Huffman Codes

- Huffman encoding
  1. Order the symbols according to their probabilities

     alphabet set : $S_1, S_2, \ldots, S_N$

     prob. of occurrence : $P_1, P_2, \ldots, P_N$

     $\Rightarrow$ the symbols are rearranged so that

     $P_1 \geq P_2 \geq \ldots \geq P_N$
  2. Apply a contraction process to the two symbols with the smallest probabilities

     replace symbols S_{N-1} and S_N by a "hypothetical" symbol, say H_{N-1}, that has a prob. of occurrence $P_{N-1}+P_N$

     the new set of symbols has N-1 members:

     $S_1, S_2, \ldots, S_{N-2}, H_{N-1}$
  3. Repeat the step 2 until the final set has only one member.

43

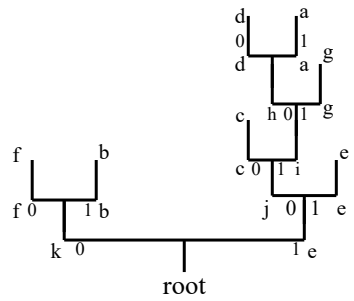
- The recursive procedure in step 2 can be viewed as the construction of a binary tree, since at each step we are merging two symbols.

  At the end of the recursion, all the symbols $S_1, S_2, \ldots, S_N$ will be leaf nodes of the tree.

The codeword for each symbol S_i is obtained by traversing the binary tree from its root to the leaf node corresponding to S_i

44

s_i	$p(s_i)$	*Step*1	*Step*2^1	*Step*2^2	*Step*2^3	*Step*2^4	*Step*2^5
a	0.05	e 0.3	e 0.3	e 0.3	e 0.3	k 0.4	*l* 0.6 (1.0)
b	0.2	b 0.2	b 0.2	b 0.2	j 0.3	ⓔ 0.3 (0.6)	k 0.4 (root)
c	0.1	f 0.2	f 0.2	f 0.2	ⓑ 0.2 (0.4)	j 0.3 (e)	
d	0.05	c 0.1	c 0.1	i 0.2 (0.3)	ⓕ 0.2 (k)		
e	0.3	g 0.1	ⓖ 0.1 (0.2)	ⓒ 0.1 (j)			
f	0.2	ⓐ 0.05 (0.1)	h 0.1 (i)				
g	0.1	ⓓ 0.05 (h)					



	C(w)
a	**10101**
b	**01**
c	**100**
d	**10100**
e	**11**
f	**11**
g	**1011**

45

- Average codeword length

$$lave \overset{\Delta}{=} \sum_i l_i p_i$$

lave is a measure of the compression ratio.

In the above example,

7 symbols ⇒3 bits fixed length code representation

lave(Huffman) = 2.6 bits

Compression ration = 3/2.6 = 1.15

46

Properties of Huffman codes

- Fixed-length symbols → variable-length codewords : error propagation
- H(s) ≤ lave < H(s)+1
  H(s) ≤ lave < P+0.086
  where P is the prob. Of the most frequently accurring symbol. The equality is achieved when all symbol probs. Are inverse powers of two.
- The Huffman code-tree can be constructed both by
  bottom-up method ← in the above example
  top-down method

47

- The code construction process has a complexity of $O(N\log_2 N)$. With presorting of the input symbol probs, code construction method with complexity O(N) has been presented in IEEE trans. ASSP-38, pp. 1619-1626, Sept. 1990.
- Huffman codes satisfy the prefix-condition : uniquely decodable no codeword is a prefix of another codeword.
- If l_i satisfy the Kraft constraint $\sum 2^{-l_i} \le 1$ then the corresponding codewords can be constructed as the first l_i bits in the fractional representation of a_i

$$a_i = \sum_{j=1}^{i-1} 2^{-l_j}, \qquad i = 1,2,\cdots,N$$

- The complement of a binary Huffman code is also a valid Huffman code.

