Datastrukturer & Algoritmer, Datalogi C
Henning Christiansen 13/11-2001

Opgaver til forelæsningen 12/11-2002

Bogens opgave 20.5 og 20.6 side 710 og nedenstående:

Opgave
Det drejer sig om at undersøge hvor godt forskellige hashfunktioner fungerer, og der leveres en række filer (se kursets hjemmeside) til hjælp. Det er let modificerede versioner af bogens eksempelfiler plus et par ekstra. Placer alle filerne i samme arbejdskatalog og oversæt dem i følgende rækkefølge:

```
javac Iterator.java
javac Collection.java
javac AbstractCollection.java
javac Set.java
javac HashSet.java
javac ReadFile.java
```

ReadFile giver værktøj, så man nemt kan læse ord ind fra en fil og bruge dem til eksperimenter med hashing. HashSet er udvidet med en metode statistics(), som siger noget om hashtabellens størrelse og hvor mange kollisioner der har været. Vi kan f.eks. bruge den som i følgende testprogram (på filen HashTest):

```
import java.io.*;
public final class HashTest
{ private static HashSet h = new HashSet();
 public static void main( String [ ] args )  throws IOException
   {  ReadFile r = new ReadFile(args[0]);
     String s = r. readWord();
     while(s!="ENDOFFILE")
     { if(!h.contains(s)) h.add(s); s = r. readWord();};
    h.statistics();
   }}
```

Den kan køres således, hvor "ugly" er en tekstfil:

```
javac HashTest.java
java HashTest ugly
```

Og der genereres følgende udskrift

```
Table Size = 3527
Used = 904
Current fill factor = 0.25630847
No. of rehashings = 5
```

Collision chains

length	no. of times
0	5017
1	834
2	380
3	77
4	33
5	19
6	4
7	2
8	2
11	2

De første oplysninger er selvforklarende. “Collision chains” angiver hvor mange kollisioner, der har været. “0” angiver hvor hashing har ramt en tom celle med det samme (det var her 5017 gange), “1” hvor der var en kollision, og så fandtes en celle i næste forsøg (837 gange), “2” at der skulle to forsøg til at fine en ledig celle og så fremdeles.

I eksempelprogrammet HashTest benyttedes den hashCode()-metode som Java har inbygget for String. Denne opgave går ud på at eksperimentere med alternative hashCode()-metoder, og da vi ikke kan omdefinere den for strenge, laver vi en klasse af objekter, som indeholder tekststrenge. Det er gjort i følgende program HashTest1:

```
import java.io.*;

public final class HashTest1
{ private static HashSet h = new HashSet();

  static class WeirdObject {
    public WeirdObject(String ss) {s=ss;};
    private String s;

 public  int hashCode()
   { return (int)s.charAt(0);};

   public boolean equals(Object o)
      {return s.equals( ((WeirdObject)o).s);};    };

 public static void main( String [ ] args )  throws IOException
  {  ReadFile r = new ReadFile(args[0]);
     String s = r. readWord();
    WeirdObject w = new WeirdObject(s);
    while(s!="ENDOFFILE")
    { w = new WeirdObject(s);
```

```
    if(!h.contains(w)) h.add(w);

      s = r. readWord();};
    h.statistics();
  }}
```

Som det fremgår er der her benyttet en meget dum hashfunktion, nemlig en som blot tager første tegn i strengen. der afspejler sig i følgende udskrift:

```
Table Size = 3527
Used = 904
Current fill factor = 0.25630847
No. of rehashings = 5

Collision chains
length      no. of times
0           139
1           77
2           46
3           116
4           92
5           216
....
150           2
151           2
152           2
154           2
```

Din opgave går ud på at eksperimentere med forskellige hashfuntioner og sammenligne med Java indbyggede for String. Prøv de to på side 686 i bogen, og se hvilken ad de tre, som fungerer bedst (NB: du bliver nødt til at udelade "%tableSize" for at det passer med standarden for hashCode()-metoden). Prøv at se om du kan konstruere en hashCode() som fungerer endnu bedre for den givne eksempelfil. Prøv evt. på andre tekstfiler. Lad dig f.eks. inspirere af opgave 20.15 side 711.

NB: Eksempelfilerne er kun moderat gennemtestede, og ikke forsynet med de bedste fejlmeddelelser.