

Description

The present invention relates to methods and apparatus for compressing data, methods and apparatus for decoding the compressed data, a method of transmitting data and data processing apparatus utilising compressed data. The above methods and apparatus all utilise an adaptive string encoding technique which involves a dictionary having a search tree and means for maintaining this search tree. The invention is applicable particularly but not exclusively to adaptive string encoding utilising the Ziv-Lempel algorithm. The basic form of this algorithm is described in IEEE Transactions IT-23, 3rd May 1977 pp337-343 "A Universal Algorithm for Sequential Data Compression" - J. Ziv and A. Lempel.

The basic Ziv-Lempel encoder has a dictionary, in which each entry has an associated index number. Initially the dictionary contains only the basic alphabet of the source. During the encoding process, new dictionary entries are formed by appending single symbols to existing entries. The dictionary may be considered to be in the form of a search tree of connected symbols of the source alphabet. Nodes in the tree correspond to specific sequences of symbols which begin at the root of the tree and the data is compressed by recognising strings of symbols in the uncompressed input data which correspond to nodes in the tree, and transmitting the index of the memory location corresponding to the matched node. A corresponding search tree is provided in the decoder which receives the index representing the compressed data and the reverse process is performed by the decoder to recover the compressed data in its original form. The search tree of the encoder gradually grows during the encoding process as further strings of symbols are identified in the input data and in order to enable the decoder to decode the compressed data, its search tree must be updated to correspond with the search tree of the encoder.

The Ziv-Lempel algorithm has been found difficult to implement in practice, since it requires an indefinitely large memory to store the search tree in its basic form. The use of data structures such as the "trie" structure disclosed by Sussenguth (ACM Sort Symposium 1962) can however greatly improve the storage efficiency and search time associated with text strings. EPA127815 (Miller & Wegman) and EPA129439 (Welch) disclose similar implementations of the Ziv Lempel algorithm based on the use of a trie structure.

In EPA127,815 (Miller & Wegman) improvements are described to the Ziv-Lempel algorithm which enhance the memory efficiency and speed up the encoding process. The dictionary is held in the form of a tree, with each node containing a single character and a pointer to the parent node which represents the prefix string. A hash table is used to determine, given a matched sub-string and the next input character, whether the extended sub-string is in the dictionary. However, the hash table requires a significant amount of memory and processing time in addition to that needed for the storage of the basic tree structure used to encode the dictionary.

EPA129,439 (Welch) discloses a high speed data compression and decompression apparatus and method in which strings of symbols in the input message are recognised and stored. Strings are entered into a string table and are searched for in the string table by means of a hashing function which utilises a hash key comprising a prior code signal and an extension character to provide a set of N hash table addresses where N is typically 1 to 4. The N RAM locations are sequentially searched and if the item is not in the N locations, it is considered not to be in the table. This procedure is stated to reduce compression efficiency but to simplify substantially the implementation.

US 4,612,532 (Bacon et al) discloses a system, not based on the Ziv-Lempel algorithm, for the dynamic encoding of a stream of characters in which each character is associated with a "follow set" table of characters which usually follow it, in order of the frequency with which they occur. These tables are of a pre-determined length and therefore the degree of branching of the tree is inevitably restricted.

US 4,464,650 (Eastman et al) discloses a method based on the Ziv-Lempel algorithm of compressing input data comprising feeding successive symbols of the data into a processor provided with a memory, generating from strings of symbols in the input data a dictionary in the form of a search tree of symbols in the memory which has paths representative of said strings, matching symbol strings in the input data with previously stored data paths in the search tree and generating from the stored paths compressed output data corresponding to the input data. However the data structure utilized for the circuitry is highly complex and furthermore a hashing function is required.

A particular problem inherent in the implementation of an encoder of the type described occurs when the search tree grows to the limit of the search tree in order to recover memory space for the storage of new strings (ie "prune"). A number of well known methods exist for performing this type of function, which are reviewed in Computer Architecture and Parallel Processing (Hwang & Briggs, McGraw Hill 1985). The commonly used techniques are LRU - Least Recently Used, applied to the Ziv Lempel algorithm by Miller & Wegman (EPA127815), LFU - Least Frequently Used, applied to a similar string encoding algorithm by

Mayne & James (Information Compression by Factorising Common strings, Computer Journal, 18,2 pp 157-160, 1975), FIFO - First In First Out, LIFO - Last In First Out, the CLOCK algorithm, and Random replacement, the last four techniques cited have not been applied to the Ziv-Lempel algorithm. In addition, it is known to reset the search tree back to the initial state which bears a penalty in terms of compression performance, and also to cease adding new strings when the memory capacity is exhausted which will give poor performance if the characteristics of the data change.

An object of the present invention is to provide an improvement with regard to these prior art techniques. Accordingly in one aspect the invention provides a method of compressing input data comprising reading successive symbols of the data with a processor provided with a memory having indexed memory locations, generating from strings of symbols in the input data a dictionary in the form of a search tree of symbols in the memory, which search tree has paths representative of said strings, matching symbol strings in the input data with previously stored paths in the search tree and generating from the stored paths compressed output data corresponding to the input data; stored symbols in the search tree being linked to form said paths by linking pointers of two distinct types; a pointer of the first type between stored symbols indicating that those stored symbols are alternative possible symbols at a given position in an input symbol sequence and a pointer of the second type between stored symbols indicating that those stored symbols both occur, in order, in a possible input symbol sequence: characterised in that when the memory is full, sequential indexed memory locations of the search tree are tested and deleted if they contain a node of the search tree which does not have a linking pointer of the second type pointing to another node and the resulting freed memory locations are made available for new dictionary entries.

In the abovementioned one aspect, the testing and deletions of indexed memory locations of the search tree corresponds to the deletion of some or all of those strings represented by the tree that do not form prefixes of other strings. This feature enables highly complex search trees to be stored in memories of limited size and provides a useful simplification in comparison with the arrangements of US 4,464,650 (Eastman et al) and EPA 127 815 (Miller & Wegman).

US 4,464,650 also discloses a corresponding method of decoding compressed data comprising feeding successive symbols of the data into a processor provided with a memory, storing in the memory a dictionary in the form of a search tree of symbols, and utilising the search tree to translate the compressed data to decoded data.

In another aspect, the invention provides a method of decoding compressed data which has been compressed by a method in accordance with the abovementioned one aspect of the present invention, comprising reading successive characters of the compressed data with a processor provided with a memory, storing in the memory a dictionary in the form of a search tree of symbols which is built up from the compressed data, and utilizing the search tree to translate the compressed data to decoded data, stored symbols in the search tree being linked by linking pointers of two distinct types; a pointer of the first type between stored symbols indicating that those symbols are associated with different decoded strings of symbols having the same number of symbols and the same prefix and are the respective last symbols of such different strings and a pointer of the second type between stored symbols indicating that those symbols are successive symbols in a string of decoded output symbols: characterized in that when the memory is full, sequential indexed memory locations of the search tree are tested and deleted if they contain a node of the search tree which does not have a linking pointer of the second type pointing to another node and the resulting freed memory locations are made available for new dictionary entries.

US 4,464,650 also discloses an encoder for compressing input data comprising a processor capable of receiving successive symbols of input data, a memory, means for storing in the memory a search tree of symbols which has paths representative of strings of symbols in the input data, and means for matching symbol strings in the input data with previously stored paths in the search tree and for generating from the stored paths compressed output data corresponding to the input data.

In another aspect the invention provides an encoder for compressing input data comprising a processor capable of receiving successive symbols of input data, a memory having indexed memory locations, means for storing in the memory a search tree of symbols which has paths representative of strings of symbols in the input data, and means for matching symbol strings in the input data with previously stored paths in the search tree and for generating from the stored paths compressed output data corresponding to the input data, stored symbols in the search tree being linked to form said paths by linking pointers of two distinct types; a pointer of the first type between stored symbols indicating that those stored symbols are alternative possible symbols at a given position in an input symbol sequence and a pointer of the second type between stored symbols indicating that those stored symbols both occur, in order, in a possible input symbol sequence: characterised in that, in use, the processor is arranged to determine when the memory is full, to test sequential indexed memory locations of the search tree, to delete a memory location if it

contains a node of the search tree which does not have a linking pointer of the second type pointing to another node and to make the resulting freed memory locations available for new dictionary entries.

US 4,464,650 also discloses a decoder for decoding compressed data comprising a processor capable of receiving successive symbols of compressed data, a memory and means for storing in the memory a dictionary in the form of a search tree of symbols, the processor being arranged to utilise the search tree to translate the compressed data to decoded data.

In another aspect, the invention provides a decoder for decoding compressed data comprising a processor capable of receiving successive symbols of compressed data, a memory having indexed memory locations and means for storing in the memory a dictionary in the form of a search tree of symbols, the processor being arranged to build up the search tree from the compressed data and to utilise the search tree to translate the compressed data to decoded data, stored symbols in the search tree being linked by linking pointers of two distinct types; a pointer of the first type between stored symbols indicating that those symbols are associated with different decoded strings of symbols having the same number of symbols and are the respective last symbols of such different strings and a pointer of the second type between stored symbols indicating that those symbols are successive symbols in a string of decoded output symbols: characterised in that, in use, the processor is arranged to determine when the memory is full, to test sequential indexed memory locations of the search tree, to delete a memory location if it contains a node of the search tree which does not have a linking pointer of the second type pointing to another node and to make the resulting freed memory locations available for new dictionary entries.

Further preferred features of the invention are claimed in the dependent claims.

A further preferred feature which may be utilised in the present invention is described by Welch in the article "A Technique for High-Performance Data Compression" in Computer - June 1984 pp8-19. This article is incorporated herein by reference and describes a method of improving the initial performance of the Ziv-Lempel algorithm. Initially, the dictionary is almost empty and a small number of bits suffice to encode the symbol strings of the input data. As the dictionary grows, the code word size is increased up to a predetermined maximum. This improves performance during the first ten to 20 thousand symbols encoded, but at the cost of increased complexity.

Embodiments of the invention are described below by way of example only with reference to Figures 1 to 13 of the accompanying drawings, of which:

Figure 1 shows one possible search tree structure for use in the invention;

Figure 2 shows another, preferred structure of a search tree for use in the invention;

Figure 3 shows the evolution of a search tree during a data compression process in accordance with the invention;

Figure 4 is a general flow diagram of an encoding algorithm which is utilised in the invention;

Figure 5 is a general flow diagram of a decoding algorithm which is utilised in the present invention;

Figure 6 is a diagram showing the insertion of a "leaf" of a search tree in a data compression process in accordance with the invention;

Figure 7 is a flow diagram showing an algorithm for updating the dictionary of the present invention;

Figure 8 shows the pruning of the search tree of Figure 3 (f);

Figure 9 is a flow diagram of an algorithm for pruning the dictionary of the invention;

Figure 10 is a schematic diagram of a data processing system utilizing an encoder in accordance with the present invention;

Figure 11 is a schematic diagram of a data processing system utilizing a decoder in accordance with the present invention;

Figure 12 is a schematic diagram of part of a communications system incorporating an encoder and decoder in accordance with the present invention, and

Figure 13 is a diagram showing the arrangement of the indexed memory locations of the memory.

Referring to Figure 1, the simplified search tree shown comprises a symbol S1 which may be followed by any one of three different symbols S2, S3 and S4. As an example, symbols S1 to S4 have been assigned the values 'c', 'a', 'e' and 'h', respectively, and the tree thus represents the four strings 'c', 'ca', 'ce' and 'ch'. Accordingly symbol S1 is linked to symbol S2 by a linking pointer "of the second type" as referred to herein above, namely a down pointer D. Since symbols S2, S3 and S4 are different possible symbols which may occur in a string in the input data after S1, they are linked by pointers "of the first type" as referred to herein above, namely right hand pointers R and left hand pointers L. Note that similar technology is used in the representation of binary trees (refer to, for example, "The Art of Computer Programming", Volume 1 and 3, by D. Knuth); this data structure is however being used to represent an m-ary search tree.

