# Andmehaldus

Bititöötlus, omaloodud voog, kirjed, puu

## *Bitid*

Ehkki kõrgkeeltes programme kirjutades ei pea me liialt pead vaevama arvuti sisemuses toimetava kahendsüsteemi üle, võib mõnel pool vastavatest teadmistest ikkagi tulu tõusta. Eriti olukordades, kus korraga on vaja üle kanda või salvestada hulgem tõeväärtusi. Nõnda näiteks hiirevajutuse sündmuse juures saab käskluse getModifiersEx kaudu küsida int-tüüpi väärtuse, mille igast bitist saab välja lugeda kas hiirenupu või mõne klaviatuurinupu asukoha. Samuti ei pääse bititehetest pakkimisalgoritmide või muude baidi sisu lahkavate tegevuste puhul.

Bititehetel kasutatakse operaatoreid &, |, <<, >> ning >>> ning neid saab kasutada täisarvude puhul. Esimene neist käitub sarnaselt kui &-tehe tavalistegi tõeväärtuste puhul, st., et tehte tulemus loetakse tõeseks ainult siis, kui mõlemad osapooled on tõesed. Ainult, et arvude puhul võetakse sellesse tehtesse kõik bitid eraldi ning tulemuse arvutamisel pannakse kõik bitid jälle üksteise järgi ritta. Näiteks

1100 ehk 12 ning
0110 ehk 6 annavad & tehte tulemusena kokku
0100 ehk 4.

Kui tarvis arvus kontollida üksiku biti asendit, siis on & hea mugav tehe. Kui &-tehe teha väärtusega, kus vaid üks bitt püsti, siis tulemus saab olla kas null või siis seesama ühe püstise bitiga väärtus. Esimesel juhul ei sattunud vastav bitt kohakuti, teisel juhul sattus.

Sarnaselt võib kontrollida, kas tegemist on paarisarvuga. Kui arvu viimane bitt on 0, siis arv jagub kahega, sest kõik vasakpoolsemad bitid on kahendsüsteemi ülesehituse tõttu paratamatult arvu 2 kordsed. Kui aga parempoolseim bitt on püsti, siis peab arv olema paaritu.

```
public class Bitid1{
  public static void main(String[] argumendid){
    int arv=6;
    if((arv & 1) == 0){
      System.out.println("Paarisarv");
    } else {
      System.out.println("Paaritu arv");
    }
  }
}
```

Märk << väljastab oma vasaku operandi väärtuse nihutatuna vasakule paremal pool kirjutatud arvu jagu kohti. Üldjuhul see annab sama tulemuse kui arvu kahega korrutamine. Kuna kahendsüsteemi bittide väärtused alates paremalt on 1, 2, 4, 8, 16 jne., siis bitte ühe võrra vasakule lükates justkui korrutataks iga moodustavat arvu kahega. Sarnaselt nagu kümnendsüsteemis suurendatakse nulli lõppu lisamisega arvu väärtust kümme korda.

```
public class Bitid2{
  public static void main(String[] argumendid){
    int arv=6;
    arv = arv << 1;
    System.out.println("Kahega korrutatult: "+arv);
  }
```

```
}
```

Tahtes arvu lähemalt uurida, tasub vaadata enamasti rohkem kui ühe biti väärtusi. Kui vaadata arvu viimast bitti ning igal korral bitid paremale nihutada, sellisel juhul trükitakse arvu bitid tagurpidises järjekorras. Kui aga vaadata igal korral arvu parempoolseima baidi vasakpoolseima biti väärtust (eraldi võetuna kaks astmes seitse ehk 128) ning siis iga võrdluse järel arvu ühe biti jagu vasakule nihutada, siis saab tulemusena kätte parempoolseima baidi kõik bitid vasakult paremale.

```
public class Bitid3{
  public static void main(String[] argumendid){
    int arv=67;
    System.out.println("Viimased kaheksa bitti:");
    for(int i=0; i<8; i++){
     if((arv & 128) !=0){System.out.print("1");}
     else {System.out.print(0);}
     arv = arv << 1;
    }
  }
}
/*
D:\Kasutajad\jaagup\java>java Bitid3
Viimased kaheksa bitti:
01000011
*/
```

## Bitinihutuskrüptograafia

Kui soovitakse tekst silma ees mõnevõrra loetamatumaks muuta, siis selleks on välja töötatud hulgem võimalusi. Aastatuhandete jooksul on levinumateks võteteks olnud tähtede järjekorra muutmine ning tähtede asendamine. Nüüdisaegsemas raalipõhises krüpteerimises on tähtede asemele tulnud baidid ja bitid ning mitmed päris põhjalikud meetodid. Kuid nii nagu näiteks EditPad Lite-nimelises ja mõnes muuski tekstiredaktoris kasutatakse ROT-13 nimelist täheasendust nihutades inglise tähestiku tähti kolmeteistkümne koha võrra, nii on siin näites liigutatud baidi bitte ringlevalt ühe koha võrra vasakule ning vasakpoolseim bitt paigutatakse parempoolseks.

```
public class Bitid4{
  /**
  * Väljastatakse arvu viimase kaheksa biti väärtused.
  */
  public static void kirjutaBitid(int arv){
    for(int i=0; i<8; i++){
     if((arv & 128) !=0){System.out.print("1");}
     else {System.out.print(0);}
     arv = arv << 1;
    }
    System.out.println();
  }
  /**
  * Nihutab arvu parempoolsed seitse bitti vasakule ning
  * kaheksanda paneb parempoolseks bitiks.
  */
  public static int keeraVasakule(int arv){
    int abi=arv & 128;
    arv = arv << 1;
    arv = arv | (abi >> 7);
    return arv;
  }
  public static void main(String[] argumendid){
    int arv=67;
    kirjutaBitid(arv);
```

```
    arv=keeraVasakule(arv);
    kirjutaBitid(arv);
    arv=keeraVasakule(arv);
    kirjutaBitid(arv);
  }
}

/*
D:\Kasutajad\jaagup\java>java Bitid4
01000011
10000110
00001101
*/
```

## Baidi bitid failist.

Suurema bitimahuga on pistmist binaarfailide puhul - olgu siis lugemise või kirjutamise juures. Madalama taseme vood ongi loodud arvestusega, et seal on käsud vaid baitide lugemiseks ja kirjutamiseks. Nõnda on lihtsam keskenduda konkreetse sihtseadme jaoks voo loomisele. Vajalikud andmetüüpide muundused saab korda ajada mähisklasside abil. Baidi lugemiseks tuleb avada voog - praeguse näite puhul failist. Iga read-käsklus loeb ühe baidi mille väärtus talletatakse int-tüüpi muutujasse parempoolseks baidiks ehk muutuja saab väärtuse vahemikus 0-255.

```
    FileInputStream fis=new FileInputStream("ataht.txt");
    int arv=fis.read();
```

