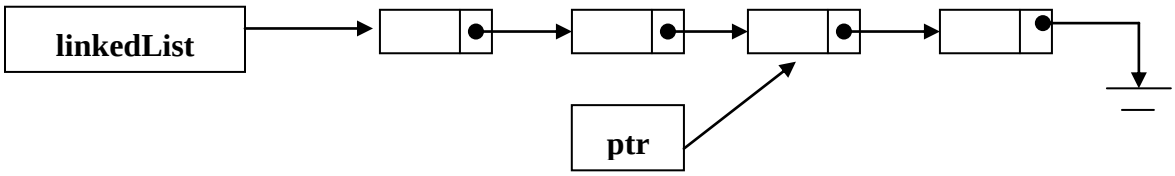


Linkitetty jono on linkitetty lista, jossa lisäykset ja poistot tehdään aina vastakkaisiin päihin listaa. Linkitetyn jonon käsittelyyn käytetään kahta **solmu -tyyppistä osoitinmuuttujaa** `front` (keula) ja `rear` (perä), joiden avulla ”pidetään kiinni” listan päistä.

```
/* Jonoon lisäys enqueue-funktiolla. Parametreina ovat osoittimen
osoittimet jonon keulaan ja häntään sekä osoitin lisättävään
tietueeseen.*/
void enqueue(NODEPTR *front, NODEPTR *rear, NODEPTR *newNode)
{
      if(*rear) {
            (*rear)->link = *newNode;
            *rear= *newNode;
      }
      else {
            *rear=*newNode;
            *front=*newNode;
      }
}

/* Jonosta poisto dequeue-funktiolla. Parametreina ovat osoittimen
osoittimet jonon keulaan ja häntään. Tässä palautetaan tietueosoitin
jonon hännässä olevaan tietueeseen*/

NODEPTR dequeue(NODEPTR *front, NODEPTR *rear)
{
      NODEPTR ptr = *front;
      if (*front) {
            *front = (*front)->link;
            if(!(*front)) //Jos jono tyhjeni
                  *rear=NULL;
            ptr->link=NULL;
      }
      return ptr;
}
```

Kutsuvassa ohjelman osassa linkitetyn jonon luonti, käsittelyfunktioiden käyttö, dynaamisen muistin varaaminen ja sen vapauttaminen tapahtuvat samalla periaatteella kuin pinon tapauksessa. Huomaa, että koodissa päivitetään molemmat osoitinmuuttujat (`front` ja `rear`) kriittisissä tilanteissa, joissa joko lisätään tyhjään jonoon tai poistetaan jonosta viimeinen solmu. Jos jono on tyhjä, palautuu NULL osoitin.

2.4.4 Linkitetty (järjestetty) lista



Linkitetty järjestetty lista eroaa staattisesta taulukosta siinä, että listan koko muuttuu ajonaikana lisäyksien ja poistojen yhteydessä. Lisäksi alkioita ei ole indeksoitu vaan ne on linkitetty ketjuun, jolloin listan alkioihin päästään käsiksi kulkemalla ketjua pitkin alkiosta toiseen. Listan käsittelyoperaatiot tarvitsevat osoitintyyppisiä apumuuttujia (`ptr`).

Linkitetyn järjestetyn listan toteutus

Koska listaa pidetään järjestyksessä jonkin avainkentän suhteen, uuden solmun lisääminen voi tapahtua mihin tahansa kohtaa listasta. Tämä aiheuttaa hieman ongelmallisuutta lisäys- ja poisto-operaatioiden toteuttamisessa. Kahteen suuntaan linkitetty lista mahdollistaa listassa liikkumisen molempiin suuntiin, jolloin lisäys- ja poisto-operaatiot ovat hieman helpompia toteuttaa ja tehokkaampia suorittaa. Tällöin solmun täytyy sisältää kaksi linkkikenttää eteen ja taaksepäin (tai vasen ja oikea). Linkitetty lista voidaan toteuttaa myös rengaslistana.

Suurien tietoalkioiden lisääminen linkitettyyn rakenteeseen on tehokasta, kun tietokenttien kopiointia ei tehdä, vaan linkitys tapahtuu muistiosoitteiden sijoituksilla. Linkitetyn listan käsittely vaatii edellä määriteltyjä **solmu -tyyppisiä osoitinmuuttujia**, joiden avulla ”pidetään kiinni” listan alusta ja käydään listaa läpi.

```
/* listarakenteen määrittely samalla tavalla kuin edellä pinolla
ja jonolla*/


typedef struct NODE *NODEPTR;

typedef struct {
      int key;
      // other datafields
      NODEPTR link;

};
```

