Adaptive Huffman and arithmetic methods are universal in the sense that the encoder can adapt to the statistics of the source. But, adaptation is computationally expensive, particularly when k-th order Markov approximation is needed for some $k > 2$. As we know, the kth order approximation approaches the source entropy rate when $k \rightarrow \infty$. For example, for English text, to do second order Markov approximation, we will need to estimate the probability of all possible (about 35^3=42,875, 35 = {a-z,(,)....etc}) triplets, which is impractical. Arithmetic codes are inherently adaptive, but it is slow and works well for binary file.

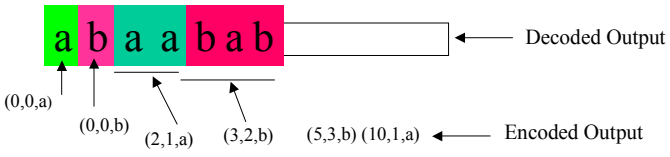
The dictionary-based methods such as the LZ-family of encoders do not use any statistical model, nor do they use variable size prefix code. Yet, they are universal, adaptive, reasonably fast and use modest amount of storage and computational resources. Variants of LZ algorithm form the basis of Unix compress, gzip, pkzip, *stacker* and for modems operating at more than 14.4 KBPS.

Dictionary Models

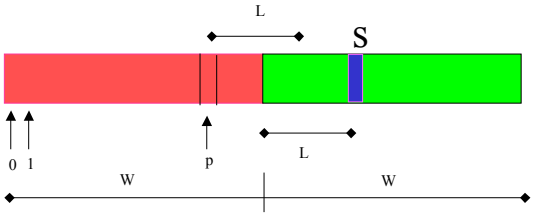
The dictionary model allows several consecutive symbols, called *phrases* stored in a dictionary, to be encoded as an address in the dictionary. Usually, an adaptive model is used where the dictionary is encoded using previously encoded text. As the text is compressed, previously encountered substrings are added to the dictionary. Almost all adaptive dictionary models originated from the original papers by Ziv and Lempel which led to several families of LZ coding techniques.
Here we will present a couple of those techniques.

LZ77 algorithms

The prior text constitutes the codebook or the dictionary. Rather than keeping an explicit dictionary, the decoded text up to current time can be used as a dictionary. The figure below shows the characters **abaabab** just decoded and the decoder is looking at the triplet (5,3,b) - number 5 denotes how far back to look into the already decoded text stream, number 3 gives the length of the phrase matched beginning the first character of yet un-encoded part of the text and the character 'b' gives the next character from input.
This yields '**aabb**' to be the next phrase added.



LZ77 Algorithm with Finite Buffer



Two buffers of finite size W, called the search(left) and the look-ahead(right)buffers are connected as a shift register. The text to be decoded is shifted in from right to left, initially placing W symbols in the right buffer and filling in the left buffer with the first character of the text. The information transmitted is (p,L,S) and the buffer is shifted L+1 places left. Actually, rather than transmitting p, the offset backward in the search buffer is transmitted. The process is repeated until text is fully encoded.

L= maximum length of the first substring from right end of the search buffer starting at position p that matches with a substring in the look-ahead buffer beginning at position 1.

S= the next symbol after the match in the right buffer.

aaaaaaaaa abacbacbb Output: (1,1,b)

aaaaaaaab acbacbbab Output: (2,1,c)

aaaaaabac bacbbabac Output: (3,4,b)

abacbacbb abacbacbc.. Output: (9,8,c)

Text= abacbacbbabacbacbc…

The decoding process is quite obvious. Since the first character is not known to the decoder, it is usually appended with a known dummy character agreed upon by the encoder and decoder. Also, note the Pattern being matched may spill over to to look-ahead as in step 3 above

♣ Read 5.3 and 5.4 from K. Sayood. Pp. 118-133.

