Adaptive Huffman Coding :-

⇒ Huffman coding requires knowledge of the probability of the source sequence, if this information is not available, than Huffman coding becomes two pass procedure :-

1. The statistics are collected in the first pass.
2. and the source is encoded in the second pass.

⇒ In order to convert this algorithm into one-pass procedure, another adaptive algorithm is developed to construct the Huffman code based on the statistics of the symbols already encountered.

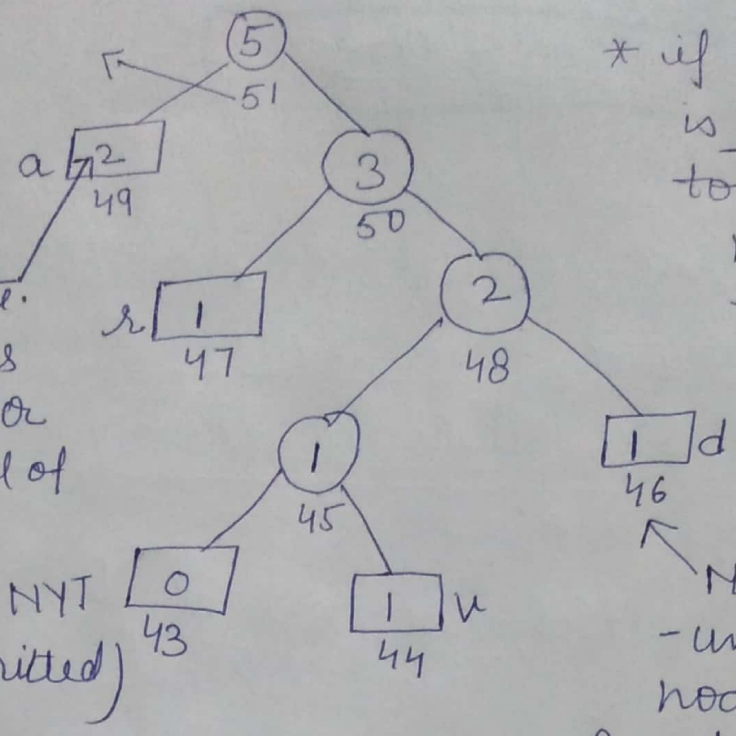
⇒ Adaptive Huffman coding maintains a dynamic code tree or binary tree with two parameters :-

1. the weight of each leaf. &
2. a node number

- the weight of each external node is simply the no. of times the symbol corresponding to the leaf has been encountered.
- the weight of each internal node is the sum of the weight of its offspring.
- the node number 'i' is a unique no. assigned to each internal and external node.

eg 1

$51 = 2m-1$
$= 2 \times 26 - 1$
$= 51$



weight of a node.
no. of occurances of the symbol or all the symbol of subtree.

Yet
(Not transmitted) NYT node

Node number - unique ID of each node.
Parent id > Right id > left id

* if the alphabet size is 'm' then the total no. of internal node is $\leq 2m-1$

Fig - (Huffman) Adaptive Huffman tree

⇒ In the adaptive Huffman coding procedure, neither transmitter nor receiver knows anything about the statistics of the source sequence at the start of transmission.

⇒ The tree at both side consists of a single node that corresponds to all symbols 'not yet transmitted (NYT)' and has a weight of zero.

⇒ An Adaptive Huffman code for an alphabet is construct as follows –

• If the source has an alphabet $(a_1, a_2 \cdots a_m)$ of size 'm', then pick 'e' and 'r' such that $m = 2^e + r$ and $0 \leq r \leq 2^e$.

∴ the letter a_k is encoded as the (e+1) bits binary representation of (k-1) if $1 \le k \le 2^r$ else a_k is encoded as the e bit representation of k-r-1.

for eg:- if m = 26 then e = 4 & r = 10

$26 = 2^4 + 10$ then the symbol a_2 is encoded as 00001 bcoz $k = 2 \le 2^r$ so (e+1) bits for (k-1) hence 5 bits for (k-1) = 1 so

$a_2 = 00001$

Similarly if the symbol is a_{22} i.e $k = 22 > 2^r$ then we require 'e' bits binary representation of (k-r-1) ⇒ (22-10-1) = '11' hence a_{22} is encoded as — 1011 [8 4 2 1 / 1 0 1 1]

⇒ When a symbol is encountered for the first time, the code for the NYT node is transmitted followed by the fixed code for a symbol.

⇒ A node for the symbol is then created and the symbol is taken out of the NYT list.

⇒ In Adaptive huffman tree, both receiver and transmitter start with the same tree structure.

⇒ the updating procedure used by both is identical.

⇒ the encoding and decoding processes remain synchronized.

Update Procedure

- The update procedure requires that the nodes be in a fixed order.

* The largest no. is given to the root of tree.
* Update Procedure preserve the sibling property of Huffman coding tree.

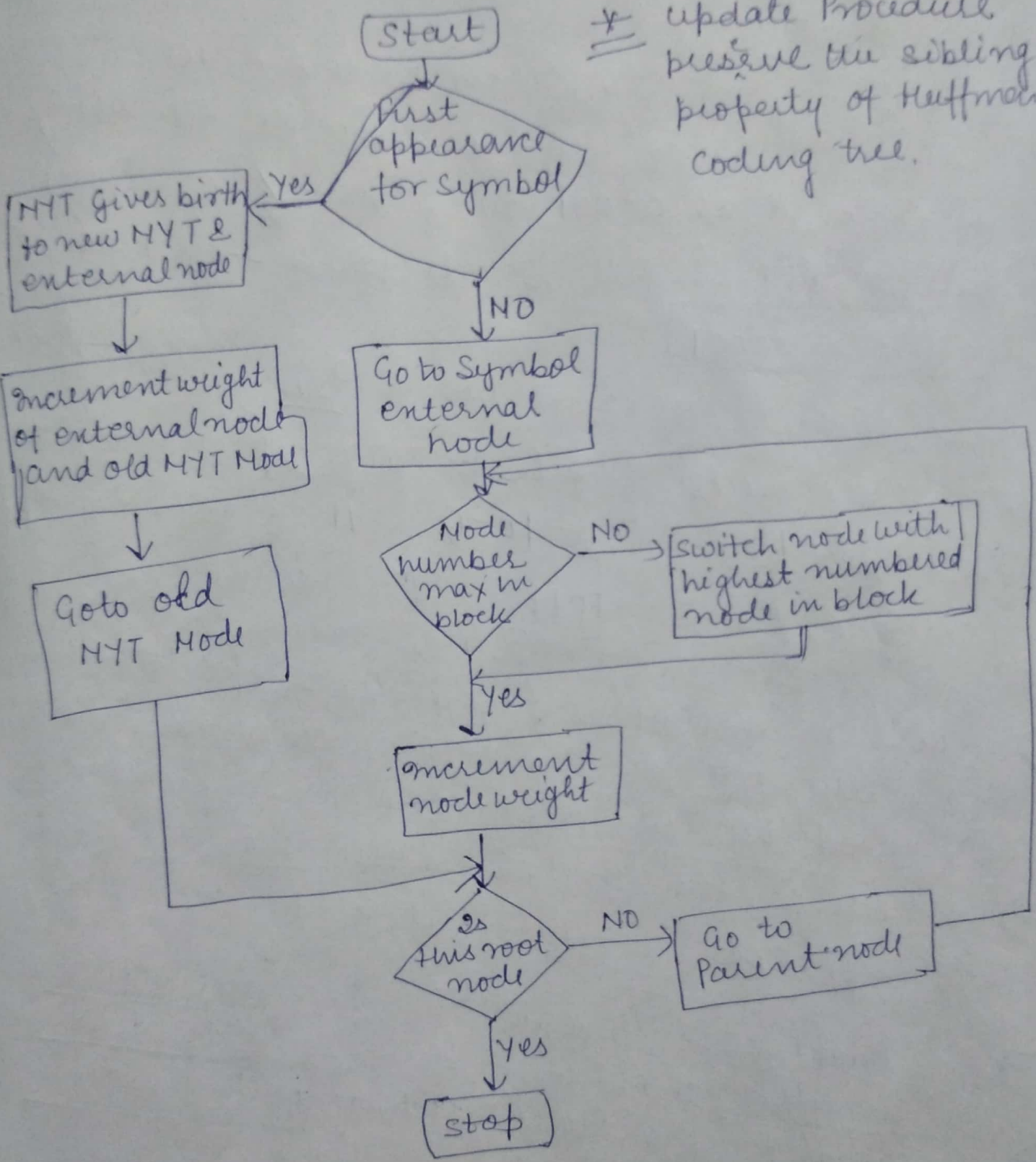


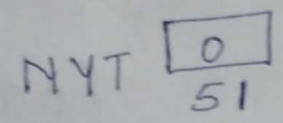
Fig:- Update Procedure for the Adaptive Huffman Coding Algorithm.