Ning töötav kood tervikuna. Eeldatakse, et fail nimega ataht.txt on olemas.

```
import java.io.*;
public class Bitid5{
  public static void kirjutaBitid(int arv){
    for(int i=0; i<8; i++){
     if((arv & 128) !=0){System.out.print("1");}
     else {System.out.print(0);}
     arv = arv << 1;
    }
    System.out.println();
  }
  public static void main(String[] argumendid) throws IOException{
    FileInputStream fis=new FileInputStream("ataht.txt");
    int arv=fis.read();
    kirjutaBitid(arv);
    fis.close();
  }
}
/*
D:\Kasutajad\jaagup\java>java Bitid5
01100001
*/
```

## Bitikaupa failikirjutus

Faili bittide kirjutamiseks on mugav teha omaette alamprogramm, mille ülesandeks on hoolitseda, et bitid ilusti ükshaaval baidiks kokku loetaks ning sobival hetkel faili kirjutataks. Et faili saab korraga kirjutada terve baidi, siis peab pidevalt arvet pidama, et õige arv bitte enne kokku saaks kui andmeid kirjutama hakatakse. Samuti peab kirjutamisel ja lugemisel teadma, kummast baidi otsast bitte paigutama hakatakse. Ning üldjuhul on bitte faili mõistlik kirjutada kaheksa kaupa, sest muul juhul pole kuigi lihtne kindlaks teha, kas ülejäänud bittidesse kirjutati nullid või on need väärtused lihtsalt täitmata.

Siin näites on püütud läbi ajada staatiliste funktsioonidega, ilma oma objekte loomata. Et aga faili väljundvoo loomisel võib tekkida erind, siis vastavat voo isendit ei sa muutujate deklareerimise juures luua. Välja aitab staatiline initsialiseerimisplokk, milles võimalik tekkiv erind ka kinni püüda ja töödelda.

Edaspidi piisab programmeerijal vaid sobiva parameetriga välja kutsuda funktsioon nimega kirjutaBittFaili. Muutujas nr loetakse, mitmenda bitiga on tegemist, muutujas malu hoitakse bittide lisamisel tekkivat poolfabrikaati enne tulemuse faili kirjutamist. Kui lisatakse püstine bitt, siis tõstetakse mälumuutuja viimane bitt üheks, muul juhul jäetakse nulliks. Püstitõstmistehe paistab järgnevalt:

```
malu |=1; //viimane bitt pannakse üheks
```

Samuti võiks kirjutada malu=malu|1. Arvu 1 puhul on ainuke püstine bitt viimane. Tehte | puhul loetakse tulemus tõeseks kui vähemalt üks osalevatest pooltest on tõene ning nõnda ükshaaval iga kahe arvu bitipaari korral. Nõnda jäävad arvuga 1 | (või) tehte puhul algse arvu kõik muud bitid paika, vaid viimane tõstetakse püsti sõltumata selle algsest väärtusest.

Edasi: kui kokku on kaheksa bitti täis kirjutatud, siis paigutatakse valmissaanud bait failivoogu ning asutakse taas tühja baidi bitte täitma. Kui aga kaheksat veel täis pole, siis nihutatakse mälumuutujas olevaid kõiki bitte ühe biti jagu vasakule nagu allpool käsus näha on.

```
malu <<=1;
```

oleks pikalt välja kirjutatuna
malu = malu << 1, mis siis tähistabki väärtuse kõigi bittide ühe jagu vasemale lükkamist nõnda, et parempoolseks bitiks tekib 0.

```
if (nr==8){
  fos.write(malu);
  malu=0;
  nr=0;
} else {
  malu <<=1;
}
```

Tervikuna kogu bitikaupa andmeid faili kirjutav rakendus.

```
import java.io.*;
public class Bitid6{
  static FileOutputStream fos;
  static int nr=0;
  static int malu=0;
  static{  //staatiline initsialiseerimisplokk
    try{
      fos=new FileOutputStream("bitijada.dat");
    } catch(IOException e){
      System.out.println(e);
    }
  }
  public static void kirjutaBittFaili(boolean bitt) throws IOException{
    if(bitt){
      malu |=1; //viimane bitt pannakse üheks
    }
    nr++;
    if (nr==8){
      fos.write(malu);
      malu=0;
      nr=0;
    } else {
      malu <<=1;
    }
  }
  public static void main(String[] argumendid) throws IOException{
     kirjutaBittFaili(false);
     kirjutaBittFaili(true);
     kirjutaBittFaili(true);
     kirjutaBittFaili(false);
     kirjutaBittFaili(false);
     kirjutaBittFaili(false);
```

```
     kirjutaBittFaili(true);
     kirjutaBittFaili(false); //kokku 98 ehk täht b
     fos.close();
  }
}
```

## Bitiväljundvoog

Et muundatud vooge tuleb mitmel pool ette, siis võib selle kirja panna ka omapoolse alamklassina. Nõnda võiks loodud klassi olla hiljem kergem soovitud muudes rakendustes kasutada. Päris standardseks vahendiks selliste voogude loomisel on FilterOutputStreami laiendamine. Siin on lihtsuse mõttes aluseks võetud FileOutputStream. Loodud uue klassi eksemplarile võib konstruktoris öelda soovitud failinime ning juba võibki asuda sinna bitte kirjutama. Bitti kirjutava funktsiooni ülesehitus on jäetud samaks kui eelmises näites. Kuid FileOutputStreami ülekatmise tõttu on siin võimalik otse kasutada kaasatulnud käsklust write ühe baidi faili kirjutamiseks. Sulgemise juures on lisatud kontroll, et andmed ikka terve baidi kaupa korraga kirjutataks. Muul juhul võiks kergesti juhtuda, et viimane bait jõuaks faili loetamatult. Voo sulgemiskäskluse edasisaatmine ülemklassile on samuti vajalik hoolitsemaks, et andmed mälupuhvrist ikka füüsilisele andmekandjale jõuaksid.

```
import java.io.*;

public class Bitid7{
  public static void main(String[] argumendid) throws IOException{
     BitiValjundVoog bvv=new BitiValjundVoog("bitijada.dat");
     bvv.kirjutaBitt(false);
     bvv.kirjutaBitt(true);
     bvv.kirjutaBitt(true);
     bvv.kirjutaBitt(false);
     bvv.kirjutaBitt(false);
     bvv.kirjutaBitt(false);
     bvv.kirjutaBitt(true);
     bvv.kirjutaBitt(false); //kokku 98 ehk täht b
     bvv.close();
  }
  static class BitiValjundVoog extends FileOutputStream{
    int nr=0;
    int malu=0;
    public BitiValjundVoog(String failinimi) throws IOException{
      super(failinimi);
    }
    public void kirjutaBitt(boolean bitt) throws IOException{
     if(bitt){
      malu |=1; //viimane bitt pannakse üheks
     }
     nr++;
     if (nr==8){
      write(malu);
      malu=0;
      nr=0;
     } else {
      malu <<=1;
     }
    }
    public int mitmesBitt(){
      return nr;
    }
    public void close() throws IOException{
      if(nr!=0){
        throw new IOException("Viimane bait pole valmis "+nr);
      }
      super.close();
    }
  }
}
```

## Bittide sisendvoog failist.

Mis kord faili kirjutatud, on kasulik sealt ka välja lugeda. Sarnaselt eelmisele näitele on voog koondatud omaette klassi, et vajadusel saaks seda sarnasena ka mõnes teises rakenduses kasutada. Voole on lisatud käsklus veel() kontrollimaks, et kas on veel bitte võimalik lugeda. Käsklus loeBitt väljastab voost tõeväärtusena järgneva biti.