It is possible to search through the search tree from symbol S1 to any of symbols S1, S3 and S4 by using the linking pointers D and R and to search to the left of symbol S4 by following the left hand pointers L. In Figure 1(b) two symbol locations are shown linked by a pointer and these constitute a list of free memory locations. This list is separate from the search tree but memory locations may be transferred between the free list and the search tree, the total memory capacity for symbols of the input data consisting of the memory locations in the search tree and the memory locations in the free list. In order to identify any of these symbols S1 to S4 it is necessary to know the "parent" of that symbol, namely the symbol connected to the immediately preceding down pointer D. This is indicated in each case by a parent indicator P for example, if the letter 'c' was stored as symbol S1, it would be the parent of S2, S3 and S4 and their parent indicators P would indicate the memory location of 'c'. In the embodiment of Figure 1, the parent indicators P are required in the search tree of the decoder only, and not in the search tree of the encoder. Otherwise the search trees of the encoder and decoder are identical.

The nodes within the search tree of which node S is a parent will be termed dependent nodes of S, or dependants of S. A dependent node of S represents a string constructed by appending a single character to the string represented by S. There can be as many dependent nodes of some node as exist symbols in the source alphabet.

Figure 2 shows a more preferred variant in which the left hand pointers L are dispensed with and the parent indicators P occur in the search tree of both the encoder and the decoder. Otherwise the search tree structure is identical to that shown in Figure 1. Again a free list of symbol locations is maintained in the memory as shown in Figure 2(b).

The dictionary or search tree will normally be initialised to include only the basic set of source symbols, subsequent entries or nodes representing strings of two or more symbols in length. In many practical applications it is desirable to provide additional symbols outside the normal source alphabet. These may represent for example a means for encoding repeated occurrences of a character (run length encoding) or indicating abnormal termination of the string matching process. (see Figure 13).

The encoding of the sequence abcabababc is described below with reference to Figure 3 which shows the evolution of the search tree during the encoding process and also table I which shows the actual contents of the dictionary which represent the search tree shown in Figure 3. As is shown below, the search tree is expanded by setting the pointers D R and P of dictionary entries 1 to M, the maximum permitted by the memory available.

TABLE I(a)

Index	Symbol	D	R	P	String Represented
1	a		(2)		a
2	b		(3)		b
3	c				c

As shown in Table I(a) and Figure 3(a) the dictionary initially contains only symbols a, b and c which are stored in dictionary entries 1, 2, and 3 respectively. They may be stored in a predetermined order and accordingly dictionary entry 1 is linked to dictionary entry 2 by a right hand linking pointer 2 which is located at dictionary entry 1 and dictionary entry 2 is linked to dictionary entry 3 by a right hand linking pointer 3 which is stored in dictionary entry 2. A preferred method is to equate the ordinal value of each symbol to its position in the table. Thus the set of N symbols would be assigned ordinal values in the range 0 to N-1 or 1 to N. This assignment may involve simply regarding the binary pattern representing the symbol as a number. These contents of the memory imply a structure as shown in Figure 3(a).

In many applications only some of the possible symbols occur, in which case the first of these structures would be preferable. In those applications in which most of the possible symbols occur, the second method will reduce the initial character search time, accordingly the second method will be assumed below.

The first symbol of the sequence is input to the encoder, in this example 'a'. As the symbol represents the first character in a string the encoder uses the ordinal value of the character, in this example 1, to access directly the node within the search tree corresponding to the character. This node forms the root of a tree representing all strings known to the encoder that begin with the character.

- 5 The next symbol of the sequence is accepted by the encoder, and the D pointer of the current node used to locate the list of dependent nodes. The encoder would in general attempt to match the next symbol to one of those contained in the dependent nodes. If the attempt fails, as in this instance because node 1 does not yet have any dependants, a codeword representing the index of the current node/dictionary entry is transmitted, in this example some representation of the numeral 1. The unmatched symbol will be used
10 to form the start of a new string.

- The encoder is now able to add the string 'ab' to its search tree, in order that the string may be more efficiently encoded when encountered again. This is accomplished by creating a new node (entry) numbered 4 in this example, in the search tree (dictionary). The node contains the character, 'b', a D pointer to dependent nodes, initially set to null, an R pointer to other nodes in the list of dependents of the
15 node's parent, in this instance set to null, and a pointer to the parent, set in this example to 1, as shown in Table I(b) and Figure 3(b)

TABLE I(b)

Index	Symbol	D	R	P	String Represented
25 1	a	4			a
2	b				b
3	c				c
30 4	b			1	ab

- The process is repeated, with 'b' used as the first character of a new string. The next symbol of the sequence is 'c' hence the encoder will attempt, using the process outlined above, to find the string 'bc' in the tree. This sequence is not yet known to the encoder, hence the index of the node representing the
35 string 'b' will be communicated to the decoder, and the string 'bc' added to the search tree. The character 'c' is used to form the start of a new string. The updated dictionary is shown in Table I(c) and the corresponding search tree in Figure 3(c).

TABLE I(c)

Index	Symbol	D	R	P	String Represented
45 1	a	4			a
2	b	5			b
3	c				c
50 4	b			1	ab
5	c			2	bc

- The next symbol 'a' is now read, being in this instance the fourth symbol in the sequence, and
55 appended to the search string. The encoder now attempts to locate the string 'ca' in its dictionary. As this string is not yet present, the index of the dictionary entry corresponding to 'c' will be communicated to the decoder and the string 'ca', which may also be considered as an index/character pair (3, 'a') added to its dictionary (search tree). This produces the dictionary shown in Table I(d) and search tree shown in Figure

3(d). The unmatched character 'a' is used to start a new search string.

TABLE I(d)

Index	Symbol	D	R	P	String Represented
1	a	4			a
2	b	5			b
3	c	6			c
4	b			1	ab
5	c			2	bc
6	a			3	ca

The next symbol 'b' is now read and appended to the search string, forming the string 'ab'. The D pointer corresponding to 'a' is used to locate the list of 'a's dependants; this pointer has the value 4 in this example. The encoder now searches for the last character in the search string amongst the list of dependants, using the R pointer. In this example the first dictionary entry examined, with index 4, contains the character 'b' and hence the encoder has matched the search string with a dictionary entry.

The next symbol 'a', being the sixth character in the input sequence, is read by the encoder and appended to the search string forming the string 'aba'. The encoder uses the D pointer of the last matched dictionary entry, numbered 4 in this example, to locate the list of dependants of the string 'ab'. As yet no dependants are known to the dictionary, hence the new string 'aba' or (4, 'a') is added to the dictionary, as shown in Table I(e) and Figure 3(e). The index 4 is communicated to the decoder, and the unmatched character 'a' used to start a new search string.

TABLE I(e)

Index	Symbol	D	R	P	String Represented
1	a	4			a
2	b	5			b
3	c	6			c
4	b	7		1	ab
5	c			2	bc
6	a			3	ca
7	a			4	aba

This corresponds to Figure 3(e).

The encoder reads the next symbol and appends it to the search string, forming the new search string 'ab'. The encoder searches in the dictionary for this string, using the procedures outlined above, and matches it with dictionary entry 4. The next symbol 'c' is read and appended to the search string, forming 'abc'. The encoder attempts to locate this string, by searching through the dependant list of 'ab', but fails to find 'abc'. The index for 'ab', namely 4, is communicated to the decoder, and the new string (4, 'c') added to the dictionary as entry number 8, as shown in Table I(f) and Figure 3(f).

TABLE I(f)

Index	Symbol	D	R	P	String Represented
1	a	4			a
2	b	5			b
3	c	6			c
4	b	7		1	ab
5	c			2	bc
6	a			3	ca
7	a		8	4	aba
8	c			4	abc

During the next string encoding cycle the encoder would match the string 'ca' with entry 6 in the dictionary, and transmit 6 to the decoder, and add 'cab' or (6, 'b') to the dictionary. During the following cycle, the encoder would match the string 'bc' with entry 5 in the dictionary, and add (5, x) to the dictionary, where x is the character following the last in the sequence shown.

The sequence 'abcababcabc' would thus have been encoded using the list of index values 1234465, which provides a small degree of compression. If however the same sequence were encountered again by the encoder it would be encoded as 'abc' (dictionary entry 8), 'aba' (dictionary entry 7), 'bc' (dictionary entry 5), and 'abc' (dictionary entry 8), resulting in the transmitted sequence 8758. If the sequence of symbols occurred at least twelve times the encoder may represent it by a single index value, thereby obtaining a high degree of compression.

The encoding algorithm is illustrated more formally in Figure 4. In step i the variables are initialised and then, repeatedly, the longest possible sequence of characters in the input message is mapped onto a string of the search tree in step ii, for example the sequence bc at the end of the input message is mapped onto the string bc in Figure 3(f). In step iii the index of the terminal node corresponding to the matched string (eg bc) or some further encoded representation thereof, is sent, and the search string set to the following character of the input message (step iv).

The process of string matching would normally be terminated when the string with the next character appended is not in the dictionary. There are however other instances in which string matching may be terminated exceptionally, for example if no character is received from the source within some specified time interval from the last character, or if the string length reaches some maximum beyond which a buffer limit may be exceeded, or if some specified time interval has occurred since the encoder began encoding the string. In the second of these examples the decoder could deduce that exceptional termination had occurred. In the first and third examples it is necessary for the encoder to transmit an indication, following transmission of the index representing the terminated string; this indication may be an additional control symbol outside the ordinal value range of the source alphabet, and would be encoded as a single codeword.

The index of the matched string and the next unmatched character must be added to the dictionary. If however, they are immediately added to the dictionary, the encoder could start using the new entry before the decoder has received sufficient information to construct an equivalent dictionary entry. Thus the previous index/character pair are added to the dictionary (v) and the present index/character pair stored (vi). If the dictionary is full it may be processed to recover some storage space. The process of adding a new dictionary entry and recovering space are further described below. In order that the dictionary entry referenced by the stored index/character pair is not deleted before being used in the updating process, the dictionary entry is marked as 'new' using the procedure further described below.

Figure 5 shows the corresponding decoding algorithm. The variables are initialised in step i. A received codeword is initially decoded in step ii to recover the transmitted index, J, representing the encoded string. The value of J allows the decoder to access directly the dictionary entry corresponding to the encoded string. The tree is then re-traced from entry J back to the root of the string, and the string is read out in step iii. For example if the value 8 were received by a decoder having the search tree of Figure 3(f) it would find the symbol c at the "leaf" of the tree at dictionary entry 8 and would re-trace the P pointers 7 and 4 to the root a and would therefore read out the sequence cba. This sequence is then reversed in the subsequent step iv to regenerate the original sequence abc and the decoder algorithm then proceeds to update the dictionary provided it is not inhibited a control character in step v.

An example of updating the search tree is shown in Figure 6 and the corresponding memory contents are shown in Table II.

TABLE II

Index	Character	D	R	P	New Entry	String Represented
1	a	4			0	a
2	b	5			0	b
3	c	6			0	c
4	b	7		1	0	ab
5	c			2	0	bc
6	a			3	0	ca
7	a		<u>n</u>	4	0	aba
8	c		<u>—</u>	4	0	abc
n	<u>b</u>		<u>8</u>	<u>4</u>	<u>1</u>	abb

It will be seen that Table II corresponds to Table I(f) but with the addition of the sequence abb at dictionary entry n. In order to insert this sequence in the search tree, the link between nodes 7 and 8 is broken, and the new node connected between these, as shown in Table II and Figure 6. The new right hand linking pointers R having the values n and 8 are shown underlined in Figure 6, together with the new symbol b (also underlined). The change to the search tree illustrated conceptually in Figure 6 is actually performed by changing the value of the R pointer at dictionary entry 7 in Table II from 8 to n and by inserting new R and P pointers 8 and 4 at dictionary entry n.

The updating algorithm is shown formally in the flow diagram of Figure 7. This applies to both the encoder and decoder. Firstly, it is determined whether the dictionary is full in step i. If the dictionary is full, the search tree must be pruned and this procedure is described below with reference to Figure 9. However, assuming that this is not the case, a blank memory slot is obtained from the free list (shown in Figure 2(b)) in step ii. The new symbol (b in the case of Table II) is then written into the memory slot (n in the case of Table II) and the necessary pointers (R at dictionary entry 7 in the case of Table II) are re-set in step iii. In step iv the parent node of the new entry is examined to see if it has any existing dependants. If there are existing dependants, then in step v the new entry is inserted into the dependant list as in the example above. If there are no existing dependants, then in step vi the new entry is connected to the parent node, setting the D pointer of the parent to the memory reference of the new entry, and the P pointer of the new entry to the memory reference of the parent node.

