

**A**daptive 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, particular 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 triplets (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, pzip, stacker and for modems operating at more than 14.4 KBPS.

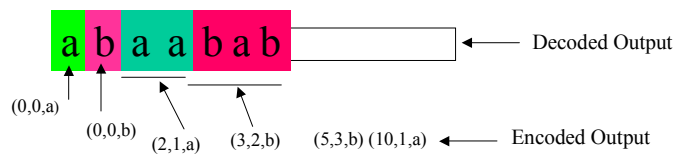
## Dictionary Models

**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.**

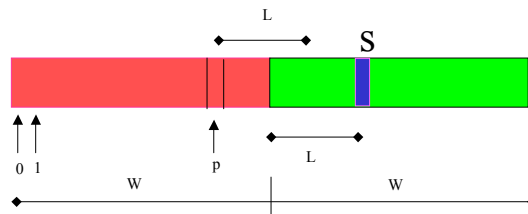
**Here we will present a couple of those techniques.**

# 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.

♣ 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 *search buffer*, which is the current dictionary, holding the recently encoded characters or symbols. The right part of the window is called *look-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 *look-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,  $l=0$ ,  $B=0$ , and 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 character 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  $(0,0,T[p])$ ;  $p \leftarrow p+1$ }

Else

{ suppose that matches occur at offsets  $m_1 < m_2 < \dots < m_s$  with lengths  $l_1, l_2, \dots, l_s$ . Let  $l = \max(l_1, l_2, \dots, l_s)$  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  $m$  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  $(B = m_{max}, l, Ch = T[p+l])$ ;

Set  $p \leftarrow p + l$

endwhile

## To Decode

*/\* 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  $p \leftarrow 1$  /\*next character of T to be decoded.\*/**

**For each triple (B, l, ch) input do**

**{If  $B=l=0$  then {T[p]:=ch ;  $p \leftarrow p+1$ };}**

**else { T[p,..p+l-1]  $\leftarrow$  T[B,B-1,..,B- l+1];**

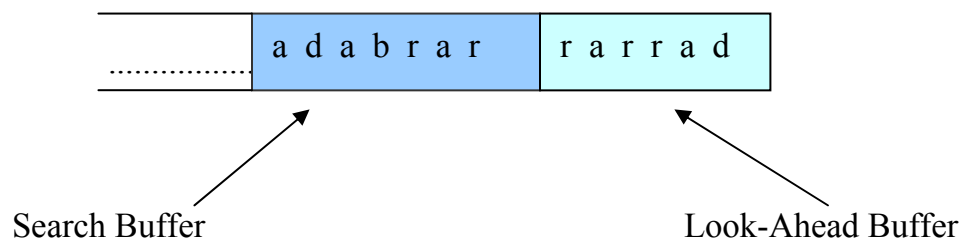
**T[p+l]  $\leftarrow$  ch**

**$p \leftarrow p + l + 1$ ;**

**Shift buffer contents left by l+1 places}**

In step 2 of encoding algorithm, 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).

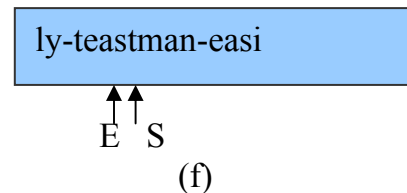
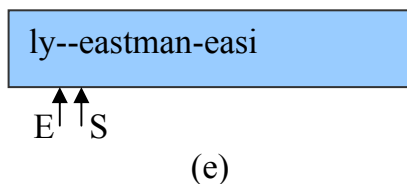
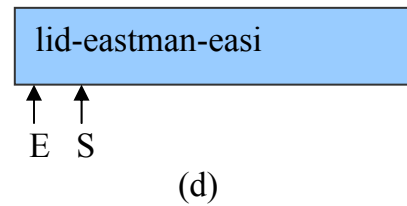
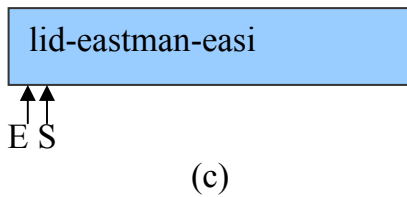
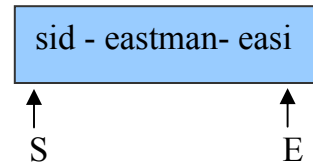
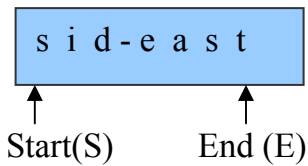
Note, the string matching operation may begin at the *search buffer* but may spill over to the *look-ahead buffer*, which may even make the length  $l$  bigger than the *look-ahead buffer*.