♣ A formal description of LZ77 with Sliding Window *W*

The main idea of the algorithm is to use a dictionary to store the strings previously encountered. The encoder maintains a sliding window *W* in which the inputs are shifted from right to left. The window is split into two parts: The s*earch buffer,* which is the current dictionary*,* holding the recently encoded characters or symbols. The right part of the window is called l*ook-ahead buffer,* containing the text to be encoded. In practical implementation, the size of the search buffer could be several thousand bytes (8k or 16K) whereas the l*ook-ahead buffer* is very small (less than 100 bytes). The encoder searches the *search buffer* looking for the longest match beginning with the first character in the *look-ahead buffer*. The encoded output is a triple (*B, l, ch*), where *B* is the *distance traversed backwards* or the *offset* in the search buffer, *l* is the length of the match and *ch* is the next character in the *look-ahead* buffer for which the match fails. In case, *l*=0, *B*=0, the character *ch* keeps the encoding process going.

To encode text $T[1...N]$ with a sliding window of W characters.

Algorithm to Encode

Set $p \leftarrow 1$ /* p points to next charater in T to be coded */

While there is text remaining to be encoded do

{Search for first $T[p]$ in the search buffer;

If T[p] does not appear then {output (0,0,$T[p]$); p←p+1}
Else
{ suppose that matches occur at offsets $m_1 < m_2 < .... < m_s$ with lengths $l_1, l_2, ... l_s$. Let $l = max(l_1, l_2 ... l_s)$ at offset $m_{max} = m_i$ for some i, $1 \leq i \leq s$. If there are more than one l_i with same value of l, take the value of max closest to the end of the search buffer. Note, the value of p is incremented by an amount l while the pattern matching operation takes place.

Output triple ($B = m_{max}$, l, $Ch = T[p+1]$);

Set $p \leftarrow p + 2$}
endwhile

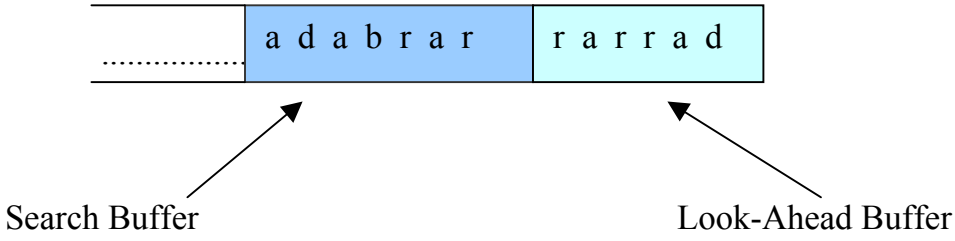
To Encode

/* Assume that the offsets are measured in the left direction beginning the last character of the search buffer while text is indexed always in the positive direction from left to right. */

Set p ← 1 /*next character of T to be decoded.*/
For each triple (B, l ,ch) input do
{If B=l=0 then {T[p]:=ch ; p←p+1;}
else { T[p,..p+l-1] ← T[B,B-1,…,B- l+1];
T[p+l] ← ch
p ←p +*l+1*;}
Shift buffer contents left by l+1 places}

In step 2 selecting the last match rather than the first or second, simplifies the encoder since the algorithm only has to keep track of the last string match details. But selecting the first match (greedy approach) may make the value of the offsets smaller and hence can be compressed further using a statistical coder such as Huffman (such a method by Berhard is called LZH).

Note, the string matching operation may begin at the *search buffer* but may spill over to the *look-ahead buffer*, which may even make the length *l* bigger than the *look-ahead buffer*.



The LZ77 method has been improved in the 1980's and 1990's by several ways:

- Use variable-size Huffman code for the length (*l*) and offset(B) fields. (A fixed format needs $\lceil \log_2 l \rceil$ bits to denote *l* for the *look-ahead buffer* and $\lceil \log_2 B \rceil$ bits for the *search buffer*.
- Increased sizes of the buffer to find longer and longer matches. The search time would increase. A more sophisticated data structure (TRIE) may improve the search time.
- Use a circular queue for the sliding window. In the sliding window, all the text characters have to be moved left after each match. A circular-queue avoids this.