Referring first to Figure 8, Figure 8(a) shows the state of the search tree following the updating procedure illustrated in Table II and Figure 6. Those symbols of the search tree which do not have D linking pointers extending downwardly from (and which therefore occur at the end of matched symbol sequences) are considered to be unlikely to be useful if the search tree is not growing at those points. Such "dead leaves" of the search tree are illustrated in Figure 8(a) in dashed lines and the new entry b which was added in the updating procedure illustrated in Figure 6 is marked as such in order to indicate that it is a "new leaf" from which further growth of the search tree might be expected.

Accordingly it is protected from pruning. It is envisaged that other leaves of the tree which were added in previous recent iterations might also be protected.

The symbols of the search tree shown in dashed lines in Figure 8(a) are pruned, resulting in the new search tree of Figure 8(b) and a free list of four pruned symbols. The new entry b is preserved. For the sake of clarity, the actual symbol sequences corresponding to the various symbols of the search tree are indicated in Figure 8 and it will be seen that the symbol sequences bc, ca, aba, and abc have been deleted whereas the symbol sequence abb which was added in the updating procedure of Figure 6 has been preserved.

The pruning procedure is shown formally in Figure 9. In step i it is checked whether the dictionary is full and ready to prune, and if so the pointer of the memory is set to the dictionary entry following the beginning of the string, namely entry 4 corresponding to symbol b Table II and Figure 8(a). In step iii it is determined whether this entry has a down pointer extending therefrom and in the case of Figure 8(a) it will be seen that symbol b has the down pointer 7. Accordingly this entry is not an end node and the algorithm proceeds to advance the pointer to the next entry in step vi (which also clears any new flag) and hence via step vii to the next entry, namely symbol a of the sequence aba which is an end node. Accordingly the algorithm then checks whether this is protected against pruning by a new entry flag and since it is not it deletes the entry in step v, adds the corresponding memory location to the free list and re-sets the pointers. The re-setting of the pointers can easily be deduced from Figure 8 by analogy with Table II and Figure 6.

This procedure eventually results in the deletion of all "end nodes" (ie those having no down pointers D extending from them to other symbols) other than those protected by a new entry flag. This completes the updating of the dictionary and the algorithm can then proceed to step vi of Figure 4 in the case of the encoding algorithm and step ii of Figure 5 in the case of the decoding algorithm. It is emphasised that the updating and pruning of the dictionary takes place identically in the search tree of both the encoder and decoder. Whereas this pruning procedure tests all memory locations of the dictionary in a single procedure, pruning may be done in stages which can be determined in a number of ways, eg number of memory locations per stage, number of freed memory locations per stage.

As an additional procedure, which may improve the performance of the compression algorithm at the expense of increased memory and processing requirement, the codewords output from the encoder may be assigned lengths which depend in some manner on the frequency with which they are used. Two methods are described below for accomplishing this, which would apply to the element shown in Figure 4 (iii) which generates codewords representing the dictionary index to be transmitted.

The first method relies on the well known prior art described in "A Method for the Construction of Minimum Redundancy Codes" by D A Huffman, Proc IRE Vol 40,9, 1952 and in "A Mathematical Theory of Communication" by C E Shannon, Bell Systems Technical Journal, Vol 27, 1948, and in "An Introduction to Arithmetic Coding" by G Langdon, IBM Journal of Research and Development, Vol 28,2, 1984. A frequency count is associated with each dictionary entry, and is incremented each time the entry is used. This frequency is used to compute the probability of occurrence of the string represented by the dictionary entry, and the codeword assigned according to the procedures defined in the prior art. The codeword length L_s associated with a string of probability P_s is

$$-\log_2(P_s) < L_s < 1 - \log_2(P_s)$$

The second method would in general be less effective than the first method, but is simpler to implement. An additional U pointer is associated with each dictionary entry, and is used to form a linked (least frequently used) LFU list of dictionary entries, not related to the structure of the search tree. This list is used to determine in an approximate fashion, the frequency order of the strings. Also associated with each dictionary entry is a length index.

When a dictionary entry is used it is moved up the LFU list, by using the U pointer to locate the dictionary entry above the current entry, and exchanging the U pointers and length indices of the two entries. Thus more frequently used entries will move towards the start of the LFU list. The length index will be associated with the order of elements in the list, ie the most frequently used dictionary entry will have a length index of 1. A code may be generated on a fixed or dynamic basis, and codewords assigned on the basis of length index, shorter codewords being assigned to dictionary entries with a low length index. This second method would be suitable for application to the compression algorithm described by Miller and Wegman in EPA 127 815, as they provide a similar list; it would be necessary to add a length index to their data structure however.

Instead of using the U pointer to exchange the currently used dictionary entry with the entry immediately above it in the linked list, an alternative procedure would be to move the currently used dictionary entry to the top position in the linked list, the entry previously in the top position being moved to the second position and so on. The linked list is now approximately representative of a Least Recently Used function whereby entries that have not been used for some time will be pushed down to the bottom of the list, and frequently used entries will reside in the top portion of the list.

As an additional procedure which may improve the performance of the encoder at the expense of additional initialisation time, the dictionary may initially contain certain strings known to occur regularly, for example 'th', 'tha', 'the', 'Th', 'The',...

As an additional procedure which is possible if the memory in which the dictionary is stored is capable of retaining information between the transmission of subsequent messages, as is commonly the case when a mass storage device is present or when the memory is non-volatile, the dictionary may initially contain the set of strings learnt during encoding/decoding of the previous message. As an example, a communication device may call a second communications device, which it may have previously communicated with. The respective encoder and decoder pairs may compare their dictionaries by generating some simple checksum, according to principles well known, and comparing said checksums. If the dictionaries are not equivalent they may be reset to some known initial state.

As an additional procedure which will improve the reliability of the algorithm in the presence of errors on the link between encoder and decoder, although these would normally be infrequent due to the common use of error detection procedures such as Automatic Repeat Request, the decoder may regard received

codewords corresponding to empty or full dictionary entries as indication of error and request the encoder to reset or reinitialise its dictionary. Alternatively, a checksum may periodically be calculated by the encoder and communicated to the decoder; the decoder calculating an equivalent function and comparing it to the received checksum.

Figure 10 shows a data processing arrangement for converting uncompressed stored data which is stored in a mass storage device 1 (eg a disc store) into corresponding compressed data, thereby releasing memory from the mass storage device. An encoder 2 incorporating a microprocessor μP is provided with a memory 3 in which is stored and generated a search tree by means of an algorithm as described above. A delay circuit 4 ensures that the dictionary is updated before a new symbol is received from the mass storage device.

Figure 11 shows a decoder arrangement for converting compressed data in the mass storage device 1 to uncompressed data and comprises a decoder 5 provided with a memory 3 in which a search tree is maintained and updated in accordance with the algorithms described above.

Figure 12 shows a terminal 6 provided with a terminal interface 7 which is arranged to transmit and receive data along a communications link by means of an encoder 2 provided with a memory 3 and delay device 4 and a decoder 5 provided with a memory 3. A dictionary is maintained in memory 3 of the encoder and decoder by means of the algorithms described above. The arrangement shown in Figure 12 can communicate with a corresponding arrangement connected to the other end of the communication link (not shown).

In designing source encoders, it is usually assumed that the input to the encoder consists of discrete symbols from the source alphabet, and hence that the symbol size is known. Many modern communication systems employ synchronous transmission, in which data is treated as a continuous bit stream rather than individual characters.

When designing a source encoder for a synchronous source which outputs non-binary symbols in binary form, several approaches may be taken. If the symbol size is constant but unknown, the character length that results in optimum source code performance can be found by search. For example, a sample of data can be taken, the assumed character size stepped from one to some maximum number of bits, and the information content estimated for each case; the ratio of information content to character size should be minimal for the best compression ratio to be achieved. The probability of occurrence P_j of each symbol j is estimated and said number of bits per sequence n is varied so as to increase the information content

$$2^n \sum_{j=1} P_j \log_2(1/P_j)$$

of the symbols.

An alternative approach is possible if a string encoding is used. A symbol size is selected arbitrarily, the assumption being that frequent strings of source symbols will not lose their identity when parsed into bit strings of different lengths. There are various points to consider when selecting the segment size (where a segment is an assumed symbol).

For a segment of length C , as there is no synchronisation between the position in the source bit stream of segments and of symbols, any regular string of symbols may coincide with any of the C bit positions in the segment. This implies that, for any frequent string of symbols, at least C variations of the string are possible, and hence short segments are likely to result in more economical use of the dictionary space.

As the encoded binary string length is proportional to the segment size, but the codeword length is proportional to the logarithm of the number of known strings, short segments will make less efficient use of the dictionary space than long segments. This is particularly relevant to the Ziv-Lempel algorithm in which a large proportion of the memory space is devoted to pointers rather than symbol storage.

Accordingly, in a preferred method in accordance with the invention wherein the input data consists of a stream of binary digits (bits) representing characters, and the symbols stored in the search tree are each made up of sequences of binary digits (bits), the number of bits per sequence is selected by the processor (which may be in response to an external command signal from a user), the number of bits per symbol of the input data either being unknown or being different from the number of bits per sequence selected by the processor.

In a variant of this method the number of bits per sequence is initially varied and the resulting compression ratio between the input data and the output data is monitored before selecting the number of

bits per sequence for the stored symbols of the search tree.

Thus it is possible to optimise the number of bits per sequence selected by the processor in order to maximise the compression ratio. However, other factors such as memory capacity or processing speed may effect the optimum number of bits per sequence to be selected by the processor.

The above described embodiments employ a method of deletion which is highly efficient in terms of memory utilization and execution speed although it is slightly sub-optimal in terms of compression efficiency. The embodiments are formed from the two data structures used to build a dictionary, the systematic search tree structure used to represent strings and the systematic tabular representation of the dictionary used to store the elements of the search tree (as discussed by Sussenguth), the order in which elements of the search tree are stored being effectively random. The method approximates to the Random deletion strategy, in the sense that the selection of portions of the search tree to be deleted does not depend on the ordering of the selected portions within the tree, however it is not truly random as the algorithm for selecting candidates for deletion involves stepping through the tabular representation of the dictionary. This has some similarity with the CLOCK approximation, however it differs in that the selection of candidate elements for deletion made on the basis of position in the tabular representation is qualified by the position of the element within the search tree.

Claims

1. A method of compressing input data comprising reading successive symbols (a,b,c) of the data with a processor (μ P) provided with a memory (3) having indexed memory locations, generating from strings of symbols in the input data a dictionary in the form of a search tree of symbols (Figures 1 to 3, 6, 8) in the memory, which search tree has paths representative of said strings, matching symbol strings in the input data with previously stored paths in the search tree and generating from the stored paths compressed output data corresponding to the input data, stored symbols (S) in the search tree being linked to form said paths by linking pointers of two distinct types; a pointer (R) of the first type between stored symbols indicating that those stored symbols are alternative possible symbols at a given position in an input symbol sequence and a pointer (D) of the second type between stored symbols indicating that those stored symbols both occur, in order, in a possible input symbol sequence: characterised in that when the memory is full, sequential indexed memory locations of the search tree (Figures 1 to 3, 6, 8) are tested and deleted if they contain a node (c, a; Figure 8(a)) of the search tree which does not have a linking pointer of the second type (D) pointing to another node (Figure 9) and the resulting freed memory locations are made available for new dictionary entries.
2. A method as claimed in Claim 1 wherein all the indexed memory locations forming the dictionary are tested before the next symbol to be stored is stored in a freed memory location.
3. A method as claimed in Claim 1 or Claim 2 modified by the proviso that recently created nodes (b - (NEW ENTRY); Figure 8(a)) of the search tree are protected against deletion.
4. A method as claimed in any of Claims 1 to 3 wherein those indexed memory locations which do not contain nodes of the search tree are linked by pointers to form a free list (Figure 1(b), Figure 2(b)) and nodes of the search tree are deleted by re-setting pointers to bypass the deleted nodes and connecting them to the free list, whereby the search tree is maintained as a connected whole.
5. A method as claimed in any one of Claims 1 to 4 wherein each node of the search tree is associated with a respective counter which is incremented each time the associated node is used, and the compressed output data comprises codewords whose lengths are related to the contents of the counters such that the shortest codewords represent the most frequently used nodes.
6. A method as claimed in any one of Claims 1 to 4 wherein the nodes of the search tree are stored in order in the memory and the order in which they are stored is re-arranged by raising the ordinal value of a node after it has been used whereby rarely used nodes acquire a low ordinal value, and are deleted.
7. A method as claimed in Claim 6 wherein the ordinal value of the used node is increased by one, and the ordinal value of the node immediately above the used node is decreased by one, whereby these two nodes exchange ordinal values.



bits per sequence for the stored symbols of the search tree.

