Implementierung des Packverfahrens im Programm Gzip

Referat von Ingo Rohloff

10. Dezember 1996

1 Informelle Beschreibung des Algorithmus

Der Gzip verbindet zwei bereits bekannte Kompressionsarten: Die Lempel-Ziv-77-Codierung (LZ77) und die Huffmancodierung. Das geschieht, indem die bei der LZ77-Codierung entstehenden Zeichen-, Längen, und Distanzcodes mit Hilfe der Huffmancodierung gespeichert werden. Die Ausgabe erfolgt in Blöcken. Für jeden Block gibt es drei Möglichkeiten, wovon die mit dem besten Packergebnis gewählt wird:

- Es werden zwei Huffmancodes berechnet, wovon der erste für die Zeichen und Längen, der zweite für die Distanzen verwendet wird. In diesem Fall müssen die Huffmancodes vor dem Block in komprimierter Form gespeichert werden, damit sie für die Dekomprimierung bekannt sind.
- Es werden zwei fest vorgegebene Huffmancodes verwendet.
- Der Block wird unkomprimiert gespeichert.

In diesem Vortrag geht es nur um die erste Möglichkeit, da bei den beiden anderen Möglichkeiten die Implementierung einfach ist.
Im Sourcecode des Programms ist der Teil für die LZ77-Codierung fast völlig unabhängig von dem Teil für die Huffmancodierung. Die einzige Verbindung sind die Funktionen ct_tally und flush_block, die beide zum Huffman-Teil gehören. ct_tally speichert einen Zeichen-, Längen oder Distanzcode für die spätere Ausgabe. flush_block gibt einen Block aus, d.h. alle bis jetzt mit ct_tally gespeicherten Codes.

2 Die Implementierung des LZ77 im Gzip

Der Gzip beschränkt die Länge der Muster beim LZ77 auf 258 Bytes und die Entfernung zur momentanen Position auf 32Kb. Um gleiche Muster zu finden wird zunächsteinmal über drei Bytes *gehasht*, d.h. für drei Bytes (B1 ... B3) wird ein 15 Bit Wert (Hash-Wert) berechnet. Die Rechenvorschrift (in C-Syntax) lautet:

```
(B1 << 10) ^ ( B2 << 5) ^ (B_3) & 07FFFh
```

Für die Verwaltung der Hash-Werte gibt es ein Verzeichnis das aus zwei Arrays besteht:

```
unsigned short prev[0x10000],head[0x8000];
```

head[Hash-Wert] ist der als letztes gefundene Index im Puffer, an dem sich der gleiche Hash-Wert ergibt. prev[Momentaner Index] ist der nächste Index mit dem gleichen Hash-Wert. Ist der Index null, so bedeutet das, daß es keine weitere Stelle im Puffer gibt, an der sich der gleiche Hash-Wert ergibt; das bedeutet gleichzeitig, das nie mit dem Anfang des Puffers (Puffer[0]) verglichen wird. Die beste Übereinstimmung zur momentanen Position wird mit folgendem Algorithmus gefunden:

```
Match=head[Hash_Wert]
Gr_Laenge=0
Wiederhole
( Wenn Puffer[Match]==Puffer[Akt_Pos] und
       Puffer[Match+Gr_Laenge]==Puffer[Akt_Pos+Gr_Laenge]
  ( Laenge=1
    Solange Puffer[Match+Laenge]==[Akt_Pos+Laenge] und
            Laenge<258
    ( Laenge++ )
    Wenn Gr_Laenge<Laenge
```

```
      ( match_start=Match
        Gr_Laenge=Laenge
        Wenn Gr_Laenge>=Reicht_schon
          return(Gr_Laenge)
      )
    )
    Match=prev[Match]
  ) Solange (Match nicht zu weit weg)
  return(Gr_Laenge)
```