Example:The different state of a 16-character buffer input : sid-eastman-easily (Example taken from David Soloman, p.157).

(a) (b)

In (a), a 16 byte array is shown with only 8 bytes occupied, S denoting start point and E denoting the end point. In (b), all 16 bytes are occupied

In (c), character ‘s’ deleted, and character ‘l’ inserted. Now , E is located left of S. In (d), two letters ‘id’ have been effectively deleted although they are still present in the buffer.

(e) (f)

. In (e), two characters ‘y-‘ has been appended and pointer E moved. In (f), the pointers show that the buffer ends at ‘teas’ and starts at ‘tman’. Inserting new symbols into the circular queue and moving the pointers is thus equivalent to shifting the contents of the queue. No actual shifting or moving is necessary.

- Eliminates the third element of the triple (ch) by adding an extra flag bit.

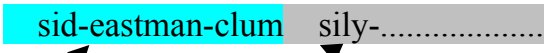
The improved version is called LZSS.

LZSS

- Uses a circular queue for *look-ahead buffer,*
- Holds search buffers (the dictionary) in a binary search tree, and
- It creates tokens with only 2 fields.

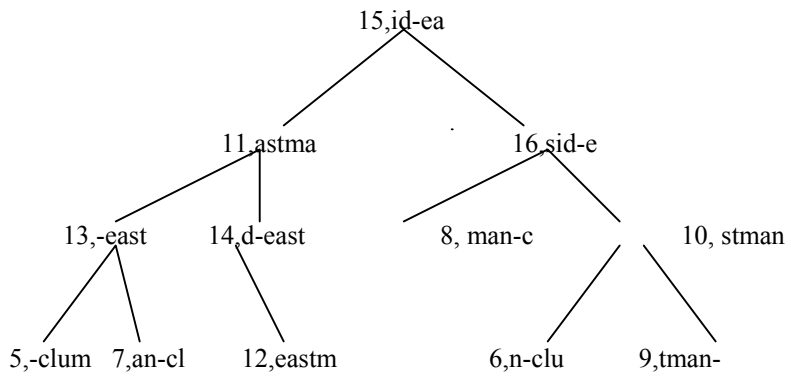
Example:

"sid-eastman-clumsily-teases-sea-sick-seals"



Temporary Search Buffer(16) Look-Ahead Buffer(5)

The encoder scans the search buffer creating 12 5-character strings (size of the look-ahead buffer), which are stored in a RAM along with a binary search tree, each node with its offset.



sid-e	16
id-ea	15
d-east	14
-east	13
eastm	12
astme	11
stman	10
tman-	9
man-c	8
an-cl	7
n-clu	6
-clum	5

The first symbol in the Look-Ahead buffer is 's'. Two words are found at offset 16 and 10 of which 16 leads to a longer match 'si' of length 2. The encoder emits (16,2). The next window is

sid-eastman-clumsily-te.....

The tree is updated by deleting 'sid-e' and 'id-ea' and inserting two new strings 'clums' and 'lumsi'. Note, the words deleted are always from the top addresses in RAM, and the words added are from the bottom of the RAM. This statement is true in general if there is a longer k-letter match. The window has to be shifted k positions.

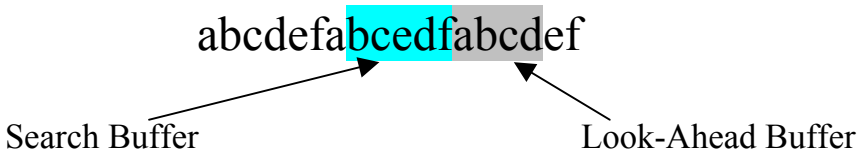
A simple procedure to update the tree is to take the first 5 letter word in the search buffer, find it in the tree, delete it, slide the buffer by one position to right, prepare a string consisting of the last 5 letters in the search buffer and add this to the tree. This has to be updated k times.

If the tree becomes unbalanced after several insertion and deletion, AVL-tree can be used. Note the number of nodes in the tree remains constant.