Example - update Procedure

→ Consider the msg [aardv] where our alphabets consists of 26. ie m = 26, hence total no. of internal nodes in a tree is 2m-1
= 51 nodes..

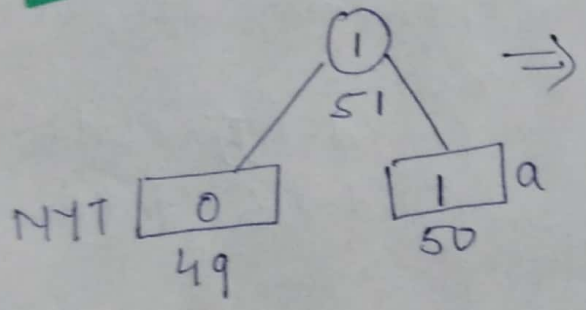
→ Hence we start unique id no. with 51 assign to the root node.

Step 1 :- Start with NYT node.



[assign id 51 & weight=0 due to NYT node]

Step 2 :- First letter 'a' is transmitted [aardv]

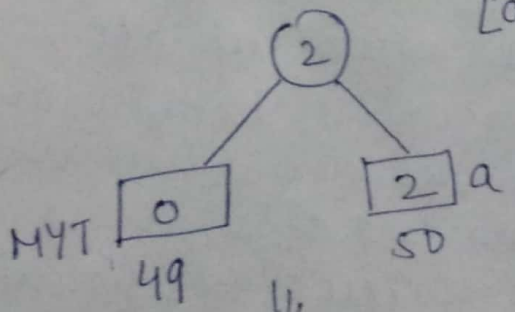


→ when a transmitted NYT node generates 2 node
① - NYT node
② - External node for 'a'

→ assign the unique id or no according to P.id > right id > left id
hence assign 50 to terminal node & 49 to New NYT node.

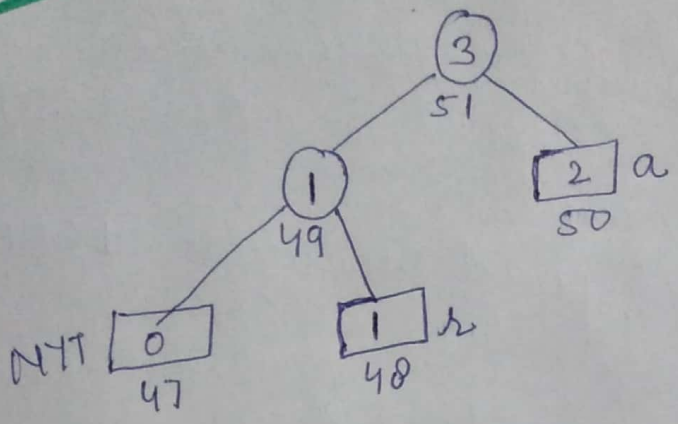
→ increment weight of terminal node & as well as root node.

Step 3 :- Second 'a' is transmitted [aardv]



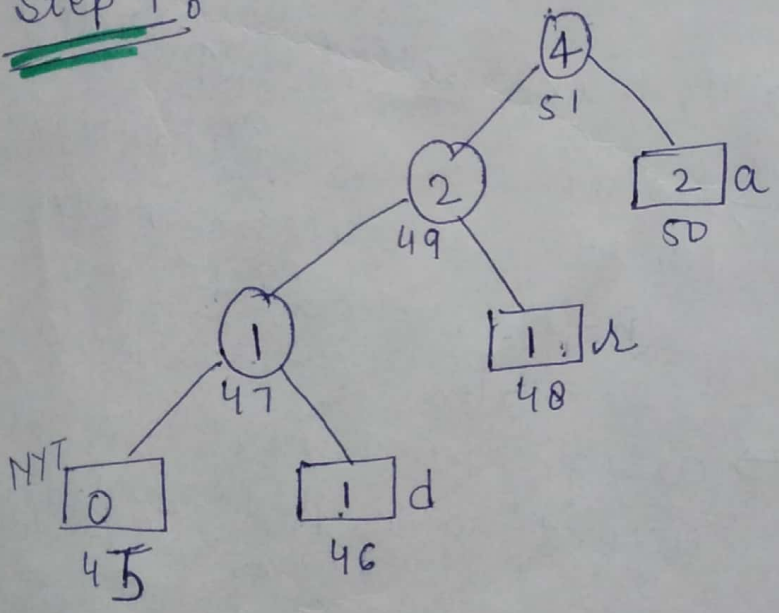
→ Thus since the transmitted cod is 1, hence the node corresponding to 'a' has the highest no. so no need to swap nodes.

Step 3 :- Now transmit 'r'. [aardv]



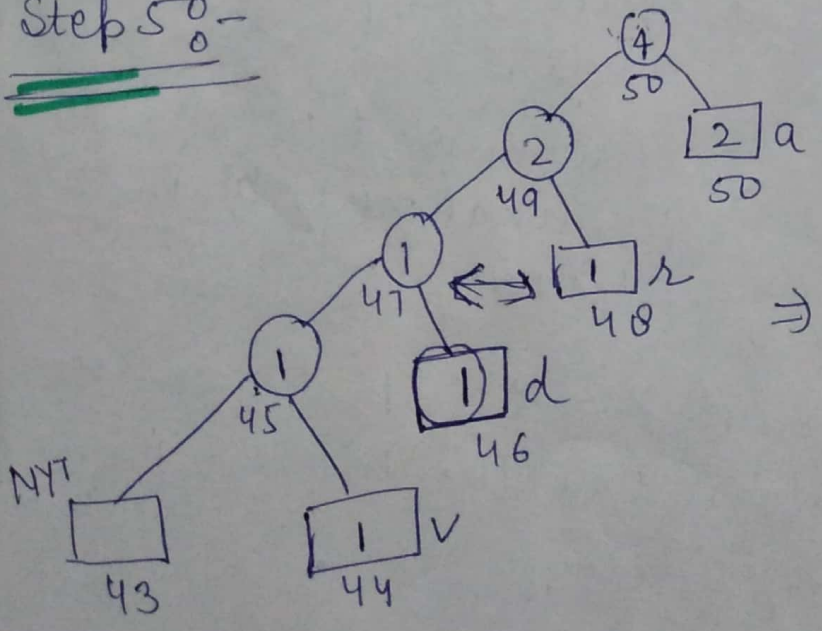
→ Send codeward to NYT node
→ NYT node generates 2 new node - new NYT node & external node for r.
→ Again no ~~need~~ update is required.

Step 4 :- [aardv]



→ No update Require
→ NYT generate two node -
(1) id - 45 - New NYT
(2) id - 46 - for 'd'

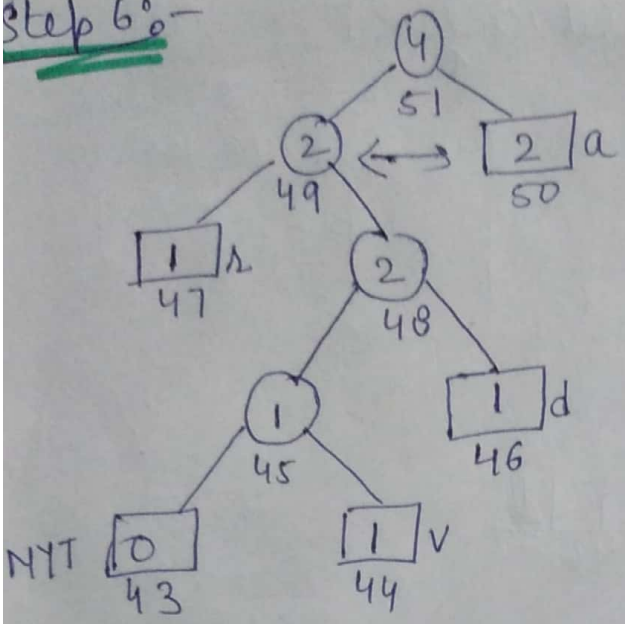
Step 5 :- [aardv]



⇒ → It require update procedure, bcoz when we increment the value of parent of 'v' then the value of 'v's grand parent become - 2 which is greater then value of '48'. hence loss of sibling property of huffman tree. so swap them

conti...