Dieser Algorithmus ist im Gzip in der Funktion

```
int longest_match(unsigned short cur_match)
```

in optimierter Form realisiert. Aufgerufen wird diese Routine mit dem ersten zu überprüfenden Index (also head[Hash-Wert]). Zurückgegeben wird die Länge der längsten gefundenen Übereinstimmung; der Index von ihr wird in der globalen Variable match_start übergeben.
Der gesamte Algorithmus braucht noch die Verwaltung der Hash-Werte und die Übergabe der erzeugten Codes an die Routinen für die Huffmancodierung. Er liegt in zwei Varianten vor, die sich in der Packrate unterscheiden. Die erste einfachere Variante heißt im Progreamm deflate_fast und sieht so aus:

```
Solange noch Zeichen uebrig
( Hash_Wert an Akt_Pos berechnen
  Match=head[Hash_Wert]
  prev[Akt_Pos]=Match
  head[Hash_Wert]=Akt_Pos
  Wenn Match!=0 und
       Akt_Pos-Match<nicht_zu_weit_weg
  ( Best_Len=longest_match(Match)
  ) ansonsten
  ( Best_Len=0 )
  Wenn Best_Len>=3
  ( Ausgabe mit ct_tally(Akt_Pos-match_start,Best_Len-3)
          /* match_start: siehe oben longest_match */
    Wenn Best_Len<="max_insert_length"
    ( An allen Positionen von Akt_Pos bis Akt_Pos+Best_Len-1
      Hash-Werte berechnen und wie oben in head[] und prev[] eintragen.
      Danach ist Akt_Pos=Akt_Pos+Best_Len.
    ) ansonsten
    ( Akt_Pos=Akt_Pos+Best_Len d.h. Uebereinstimmung ueberspringen )
  ) ansonsten
  ( Ausgabe mit ct_tally(0,Puffer[Akt_Pos])
    Akt_Pos++
  )
  Wenn ct_tally sagt Flush
  ( flush_block aufrufen (Block Huffman-codiert ausgeben) )
)
flush_block aufrufen
```

Die zweite Variante ist eine Modifikation des ursprünglichen LZ77: Wenn eine Übereinstimmung gefunden wurde, wird die Ausgabe verzögert, bis man an der nächsten Stelle nach Übereinstimmungen gesucht hat. Gibt es eine neue Übereinstimmung die länger als die alte ist, gibt man das Zeichen vor der neuen Überenstimmung aus und behält die neue Übereinstimmung, deren Ausgabe man wieder verzögert. Ist die alte Übereinstimmung besser so wird sie normal ausgegeben. Diese sogenannte “lazy evaluation” führt zu besseren Packraten, weil man sich Distanzcodes spart. Beispiel (im Gzip kommt immer erst die Länge und dann die Distanz der Übereinstimmung):

```
Text: laute klavier klaute und so weiter
                    ^ Momentane Position
ohne lazy evaluation: (3,8) (4,15) u n ...
mit  lazy evaluation:   k   (6,15) u n ...
=> Distanzcode 8 gespart.
```

Der Algorithmus ist in der Funktion deflate verwirklicht und lautet:

```
Zeichen_Da=falsch
Best_Len=0
match_start=0
solange noch Zeichen uebrig
( Hash_Wert an Akt_Pos berechnen
  Match=head[Hash_Wert]
  prev[Akt_Pos]=Match
  head[Hash_Wert]=Akt_Pos
  Prev_Len=Best_Len
  Prev_Match=match_start
  Best_Len=0
  Wenn Match!=0 und
       Prev_Len< "max_lazy_match" und
       Akt_Pos-Match<nicht_zu_weit_weg
  ( Best_Len=longest_match(Match)
    Wenn Best_Len==3 und
         Akt_Pos-match_start<lohnt_sich_nicht
    ( Best_Len=0 )
  )
  Wenn Prev_Len>=3 und
       Best_Len<=Prev_Len
  ( Ausgabe mit ct_tally(Akt_Pos-1-Prev_Match,Prev_Len-3)
    An allen Positionen von Akt_Pos bis Akt_Pos+Prev_len-2
    Hash-Werte berechnen und wie oben eintragen.
    Danach ist Akt_Pos=Akt_Pos-1+Prev_len
    Prev_Len=0
    Best_Len=0
    Zeichen_Da=falsch
  ) ansonsten wenn Zeichen_Da
  ( Ausgabe mit ct_tally(0,Puffer[Akt_Pos-1])
    Akt_Pos++
  ) ansonsten (d.h. wenn nicht Zeichen_Da)
  ( Zeichen_Da=wahr
    Akt_Pos++
  )
  Wenn ct_tally sagt Flush
  ( flush_block aufrufen )
)
Wenn Zeichen_Da
( Ausgabe mit ct_tally(0,Puffer[Akt_Pos-1]) )
flush_block aufrufen
```