Thus it is possible to optimise the number of bits per sequence selected by the processor in order to maximise the compression ratio. However, other factors such as memory capacity or processing speed may effect the optimum number of bits per sequence to be selected by the processor.

- 5 The above described embodiments employ a method of deletion which is highly efficient in terms of memory utilization and execution speed although it is slightly sub-optimal in terms of compression efficiency. The embodiments are formed from the two data structures used to build a dictionary, the systematic search tree structure used to represent strings and the systematic tabular representation of the dictionary used to store the elements of the search tree (as discussed by Sussenguth), the order in which
10 elements of the search tree are stored being effectively random. The method approximates to the Random deletion strategy, in the sense that the selection of portions of the search tree to be deleted does not depend on the ordering of the selected portions within the tree, however it is not truly random as the algorithm for selecting candidates for deletion involves stepping through the tabular representation of the dictionary. This has some similarity with the CLOCK approximation, however it differs in that the selection of
15 candidate elements for deletion made on the basis of position in the tabular representation is qualified by the position of the element within the search tree.

Claims

- 20 1. A method of compressing input data comprising reading successive symbols (a,b,c) of the data with a processor (μ P) provided with a memory (3) having indexed memory locations, generating from strings of symbols in the input data a dictionary in the form of a search tree of symbols (Figures 1 to 3, 6, 8) in the memory, which search tree has paths representative of said strings, matching symbol strings in the input data with previously stored paths in the search tree and generating from the stored paths
25 compressed output data corresponding to the input data, stored symbols (S) in the search tree being linked to form said paths by linking pointers of two distinct types; a pointer (R) of the first type between stored symbols indicating that those stored symbols are alternative possible symbols at a given position in an input symbol sequence and a pointer (D) of the second type between stored symbols indicating that those stored symbols both occur, in order, in a possible input symbol sequence:
30 characterised in that when the memory is full, sequential indexed memory locations of the search tree (Figures 1 to 3, 6, 8) are tested and deleted if they contain a node (c, a; Figure 8(a)) of the search tree which does not have a linking pointer of the second type (D) pointing to another node (Figure 9) and the resulting freed memory locations are made available for new dictionary entries.
- 35 2. A method as claimed in Claim 1 wherein all the indexed memory locations forming the dictionary are tested before the next symbol to be stored is stored in a freed memory location.
3. A method as claimed in Claim 1 or Claim 2 modified by the proviso that recently created nodes (b - (NEW ENTRY); Figure 8(a)) of the search tree are protected against deletion.
4. A method as claimed in any of Claims 1 to 3 wherein those indexed memory locations which do not contain nodes of the search tree are linked by pointers to form a free list (Figure 1(b), Figure 2(b)) and nodes of the search tree are deleted by re-setting pointers to bypass the deleted nodes and connecting them to the free list, whereby the search tree is maintained as a connected whole.
45
5. A method as claimed in any one of Claims 1 to 4 wherein each node of the search tree is associated with a respective counter which is incremented each time the associated node is used, and the compressed output data comprises codewords whose lengths are related to the contents of the counters such that the shortest codewords represent the most frequently used nodes.
50
6. A method as claimed in any one of Claims 1 to 4 wherein the nodes of the search tree are stored in order in the memory and the order in which they are stored is re-arranged by raising the ordinal value of a node after it has been used whereby rarely used nodes acquire a low ordinal value, and are deleted.
55
7. A method as claimed in Claim 6 wherein the ordinal value of the used node is increased by one, and the ordinal value of the node immediately above the used node is decreased by one, whereby these two nodes exchange ordinal values.



8. A method as claimed in Claim 6 wherein the ordinal value of the used node is increased to the maximum value, and the ordinal values of all the nodes above the used node are decreased by one.
- 5 9. A method as claimed in any one of Claims 6 to 8 wherein each node of the search tree has an associated length index and the compressed output data comprises codewords whose lengths are related to the length indices such that the shortest codewords represent nodes with the highest ordinal values.
- 10 10. A method as claimed in any preceding claim wherein the input data includes spaces between sequences of symbols and the process of generating a new stored path of the search tree is terminated when such a space is detected.
- 15 11. A method as claimed in any preceding claim wherein the input data consists of a stream of binary digits (bits) representing characters, and the symbols (S) stored in the search tree are each made up of sequences of binary digits (bits), and wherein the number of bits per sequence is selected by the processor (μP), the number of bits per character of the input data either being unknown or being different from the number of bits per sequence selected by the processor.
- 20 12. A method as claimed in Claim 11 wherein the processor (μP) is arranged to perform the selection in response to an external command signal from a user.
- 25 13. A method as claimed in either Claim 11 or Claim 12 wherein the number of bits per sequence is initially varied and the resulting compression ratio between the input data and the output data is measured, and the number of bits per sequence for the stored symbols of the search tree is selected on the basis of the measured compression ratio.
- 30 14. A method as claimed in any preceding claim wherein each linking pointer of the second type (D) points to a first stored symbol of an ordered list (S2, S3, S4) of alternative stored symbols of the search tree and successive stored symbols in said ordered list are connected by linking pointers of the first type (R) which point to each succeeding stored symbol in said list.
- 35 15. A method as claimed in any of Claims 1 to 13 wherein each linking pointer of the second type (D) points to any one of a list of alternative stored symbols of the search tree, the stored symbols in said list being connected to each other by pointers of the first type (R) pointing in one direction and also by pointers of the first type pointing in the opposite direction (L) so as to enable any stored symbol in the list to be accessed.
- 40 16. A method as claimed in either Claim 14 or Claim 15 wherein linking pointers of the first type (R) are used to search through the list of alternative stored symbols to find a match for the most recently read input symbol and, if a match is found, to obtain the pointer of the second type (D), if present.
- 45 17. A method as claimed in any preceding claim wherein, before the input data is processed, the memory is initially provided with stored symbols (a,b,c) each of which corresponds to a sequence of symbols likely to occur in the input data, said initially provided stored symbols being stored as nodes in the search tree.
18. A method as claimed in any one of Claims 1 to 17 wherein the dictionary is reinitialised in response to a command signal received, in use, from an associated decoder.
- 50 19. A method as claimed in any of Claims 1 to 17 wherein, in use, a checksum for the dictionary is periodically calculated and a corresponding output signal generated.
- 55 20. A method as claimed in any of Claims 1 to 17 wherein the dictionary is retained for further use, and including performing a checksum calculation on the dictionary prior to any such further use, receiving a corresponding checksum from another such dictionary, comparing the checksums and reinitialising the dictionary if the checksums are not identical.



21. A method of storing data comprising compressing the data by a method as claimed in any of Claims 1 to 17, and storing the compressed data in a mass storage medium (1).

22. A method of decoding compressed data which has been compressed by a method as defined in any one of Claims 1 to 20 comprising reading successive characters of the compressed data with a processor (μ P) provided with a memory (3), storing in the memory a dictionary in the form of a search tree (Figures 1 to 3, 6, 8) of symbols which is built up from the compressed data, and utilizing the search tree to translate the compressed data to decoded data, stored symbols (S) in the search tree being linked by linking pointers of two distinct types; a pointer of the first type (R) between stored symbols indicating that those symbols are associated with different decoded strings of symbols having the same number of symbols and the same prefix and are the respective last symbols of such different strings and a pointer of the second type (D) between stored symbols indicating that those symbols are successive symbols in a string of decoded output symbols: characterized in that when the memory is full, sequential indexed memory locations of the search tree (Figures 1 to 3, 6, 8) are tested and deleted if they contain a node (c, a; Figure 8(a)) of the search tree which does not have a linking pointer of the second type (D) pointing to another node (Figure 9) and the resulting freed memory locations are made available for new dictionary entries.

23. A method as claimed in Claim 22 wherein all the indexed memory locations forming the dictionary are tested before the next symbol to be stored is stored in a freed memory location.

24. A method as claimed in Claim 22 or claim 23 modified by the proviso that recently created nodes (b - (NEW ENTRY); Figure 8(a)) of the search tree are protected against deletion.

25. A method as claimed in any of Claims 22 to 24 wherein those indexed memory locations which do not contain nodes of the search tree are linked by pointers to form a free list (Figure 1(b), Figure 2(b)) and nodes of the search tree are deleted by re-setting pointers to bypass the deleted nodes and connecting them to the free list, whereby the search tree is maintained as a connected whole.

26. A method as claimed in any one of Claims 22 to 25 wherein each node of the search tree is associated with a respective counter which is incremented each time the associated node is used, the shortest codewords are allocated to the most frequently used nodes, and the compressed data to be decoded comprises codewords of which the shortest codewords represent the most frequently used nodes of the search tree of a corresponding encoder.

27. A method as claimed in any one of Claims 22 to 25 wherein the nodes of the search tree are stored in order in the memory and the order in which they are stored is re-arranged by raising the ordinal value of a node after it has been used whereby rarely used nodes acquire a low ordinal value, and are deleted.

28. A method as claimed in Claim 27 wherein the ordinal value of the used node is increased by one, and the ordinal value of the node immediately above the used node is decreased by one, whereby these two nodes exchange ordinal values.

29. A method as claimed in Claim 27 wherein the ordinal value of the used node is increased to the maximum value, and the ordinal values of all the nodes above the used code are decreased by one.

30. A method as claimed in any of Claims 27 to 29 wherein each node of the search tree has an associated length index such that the shortest codewords represent nodes with the highest ordinal values and the compressed data to be decoded comprises codewords whose lengths are related to the length indices such that the shortest codewords represent highest ordinal value nodes of the search tree of a corresponding encoder.

31. A method as claimed in Claim 22 wherein each linking pointer of the second type (D) points to a first stored symbol of an ordered list (S2, S3, S4) of alternative stored symbols of the search tree and successive stored symbols in said ordered list are connected by linking pointers of the first type (R) which point to each succeeding stored symbol in said list.



32. A method as claimed in Claim 22 wherein each linking pointer of the second type (D) points to any one of a list of alternative stored symbols of the search tree, the stored symbols in said list being connected to each other by pointers of the first type (R) pointing in one direction and also by pointers of the first type pointing in the opposite direction (L) so as to enable any stored symbol in the list to be accessed.

33. A method as claimed in any one of Claims 22 to 32 including detecting when received compressed data corresponds to empty or free memory locations and generating a corresponding output signal.

34. A method as claimed in any one of Claims 22 to 32 wherein the dictionary is retained for further use, and including performing a checksum calculation on the dictionary prior to any such further use, receiving a corresponding checksum from another such dictionary, comparing the checksums and reinitializing the dictionary if the checksums are not identical.

35. A method of transmitting data comprising encoding the data by a method as claimed in any of Claims 1 to 20, transmitting the resulting compressed data to a remote location, and decoding the compressed data by a corresponding method as claimed in the appropriate one of Claims 22 to 32.

36. An encoder for compressing input data comprising a processor (μ P) capable of receiving successive symbols (a,b,c) of input data, a memory (3) having indexed memory locations, means for storing in the memory a search tree (Figures 1 to 3, 6, 8) of symbols which has paths representative of strings of symbols in the input data, and means for matching symbol strings in the input data with previously stored paths in the search tree and for generating from the stored paths compressed output data corresponding to the input data, stored symbols (s) in the search tree (Figures 1 to 3, 6, 8) being linked to form said paths by linking pointers of two distinct types; a pointer of the first type (R) between stored symbols indicating that those stored symbols are alternative possible symbols at a given position in an input symbol sequence and a pointer of the second type (D) between stored symbols indicating that those stored symbols both occur, in order, in a possible input symbol sequence: characterised in that, in use, the processor (μ P) is arranged to determine when the memory (3) is full, to test sequential indexed memory locations of the search tree, to delete a memory location if it contains a node (c, a; Figure 8(a)) of the search tree which does not have a linking pointer of the second type (D) pointing to another node and to make the resulting freed memory locations available for new dictionary entries.