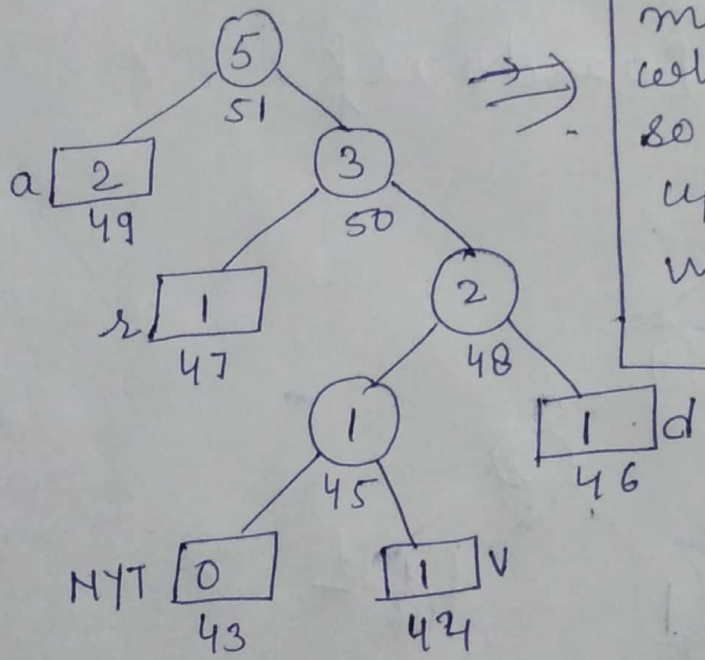
Step 6:-



⇒
- → Swap 47 with node 48, which has the largest no. in block.
- → increment node 48 and move to its parents 49.
- → again swap the 49 with 50

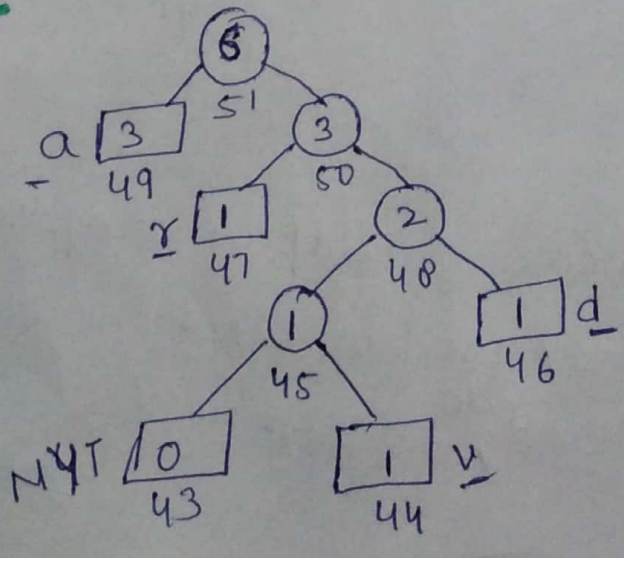
⇓

Step 7:-



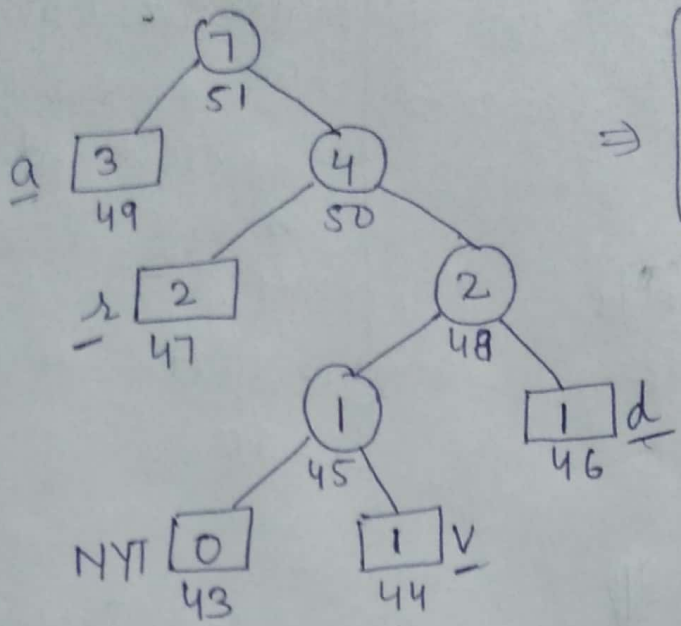
→ After swapping move to 51 which is highest so no need to update just increment the value of 51

Step-8 if the msg - [aardvar]



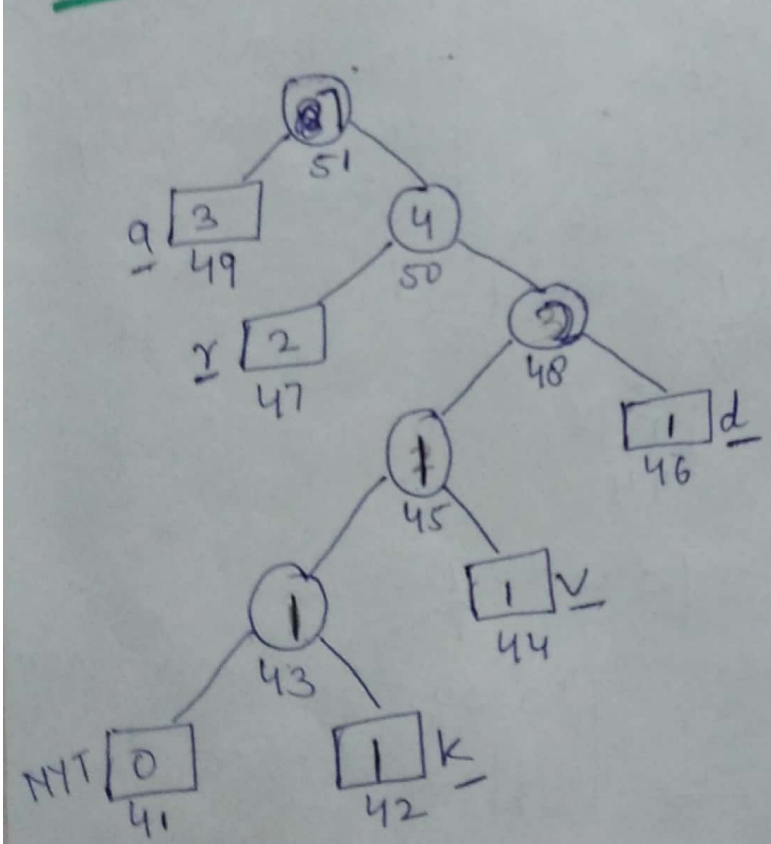
⇒
- 'a' symbol is already appeared, hence just increase the value of node 49 external node
- → ~~go to~~ no need to update.

Step 9:- [aardva r k]

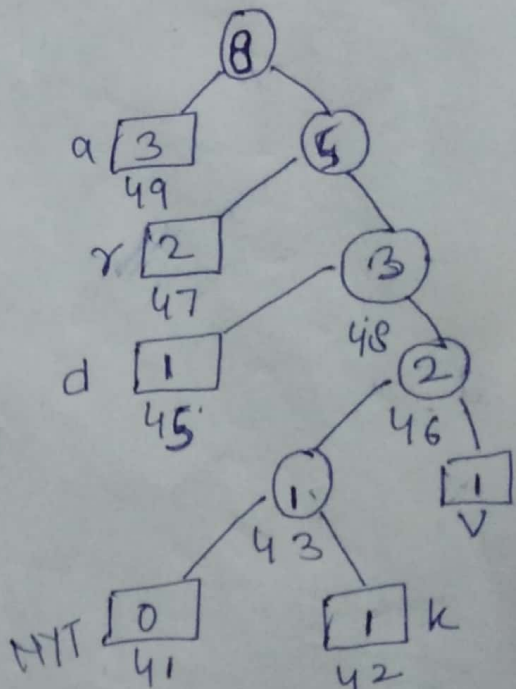


⇒ [same as in step 8.]