Der letzte wichtige Punkt ist die Pufferverwaltung. Kommt der Gzip-Algorithmus in die Nähe des Pufferendes, werden die oberen 32Kb an den Anfang des Puffers kopiert und der Rest wieder aufgefüllt. Wichtig ist dabei, das bei dieser Operation das Hashverzeichnis angepaßt werden muß. Alle Einträge in head und prev die größer als 32767 sind (also in der oberen Hälfte des Puffers lagen) werden um 32768 erniedrigt, alle anderen auf null gesetzt und damit für ungültig erklärt, weil sie nicht mehr im Puffer sind. Dieses Verfahren ist in der Funktion `fill_window` verwirklicht.

3 Die Huffmancodierung im Gzip

Im Gzip werden Längen und Zeichen in einem Huffmancode zusammengefaßt, so daß man bei der Dekodierung erkennen kann, ob es sich um eine Länge oder ein Zeichen handelt. Bei einer Länge folgt als nächstes ein Distanzcode, der für die LZ77 Dekodierung benötigt wird.
Wenn man Huffmancodes erzeugen will, muß man zunächsteinmal eine Häufigkeitsverteilung über die auftauchenden Codes erstellen. Diese Aufgabe übernimmt im Gzip die Funktion

```
int ct_tally (int dist, int lc)
```

Sie wird von dem LZ77-Teil augerufen und bekommt als Parameter die Distanz und den Längen- oder Zeichencode. Ist die Distanz null, wird *lc* als Zeichen angesehen, ansonsten als Länge. Längen und Zeichen werden in dem Array l_buf abgespeichert, wobei zur Unterscheidung ob es sich um eine Länge oder ein Zeichen handelt noch ein Bitarray f_buf mitgeführt wird. Ein gesetztes Bit bedeutet daß es sich um eine Länge handelt (hinter der dann ein Distanzcode folgen muß), ein gelöschtes, das es sich um ein Zeichen handelt. Die Distanzen werden im Array d_buf gespeichert.
Gleichzeitig inkementiert die Funktion die Absolute Häufigkeit der Codes die sie übergeben bekommt. Die Häufigkeiten werden dabei in zwei Arrays – eines für die Distanzen und eines für die Längen und Zeichen – gespeichert, die später zur Erzeugung der Huffmancodes benutzt werden. Ihre Deklaration sieht so aus:

```
typedef struct ct_data {
    union {
        ush  freq;       /* frequency count */
        ush  code;       /* bit string */
    } fc;
    union {
        ush  dad;        /* father node in Huffman tree */
        ush  len;        /* length of bit string */
    } dl;
} ct_data;
ct_data dyn_ltree[286*2+1];   /* literal and length tree */
ct_data dyn_dtree[30*2+1]; /* distance tree */
```

Während ct_tally werden also dyn_ltree[].fc.freq und dyn_dtree[].fc.freq inkrementiert. Wichtig dabei ist, daß es nur 30 Distanzcodes und 29 Längencodes gibt. Die Frage ist, wie kann man 256 mögliche Längen und über 32000 Distanzen auf diese Codes abbilden. Die Antwort lautet, daß es zu jedem möglichen Längen- bzw. Distanzcode noch sogenannte Extra-Bits gibt. Man weist mehreren Längen einen Längencode zu und beim codieren wird der Längencode gesendet und direkt darauffolgend die Extra-Bits, die angeben welche Länge gemeint ist. Beispiel:
Der Längencode 13 hat 2 Extra-Bits, d.h. man kann mit ihm 4 Längen speichern, nämlich 20,21,22 und 23. Will man nun die Länge 21 senden, sendet man erst den Huffmancode für den Längencode 13 und danach noch die zwei Bits 01. Die 29 Längencodes sind in dyn_ltree von Index 257 bis 285 gespeichert. Index 256 ist der End-Of-Block Code, der natürlich in jedem Block genau einmal vorkommt. Die restlichen Elemente von dyn_ltree werden zum Aufbauen des Huffman-Baumes verwendet (gleiches gilt für dyn_dtree).
Wenn ein Block gesendet werden soll, muß aus den Häufigkeiten der Huffmancode erzeugt werden. Die Funktion dafür heißt build_tree und geht dabei in drei Schritten vor:

1. Der Huffmanbaum wird erzeugt.
2. Aus dem Baum werden die Längen für die einzelnen Codes berechnet.
3. Mit Hilfe der Längenangaben werden die Codes erzeugt.

Wie bereits bekannt, ist es für den Huffman-Algorithmus nötig immer die kleinsten beiden Häufigkeiten zu finden und zu einer zusammenzufügen. Das Auffinden der kleinsten Häufigkeiten geschieht im Gzip, indem zunächst ein binärer Baum erzeugt wird, dessen Knoten jeweils die Nummer eines Elementes des zu bearbeitenden Baumes darstellt.
Dieser "Nummern-Baum" wird so sortiert, daß die Häufigkeiten der Elemente, deren Nummer in den Kindern eines Knotens gespeichert sind, größer gleich der Häufigkeit des Elementes ist, dessen Nummer im Knoten selber gespeichert ist. Das bedeutet, daß die Wurzel dieses Baumes die Nummer des Elementes enthält, das die kleinste Häufigkeit hat. Der "Nummern-Baum" wird in dem Array heap gespeichert. Die Idee dabei ist, daß die Wurzel bei heap[1] liegt und die Kinder von heap[n] heap[n*2] und heap[n*2+1] sind. Auf die Häufigkeit eines Knotens im heap kann man mit tree[heap[n]].fc.freq zugreifen. Zunächst werden alle Nummern der Elemente die mehr als einmal vorkamen, d.h. deren Häufigkeit größer 0 ist, sequentiell bei heap[1] beginnend eingetragen. Die Länge des heaps wird in heap_len gespeichert. Um den heap zu ordnen, wird die Funktion pqdownheap(tree,heap_index) verwendet; sie arbeitet nach folgendem Algorithmus:

```
v = heap[heap_index]
j = heap_index*2            (d.h. linkes Kind von heap_index)
```

Längencode	Extra-Bits	Längen	Distanzcode	Extra-Bits	Distanzen
0	0	3	0	0	0
1	0	4	1	0	1
2	0	5	2	0	2
3	0	6	3	0	3
4	0	7	4	1	4-5
5	0	8	5	1	6-7
6	0	9	6	2	8-11
7	0	10	7	2	12-15
8	1	11-12	8	3	16-23
9	1	13-14	9	3	24-31
10	1	15-16	10	4	32-47
11	1	17-18	11	4	48-63
12	2	19-22	12	5	64-95
13	2	23-26	13	5	96-127
14	2	27-30	14	6	128-191
15	2	31-34	15	6	192-255
16	3	35-42	16	7	256-383
17	3	43-50	17	7	384-511
18	3	51-58	18	8	512-767
19	3	59-66	19	8	767-1023
20	4	67-82	20	9	1024-1535
21	4	83-98	21	9	1536-2047
22	4	99-114	22	10	2048-3071
23	4	115-130	23	10	3072-4095
24	5	131-162	24	11	4096-6143
25	5	163-194	25	11	6144-8191
26	5	195-226	26	12	8192-12287
27	5	227-257 !	27	12	12288-16383
28	0	258	28	13	16384-24575
			29	13	24575-32767

Tabelle 1: Verteilung der Längen und Distanzen auf Längen- und Distanzcodes

```
Solange j<=heap_len
( Wenn j<heap_len und
       Haeufigkeit heap[j+1] < Haeufigkeit von heap[j]
  ( j++ )
  Wenn Haeufigkeit von v <= Haeufigkeit von heap[j]
  ( Schleifenabbruch )
  heap[heap_index] = heap[j]
  heap_index = j
  j = j*2
)
heap[heap_index] = v
```