48

Huffman Decoding

- Bit-Serial Decoding : fixed input bit rate — variable output symbol rate

  (Assume the binary coding tree is available to the decoder)
  In practice, Huffman coding tree can be reconstructed from the symbol-to-codeword mapping table that is known to both the encoder and the decoder

Step 1:

Read the input compressed stream bit-by-bit and traverse the tree until a leaf node is reached.

Step 2:

As each bit in the input stream is used, it is discarded. When the leaf node is reached, the Huffman decoder outputs the symbol at the leaf node. This completes the decoding for this symbol.

Step 3:

Repeat step1 and 2 until all the input is consumed.

- Since the codeword is variable in length, the decoding bit rate is not same for all symbols.

49

- Lookup-table-Based Decoding : constant decoding rate for all symbols — variable input rate

  The look-up table is constructed at the decoder from the symbol-to-codeword mapping table. If the longest codeword in this table is L bits, then a 2^L entry lookup table is needed. : space constraints image/video longest L = 16 to 20.

Look up table construction:

- Let C_i be the codeword that corresponds to symbol S_i.Assume C_i has l_i bits. We form an L-bit address in which the first l_i bits are C_i and the remaining L- l_i bits take on all possible combinations of "0" and "1". Thus, for the symbol s_i, there will be 2^{L-l_i} addresses.
- At each entry we form the two-tuple (s_i, l_i).

50

Decoding Processes:

1. From the compressed input bit stream, we read in L bits into a buffer.
2. We use the L-bit word in the buffer as an address into the lookup table and obtain the corresponding symbol, say s_k. Let the codeword length be l_k. We have now decode one symbol.
3. We discard the first l_k bits from the buffer and we append to the buffer, the next l_k bits from the input, so that the buffer has again L bits.
4. Repeat step 2 and 3 until all of the symbols have been decoded.

51

Memory Efficient and High-Speed Search Huffman Coding, by R. Hashemian IEEE trans. Communications, Oct. 1995, pp. 2576-2581

- Due to variable-length coding, the Huffman tree gets progressively sparse as it grow from the root
  - Waste of memory space
  - A lengthy search procedure for locating a symbol

Ex: if K-bit is the longest Huffman code assigned to a set of symbols, the memory size for the symbols may easily reach 2^K words in size.

→ It is desirable to reduce the memory size from typical value of 2^K, to a size proportional to the number of the actual symbols.

  - Reduce memory size
  - Quicker access

52

- Ordering and clustering based Huffman Coding

  groups the codewords (tree nodes) within specified codeword lengths
- Characteristics of the proposed coding cheme:
  1. The search time for more frequent symbols (shorter codes) is substantially reduced compare to less frequent symbols, resulting in an overall faster response.
  2. For long codewords the search for the symbol is also speed up. This is achieved through a specific partitioning technique that groups the code bits in a codeword, and the search for a symbol is conducted by jumping over the groups of bits rather than going through the bit individually.
  3. The growth of the Huffman tree is directed toward on side of the tree.
  – Single side growing Huffman tree (SGH-tree)

53

Ex: H=(S, P)

$S=\{S_1, S_2,\ldots, S_n\}$
$P=\{P_1, P_2,\ldots, P_n\}$
└→No. of occurrence

TABLE I
Reduction Process In The Source List

s_1	48	s_1	48	s_1	48	s_1	48	s_1	48	a_5	52
s_2	31	s_2	31	s_2	31	s_2	31	s_2	31	s_1	48
s_3	7	s_3	7	a_2	8	a_3	13	a_4	21		
s_4	6	s_4	6	s_3	7	a_2	8				
s_5	5	s_5	5	s_4	6						
s_6	2	a_1	3								
s_7	1										

Merge
Insert (in descending order)

- For a given source listing H, the table of codeword length uniquely groups the symbols into blocks, where each block is specified by its codeword length (CL).

