An Introduction to Video Compression in C/C++

Fore June

Chapter 1

Huffman Encoding

8.1 Introduction

The 3D run-level tuples discussed in the last chapter can be encoded by an entropy encoder, which in general yields significant compression. Entropy encoding is a reversible or lossless process; exact data can be recovered in the decoding process from the encoded data. Arithmetic coding and Huffman coding are two popular methods of entropy encoding with the former giving slightly better results and consuming more computing power. This kind of encoding is also referred to as variable-length coding (VLC) because the codewords representing symbols are of varying lengths. We shall only discuss the Huffman encoding method here.

The commonly used generalized ASCII code uses 8 bits to represent a character and unicode uses 16 bits to do so; these are fixed-length codes, which are simple but inefficient in the representation. Huffman coding assigns a variable-length codeword to each symbol (or tuple here) based on the probability of the occurrence of the symbol. Frequently occurring symbols are represented with short codewords whilst less common symbols are represented with longer codewords; in this way we have a shorter average codeword length and thus save space to store the codewords, leading to data compression.

We say that a code has the **prefix property** and is a prefix code if no codeword is the prefix, or start of the codeword for another symbol. A code with codewords {1, 01, 00} has the prefix property; a code consisting of {1, 0, 01, 00} does not, because "0" is a prefix of both "01" and "00". A non-prefix code like { 1, 0, 01, 00 } cannot be instantaneously decoded because when we receive a bitstream such as "001", we do not know whether it consists of { '0', '0', '1' }, or { '0', '01' } or { '00', '1' }. On the other hand, a prefix code can be instantaneously decoded. That is, a message can be transmitted as a sequence of concatenated codewords, without any out-of-band markers to frame the words in the message. The receiver can decode the message unambiguously, by repeatedly finding and removing prefixes that form valid codewords, which are impossible if the message is formed by a non-prefix code as shown in the above example. Prefix codes are also known as prefix-free codes, prefix condition codes, comma-free codes, and instantaneous codes.

Not only that Huffman codes are prefix codes, they are also optimal in the sense that no other prefix code can yield a shorter average codeword length than a corresponding Huffman code. It is the foundation of numerous compression applications, including text compression, audio compression, image compression, and video compression. It is a building block of many contemporary multi-media applications.

Huffman code was developed by David A. Huffman while he was a Ph.D. student at MIT, and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes." Huffman codes are easiest to understand and implement if we use trees to represent them though

the tree concept was not used when Huffman first developed the code. Briefly, Huffman tree is a **binary tree** (an ordered 2-ary tree) with a weight associated with each node. Let us first consider some simple examples to understand its principles.

Consider the following two codes.

Symbol	Code 1	Code 2
a	000	000
b	001	11
c	010	01
d	011	001
e	100	10

Code 1 is a fixed-length code with codeword length 3 and Code 2 is a variable-length code. Both of them have the prefix property (note that a fixed-length code always has the prefix property). They can be represented by binary trees like those shown in Figure 8-1. Decoding a bitstream is simply a process of traversing the binary tree; we start from the root of the tree and go left or right based on whether the current bit examined is 0 or 1 until we reach a leaf, which is associated with a symbol; we then start from the root again and examine the next bit and so on. For example, Code 2 decodes the string "000100111" uniquely to "aecb". The average codeword length of a code is equal to the average depths of of the leaves of the corresponding. It is obvious that Code 2 always has a shorter average codeword length when compared to Code 1, as the tree for Code 2 has a shorter average depth of its leaves.

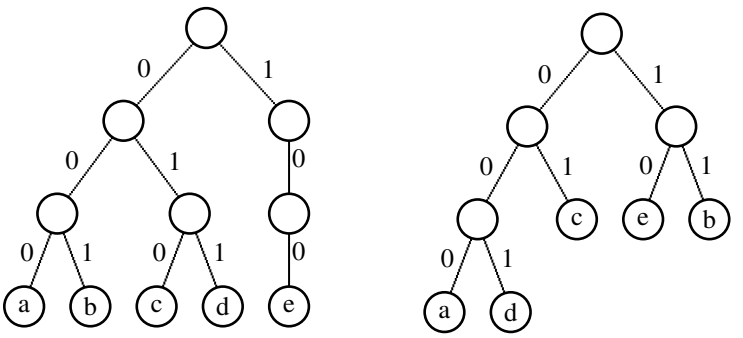


Figure 8-1. Binary Tree Representations of Code1 (left) and Code 2 (right)

8.2 Huffman Codes

Consider an example that the probabilities of the occurrence of symbols are known as shwon in the table below. In this case, we can calculate the average lengths $\overline{L}$ of the codewords of the two codes. Obviously, Code 1 is a fixed-length code and the average length is 3. For Code 2, we need to take into account the frequency of occurrence of each symbol:

$$\begin{aligned} \text{Code 1} : \overline{L} &= 3\ bits \\ \text{Code 2} : \overline{L} &= 0.35 \times 2 + 0.20 \times 2 + 0.20 \times 3 + 0.15 \times 3 + 0.10 \times 3 = 2.45\ (bits) \end{aligned} \tag{8.1}$$

Symbol	Frequency	Code 1	Code 2
a	0.35	000	00
b	0.20	001	10
c	0.20	010	011
d	0.15	011	010
e	0.10	100	110