The token created has only two elements if no match is found; the character is transmitted without any change with a flag. The flags could be collected in 1 byte and 8 tokens could be transmitted together. Typical size of search buffer is 2 to 8 Kbytes and look-ahead buffer 32 bytes.

LZ78 (Lempel-Ziv-78)

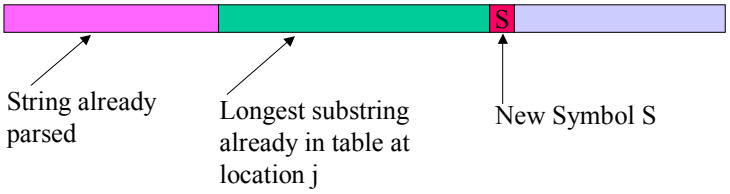
One of the major drawbacks of LZ77 is that there is an implicit assumption that like patterns occur close together so that they can be found during string matching operation. If the like patterns are separated by gaps longer than the buffer size, LZ77 will not compress at all. An extreme example is :



There will be no string match and each character will be sent with a flag, leading to expansion rather than compression. For another example, say the word "economy" occurs many times in the text but they occur sufficiently far away so that it will never be compressed. A better strategy will be to store the common occurring strings in a dictionary rather than letting them slide away. It means it does not have a window to limit how far back the substrings can be referenced. This is the basic principle of LZ78, which builds up the dictionary of common phrases. The decoder performs identical operation creating the same dictionary dynamically and in sync. The output is a sequence of tokens consisting of two items <*i, c*>, *i* = a pointer address to the dictionary and *'c*' is the next character.

LZ78 Algorithm

The family of LZ algorithms use an adaptive dictionary based scheme to compress text strings. The basic idea is to replace a substring of the text with a *pointer* (initially 0) in a table (codebook or dictionary) where that substring occurred previously.



Transmit (j,S) and repeat process beginning the next symbol after S. Enter at current *pointer +1* location the longest substring concatenated with with S. Initialize j=0.

Example

Message : aa_bbb_cccc_ddddd_e

Pointer	Longest Substring
1	a
2	a_
3	b
4	bb
5	_
6	c
7	cc
8	c_
9	d
10	dd
11	dd_
12	e

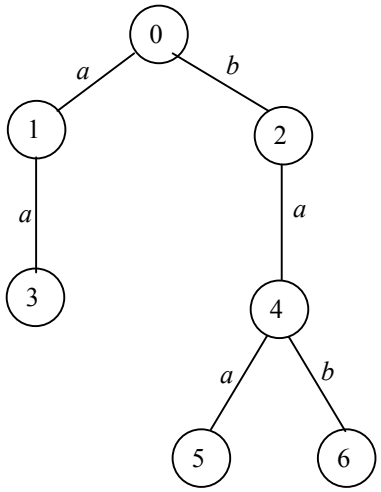
Transmitted Information (j,S)
0,a
1,_
0,b
3,b
0,_
0,c
6,c
6,_
0,d
9,d
10,_
0,e

The decoder can build an identical table at the receiving end.

The LZ78 can be looked upon as a parsing of the input strings as 'phrases', which are entered in the static dictionary. Thus, the string '*abaababaa*' is parsed into phrases '*a', 'b', 'aa', 'ba', 'baa'* and entered into phrase dictionary as

Phrase #	**Phrase**	**Output Token**
1	*a*	(0,*a*)
2	*b*	(0,*b*)
3	*aa*	(1,*a*)
4	*ba*	(2,*a*)
5	*baa*	(4,*a*)

where phrase number 0 stands for null phrase. Using a table to store the phrases is not very storage efficient. A more efficient method is to use a data structure called TRIE (or digital search tree) as shown below. The character of each phrase specifies a path from the root of the TRIE to the node that contains the number of phrase. The characters to be encoded are used to traverse the TRIE until the path is blocked either because there is no onward path for indicated character or leaf node is reached. The node at which block occur gives the phrase number for output. The character is appended to the output and a new node is created corresponding to a new phrase in the codebook or dictionary.