54

TABLE II
TABLE OF CL-RECORDING

s_i	s_{i-1}	CL
s_7	s_6	5
a_1	s_5	4
s_4	s_3	4
a_2	a_3	3
a_4	s_2	2
s_1	a_5	1

TABLE III
TABLE OF CODEWORD LENGTHS (TOCL)

CL	symbols
1	s_1
2	s_2
3	
4	s_3, s_4, s_5
5	s_6, s_7

CL: codeword length

- Each block of symbols, so defined, occupies one level in the associated Huffman tree.

55

- Algorithm 1: Creating a Table of Codeword Lengths
  1. The symbols are listed with the probabilities in decending order (the ordering of the symbols with equal probabilities is assumed indifferent).
     Next, the pair of symbols at the bottom of the ordered list are merged and as a result a new symbol a_1 is created. The symbol a_1, with probability equal to the sum of the probabilities of the pair, is then inserted at the proper location in the **ordered list**.
     To record this operation a codeword length recording (CLR) table is created which consists of three columns:
     : Columns 1 and 2 hold the last pair of symbols before being merged, and column 3, initially empty, is identified as the codeword length (CL) column (Table II).

56

In order to make the size of the CLR table small and the hardware design simpler, the new symbol a_1 (in general a_j) is selected such that its inverse $\overline{a_1}\,(\overline{a_1}=a_1 \cdot \overline{a_j})$ represents the associated table address.

Composite symbol

e.g. For an 8-bit address word,

$a_1 \rightarrow$ The first row in the CLR table, is given the value of 1111 1110

$\overline{a} \rightarrow$ 0000 0001

$a_2 \rightarrow$ 1111 11101 ($\overline{a}_2 \rightarrow$ 0000 0010)

- The sign of a composite symbol is different from an original symbol.

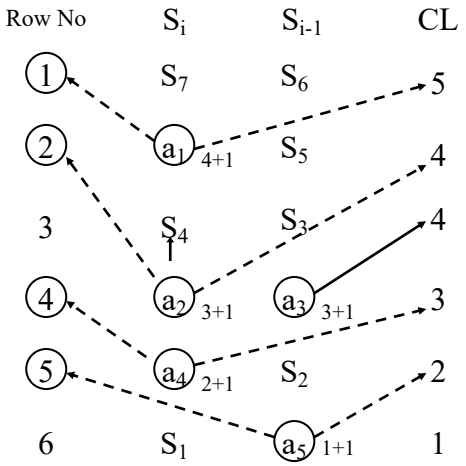
→ A dedicated sign (MSB) detector is all that is needed to distinguish between an original symbol and a composite symbol.

57

2. Continue applying the same procedure, developed for a single row in step 1, and construct the entire CLR table. Note that table || contains both the original symbol s_i and the composite ones a_j (carrying opposite signs).
3. The third column in Table II, designated by CL, is assigned to hold the codeword lengths. The fill up this column we start from the last row in the CLR and enter 1. This designates the codeword length for both s_i and a_5.
   Next, we check for the signs of each s_i and a_5; if positive (MSB = 0) we skip, otherwise, the symbol is a composite one (a_5)and its binary inverse ($\overline{a}_5$=00…00101)
   is a row address for table II. We now increment the number in the CL column, and assign a new value (2 in this example) to the CL column in row $\overline{a}_j$ (5 in this example), and proceed applying.

58

- The same operation to other rows in the CLR table, as we move to the top, until the CL column is completely filled.



59

role:

(i) S_i, S_{i-1} 中有一為 composite, 則以 composite 為 CL 增加及 new address 之基礎

(ii)若S_i, S_{i-1}皆為 original, skip this row and CL 不變

4. Table II indicates that each original symbol in the table has its codeword length (CL) specified.
   Ordering the symbols according the their CL values gives Table III.

Associated with the so-obtained TOCL one can actually generate a number of Huffman tables (or Huffman trees), each of which being different in codewords but identical in the codeword lengths.

60

Single-Side growing Huffman table (SGHT)

— TableIV

Single-Side growing Huffman tree (SGH-Tree)

— Fig. 1

are adopted.