Käsuga loeJargmine võetakse algandmeid sisaldavast failist ette järgmine bait. Seda nii voo avamisel kui olukorras, kus baidist kaheksa bitti juba loetud on. Kui failist lugemisel saabub -1 ehk voo lõputunnus, jäetakse siingi meelde, et enam midagi lugeda pole. Selle järgi funktsioon veel() teabki küsijale teada anda, et kas midagi lugeda on. Et bitid jääksid lugemisel näha ilusti vasakult paremale, selleks võrreldakse siingi vasakult seitsmenda biti väärtust ning iga küsimise järel liigutatakse kogu mälus oleva muutuja sisu ühe biti jagu vasakule. Ning nõnda võibki omaloodud voost lugeda bitte sarnaselt nagu FileInputStreamist loetakse baite, DataInputStreamist reaalarve või BufferedReaderist ridu. Lihtsalt kontrollitakse, kas midagi lugeda onn. Ning vastavalt biti väärtusele trükitakse välja praegu lihtsalt üks või null.

```
     while(bsv.veel()){
      System.out.print((bsv.loeBitt())?"1":"0");
     }
```

Näitrakenduse kood tervikuna.

```
import java.io.*;

public class Bitid8{
  public static void main(String[] argumendid) throws IOException{
     BitiSisendVoog bsv=new BitiSisendVoog("bitijada.dat");
     while(bsv.veel()){
       System.out.print((bsv.loeBitt())?"1":"0");
     }
     bsv.close();
  }
  static class BitiSisendVoog extends FileInputStream{
    int nr=0;
    boolean veel=true;
    int malu=0;
    public BitiSisendVoog(String failinimi) throws IOException{
      super(failinimi);
      loeJargmine();
    }

    public boolean veel(){
      return veel;
    }

    private void loeJargmine() throws IOException{
        malu=read();
        nr=0;
        if(malu==-1){veel=false;}
    }

    public boolean loeBitt() throws IOException{
      if(!veel){throw new IOException("Bitid otsas");}
      boolean vastus=(malu&128)!=0; //kas seitsmes bitt on püsti
      malu<<=1;
      nr++;
      if(nr==8){loeJargmine();}
      return vastus;
    }
    public int mitmesBitt(){
      return nr;
    }
  }
```

}

## *Kokkuvõtteks*

Binaarfailidega ümber käimisel on bitioperatsioonid paratamatud. Enamik tulemusi õnnestub leida nii harilike arvutuste kui spetsiaalsete bititehete abil. Osa lõpus näidati, kuidas bittide lugemiseks ja kirjutamiseks võib koostada vooklassid nagu need mitmete muude andmetüüpidega ümber käimiseks juba olemas on.

## *Ülesandeid*

### Bitid

* Teata, kas arvu parempoolne bitt on 1
* Teata arvu kõik bitid
* Küsi kahe inimese vanused, talleta nende
  väärtused ühes int-muutujas. Väljasta väärtused

### Bitimuster

* Trüki ekraanile kasutaja antud arvu bitid.
* Kasutaja tipitud nullide ja ühtede jada salvesta täisarvuna tekstifaili.
  Hiljem loe see arv failist ning väljasta taas ekraanile.
* Ühes tekstifailis on tühikutest ja ristidest koosnev muster (32 rida, 64
  tähte reas). Talleta see muster arvudena biti kaupa teise faili. Hiljem loe
  arvud failist ning taasta muster.

### DNA ahela pakkimine

DNA-ahel koosneb neljast eri tüüpi osast. Tähistame iga desoksüribonukleotiidi järgneva bitikombinatsiooniga.

G - 00
C - 01
A - 10
T - 11

* Talleta ahel CGCACGCTCACTCACG sõnena.
* Trüki tähed tsükli abil ükshaaval, tühikutega eraldatult ekraanile
* Trüki tähed välja neile vastavate koodidena
* Talleta ahel vastavate bittide abil ühes neljabaidises täisarvus

* Loe ahel sellest arvust taas välja ning trüki tulemus.

* Luba ahel sisestada klaviatuurilt (max 256 sümbolit).
  Andmed talletatakse bittidena arvumassiivis. Ahela tegelik pikkus
  salvestatakse massiivi esimeses baidis.
* Loo arvule biti lisamiseks alamprogramm.
* Hoolitse, et programm töötab viisakalt ka vigase sisestuse korral.

## Bitinihutus

* Küsi kasutajalt arv (<256) ning väljasta see kujul, kus vasakpoolseim bitt on tõstetud paremale ning ülejäänud ühe võrra vasakule.
* Lisaks eelmisele küsi kasutajalt, mitme biti võrra tahetakse arvu bitte keerata.
* Võrreldes eelmisega keera kasutaja etteantud arvu võrra terve faili iga baidi bitte.

## *Andmestruktuurid*

Andmestruktuuridest räägitakse enamike programmeerimiskeelte juures. Pole siis põhjust ka Java puhul sellest hoiduda - eriti kui täiesti kõlbulikud vahendid keele juurde loodud on.

Esmapilgul võib nimistu käsitlemine mõttetunagi tunduda, sest Java juures on ju olemas Java Collections Framework, mille puhul igaühel õigus priipärast kasutada LinkedList-tüüpi objekti ning selle kaudu elemente hoida, järjestada ja muidki meelde tulevaid toiminguid ette võtta. Tegemist mõnevõrra samalaadse küsimusega, et kas mul on põhjust teada ruutjuure leidmise algoritmi. Iseenesest on vastav käsklus nii enamikes programmeerimiskeeltes kui muude arvutamist võimaldavate seadmete juures olemas. Samas aga, kui kätte juhtub Felixi-nimeline mehhaaniline arvutusmasin millel küll peal korralikud vahendid liitmiseks-lahutamiseks, kuid ruutjuurest mitte märkigi, võib algoritmi teadmine või tuletusoskus päris ilusti tee kätte näidata. Samuti aitab algoritmi aimamine mõista, milliste väärtuste puhul ülesanne arvutile raskem ja millal kergem lahendada on.