Step 10:- [aardvark]



⇒



* require update bcoz grandparent of 'k' is 45 which is not max in his block so swap 45 with 46 and do increment.

Encoding Procedure

* Initially, the tree at both the encoder & decoder consists of a single node, the NYT node.

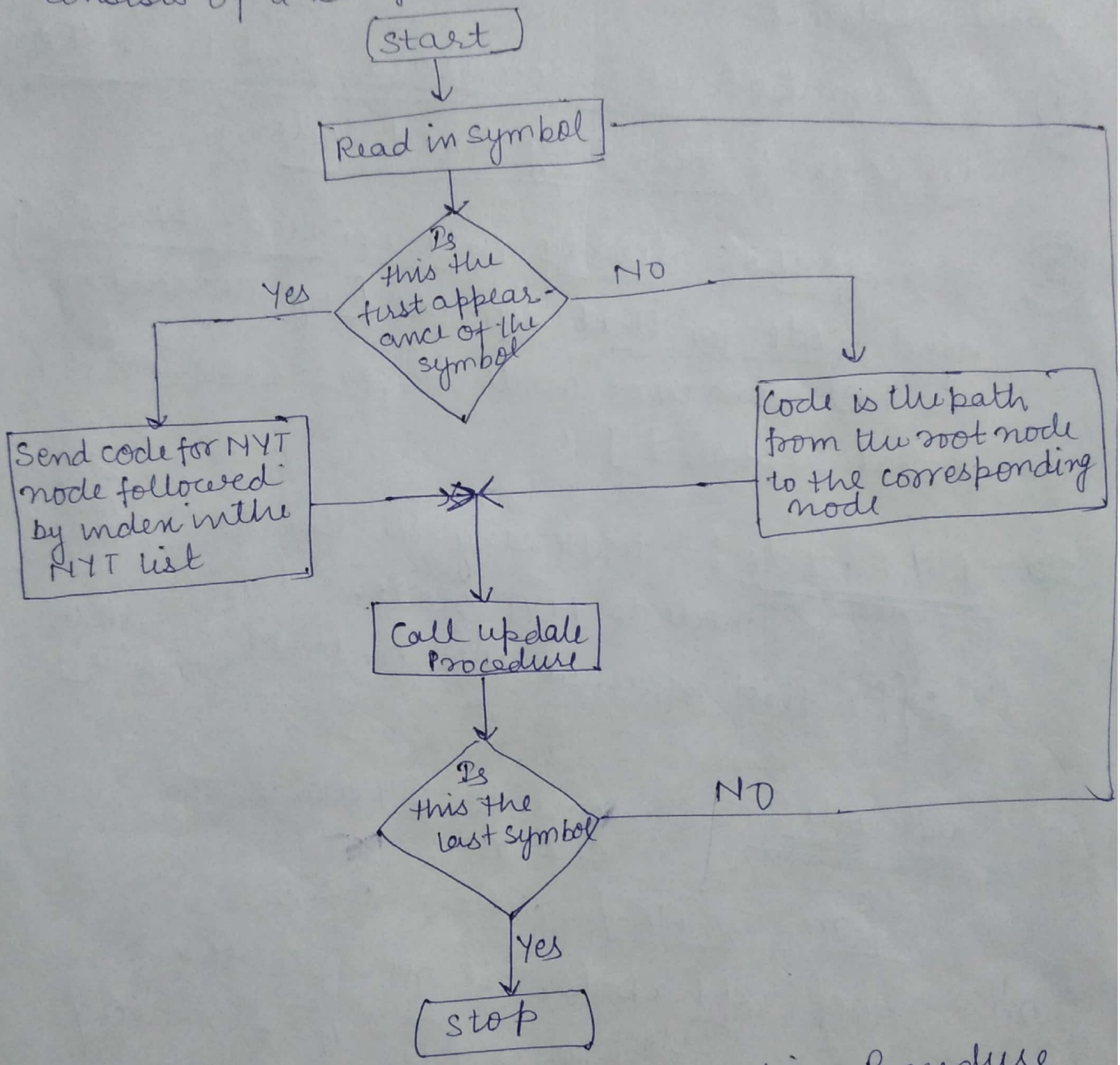


Fig:- Flowchart of the Encoding Procedure.

Encoding Procedure :-

- Encoding Procedure follows two steps to encode the alphabet

1. If a symbol is encountered first time then NYT code followed by fixed code.
2. If a symbol already encountered then only send code for that symbol [by traversing the tree from root node to external node of a that alphabet]

⇒ NYT code :- traversing the tree from root node to NYT node.

- [for first symbol NYT code is blank or no code]

⇒ Fixed code :- It consider 2 parameters e & r.

$m = 2^e + r$ where $0 \le r \le 2^e$

m - no. of alphabets. if $m = 26$ then

$26 = 2^4 + 10 \Rightarrow$ $e = 4$ & $r = 10$

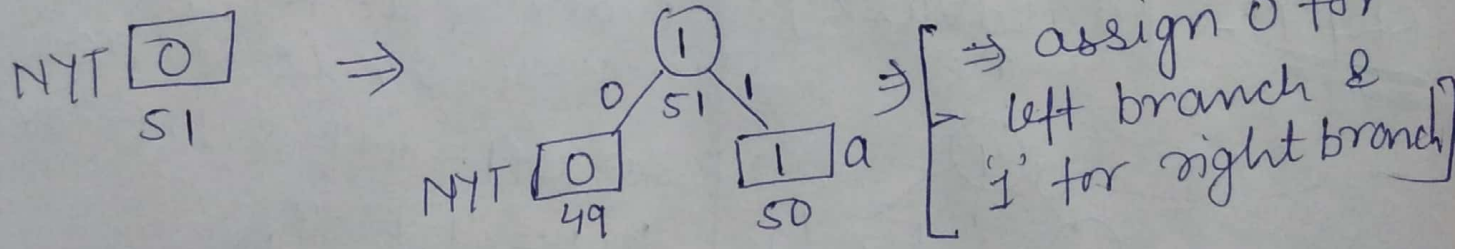
1. if $1 \le k \le 2r \Rightarrow a_k$ is encoded as (e+1) bit binary representation of (k-1). [k - ~~idex~~ index]
2. if $k > 2r \Rightarrow a_k$ is encoded as e-bit binary representation of (k-r-1).

codeward for 'a' ⇒ NYT Code + Fixed Code

eg encode [aardvaark] using Adaptive Huffman code encoding Procedure.

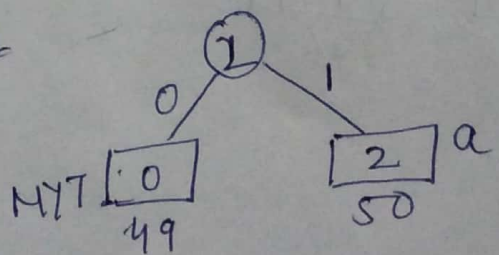
Step 1 :- [aardvark]
↑

→ the first symbol encode is the letter a.
a is the first alphabet hence k=1
k=1 ⇒ $k < 2^r$ [r=10] ⇒ k<20
so we required e+1 = 4+1 = 5 bits binary representation of (k-1) i.e for '0'
ie a ⇒ NYT code + fixed code
⇒ — + 00000
a ⇒ 00000
[initially only single node which is NYT so no NYT code for 'a']



Step 2 :- [aardvark]
↑

again 'a' already appeared so, we have already externode for a, hence for codeward we simply traverse from root to node &
codeward for 'a' is = 1



Step 3 :- [aardvark]

next ∞ symbol is - 'r'