TABLE IV
SINGLE-SIDE GROWING HUFFMAN TABLE (SGHT

CL	symbols	Huffman Code
1	s_1	0
2	s_2	10
4	s_3	1100
4	s_4	1101
4	s_5	1110
5	s_6	11110
5	s_7	11111

61

- **For a uniformly distributed source SGH-Tree becomes full.**

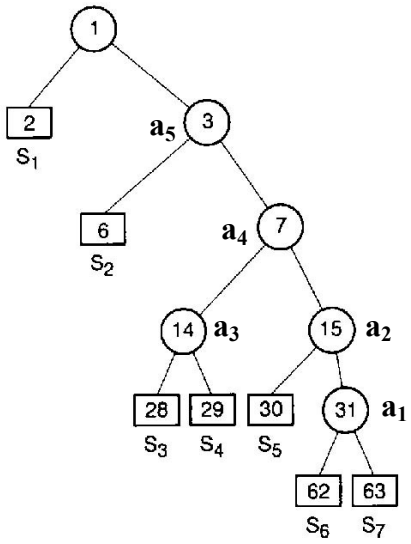


Fig. 1. A 5-level SGH-tree.

62

- Algorithm 2: Constructing a SGHT from a TOCL
  - Start from the first row of the table and assign an "all zero" codeword C_1 = 00…0 to the symbol S_1.
  - Next we increment this codeword and assign the new value to the next symbol in the table. Similarly, we proceed coreating codewords for the rest of the symbols in the same row of the TOCL.
  - When we change rows, we have to expand the last codeword, after being incremented, by placing extra zeros to the right, until the codeword length matches the level (CL).

63

In general we can write :

$$C_1 = 00\cdots 0$$

$$C_{i+1} = (C_i + 1) * 2^{p-q}$$

and

$$C_n = 11\cdots 1$$

where p and q are the codeword lengths for s_i and s_{i+1}, respectively, and S_n (associated with C_n) denotes the terminal symbol.

C_1 and C_n have unique forms → easy to verify

64

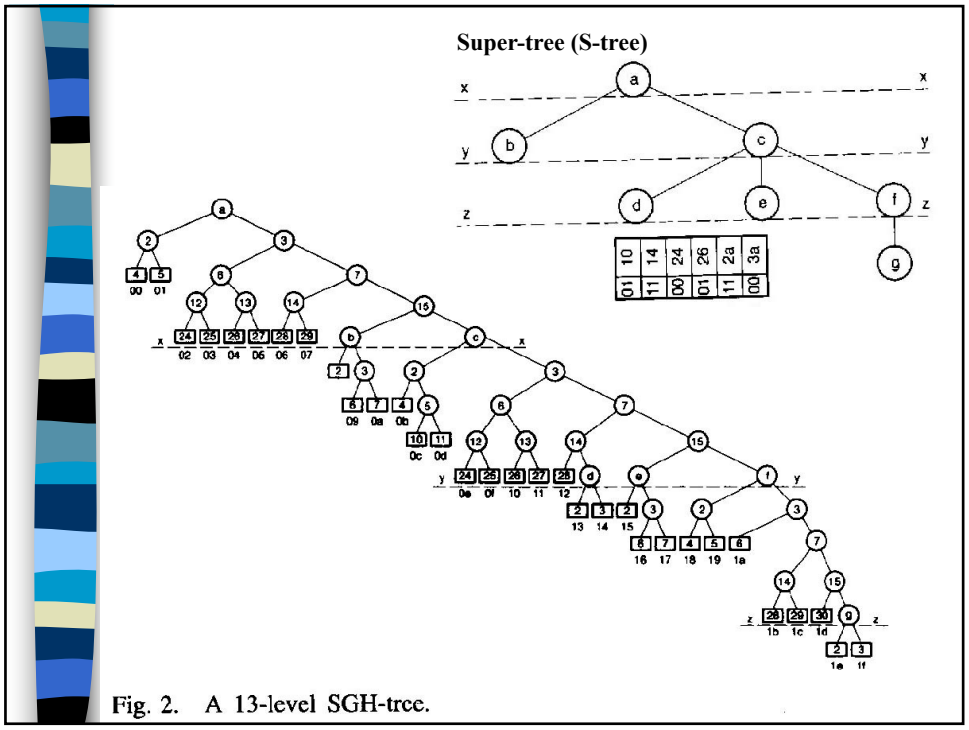


Fig. 2. A 13-level SGH-tree.