Andmestruktuuridele "käsitsi" lähenemiseks ei pruugi sugugi liialt ebatavalist kohta otsida. Kui näiteks asuda programmi abil XMLi andmepuud uurima või muutma, ei pääse enamasti mööda elemente ringi liikumisest. Kui püüda failisüsteemis toimetada, siis seal on tegemist samuti hierarhiliste andmetega, milles liikumisel tuleb arvestada nii enese asukohta, ülemkatalooge kui alanevaid faile ja katalooge. Kolmandaks sarnaseks näiteks sobiks ehk Swingi puu: üksikute okste ja lehtede abil tuleb kõik küllaltki nullist valmis ehitada. Ning nagu õppejõud Isotamm Tartu Ülikooli kompilaatorite kursuse esimeses loengus ütles: kui andmed on õnnestunud mälus kahendpuusse ajada, siis sageli on programmeerija jaoks pool tööd valmis.

Järgnevalt püüame võimalikult lihtsate näidete varal levinumad andmete mälus paiknemisega seotud võimalused läbi mängida, et hiljem oleks oskust ja julgust omal jõul vajadusel ka keerukamaid struktuure ehitada.

## *Nimistu*

Tõenäoliselt levinuimaks andmestruktuuriks on ahel. Ehk jada, kus igal elemendil on oma väärtus ning siis ka osuti järgmisele isendile ahelas või tühiväärtus tähistamaks ahela lõppu. Siinsetes näidetes on andmetüübiks võetud lihtsuse mõttes int, kuid andmeteks võib iseenesest olla ükskõik milline liht- või struktuurtüüp. C++is kasutatakse malle võimaldamaks kord valmis ehitatud andmestruktuurivahendeid kasutamaks igasuguste tüüpidega. Javas tavatsetakse andmetüübiks panna Object, siis on võimalik sellesse muutujasse paigutada igasugu struktuurtüüpide eksemplare sõltumata nende tegelikust andmetüübist. Siin näidetes aga on piirdutud int-väärtusega.

## Üksik rakk

```
public class Rakk{
  int sisu;
  Rakk jargmine;
}
```

## Seotud rakud

Sarnaselt loodud Rakkude eksemplare võib rahus üksteisega siduda. Siin näites koostatakse kaks Raku eksemplari. Ühele antakse väärtuseks 13, teisele 20. Ning kuna esimese raku muutuja jargmine pannakse osutama teisele rakule, siis avaldise r1.jargmine.sisu väärtuseks on teise raku sisu ehk 20.

```
public class Rakukatse1{
  public static void main(String argumendid[]){
    Rakk r1=new Rakk();
    r1.sisu=13;
    System.out.println(r1.sisu);
    Rakk r2=new Rakk();
    r2.sisu=20;
    r1.jargmine=r2;
    System.out.println(r1.jargmine.sisu);
  }
}
```

## Pikem ahel

Rakke võib nõnda ka rohkem üksteisele sabasse lükkida. Keele poolt piirangut ei ole, vaid suurtel arvudel võib lihtsalt mälumahuga probleeme tekkida. Siinne r1.jargmine.jargmine.sisu ja sealt seest paistev r3-e väärtus lihtsalt tutvustab võimalikku jadastamist.

```
public class Rakukatse1{
  public static void main(String argumendid[]){
    Rakk r1=new Rakk();
    r1.sisu=13;
    System.out.println(r1.sisu);
```

```
    Rakk r2=new Rakk();
    r2.sisu=20;
    r1.jargmine=r2;
    System.out.println(r1.jargmine.sisu);
    Rakk r3=new Rakk();
    r3.sisu=7;
    r2.jargmine=r3;
    System.out.println(r1.jargmine.jargmine.sisu);
  }
}
```

## Vähem muutujaid

Kui ahelas on lülisid rohkem, siis pole mõistlik igale elemendile omaette muutujat luua. Piisab, kui osuteid mööda on võimalik sobivasse kohta liikuda. Nõnda võib muutuja uus abiga luua pika rivi ilma, et oleks rohkem muutujaid juurde tarvis. Alguse tarvis on muutuja ikka vajalik, sest sealtkaudu pääseb viiteid pidi ahela sees olevatele elementidele ligi.

```
public class Rakukatse2{
  public static void main(String argumendid[]){
    Rakk algus=new Rakk();
    algus.sisu=13;
    Rakk uus=new Rakk();
    uus.sisu=20;
    algus.jargmine=uus;
    uus.jargmine=new Rakk();
    uus=uus.jargmine;
    uus.sisu=7;
    System.out.println(algus.sisu);
    System.out.println(algus.jargmine.sisu);
    System.out.println(algus.jargmine.jargmine.sisu);
  }
}
```

## Ahela läbimine tsükliga

Võrreldes eelmisega ongi siin sama muutujate arvu juures tsükli abil hulga pikem ahel kokku pandud. Iga ringi juures lihtsalt luuakse uus eksemplar, antakse sellele sisu ning ollakse taas valmis uue elemendi loomiseks.

```
public class Rakukatse3{
  public static void main(String argumendid[]){
    Rakk algus=new Rakk();
    algus.sisu=10;
    Rakk uus=algus;
    for(int i=20; i<=100; i=i+10){
      uus.jargmine=new Rakk();
      uus=uus.jargmine;
      uus.sisu=i;
    }
    System.out.println(algus.sisu);
    System.out.println(algus.jargmine.sisu);
    System.out.println(algus.jargmine.jargmine.sisu);
  }
}
```

## Väljatrükk

Mis kord sisse kirjutatud, see ka hea välja lugeda. Olgu siis kontrolli eesmärgil või pärastpoole ka tunduvalt asjalikumate ettevõtmiste tarvis. Jada lõppu saab null-tunnuse järgi kontrollida, sest õnnelikult on Java keele juures määratud, et väärtustamata isendimuutujad on nullilise väärtusega algväärtustatud. Nõnda, et kui

ahela viimasel rakul on küll andmete juures väärtus, kuid järgmise elemendi tarvis pole midagi juurde pandud, siis võib uskuda, et muutuja jargmine väärtuseks on null.

Jada trükkimisel piisab ühest muutujast juhul, kui piisab vaid ühekordsest läbimisest. Ning kuna peaprogrammis on muutuja "algus" kindlalt paigas, siis pole karta, et ahela algus võiks kogemata käest lipsata.