Järjestettyyn listaan **lisääminen**:

```
void listInsert(NODEPTR *list, NODEPTR *newNode){
      NODEPTR ptr=*list;

      if(*list) {
            /* lisätään listan alkuun*/
            if(ptr->key >=(*newNode)->key) {
                  (*newNode)->link=ptr;
                  *list=*newNode;
            }
            /* haetaan listassa oikea paikka*/
            else {
                  while(ptr->link &&
                              ptr->link->key < (*newNode)->key)
                        ptr=ptr->link;

                  if(ptr->key < (*newNode)->key){
                  /* lisätään välille*/
                        (*newNode)->link=ptr->link;
                        ptr->link=*newNode;
                  }
                  /* lisätään loppuun*/
                  else {
                        ptr->link=*newNode;
                  }
            }
      }
      else *list=*newNode; /*tyhjän listan tapaus*/

}
```

Järjestetystä listasta **poisto**

```
listPtr listDelete(NODEPTR *list, int key)
{
      listPtr ptr=*list,temp;
      if(*list) {
            /* poistetaan solmu listan alusta*/
            if(ptr->key ==key) {
                  *list=ptr->link;
                  ptr->link=NULL;
                  return ptr;
            }
            /* haetaan listassa oikea paikka*/
            else {
                  while(ptr->link && ptr->link->key !=key)
                        ptr=ptr->link;

                  if(ptr->link && ptr->link->key ==key){
                  /* poistetaan solmu listasta*/
                        temp=ptr->link;
                        ptr->link=ptr->link->link;
                        temp->link=NULL;
                        return temp;
                  }
                  /* Avaimen mukaista solmua ei löydy*/
                  else return noSuchNodeExist(key);
            }
      }
      else return emptyList();/* Handle empty list case*/

}
```

Listan **tulostaminen**:

```
void printList(NODEPTR *list){
      NODEPTR ptr=*list;
      while(ptr) {
            printf("solmu %i\n",ptr->key);
            ptr=ptr->link;
      }
}
```

Kutsuvassa ohjelman osassa linkitetyn listan luonti, käsittelyfunktioiden käyttö, dynaamisen muistin varaaminen ja sen vapauttaminen tapahtuvat samalla periaatteella kuin pinon ja jonon tapauksissa.

2.4.5 Dynaamisten rakenteiden sovellusesimerkki

Abstrakteilla tietotyypeillä **pino** ja **jono** voidaan toteuttaa kaksi tärkeää **verkkojen läpikäyntialgoritmia** *syvyys-ensin-haku* ja *leveys-ensin-haku*. Verkkorakenteiden esittämiseen tietokoneohjelmissa voidaan käyttää **linkitettyjä listoja**. Verkot ovat tietorakenteita, jotka koostuvat solmuista ja niitä yhdistävistä väleistä (yhteyksistä). Solmujen läpikäynti on mahdollista vain solujen välisten yhteyksien kautta.

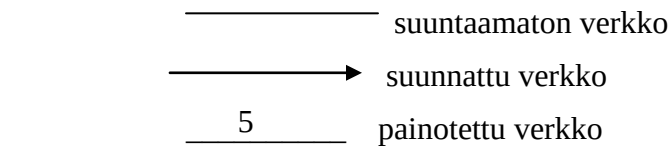
Verkon määritelmä

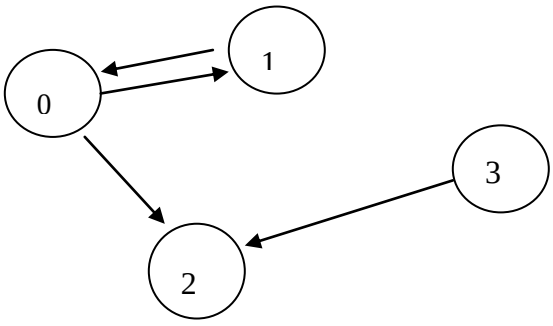
Verkko (eli Graafi, Graph) **G** on järjestetty pari,

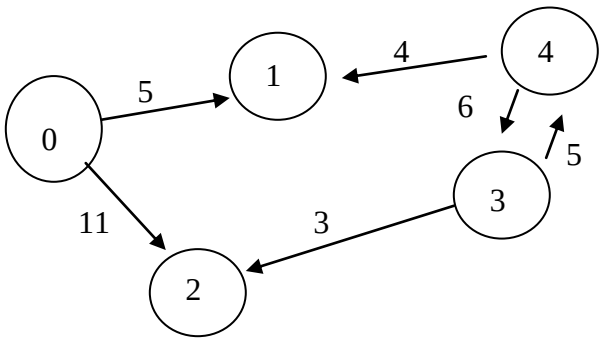
G = (V, E), missä **V** on solmujen epätyhjä joukko ja välien joukko **E** on relaatio solmujen joukossa. Relaatio on karteesisen tulojoukon osajoukko $E \subset V \times V$. Karteesinen tulojoukko *E* on järjestettyjen parien (x, y) joukko, jossa $x, y \in V$ eli