65

TABLE VI
SINGLE-SIDE GROWING HUFFMAN TABLE (SGHT)

X				Y				Z			
CL	Symbols	Huffman	Code	CL	Symbols	Huffman	Code	CL	Symbols	Huffman	Code
2	00	00		9	13	1111	1101	0	12	1111	1100
2	01	01		9	14	1111	1101	1	12	1111	1101
4	02	1000		9	15	1111	1110	0	12	1111	1110
4	03	1001		10	16	1111	1110	10	13	1111	1111
4	04	1010		10	17	1111	1110	11			
4	05	1011		10	18	1111	1111	00			
4	06	1100		10	19	1111	1111	01			
4	07	1101		10	1a	1111	1111	10			
5	08	1110	0	12	1b	1111	1111	1100			
6	09	1110	10	12	1c	1111	1111	1101			
6	0a	1110	11	12	1d	1111	1111	1110			
6	0b	1111	00	13	1e	1111	1111	0			
7	0c	1111	010	13	1f	1111	1111	1			
7	0d	1111	011								
8	0e	1111	1000								
8	0f	1111	1001								
8	10	1111	1010								
8	11	1111	1011								
8	12	1111	1100								

66

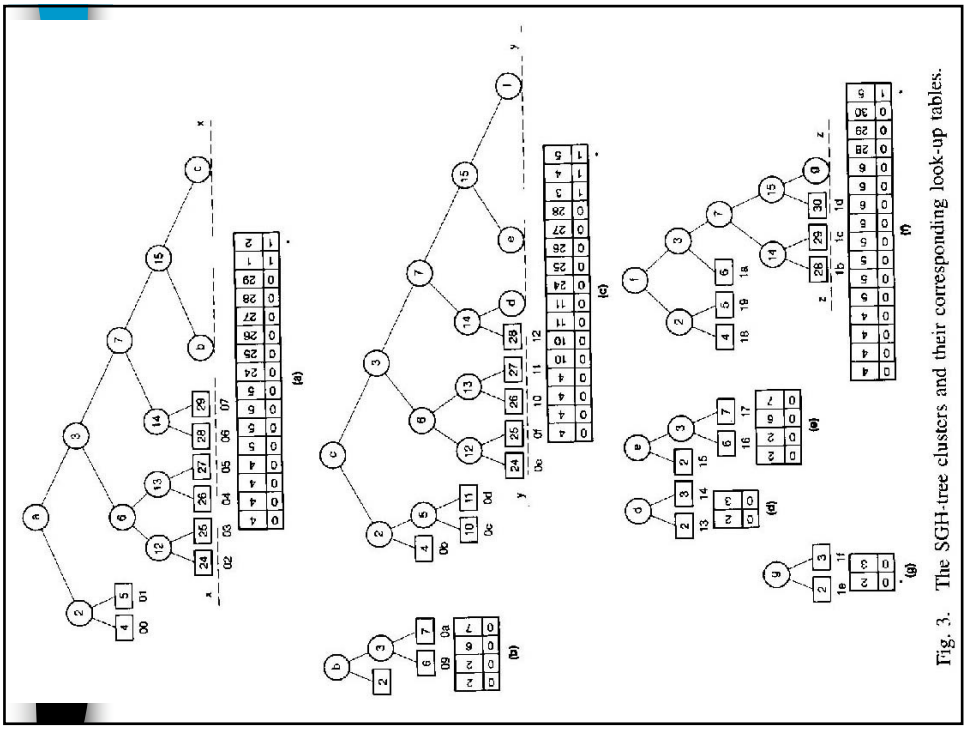


Fig. 3. The SGH-tree clusters and their corresponding look-up tables.