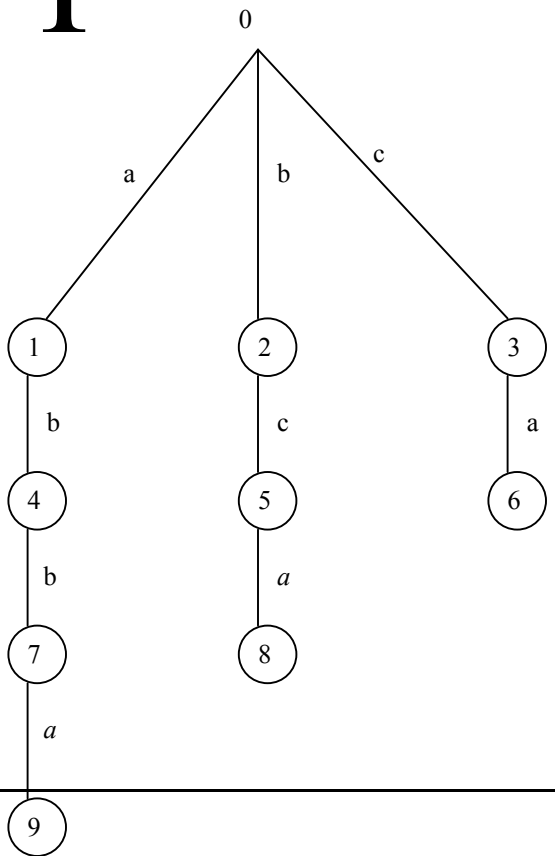
If the input alphabet is large, the TRIE may have several pointers emanating from each node which gives rise to the problem of allocating enough storage at the beginning of each node for all possible future pointers. A linked list data structure to represent sparse pointer array may do a better job. A faster and simpler method is to use a hash table in which the current node number and the next input character are hashed to determine where the next node can be found.

The TRIE data structure continues to grow as coding proceeds and eventually it may become too large. Several strategies can be used when memory is full. The TRIE is removed and the process is initialized again. Stop any further updates at the cost of less compression. Partially rebuild it using only the last few hundred bytes of coded text so that some knowledge from prior adaptation is retained.

Encoding for LZ78 is faster than LZ77 but decoding is slower since the decoder must store the parsed phrases. One variant of the LZ78 scheme, called LZW has been used widely in compression systems.

LZW (Lempel-Ziv-Welch Algorithm)

The main difference between LZW and LZ78 is that the encoding consists of a string of phrase numbers and the explicit 'next' character are not part of the output. This is done by initializing the dictionary or the TRIE with all letters of the alphabet.



Example 1

"*abcabbcabba*". The dictionary *D* is initialized with three nodes 1, 2 and 3 corresponding to the alphabet *A*=(*a, b, c*).

Encoding

a is in *D, ab* not in *D*, add 4,output 1
b is in D, *bc* not in *D*, add 5,output 2
c is in *D*, *ca* not in *D*, add 6,output 3
ab is in *D, abb* not in *D*, add 7,output 4

bc is in *D, bca* not in *D*, add 8,output 5
abb is in *D, abba* not in *D*, add 9,output 6
a is in D, output 1