37. An encoder as claimed in Claim 36 wherein the processor is further arranged to test all the indexed memory locations forming the dictionary before the next symbol to be stored is stored in a freed memory location.

38. An encoder as claimed in Claim 36 wherein the processor is arranged to protect recently created nodes (b (NEW ENTRY); Figure 8(a)) of the search tree against deletion.

39. An encoder as claimed in Claim 36 wherein the processor is arranged to link by pointers those indexed memory locations which do not contain nodes of the search tree to form a free list (Figure 1(b), Figure 2(b)) and to delete nodes of the search tree by re-setting pointers to bypass the deleted nodes and connecting them to a free list, whereby the search tree is maintained as a connected whole.

40. An encoder as claimed in Claim 36 wherein each node of the search tree is associated with a respective counter which is incremented each time the associated node is used, and the processor is arranged to provide compressed output data comprising codewords whose lengths are related to the contents of the counters such that the shortest codewords represent the most frequently used nodes.

41. An encoder as claimed in Claim 36 wherein the nodes of the search tree are stored in order in the memory and the processor is arranged to re-arrange the order in which they are stored by raising the ordinal value of a node after it has been used whereby rarely used nodes acquire a low ordinal value, and to delete nodes having the lowest ordinal values.

42. An encoder as claimed in Claim 41 wherein the ordinal value of the used node is increased by one, and the ordinal value of the node immediately above the used node is decreased by one, whereby these two nodes exchange ordinal values.



43. An encoder as claimed in Claim 41 wherein the ordinal value of the used node is increased to the maximum value, and the ordinal values of all the nodes above the used node are decreased by one.
- 5 44. An encoder as claimed in Claims 41 to 43 wherein each node of the search tree has an associated length index and the processor is arranged to provide compressed output data comprising codewords whose lengths are related to the length indices such that the shortest codewords represent nodes with the highest ordinal values.
- 10 45. An encoder as claimed in any one of Claims 36 to 44 for encoding input data including spaces between sequences of symbols and wherein the processor is arranged to terminate the process of generating a new stored path of the search tree when such a space is detected.
- 15 46. An encoder as claimed in any one of Claims 36 to 45 for encoding input data consisting of a stream of binary digits (bits) representing characters, wherein the symbols (S) stored in the search tree are each made up of sequences of binary digits (bits), and wherein the processor is arranged to select the number of bits per sequence such as to be different from the number of bits per character of the input data, if known.
- 25 47. An encoder as claimed in Claim 46 wherein the processor (μ P) is arranged to perform the selection in response to an external command signal from a user.
48. An encoder as claimed in either Claim 46 or Claim 47 wherein the number of bits per sequence selected by the processor is initially varied and the resulting compression ratio between the input data and the output data is measured, and the number of bits per sequence for the stored symbols of the search tree is selected on the basis of the measured compression ratio.
- 30 49. An encoder as claimed in any one of Claims 36 to 48 wherein each linking pointer of the second type (D) points to a first stored symbol of an ordered list (S2, S3, S4) of alternative stored symbols of the search tree and successive stored symbols in said ordered list are connected by linking pointers of the first type (R) which point to each succeeding stored symbol in said list.
- 35 50. An encoder as claimed in any one of Claims 36 to 48 wherein each linking pointer of the second type (D) points to any one of a list of alternative stored symbols of the search tree, the stored symbols in said list being connected to each other by pointers of the first type (R) pointing in one direction and also by pointers of the first type pointing in the opposite direction (L) so as to enable any stored symbol in the list to be accessed.
- 40 51. An encoder as claimed in any one of Claims 36 to 50 wherein, in use, the processor is responsive to a command signal received from an associated decoder to reinitialize its dictionary.
- 45 52. An encoder as claimed in any of Claims 36 to 50 wherein, in use, the processor periodically calculates a checksum for the dictionary and generates a corresponding output signal.
53. An encoder as claimed in any one of Claims 36 to 50 wherein the dictionary is retained for further use and the processor is arranged to perform a checksum calculation on the dictionary prior to any such further use and to generate a corresponding output signal, to compare this checksum with a corresponding checksum received, in use, from an associated decoder, and to reinitialise the dictionary to an initial state if the checksums are not identical.
- 50 54. A decoder for decoding compressed data comprising a processor (μ P) capable of receiving successive symbols of compressed data, a memory (3) having indexed memory locations and means for storing in the memory a dictionary in the form of a search tree (Figures 1 to 3, 6, 8) of symbols, the processor being arranged to build up the search tree from the compressed data and to utilize the search tree to translate the compressed data to decoded data, stored symbols (S) in the search tree being linked by linking pointers of two distinct types; a pointer of the first type (R) between stored symbols indicating that those symbols are associated with different decoded strings of symbols having the same number of symbols and are the respective last symbols of such different strings and a pointer of the second type (D) between stored symbols indicating that those symbols are successive symbols
- 55



in a string of decoded output symbols: characterized in that, in use, the processor (μP) is arranged to determine when the memory (3) is full, to test sequential indexed memory locations of the search tree, to delete a memory location if it contains a node (c, a; Figure 8(a)) of the search tree which does not have a linking pointer of the second type (D) pointing to another node and to make the resulting freed memory locations available for new dictionary entries.

55. A decoder as claimed in Claim 54 wherein in use, each linking pointer of the second type (D) points to a first stored symbol of an ordered list (S2, S3, S4) of alternative stored symbols of the search tree and successive stored symbols in said ordered list are connected by linking pointers of the first type (R) which point to each succeeding stored symbol in said list.

56. A decoder as claimed in Claim 54 wherein in use, each linking pointer of the second type (D) points to any one of a list of alternative stored symbols of the search tree, the stored symbols in said list being connected to each other by pointers of the first type (R) pointing in one direction and also by pointers of the first type pointing in the opposite direction (L) so as to enable any stored symbol in the list to be accessed.

57. A decoder as claimed in any one of Claims 54 to 56 wherein the processor is further arranged to test all the indexed memory locations forming the dictionary before the next symbol to be stored is stored in a freed memory location.

58. A decoder as claimed in any one of Claims 54 to 56 wherein the processor is arranged to protect recently created nodes (b (NEW ENTRY); Figure 8(a)) of the search tree against deletion.

59. A decoder as claimed in any one of Claims 54 to 58 wherein the processor is arranged to link by pointers those indexed memory locations which do not contain nodes of the search tree link by pointers to form a free list (Figure 1(b), Figure 2(b)) and to delete nodes of the search tree by re-setting pointers to bypass the deleted nodes and connecting them to the free list, whereby the search tree is maintained as a connected whole.

60. A decoder as claimed in any one of Claims 54 to 59 wherein each node of the search tree is associated with a respective counter which is incremented each time the associated node is used, and the processor is arranged to provide compressed output data comprising codewords whose lengths are related to the contents of the counters such that the shortest codewords represent the most frequently used nodes.

61. A decoder as claimed in any one of Claims 54 to 59 wherein the nodes of the search tree are stored in order in the memory and the processor is arranged to re-arrange the order in which they are stored by raising the ordinal value of a node after it has been used whereby rarely used nodes acquire a low ordinal value, and to delete nodes having the lowest ordinal values.

62. A decoder as claimed in Claim 61 wherein the ordinal value of the used node is increased by one, and the ordinal value of the node immediately above the used node is decreased by one, whereby these two nodes exchange ordinal values.

63. A decoder as claimed in Claim 61 wherein the ordinal value of the used node is increased to the maximum value, and the ordinal values of all the nodes above the used node are decreased by one.

64. A decoder as claimed in any one of Claims 61 to 63 wherein each node of the search tree has an associated length index and the processor is arranged to provide compressed output data comprising codewords whose lengths are related to the length indices such that the shortest codewords represent nodes with the highest ordinal values.

65. A decoder as claimed in any one of Claims 54 to 64 wherein, in use, the processor is arranged to detect when received compressed data corresponds to empty or free memory locations and to generate a corresponding output signal for transmission to an associated encoder.



66. A decoder as claimed in any one of Claims 54 to 65 wherein the dictionary is retained for further use and the processor is arranged to perform a checksum calculation on the dictionary prior to any such further use and to generate a corresponding output signal for transmission to an associated encoder, to compare this checksum with a corresponding checksum received, in use, from said encoder, and to reinitialize the dictionary if the checksums are not identical.

67. Data processing apparatus comprising an encoder (2) as claimed in any of Claims 36 to 53, a corresponding decoder (5) as claimed in any of Claims 54 to 66 and a data link for transmitting the compressed data between the encoder and decoder.

68. Data processing apparatus comprising an encoder (2) as claimed in any of Claims 36 to 50 a corresponding decoder (5) as claimed in any of Claims 54 to 57 and a mass storage medium (1) accessible by the encoder and decoder.

Patentansprüche

1. Verfahren zum Komprimieren von Eingabedaten, das aufweist:
 Lesen aufeinanderfolgender Symbole (a, b, c) der Daten mit einem Prozessor (μ P), der mit einem Speicher (3) versehen ist, der indexierte Speicherstellen aufweist,
 Erzeugen eines Verzeichnisses in der Form eines Symbol- Suchbaumes (Fig. 1 bis 3, 6, 8) in dem Speicher, und zwar von Symbol-Zeichenketten in den Eingabedaten, wobei der Suchbaum Pfade aufweist, die für die Zeichenketten repräsentativ sind,
 Abgleichen von Symbol-Zeichenketten in den Eingabedaten mit vorher gespeicherten Pfaden in dem Suchbaum und
 Erzeugen, und zwar aus den gespeicherten Pfaden, von komprimierten Ausgabedaten entsprechend den Eingabedaten, wobei gespeicherte Symbole (S) in dem Suchbaum verknüpft sind, um die Pfade durch verknüpfende Zeiger zweier unterschiedlicher Typen zu bilden;
 wobei ein Zeiger (R) des ersten Typs zwischen gespeicherten Symbolen anzeigt, daß diese gespeicherten Symbole alternativ mögliche Symbole bei einer gegebenen Position in einer Eingabesymbolsequenz sind, und wobei ein Zeiger (D) des zweiten Typs zwischen gespeicherten Symbolen anzeigt, daß jene gespeicherten Symbole in einer Reihenfolge in einer möglichen Eingabesymbolsequenz beide auftreten,
dadurch gekennzeichnet, daß
 wenn der Speicher voll ist, sequentiell indexierte Speicherstellen des Suchbaumes (Fig. 1 bis 3, 6, 8) geprüft und gelöscht werden, falls sie einen Knoten (c, a; Fig. 8 (a)) des Suchbaumes enthalten, der keinen verknüpfenden Zeiger des zweiten Typs (D) aufweist, der zu einem anderen Knoten (Fig. 9) zeigt, und die sich ergebenden frei gemachten Speicherstellen für neue Verzeichniseinträge verfügbar gemacht werden.
2. Verfahren gemäß Anspruch 1, worin alle indexierten Speicherstellen, die das Verzeichnis bilden, geprüft werden, bevor das nächste zu speichernde Symbol in einer frei gemachten Speicherstelle gespeichert wird.
3. Verfahren gemäß Anspruch 1 oder Anspruch 2, das unter der Bedingung abgeändert ist, daß neulich erzeugte Knoten (b (NEUER EINTRAG); Fig. 8 (a)) des Suchbaumes gegen Löschen geschützt sind.
4. Verfahren gemäß einem der Ansprüche 1 bis 3, wobei jene indexierten Speicherstellen, die keine Knoten des Suchbaumes enthalten, durch Zeiger verknüpft sind, um eine freie Liste (Fig. 1 (b), Fig. 2-(b)) zu bilden, und wobei Knoten des Suchbaumes durch Rücksetzen von Zeigern gelöscht werden, um die gelöschten Knoten zu umgehen, und indem sie mit der freien Liste verbunden werden, wodurch der Suchbaum als ein verbundenes Ganzes aufrechterhalten wird.
5. Verfahren gemäß einem der Ansprüche 1 bis 4, wobei jeder Knoten des Suchbaumes zu einem jeweiligen Zähler gehört, der jedes Mal inkrementiert wird, wenn der zugehörige Knoten verwendet wird, und wobei die komprimierten Ausgabedaten Codewörter aufweisen, deren Längen mit den Inhalten der Zähler in Beziehung stehen, so daß die kürzesten Codewörter die am häufigsten verwendeten Knoten darstellen.



