TU Ilmenau, Fakultät für Informatik und Automatisierung
FG Komplexitätstheorie und Effiziente Algorithmen
Univ.-Prof. Dr. M. Dietzfelbinger, Philipp Schlag, Stefan Walzer, Alexander Jeuk
https://moodle2.tu-ilmenau.de/course/view.php?id=2465
http://kt30.theoinf.tu-ilmenau.de

TECHNISCHE UNIVERSITÄT
**ILMENAU**

# Algorithmen und Datenstrukturen SS 2020 – Praktikum 6

Zur Bearbeitung zwischen 13. Juli - 17. Juli (KW 29)

Ziel dieses Praktikums ist es, einen codierten Text mit Hilfe eines Codierungsbaumes zu decodieren und umgekehrt einen Text mit Hilfe eines optimalen Codierungsbaumes zu codieren. Wiederholen Sie dazu Kapitel 10.2 („Huffman-Codes“) aus der Vorlesung.[1]

## Vorbemerkungen

Das zugrundeliegende Alphabet ist der ASCII-Zeichensatz, d. h., Bit-Strings der Länge 7 bzw. Zahlen aus $[2^7] = \{0, 1, \ldots, 127\}$. Ziel von Aufgabe 1 ist es, einen wie in Listing 1 angegebenen codierten Text zu dem ursprünglichen Text wie in Listing 2 zu decodieren. Umgekehrt soll in Aufgabe 2 ein Text wie in Listing 2 mit Hilfe eines optimalen Codierungsbaumes (Huffman-Baum) zu einem Text wie in Listing 1 codiert werden.

Listing 1: Codierter Text

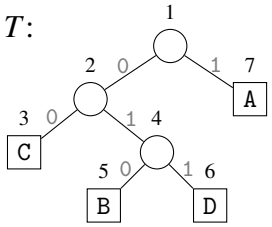
```
1  00110000110110000101100010011000001
   11101000000111111011000001 0111
```

Listing 2: Decodierter Text

```
1  AAABCCDAAADCCBAAA
```

Der Einfachheit halber enthält der codierte Text dabei jedes Bit als ASCII-Zeichen `0` bzw. `1` und besteht grundsätzlich aus zwei Abschnitten:

(1) Der erste Abschnitt enthält eine kompakte Repräsentation des verwendeten Codierungsbaums $T$. Der Baum wird dazu in Präorder-Reihenfolge durchlaufen. Für jeden besuchten Knoten $v$ wird eine `0` geschrieben, falls $v$ ein innerer Knoten ist. Ist $v$ ein Blatt, wird eine `1` gefolgt von der Blattbeschriftung bestehend aus sieben Bits geschrieben. In unserem Beispiel ergibt sich:




Codierungsbaum $T$ gespeichert:

| Präorder-Nr. | 1 | 2 | 3 | | | 4 | 5 | | | 6 | | | 7 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bitsequenz | 0 | 0 | 1 | 1000011 | | 0 | 1 | 1000010 | | 1 | 1000100 | | 1 | 1000001 | |

Man beachte, dass keine spezielle Markierung notwendig ist, um das Ende der Beschreibung des Codierungsbaums und den Beginn des zweiten Abschnittes zu erkennen. Ebenso lässt sich der Codierungsbaum eindeutig aus der gespeicherten Bitsequenz rekonstruieren.

(2) Im zweiten Abschnitt folgt der mit Hilfe des Codierungsbaums $T$ codierte Text. Im Beispiel von oben repräsentiert `11101000000111111011000001 0111` dann den Text `AAABCCDAAADCCBAAA`.

Die Teilaufgaben innerhalb einer Aufgabe bauen direkt aufeinander auf. Sie können (und sollten) deshalb Lösungen aus vorherigen Teilaufgaben, die alle Tests bestanden haben, in späteren Teilaufgaben wiederverwenden.