Code 2 is a prefix code and has a significantly shorter average codeword length than that of Code 1. *But is Code 2 the best code for the given frequencies? Can we do better than Code 2, creating a code that has shorter $\overline{L}$ than Code 2?* This question can be answered by constructing a Huffman tree to obtain a Huffman code for the given symbols and frequencies. We start from a forest of trees; each tree in the forest has only one single node (which is a root as well as a leaf); each root consists of the symbol and the weight (probability) of it. In this example, we totally have five single-noded trees as shown in Figure 8-2:

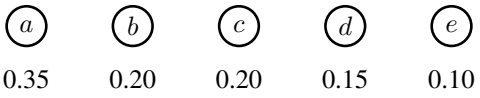


Figure 8-2. A Forest of Single-noded Trees

Next, we merge the two trees whose roots have lowest weights and calculate the sum of the two weights. We assign the sum as the weight to the root of the merged tree. The resulted forest is shown in Figure 8-3, where we have merged two pairs of roots that have the lowest weights.

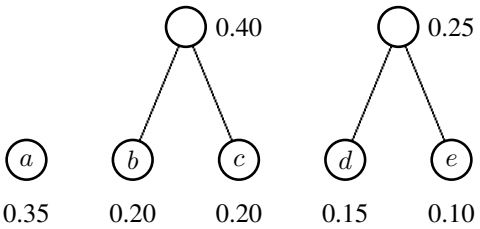


Figure 8-3. Merging Two Pairs of Roots with Lowest Weights

We repeat the above merging process until there is only one tree in the forest. Figure 8-4 and Figure 8-5 show two more iterations of the process. Note that for clarity of presentation, some node positions have been rearranged.

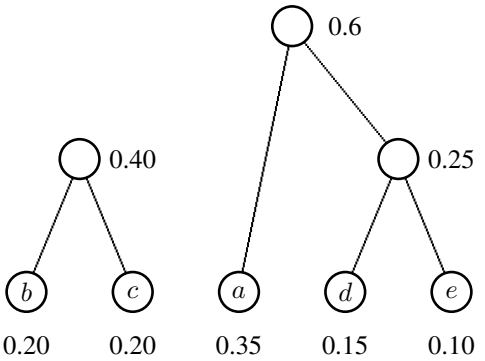


Figure 8-4. Merging Two More Roots with Lowest Weights

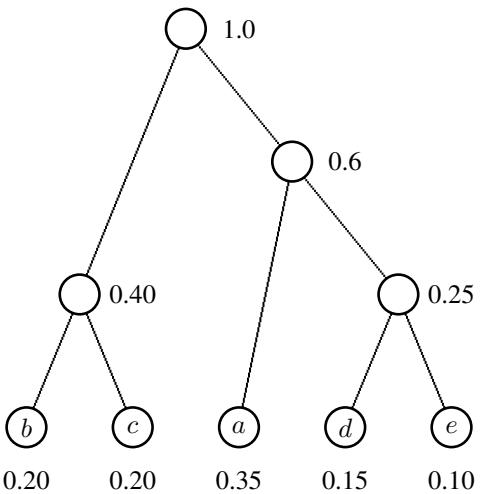


Figure 8-5. Merging All Trees Yields a Huffman Tree

As shown in Figure 8-5, the final single tree obtained is the Huffman tree that we need. To obtain the codeword for a symbol, we traverse the tree, starting from the root, until we arrive at a leaf, which contains the symbol; in the traversal, we generate a 0 on going left and a 1 on going right (it works just as well if we generate a 1 on going left and a 0 on right) as shown in Figure 8-6. The sequence of 1's and 0's parsed in the traversal from the root to the symbol is the codeword. The following table shows the Huffman code obtained from the Huffman tree of Figure 8-6.

Symbol	Codeword	Length
a	10	2
b	00	2
c	01	2
d	110	3
e	111	3

The average length of the Huffman code is

$$\overline{L} \quad = 0.35 \times 2 + 0.20 \times 2 + 0.20 \times 2 + 0.15 \times 3 + 0.10 \times 3 = 2.25 \; (bits) \tag{8.2}$$

which is shorter than that of Code 2 (2.45 bits) discussed above. Actually, one can prove that

Huffman code is optimal. That is, a Huffman code always gives the shortest average code length of all prefix codes for a given set of probabilities of occurrences of symbols. (We'll skip the proof here.)

Note that the symbols are not limited to alphabets and letters. They can be any quantitative values or even abstract objects. Note also that a Huffman decoder does not need to know the probability distribution of the symbols in order to decode them. It only needs to know the Huffman tree to decode a bit-stream consisting of codewords encoded by a Huffman code. In the decoding process, we start from the root of the tree and traverse the tree according to the 0's and 1's we read from the bit-stream until we reach a leaf to recover the encoded symbol; we then start from the root again and read in further bits for traversal to obtain the next symbol and so on. For example, suppose the following is the output bit-stream resulted from encoding a sequence of symbols using the Huffman tree of Figure 8-6:

$$00101110110 \tag{8.3}$$

Upon decoding, we start from the tree root and first read in '0 0', reaching symbol 'b'; we then start from the root again and read in '1 0', reaching symbol 'a'; next, the bit sequence '1 1 1' gives symbol 'e'; next, '0 1' gives 'c' and finally '1 0' gives 'a'. Therefore, the encoded sequence of symbols of the bit-stream (8.3) is "b a e c a".