6. Verfahren gemäß einem der Ansprüche 1 bis 4, wobei die Knoten des Suchbaumes in einer Reihenfolge in dem Speicher gespeichert werden und die Reihenfolge, in der sie gespeichert werden, neu angeordnet wird durch Erhöhen der Ordnungszahl eines Knotens, nachdem er verwendet worden ist, wodurch selten verwendete Knoten eine niedrige Ordnungszahl annehmen und gelöscht werden.

7. Verfahren gemäß Anspruch 6, wobei die Ordnungszahl des verwendeten Knotens um eins erhöht wird und worin die Ordnungszahl des unmittelbar über dem verwendeten Knoten liegenden Knotens um eins erniedrigt wird, wodurch diese zwei Knoten Ordnungszahlen austauschen.

8. Verfahren gemäß Anspruch 6, wobei die Ordnungszahl des verwendeten Knotens auf den Maximalwert erhöht wird, und wobei die Ordnungszahlen aller Knoten über dem verwendeten Knoten um eins erniedrigt werden.

9. Verfahren gemäß einem der Ansprüche 6 bis 8, wobei jeder Knoten des Suchbaumes einen zugehörigen Längenindex aufweist und wobei die komprimierten Ausgabedaten Codewörter aufweisen, deren Längen mit den Längenindizes in Beziehung stehen, so daß die kürzesten Codewörter Knoten mit den höchsten Ordnungszahlen darstellen.

10. Verfahren gemäß einem der vorhergehenden Ansprüche, wobei die Eingabedaten Leerzeichen zwischen Symbolsequenzen aufweisen und wobei der Prozeß des Erzeugens eines neuen gespeicherten Pfades des Suchbaumes beendet wird, wenn solch ein Leerzeichen erfaßt wird.

11. Verfahren gemäß einem der vorhergehenden Ansprüche, wobei die Eingabedaten aus einem Strom binärer Ziffern (Bits) bestehen, die Zeichen darstellen, und wobei jedes der in dem Suchbaum gespeicherten Symbole (S) aus Sequenzen von binären Ziffern (Bits) aufgebaut ist, und wobei die Anzahl von Bits pro Sequenz durch den Prozessor (μ P) ausgewählt wird, wobei die Anzahl von Bits pro Zeichen der Eingabedaten entweder unbekannt ist oder von der von dem Prozessor ausgewählten Anzahl von Bits pro Sequenz unterschiedlich ist.

12. Verfahren gemäß Anspruch 11, wobei der Prozessor (μ P) angeordnet ist, um die Auswahl in Antwort auf ein externes Befehlssignal von einem Anwender auszuführen.

13. Verfahren gemäß einem der Ansprüche 11 oder 12, wobei die Anzahl von Bits pro Sequenz anfänglich variiert wird und das sich ergebende Komprimierungsverhältnis zwischen den Eingabedaten und den Ausgabedaten gemessen wird, und wobei die Anzahl von Bits pro Sequenz für die gespeicherten Symbole des Suchbaumes auf der Grundlage des gemessenen Komprimierungsverhältnisses ausgewählt wird.

14. Verfahren gemäß einem der vorhergehenden Ansprüche, wobei jeder verknüpfende Zeiger des zweiten Typs (D) auf ein erstes gespeichertes Symbol einer geordneten Liste (S2, S3, S4) alternativer gespeicherter Symbole des Suchbaumes zeigt und nacheinander gespeicherte Symbole in der geordneten Liste durch verknüpfende Zeiger des ersten Typs (R) verbunden werden, die auf jedes nachfolgende gespeicherte Symbol in der Liste zeigen.

15. Verfahren gemäß einem der Ansprüche 1 bis 13, wobei jeder verknüpfende Zeiger des zweiten Typs (D) auf irgendeinen einer Liste alternativer gespeicherter Symbole des Suchbaumes zeigt, wobei die gespeicherten Symbole in der Liste miteinander durch Zeiger des ersten Typs (R) verbunden sind, die in eine Richtung zeigen, und ebenso durch Zeiger des ersten Typs, die in die entgegengesetzte Richtung (L) zeigen, damit auf irgendein gespeichertes Symbol in der Liste zugegriffen werden kann.

16. Verfahren gemäß einem der Ansprüche 14 oder 15, wobei die verknüpfenden Zeiger des ersten Typs (R) verwendet werden, um durch die Liste alternativer gespeicherter Symbole zu suchen, um ein Gegenstück für das zuletzt gelesene Eingabesymbol zu finden, und wenn ein Gegenstück gefunden ist, den Zeiger des zweiten Typs (D) zu erhalten, falls vorhanden.

17. Verfahren gemäß einem der vorhergehenden Ansprüche, wobei, bevor die Eingabedaten verarbeitet werden, der Speicher anfänglich mit gespeicherten Symbolen (a, b, c) versehen wird, von denen jedes einer Symbolsequenz entspricht, die wahrscheinlich in den Eingabedaten auftreten, wobei die anfäng-



lich bereitgestellten gespeicherten Symbole als Knoten in dem Suchbaum gespeichert werden.

18. Verfahren gemäß einem der Ansprüche 1 bis 17, wobei das Verzeichnis neu initialisiert wird in Antwort auf ein Befehlssignal, das, bei Verwendung, von einem zugehörigen Decodierer empfangen wird.

19. Verfahren gemäß einem der Ansprüche 1 bis 17, wobei, bei Verwendung, eine Prüfsumme für das Verzeichnis periodisch berechnet wird und ein entsprechendes Ausgabesignal erzeugt wird.

20. Verfahren gemäß einem der Ansprüche 1 bis 17, wobei das Verzeichnis zur weiteren Verwendung bewahrt wird und das Verfahren ein Durchführen einer Prüfsummenberechnung an dem Verzeichnis vor irgendeiner solchen weiteren Verwendung, ein Empfangen einer entsprechenden Prüfsumme von einem anderen solchen Verzeichnis, ein Vergleichen der Prüfsummen und ein Neu-Initialisieren des Verzeichnisses aufweist, wenn die Prüfsummen nicht identisch sind.

21. Verfahren zum Speichern von Daten, das ein Komprimieren der Daten nach einem Verfahren gemäß einem der Ansprüche 1 bis 17 und ein Speichern der komprimierten Daten in einem Massenspeichermedium (1) aufweist.

22. Verfahren zum Decodieren komprimierter Daten, die durch ein Verfahren komprimiert worden sind, wie festgelegt in einem der Ansprüche 1 bis 20, wobei das Verfahren aufweist:
Lesen aufeinanderfolgender Zeichen der komprimierten Daten mit einem Prozessor (μ P), der mit einem Speicher (3) versehen ist,
Speichern eines Verzeichnisses in der Form eines Suchbaumes (Fig. 1 bis 3, 6, 8) von Symbolen in dem Speicher, der aus den komprimierten Daten aufgebaut ist, und
Verwenden des Suchbaumes, um die komprimierten Daten in decodierte Daten zu übersetzen, wobei gespeicherte Symbole (S) in dem Suchbaum durch verknüpfende Zeiger zweier unterschiedlicher Typen verknüpft sind;
wobei ein Zeiger des ersten Typs (R) zwischen gespeicherten Symbolen anzeigt, daß diese Symbole zu unterschiedlichen decodierten Symbol-Zeichenketten gehören, die die gleiche Anzahl von Symbolen und den gleichen Präfix haben und die jeweiligen letzten Symbole solcher unterschiedlicher Zeichenketten sind, und wobei ein Zeiger des zweiten Typs (D) zwischen gespeicherten Symbolen anzeigt, daß diese Symbole aufeinanderfolgende Symbole in einer Zeichenkette decodierter Ausgabesymbole ist, dadurch gekennzeichnet, daß,
wenn der Speicher voll ist, sequentiell indexierte Speicherstellen des Suchbaumes (Fig. 1 bis 3, 6, 8) geprüft und gelöscht werden, wenn sie einen Knoten enthalten (c, a; Fig. 8 (a)) des Suchbaumes, der keinen verbindenden Zeiger des zweiten Typs (D) aufweist, der zu einem anderen Knoten (Fig. 9) zeigt, und daß die resultierenden freigemachten Speicherstellen für neue Verzeichniseinträge verfügbar gemacht werden.

23. Verfahren nach Anspruch 22, wobei alle die indexierten Speicherstellen, die das Verzeichnis bilden, geprüft werden, bevor das nächste zu speichernde Symbol in einer freigemachten Speicherstelle gespeichert wird.

24. Verfahren nach Anspruch 22 oder Anspruch 23, abgeändert unter der Voraussetzung, daß neulich erzeugte Knoten (b (NEUER EINTRAG); Fig. 8 (a)) des Suchbaumes gegen Löschen geschützt werden.

25. Verfahren nach einem der Ansprüche 22 bis 24, wobei jene indexierte Speicherstellen, die keine Knoten des Suchbaumes enthalten, durch Zeiger verknüpft werden, um eine freie Liste zu bilden (Fig. 1 (b), Fig. 2 (b)), und Knoten des Suchbaumes durch Rücksetzen von Zeigern gelöscht werden, um die gelöschten Knoten zu umgehen, und in dem sie mit der freien Liste verbunden werden, wodurch der Suchbaum als verbundenes Ganzes aufrechterhalten wird.

26. Verfahren nach einem der Ansprüche 22 bis 25, wobei jeder Knoten des Suchbaumes zu einem jeweiligen Zähler gehört, der jedesmal inkrementiert wird, wenn der Knoten verwendet wird, wobei die kürzesten Codewörter den am häufigsten verwendeten Knoten zugewiesen werden, und wobei die zu decodierenden komprimierten Daten Codewörter aufweisen, von denen die kürzesten Codewörter die am häufigsten verwendeten Knoten des Suchbaumes eines entsprechenden Codierers darstellen.



27. Verfahren nach einem der Ansprüche 22 bis 25, wobei die Knoten des Suchbaumes in einer Reihenfolge in dem Speicher gespeichert werden und die Reihenfolge, in der sie gespeichert werden, durch Erhöhen der Ordnungszahl eines Knotens neu angeordnet wird, nachdem er verwendet worden ist, wodurch selten verwendete Knoten eine niedrige Ordnungszahl annehmen und gelöscht werden.

28. Verfahren gemäß Anspruch 27, wobei die Ordnungszahl des verwendeten Knotens auf den Maximalwert erhöht wird, und die Ordnungszahlen aller der Knoten oberhalb des verwendeten Knotens um eins erniedrigt werden.

29. Verfahren gemäß Anspruch 27, wobei die Ordnungszahl des verwendeten Knotens auf den Maximalwert erhöht wird, und die Ordnungszahlen aller Knoten oberhalb des verwendeten Codes um eins erniedrigt werden.

30. Verfahren gemäß einem der Ansprüche 27 bis 29, wobei jeder Knoten des Suchbaumes einen zugehörigen Längenindex aufweist, so daß die kürzesten Codewörter Knoten mit den höchsten Ordnungszahlen darstellen, und wobei die zu decodierenden komprimierten Daten Codewörter aufweisen, deren Längen mit den Längenindizes in Beziehung stehen, so daß die kürzesten Codewörter Knoten des Suchbaums eines entsprechenden Codierers höchster Ordnungszahl darstellen.

31. Verfahren gemäß Anspruch 22, wobei jeder verknüpfende Zeiger des zweiten Typs (D) zu einem ersten gespeicherten Symbol einer geordneten Liste (S2, S3, S4) alternativer gespeicherter Symbole des Suchbaumes zeigt, und wobei aufeinanderfolgende gespeicherte Symbole in der geordneten Liste durch verknüpfende Zeiger des ersten Typs (R) verbunden sind, die auf jedes nachfolgende gespeicherte Symbol in der Liste zeigen.

32. Verfahren gemäß Anspruch 22, wobei jeder verknüpfende Zeiger des zweiten Typs (D) zu irgendeiner Liste alternativ gespeicherte Symbole des Suchbaumes zeigt, wobei die gespeicherten Symbole in der Liste miteinander durch Zeiger des ersten Typs (R) verbunden sind, die in eine Richtung zeigen, und auch durch Zeiger des ersten Typs, die in die entgegengesetzte Richtung (L) zeigen, so daß auf jedwedes gespeicherte Symbol in der Liste zugegriffen werden kann.