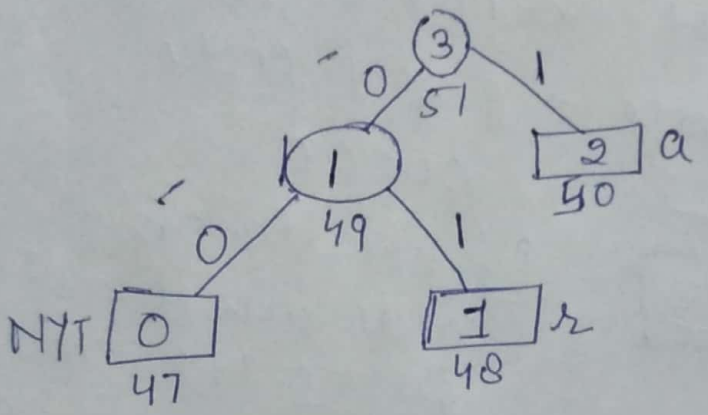
for r - k = 18 < 2r so e+1 = 5 bits for

k-1 ⇒ 18-1 = 17.

codeward for r - NYT Code + Fixed code

- ~~of~~ 10001

r - 010001



Step 4 :- [aardvark]

⇒ index of d - k = 4

K = 4 < 2r ie 4 < 20

so e+1 = 4+1 = 5 bit for

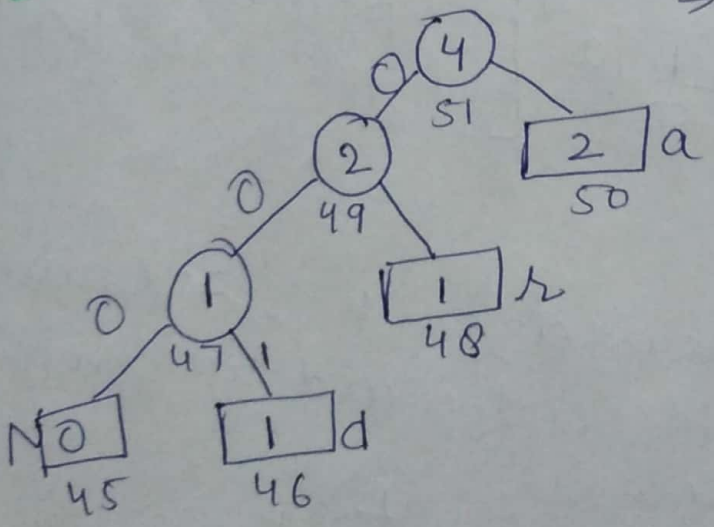
k-1 = 4-1 = 3.

Codeward for d

⇒ NYT Code + fixed code
(Root to NYT node)

d = 00 + 00011

d = 0000011



Step 5 :- [aardvark]

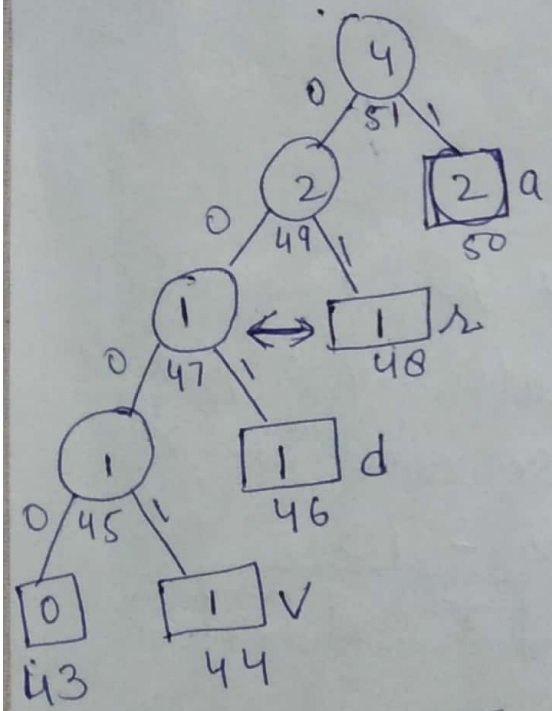
index for v − k = 22

k = 22 > 21 ie 'e' but for binary representation of (k − r − 1) ⇒ (22 − 20 − 1) = 11'

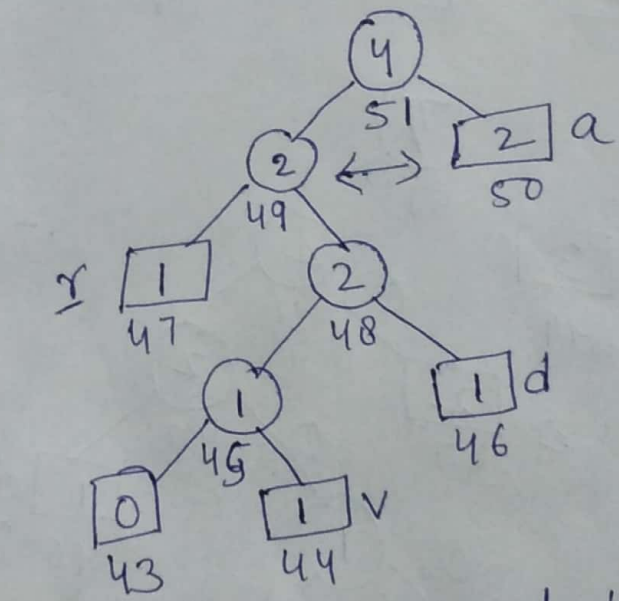
Codeword for v ⇒ NYT + Codeword

⇒ 000 + 01011

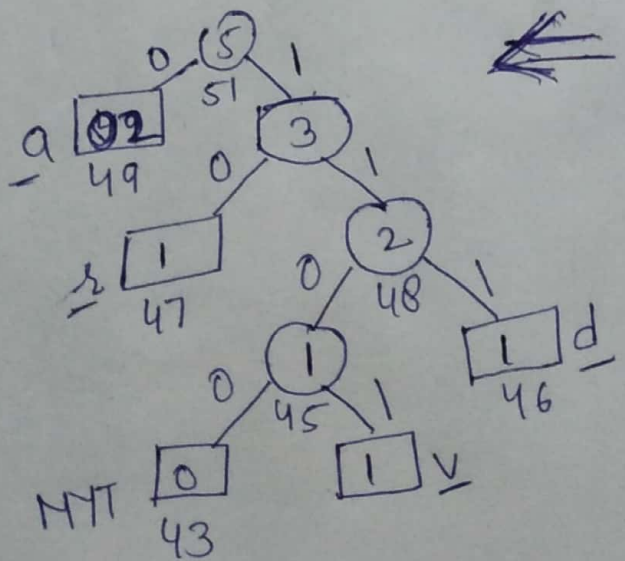
v ⇒ 00001011



* need to update ~~tree~~ swap 47 with 48

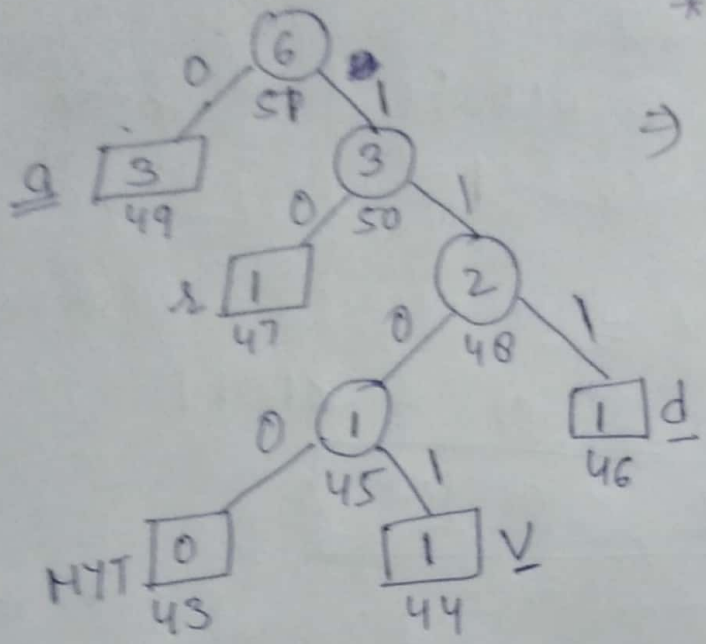


* swap 49 with 50.