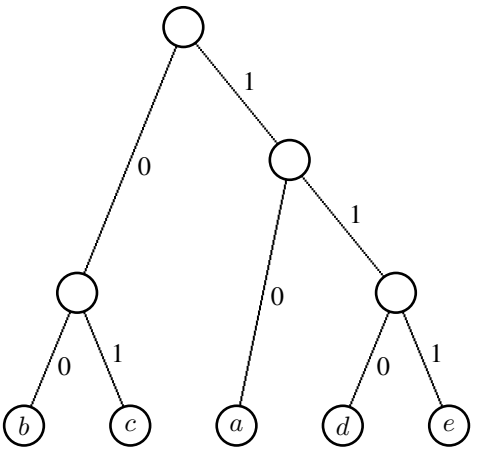


Figure 8-6. Traversing the Huffman Tree

8.3 Huffman Tree Properties

A Huffman tree can be conveniently constructed using a priority queue. A priority queue is a special queue that associates each element with a priority value. A general queue has the property of "First In First Out (FIFO)". That is, the first element that enters the queue is the first to be removed. In other words, the first entered-element is always at the front of the queue which is the next element to be deleted. This happens in a cashier station of a supermarket; the customer who arrives first at the cashier is the first one to be served. On the other hand, a priority queue does not follow the FIFO scenario; it is the element that has the highest priority in the queue that will be deleted first. This could happen when the president of a big company meets a number

of visitors; she would first meet the most important visitor before meeting other less important people. Therefore, the element at the front of a priority queue always has the highest priority and is the first one to be deleted. In other words, when an element with a priority higher than the priorities of all the elements currently in the queue enters the queue, it will be moved to the front of the queue.

In our application, we can set the priority of a node (root of a tree) to be the reciprocal of its weight. That is, the lower the weight, the higher the priority. As an example, in Figure 8-2, node **e** has the highest priority, followed by node **d**, and node **a** has the lowest priority. With this association, a Huffman tree can be constructed by the following steps:

1. Start with a forest consisting of single-node trees; the root of a tree contains a symbol and its weight, which is the reciprocal of its priority value.
2. Insert the roots (nodes) of all trees of the forest into a priority queue.
3. Delete two nodes from the priority queue; the two deleted nodes always have the least weights (highest priorities).
4. Merge the deleted nodes to form a new root with combined weights; insert the new root back to the priority queue.
5. Repeat steps 3 - 4 until the priority queue is empty.

In the above steps, when the priority queue is empty, a single tree is formed and it is the required Huffman tree. In the process, we assume that a root always links to the rest of the nodes of the tree.

The priority queue implementation is straightforward and intuitive. However, it is not the most efficient method. and we shall not use it here. To make more efficient or customized implementations, it is helpful to learn some properties of a Huffman tree.

Firstly, we have learned that a Huffman tree is optimal. However, it is not unique. Given a set of frequencies, we can have more than one Huffman tree that yields an optimal average codeword length; different trees can be constructed by interchanging the assignment of 0 and 1 to the left and right traversal or by merging roots with equal weights in different orders.

Secondly, all internal nodes (non-leaves) of a Huffman tree always have two children. The binary tree that represents Code 1 in Figure 8-1 will never occur in a Huffman tree regardless of the occurrence frequencies of the symbols.

Thirdly, if the weights of the symbols are changing, the Huffman tree needs to be recomputed dynamically. This can be done by utilizing the *Sibling Property*, which defines a binary tree to be a Huffman tree if and only if:

1. all leaf nodes have non-negative weights,
2. all internal nodes have exactly two children,
3. the weight of each parent node is the sum of its childrenś weights, and
4. the nodes are numbered in increasing order by non-decreasing weight so that siblings are assigned consecutive numbers or rank, and most importantly, their parent node must be higher in the numbering.

The *Sibling Property* is usually used in *Dynamic Huffman Coding*, where we encode a stream of symbols on the fly and the symbol statistics changes as we read in more and more symbols.

Finally, we shall prove a lemma concerning binary trees to help us simplify the implementation of a Huffman tree when we use an array to implement it. Consider a binary tree where

n_0 = number of leaves (nodes of degree 0)
n_1 = number of nodes of degree 1 (nodes having one child)
n_2 = number of nodes of degree 2 (nodes having two children)

Lemma:

For a non-empty binary tree,

$$n_0 = n_2 + 1 \tag{8.4}$$

Proof:

The total number of nodes in the tree is
$$n = n_0 + n_1 + n_2 \tag{8.5}$$

Except the root, a node always has a branch leading to it. Thus the total number of branches is
$$n_B = n - 1 \tag{8.6}$$

But all branches stem from nodes of degree 1 or 2, so
$$n_B = n_1 + 2 \times n_2 \tag{8.7}$$

Combining (8.6) and (8.7), we have
$$n - 1 = n_1 + 2 \times n_2 \tag{8.8}$$

yielding
$$n_0 + n_1 + n_2 - 1 = n_1 + 2 \times n_2 \tag{8.9}$$

Simplifying (8.9), we obtain
$$n_0 = n_2 + 1$$

which is the result we want to prove.

From Huffman tree properties discussed above, we know that a Huffman tree does not have any node of degree 1 (i.e. $n_1 = 0$) and thus the number of internal nodes is equal to n_2. Also, the number of symbols is equal to the number of leaves (n_0) in the tree. Therefore, if we have n symbols, we know from the Lemma that the corresponding Huffman tree will have $n_2 = n_0 - 1 = n - 1$ internal nodes. To store a Huffman tree, we need an entry for each of the left and right child pointers and an entry for each symbol. The total number of entries N_T in the table that holds the Huffman Tree is equal to the number of pointers plus the number of symbols; this is given by

$$N_T = 2 \times (n - 1) + n = 3 \times n - 2 \tag{8.10}$$

For instance, consider a Huffman tree consisting of five symbols: a, b, c, d, e as shown in Figure 8-7.

