Objektorientierte Programmierung, Übungsaufgabe 4

Komprimierung

Jan Tammen <foobar@fh-konstanz.de>
Christoph Eck <ceck@fh-konstanz.de>

24. Juni 2005

Korrekturen und Ergänzungen

1 Speicherung des Huffman-Baums

Um den Huffman-Baum beim Decodieren einer Datei zu rekonstruieren, müssen in der codierten Datei ebenfalls die Häufigkeitsverteilungen der im Text vorkommenden Zeichen abgespeichert werden.

Um dies möglichst speicherplatzsparend zu erreichen, muss zunächst ein Weg gefunden werden, mit dem sich *bit*weise in eine Datei schreiben bzw. aus ihr lesen lässt. C++ bietet standardmäßig nur einen *byte*weisen Zugriff. Würde man die Daten byteweise ablegen, würde für jedes Zeichen max. 3 Byte (ausgehend vom ASCII-Code) verwendet; für die Häufigkeit entsprechend „Länge der Häufigkeitszahl“ $*$ 1 Byte.

Für die Speicherung des Bitcodes wird dies noch deutlicher: würde man jedes „Bit“ des Codes als ein `char` (1 Byte) speichern, wäre die codierte Datei schnell größer als die Quelldatei.

1.1 Speicherstruktur

Mit dem bitweisen Dateizugriff lässt sich etwa folgende Speicherstruktur für die codierte Datei einsetzen.

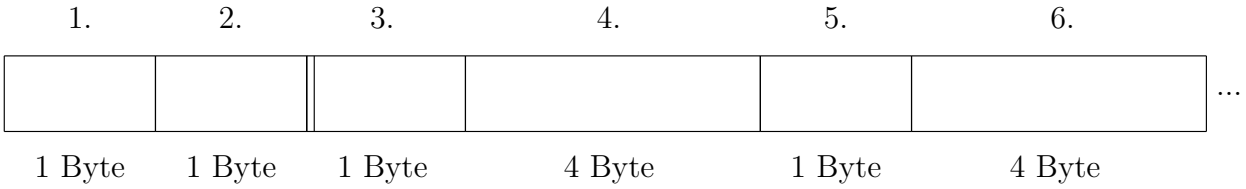

Abbildung 1: Beispiel Speicherstruktur

Dabei gliedert sich die Struktur in die folgenden Teile: (es wird bereits das Speichern des Bitcodes am Ende der Datei vorgesehen)

1. Anzahl der am Ende der Datei zu ignorierenden Bits. Diese Angabe wird benötigt, da die Bitanzahl stets eine ganzzahliges Vielfaches von 8 ist – im Huffman-Code kann der Code allerdings auch eine andere Länge haben. Ist dies der Fall, wird das Ende der Datei mit Nullen aufgefüllt und beim Decodieren die entsprechenden Anzahl an Bits ignoriert. (**1 Byte**)

2. Anzahl der nachfolgend codierten Zeichen (plus deren Häufigkeiten). Durch diese Angabe lässt sich später das Ende der Zeichenhäufigkeiten und der Beginn der codierten Datei ermitteln. (**1 Byte**)
3. Ab hier beginnt der „dynamische“ Teil der Datei. Es folgt jeweils zuerst das codierte ASCII-Zeichen (**1 Byte**), anschließend folgt
4. die dazugehörige Häufigkeit. (**4 Byte**, entspricht Datentyp `int`)
5. Nun folgt der nächste Block aus Zeichen und
6. Häufigkeit

Damit lässt sich der Speicherbedarf für die Ablage des Huffman-Baums wie folgt angeben:

$$1 \text{ Byte} + (\text{Anzahl Zeichen} * 5 \text{ Byte})$$

2 Text codieren

Durch Traversieren des Baumes erhält man die Bitcodes der einzelnen Zeichen.

2.1 Pseudocode

```
1  function traversiereBaum (KnotenZeiger p, Stack s)
2  if <p ≠ 0>
3      if <p ist AussenKnoten>   /// Endknoten ist erreicht
4          code ←−−<Inhalt von s>
5          s.pop()               /// Letztes "Bit" entfernen
6          codeTabelle.insert(<Zeichen von p>, code)
7      else                  /// Kein Endknoten, weiter traversieren
8          s.push_back(0)    /// prefix: gehe links: 0
9          traversiereBaum( <linkes Kind von p>, s )
10
11         s.push_back(1)    /// infix; gehe rechts: 0
12
13         traversiereBaum( <rechtes Kind von p>, s )
14         s.pop_back()      /// postfix; eine Ebene zurueckgehen
15     end if
16 end if
17 end function
```

Listing 1: Pseudocode Traversieren

2.2 Speicherung der codierten Daten

Ausgehend von der obigen Annahme, dass ein bitweiser Zugriff auf die Ausgabedatei besteht, können nun beim Codieren einer Datei die Bitcodes für die einzelnen Zeichen speichersparend ausgegeben werden. Der Bitcode-String wird dabei jeweils Zeichen für Zeichen durchlaufen und die einzelnen „Bits“ in einen Ausgabepuffer geschrieben.

Sind dort nun 8 Zeichen abgelegt, wird das Byte mittels eines Filestream-Objekts in die Ausgabedatei geschrieben. Wie oben beschrieben, müssen am Ende des Codiervorgangs eventuell „fehlende“ Bits durch Nullen ergänzt werden; diese können dann beim Decodieren einfach ignoriert werden.

Der Bitcode `0010001110011011` z.B. würde nun wie folgt direkt im Anschluss an die Häufigkeitsverteilung in die Datei geschrieben werden:

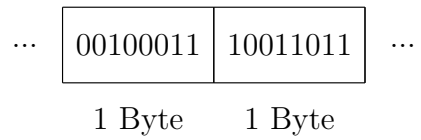

Abbildung 2: Beispiel Speicherstruktur, codierte Daten

3 Text decodieren

Hierbei muss zunächst aus der Quelldatei der Block am Anfang mit der Häufgikeitsverteilung gelesen und der Huffman-Baum rekonstruiert werden. Anschließend steht der Dateizeiger an der Stelle, an welcher der Bitcode beginnt. Nun wird Bit für Bit eingelesen und der Baum nach dem entsprechenden Zeichen durchsucht.