Näidatud trükkimiskäsklust läheb olukorra kontrollimiseks vaja mitmes järgmiseski näites.

```
public class Rakukatse4{
  public static void trykiJada(Rakk jooksev){
    while(jooksev!=null){
      System.out.println(jooksev.sisu);
      jooksev=jooksev.jargmine;
    }
  }

  public static void main(String argumendid[]){
    Rakk algus=new Rakk();
    algus.sisu=10;
    Rakk uus=algus;
    for(int i=20; i<=100; i=i+10){
      uus.jargmine=new Rakk();
      uus=uus.jargmine;
      uus.sisu=i;
    }
    trykiJada(algus.jargmine.jargmine);
  }
}
```

## Vahelepanek

Võrreldes eelnenud näitega lisatakse siin vahelepaneku võimalus. Ahela puhul on vahelepanek märkimisväärselt tähtis omadus. Kui andmeid hoitakse massiivis ning on vaja midagi lisada, siis üldjuhul ei õnnestu toiming muidu, kui tuleb märgatav osa massiivi andmetest ümber tõsta. Ahela puhul aga piisab vaid uue raku juures viited sobivalt paika sättida. Nii et uue raku järgmine näitaks tegeliku järgmise peale. Ning et vahelepandud rakule eelneva raku edasi-viide näitaks uuele rakule.

```
    Rakk uus=new Rakk();
    uus.sisu=vaartus;
    uus.jargmine=eelmine.jargmine;
    eelmine.jargmine=uus;
```

Näide tervikuna:

```
public class Rakukatse5{
  public static void trykiJada(Rakk jooksev){
    while(jooksev!=null){
      System.out.println(jooksev.sisu);
      jooksev=jooksev.jargmine;
    }
  }

  public static void lisaVahele(Rakk eelmine, int vaartus){
    Rakk uus=new Rakk();
    uus.sisu=vaartus;
    uus.jargmine=eelmine.jargmine;
    eelmine.jargmine=uus;
  }

  public static Rakk looJada(int vahim, int suurim, int vahe){
    Rakk algus=new Rakk();
    algus.sisu=vahim;
    Rakk uus=algus;
    for(int i=vahim+vahe; i<=suurim; i=i+vahe){
      uus.jargmine=new Rakk();
      uus=uus.jargmine;
      uus.sisu=i;
    }
```

```
    return algus;
  }

  public static void main(String argumendid[]){
    Rakk algus=looJada(20, 100, 10);
    lisaVahele(algus.jargmine, 35);
    trykiJada(algus);
  }
}

/*
D:\arhiiv\naited\io\muu>java Rakukatse5
20
30
35
40
50
60
70
80
90
100
*/
```

## Järjestamine

Märgatava osa maailma arvutite jõudlusest pidada võtma mitmesuguste andmete järjestamine ja otsimine. Heaks lihtsaks sorteerimisalgoritmiks peaks olema võimalus elemente üksteise järgi väärtuste järjekorras sobivasse kohta vahele panna. Lisaks tavalisele sobivasse kohta paigutamisele tekib kaks erandlikku olukorda: lisatav on kõigist andmetest kas suurem või väiksem. Kui soovitakse ahela lõppu panna, siis võib seda mõnes mõttes ette kujutada kui elemendi lisamist viimase elemendi ja tema poolt viidatava tühiväärtuse vahele. Ette lisamine aga mõnevõrra keerulisem: paigast nihkub ka muidu muutumatuna tunduv ahela algus. Siinses näites ei pruugi sellest midagi suuremat hullu olla. Aga kohtades, kus ahela algus on kirjas mitmes paigas, et oleks mugav andmetele ligi pääseda, võib alguse asukoha muutmine muresid põhjustada. Et algus võib lisamisel paigast nihkuda, selleks siis jäetakse iga käigu juures meelde ka uus algus pärast vastavat sammu. Toiming, mida eelnenud näidete juures tarvis polnud.

```
    algus=lisaVaartus(algus, 5);
```

Kuna vahele lisamisel on vaja ligi pääseda nii uuest sõlmest ette- kui tahapoole jäävatele sõlmedele, siis sobiva koha otsimisel tuleb meeles pidada nii jooksev element, mille väärtust võrreldakse vahelepandava väärtusega kui ka jooksvale elemendile eelneva elemendi osuti, et oleks vahelepanekul võimalus kõik osutid korralikult kätte saada ja paika sättida.

```
   while(jooksev!=null && vaartus>jooksev.sisu){
     eelmine=jooksev;
     jooksev=jooksev.jargmine;
   }
```

Ning vahelepaigutamistega näide kogu  pikkuses.

```
public class Rakukatse6{
  public static void trykiJada(Rakk jooksev){
    while(jooksev!=null){
      System.out.println(jooksev.sisu);
      jooksev=jooksev.jargmine;
```

```
    }
  }

  public static void lisaVahele(Rakk eelmine, int vaartus){
    Rakk uus=new Rakk();
    uus.sisu=vaartus;
    uus.jargmine=eelmine.jargmine;
    eelmine.jargmine=uus;
  }

  public static Rakk lisaEtte(Rakk algus, int vaartus){
    Rakk uusAlgus=new Rakk();
    uusAlgus.sisu=vaartus;
    uusAlgus.jargmine=algus;
    return uusAlgus;
  }

  public static Rakk lisaVaartus(Rakk algus, int vaartus){
   Rakk jooksev=algus;
   if(vaartus<jooksev.sisu){
     return lisaEtte(jooksev, vaartus);
   }
   Rakk eelmine=jooksev;
   jooksev=jooksev.jargmine;
   while(jooksev!=null && vaartus>jooksev.sisu){
     eelmine=jooksev;
     jooksev=jooksev.jargmine;
   }
   lisaVahele(eelmine, vaartus);
   return algus;
  }

  public static Rakk looJada(int vahim, int suurim, int vahe){
    Rakk algus=new Rakk();
    algus.sisu=vahim;
    Rakk uus=algus;
    for(int i=vahim+vahe; i<=suurim; i=i+vahe){
      uus.jargmine=new Rakk();
      uus=uus.jargmine;
      uus.sisu=i;
    }
    return algus;
  }

  public static void main(String argumendid[]){
    Rakk algus=looJada(20, 100, 10);
    algus=lisaVaartus(algus, 45);
    algus=lisaVaartus(algus, 48);
    algus=lisaVaartus(algus, 5);
    algus=lisaVaartus(algus, 500);
    trykiJada(algus);
  }
}
/*
D:\arhiiv\naited\io\muu>java Rakukatse6
5
20
30
40
45
48
50
60
70
80
90
100
500

*/
```

Jadasse elementide lisamiseks ei pea sugugi tervet jada valmis kirjutama. Kui tahetakse olemasolevaid väärtusi jadasse ja ilusti ritta paigutada, siis piisab alustuseks tühjast jadast, ehk lihtsalt Rakutüüpi muutujast millel väärtuseks null. Nõnda, et jada algus ja lõpp on samas kohas ning sees polegi midagi kahtlast. Vaid väärtuse lisamisel peab arvestama, et alguseks võib olla ka tühiväärtus ning sellisel juhul tuleb täiendav

kontroll lisada. Et kui jada juhtub tühi olema, siis lisatav element saab jada ainukeseks elemendiks ja ühtlasi ka jada alguseks.

```
  public static Rakk lisaVaartus(Rakk algus, int vaartus){
   Rakk jooksev=algus;
   if(algus==null){
     Rakk uus=new Rakk();
     uus.sisu=vaartus;
     return uus;
   }
...
}
```