$$V \times V = \{(x, y) | x \in V \text{ ja } y \in V\}$$

Verkon kuvauksia







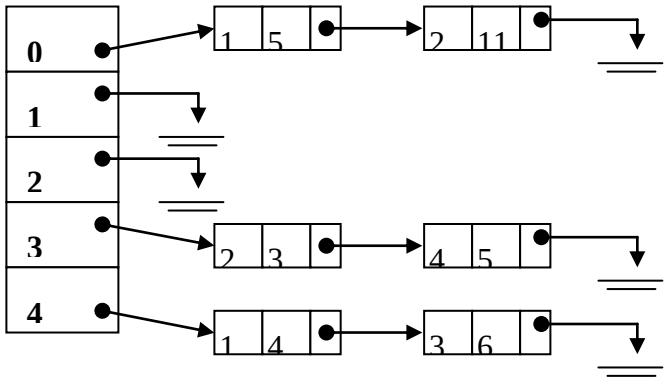
Verkkojen esittäminen ohjelmissa

Verkot voidaan esittää tietokoneohjelmissa esimerkiksi matriisien avulla. Alla ns. kustannusmatriisiesitys eli edelliselle painotettu verkolle:

$$\begin{bmatrix} \infty & 5 & 11 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 3 & \infty & 5 \\ \infty & 4 & \infty & 6 & \infty \end{bmatrix}$$

Matriisin alkio `aij` ilmoittaa yhteyden painon verkon solmusta `i` solmuun `j`. Matriisiesityksessä ∞ merkitsee, että yhteyttä ei ole. Verkon solmut voidaan käydä läpi kahdella sisäkkäisellä toistosilmukalla → Kompleksisuusluokka **O(N^2)**.

Linkitettyjen listojen avulla verkot esitetään ns. **vieruslistaesityksenä**. Siinä muodostetaan osoitintaulukko, jonka kukin alkio vastaa tiettyä verkon solmua. Alkiot viittaavat linkitettyyn listaan, joka sisältää kaikki kyseisen solmun vierussolmut.



Verkkojen läpikäyntialgoritmit

Yleisin tapa käydä verkon solmut läpi on joko *syvyys-ensin-haku* tai *leveys-ensin-haku* periaate.

Syvyys-ensin algoritmi perustuu pinon käyttöön ja *leveys-ensin* algoritmi jonon käyttöön.

Syvyys-ensin-haussa lähdetään liikkeelle jostakin lähtösolmusta, joka merkitään käydyksi, ja edetään seuraavaan käymättömään solmuun, joka merkitään käydyksi. Näin edetään verkossa niin ”syvälle” kuin mahdollista. Jos eteneminen ei ole mahdollista, palataan taaksepäin (käytyyn) solmuun, josta päästään taas eteenpäin käymättömiin solmuihin. Paluun toteuttaminen edellyttää pinon käyttöä.

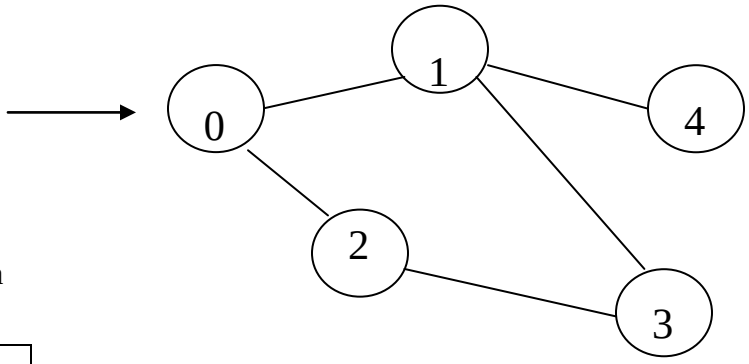
Leveys-ensin-haussa lähdetään liikkeelle jostakin lähtösolmusta, joka merkitään käydyksi, ja käydään seuraavaksi läpi kaikki lähtösolmun lapsisolmut. Seuraavaksi käydään läpi kaikki ensimmäisen lapsisolmun lapsisolmut ja sitten seuraavan lapsisolmun lapsisolmut jne. Leveyshaun toteuttaminen edellyttää jonon käyttöä.

Syvyys-ensin-haku

1. Valitaan jokin käymätön solmu alkusolmuksi, merkitään se käydyksi ja laitetaan pinoon.
2. Jos kaikki käyty lopetetaan.
3. Jos ei ole käymättömiä vierussolmuja, siirrytään kohtaan 4. Muutoin edetään väliä pitkin johonkin käymättömään vierussolmuun, merkitään se käydyksi ja laitetaan pinoon sekä siirrytään kohtaan 2.
4. Otetaan pinosta solmu. Jos pino on tyhjä, siirrytään kohtaan 1, muutoin siirrytään kohtaan 2.