(updated tree)

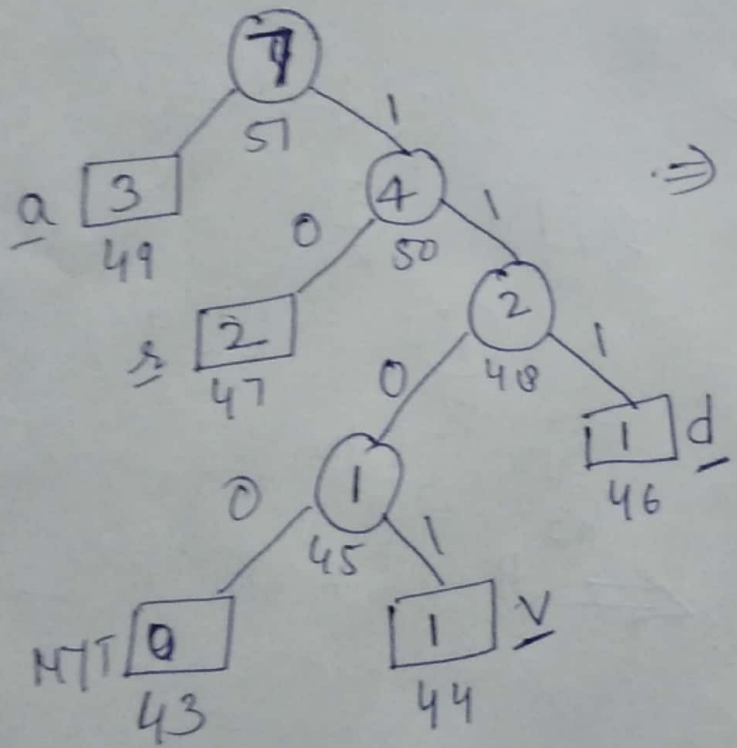
Step 6:- [aardvark]
↑



* as 'a' already appeared so code word for a is '0' [traverse from root to external node 'a']

⇒ **a = 0**

Step 7:- [aardvark]
↑



⇒ * As 'r' symbol already appeared so code for 'r' is - 10

r = 10

Step 8:- [aardvark]
↑

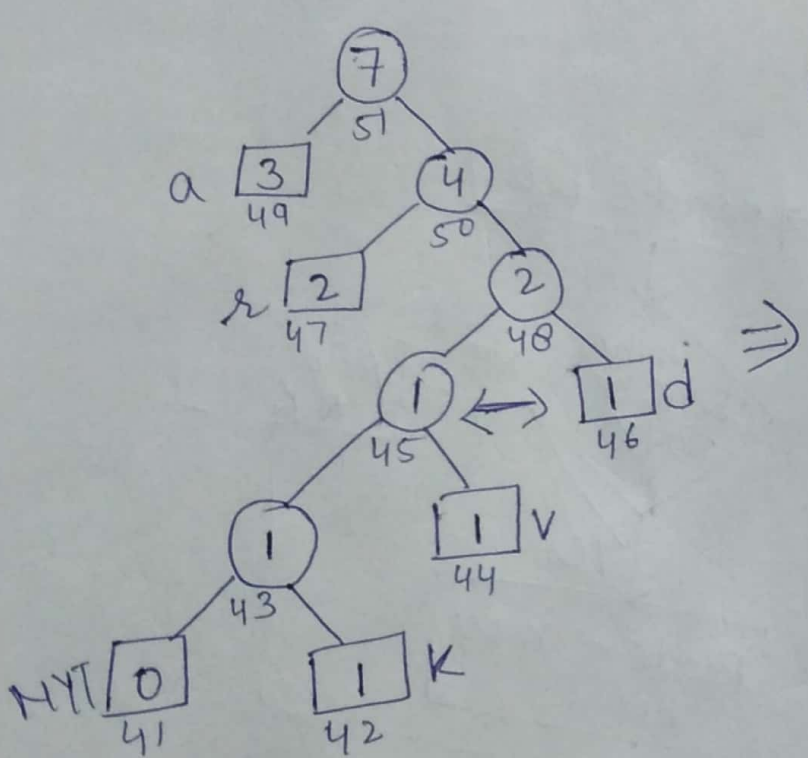
'k' appears first time, index for k is - 11

$11 < 2r$ so $e+1 = 4+1$ bit for binary representation of ~~10~~ $11-1 = 10$

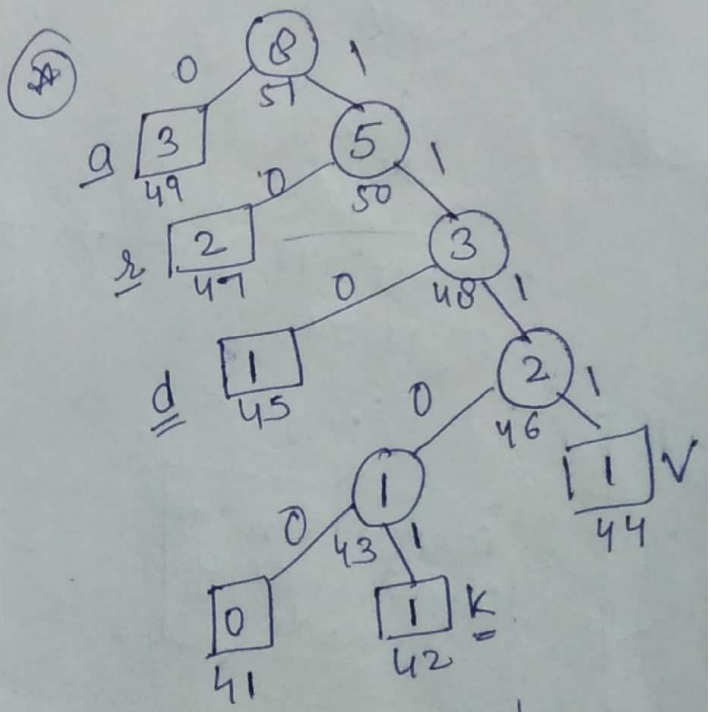
Codeward for 'k' = NYT code + fixed code

= 11000 + 01010

k ⇒ 1100001010



⇒



(*) require update ⇒ swap 45 with 46'

hence the codeward for [aardvark] is

000000 1 0 1 000 | 0000001 | 000101 | 1 | 0 | 10 | 1100001010

a a r d v a r k.