Kui nüüd tühja jadasse asutakse ükshaaval elemente lisama ning iga lisamise puhul hoolitsetakse, et elemendid ikka soovitud järjekorda jääksid, siis tulemuseks ongi soovitult sorditud jada.

```
public class Rakukatse7{
  public static void trykiJada(Rakk jooksev){
    while(jooksev!=null){
      System.out.println(jooksev.sisu);
      jooksev=jooksev.jargmine;
    }
  }

  public static void lisaVahele(Rakk eelmine, int vaartus){
    Rakk uus=new Rakk();
    uus.sisu=vaartus;
    uus.jargmine=eelmine.jargmine;
    eelmine.jargmine=uus;
  }

  public static Rakk lisaEtte(Rakk algus, int vaartus){
    Rakk uusAlgus=new Rakk();
    uusAlgus.sisu=vaartus;
    uusAlgus.jargmine=algus;
    return uusAlgus;
  }

  public static Rakk lisaVaartus(Rakk algus, int vaartus){
   Rakk jooksev=algus;
   if(algus==null){
     Rakk uus=new Rakk();
     uus.sisu=vaartus;
     return uus;
   }
   if(vaartus<jooksev.sisu){
     return lisaEtte(jooksev, vaartus);
   }
   Rakk eelmine=jooksev;
   jooksev=jooksev.jargmine;
   while(jooksev!=null && vaartus>jooksev.sisu){
     eelmine=jooksev;
     jooksev=jooksev.jargmine;
   }
   lisaVahele(eelmine, vaartus);
   return algus;
  }

  public static void main(String argumendid[]){
     int[] arvud={4,2,56,2,3,6,4};
     Rakk algus=null;
     for(int i=0; i<arvud.length; i++){
       algus=lisaVaartus(algus, arvud[i]);
     }
     trykiJada(algus);
  }
}

/*
D:\arhiiv\naited\io\muu>java Rakukatse7
2
2
3
4
4
6
```

```
56
*/
```

## Ülesandeid

## Pinu

* Kasutajalt küsitakse kaks arvu ning väljastatakse need vastupidises järjekorras.
* Iga kord, kui kasutaja kirjutab positiivse arvu, lisatakse see pinu lõppu (peale). Kui kasutaja kirjutab nulli, siis väljastatakse pinu pealmine element ning eemaldatakse see pinust.
* Realiseeri sarnane pinu nii massiivi kui viitadega ühendatud elementide abil.

## Järjekord

* Küsi kasutajalt kaks arvu ning väljasta neist esimene.
* Iga käsuga saab kasutaja määrata, kas ta soovib arvu lisada või küsida. Sisestatud arv lisatakse lõppu, eemaldamise puhul väljastatakse arv ja eemaldatakse algusest. Sisestatud arvude hulk võib olla piiratud massiivi maksimaalse pikkusega.
* Realiseeri järjekord nii massiivi kui viitadega ühendatud elementide abil. Massiivi puhul võib samaaegne järjekorras olevate elementide arv olla piiratud massiivi pikkusega, pideval lisamisel ja eemaldamisel aga piiranguid ei seata.

## Osutiring

* Loo andmetüüp, mis koosneks täisarvust ning osutist samatüübilisele rakule. Loo kaks eksemplari ning pane nad üksteisele näitama. Testi tulemust.
* Loo alamprogramm, mis kontrolliks, kas etteantud osutiahel moodustab silmuse ning viimasel juhul teata, mitmest elemendist ringahel koosneb.
* Loo alamprogramm ringikujulisse osutiahelasse elemendi lisamiseks ning teine sellisest ahelast elemendi eemaldamiseks. Nende kahe abil ühenda kaks ringikujulist osutiahelat.

## *Kahendpuu*

Suuremate andmemahtude korral lähevad ahelad pikaks ning soovitud kohta jõudmine võtab palju samme. Liikumist konkreetse sõlmeni aitavad lühendada otseteed ehk viited sõlmedeni ahela keskosas.

Levinud andmestruktuuriks on kahendpuu: iga sõlme juures on andmeplokk ning osutid kahele sõlmele, mida tinglikult võib nimetada vasakuks ja paremaks haruks.

## Üksik sõlm

```
public class Solm{
  int sisu;
  Solm vasak;
  Solm parem;
}
```

Nõnda andmeid paigutades võib õnnestuda tulemus, kus suhteliselt lühikese teega on võimalik jõuda iga väärtuseni. Kaks astmes 10 on 1024, astmes 20 juba üle miljoni. Siit paistab, et kui andmepuu on tasakaalustatud, st., et puu üksikud harud on ligikaudu ühepikkused, siis ka miljoni väärtuse salvestamisel ei pruugi iga väärtuse leidmiseks kuluda oluliselt üle kahekümne sammu. Lineaarse ahela puhul tulnuks aga läbi käia keskeltläbi pool miljonit sõlme.

Osutid, millele pole väärtust omistatud, sisaldavad tühiväärtust null. Nii on kindlalt võimalik määrata, kus miski haru lõpeb. Puu võib koosneda ka vaid ühest sõlmest.

```
public class Solmetest1{
  public static void main(String[] argumendid){
    Solm s1=new Solm();
    s1.sisu=7;
    System.out.println(s1.sisu+" "+s1.vasak+" "+s1.parem);
  }
}
```

## Kahe haruga puu

Kui sõlmesid loodud rohkem, siis saab nad omavahel ühendada. Praeguses näites esialgu igale sõlmele oma väärtus ning siis paigutatakse esimese sõlme vasaku haru külge teine sõlm ning parema haru külge kolmas sõlm. Avaldis s1.parem.sisu annab nõnda kolmanda sõlme väärtuse.

```
public class Solmetest2{
  public static void main(String[] argumendid){
    Solm s1=new Solm();
    Solm s2=new Solm();
    Solm s3=new Solm();
    s1.sisu=4;
    s2.sisu=3;
    s3.sisu=8;
    s1.vasak=s2;
    s1.parem=s3;
    System.out.println(s1.sisu+" "+s1.vasak.sisu+" "+s1.parem.sisu);
  }
}
```