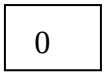
Esimerkki: Valitaan 0 alkusolmuksi ja valitaan systemaattisesti pienin mahdollinen vierussolmu. Taulukossa T = on käyty, F = käymätön solmu.

alkusolmu



Valitaan 0 ja laitetaan se pinoon

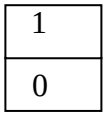
T	F	F	F	F
0	1	2	3	4



Solmulla 0 on käymättömiä vierussolmuja,

valitaan 1 ja laitetaan se pinoon

T	T	F	F	F
0	1	2	3	4



Solmulla 1 on käymättömiä vierussolmuja,

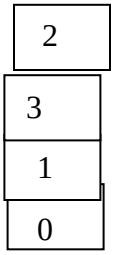
valitaan 3 ja laitetaan se pinoon

T	T	F	T	F
0	1	2	3	4

Solmulla 3 on käymättömiä vierussolmuja,

valitaan 2 ja laitetaan se pinoon

T	T	T	T	F
0	1	2	3	4



Ei käymättömiä vierussolmuja

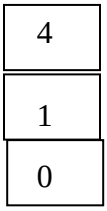
Otetaan 2 pois ja yritetään jatkaa solmusta 3.

Solmulla 3 ei käymättömiä vierussolmuja.

Otetaan 3 pois ja yritetään jatkaa solmusta 1

Solmulla 1 on käymättömiä vierussolmuja,

valitaan 4 laitetaan se pinoon



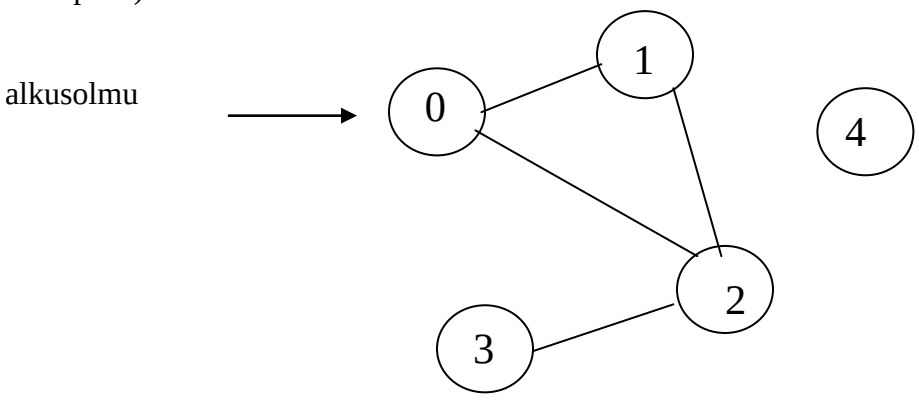
T	T	T	T	T
0	1	2	3	4

Kaikki käyty lopetetaan.

Leveys-ensin-haku

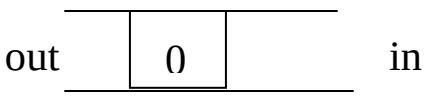
1. Valitaan jokin käymätön solmu alkusolmuksi, merkitään se käydyksi ja laitetaan jonoon.
2. Jos kaikki käyty, lopetetaan.
3. Otetaan jonosta solmu ja merkitään kaikki sen käymättömät vierussolmut käydyksi ja laitetaan ne jonoon
4. Jos jono on tyhjä ja on käymättömiä solmuja, mennään kohtaan 1, muutoin kohtaan 2.

Esimerkki: (Valitaan 0 alkusolmuksi ja valitaan vierussolmu systemaattisesti pienimmästä suurimpaan)



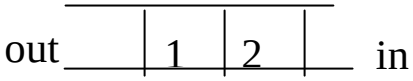
Valitaan 0 ja laitetaan se jonoon

T	F	F	F	F
0	1	2	3	4



0 pois, käymättömät vierussolmut 1 ja 2 jonoon

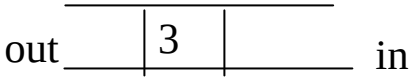
T	T	T	F	F
0	1	2	3	4



1 pois, ei käymättömiä vierussolmuja

2 pois, käymätön vierussolmu 3 jonoon

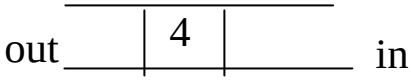
T	T	T	T	F
0	1	2	3	4



3 pois, ei käymättömiä vierussolmuja ja jono tyhjä.