Decoding Procedure :-

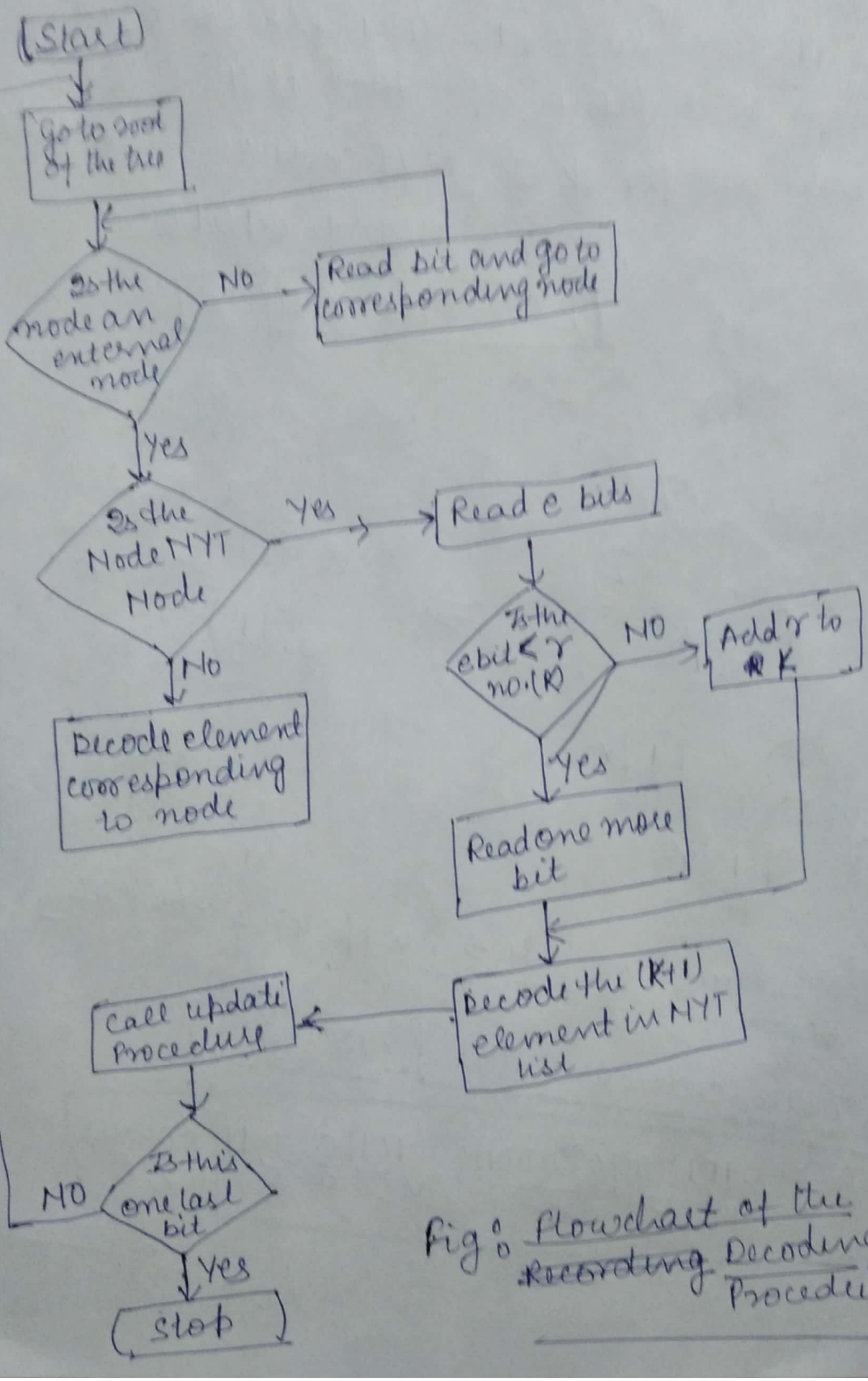


Fig :- Flowchart of the Decoding Procedure

- In decoding procedure we traverse the tree in a manner identical to that used in the encoding procedure.
- Once a leaf is encountered, the symbol corresponding to that leaf is decoded.
- If the leaf is the NYT node, then, we check the next e bits to see if the resulting number is less than r.
- If it is less than r, we read in another bit to complete the code for the symbol.
- The index for the symbol is obtained by adding one to the decimal no. corresponding to the 'e' or 'e+1' bit binary string.
- Once the symbol has been decoded, the tree is updated and next received bit is used to start another traversal down the tree.

eg → (Next)

eg:- The binary string generated by encoding procedure is

00000101000100000110001011O

c = 4 m = 26
r = 10

step1:- Initially, the decoder tree consists only of the NYT node.

step2:- Read first 4 bits - 0000 ; (bcoz c=4 so take first 4) the value of 4 bits 0000 corresponds to decimal value of 0, and

0 < r ; 0 < 10,

so we read one more bit for the entire code - ie - 00000 .

step3:- Adding one to the decimal value corresponding to the binary string, then we get the index of symbol

ie → 00000 → 0+1 = 1 ie symbol a

Note:- Tree will be constructed same manner as in encoding and update procedure.

step4:- Next bit is '1' - this trace the path from root node to external node. ie '1' is corresponds to next 'a',

Step5°:- Next bit is a '0' which trace the path from root to NYT node, hence take next 4 bits - 1000.

⇒ 1000 - '8' < 10 so take 1 bit more ie 10001 - '17' hence the index is 17+1 = 18 ie next symbol is ~~to~~ 'r'

Note → we just reverse the encode procedure

eg ⇒ if we do k-1 in encoding then we do k+1 in decoding.

⇒ if we do k-r-1 in encoding then we do k+r+1 in decoding.

⇒ we take e = 4 bit, beoz we use e=4 in encoding procedure same with r = 10.

Now decoded string - aar.

Step6°: Now ~~take~~ next two bit is '00' ie trace the path root to NYT, so again take next 4 bit - 0001 which 0001 = 1 < 10 (decimal eqv.) so take next 1 bit more ie 00011 ⇒ '3' hence the index is 3+1=4 hence next symbol is 'd'.

⇒ decoded string - 'aard'

Step 7:- Now check next 3 bits - 000'
- the path from root to NYT node

Note:- we increase 1 bit at every step to find the NYT code - bcoz codeward = NYT + FC in encoding procedure;
→ every time one bit is increased in NYT code in Encoding procedure so same as in decoding procedure

Step 8 - Now take next 4 bit - 1011. (e=4)
1011 - 11 > 10 [r=10] hence consider only e=4 bit.
Now for index value calculate
k + r + 1 ⇒ 11 + 10 + 1 = 22
hence 22nd index symbol is 'v'

So the decoded string - aardv.

Step 9:- Now remaining string is only '0' hence it is the trave path from root to external symbol node which is (a)

hence the decoded string is - aardva

Note:- tree will be same as in encoding and decoding Procedure.

Golomb Code :-

⇒ **Unary code :-** the unary code for a +ve n is simply ~~n's~~ n is followed by a 0.

eg code for 4 - 11110
" " 7 - 11111110

⇒ Golomb-Rice codes belongs to a family of codes designed to encode integers with the assumption that :- "the larger an integer, the lower its probability of occurrence."

⇒ the unary code is the simplest coding for such situation.

⇒ unary code is optimal when $A = \{1, 2, 3 \cdots\}$ and probability model is used. $P(K) = \frac{1}{2^K}$

⇒ Golomb code is actually a family of codes parameterized by an integer $m > 0$.

⇒ In golomb code we take a parameter $m > 0$, ~~with an integer $n > 0$~~, and represent an integer $n \geq 0$ with using two numbers 'q' & 'r'

where $q = \left\lfloor \frac{n}{m} \right\rfloor$ & $r = n - qm$

[*q - quotient
*r - remainder]

⇒ 'q' can take on values 0, 1, 2 --- and represented by unary code of q.

⇒ 'r' = {0, 1, 2 -- m-1}.

⇒ (a) if m is a power of two, we use $\log_2 m$ bits to represent r

(b) if not then also use $\lceil \log_2 m \rceil$ bits.

⇒ 'r' is coded by fixed-length binary code {takes $\lfloor \log_2 m \rfloor$ or $\lceil \log_2 m \rceil$ bits}

⇒ 'r' is codes as follows —

⇒ (a) first $2^{\lceil \log_2 m \rceil} - m$ values represented by $\lfloor \log_2 m \rfloor$ bits

⇒ (b) rest of the values by $\lceil \log_2 m \rceil$ bit binary representation of $r + 2^{\lceil \log_2 m \rceil} - m$

hence Golomb code ⇒ codeward for q + code for 'r'

| q | r |

Golomb code.

eg :- Design a golomb code for $m = 5$ where $n = 0$ to $n = \{0, 1, 2, 3 \rightarrow 15\}$

Here $m = 5$; hence

$$q = \lfloor n/m \rfloor \quad \& \quad r = n - qm$$

⇒ for coding q we use unary code method.

⇒ $\lceil \log_2 5 \rceil = 3$ and $\lfloor \log_2 5 \rfloor = 2$

⇒ hence for coding 'r' -

① first $2^{\lceil \log_2 5 \rceil} - m \Rightarrow 2^3 - 5 \Rightarrow 8 - 5 = 3$ values of $r = (0, 1, 2)$ will be represented by 2 bit binary representation of r.