The LZ77 method has been improved in the 1980's and 1990's in 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.



- The different states of a 16-character buffer input : sid-eastman-easily (Example taken from David Soloman, p.157). 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 (by moving the start pointer to first '-') although they are still present in the buffer. In (e), two characters 'y-' have been appended and pointer E moved two places right. 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.

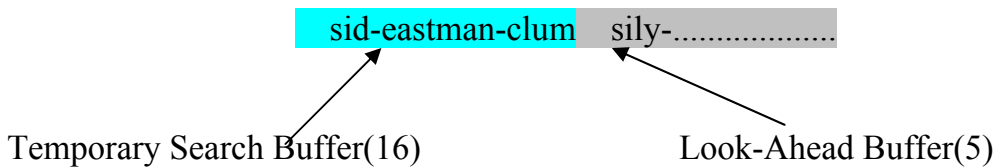
## LZSS

- Uses a circular queue for *look-ahead buffer*,

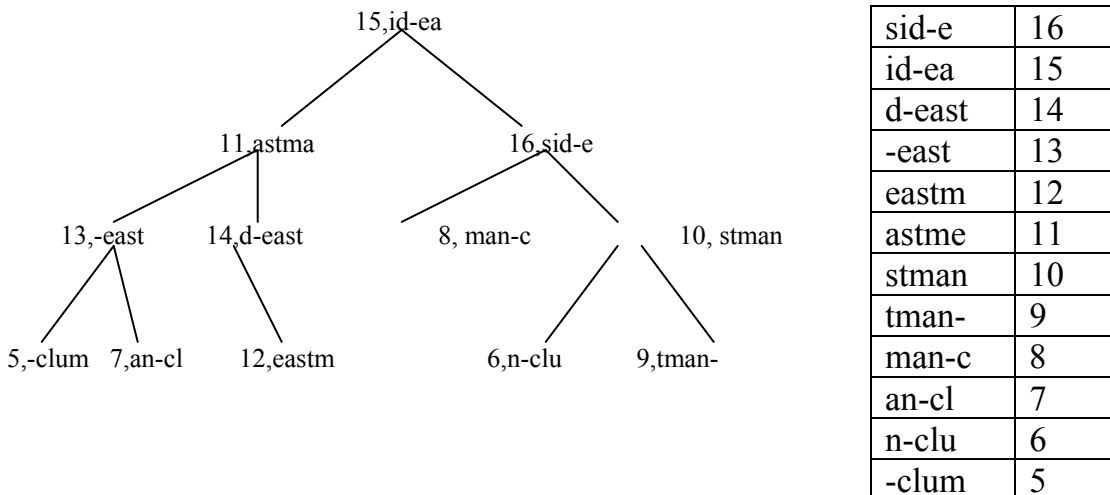
- Holds search buffers (the dictionary) in a binary search tree, and
- It creates tokens with only 2 fields.

Example:

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



The encoder scans the search buffer having 12 5-character strings, which are five consecutive characters or 5-grams of the search buffer. They are stored in a RAM along with a binary search tree [a height balanced search tree like AVL tree], each node with its offset. ( A binary search tree is a binary tree where the left subtree of every node X contains nodes smaller than X, and the right subtree contains nodes greater than X. A lexicographic 'less than' or 'greater than' relation is used for this purpose including the special characters. For this example, the character '-' is assumed to be smallest. The addresses 0-4 in the RAM holds the look ahead buffer.



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.

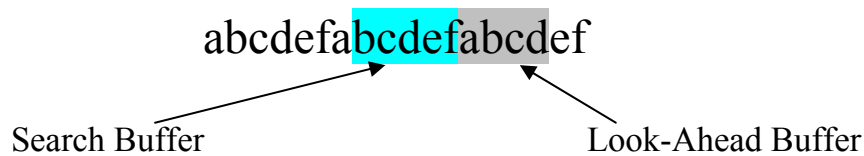
A simple procedure to update the tree is to take the first 5 letter word in the search buffer, find it in the tree, delete it, slide the buffer by one position to right, prepare a string consisting of the last 5 letters in the search buffer and add this to the tree. This has to be updated k times.