[1] Achten Sie darauf, den aktualisierten Foliensatz als Grundlage zu verwenden.

**Hinweise:**

- In C/C++ ist ein `char` Zeichen dasselbe wie sein *Index im ASCII-Zeichensatz*, z. B. ist `'0' == 48`. Wenn Sie in einem Array (`vector`) Informationen zu ASCII-Zeichen ablegen, bietet es sich daher an, als Index die Zeichen selbst zu verwenden.
- Achten Sie darauf, beim Codieren nicht nur die *druckbaren Zeichen*, sondern auch *Leerzeichen* sowie *Steuerzeichen* korrekt zu behandeln. Falls `cin.get(c).good() == true` gilt, haben Sie mit dieser Kombination das nächste ASCII-Zeichen von der Standardeingabe in die Variable `char c` eingelesen, ohne Leer- oder Steuerzeichen zu überlesen, wie es bei `cin >> c` passiert.

  **Bemerkung:** Wenn Sie die Starthilfen verwenden, wird sich automatisch darum gekümmert.
- Der *Abschluss einer Zeile* (in C++ durch `cout << endl` oder `cout << '\n'` auf der Standardausgabe erzeugt) unterscheidet sich zwischen Windows (`CR LF`) und Linux/macOS (nur `LF`). Die Beispieldateien bestehen daher jeweils nur aus einer einzelnen Zeile.[2]
- Für die explizite Konstruktion eines binären Codierungsbaums $T$ ist folgende Struktur (bekannt aus Praktikum 2) bereits vorgegeben:

```
1  struct Tree {
2    Tree *left; char letter; Tree *right;
3    ~Tree() { delete left; delete right; }
4  };
```

- Als **Starthilfe** gibt es dieses Mal jeweils zwei Dateien:
  - Eine Headerdatei mit Endung `.h`, die nützliche Definitionen (z. B. die obige Struktur `Tree`) enthält. Insbesondere kümmert sie sich darum, Ihnen die Ein- und Ausgabe zu erleichert (damit Sie sich um die oben beschriebenen Besonderheiten keine Sorgen machen müssen). Sie müssen keine Änderungen an dieser Datei vornehmen, um die zugehörigen Aufgaben zu lösen.
  - Eine C++-Quelltextdatei mit Endung `.cpp`. Diese enthält mehrere Methoden, die von Ihnen implementiert werden müssen. Der Startpunkt für Ihr Programm ist immer die Methode `void run()`.[3]

  Zum Bewerten mit dem DOMjudge-System müssen Sie dann immer **beide Dateien gleichzeitig hochladen**.

[2] Die Starthilfe in Aufgabe 2 nutzt diese Vereinbarung aus, damit dort die Debug-Konfiguration `INTERACTIVE` problemlos funktioniert. Wenn Sie stattdessen Eingabedateien mit mehreren Zeilen codieren wollen, müssen Sie `if(c == '\n') break;` (Zeile 46) in `encode.h` auskommentieren.

[3] Diese wird am Ende der in der Headerdatei definierten `main`-Methode aufgerufen.

## **Aufgabe 1** (Decodieren mittels Codierungsbaum)

Aus einem codierten Text (Listing 1) soll der zugehörige decodierte Text (Listing 2) bestimmt (und ausgegeben) werden. Die Starthilfe liest die per `<` an `cin` übergebene Eingabedatei (z. B. `./decode < sample.in`) zunächst komplett ein, macht diese über das Objekt `input` verfügbar, und ruft dann die Methode `void run()` auf. Innerhalb dieser müssen Sie sich um die Erzeugung der erwarteten Ausgabe kümmern. Die Eingabe können Sie dabei mit folgenden Mechanismen (von links nach rechts) lesen:

- Der Test `if(input)` liefert genau dann `true`, wenn noch nicht alle „Bits“ (ASCII-Zeichen `0` und `1`) der Eingabe gelesen wurden.
- `input >> b` für `bool b` liest das nächste Bit aus der Eingabe und speichert es in einer `bool`-Variablen `b`.
- `input >> c` für `char c` liest die nächsten sieben Bits aus der Eingabe und speichert das entsprechende ASCII-Zeichen in einer `char`-Variablen `c`.