Sõlmede ahela saab ka pikema ehitada. Ning enamasti ongi andmepuus märgatavalt enam kui paar-kolm väärtust. Nagu näha, luuakse siin algul sõlmed, määratakse neile väärtused ning siis asutakse sõlmesid ühendama. Pärast on võimalik asukohtade järgi väärtused välja küsida.

```
public class Solmetest3{
```

```
  public static void main(String[] argumendid){
    Solm s1=new Solm();
    Solm s2=new Solm();
    Solm s3=new Solm();
    Solm s4=new Solm();
    Solm s5=new Solm();
    Solm s6=new Solm();
    s1.sisu=4;
    s2.sisu=3;
    s3.sisu=8;
    s4.sisu=5;
    s5.sisu=7;
    s6.sisu=10;
    s1.vasak=s2;
    s1.parem=s3;
    s2.vasak=s4;
    s3.vasak=s5;
    s3.parem=s6;
    System.out.println(
       s1.sisu+" "+s1.vasak.sisu+" "+s1.vasak.vasak.sisu+" "+
       s1.parem.sisu+" "+s1.parem.vasak.sisu+" "+s1.parem.parem.sisu);
  }
}
```

## Rekursioon

Kui sõlmesid on rohkem, siis peab nendega toimetamiseks miski üldisema mooduse leidma. Ahela elementide läbimiseks sobib tsükkel, puu puhul on aga tsükliga vähem ette võtta. Iseenesest on võimalik puu läbimiseks koostada abiahel kohtadest, millises puu juureni ulatuvas harus parajasti ollakse ning nõnda meeles pidades püüda kogu puu järjestikku läbi käia. Teiseks võimaluseks on rekursioon, ehk alamprogrammid oma väljakutsete kaudu peavad meeles, millises harus parajasti ollakse. Ning et korraga võib samast alamprogrammist töös olla rohkem kui üks eksemplar. Toimingud liiguvad kõige viimati käivitatud eksemplaris. Ülejäänud on ootel, kuniks välja kutsutud eksemplar oma tööga ühele poole saab.

Siin näites ongi väjatrükk rekursiooni hooleks jäetud. Alamprogrammile antakse ette sõlm ning alamprogrammi ülesandeks on trükkida välja selle sõlme ning kõigi alanevate sõlmede sisu. Enese sees olevate sõlmede väljatrükiks on lubatud kasutada alamprogrammi ennast. Rekursiooni baasiks ehk olukorraks, kus enam uuesti samasse alamprogrammi ei siseneta on juhtum, kui trükitav sõlm puudub, ehk muutuja väärtuseks on null. Nõnda siis juure ning kahe alaneva elemendiga puus kutsutakse alamprogrammi välja päris mitu korda. Kõigepealt juure jaoks. Kuna juure muutuja pole null, siis liigutakse edasi. Trükitakse juurelemendi andmeploki väärtus ning edasi kutsutakse sama funktsioon välja vasaku haru jaoks. Juure trükkimist alustanud funktsiooni eksemplar jääb vasaku haru trükkimise ajaks ootele. Vasaku haru trükkimise eksemplar väljastab kõigepealt andmesegmendi väärtuse ning siis proovib vasakust harust omakorda vasakut haru trükkida. Et sealne osuti on tühi, siis kolmandast trükkimisfunktsiooni eksemplarist jõutakse kiiresti tagasi ning proovitakse vasaku haru paremat haru välja trükkida. Et ka sealne pool on tühi, siis väljutakse juure vasakut haru trükkivast funktsioonist ning alustatakse parema poole trükkimist. Ning parema poole puhul tulevad samasugused kolm väljakutset. Kõigepealt sõlme enese oma ning siis kaks lühikest käivitust tühjade osutite tarbeks. Samasugust juhtumist võimegi näha s3-e ehk sõlme väärtusega 8 väljatrükil. Hilisem kogu puu väljatrükk paigutab kõik väärtused ekraanile.

```
public class Solmetest4{
  //trükitakse väärtused vastavast sõlmest alates
  public static void trykiSolm(Solm s){
    if(s==null){return;}
    System.out.print(s.sisu+" ");
```

```
    trykiSolm(s.vasak);
    trykiSolm(s.parem);
  }

  public static void main(String[] argumendid){
    Solm s1=new Solm();
    Solm s2=new Solm();
    Solm s3=new Solm();
    Solm s4=new Solm();
    Solm s5=new Solm();
    Solm s6=new Solm();
    s1.sisu=4;
    s2.sisu=3;
    s3.sisu=8;
    s4.sisu=5;
    s5.sisu=7;
    s6.sisu=10;
    s1.vasak=s2;
    s1.parem=s3;
    s2.vasak=s4;
    s3.vasak=s5;
    s3.parem=s6;
    trykiSolm(s3);
    System.out.println();
    trykiSolm(s1);
  }
}
```

Sõlmede paiknemine ja väljatrüki tulemused.

```
/*
     4
    / \
   3   8
  /   / \
 5   7  10


C:\jaagup>java Solmetest4
8 7 10
4 3 5 8 7 10

*/
```

## Järjestamine

Kahendpuud rakendatakse sageli väärtuste järjestamisel. Eeliseks ahelate ees just võimalus, et loetelu keskel paiknevate liikmeteni jõuab tunduvalt vähesema sammude arvuga. Kui andmete puusse paigutades hoolitseda, et elemendist ühele poole jääks alati temast väiksem ning teisele poole temast suurem väärtus, siis lõpuks ongi võimalik kätte saada nii järjestatud väärtused kui vajadusel suhteliselt vähese sammude arvuga kindlaks teha, kas soovitud väärtus puus leidub. Siinses näites korduvaid väärtusi puusse ei lisata. Suuremad väärtused paigutatakse paremale ning väiksemad vasakule.

Alamprogrammile antakse ette sõlm, kuhu juurde panna ning uus väärtus mida panna. Alamprogrammil on õigus iseennast teiste parameetritega välja kutsuda. Nõnda võib vajadusel otsida sisule sobivat kohta algsele sõlmele alanejate hulgas.

```
 public static void lisa(Solm s, int sisu){
```

Kui vastav väärtus juba andmepuus olemas on, siis teist samasugust ei lisata, vaid toiming katkestatakse.

```
    if(s.sisu==sisu){return;}
```

Kui uus väärtus on jooksva sõlme väärtusest suurem, siis püütakse uus väärtus paigutada paremale.

```
    if(sisu>s.sisu){
```

Kui jooksva elemendi parempoolne haru on tühi

```
      if(s.parem==null){
```

Siis luuaks sinna lihtsalt uus sõlm ning pannakse uus väärtus selle sisse.

```
        s.parem=new Solm();
        s.parem.sisu=sisu;
      } else {
```

Kui aga element juhtus paremal juba olemas olema, siis käivitatakse sama lisamise alamprogramm juba selle paremal pool asuva elemendi suhtes. Hakatakse siis võrdlema, kas vastav väärtus on tolle parempoolse elemendi oma ning võib puusse lisama jätta või tuleb omakorda järgnevat paika otsima asuda.

```
        lisa(s.parem, sisu);
      }
    }
```