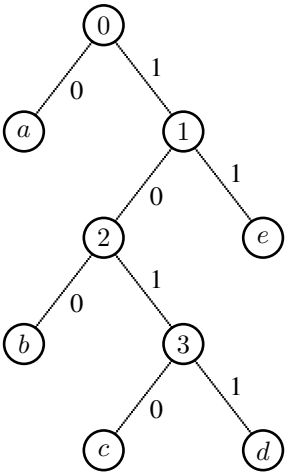


Figure 8-7. A Huffman Tree Consisting of Five Symbols

The following table shows the corresponding Huffman code of Figure 8-7.

Symbol	Codeword
a	0
b	100
c	1010
d	1011
e	11

In this example, the size of the table N_T that implements the Huffman tree is

$$N_T = 3 \times n - 2 = 3 \times 5 - 2 = 13 \tag{8.11}$$

The Huffman tree is represented by the following table (Table 8-1), where traversing left gives a 0, and traversing right gives a 1 and $N_T = 13$. Note that in the table, the root is pointing at the highest table location ($N_T - 1$) and when traversing the tree we move from the top of the table down to a location that contains a symbol. When we reach a location whose index is smaller than N_T, we know that we have reached a terminal node (leaf) containing a symbol.

Table 8-1

Table Index	Table Content	Comments
12	0	left child of node 0 (root)
11	10	right child of node 0
10	8	left child of node 1
9	4	right child of node 1
8	1	left child of node 2
7	6	right child of node 2
6	2	left child of node 3
5	3	right child of node 3
4	'e'	symbol at leaf
3	'd'	symbol at leaf
2	'c'	symbol at leaf
1	'b'	symbol at leaf
0	'a'	symbol at leaf

The following piece of C-like pseudo code shows how we traverse the table to obtain the symbols; in the code, htree[] is the table containing the Huffman tree:

```
loc = 3 * N - 3;              //start from root, N = # of symbols
do {
  loc0 = loc;                 //in is data pointer pointing to
                              //  encoded data
  if ( read_one_bit( in ) == 0 ) //a 0, go left
      loc = htree[loc0];
  else
      loc =  htree[loc0+1];   //a 1, go right

} while ( loc >= N );         //traverse until reach leaf
return htree[loc];            //return symbol
```

Table 8-1 can be simplified if we assert that the number of symbols n is smaller than a certain value and we associate each symbol with a value in the range 0 to $n-1$. For example, if we assert $n \leq 128$, which actually applies to many video compression applications, then

1. each pointer can be represented by a byte, with a left child denoted by the upper byte of a 16-bit word and right child by the lower byte,
2. table locations signify symbol values and do not need extra entries to hold the symbols, and
3. table size N_T is reduced to $n-1$.

The above Huffman tree example where the number of symbols is 5 can now be represented by a table with size of $5-1=4$ as shown below.

Table 8-2

Table Index (Symbol)	Left Child	Right Child	Comments
3 (8)	0 (a)	7	left, right children of node 0 (root)
2 (7)	6	4 (e)	left, right children of node 1
1 (6)	1 (b)	5	left, right children of node 2
0 (5)	2 (c)	3 (d)	left, right children of node 3

Table 8-2 only has 4 entries while Table 8-1 has 13 entries. The symbols are represented by the table indices with 0 representing 'a', 1 representing 'b' and so on. To resolve the case if a table entry holds a pointer or an actual symbol value, we've added the value N_T to all table indices before saving them as pointers. Therefore, in a table entry, if the pointer value is smaller than N_T, we know that it is a terminal node (symbol). The following piece of code shows how to decode such a table that represents a Huffman tree; a left mask and right mask are used to extract the correct pointer value.

```
                              //N = # of symbols
left_mask = 0xFF00;           //to extract upper byte(left child)
right_mask = 0x00FF;          //to extract lower byte(right child)
loc = ( N - 1 ) + N;          //start from root; add offset N to
                              // distinguish pointers from symbols
do {
  loc0 = loc - N;             //loc0 is real table location
  if ( read_one_bit( in ) == 0 ){  //a 0, go left
      loc = ( htree[loc0] & left_mask ) >> 8;
 } else{
      loc =  htree[loc0] & right_mask;
 }
} while ( loc >= N );         //traverse until reaches leaf
return loc;                   //symbol value = loc
```

8.4 Pre-calculated Huffman-based Tree Coding

The Huffman coding process has a disadvantage that the statistics of the occurrence of the symbols must be known ahead of the encoding process. Though we do not need to transmit the probability table to the decoder, we do need it before we can do any encoding. The probability table for a large video cannot be calculated until after the video data have been processed which may introduce unacceptable delay into the encoding process. Because of these, practical video coding standards define sets of codewords based on the probability distributions of generic video data. The following example is a pre-calculated Huffman table taken from MPEG-4 Visual (Simple Profile), which uses 3D run-level coding discussed before to encode quantized coefficients. A total of 102 specific combinations of (*run, level, last*) have variable-length codewords assigned to them and part of these are shown in Table 8-3. Each codeword can be up to 13 bits long and the last bit is the sign bit 's', which indicates if the decoded coefficient is positive (0) or negative (1). Any (*run, level, last*) combination that is not listed in the table is coded using an escape sequence; a special ESCAPE code of 0000011 is first transmitted followed by a 13-bit fixed-length codeword describing the values of *run, level*, and *last*. A valid codeword cannot contain more than eight consecutive zeros. Therefore, a sequence consisting of eight or more consecutive zeros, "00000000..." indicates an error in the encoded bitstream or possibly a start code, which might contain a long sequence of zeros. We shall use Table 8-3 in our entropy-encoding stage shown in Figure 7-1. The full implementation of the Huffman encoding and decoding for our video codec is discussed in the next section.