**(a) Codierungsbaum rekonstruieren (p6-a1-a-tree):**

Bei Eingabe eines codierten Textes wie in Listing 1 soll zunächst nur der darin beschriebene binäre Codierungsbaum (explizit) rekonstruiert und in der folgenden Form ausgegeben werden:

```
1 ((C,(B,D)),A)
```

Den auf die Beschreibung des Codierungsbaumes folgenden codierten Text können Sie einfach ignorieren. (Insbesondere müssen Sie Ihn gar nicht lesen.)

**Wenn Sie die Starthilfe verwenden:**

- Die Methode `Tree* readTree()` soll die Eingabe ab der aktuellen Leseposition verarbeiten und daraus einen Codierungsbaum konstruieren. Wenn die Beschreibung des Codierungsbaums zu Ende ist, soll ein Zeiger auf dessen Wurzel zurückgegeben werden.
- Die Ausgabe von `Tree *t` erfolgt durch `cout << t` gemäß der obigen Form.

**(b) Decodieren (p6-a1-b-decode):**

Nutzen Sie den in Teilaufgabe (a) konstruierten binären Codierungsbaum nun, um den zweiten Teil der Eingabe (den codierten Text) zu decodieren. Die erwartete Ausgabe hat nun also die Form von Listing 2:

```
1 AAABCCDAAADCCBAAA
```

**Wenn Sie die Starthilfe verwenden:**

- Die Methode `char decodeChar(Tree *codingTree)` wird mit einem Zeiger auf einen Knoten des Codierungsbaumes aufgerufen. Ausgegeben werden soll die Beschriftung (`char` ASCII-Zeichen) des Blattes, das erreicht wird, wenn die nächsten Bits in der Eingabe zum Navigieren in die zugehörigen Unterbäume genutzt werden.
- Die Beschriftungen der Blätter können Sie wie gewohnt ganz einfach per `cout` ausgeben.

## **Aufgabe 2** (Codieren mittels (optimalem) Huffman-Baum)

Eine Text aus ASCII-Zeichen wie in Listing 2 soll mit Hilfe eines optimalen Codierungsbaumes wie in Listing 1 codiert werden. Die Starthilfe liest die per `<` an `cin` übergebene Eingabedatei (z. B. `./encode < sample.in`) zunächst komplett ein, macht diese über das Objekt `input` verfügbar, und ruft dann die Methode `void run()` auf. Innerhalb dieser müssen Sie sich um die Erzeugung der erwarteten Ausgabe kümmern. Die Eingabe können Sie dabei mit folgenden Mechanismen (von links nach rechts) lesen:

- Der Test `if(input)` liefert genau dann `true`, wenn noch nicht alle ASCII-Zeichen der Eingabe gelesen wurden.
- `input >> c` für `char c` liest das nächste ASCII-Zeichen aus der Eingabe und speichert es in einer `char`-Variablen `c`.

**(a) Huffman-Algorithmus ausführen (p6-a2-a-huffman):**

Bei Eingabe eines zu codierenden Textes wie in Listing 2 sollen zunächst nur die Buchstabenhäufigkeiten bestimmt und der Huffman-Algorithmus dafür ausgeführt werden. Achten Sie darauf, nur solche Buchstaben (ASCII-Zeichen) beim Huffman-Algorithmus zu betrachten, die auch tatsächlich (mit positiver Häufigkeit) im Text vorkommen. Die Ausgabe soll dann die folgende Form haben:

```
1 label: B D C A
2 p: 2 2 4 9 4 8 17
3 pred: 4 4 5 6 5 6
4 mark: 0 1 0 1 1 0
```

Die Zeilen 2 bis 4 enthalten die Werte der Arrays `p`, `pred` und `mark` des Huffman-Algorithmus. Die erste Zeile enthält die $n$ tatsächlich im Text vorkommenden ASCII-Zeichen $a_i$, wobei $a_i = \texttt{label}[i]$ im Text genau $\texttt{p}[i]$ mal vorkommt.