67

TABLE VII

Memory (RAM) Space Associated with Table VI and Figs. 3 and 4

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	00	01							02	03	04	05	06	07		
1	08		09	0a	0b		0c	0d					0e	0f	10	11
2	12				13	14	15		16	17	18	19	1a			
3							1b	1c	1d		1e	1f				

68

Storage Allocation

- For non-uniform source, SGH-Tree becomes sparse.

How to

i) optimize the storage space

ii) provide quick access to the symbol (data)

key Idea:

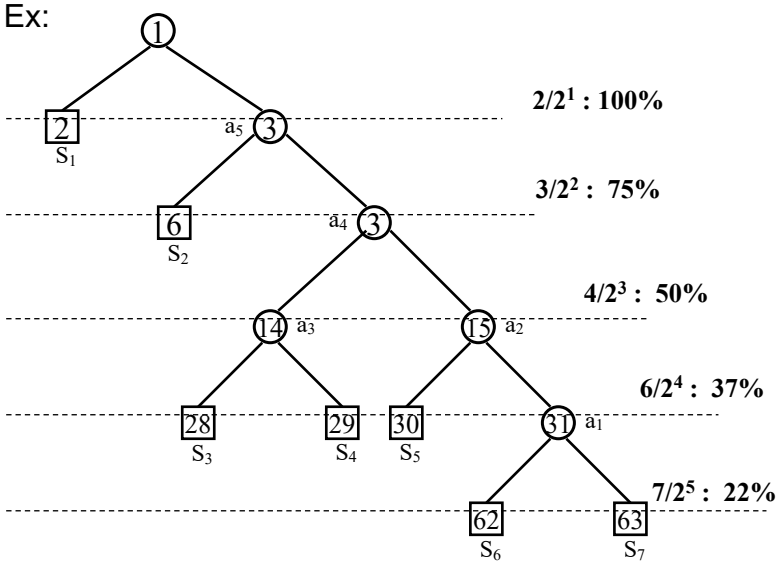
Break down the SGH-Tree into smaller clusters (subtrees) such that the memory efficiency increases.

The memory efficiency B for a K-level binary Huffman tree

$$B_K = \frac{\text{No. of effective leaf nodes}}{2^K} \times 100\%$$

69

- Ex:



70

- Remarks:
  1. The efficiency changes only when we switch to a new level (or equivalently to a new CL), and it decreases as we proceed to the higher levels.
  2. Memory efficiency can be interpreted as a measure of the performance of the system in terms of memory space requirement; and it is directly related to the sparsity of the Humman tree.
  3. Higher memory efficiency for the top levels (with smaller CL) is a clear indication that partitioning the tree into smaller and less sparse clusters will reduce the memory size. In addition, clustering also helps to reduce the search time for a symbol.
- Definition:

  A cluster (subtree) T_i with minimum memory efficiency (MME) B_i, if there is no level in T_i memory efficiency less than B_i.

71

SGH-Tree Clustering

- Given a SGH-tree, as shown in Fig. 2, depending on the MME (or CL) assigned, the tree is partitioned by a cut line, x-x, at the Lth level (L=4 for the choice of MME=50%, in this example).
- The first cluster (subtree), as shown in Fig.3(a), is formed by removing the remainder of the tree beyond the cut-line x-x.
- The **cluster length** is defined to be the maximum path length from the root to a node within the cluster
  $\rightarrow$ the cluster length for the first cluster is 4.
- Associated with each cluster a **look up table** (LUT) is assigned, as shown at the bottom of Fig.3(a), to provide the **addressing information** for the corresponding **terminal node** (symbol) within the cluster, or beyond.