D.h. heap[heap_index] ist zunächst die Wurzel eines ansonsten richtig vorsortierten Teilheaps und wird in diesen Heap eingeordnet, so daß dieser Teilheap komplett sortiert ist. Danach steht bei heap[heap_index] die Nummer des Elementes mit der kleinsten Häufigkeit innerhalb des Teilheaps.
Am Anfang wird der Heap geordnet, indem man eine Schleife von heap_len/2 bis 1 durchläuft und dabei immer pqdownheap mit dem Schleifenindex als Parameter aufruft.
Ist der Heap so vorsortiert, wird der eigentliche Huffman-Baum folgendermaßen aufgebaut:

```
(heap_len gibt an wieviele Codes im Heap gespeichert sind)
node = erste freie Nummer im tree
heap_max = Letzter Index im heap
Wiederhole
( n = heap[1]           (d.h. Nummer des Elements mit kleinster Haeuf.)
  heap[1]=heap[heap_len--]    (Letztes El. des heaps an die Spitze kopieren)
  pqdownheap(1)               (neu sortieren)
  m=heap[1]             (jetzt hat man in n und m die Nummern
                         der Elemente mit den kl. H.)

  heap[--heap_max]=n    (Am Ende von heap, werden die Nodes des kompletten
  heap[--heap_max]=m     Baumes mit absteigender Haeufigkeit sortiert)
```

```
  tree[node].fc.freq=tree[n].fc.freq + tree[m].fc.freq
                       (neuen Node erzeugen mit H. von n und m zusammen)
  tree[n].dl.dad=node  (Der neue Node ist Vater von den Nodes
  tree[m].dl.dad=node   mit den Nummern n und m)

  heap[1]=node         (Neuen Node anstelle der beiden alten eintragen)
  pqdownheap(1)        (heap wieder durchsortieren)
) Solange heap_len>1
heap[--heap_max]=heap[1]
```

Wie man sieht, werden am Ende von heap die einzelnen Nodes von tree mit absteigender Häufigkeit geordnet und heap_max gibt den Index der Wurzel an. Diese Ordnung ist nötig um im 2.Schritt die Längen der einzelnen Codes mit der Funktion gen_bitlen zu erzeugen.

```
Setze alle Laengen in bl_count[] auf null.
overflow=0
tree[heap[heap_max]].dl.len=0                (Wurzel bekommt Laenge null)
Von h=heap_max+1 bis Ende von heap wiederhole
( n=heap[h]
  bits=tree[tree[n].dl.dad].dl.len + 1       (Ein bit laenger als der Vater )
  Wenn bits > 15                             (max. Laenge ist 15 Bits)
  ( bits=15, overflow++ )
  tree[n].dl.len=bits                        (dad ist damit ueberschrieben)
  Wenn n<=max_code       (Handelt es sich um ein Blatt des Huffman-Baumes?)
  ( bl_count[bits]++ )   (Dann haben wir einen Code mit Laenge bits mehr)
)
Wenn overflow==0
( Beende funktion )      (Alle Huffmancodes <=15 Bit)

Wiederhole     (L"angenverteilung so ausgleichen das alle codes<= 15bit)
( bits=14
  Solange (bl_count[bits]==0)  (Suche die groesste Codelaenge die vorkommt)
  ( bits-- )
  bl_count[bits]--           (Kommt einmal weniger vor, dafuer
  bl_count[bits+1] += 2       gibts die naechst groessere zweimal mehr)
  bl_count[15]--             (Es gibt jetzt einen Node weniger mit
  overflow -= 2               laenge=15 und 2 weniger mit laenge>15)
) Solange overflow>0      (Overflow ist immer ein Vielfaches von 2)

h=letzter Index von heap + 1       (Laengen neu berechnen)
Von bits=15 bis 1 wiederhole
( n=bl_count[bits]
  Solange n>0
  ( m=heap[--h]
    wenn (m<=maxcode)
    ( tree[m].len=bits
      n--
    )
  )
)
```

