**Technische Universität Berlin** **SS 2013**
**Fakultät II, Institut für Mathematik**
Sekretariat MA 5–2, Frau Kiefer

Prof. Dr. Martin Skutella
Torsten Gellert
Martin Groß
Dr. Max Klimm

Aylin Acikel, Katharina Bütow, Christian Döblin,
Alexander Hopp, Daniel Kuske, Olivia Röhrig, Robert Rudow
Daniel Schmand, Hendrik Schrezenmaier, Mona Setje-Eilers
Judith Simon, Sebastian Spies, Fabian Wegscheider, Jan Zur

# Computerorientierte Mathematik II

## Huffman-Codes

## 3. Programmieraufgabe

Abnahme: spätestens am 02./03.05.2013 (je nach Gruppennummer).

**Ungerade Gruppennummern geben spätestens am Donnerstag,**
**Gerade Gruppennummern geben spätestens am Freitag ab.**

Diese Aufgabe ist **anspruchsvoll** und bietet für alle drei Gruppenmitglieder parallele Arbeit. Fangt unbedingt so früh wie möglich mit der Einteilung an!

In dieser Programmieraufgabe sollt ihr ein Programmpaket zum Komprimieren und Dekomprimieren von beliebigen Dateien schreiben, das auf dem Huffman-Algorithmus basiert. Dazu sollen die Quelldateien jeweils byteweise (1 Byte = 8 Bits) eingelesen werden. Die eingelesenen 8 Bits werden dann als Zeichen im Sinne des Huffman-Algorithmus interpretiert und die Datei entsprechend komprimiert.

(a) Legt ein neues Projekt `programs3` an, kopiert die Pakete `binTree` und `priorityQueue` aus der Programmieraufgabe 2 hinein. Ladet euch die Vorgaben zum package `BitIO` mit den Klassen `BitInputFile` und `BitoutputFile` von der Homepage herunter und legt diese ebenfalls als eigenes Paket ab.

(b) **Klasse für Huffman-Bäume.** Erstellt zunächst eine Klasse `HuffmanToken`, die ein Zeichen und seine Häufigkeit modelliert. Auch wenn nur $2^8 = 256$ verschiedene Zeichen auftreten können, bietet es sich an, die Zeichen als `int` zu modellieren, da in Java ein Byte Werte von -128 bis 127 annimmt, was für unsere Zwecke ungünstig ist.

Implementiert nun eine Klasse `HuffmanTree`, die von `BinTree<HuffmanToken>` abgeleitet wird. Da im Huffman-Algorithmus zwei Huffman-Bäume anhand ihrer Häufigkeiten in den Wurzelknoten verglichen werden müssen, soll die Klasse außerdem das Interface `Comparable<HuffmanTree>` implementieren. Gegebenenfalls müsst ihr Anpassungen an eurer `BinTree`-Implementation vornehmen, z.B. indem ihr zusätzliche get- und set-Methoden implementiert. Es sei euch auch ausnahmsweise gestattet, die innere Klasse `BinTreeNode` public zu machen.

(b) **Kompressionsprogramm.** Schreibt ein Programm `Compress`, welches eine beliebige Datei einliest, einen Huffman-Baum dazu erzeugt und die Datei mit dessen Hilfe komprimiert in eine neue Datei schreibt. Der Huffman-Baum wird mit dem in der Vorlesung behandelten Huffman-Algorithmus erstellt und benutzt die in Teil (a) implementierte Klasse `HuffmanTree` sowie eure `PriorityQueue` aus der letzten Programmieraufgabe.

Die zu kodierenden Zeichen sollen die einzelnen Bytes der Datei sein. Zum effizienten byteweisen Einlesen einer Datei eignet sich der `BufferedInputStream`, der seinerseits einen `FileInputStream` erwartet.

Zum Schreiben der Zieldatei braucht ihr die Klasse `BitOutputFile` aus dem Package `BitIo` von der Homepage, die einzelne Bits schreiben kann (siehe JavaDoc). Die Klasse kann sowohl Bytes schreiben als auch einzelne Bits, zwischen den beiden Modi kann beliebig umgeschaltet werden.

Die Ausgabedateien bestehen aus zwei Teilen. Zu Beginn steht die Codetabelle, anschließend die kodierten einzelnen Bytes der Eingabedatei als Bitfolgen. Die ausgegebenen Dateien sollen folgendes Format haben:

- Zu Beginn steht ein *Byte*. Dieses stellt den Wert des höchsten Zeichens dar, welches in der zu kodierenden Datei vorkam. Dieses Zeichen benötigt ihr um zu wissen, wann die Codetabelle aufhöhrt und wann die codierte Datei anfängt
- Es folgt die Codetabelle. Sie besteht aus Paaren von genau einem *Byte* und einigen *Bits*. Das Byte ist ein eingelesenes Zeichen, die Bits stellen die für die Kodierung des Zeichens benutzte Bitfolge dar. Die gelesenen Zeichen sind sortiert, sodass das Zeichen mit dem höchsten Wert als letztes kommt. Es werden nur Zeichen in die Tabelle eingetragen, die auch tatsächlich in der Eingabedatei vorkommen.
- Sobald das höchste vorkommende Zeichen mitsamt seiner Kodierung geschrieben worden ist, endet die Tabelle. Es folgt die Bitdarstellung der kodierten Datei.