Table 8-3

Run	Level	Last	Code
0	1	0	10s
1	1	0	110s
2	1	0	1110s
0	2	0	1111s
0	1	1	0111s
3	1	0	01101s
4	1	0	01100s
5	1	0	01011s
0	3	0	010101s
1	2	0	010100s
6	1	0	010011s
7	1	0	010010s
8	1	0	010001s
9	1	0	010000s
1	1	1	001111s
2	1	1	001110s
3	1	1	001101s
4	1	1	001100s
0	4	0	0010111s
10	1	0	0010110s
11	1	0	0010101s
12	1	0	0010100s
5	1	1	0010011s
6	1	1	0010010s
7	1	1	0010001s
8	1	1	0010000s
ESCAPE			0000011s
..	..	..	..

8.5 Huffman Coding Implementation

Because Huffman coding involve reading and writing one bit at a time, we need to first develop some functions that can process an arbitrary number of bits of a file. We provide the program "fbitios.cpp" along with its header file "fbitios.h" that has this capability. The files can be downloaded from this book's web site at *http://www.forejune.com/vcompress*. We do not intend to discuss the details of this program as it does not directly relate to video compression. All we need to know is how to use it, which is straightforward. The following are the prototypes of the functions that have been grouped to a class that does the job that processes files on a bit-basis.

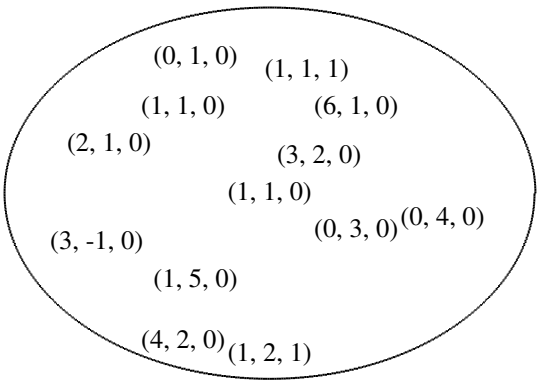
```
class  BitFileIO{
public:
  BitFileIO ( char *argv_in,  char *argv_out );   //constructors
  BitFileIO :: BitFileIO( char *argv_out, int in_out );
  int  inputBit();                        //input one bit from file
  void outputBit( int bit );              //output one bit to file
  long inputBits( int n );                //input n bits from file
  void outputBits(unsigned long data, int n);//send n bits to file
  void closeOutput();                     //close_output
  void closeInput();                      //close input
};
```

The constructor **BitFileIO** (char **argv_out*, int *in_out*) opens the file specified by *argv_out* for input or output depending on the value of *in_out*. If *in_out* is 1, the file is opened for input, otherwise

it is for output. The main purpose of the Huffman code here is to encode the run-level codewords of the DCT coefficients after forward quantization and reordering. Recall that we have defined a class to represent a 3D run-level codeword:

```
class Run3D {
public:
  unsigned char run;
  short level;
  char  last;
};
```

Each Run3D object represents a run-level codeword, and the Huffman code is used to encode these codewords. We define a class called *RunHuff* that will help encode and decode run-level codewords with a Huffman code. This class "has-a" Run3D class. As you'll see, we'll insert *RunHuff* objects into a **set**, which involves the ordering of nodes. **Sets** and **bags** are data structures that are useful in problems in which multiple collections naturally occur. A **set** is an unordered collection of elements in which each element is unique like the figure below, which shows a set of 3D run-level codewords.



On the other hand, a **bag** can have duplicate elements. Actually, the concept of a **set** underlies much of mathematics and is as well an integral part of many computing algorithms. The fundamental operations of a set include adding and removing elements, testing for inclusion of an element, and forming unions, intersections and differences of other sets.

Theoretically, a set is a collection of unordered elements; there is no key associated with each element and thus we cannot order the elements in a set. However, the people who wrote the STL C++ library have modified (or wrongly interpreted) the definition of a set and allow an element associate with a key. Therefore, the elements in an STL C++ set can be ordered. The modified set behaves more like a **map**. **Maps** and **multimaps** are data structures that are useful in problems in which multiple collections naturally occur.

A **map**, sometimes also referred to as a dictionary or a table is an associative container where records (data) are specified by key values. It is an indexed collection; the key can be used to generate an index to access the corresponding record. A map can be considered as a collection of associations of key and value pairs. The key is used to find the value as shown below:

$$
\begin{aligned}
&key_1 \rightarrow value_1 \\
&key_2 \rightarrow value_2 \\
&key_3 \rightarrow value_3 \\
&.... \\
&key_n \rightarrow value_n
\end{aligned}
$$

For example, we can use the telephone number as the key to lookup the information of a person. The keys in a map must be ordered and unique. A multimap is similar to a map except that a multimap permits multiple entries to be accessed using the same key value. (i.e. The keys in a multimap do not need to be unique.)