Nach diesem Schritt steht in tree[Code].len die Länge des Huffmancodes und in bl_count[] die Längenverteilung, d.h. wieviele Codes es mit mit einer bestimmten Länge gibt.
Im letzten Schritt werden die echten Codes aus der Längenverteilung in bl_count[] und den Angaben in tree[].dl.len in der Funktion gen_codes erzeugt:

```
code=0
Von bits=1 bis 15 wiederhole
( code= (code + bl_count[bits-1]) << 1
  next_code[bits] = code
)
```

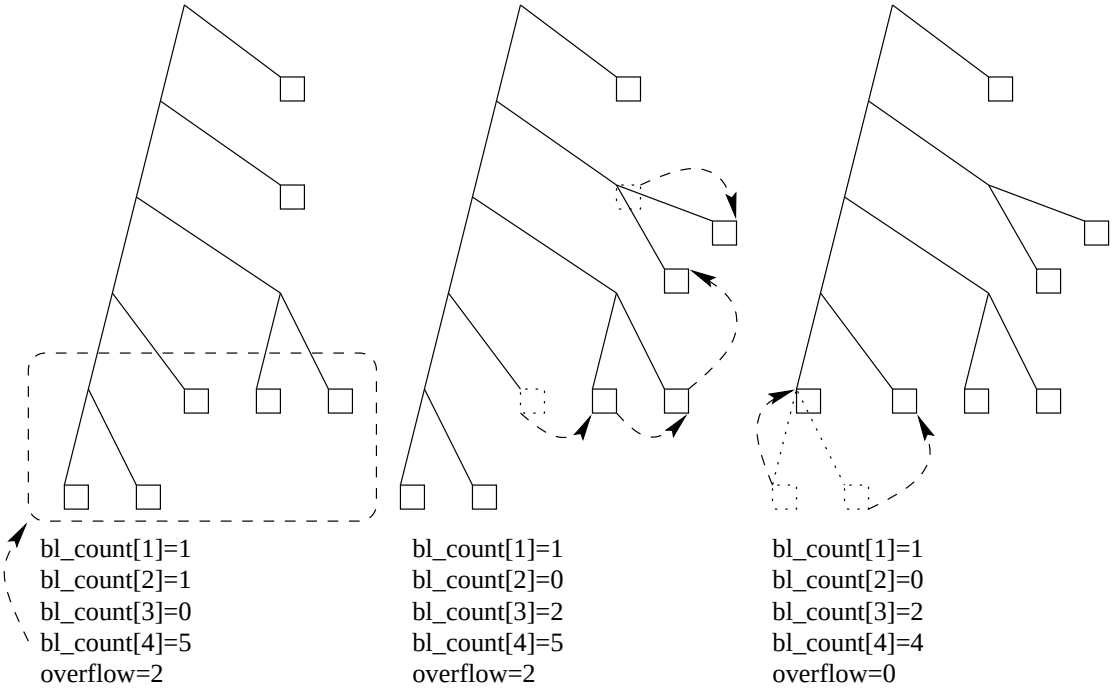

Abbildung 1: Beispiel für eine Längenanpassung bei einer Maximallänge von 4

```
Von n=0 bis letztes Zeichen im Baum  wiederhole
( len=tree[n].dl.len
  Wenn len>0
  ( tree[n].fc.code = next_code[len]
    next_code[len]++
  )
)
```

Mit den so erzeugten Codes kann man die in d_buf und l_buf gespeicherten Distanzcodes bzw. Zeichen-/Längencodes schnell codieren. Allerdings muß man wie schon erwähnt die Huffmancodes vor dem eigentlichen Block speichern, damit eine Decodierung möglich wird.

4 Das Speichern der Huffmancodes von d_tree und l_tree

Der Gzip speichert von d_tree und l_tree nur die Codelängen zu jedem Distanzcode bzw. Zeichen/Längencode ab. Aus diesen Daten läßt sich die Längenverteilung bl_count[] erstellen und damit hat man alles was man für die Funktion gen_codes benötigt.
Das Abspeichern erfolgt, wie nicht anders zu erwarten, wieder komprimiert. Verwendet wird dabei eine RLE-Kompression, gekoppelt mit einem *dritten* Huffmancode. Das Alphabet dieses dritten Huffmancodes besteht aus nur 19 Zeichen. Zeichen 0-15 sind für die möglichen Codelängen in d_tree und l_tree zuständig, Zeichen 16,17 und 18 sind die drei für die RLE-Kompression notwendigen Sonderzeichen. Diese drei Sonderzeichen haben folgende Bedeutung:

16: Das letzte gesendete Zeichen wird 3-6 mal wiederholt. Dazu werden noch zwei zusätzliche Bits nach diesem Sonderzeichen gesendet.

