Objektorientierte Programmierung, Übungsaufgabe 4

Komprimierung

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

15. Juni 2005

1 Einleitung

Ein beliebiger Text soll mittels der *Huffman-Codierung* komprimiert werden. Bei diesem Verfahren werden zunächst die Häufigkeiten der Zeichen im Quelltext bestimmt. Anschliessend werden die einzelnen Zeichen mit einem Bitcode variabler Länge codiert; häufig auftretende Zeichen bekommen einen kurzen, selten auftretende Zeichen einen längeren Code zugewiesen.

Für einen beliebigen Eingabetext soll zum einen der optimale *Huffman-Code*, zum anderen der theoretische sowie der praktische *Reduktionsfaktor* ermittelt werden.

1.1 Algorithmus

Der Algorithmus zur Erstellung der Huffman-Codierung lässt sich wie folgt zusammenfassen:

1. Bestimmung der Zeichenhäufigkeiten im Quelltext.
2. Für jedes auftretende Zeichen einen **Blattknoten** generieren; dieser Knoten enthält das Zeichen c sowie die Häufigkeit h.
3. Die beiden Knoten mit der geringsten Häufigkeit bekommen einen gemeinsamen Elternknoten; dieser enthält die **Summe der Häufigkeiten** der beiden Kindknoten, sowie eine mit '0' bezeichnete **Verzweigung** zum linken und eine mit '1' bezeichnete **Verzweigung** zum rechten Kindknoten.
4. Die beiden Kindknoten als bearbeitet, den neu erzeugte Elternknoten als nicht bearbeitet markieren.
5. Falls es noch unbearbeitete Knoten gibt: fortfahren mit Schritt 3.
6. Abschließend wird der Baum traversiert und dabei der Bitcode für die einzelnen Zeichen erstellt (links: 0, rechts: 1)

2 Datenstrukturen und Methoden

2.1 Bestimmung der Zeichenhäufigkeiten

Für die Speicherung der Zeichenhäufigkeiten wird der STL-Datentyp `vector` benutzt. Die Quelldatei wird Zeichen für Zeichen eingelesen und dabei wird jeweils der entsprechende Wert für das aktuelle Zeichen im Array inkrementiert. Als Schlüssel dient dabei der ASCII-Code des Zeichens.

2.2 Huffman-Baum aufbauen

2.2.1 Datenstruktur: Knoten und Blätter

Für die inneren Knoten und Blätter (hier „AussenKnoten“ genannt) wird eine kleine Klassenhierarchie verwendet, deren Struktur im folgenden Diagramm aufgezeigt ist.

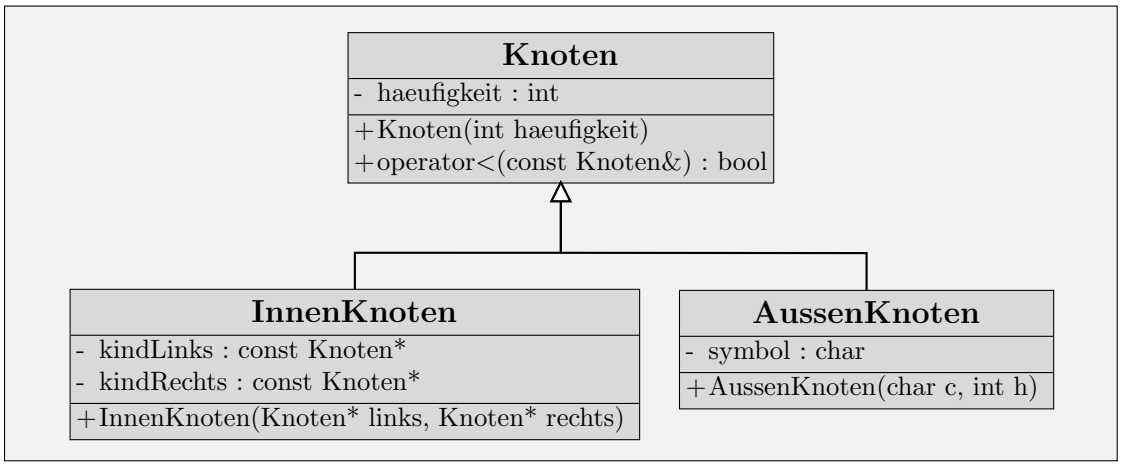

Abbildung 1: UML-Klassendiagramm

In der Basisklasse `Knoten` wird die Häufigkeit des Zeichens bzw. die Summenhäufigkeit der beiden Kindsknoten abgelegt. Weiterhin gibt es einen Vergleichsoperator, welcher für die Sortierung innerhalb der Vorrangwarteschlange benötigt wird.

Ein `InnenKnoten` hat zusätzlich zwei Kindsknoten, welche über entsprechende Zeiger erreicht werden können. Ein `AussenKnoten` hingegen hat keine weiteren Verzweigungen, allerdings wird hier noch das durch ihn repräsentierte Zeichen gespeichert.

2.2.2 Algorithmus

Um den Huffman-Baum zu erstellen, wird zunächst für jedes Element des Zeichenhäufigkeiten-Vektors ein Blattknoten angelegt und anschließend in eine Vorrangwarteschlange (die STL bietet hier den effizienten Datencontainer `priority_queue` auf Basis eines Heaps an) eingefügt. Somit werden die Blätter automatisch nach ihrer Häufigkeit aufsteigend sortiert vorgehalten (die Sortierung erfolgt über den `operator<`).