**Hinweise:**

- Achten Sie darauf, dass die Indizierung in C++ bei 0 beginnt, in den Vorlesungsfolien hingegen bei 1.
- Manchmal gibt es mehrere optimale Codierungsbäume. Um genau den Huffman-Baum der erwarteten Ausgabe (in der zugehörigen `ans`-Datei codiert) zu konstruieren, müssen Sie die folgenden **Regeln** beachten:

  (1) Die tatsächlich vorkommenden ASCII-Zeichen $\texttt{label}[0], \ldots, \texttt{label}[n-1]$ und deren zugehörigen Häufigkeiten $\texttt{p}[0], \ldots, \texttt{p}[n-1]$ sollen (wie vom Algorithmus gewünscht) nach wachsenden Häufigkeiten sortiert sein:

  $$\texttt{p}[0] \leq \texttt{p}[1] \leq \cdots \leq \texttt{p}[n-1]\,.$$

  Zusätzlich soll immer dann, wenn zwei Buchstaben die gleiche Häufigkeit haben, der kleinere Buchstabe (d. h. das ASCII-Zeichen mit kleinerem Index) links des größeren stehen:[4]

  Für alle $i \in \{0, 1, \ldots, n-2\}$ gilt: Wenn $\texttt{p}[i] = \texttt{p}[i+1]$, dann $\texttt{label}[i] < \texttt{label}[i+1]$.

  (2) Immer dann, wenn es mehrere (Kunst-)Buchstaben mit gleicher Häufigkeit gibt, soll zunächst derjenige mit kleinstem Index gewählt werden, d. h., der in den Arrays `p`/`pred`/`mark` am weitesten vorne/links steht.

  (3) Immer dann, wenn zwei seltenste (Kunst-)Buchstaben gewählt wurden, soll der zuerst gewählte das linke 0-Kind und der als Zweites gewählte das rechte 1-Kind des neuen Kunstbuchstabens werden.

  Die Pseudocode-Implementierung in der Vorlesung erfüllt bereits die Regeln (2) und (3).

[4] Im Beispiel haben B und D die gleiche Häufigkeit 2. Da im ASCII-Zeichensatz (und für `char`-Werte) `'B' == 66 < 68 == 'D'` gilt, steht B in den Arrays vor D.

**Wenn Sie die Starthilfe verwenden:**

- `void huffman()` soll den Huffman-Algorithmus durchführen. Dabei sollen insbesondere die beiden Arrays `vector<unsigned int> pred` und `vector<bool> mark` erzeugt werden.
- Sie können `input.countChars(...)` zum Zählen der ASCII-Zeichen wie folgt verwenden:

```
1  unsigned int n;
2  vector<int> labelIndex;
3  vector<char> label;
4  vector<unsigned int> p;
5  input.countChars(n, labelIndex, label, p);
```

Nach dem Aufruf enthält `n` die Anzahl an verschiedenen ASCII-Zeichen in der Eingabe. Für jedes dieser Zeichen gibt es einen Eintrag in `label` (das Zeichen selbst) und `p` (absolute Häufigkeit dieses Zeichens), die Position dieser Einträge ist in `labelIndex` notiert. Die oben genannte Regel (1) ist erfüllt. Für Listing 2 ergibt sich z. B.:

`n`: 4

`labelIndex`:

| 0 | … | 64 | 65 | 66 | 67 | 68 | 69 | … | 127 |
|---|---|---|---|---|---|---|---|---|---|
| '\0' | … | '@' | 'A' | 'B' | 'C' | 'D' | 'E' | … | |
| −1 | … | −1 | 3 | 0 | 2 | 1 | −1 | … | −1 |