Ning vasaku poolega sama lugu.

```
    if(sisu<s.sisu){
      if(s.vasak==null){
        s.vasak=new Solm();
        s.vasak.sisu=sisu;
      } else {
        lisa(s.vasak, sisu);
      }
    }
  }
```

Ning näitrakendus tervikuna.

```
public class Solmetest5{
  public static void tryki(Solm s){
    if(s==null){return;}
    tryki(s.vasak);
    System.out.print(s.sisu+" ");
    tryki(s.parem);
  }

  public static void lisa(Solm s, int sisu){
    if(s.sisu==sisu){return;}
    if(sisu>s.sisu){
      if(s.parem==null){
        s.parem=new Solm();
        s.parem.sisu=sisu;
      } else {
        lisa(s.parem, sisu);
      }
    }
    if(sisu<s.sisu){
      if(s.vasak==null){
        s.vasak=new Solm();
        s.vasak.sisu=sisu;
      } else {
        lisa(s.vasak, sisu);
      }
    }
  }

  public static void main(String[] argumendid){
    Solm s=new Solm();
    s.sisu=3;
```

```
    lisa(s, 2);
    lisa(s, 6);
//    tryki(s);
    int[] arvud={4, 3, 3, 1, 9, 8, 0};
    for(int nr=0; nr<arvud.length; nr++){
      lisa(s, arvud[nr]);
    }
    tryki(s);
  }
}
/*
D:\arhiiv\naited\io\muu>java Solmetest5
0 1 2 3 4 6 8 9
*/
```

## Otsimine

Kui andmed on kindla korra järgi puusse paigutatud, siis saab selle sama korra järgi neid ka otsida. Või kui andmed on lihtsalt puus, siis saab tulemuse kätte kogu puu läbi vaadates. Siin vaadatakse otsimisel läbi jooksev element ning mõlema poole alampuud ning kui kusagilt otsitav leiti, siis väljastatakse selle kohta "jah".

```
  public static boolean otsi(Solm s, int sisu){
    if(s==null)return false;
    if(s.sisu==sisu)return true;
    if(otsi(s.vasak, sisu))return true;
    if(otsi(s.parem, sisu))return true;
    return false;
  }
```

Otsimist annaks optimeerida, eeldades, et suuremad väärtused asuvad paremal ja väiksemad vasakul. Näiliselt tunduks tegemist olema ainult paari väikese võrdluse lisamisega, kuid suuremate andmemahtude korral võib kiiruse võit olla tohutu. Miljoni väärtuse korral kuluks tasakaalustatud puus sobiva väärtuse leidmiseni paarkümmend sammu, täisläbivaatluse korral aga keskeltläbi pool miljonit.

Järgnevalt siiski otsingunäide täisläbivaatluse kohta. Otsingut kiirendavad võrdlused võiks lugeja mõttes siia juurde lisada.

```
public class Solmetest6{
  public static void tryki(Solm s){
    if(s==null){return;}
    tryki(s.vasak);
    System.out.print(s.sisu+" ");
    tryki(s.parem);
  }

  public static void lisa(Solm s, int sisu){
    if(s.sisu==sisu){return;}
    if(sisu>s.sisu){
      if(s.parem==null){
        s.parem=new Solm();
        s.parem.sisu=sisu;
      } else {
        lisa(s.parem, sisu);
      }
    }
    if(sisu<s.sisu){
      if(s.vasak==null){
        s.vasak=new Solm();
        s.vasak.sisu=sisu;
      } else {
        lisa(s.vasak, sisu);
      }
    }
  }

  public static boolean otsi(Solm s, int sisu){
```

```
    if(s==null)return false;
    if(s.sisu==sisu)return true;
    if(otsi(s.vasak, sisu))return true;
    if(otsi(s.parem, sisu))return true;
    return false;
  }

  public static void main(String[] argumendid){
    Solm s=new Solm();
    s.sisu=3;
    int[] arvud={4, 3, 3, 1, 9, 8, 0};
    for(int nr=0; nr<arvud.length; nr++){
      lisa(s, arvud[nr]);
    }
    tryki(s);
    if(otsi(s, 7)){
      System.out.println("Leiti");
    } else {
      System.out.println("Ei leitud");
    }
  }
}
```

## *Kokkuvõtteks*

Kahendpuu võimaldab andmeid mällu järjestatult paigutada ning soovitud väärtusi kiiremini lugeda kui ahelas see võimalik oleks. Puu läbimiseks või andmete lisamiseks kasutatakse rekursiivseid algoritme.

## *Ülesandeid*

## Andmepuu

* Loo andmetüüp, mis koosneks kuni 30 tähe pikkusest tekstist ning viidast samatüübilisele rakule. Loo sellisest tüübist eksemplar, väärtusta ning trüki tulemus.
* Koosta selle abil lõik ülikooli struktuurist, kusjuures iga raku tekst on vastava üksuse nimi ning viit näitab tase kõrgemal olevale üksusele. Samaaegselt hoia viitu kõigile rakkudele massiivis ning trüki iga raku kohta välja, mis ta nimi ning millistesse kõrgematesse üksustesse ta kuulub.
* Lisaks eelmisele lisa võimalused andmete ja seoste lisamiseks ja muutmiseks klaviatuuri abil.

## Trepitud kahendpuu

* Loo andmetüüp, mis koosneks täisarvust ning osutist kahele samatüübilisele rakule. Loo kolm eksemplari, nii et kaks järgmist oleksid esimese küljes. Trüki tulemused esimese kaudu.
* Loo alamprogramm, millele antakse ette osuti vastavat tüüpi rakule. Kui osuti väärtus pole null, siis trüki väärtus ning käivita sama alamprogramm mõlema alaneva osuti puhul.

* Hoolitse trükkimisel, et treppimise abil oleks näha, milline väärtus millise alla kuulub. Väljasta puus leiduvate väärtuste summa.

## Morse

* Koosta pliiatsi ja paberi abil morsemärkidest kahendpuu.
* Koosta sarnane puu arvutisse omaloodud andmetüübi ja osutite abil.
  Väljasta oma puu abil jada ... --- ... tulemus.
* Loe oma andmepuust taas välja massiivi igale tähele vastava jada tarvis. Koosta arhiveerimisprogramm tekstifaili morsestamiseks ning taastekstistamiseks.

```
a  .-                 w  .--
b  -...               ä  .-.-
c  -.-.               öõ ---.
d  -..                ü  ..--
e  .                  x  -..-
f  ..-.               y  -.--
g  --.
h  ....               1  .----
i  ..                 2  ..---
j  .---               3  ...--
k  -.-                4  ....-
l  .-..               5  .....
m  --                 6  -....
n  -.                 7  --...
o  ---                8  ---..
p  .--.               9  ----.
q  --.-               0  -----
r  .-.
s  ...                .  .-.-.-
z  --..               ,  --..--
t  -                  ?  ..--..
u  ..-
v  ...-
```