Valitaan 4 ja laitetaan se jonoon

T	T	T	T	T
0	1	2	3	4



Kaikki käyty, lopetetaan (Kaksi yhtenäistä komponenttia)

Algoritmien analysointia

Sekä *syvyys-ensin* että *leveys-ensin* algoritmeissa jokainen vieruslista käydään läpi korkeintaan kerran, joten algoritmien kompleksisuusluokka on **O(N)** eli ne ovat erittäin hyviä algoritmeja. Molemmilla algoritmeilla verkko voi olla suunnattu tai suuntaamaton, yhtenäinen tai epäyhtenäinen. Algoritmeilla saadaan helposti myös selville verkon yhtenäisten komponenttien lukumäärä.

Tietorakenteet ja algoritmit

Kari Peisa

Ramk/Tekniikka ja liikenne 2006

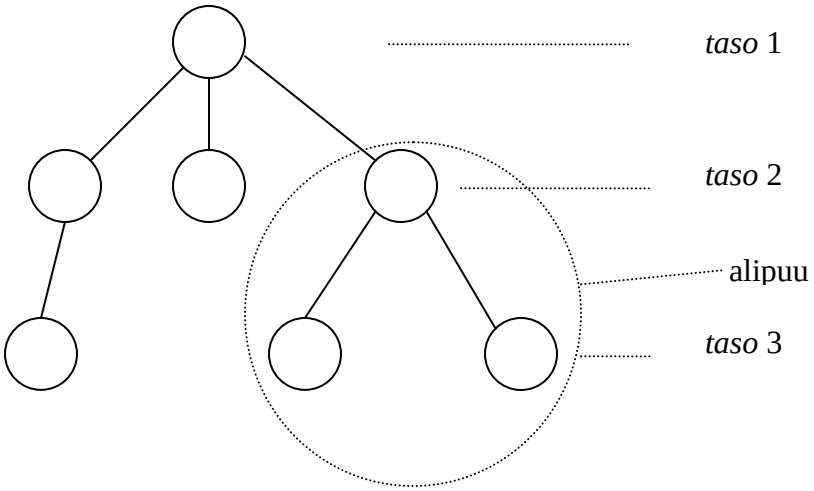
Kari.peisa@ramk.fi

2 Tietorakenteet

2.5 Puut

Määritelmä: Puu on yhtenäinen, syklitön ja suuntaamaton verkko.

Puut voidaan toteuttaa myös suunnattuna verkkona.



Juuri. Jokaisella puulla yksi solmu toimii juurena.

Mikä tahansa solmu voi toimia juurena. Puun solmu on **isäsolmu**, jos se haarautuu (juuresta katsoen) yhdeksi tai useammaksi **lapsisolmuksi**. Lapsien lukumäärä = **solmun aste** (degree). Jokaisella solmulla (paitsi juurella) on vain yksi isä. **Haarautumissolmu** on solmu, jolla on sekä isä että lapsi(a). **Lehtisolmu** (tai päätesolmu) on solmu, jolla ei ole lapsia.

Puun tasot

Juurisolmu tasolla 1, sen lapset tasolla 2, näiden lapset tasolla 3 jne.

Tasojen lukumäärä = "alimpien" lehtisolmujen taso = **puun korkeus**

Puun ominaisuuksia

- **Maksimaalisesti syklitön**. "Yksikin väli lisää --> sykli"
- **Minimaalisesti yhtenäinen**. "Yksikin väli pois --> epäyhtenäinen verkko"
- Puussa on **1-käsitteinen polku juuresta jokaiseen muuhun solmuun**

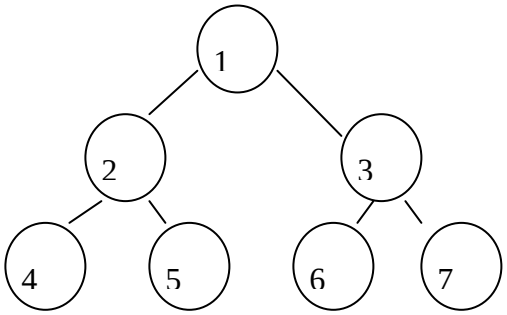
Puita käytetään paljon **tallennusrakenteina**. Tällöin jo puun rakentamisessa noudatetaan määrättyjä periaatteita, jolloin puun läpikäyminen tai tietyn avaindatan etsiminen puusta voidaan toteuttaa tehokkaasti. Seuraavassa tarkastellaan esimerkkinä **binääripuuta**.

2.5.1 Binääripuu

Määritelmä: Binääripuu on joko tyhjä tai se on sellainen suunnattu puu, jonka jokaisella solmulla on korkeintaan kaksi lasta.