To utilize an STL C++ set (which behaves like a map) properly, we have to define the comparison operator "<" so that *RunHuff* objects (nodes) can be ordered. In this case, we order the objects using the values of (run, level, last) of the *Run3D* objects. We first compare the runs of the two objects, whichever is smaller determines the smaller *RunHuff* object. If the runs are equal, we compare levels and so on. The data member index will point to the table location where the run-level codeword will be saved. Note that in the STL C++ set data abstraction, the equality testing operator (operator ==) is not used to test objects for equality. Instead, two objects X and Y are considered to be equal when "$X < Y$" and "$Y < X$" are both **false**. Each element in a set is unique. If one tries to insert an element which is already in the set, the "insert" operation will be ignored. The class *RunHuff* is defined below.

Program Listing 8-1: *RunHuff* Class

```
-------------------------------------------------------------------------
class RunHuff {
public:
  run3D r;
  unsigned int codeword;
  char hlen;            //length of the Huffman codeword
  short index;          //table index where codeword saved
  RunHuff() {}          //constructors
  RunHuff ( run3D a, unsigned c, char len, short idx )
  {
    r = a, codeword = c, hlen = len; index = idx;
  }
  //'<' operator is to order run-level tuples so that
  //  they can be saved in a binary tree ( set )
  friend bool operator < ( RunHuff left, RunHuff right ) {
    if ( left.r.run < right.r.run )
      return true;
    if ( left.r.run >  right.r.run )
      return false;
    //runs equal
    if ( left.r.level < right.r.level )
      return true;
    if ( left.r.level > right.r.level )
      return false;
    //both runs and levels equal
    if ( left.r.last > right.r.last )
      return true;
    return false; //so, left object is not smaller than the right
  }
};
-------------------------------------------------------------------------
```

We assume that a pre-calculated Huffman Table like the one shown in Table 8-3 is provided. Listing 8-2 presents the class *Hcodec* that makes use of a pre-calculated Huffman Table to build the encoder and the decoder. We declare a variable *htable* to be a map to collect *RunHuff* objects, each of which contains a 3D run-level tuple and the corresponding Huffman codeword (Table 8-2).

Program Listing 8-2: *Hcodec* class for Huffman Coding-Decoding

```
-------------------------------------------------------------------------
class Hcodec
{
private:
  const static int NSymbols = 256;   //maximum symbols allowed
  const static short ESC = 127;      //Escape run
  const static short ESC_HUF_CODE = 0x60;  //Escape code
  const static short EOS = 126;      //End of Stream 'symbol' ( run )
  const static short EOS_HUF_CODE = 0x0a;  //End of Stream 'symbol'
  bool tableNotBuilt;
  set<RunHuff> htable;

public:
  //default constructor
  Hcodec();
  //use a set ( htable ) to collect all pre-calculated run-level
  //  and Huffman codewords
  void build_htable ();
  void escape_encode ( BitFileIO *outputs, Run3D &r );
  void huff_encode ( Run3D runs[], BitFileIO *outputs );
  //Encode End of Stream symbol
  void huff_encode_EOS ( BitFileIO *outputs );
  short huff_decode( BitFileIO *inputs, Dtables &d,  Run3D runs[] );
};
-------------------------------------------------------------------------
```

The function **build_htable()** of the class *Hcodec* has all codewords and run-level tuples hard-coded in the code and saved in variables *hcode, runs, levels,* and *last* respectively. The function collects all these pre-calculated run-level tuples and Huffman codewords and saves them in the set *htable*. We can conveniently do this using the C++ Standard Template Library (STL) **set**. We use the member function **insert()** of the **set** class to insert all *RunHuff* objects into the set as shown below. (As a matter of fact, the STL **set** class is implemented using a balanced binary tree structure, which gives efficient search and insert operations. This is why we need to define a comparison operator in *htable* in order to use the STL **set** since a binary tree is fully ordered.) In the function, the special run value 127 is used to represent the ESC (escape) symbol. Listing 8-3 shows the code of this function:

Program Listing 8-3: Function build_htable() of Class *Hcodec*

```
-------------------------------------------------------------------------
//htable is a set of RunHuff objects
void Hcodec::build_htable() {
    //N = number of pre-calculated codewords with positive levels
    //  In practice, N should be larger than 100
    short i, j, k, N = 11;
    //lengths of Huffman codewords (not including sign-bit)
    char hlen[] = { 2, 3, 4, 4, 4, 5, 5, 5, 6, 6,  7 };
    //Huffman codewords, 0x60 for ESC, 0x3a for EOS;
    //  a codeword should NEVER be a prefix of another
    unsigned short hcode[] = {0x01,0x3,0x7,0xf,0xe, 0x16, 0x6, 0x1a,
                                    0x2a, EOS_HUF_CODE, ESC_HUF_CODE};
    //data of 3D run-level tuples ( codewords )
    unsigned char runs[] = {0, 1, 2, 0, 0, 3, 4, 5, 0, EOS, ESC};
```

```
    //Note: don't set Escape level to 0, otherwise leads to duplicate key
    short levels[] = {1, 1, 1, 2, 1, 1, 1, 1, 3, 1, 1};
    char  lasts[] = {0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0 };
    Run3D r;              //a 3D run-level codeword ( tuple )
    RunHuff rf[128];      //table containing RunHuff objects
    //inserting RunHuff objects into rf[]
    k = 0; j = 0;
    for ( i = 0; i < N; i++ ) {
      r.run = runs[i];
      r.level = levels[i];
      r.last = lasts[i];
      //construct a RunHuff object, positive level, so sign=0
      rf[k++] =  RunHuff ( r, hcode[i] << 1,  hlen[i], j++ );
      //do the same thing for negative level, sign = 1
      r.level = (short) -r.level;
      rf[k++] = RunHuff ( r, (short)((hcode[i]<<1) | 1), hlen[i], j++ );
    }
    k = 2 * N;            //insert all 2N RunHuff objects into htable
    for ( i = 0; i < k; ++i )
      htable.insert ( rf[i] );
    tableNotBuilt = false;
}
-------------------------------------------------------------------------
```