72

- To identify other clusters, in the tree, we draw more cut lines y-y, and z-z, each L levels apart.
- More clusters are generated, each of which starting from a single node (**root of the cluster**) and expanded until it is terminated either by terminal nodes, or nodes being intercepted by the next cute line.
- Next, we construct a **super-tree (s-tree)** corresponding to a SGH-Tree. In a s-tree **each cluster** is represented by a **node**, and the links connecting these nodes, representing the branching nodes in the SGH-tree, shared between two clusters.
- The super-table (ST) associated with the s-tree is shown at the bottom of the tree.
  Note that the s-tree has 7 nodes, one for each cluster, while its ST has 6 entries. This is because the root cluster a is left out and the table starts from cluster b.

73

Entries in the ST and the LUT's

- There are two numbers in each location in the ST
  - the first number identifies the cluster length
  - the 2nd number is the offset memory address for that cluster

Ex:

	binary	Hexa
f	11	$2a_H$

cluster length : 11+1 = 100, or 4

$2a_H$: the starting address of the corresponding LUT, in the memory (see table Ⅶ)

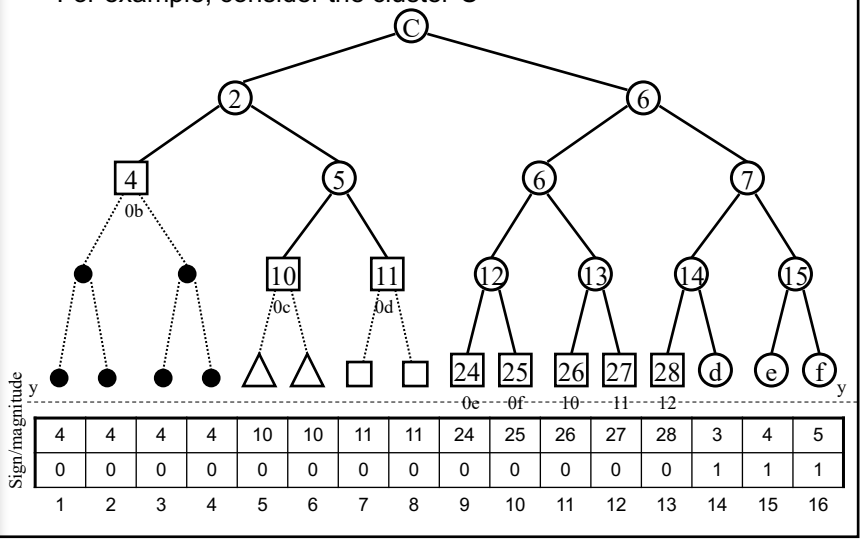
$\Rightarrow$ the cluster f start at address $2a_H$ in the memory table Ⅶ. (i.e. symbol 18)

- Each entry in a LUT is an integer in sign/magnitude format.
  Positive integer, with 0 sign, correspond to the nodes existed in the cluster, while negative numbers, with 1 sign, represent the nodes being cut by the next cut line.

74

The magnitude of a negative integer specifies a location in the ST

- The magnitude of a negative integer specifies a location in the ST, for further search.
  For example, consider the cluster-C



4	4	4	4	10	10	11	11	24	25	26	27	28	3	4	5
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

75

In the 15th entry of the above LUT we find ¼ as sign/magnitude at that location.

the negative sign (1) indicates that we have to move to other cluster, and 4 refers to the 4th entry in the ST, which corresponds to cluster e, and contains 01 and 26H numbers.

The first number, 01, indicates the cluster length is 01+1=10(or 2), and the 2nd number, shows the starting address of cluster e in the memory.