Anschließend werden in einer Schleife jeweils zwei Elemente aus der Schlange entfernt, um daraus den Elternknoten zu erstellen (Addition der Häufigkeiten und Verzeigerung der Kindsknoten). Die Schleife endet, wenn nur noch ein Element in der Schlange vorhanden ist – die Wurzel des Baumes ist erreicht.

2.2.3 Pseudocode

```
pq ←—<Vorrangwarteschlange>
sf ←—<Array mit Zeichenhäufigkeiten>   /// Länge 256, initialisiert mit 0

for <i> from 0 to <Länge sf> do
    if <sf[i] ≠ 0> then      /// Häufigkeit ist nicht Null
        Knoten node()
        node.zeichen ←—i
        node.haeufigkeit ←—sf[i]
        pq.push( node )
    end if
end for

while <Größe von pq > 1> do
    kindLinks ←—pq.top()   /// oberstes Element holen
    pq.pop()               /// anschliessend loeschen
    kindRechts ←—pq.top()  /// naechstes holen
    pq.pop()               /// ebenfalls loeschen

    elternKnoten ←—<neuer Knoten mit Summe + Zeigern>
    pq.push( elternKnoten ) /// neu erstellten Knoten einfuegen
end while
```

Listing 1: Pseudocode Aufbau Huffman-Baum

2.3 Textuelle Repräsentation des Huffman-Baums

Um die codierte Datei später wieder decodieren zu können, muss eine möglichst kompakte Darstellung des Code-Baumes in einer Datei gespeichert (z.B. am Anfang der codierten Datei) und zusätzlich zum Empfänger übertragen werden.

Dazu müssen in diese Code-Datei lediglich alle vorkommenden Zeichen mit der dazugehörigen Häufigkeit gespeichert werden; daraus lässt sich dann vor dem Decodieren der Huffman-Baum mittels des oben aufgeführten Algorithmus' wieder aufbauen.

2.4 Text codieren

Um nicht bei jedem Eingabe-Zeichen den Baum nach dem Code durchsuchen zu müssen, wird zunächst eine Zuordnungsstruktur (STL-Datencontainer `map`) eingesetzt. Dort werden alle Zeichen mit dem dazugehörigen Code eingetragen.

Den Code für jedes Zeichen erhält man durch **Traversieren** des Huffman-Baums. Dabei wird das aktuelle Code-Bit (links: 0, rechts: 1) in einen Stack eingefügt. Beim Erreichen eines Blattes wird der komplette Code samt Zeichen in die Zuordnungstabelle eingetragen und das letzte Bit aus dem Stack entfernt und mit dem Traversieren fortgefahren.

2.5 Ermittlung des Reduktionsfaktors

Beim Einlesen des Quelltextes wird die Gesamtanzahl der Zeichen ermittelt; multipliziert man diesen Wert mit der vom verwendeten Zeichensatz benötigten Bitanzahl (z.B. ASCII 8Bit), erhält man den Gesamtspeicherbedarf des Quelltextes.

Der Speicherbedarf des codierten Textes ergibt sich aus der Summe der Zeichenhäufigkeiten multipliziert mit der Codelänge des Zeichens.

Setzt man diese beiden Werte in Relation zueinander, erhält man den *theoretischen Reduktionsfaktor*. Für den *praktischen Reduktionsfaktor* muss zusätzlich der Speicherbedarf der Code-Tabelle berücksichtigt werden. Dieser errechnet sich aus der Anzahl der verschiedenen Zeichen multipliziert mit 5 Byte (typischerweise 1 Byte für das Zeichen (`char`) + 4 Byte für die Häufigkeit (`int`)).

3 Programmaufbau

Alle Funktionen, die der Erstellung des Huffman-Codes – und schließlich der Codierung/Decodierung von Dateien – dienen (Zeichenhäufigkeit bestimmen, Huffman-Baum erstellen, etc.) werden in einer Anwendungsklasse `Huffman` gekapselt; nachfolgend das UML-Klassendiagramm.

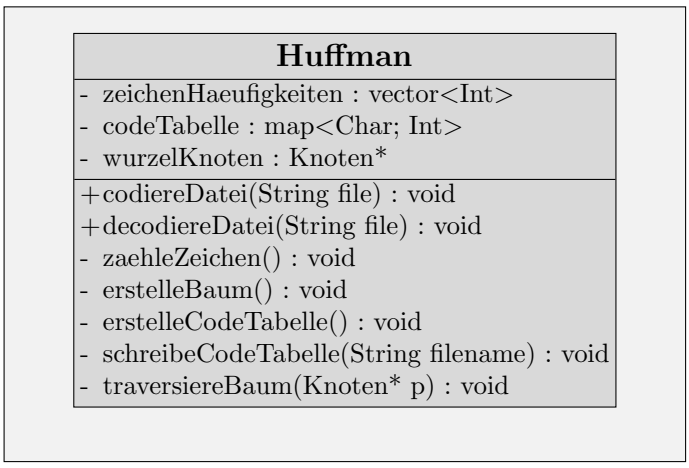

Abbildung 2: UML-Klassendiagramm