17: Steht für 3-10 mal Zeichen 0. Dazu werden noch 3 zusätzliche Bits gesendet.

18: Steht für 11-138 mal Zeichen 0. Es müssen also noch 7 Bits zusätzlich gesendet werden.

Der dritte Huffmancode wird ebenso wie die beiden anderen Huffmancodes aus der Häufigkeit der auftretenden Zeichen gewonnen. Diese Häufigkeitsverteilung wird mit Hilfe der Funktion scan_tree gewonnen und wird in bl_tree[Zeichen].fc.freq gespeichert. bl_tree ist wie dyn_ltree und dyn_dtree vom Typ ct_data.

```
tree[max_code+1].dl.len=-1 (Damit letzter Code geschrieben wird
```

```
count=0                      => sicher nextlen!=curlen siehe unten)
max_count=7                 (Sonderzeichen => 1 Zeichen + max  6mal wdh)
min_count=4                 (Sonderzeichen => 1 Zeichen + mind 3mal wdh)
nextlen=tree[0].dl.len
prevlen=-1                  (gibts noch nicht)
Wenn nextlen==0
( max_count=138             (Max  138 Nuller auf einmal kodieren)
  min_count=3               (Mind   3 Nuller fuer Sonderzeichen noetig)
)
Von n=0 bis Letzter Code in tree wiederhole
( curlen=nextlen
  nextlen=tree[n+1].dl.len
  count++
  Wenn count==max_count oder           (D.h. neuen code anfangen)
       curlen!=nextlen
  ( Wenn count<min_count               (Wenn keinen Sondercode benutzen)
    ( bl_tree[curlen].fc.freq+=count ) (=> count mal curlen senden)
    ) ansonsten wenn curlen!=0         (Wenn keinen Sondercode0 benutzen)
    ( Wenn curlen!=prevlen             (Wenn letzter Sondercode f"ur
      ( bl_tree[curlen].fc.freq++ )     anderes Zeichen => Zeichen senden)
      bl_tree[16].fc.freq++            (Sondercode f"ur Wiederholung)
    ) ansonsten wenn count<=10         (Wenn Sondercode f"ur <=10mal 0)
    ( bl_tree[17].fc.freq++ )
    ) ansonsten                        (Wenn Sondercode f"ur >10mal 0)
    ( bl_tree[18].fc.freq++ )

    count=0
    prevlen=curlen
    Wenn nextlen==0                    (Wenn jetzt nuller kommen)
    ( max_count=138
      min_count=3
    ) ansonsten wenn curlen==nextlen   (Wenn nochmal selbes Zeichen kommt)
    ( max_count=6                      (=> braucht man es nicht nochmal
      min_count=3                       fuer die Wdh zu speichern)
    ) ansonsten
    ( max_count=7                      (1 Zeichen + max  6 mal Wdh)
      min_count=4                      (1 Zeichen + mind 3 mal Wdh)
    )
  )
)
```

scan_tree wird mit dyn_ltree und dyn_dtree als Parameter aufgerufen. Danach sind die Häufigkeiten in bl_tree gesetzt und die Codes für die Zeichen in bl_tree können mit build_tree erzeugt werden. Der einzige Unterschied ist das bei bl_tree die Codelänge auf 7 Bits begrenzt ist und nicht auf 15.
Nachdem auch die bl_tree codes erzeugt wurden können die Längenverteilungen aller drei Bäume gesendet werden. Das geschieht mit der Funktion send_all_trees:

```
Sende Anzahl der L"angen in dyn_ltree
Sende Anzahl der L"angen in dyn_dtree
Sende Anzahl der L"angen in bl_tree
Sende die Codel"angen f"ur bl_tree mit jeweils 3 Bits
  (Codel"angen von 0-7 m"oglich)
send_tree(dyn_ltree)
send_tree(dyn_dtree)
```