② Rest values of $r = (3, 4)$ will be represented by 3 bit binary representation of $(r + 2^{\lceil \log_2 m \rceil} - m) \Rightarrow [r + (2^3 - 5)]$

$\Rightarrow (r + 3)$.

hence codeward for $m = 5$ (Golomb method)

n	q	r	codeward
0	0	0	000 [0-q & 00 for r (2 bit)
1	0	1	001
2	0	2	010
3	0	3	0110 [r=3 so (r+3) with 3 bit)
4	0	4	0111 [r=4 so (r+4) " " ")
5	1	0	1000 [q=1 so single 1 followed by 0 & r=0 with 2 bit]

n	q	r	codeward
6	1	1	~~1011~~ 1001
7	1	2	1010
8	1	3	10110
9	1	4	10111
10	2	0	110-00
11	2	1	11001
12	2	2	11010
13	2	3	110110
14	2	4	110111
15	3	0	111000

(r = 1 2bit
(r = 2 -2 bit)

&

here q = 2 so
110

[q = 3 so
1110]

* [unary code for q]

⇒ Golomb code is optimal for the Probability model -

$$P(n) = p^{n-1}q$$

$$q = 1 - P$$

where $$m = \left\lceil -\frac{1}{\log_2 P} \right\rceil$$

Rice Codes :-

- Originally developed by Robert F. Rice.
- Rice code can be viewed as an adaptive Golomb code.
- In Rice code, a sequence of nonnegative integers is divided into blocks of J integers apiece.
- Each block is encoded with several options, ~~each of these~~ and (mostly from Golomb codes)
- Each block is encoded with each of these options, and the option resulting in the least number of coded bits is selected.
- The particular option used is indicated by an identifier attached to the code for each block.
- The implementation of rice code in the recommendation for lossless compression from the consultative committee on space Data standards (CCSDS)

CCSDS Recommendation for lossless Compression

→ This is an application of the Rice algorithm.

- This algorithm consists of a preprocessor and a binary coder. ie modeling step & coding step respectively.
- The preprocessor removes correlation from the input and generates a sequence of +ve integers
- the sequence has the property that smaller values are more probable than larger values,
- The Binary Coder generates a bitstream to represent the integer sequence.
- The Preprocessor functions as follows :-
  - given a sequence $\{y_i\}$ and generate a prediction $\hat{y}_i$ for each y_i
  - a simple way to generate a prediction is

$$\hat{y}_i = y_{i-1}$$

  - Then generate a sequence whose elements are the difference b/w y_i and its predicted value $\hat{y}_i$ ie

$$d_i = y_i - \hat{y}_i$$

  - The 'di' value will have a small magnitude when our prediction is good and a large value when it is not,

- Now let y_{max} and y_{min} be the largest and smallest values that the sequence $\{y_i\}$ takes on.
- Now assume that the value of $\hat{y}$ will be confined to the range $[y_{min}, y_{max}]$ define

$$T_i = \min\{y_{max} - \hat{y}_i, \hat{y}_i - y_{min}\}$$

- hence the sequence $\{d_i\}$ can be converted into a sequence of nonnegative integer $\{x_i\}$ using the following mapping :-

$$x_i = \begin{cases} 2d_i & -- \quad 0 \le d_i \le T_i \\ 2|d_i| - 1 & -- \quad -T_i \le d_i < 0 \\ T_i + |d_i| & -- \quad \text{otherwise} \end{cases}$$

- The value of x_i will be small whenever the magnitude of d_i is small
- ie the value of x_i will be small with higher probability.
- The sequence $\{x_i\}$ is divided into segments with each segment being further divided into blocks of size J.
- It is recommended by CCSDS that I have a value of 16

Now Each block is coded using one of the following options :-

(1) Fundamental sequence :- This is a unary code ie number 'n' is represented by a sequence of n 1s followed by a '0' or vice versa

(2) Split sample options :-

- These options consists of a set of codes indexed by a parameter m.
- The code for a k-bit number n using the m^{th} split sample option consists of the m least significant bits of k followed by a unary code representing the k-m most significant bits.

(3) Second extension option :-

- The second extention option is useful for sequences with low entropy- when in general, many of the values of x_i will be zero.
- In this the sequence is divided into consecutive pairs of samples. Each pair is used to obtain an index Y -

$$Y = \frac{1}{2}(x_i + x_{i+1})(x_i + x_{i+1} + 1) + x_{i+1}$$

- The value of γ is encoded using a unary code.
- The value of γ is an index to a lookup table with each value of γ corresponding to a pair of values x_i, x_{i+1}'.

4. **Zero block option :-**
   - The zero block option is used when one or more of the blocks of x_i are zero - generally when we have long sequences of y_i that have the same value.
   - In this case the nos. of zero blocks are transmitted using the code.

X. The Rice code has been used in several space applications.

Tunstall Codes :-

⇒ In the tunstall code, all codewards are of equal length. However, each codeward represents a different number of letters.

⇒ The main advantage of a Tunstall code is that errors in codewards do not propogate, unlike other variable-length codes, such as Huffman codes, in which an error in one codeword will cause a series of errors to occur.

⇒ The design of a code that has a fixed codeword length but a variable number of symbols per codeward should satisfy the following conditions :-

(1) We should be able to parse a source o/p sequence into sequences of symbols that appear in the codebook.

(2) We should maximize the average no. of source symbols represented by each codeward.

⇒ Tunstall gives a simple algorithm that fulfills these conditions.

⇒ The Tunstall's algorithm is as follows :-

Algorithm :-

- Suppose we want an n-bit Tunstall code for a source that generates iid letters from an alphabet of size N.
- The number of codewords is 2^n.
- Now start with N letters of the source alphabet in our codebook.
- Remove the entry in the codebook that has the highest probability and add the N strings obtained by concatenating this letter with every letter with alphabet (including self)
- This will increase the size of the codebook from N to N+(N-1).
- The probability of the new entries will be the product of the probabilities of the letters concatenated to form the new entry.
- Now look N+(N-1) entries in the codebook and find the entry that has the highest probability, keeping in mind that the entry with the highest probability may be a concatenation of symbols.

- Each time we perform this operation we increase the size of the codebook by N-1,
- Therefore, this operation can be performed K times, where

$$N + k(N-1) \leq 2^n$$

eg:- Design a 3-bit Tunstall code for a memoryless source with the following alphabet:- A = {A, B, C}

P(A) = 0.6, P(B) = 0.3, P(C) = 0.1

Letter	Probability
A	0.60
B	0.30
C	0.10

n = 3 bit
So no. of codewords = $2^n = 2^3 = 8$

Step:1 - first check the letter which has highest probability.

→ ~~Letter 'A' = 0.6~~ Letter A = .60 has the highest Probability, remove it from the list and add all two letter strings beginning with A.

→ After one iteration we have 5 entries in our codebook.

Sequence	Probability	
B	0.30	
C	0.10	
AA	0.36	A × A [.60 × .60]
AB	0.18	A B [.60 × .30]
AC	0.06	A C [0.60 × .10]

Step 2°:- Going through one more iteration in this we remove 'AA = 0.36' & add all strings begening with AA.

Sequence	Probability
B	0.30
C	0.10
AB	0.18
AC	0.06
AAA	0.216
AAB	0.11
AAC	0.036

Now total entries is 7 which less than the final code book size is 8.

Step 3°:- Going through the another iteration which increase the size of codebook 10, which is greater than the maximum size of 8. Therefore we just go only one iteration.

Hence the final 3 bit Tunstall code is.

~~for the 100~~

Sequence	Codeward
B	000
C	001
AB	010
AC	011
AAA	100
AAB	101
AAC	110

Now if we want to code the Msg

AAABAABAABAABAAA

~~100 000~~

AAA B AAB AAB AAB AAA

100 000 101 101 101 100

AAA - 100
B - 000
AAB - 101

$\Rightarrow$

Application of Huffman Coding :- (1)

(1) Lossless Image Compression :-

⇒ A simple application of Huffman coding to image compression would be to generate a Huffman code for the set of values that any pixel may take.

⇒ For monochrome images, this set usually consists of integers from 0 to 255.

⇒ Example of such images are contained in the accompanying data sets.

⇒ There are many s/ws to generate a Huffman code for each image, and then encode the image using the Huffman code.

⇒ The Huffman code is stored along with the compressed image as the code will be required by the decoder to reconstruct the image.

⇒ The original image, ~~compress~~ representation uses 8 bits/pixel i.e the image consists of 256 rows of 256 pixels.

so the uncompressed representation uses 65,536 bytes.

⇒ the number of bytes in the compressed representation includes the number of bytes needed to store the Huffman code.

⇒ Notice that the compression ratio is different for different images. This can cause some problems in certain applications where it is necessary to know in advance how many bytes will be needed to represent a particular data set.

Text Compression :- It is natural for Huffman coding. The reduction in file size is useful, we could have obtained better compression if we first remove the structure existing in the form of correlation b/w the symbols in the file obviously, there is a large amount of correlation in the text

Audio compression :- The CD Quality audio data is much susceptible for compression. Amount of data stored in one CD is enormous Transmitting this data would require huge channel Capacity compression is useful in this case.

=