After we have collected all the pre-calculated run-level Huffman codewords in the **set** *htable*, the encoding of 3D run-levels tuples becomes simple. To encode a run-level codeword, all we need to do is to lookup the **set** *htable*; if the run-level codeword is in the set, we output the Huffman codeword along with the sign-bit; if it is not in the set, we "escape" and output the run-level codeword "directly".

Suppose the array *runs*[] contains all the run-level codewords of a macroblock; the following piece of code shows how to encode them using the pre-calculated Huffman codewords saved in *htable*. In the code, we define an iterator *itr* to traverse *htable*. If it finds the run-level tuple in *htable*, it outputs the Huffman codeword and the sign-bit, otherwise it escape-encodes the tuple by first outputting the ESCAPE code followed by a fixed-length codeword for the tuple. (In C++, an iterator is an object that can access members of an array or a container class. It has the ability to traverse the elements in a certain range using certain operators such as increment (++) and dereference (*). Very often, it is used in a manner similar to pointers.). The following function of Listing 8-4, **huff_encode()** does the job:

Program Listing 8-4: Function huff_encode() of Class *Hcodec*

```
-------------------------------------------------------------------------
void Hcodec::huff_encode ( Run3D runs[], BitFileIO *outputs )
{
  short i, j, k;
  set<RunHuff>::iterator itr;          //iterator to traverse htable

  if ( tableNotBuilt )
      build_htable();
  i = 0;  k = 0;
  while ( i < 64 ) {                 //a macroblock has at most 64 samples
    RunHuff rf ( runs[k], 0, 0, 0 ); //construct a RunHuff object;
                                     //  only runs[k] is relevant
    if ( (itr = htable.find ( rf )) != htable.end() )   //found
```

```
      outputs->outputBits ( itr->codeword, itr->hlen + 1);
    else                                                    //not found
      escape_encode( outputs, rf.r );//need to 'escape encode' 3D tuple
    if ( runs[k].last ) break;        //end of run-level codewords
    i += ( runs[k].run + 1 );      //for handling the special case of whole
                                   //  block of run being 0
    k++;
  }
}
-------------------------------------------------------------------------
```

To encode a run-level tuple using the ESC symbol, we first check if the level is negative. If it is negative, we output a '1' bit otherwise we output a '0' bit. We then send the special code for the ESC symbol followed by the binary numbers representing the values of run, level and last. The following piece of code shows how to do this precisely.

```
void Hcodec::escape_encode ( BitFileIO *outputs, Run3D &r ) {
  if ( r.level < 0 ) {                 //value of level negative
    outputs->outputBit ( 1 );           //output sign-bit first
    r.level = -r.level;                //change level to positive
  } else
    outputs->outputBit ( 0 );           //value of level negative
  outputs->outputBits(ESC_HUF_CODE, 7); //ESCAPE code
  if ( r.run == 64 ) r.run = 63;       //r.level differentiates
                                       // whether last element nonzero
  outputs->outputBits ( r.run, 6 );     //6 bits for run value
  outputs->outputBits ( r.level, 8 );   //8 bits for level value
  outputs->outputBit ( r.last );        //1 bit for last value
}
```

The functions **huff_encode**() and **escape_encode**() encode the 3D run-level tuples of a macroblock. Also, at the end of the encoding process, we need to encode a special symbol 'End of Stream' (EOS) to indicate that there are no more encoding data. When the decoder sees the symbol EOS, it knows that the decoding process is done.

In the encoding process, the Huffman tree is not needed. However, to decode the encoded bitstream, we have to use the Huffman tree to recover the 'symbols' (run-level tuples). We can easily construct the Huffman tree in the form of a table from the **set** *htable*. Note that here we build the Huffman tree from pre-calculated codewords, not from symbol weights as people normally do. Therefore, the process is a lot simpler as Huffman codewords have been provided. The following function **build_huff_tree**() builds the Huffman tree from *htable* and saves it in the table (array) *huf_tree*[] of data type short. An entry of *huf_tree*[] holds a node's pointer to a child or an index ('symbol') if the node is a leaf; the index points to an entry of another table,*run_table*[], which contains the actual run-level tuple (see Table 8-2). For convenience of programming, we put *huf_tree*[] and *run_table*[] in a public class (i.e. a struct) called *Dtables*. (The 'D' stands for 'Decoding'.) Listing 8-5 shows the class *Dtables* and the function build_huff_tree().