33. Verfahren gemäß einem der Ansprüche 22 bis 32, einschließlich Erfassen, und zwar wenn empfangene komprimierte Daten leeren oder freien Speicherstellen entsprechen, und Erzeugen eines entsprechenden Ausgabesignals.

34. Verfahren gemäß einem der Ansprüche 22 bis 32, wobei das Verzeichnis zur weiteren Verwendung aufbewahrt wird, und einschließlich Ausführen einer Prüfsummenberechnung an dem Verzeichnis vor irgendeiner solchen weiteren Verwendung, Empfangen einer entsprechenden Prüfsumme von einem anderen solchen Verzeichnis, Vergleichen der Prüfsummen und Neu-Initialisieren des Verzeichnisses, wenn die Prüfsummen nicht identisch sind.

35. Verfahren zum Übertragen von Daten, das aufweist:
Codieren der Daten nach einem Verfahren gemäß einem der Ansprüche 1 bis 20, Übertragen der resultierenden komprimierten Daten an einen entfernten Ort und Decodieren der komprimierten Daten nach einem entsprechenden Verfahren gemäß dem jeweils geeigneten Anspruch der Ansprüche 22 bis 32.

36. Codierer zum Komprimieren von Eingabedaten, der aufweist:
einen Prozessor (μP), der in Lage ist, aufeinanderfolgende Symbole (a, b, c) von Eingabedaten zu empfangen,
einen Speicher (3), der indexierte Speicherstellen hat,
eine Einrichtung zum Speichern eines Suchbaumes (Fig. 1 bis 3, 6, 8) von Symbolen in dem Speicher, welcher Pfade aufweist, die für die Symbol-Zeichenketten in den Eingabedaten stellvertretend sind, und
eine Einrichtung zum Abgleichen von Symbol-Zeichenketten in den Eingabedaten mit vorher gespeicherten Pfaden in dem Suchbaum und zum Erzeugen, und zwar aus den gespeicherten Pfaden, von komprimierten Ausgabedaten entsprechend den Eingabedaten, wobei gespeicherte Symbole (S) in dem Suchbaum (Fig. 1 bis 3, 6, 8) verknüpft sind, um die Pfade durch verknüpfende Zeiger zweier unterschiedlicher Typen zu bilden;



wobei ein Zeiger des ersten Typs (R) zwischen gespeicherten Symbolen anzeigt, daß diese gespeicherten Symbole alternativ mögliche Symbole bei einer gegebenen Position in einer Eingabesymbolsequenz sind, und wobei ein Zeiger des zweiten Typs (D) zwischen gespeicherten Symbolen anzeigt, daß diese gespeicherten Symbole in einer Reihenfolge in einer möglichen Eingabesymbolsequenz beide auftreten,

dadurch gekennzeichnet, daß

bei Verwendung, der Prozessor (μP) angeordnet ist, um zu bestimmen, wenn der Speicher (3) voll ist, um sequentielle indexierte Speicherstellen des Suchbaumes zu prüfen, um eine Speicherstelle zu löschen, falls sie einen Knoten (c, a; Fig. 8 (a)) des Suchbaumes enthält, der keinen verknüpfenden Zeiger des zweiten Typs (D) aufweist, der zu einem anderen Knoten zeigt, und um die sich ergebenden freigemachten Speicherstellen für neue Verzeichniseinträge verfügbar zu machen.

37. Codierer gemäß Anspruch 36, wobei der Prozessor weiterhin angeordnet ist, um alle die indexierten Speicherstellen, die das Verzeichnis bilden, zu prüfen, bevor das nächste zu speichernde Symbol in einer freigemachten Speicherstelle gespeichert wird.

38. Codierer gemäß Anspruch 36, wobei der Prozessor angeordnet ist, um kürzlich erzeugte Knoten (b - (NEUER EINTRAG); Fig. 8 (a)) des Suchbaumes gegen Löschen zu schützen.

39. Codierer gemäß Anspruch 36, wobei der Prozessor angeordnet ist, um mittels Zeigern jene indexierten Speicherstellen zu verknüpfen, die keine Knoten des Suchbaumes bilden, um eine freie Liste (Fig. 1 (b), Fig. 2 (b)) zu bilden, und um Knoten des Suchbaumes durch Rücksetzen von Zeigern zu löschen, um die gelöschten Knoten zu umgehen, und indem sie mit einer freien Liste verbunden werden, wodurch der Suchbaum als ein verbundenes Ganzes aufrechterhalten wird.

40. Codierer gemäß Anspruch 36, wobei jeder Knoten des Suchbaumes zu einem jeweiligen Zähler gehört, der jedesmal inkrementiert wird, wenn der zugehörige Knoten verwendet wird, und wobei der Prozessor angeordnet ist, um komprimierte Ausgabedaten bereitzustellen, die Codewörter aufweisen, deren Längen mit den Inhalten der Zähler in Beziehung stehen, so daß die kürzesten Codewörter die am häufigsten verwendeten Knoten darstellen.

41. Codierer gemäß Anspruch 36, wobei die Knoten des Suchbaumes in einer Reihenfolge in dem Speicher gespeichert werden und wobei der Prozessor angeordnet ist, um die Reihenfolge, in der sie gespeichert sind, neu anzuordnen durch Erhöhen der Ordnungszahl eines Knotens, nachdem er verwendet worden ist, wodurch selten verwendete Knoten eine niedrige Ordnungszahl annehmen, und um Knoten mit den niedrigsten Ordnungszahlen zu löschen.

42. Codierer gemäß Anspruch 41, wobei die Ordnungszahl des verwendeten Knotens um eins erhöht wird, und wobei die Ordnungszahl des Knotens unmittelbar oberhalb des verwendeten Knotens um eins erniedrigt wird, wodurch diese zwei Knoten Ordnungszahlen austauschen.

43. Codierer gemäß Anspruch 41, wobei die Ordnungszahl des verwendeten Knotens auf den Maximalwert erhöht wird und wobei die Ordnungszahlen, all der Knoten oberhalb des verwendeten Knotens um eins erniedrigt werden.

44. Codierer gemäß Anspruch 41 bis 43, wobei jeder Knoten des Suchbaumes einen zugehörigen Längenindex aufweist, und wobei der Prozessor angeordnet ist, um komprimierte Ausgabedaten bereitzustellen, die Codewörter aufweisen, deren Längen mit den Längenindizes in Beziehung stehen, so daß die kürzesten Codewörter Knoten mit den höchsten Ordnungszahlen darstellen.

45. Codierer gemäß einem der Ansprüche 36 bis 44 zum Codieren von Eingabedaten einschließlich Leerzeichen zwischen Symbolsequenzen und wobei der Prozessor angeordnet ist, um den Vorgang des Erzeugens eines neuen gespeicherten Pfades des Suchbaumes zu beenden, wenn solch ein Leerzeichen erfaßt ist.

46. Codierer gemäß einem der Ansprüche 36 bis 45 zum Codieren von Eingabedaten, die aus einem Strom binärer Ziffern (Bits) bestehen, die Zeichen darstellen, wobei die Symbole (S), die in dem Suchbaum gespeichert sind, jedes aus einer Sequenz binärer Ziffern (Bits) aufgebaut ist, und wobei der



Prozessor angeordnet ist, um die Anzahl von Bits pro Sequenz auszuwählen, um sich von der Anzahl von Bits pro Zeichen der Eingabedaten zu unterscheiden, falls bekannt.

47. Codierer gemäß Anspruch 46, wobei der Prozessor (μP) angeordnet ist, um die Auswahl in Antwort auf ein externes Befehlssignal von einem Anwender auszuführen.

48. Codierer gemäß Anspruch 46 oder Anspruch 47, wobei die Anzahl von Bits pro Sequenz, die durch den Prozessor ausgewählt wird, anfänglich variiert wird und das resultierende Komprimierungsverhältnis zwischen den Eingabedaten und den Ausgabedaten gemessen wird, und wobei die Anzahl von Bits pro Sequenz für die gespeicherten Symbole des Suchbaumes auf der Grundlage des gemessenen Komprimierungsverhältnisses ausgewählt wird.

49. Codierer gemäß einem der Ansprüche 36 bis 48, wobei jeder verbindende Zeiger des zweiten Typs (D) zu einem ersten gespeicherten Symbol einer geordneten Liste (S2, S3, S4) alternativer gespeicherter Symbole des Suchbaumes zeigt, und wobei aufeinanderfolgende gespeicherte Symbole in der geordneten Liste durch verbindende Zeiger des zweiten Typs (R) verbunden werden, die zu jedem nachfolgenden gespeicherten Symbol in der Liste zeigen.

50. Codierer gemäß einem der Ansprüche 36 bis 48, wobei jeder verbindende Zeiger des zweiten Typs (D) zu irgendeinem aus einer Liste alternativer gespeicherter Symbole des Suchbaumes zeigt, wobei die gespeicherten Symbole in der Liste miteinander durch Zeiger des ersten Typs (R) verbunden sind, die in eine Richtung zeigen, und ebenso durch Zeiger des ersten Typs, die in die entgegengesetzte Richtung (L) zeigen, so daß auf jedwedes gespeicherte Symbol in der Liste zugegriffen werden kann.

51. Codierer gemäß einem der Ansprüche 36 bis 50, wobei, bei Verwendung, der Prozessor auf ein Befehlssignal anspricht, das von einem zugehörigen Decodierer empfangen wird, um sein Verzeichnis neu zu initialisieren.

52. Codierer gemäß einem der Ansprüche 36 bis 50, wobei, bei Verwendung, der Prozessor periodisch eine Prüfsumme für das Verzeichnis berechnet und ein entsprechendes Ausgabesignal erzeugt.

53. Codierer gemäß einem der Ansprüche 36 bis 50, wobei das Verzeichnis zur weiteren Verwendung aufbewahrt wird, und wobei der Prozessor angeordnet ist, um eine Prüfsummenberechnung an dem Verzeichnis vor irgendeiner solchen weiteren Verwendung durchzuführen und um ein entsprechendes Ausgabesignal zu erzeugen, um diese Prüfsumme mit einer entsprechenden Prüfsumme zu vergleichen, die, bei Verwendung, von einem zugehörigen Decodierer empfangen wird, und um das Verzeichnis auf einen anfänglichen Zustand neu zu initialisieren, wenn die Prüfsummen nicht identisch sind.

54. Decodierer zum Decodieren komprimierter Daten, der aufweist:
einen Prozessor (μP), der in der Lage ist, aufeinanderfolgende Symbole komprimierter Daten zu empfangen,
einen Speicher (3), der indexierte Speicherstellen aufweist, und
eine Einrichtung zum Speichern eines Verzeichnisses in der Form eines Suchbaumes (Fig. 1 bis 3, 6, 8) von Symbolen in dem Speicher, wobei der Prozessor angeordnet ist, um den Suchbaum aus den komprimierten Daten aufzubauen und den Suchbaum zu verwenden, um die komprimierten Daten in decodierte Daten zu übersetzen, wobei gespeicherte Symbole (S) in dem Suchbaum durch verknüpfende Zeiger zweier unterschiedlicher Typen verknüpft sind;
wobei ein Zeiger des ersten Typs (R) zwischen gespeicherten Symbolen anzeigt, daß diese Symbole zu unterschiedlichen decodierten Symbol-Zeichenketten gehören, die die gleiche Anzahl von Symbolen aufweisen und die jeweiligen letzten Symbole solcher unterschiedlicher Zeichenketten sind, und wobei ein Zeiger des zweiten Typs (D) zwischen gespeicherten Symbolen anzeigt, daß diese Symbole aufeinanderfolgende Symbole in einer Zeichenkette decodierter Ausgabesymbole sind, dadurch gekennzeichnet, daß
bei Verwendung, der Prozessor (μP) angeordnet ist, um zu bestimmen, wenn der Speicher (3) voll ist, um sequentielle indexierte Speicherstellen des Suchbaumes zu prüfen, um eine Speicherstelle zu löschen, falls sie einen Knoten (c, a; Fig. 8 (a)) des Suchbaumes enthält, der keinen verknüpfenden Zeiger des zweiten Typs (D) aufweist, der zu einem anderen Knoten zeigt, und um die sich ergebenden frei gemachten Speicherstellen für neue Verzeichniseinträge verfügbar zu machen.