Parsing: a b c ab bc abb a

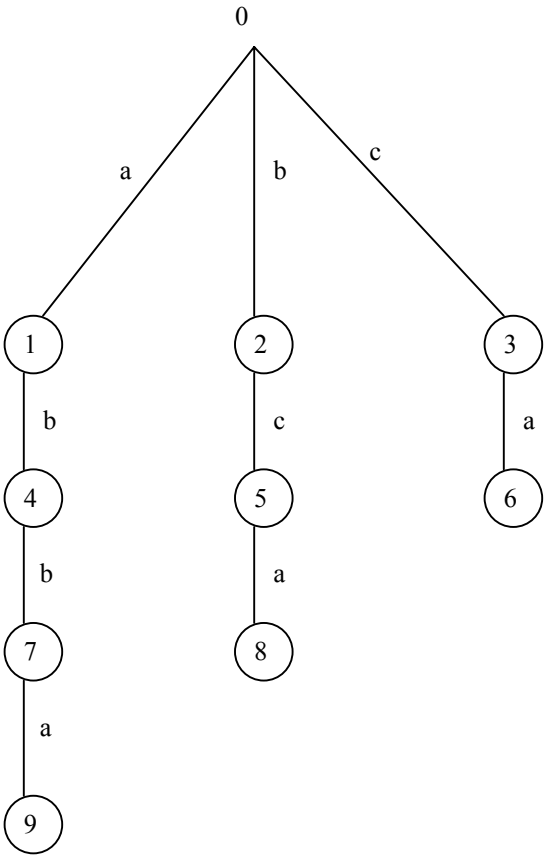
Encoder output: 1234571

The decoder does the reverse operation. It starts with initial dictionary *D* and keeps adding new no as it receive the node sequences from the encoder.

Decode 1234571

1	→	output *a*	→	*a* is in *D*
2	→	output *b*	→	*ab* not in *D* add 4
3	→	output *c*	→	*bc* not in D add 5
4	→	output *ab*	→	*ca* not in *D*, add 6
5	→	output *bc*	→	*abb* not in D add 7
7	→	output *abb*	→	*bca* not in D add 8
1	→	output *a*	→	*abba* is in D add 9

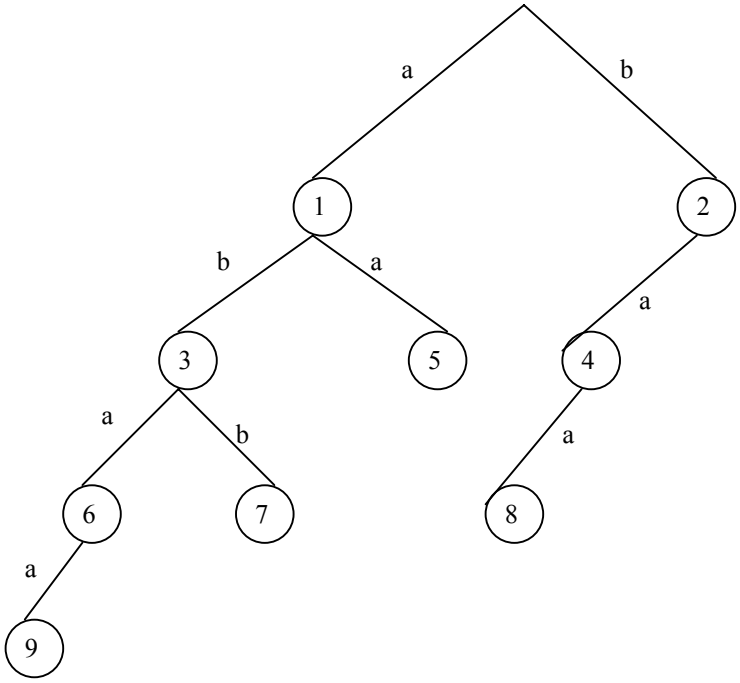
Note how it is creating a new node. Immediately, after putting the output, it creates a string : last phrase concatenated with the first character of the current phrase. If this is not in the dictionary, it creates a new node with the next available number.



Example 2

T = aba ab ab ba aba abaa

a b a ab ab ba aba abaa



Note the encoder has used the phrase 9 immediately after it has been constructed.
The final output of the encoder is: 12133469

Decoding

The decoding will proceed smoothly till number 6 producing output *abaababbaaba*…. and creating phrases upto 8 in the dictionary, but does not know what phrase 9 is! Fortunately, the decoder knows the beginning of new phrase – it is *abax*… where *x* is unknown yet. If we now concatenate the last phrase with this new phrase, the text should look like: ….*aba abax*…. But the phrase *aba a* is not in the dictionary so phrase 9 should have been *abaa* , which means that the character *x* is *a* . Thus phrase 9 must be '*abaa*' and decoding will proceed.

Whenever a phrase is referenced as soon as the encoder has created it, the last character of the phrase must be same as the first character.