Since each update operation deletes an entry from the tree and also adds another entry in the tree, the size of the search tree never changes; it remains constant.. More precisely the number of nodes in the tree equals length of the search buffer minus length of the look ahead buffer plus 1. For this example, it is  $16-5+1=12$ . The RAM also contains same number of entries. The AVL tree keeps the height of the tree balanced after insertions and deletions.

In contrast to LZ 77, LZSS does not send a 3-field token. When there is match, it sends only two tokens (B, l) – the offset and length of the match. If there is no match, it will send the raw text character in its uncompressed (8 bit ASCII) format. To differentiate between tokens and uncompressed characters, each is preceded by a one bit flag. In practice, values are chosen such that a token can be fitted into multiple of bytes. A typical choice is search buffer size 2 Kbytes ( 11 bit address for B) and look ahead buffer size 32 ( 5 bits for l). With this choice the encoder will emit 2-byte tokens and 1-byte uncompressed characters. To preserve 'byte integrity', the flags are collected in 1 byte and 8 tokens/ASCII codes could be transmitted together. Thus eight output items are transmitted together: a 8-bit flag followed by 8 items each 1 or 2 byte long.

# LZ78 (Lempel-Ziv-78)

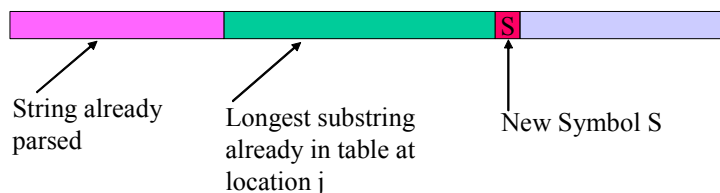
One of the major drawbacks of LZ77 is that there is an implicit assumption that like patterns occur close together so that they can be found during string matching operation. If the like patterns are separated by gaps longer than the search buffer size, LZ77 will not compress at all. An extreme example 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  $\langle j, S \rangle$ ,  $j$  = a pointer address to the dictionary and 'S' is the next character.

## LZ78 Algorithm

The family of LZ algorithms use an adaptive dictionary based on a 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 S. Initialize  $j=0$ .



## Example

Message : aa\_bbb\_cccc\_ddddd\_e

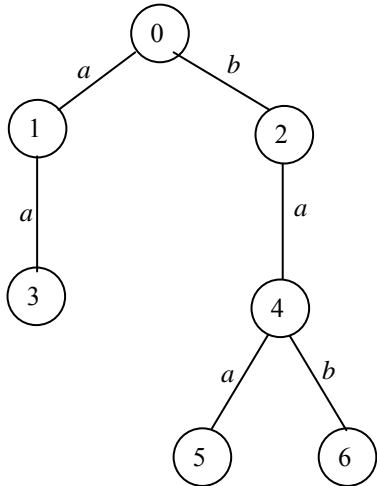
Pointer	Longest Substring	Transmitted Information (j,S)
1	a	0,a
2	a_	1,_
3	b	0,b
4	bb	3,b
5	_	0,_
6	c	0,c
7	cc	6,c
8	c_	6,_
9	d	0,d
10	dd	9,d
11	dd_	10,_
12	e	0,e

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	<i>a</i>	(0, <i>a</i> )
2	<i>b</i>	(0, <i>b</i> )
3	<i>aa</i>	(1, <i>a</i> )
4	<i>ba</i>	(2, <i>a</i> )
5	<i>baa</i>	(4, <i>a</i> )

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.



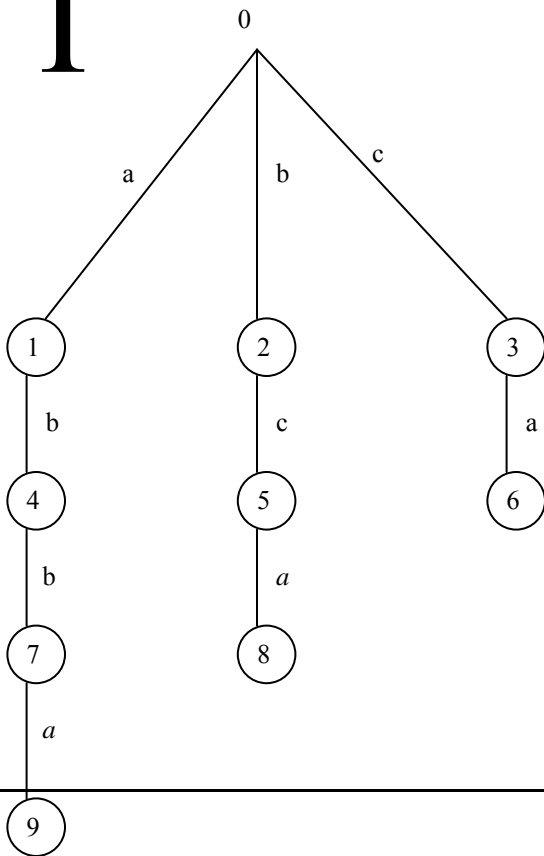
If the input alphabet is large, the TRIE may have only a few pointers emanating from each node which gives rise to the problem of allocating enough storage at the beginning of each node for all possible future pointers. A linked list data structure to represent sparse pointer array may do a better job. 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)