Rekursiivinen määrittely: Binääripuu on äärellinen solmujen joukko, joka on joko tyhjä tai koostuu juurisolmusta sekä kahdesta erillisestä binääripuusta, joita kutsutaan juurisolmun vasemmaksi ja oikeaksi **alipuuksi**. (Binääripuiden algoritmit ovat yleensä rekursiivisia).

Binääripuun solmujen **indeksointi**

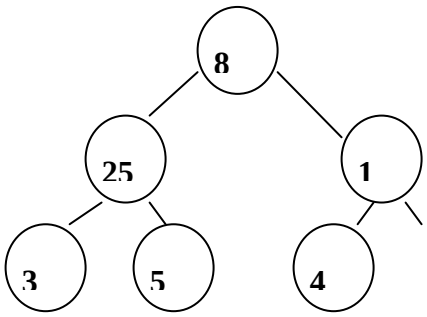


- Juuren indeksi = 1, juuren taso = 1
- Lapsen indeksi = 2 * isän indeksi tai 2 * isän indeksi + 1
- Isän indeksi = $\lfloor$ lapsen indeksi / 2$\rfloor$ = suurin kokonaisluku, joka on pienempi tai yhtä suuri kuin lapsen indeksi jaettuna kahdella.
- välien lukumäärä = solmujen lukumäärä - 1
- Tasolla k korkeintaan 2^{k-1} solmua
- Kun kaikki solmupaikat täytetty, on puun korkeus l. tasojen lukumäärä = k sekä solmujen lukumäärä = 2^k-1,
- Alimmalla tasolla 2^k nollalinkkiä

Täydellinen binääripuu
Binääripuuta, jonka tasot ovat täynnä solmuja eli kaikilla lehtisolmuilla on sama korkeus ja kaikkien sisäsolmujen aste on kaksi, kutsutaan **täydelliseksi binääripuuksi** (**complete binary tree**). Melkein täydellisessä binääripuussa tasot ovat täynnä lukuun ottamatta mahdollisesti viimeistä tasoa, joka voi olla osittain täytetty vasemmalta lähtien.

Binääripuun solmujen indeksointiperiaatteen mukaan binääripuu voidaan esittää myös peräkkäisenä indeksoituna listana. Tämä tarkoittaa sitä, että (melkein) täydellinen binääripuu voidaan esittää esimerkiksi C-kielen taulukossa siten, että jokaista binääripuun solmua vastaa yksi taulukon solu ja myös päinvastoin. Taulukon koko tulee määritellä yhtä suuremmaksi kuin solmujen lukumäärä.

Esimerkki. Täydellisen (myös melkein) binääripuun esittäminen C-taulukossa.



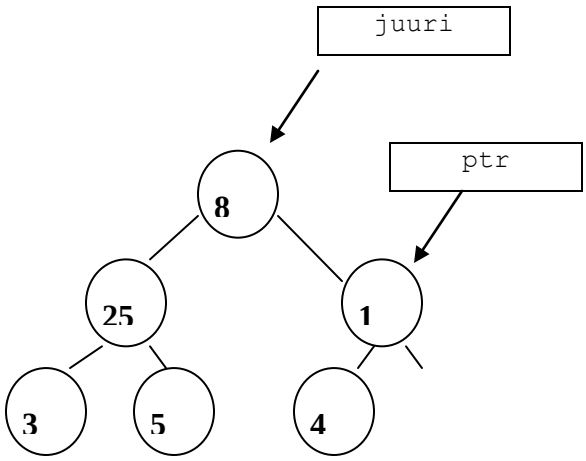
-	**8**	**25**	**1**	**3**	**5**	**4**
0	1	2	3	4	5	6 ← Indeksi C-taulukossa
-	1	2	3	4	5	6 ← Indeksi Puussa

Solmun **`taulu[k]`** lapset ovat **`taulu[2k]`** ja **`taulu[2k+1]`**. Lisäksi lapsisolmun **`taulu[k]`** isäsolmu on **`taulu[k/2]`**, olipa kyse kummasta lapsesta hyvänsä, kun **`k`** on C-kielen kokonaislukumuuttuja. C-kielen taulukossa binääripuu toteutetaan siten, että juurisolmu talletetaan indeksille 1. Tällöin binääripuussa voidaan ”liikkua” alaspäin tai ylöspäin taulukon indeksimuuttujan avulla. Tämä mahdollistaa erilaisten puun läpikäyntialgoritmien toteuttamisen myös taulukkopohjaisesti.