- A positive entry in a LUC, indicates that the symbol is already found, and the magnitude comprises three pieces of information

i) location of the symbol in the memory

ii) the pertinent codeword

iii) the codeword length

76

Read the positive integer in binary form, and identify the most significant 1 bit (MS1B). The **position** of the MS1B in the binary number specifies the **codeword length**, the **rest** of the **bits** on the **right side** of the MS1B gives the codeword, c_j, and the **magnitude of c_j** is, in fact, the **relative address** of the symbol in the memory (Table Ⅶ)

77

- For example

1	0	4
2	0	4
3	0	4
4	0	4
5	0	5
6	0	5
7	0	5
8	0	5
9	0	24
10	0	25
11	0	26
12	0	27
13	0	28
14	0	29
15	1	1
16	1	2

07 symbol

0 → the symbol is found

8 7 6 5 ④ 3 2 1 0
29 = 0 0 0 0 1 1 1 0 1

Codeword length = 4

Symbol 07 is located at 1101=d_H
(See Table Ⅶ)

0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00	01	·	·	·	·	·	·	02	03	04	05	06	07	·	·

The 1st row of the table Ⅶ

$29)_H = f)_H + d)_H$

indication of the tree-depth of the node
Cluster length (codeword length)

memory location (address) if the memory table is offset by f

78

Huffman Decoding

- The decoding procedure basically starts by receiving an L-bit code c_j, where L is the length of the top cluster in the associated SGH-Tree (or the SGHT). This L-bit code c_j is then used as the address to the associated look-up table [fig.3(a)]

Example:

1. Received codeword : 01100 1011…
   first L=4 bit 0110=6 as an address to the LUT given in Fig.3(a).
   The content of the table at this location is 0/5, as the sign/magnitude.
   0→the symbol is located in this cluster
   5=00000101, the MS1B is at location 2
   ⇒ CL=2. Next to the MS1B we have 01, which represents the codeword
   ⇒ the symbol (01H) is found at the address 01 in the memory (Table Ⅶ)

MSB

MS1B

79

2. R'd codeword : 110110011…
   the first 4 bit 1101 = d)$_H$ at the d-th location of the LUT (Fig.3(a)) we get 0/29

MSB

The symbol is at this cluster

MS1B

29 = 00011101

The address to the memory is 1101=d)$_H$
And the symbol is found to be 07.

3. R'd codeword = 1111 1111 1111 0010…
   1111→ the symbol is not in cluster a and refer to location 2 in the ST, assigned to cluster C.

11)$_2$|14)$_H$: the cluster length is 11+1=100 or 4

The offset for the symbol memory designate for cluster C : the memory addressing (table VII) of cluster C started at 14)$_H$

80

Our next move is to select another slice of (with length 4) from the bit stream, which 1111 again. From the 15th location of the LUT for cluster C we get 1/5

The symbol is not in cluster C

Location 5 of the ST → cluster f

The content is $11)_2$ $2a)_H$
11+1 = 100 : or 4 : cluster length
$2a)_H$: the offset of memory location of cluster f
⇒ The symbol is not located yet, we have to choose another 4 bit slice from the bit stream : 1111

we move to the 15th location in the LUT for cluster f. Here we find 1/6 and refer to the 6th item in the ST.
The data here is read $00)_2$ and $3a)_H$

$00)_2$ → CL of cluster g is 00+1=01 ; or 1

$3a)_H$ → the memory location offset of cluster g.

81

Take 1 bit from the bit stream which is "0" referring the LUT of cluster g, location "0" gives 0/2
So the symbol is in cluster g, and 2 = 00000010 ⇒
(i) MS1B is located at location 1 ⇒ CL = 1
(ii) to the right of MS1B is a single 0 identifying the codeword, and
(iii) the symbol (1e) is at location 3a+0 = 3a in the memory (tableⅦ)

⇒ The overall codeword is given as
1111, 1111, 1111, 0, with CL = 4+4+4+1 = 13

82

- Remarks:

1. For high probable symbols with short codewords (4 bits or less) the search for the symbol is very fast, and is completed in the first try. For longer codewords, however, the search time grows almost proportional to the codeword length.

   If CL is the codeword length
   L is the maximum level selected for each cluster
   (L = 4 in the above example) then the search time is closely proportional to 1+CL/L

2. Increasing L:

   i. Decreasing search time → speed up decoding

   ii. Growing

   ⇒ Trade-off

83