**T**



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

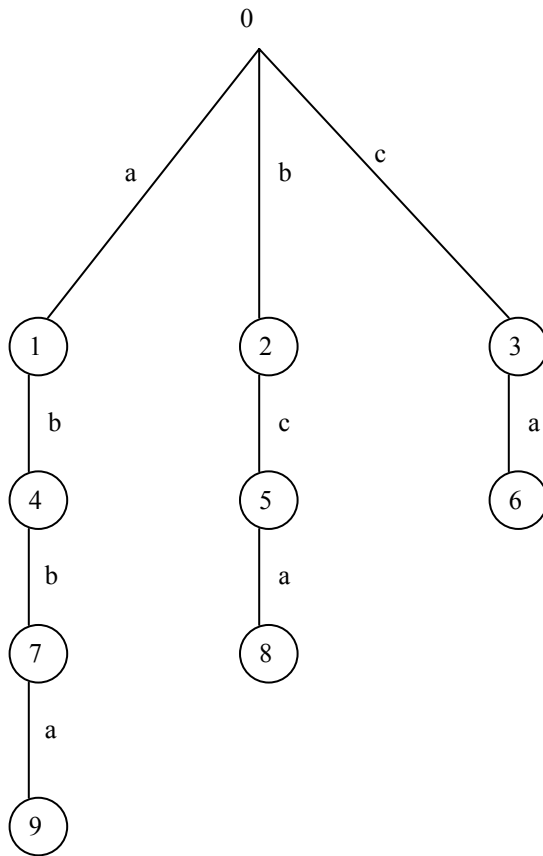
Parsing:      a b c ab bc abb a

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***

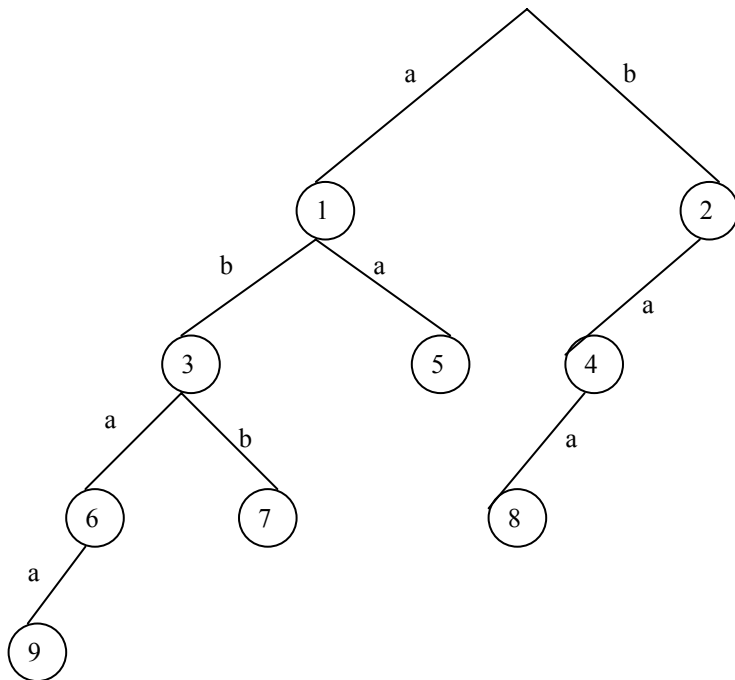
1	→	output <i>a</i>	→	<i>a</i> is in <i>D</i>
2	→	output <i>b</i>	→	<i>ab</i> not in <i>D</i> add 4
3	→	output <i>c</i>	→	<i>bc</i> not in <i>D</i> add 5
4	→	output <i>ab</i>	→	<i>ca</i> not in <i>D</i> , add 6
5	→	output <i>bc</i>	→	<i>abb</i> not in <i>D</i> add 7
7	→	output <i>abb</i>	→	<i>bca</i>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