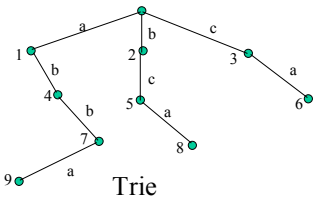
Despite this little problem in decoding, LZW works well giving good compression and efficient implementation.The following description of the algorithm is based on the description in WMB [1990]. Note ++ means concatenation

Encoding Algorithm

1	Set *p*=1 */* p, an index to text T[1…N]. */*
2	For *d* = 0 to *q-1* do *D[d] = d* */* D* is the TRIE and assume alphabet, *A=(0,1,2,..,q-1)* is represented by numbers which also denote the first *q* nodes or phrase numbers. **/*
3	*D = q-1* /* D points to last entry in the dictionary. The next node number starts at *q */*
4	While input stream not exhausted do 4.1 Trace TRIE *D* to find the largest match beginning *T[p]*. Suppose, the match terminate at phrase number *c* and the length of the match is *k* 4.2 output code *c* 4.3 *d = d+1* /* Add a new entry to TRIE. */ 4.4 *p=p+k* 4.4 Set *D[d] = D[c]++T[p]* */* Create a new phrase by connecting the last phrase with first character of next phrase. */*

LZW Algorithm

This algorithm eliminates the need to transmit the ‘next’ character as in the LZ78 algorithm.The dictionary is initialized to contain all characters in the alphabet. New phrases are added to the dictionary by appending the first character of next phrases . The algorithm is best described by using a ‘trie’ data structure to represent all distinct phrases in the dictionary . The algorithm is illustrated below.



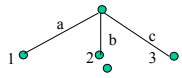
Trie

Alphabet = (a,b,c)

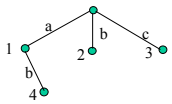
Text = abcabbcabba

Transmitted message = 1234571

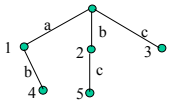
Text=abcabbcabba



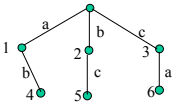
Text =a b..



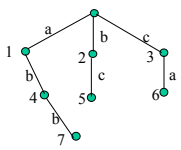
..b c..



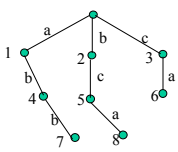
..c a..



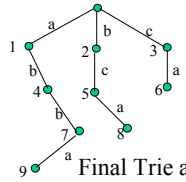
..ab . b..

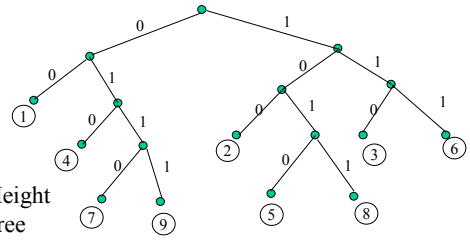


..bc a..



..abb a





Final Trie and its Height Balanced Binary Tree

Transmitted Code= 1234571=‘001001100101010011000’

Decoding Algorithm

Setp1,2,3 are same as in encoding setting up the initial TRIE or dictionary. Let the code sequence be $S=c_1\ c_2\ \ldots c_k$

```
Step 4:     Decode c1     - output D(c1)
Step 5:     for j=2 to k do
            begin
              If cj is in D, then {
                 output D(cj),Create a new_phrase by concatenating cj-1 with the
                 first character of ci  if this phrase is not in D ;
              }
              else
              {
                 new_phrase = D(cj-1)++F(cj-1);
                 Output new_phrase
              }
        /*F(ci-1) is the first character of the last phrase decoded.*/

                 d=d+1;

                D(d) = new  phrase  /*Enter a new  phrase number in D*/.
            end
```

LZW has been fine-tuned and has several variants. The Unix compress is one such variant. Compress uses a variable-length code to represent the phrase number and puts a maximum limit to the size of the phrase number. If afterwards the compression performance degrades, the dictionary is re-built from scratch.