Program Listing 8-5: Constructing Huffman Tree

```
-------------------------------------------------------------------------
class Dtables {
public:
  short huf_tree[1024];    //table containing Huffman Tree
  Run3D run_table[512];    //table containing run-level codewords
};
```

```
void Hcodec::build_huff_tree ( Dtables &d )
{
  set<RunHuff>::iterator itr;     //iterator to traverse set htable

  short i, j, n0, free_slot, loc, loc0, root, ntotal;
  unsigned int mask, hcode;

  n0 = NSymbols;                  //number of symbols (# of leaves in tree)
  ntotal = 2 * n0 - 1;            //Huffman tree has  n0-1  internal nodes
  root = 3 * n0 - 3;              //location of root with offset n0 added
  free_slot = root - 2;           //next free table entry for filing in
                                  //  with a pointer or an index
  for ( i = 0; i < ntotal; ++i )  //initialize the table
    d.huf_tree[i] = -1;           //all entries empty
  for ( itr = htable.begin(); itr != htable.end(); ++itr ) {
     if ( itr->r.level < 0 ) continue; //only save positive levels
     //save run-level codeword;divide by 2 as only postive levels saved
     d.run_table[itr->index/2]=itr->r;
     loc = root;                  //always start from root
     mask = 0x01;                 //for examining bits of Huffman codeword
     hcode = itr->codeword >> 1;  //the rightmost bit is sign-bit
     for ( i = 0; i < itr->hlen; ++i ){ //traverse the Huffman codeword
         loc0 = loc - n0;               //everything shifted by offset n0

        if ( i == ( itr->hlen - 1 ) ){  //last bit, should point to leaf
          if ( (mask & hcode) == 0 )    //a 0, save it at 'left' leaf
            d.huf_tree[loc0] = itr->index/2;
          else                          //a 1, save it at 'right' leaf
            d.huf_tree[loc0-1] = itr->index/2;
          continue;               //get out of for i for loop,next codeword
        }
        if ( (mask & hcode) == 0 ){     //a 0 ( go left )
          if (d.huf_tree[loc0] == -1){  //slot empty
            d.huf_tree[loc0]=free_slot; //point to left new child
            free_slot -= 2;             //next free table entry
          }                             //else : already has left child
          loc = d.huf_tree[loc0];       //follow the left child
        } else {                        //a 1 ( go right )
          if (d.huf_tree[loc0-1]== -1){ //slot empty
            d.huf_tree[loc0-1]=free_slot;//point to right new child
            free_slot -= 2;
          }                             //else: already has right child
          loc = d.huf_tree[loc0-1];     //follow the right child
        }
        mask <<= 1;                     //consider next bit
     } //for i
  } //for itr
}
-------------------------------------------------------------------------
```

After we have built the Huffman tree, decoding becomes simple. We read in a bitstream and traverse the tree starting from the root. If the bit read is a 0, we traverse left, otherwise we traverse right until we reach a leaf where we recover a symbol. If the symbol is an ESCAPE code, we need to further read in a fixed-number of bits to determine the 'symbol' (the run-level tuple). Then we read in another bit and start the tree-traversal from the root again. The details are shown in the

function **huff_decode**() listed below.

Program Listing 8-6: Huffman Decoder

```
-------------------------------------------------------------------------
short Hcodec::huff_decode( BitFileIO *inputs, Dtables &d,  Run3D runs[] )
{
  short n0, loc, loc0, root, k;
  char c, sign;
  bool done = false;
  Run3D r;

  n0 = NSymbols;                          //number of symbols
  root = 3 * n0 - 3;                      //points to root of tree
  k = 0;
  while ( !done ) {
    loc = root;                           //starts from root
    sign = inputs->inputBit();            //sign-bit
    do {
      loc0 = loc - n0;
      c = inputs->inputBit();             //read one bit
      if (c < 0) {done = true; break;}    //no more data, done
      if ( c == 0 )                       //a 0, go left
        loc = d.huf_tree[loc0];
      else                                //a 1, go right
        loc = d.huf_tree[loc0-1];
    } while ( loc >= n0 );                //traverse until reaches leaf
    r = d.run_table[loc];
    if ( r.run == ESC ) {  //ESCAPE code, read actual run-level tuple
        r.run  = inputs->inputBits ( 6 );    //read 6 bits for run
        r.level  = inputs->inputBits ( 8 );  //read 8 bits for level
        r.last = inputs->inputBit();         //read 1 bit for last
        if ( sign )           //if sign is 1, level should be negative
          r.level = -r.level;
     } else {                                //not ESCAPE code
        if ( sign )                          //1 => negative
         r.level = -r.level;
    }
    if ( (r.run == 63) && (r.level == 0) ) r.run = 64;  //whole block 0
    runs[k++] = r;                           //save tuple in table runs[]
    if ( r.last )                            //end of macroblock
      break;
  }  //while
  if ( done ) return -1;                    //if (done) => no more data
  else return 1;
}
-------------------------------------------------------------------------
```

Putting all these together, we provide two driver programs,

```
test_huf_encode.cpp, and
test_huf_decode.cpp
```

for you to do the testing of the concepts discussed above. (The programs can be downloaded from the web site of this book, *http://www.forejune.com/vcompress/*.) A Makefile is provided for you to create the executables **test_huf_encode** and **test_huf_decode**. You may use the command,

```
$test_huf_encode ../data/beach.dct ../data/beach.huf
```

to encode the DCT coefficients of the file “beach.dct” and save the Huffman codeword bitstream in “beach.huf”. The command

```
$test_huf_decode ../data/beach.huf t.dec
```

decodes the Huffman codeword bitstream saved in “beach.huf” and saves the decoded DCT coefficients in the file “t.dec”. The original data file “beach.dct” and recovered data file “t.dec” are not the same as quantization and rounding errors have occurred in the process.