2.5.2 Linkitetty binääripuu

Dynaaminen binääripuun rakenne muodostetaan seuraavalla solmun tietuemäärittelyllä:

```
typedef struct tree_node *tree_ptr;

typedef struct tree_node {
      int key;
      /* other data fields */

      tree_ptr left_child;
      tree_ptr right_child;

}tree_node;
```

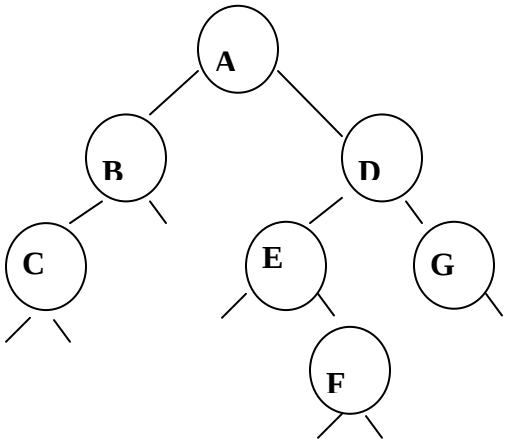


Linkitettyä binääripuuta hallitaan samalla tavalla kuin edellä olevia linkitettyjä rakenteita (pino, jono, lista). Puun juurisolmun osoite pidetään ohjelman hallinnassa ja puun läpikäynti toteutetaan solmutyyppisen osoitinmuuttujan ja solmujen sisältämien linkkikenttien `left_child` ja `right_child` avulla.

Seuraavassa esitellään ensin kolme **binääripuun läpikäyntialgoritmia** ennen varsinaista binääripuun rakentamista. Vasta tämän jälkeen tarkastellaan **binääristä etsintäpuuta**, joka varsinaisesti on esimerkki tallennusrakenteesta ja määritellään sille lisäys-, poisto- ja hakualgoritmit sekä sovelletaan em. läpikäyntialgoritmeja tulostettaessa puun solmujen avainkentät halutussa järjestyksessä.

2.5.3 Binääripuun läpikäynti

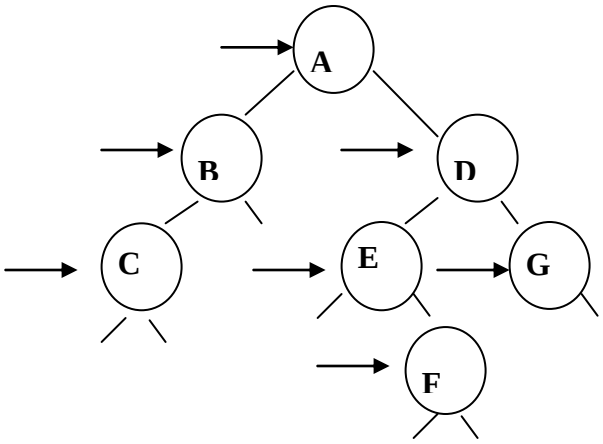
Binääripuun läpikäynti voidaan halutessa toteuttaa edellä esiteltyjen verkkojen läpikäyntialgoritmien (syvyys-ensin tai leveys-ensin) avulla, mutta silloin puusta tulisi rakentaa erillinen vieruslistaesitys (kts. pinon ja jonon sovelluksia). Seuraavassa esitellään kolme yleistä binääripuun läpikäyntialgoritmia, joilla binääripuuhun tallennettu tieto voidaan käsitellä systemaattisesti ja tehokkaasti ilman erillistä vieruslistaesitystä.



Tarkastellaan esimerkkinä oheista binääripuuta

Esijärjestys

Puu käydään läpi juuresta lähtien edeten aina vasempaan lapsisolmuun ennen oikeaa ja käsittelemällä solmut siinä järjestyksessä, kun ne voidaan ”nähdä vasemmalta”.



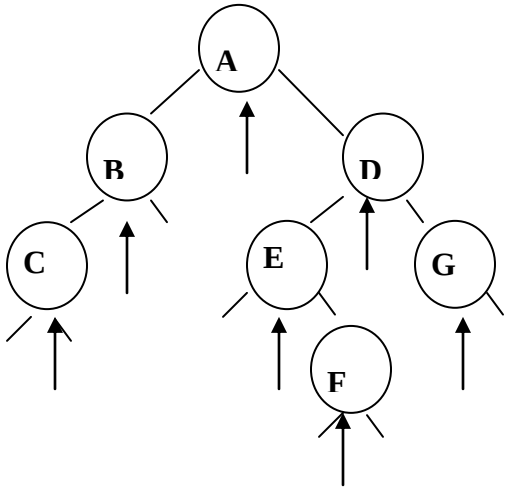
A B C D E F G

C-kielinen algoritmi (tässä tulostetaan kunkin solmun avainkenttä)

```
void preorder(tree_ptr ptr){
      if(ptr){
            printf("%d ",ptr->key);
            preorder(ptr->left_child);
            preorder(ptr->right_child);
      }
}
```

Sisäjärjestys

Puu käydään läpi juuresta lähtien edeten aina vasempaan lapsisolmuun ennen oikeaa ja käsittelemällä solmut siinä järjestyksessä, kun ne voidaan ”nähdä alhaalta”.