Gebt am Ende die Kompressionsrate aus, also wie groß die komprimierte Datei im Verhältnis zur Originaldatei ist. Es reicht hier die Angabe der *netto*-Rate, also ohne die Länge der Codetabelle.

(c) **Dekompressionsprogramm.** Schreibt ein Programm `Decompress`, das eine von `Compress` geschriebene Datei liest und in die originale Datei zurückwandelt, die aber nicht überschrieben werden soll. Es konstruiert aus der gespeicherten Codetabelle den Huffman-Baum und dekodiert damit die Bitfolgen. Dazu müssen die Bitfolgen eingelesen werden und der Baum manuell wieder aufgebaut werden. Die Bitfolgen lest ihr ein mit Hilfe von `BitIo.BitInputFile`. Die Ausgabe in die dekomprimierte Datei – wieder byteweise – mit Hilfe der Klassen `FileOutputStream` und `BufferedOutputStream`.

Anschließend kann der Text dekomprimiert werden, dabei wird ausgehend von der Wurzel der Baum entsprechend den gelesenen Bits durchsucht, bis ein Blatt erreicht worden ist. Dann wurde ein Byte gelesen und es kann an der Wurzel fortgefahren werden.

**Vorgaben:** Was die Gestaltung Eures Programms angeht, seid ihr recht frei, es genügt jedoch ein simples Kommandozeilenprogramm (so wie z. B. die beiden LINUX-Kommandos `zip` bzw. `unzip`). Haltet euch aber in jedem Fall an folgende (sinnvolle) Vorgaben:

- Ihr dürft keine Dateien komplett in den Speicher lesen. Lest deshalb zum Komprimieren die Originaldatei zweimal ein, d. h. einmal zum Zählen der Häufigkeiten und einmal zum eigentlichen Komprimieren.
- Dateien, die nur aus der Wiederholung eines und desselben Bytes bestehen, müssen ebenfalls komprimiert werden können. Der Codebaum hat in diesem Fall eine besondere Gestalt, die ihr gegebenenfalls gesondert behadeln müsst.
- Werden die beiden Programme ohne Argumente aufgerufen, wird eine aussagekräftige *Usage* ausgegeben. Der Aufruf der Programme soll wie folgt lauten:
  `java Compress dateiname [zieldatei]`
  `java Decompress dateiname [zieldatei].`
  Wird keine Zieldatei angegeben, wird die Ausgangsdatei auf keinen Fall überschreiben, sondern Default-Endungen wie `.com` für die komprimierte und `.decom` für die dekomprimierte Datei benutzt.
- Das Programm terminiert mit einer „ordentlichen“ Fehlermeldung, wenn keine Quelldatei mit dem angegebenen Namen gefunden wurde. Es soll ebenfalls mit einer Fehlermeldung terminieren und nichts geschrieben werden, wenn eine Datei mit dem Namen für die Zieldatei bereits existiert.
- Auf der Homepage stellen wir euch in `examples.zip` einige Beispiel-Dateien zum Testen zur Verfügung. Ihre Namen verraten, wie gut sie sich etwa komprimieren lassen. Wir behalten uns aber vor, eure Programme auch mit Dateien zu testen, die ihr nicht vorher hattet. Für die Dateien `good1` bis `good5` stellen wir euch dort auch die komprimierten Dateien `good1.cmp` bis `good5.cmp` zur Verfügung, mit denen ihr euer Dekompressionsprogramm unabhängig vom Kompressionsprogramm testen könnt.

**Hinweise:**

- Für den Algorithmus muss es sowohl möglich sein, neue Huffman-Bäume zu erzeugen, die ein Zeichen mit seiner Häufigkeit repräsentierten, als auch einen neuen Baum als Verschmelzung von zwei anderen Bäumen zu erzeugen.
- Ihr werdet sicherlich für die praktische Handhabung des Baums weitere Methoden benötigen: Zum Beispiel sollte es möglich sein für ein Token das zugehörige Codewort (rekursiv) erzeugen zu können. Nutzt beim Aufbau des Codeworts entweder eine `LinkedList<Boolean>` oder einen `String`.
- Ihr könnt eure Max-Queue aus der vorigen Aufgabe nutzen, indem ihr vor dem Einfügen in die Queue die Häufigkeiten mit -1 multipliziert.
- Die Methode `beginBitMode()` in `BitInputFile` gibt zurück wie viele Bits gelesen werden können. Das ist nützlich, wenn ihr die Code-Tabelle aus dem komprimierten File auslest.

Viel Spaß und Erfolg!