`label`:

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| 'B' | 'D' | 'C' | 'A' |

`p`:

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| 2 | 2 | 4 | 9 |

- Vergessen Sie nicht, die Größe von `p` (mittels `p.resize(...)`) anzupassen, damit Sie die Häufigkeiten der Kunstbuchstaben zwischenspeichern können.
- Dank `encode.h` können Sie den Inhalt von `vector`-Instanzen ganz einfach per `cout` wie gewünscht ausgeben, z. B.: `cout << label << endl`.

**(b) Huffman-Baum konstruieren (p6-a2-b-tree):**

Bei Eingabe eines zu codierenden Textes wie in Listing 2 soll jetzt der in Teilaufgabe (a) (implizit) bestimmte binäre Codierungsbaum explizit konstruiert und wie folgt ausgegeben werden:

```
((C,(B,D)),A)
00110000110110000101100010011000001
```

Die erste Zeile stellt einen Codierungsbaum $T$ wie in Aufgabe 1 (a) dar. Bei der zweiten Zeile handelt es sich um die codierte Form von $T$. Dies ist also der erste Teil von Listing 1.

**Wenn Sie die Starthilfe verwenden:**

- Die Methode `Tree* computeTree()` soll den durch `pred` und `mark` beschriebenen optimalen Codierungsbaum explizit konstruieren und einen Zeiger auf dessen Wurzel zurückgeben.
- Die Ausgabe von `Tree *t` mittels `cout << t` erzeugt dann genau die in der ersten Zeile verwendete Darstellung.
- `void writeTree(Tree *codingTree)` soll den Codierungsbaum in codierter Form, d. h., wie in der zweiten Zeile, als „Bitstring“[5] ausgeben.

  Für eine etwas einfachere Ausgabe können Sie das Objekt `output` statt `cout` verwenden:

  - `output << b` für `bool b` schreibt das Bit `b` als einzelnes ASCII-Zeichen `0` (falls `b == false == 0`) bzw. `1` (falls `b == true == 1`) in die Ausgabe.
  - `output << c` für `char c` schreibt das ASCII-Zeichen `c` als entsprechende Folge von sieben `0`/`1`en in die Ausgabe.

**(c) Codieren (p6-a2-c-encode):**

Nutzen Sie Ihre Ergebnisse aus Teilaufgaben (a) und (b), um bei Eingabe eines zu codierenden Textes wie in Listing 2 nun die zugehörige codierte Ausgabe (bestehend aus dem codierten Huffman-Baum und dem eigentlichen codierten Text) wie in Listing 1 zu erzeugen:

```
00110000110110000101100010011000001111010000001111101100000010111
```

**Hinweise:**

- Achten Sie darauf, dass die Ausgabe nur aus einer einzelnen Zeile von `0`en und `1`en besteht.
- Für den zweiten Teil der Ausgabe sollten Sie keine explizite Repräsentation des Huffman-Baums wie in Teilaufgabe (b) verwenden. Nutzen Sie stattdessen einfach nur die Arrays `pred` und `mark`.

**Wenn Sie die Starthilfe verwenden:**

- Die Methode `void encodeChar(unsigned int index)`, aufgerufen für den Index eines Blattes/Buchstabens, soll die zugehörige Blattbeschriftung in codierter Form (d. h., die Beschriftung der Kanten auf dem Weg von der Wurzel zu diesem Blatt) in die Ausgabe schreiben. Wie in Teilaufgabe (b) bei `writeTree` können Sie dafür `output << b` für `bool b` verwenden.
- Die Methode `void encodeText()` soll dann `encodeChar` nutzen, um den gesamten Text mittels des optimalen Codierungsbaumes zu codieren.

[5] Ein einzelnes Bit soll als ASCII-Zeichen `0` bzw. `1` ausgegeben werden.