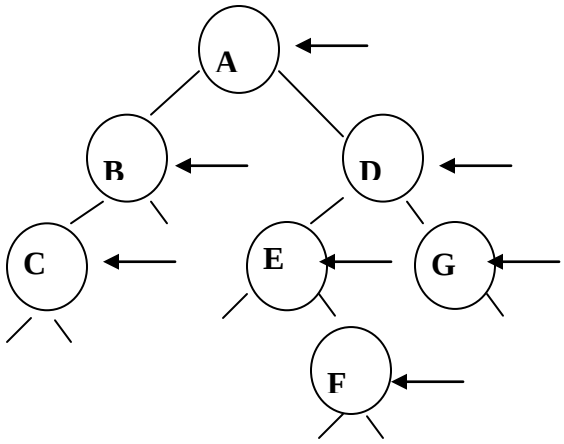
C B A E F D G

C-kielinen algoritmi

```
void inorder(tree_ptr ptr){
      if(ptr){
            inorder(ptr->left_child);
            printf("%d ",ptr->key);
            inorder(ptr->right_child);
      }
}
```

Jälkijärjestys

Puu käydään läpi juuresta lähtien edeten aina vasempaan lapsisolmuun ennen oikeaa ja käsittelemällä solmut siinä järjestyksessä, kun ne voidaan ”nähdä oikealta”.



C B F E G D A

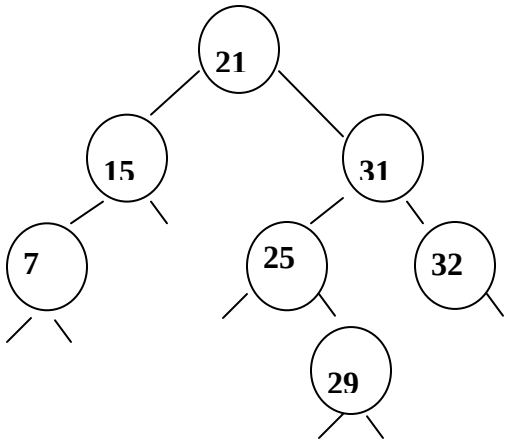
C-kielinen algoritmi

```
void postorder(tree_ptr ptr){
      if(ptr){
            postorder(ptr->left_child);
            postorder(ptr->right_child);
            printf("%d ",ptr->key);
      }
}
```

2.5.4 Binäärinen etsintäpuu

Binäärinen etsintäpuu (binary searc tree) täyttää seuraavat ominaisuudet:

- Etsintäpuu on binääripuu.
- Jokaisella solmulla tulee olla yksikäsitteinen avainkenttä.
- Kunkin solmun avainkenttä on suurempi kuin kaikkien sen vasemmassa ei-tyhjässä alipuussa olevien solmujen avainkenttien arvot.
- Kunkin solmun avainkenttä on pienempi kuin kaikkien sen oikeassa ei-tyhjässä alipuussa olevien solmujen avainkenttien arvot.
- Sisäjärjestys antaa puussa olevien solmujen avainkenttien arvot suuruusjärjestyksessä



Tiedon hakeminen (tai lisääminen) etsintäpuusta on kompleksisuusluokkaa **O(LOG N)**, kun etsintäpuu **tasapainossa** (ns. **AVL**-puu l. **A**delson-**V**elskii-**L**andis-puu). **Tasapainossa olevassa etsintäpuussa jokaisen solmun alipuiden korkeusero on korkeintaan yksi**. Edellä oleva etsintäpuu on tasapainossa.

Etsittäessä avaindataa etsintäpuusta voidaan hakualgoritmissa edetä suoraan alaspäin puurakenteessa ilman paluuta edellä lueteltujen ominaisuuksien perusteella, mikä takaa joka tapauksessa lineaarisen kompleksisuuden (**O(N)**). Tasapainossa olevassa etsintäpuussa, jossa on **N** solmua, siirryttäessä juuresta alimmalle tasolle voidaan ottaa korkeintaan **LOG N** askelta, joten silloin algoritmin kompleksisuusluokkaa on logaritminen. Etsintäpuun tasapainottamisen algoritmeja ei tarkastella tässä kurssissa.

Ohessa on koodiesimerkki rekursiivisesta hakualgoritmista avainkentän perusteella.

```
tree_ptr binSearch(tree_ptr root, int key) {
      if(!root) return NULL;
      if(root->key==key)
            return root;
      if(key < root->key)
            return binSearch(root->left_child, key);
      else binSearch(root->right_child,key);
}
```