send_tree hat eine ähnliche Struktur wie scan_tree, da in scan_tree die RLE-Kompression praktisch schon durchgeführt wurde um auch die Häufigkeiten der RLE-Sonderzeichen zu erfassen:

```
count=0                      (tree[max_code+1].dl.len=-1 schon passiert)
max_count=7
min_count=4
```

```
nextlen=tree[0].dl.len
prevlen=-1                    (gibts noch nicht)
Wenn nextlen==0
( max_count=138
  min_count=3
)
Von n=0 bis Letzter Code in tree wiederhole
( curlen=nextlen
  nextlen=tree[n+1].dl.len
  count++
  Wenn count==max_count oder
       curlen!=nextlen
  ( Wenn count<min_count
    ( Solange count!=0
      ( Huffman-Code f"ur curlen senden
        count--
      )
    ) ansonsten wenn curlen!=0
    ( Wenn curlen!=prevlen
      ( Huffman-Code f"ur curlen senden
        count--
      )
      Huffman-Code f"ur Sonderzeichen 16 senden
      Sende (count-3) mit 2 Bit Breite
    ) ansonsten wenn count<=10
    ( Huffman-Code f"ur Sonderzeichen 17 senden
      Sende (count-3) mit 3 Bit Breite
    ) ansonsten
    ( Huffman-Code f"ur Sonderzeichen 18 senden
      Sende (count-11) mit 7 Bit Breite
    )
    count=0
    prevlen=curlen
    Wenn nextlen==0
    ( max_count=138
      min_count=3
    ) ansonsten wenn curlen==nextlen
    ( max_count=6
      min_count=3
    ) ansonsten
    ( max_count=7
      min_count=4
    )
  )
)
```

Jetzt muß nur noch der eigentliche Inhalt des Blockes mit compress_block(ltree,dtree) gesendet werden:

```
lx=0
dx=0
Wiederhole
( lc=l_buf[lx++]
  Wenn f_buf sagt lc ist Zeichen
  ( Sende Huffmancode f"ur Zeichen
  ) ansonsten
  ( code = L"angencode f"ur lc
    Sende Huffmancode f"ur (code+257)
    extra = Zusatzbits f"ur L"angencode(code)
    Wenn extra!=0
    ( lc -= Grundl"ange f"ur diesen L"angencode
      Sende (lc) mit extra Bits Breite
```

```
    )
    dist=d_buf[dx++]
    code = Distanzcode f"ur dist
    Sende Huffmancode f"ur code
    extra = Zusatzbits f"ur Distanzcode(code)
    Wenn extra!=0
    ( dist -= Grunddistanz f"ur diesen Distanzcode
      Sende (dist) mit extra Bits Breite
    )
) Solange Noch Zeichen in l_buf
Sende Huffmancode f"ur (EndOfBlock)
```

"Nachwort"

Ich habe leider erst nach der Bearbeitung des Themas festgestellt, daß es sich bei vielen der verwendeten Strukturen und Algorithmen um echte Standardlösungen handelt, die eigentlich jeder Informatiker kennen sollte. Praktisch alle der interessanten Algorithmen sind z.B. in Robert Sedgewick's Buch "Algorithms in C" zu finden. Tatsächlich steht in diesem Buch eine detaillierte Beschreibung zur Implementierung des Aufbaus eines Huffmancodes, die teilweise identisch ist mit der die im gzip verwendet wird.
Ich kann nur jedem Studienanfänger raten sich ein ähnliches Buch anzuschaffen, da zumindest in unserer Grundstudiumsvorlesung kaum Algorithmen (und besonders nicht deren Implementierung) besprochen worden sind, obwohl man bei der Lösung von Informatikproblemen praktisch immer auf diese Standardalgorithmen stößt.

Verwendete Release: gzip 1.2.4. Zu finden z.B. im LEO unter
/pub/comp/gnu/gzip/gzip-1.2.4.tar.gz