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 *aba* –and that the last character of this phrase (which is unknown yet) is the first character of the phrase 6. Since, phrase 9 will be constructed by appending one character to phrase 6, phrase 9 must begin with same character as phrase 6, which 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 “Managing Gigabytes” [WMB,1990]. Note ++ means concatenation

## **Encoding Algorithm**

```
1      Set  $p=1$  /*  $p$ , an index to text  $T[1..N]$ . */
      /* assume alphabet,  $A=(0,1,2,\dots,q-1)$  */
       $q$ = size of the alphabet;
      previous-phrase= null;

2      For  $d= 0$  to  $q-1$  do  $D[d] = \text{character 'd'}$ 
      /*  $D$  is the TRIE with a root and  $q$  children. The leaves
      are numbered  $0,1, \dots,q-1$ . Each edge is recognized as
      a phrase  $D(d)$  */

3       $d=d+1$  /* The node number for next entry in the
      dictionary. The next node number starts at  $q$  */

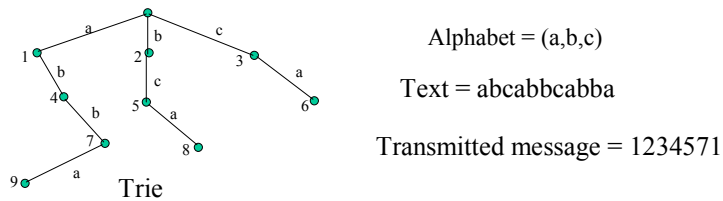
4      while input stream not exhausted do

      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  $l$ .
       $\text{New-phrase} = \text{previous phrase} + \text{first character of}$ 
       $\text{the matched pattern}$ ;
       $D(d) = \text{new-phrase}$ ;
      Add an edge for  $D(d)$  extending the TRIE by one
      node if  $D(d)$  already does not exist in  $D$ . /* The
      case where the edge already exists occurs only for
      phrases corresponding to single symbols. */
```

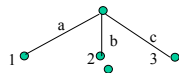
```
d = d+1 /* next entry to TRIE. */  
previous-phrase = new-phrase  
output code c  
p = p+l;  
endwhile
```

# LZW Algorithm

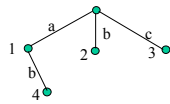
This algorithm eliminates the need to transmit the 'next' character as in the LZ78 algorithm. The dictionary is initialized to contain all characters in the alphabet. New phrases are added to the dictionary by appending the first character of next phrases. The algorithm is best described by using a 'trie'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



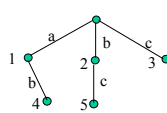
Text=abcabbcabba



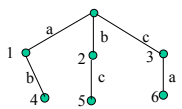
Text=a b..



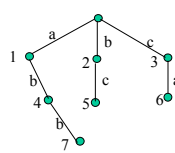
..b c..



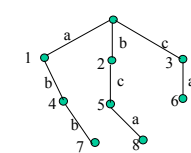
..c a..



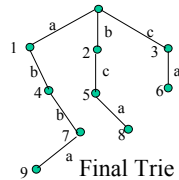
..ab . b..



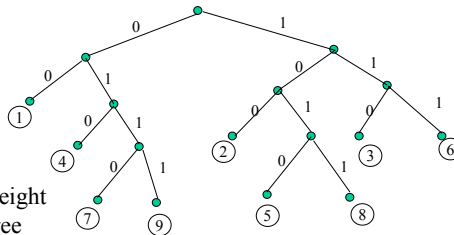
..bc a..



..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  $S=c_1 c_2 \dots c_k$

```

Decode  $c_1$  - output  $D(c_1)$ 
for  $j=2$  to  $k$  do
  begin
    If  $c_j$  is in  $D$ , then {
      output  $D(c_j)$ , Create a new phrase by concatenating  $c_{j-1}$  with the
      first character of  $c_j$ 
    }
    else
    {
       $D(c_j) = D(c_{j-1}) + F(c_{j-1})$ ;
      Output  $D(c_j)$ 
    }
  }
/* $F(c_{j-1})$  is the first character of the last phrase decoded.*/
  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.