55. Decodierer gemäß Anspruch 54, wobei, bei Verwendung, jeder verknüpfende Zeiger des zweiten Typs (D) zu einem ersten gespeicherten Symbol einer geordneten Liste (S2, S3, S4) alternativer gespeicherter Symbole des Suchbaumes zeigt und wobei nacheinander gespeicherte Symbole in der geordneten Liste durch verknüpfende Zeiger des ersten Typs (R) verbunden sind, die zu jedem nachfolgenden gespeicherten Symbol in der Liste zeigen.
56. Decodierer gemäß Anspruch 54, wobei, bei Verwendung, jeder verknüpfende Zeiger des zweiten Typs (D) zu irgendeinem aus einer Liste alternativer gespeicherter Symbole des Suchbaumes zeigt, wobei die gespeicherten Symbole in der Liste untereinander durch Zeiger des ersten Typs (R) verbunden sind, die in eine Richtung zeigen, und ebenso durch Zeiger des ersten Typs, die in die entgegengesetzte Richtung (L) zeigen, damit auf irgendein gespeichertes Symbol in der Liste zugegriffen werden kann.
57. Decodierer gemäß einem der Ansprüche 54 bis 56, wobei der Prozessor weiterhin angeordnet ist, um alle die indexierten Speicherstellen, die das Verzeichnis bilden, zu prüfen, bevor das nächste zu speichernde Symbol in einer freigemachten Speicherstelle gespeichert wird.
58. Decodierer gemäß einem der Ansprüche 54 bis 56, wobei der Prozessor angeordnet ist, um neulich erzeugte Knoten (b (NEUER EINTRAG); Fig. 8 (a)) des Suchbaumes gegen Löschen zu schützen.
59. Decodierer gemäß einem der Ansprüche 54 bis 58, wobei der Prozessor angeordnet ist, um mittels Zeigern jene indexierten Speicherstellen zu verknüpfen, die keine Knoten des Suchbaumes, der durch Zeiger verknüpft ist, enthalten, um eine freie Liste (Fig. 1 (b), Fig. 2 (b)) zu bilden, und um Knoten des Suchbaumes durch Rücksetzen von Zeigern zu löschen, um die gelöschten Knoten zu umgehen und indem sie mit der freien Liste verbunden werden, wodurch der Suchbaum als ein verbundenes Ganzes aufrechterhalten wird.
60. Decodierer gemäß einem der Ansprüche 54 bis 59, wobei jeder Knoten des Suchbaumes zu einem jeweiligen Zähler gehört, der jedesmal inkrementiert wird, wenn der zugehörige Knoten verwendet wird, und wobei der Prozessor angeordnet ist, um komprimierte Ausgabedaten bereitzustellen, die Codewörter aufweisen, deren Längen mit den Inhalten der Zähler in Beziehung stehen, so daß die kürzesten Codewörter die am häufigsten verwendeten Knoten darstellen.
61. Decodierer gemäß einem der Ansprüche 54 bis 59, wobei die Knoten des Suchbaumes in einer Reihenfolge in dem Speicher gespeichert werden und wobei der Prozessor angeordnet ist, um die Reihenfolge, in der sie angeordnet sind, neu anzuordnen durch Erhöhen der Ordnungszahl eines Knotens, nachdem er verwendet worden ist, wodurch selten verwendete Knoten eine niedrige Ordnungszahl annehmen, und um Knoten mit den niedrigsten Ordnungszahlen zu löschen.
62. Decodierer gemäß Anspruch 61, wobei die Ordnungszahl des verwendeten Knotens um eins erhöht wird, und wobei die Ordnungszahl des Knotens unmittelbar oberhalb des verwendeten Knotens um eins erniedrigt wird, wodurch diese zwei Knoten Ordnungszahlen austauschen.
63. Decodierer gemäß Anspruch 61, wobei die Ordnungszahl des verwendeten Knotens auf den Maximalwert erhöht wird, und wobei die Ordnungszahlen aller Knoten oberhalb des verwendeten Knotens um eins erniedrigt werden.
64. Decodierer gemäß einem der Ansprüche 61 bis 63, wobei jeder Knoten des Suchbaumes einen zugehörigen Längenindex hat, und wobei der Prozessor angeordnet ist, um komprimierte Ausgabedaten bereitzustellen, die Codewörter aufweisen, deren Längen mit den Längenindizes in Beziehung stehen, so daß die kürzesten Codewörter Knoten mit den höchsten Ordnungszahlen darstellen.
65. Decodierer gemäß einem der Ansprüche 54 bis 64, wobei, bei Verwendung, der Prozessor angeordnet ist, um zu erfassen, wenn empfangene komprimierte Daten leeren oder freien Speicherstellen entsprechen, und um ein entsprechendes Ausgabesignal zur Übertragung an einen zugehörigen Codierer zu erzeugen.



66. Decodierer gemäß einem der Ansprüche 54 bis 65, wobei das Verzeichnis zur weiteren Verwendung aufbewahrt wird, und wobei der Prozessor angeordnet ist, um eine Prüfsummenberechnung an dem Verzeichnis vor irgendeiner solchen weiteren Verwendung auszuführen und um ein entsprechendes Ausgabesignal zur Übertragung an einen zugehörigen Codierer zu erzeugen, um diese Prüfsumme mit einer entsprechenden Prüfsumme zu vergleichen, die, bei Verwendung, von dem Codierer empfangen wird, und um das Verzeichnis neu zu initialisieren, wenn die Prüfsummen nicht identisch sind.

67. Datenverarbeitungsvorrichtung, die einen Codierer (2) gemäß einem der Ansprüche 36 bis 53, einen entsprechenden Decodierer (5) gemäß einem der Ansprüche 54 bis 66 und eine Datenverbindung zur Übertragung der komprimierten Daten zwischen dem Codierer und dem Decodierer aufweist.

68. Datenverarbeitungsvorrichtung, die einen Codierer (2) gemäß einem der Ansprüche 36 bis 50, einen entsprechenden Decodierer (5) gemäß einem der Ansprüche 54 bis 57 und ein Massenspeichermedium (1), auf das der Codierer und der Decodierer zugreifen kann, aufweist.

Revendications

1. Un procédé de compression de donnée d'entrée comprenant les étapes consistant à : lire des symboles successifs (a, b, c) de la donnée à l'aide d'un processeur (μ P) pourvu d'une mémoire (3) à emplacements indexés de mémoire, engendrer dans la mémoire, à partir de chaînes de symboles de la donnée d'entrée, un dictionnaire en forme d'arbre de recherche de symboles (Figures 1 à 3, 6, 8), arbre de recherche qui est pourvu de trajets représentatifs desdites chaînes, faire concorder des chaînes de symboles de la donnée d'entrée avec des trajets mémorisés au préalable de l'arbre de recherche et engendrer à partir des trajets mémorisés une donnée comprimée de sortie correspondant à la donnée d'entrée, des symboles mémorisés (S) de l'arbre de recherche étant reliés pour former lesdits trajets en reliant des indicateurs de deux types distincts; un indicateur (R) du premier type entre des symboles mémorisés indiquant que ces symboles mémorisés sont des variantes possibles de symboles à une position donnée dans une séquence de symboles d'entrée et un indicateur (D) du deuxième type entre des symboles mémorisés indiquant que ces symboles mémorisés apparaissent l'un et l'autre, dans l'ordre, dans une séquence éventuelle de symboles d'entrée : caractérisé en ce que, lorsque la mémoire est pleine, des emplacements indexés séquentiels de mémoire de l'arbre de recherche (Figures 1 à 3, 6, 8) sont essayés et sont annulés s'ils contiennent un noeud (c, a; Figure 8(a)) de l'arbre de recherche qui ne possède pas d'indicateur de liaison du deuxième type (D) indiquant un autre noeud (Figure 9), et les emplacements libérés de mémoire qui en résultent sont rendus disponibles pour de nouvelles entrées du dictionnaire.
2. Un procédé selon la revendication 1 dans lequel tous les emplacements indexés de mémoire formant le dictionnaire sont essayés avant que le symbole suivant à mémoriser ne soit mémorisé dans un emplacement libéré de mémoire.
3. Un procédé selon la revendication 1 ou la revendication 2 modifié par la stipulation que des noeuds récemment créés (b (NOUVELLE ENTREE), Figure 8(a)) de l'arbre de mémoire sont protégés contre une annulation.
4. Un procédé selon l'une quelconque des revendications 1 à 3, dans lequel les emplacements indexés de mémoire qui contiennent pas de noeuds de l'arbre de recherche sont reliés à des indicateurs afin de former une liste libre (Figure 1(b), Figure 2(b)) et des noeuds de l'arbre de recherche sont annulés en restaurant des indicateurs afin de contourner les noeuds annulés et de les lier à la liste libre, grâce à quoi l'arbre de recherche est maintenu sous forme d'un ensemble lié.
5. Un procédé selon l'une quelconque des revendications 1 à 4 dans lequel chaque noeud de l'arbre de recherche est associé à un compteur respectif qui est incrémenté chaque fois que le noeud associé est utilisé, et la donnée comprimée de sortie comprend des mots de code dont les longueurs sont en relation avec les contenus des compteurs d'une manière telle que les mots de code les plus courts représentent les noeuds les plus fréquemment utilisés.
6. Un procédé selon l'une quelconque des revendications 1 à 4 dans lequel les noeuds de l'arbre de recherche sont mémorisés dans l'ordre dans la mémoire et l'ordre dans lequel ils sont mémorisés est



agencé de nouveau en élevant la valeur ordinale d'un noeud lorsqu'il a été utilisé de sorte que des noeuds rarement utilisés acquièrent une faible valeur ordinale, et sont annulés.

7. Un procédé selon la revendication 6 dans lequel la valeur ordinale du noeud utilisé est augmentée de un, et la valeur ordinale du noeud situé immédiatement au-dessus du noeud utilisé est diminuée de un, de sorte que ces deux noeuds échangent des valeurs ordinales.
8. Un procédé selon la revendication 6 dans lequel la valeur ordinale du noeud utilisé est augmentée à la valeur maximale, et les valeurs ordinales de tous les noeuds situés au-dessus du noeud utilisé sont diminuées de un.
9. Un procédé selon l'une quelconque des revendications 6 à 8 dans lequel chaque noeud de l'arbre de recherche est pourvu d'un index de longueur qui lui est associé et la donnée comprimée de sortie comprend des mots de code dont les longueurs sont en relation avec les indices de longueurs de sorte que les mots de code les plus courts représentent des noeuds dont les valeurs ordinales sont les plus élevées.
10. Un procédé selon l'une quelconque des revendications précédentes dans lequel les données d'entrée incluent des espaces entre des séquences de symboles et le procédé de génération d'un nouveau trajet mémorisé de l'arbre de recherche est terminé lorsqu'un tel espace est détecté.
11. Un procédé selon une revendication précédente quelconque dans lequel la donnée d'entrée consiste en un train de chiffres binaires (bits) représentant des caractères, et les symboles (S) mémorisés dans l'arbre de recherche sont constitués chacun de séquences de chiffres binaires (bits), et dans lequel le nombre de bits par séquence est choisi par le processeur (μP), le nombre de bits par caractère de la donnée d'entrée étant, soit inconnu, soit différent du nombre de bits par séquence choisi par le processeur.
12. Un procédé selon la revendication 11 dans lequel le processeur (μP) est agencé pour effectuer le choix en réponse à un signal extérieur d'ordre venant d'un utilisateur.
13. Un procédé selon la revendication 11 ou la revendication 12 dans lequel on fait varier initialement le nombre de bits par séquence et le rapport de compression qui en résulte entre la donnée d'entrée et la donnée de sortie est mesuré, et le nombre de bits, par séquence, des symboles mémorisés de l'arbre de recherche est choisi sur la base du rapport de compression mesuré.
14. Un procédé selon une revendication précédente quelconque dans lequel chaque indicateur de liaison du deuxième type (D) indique un premier symbole mémorisé d'une liste ordonnée (S2, S3, S4) de variantes de symboles mémorisés de l'arbre de recherche, et des symboles mémorisés successifs dans ladite liste ordonnée sont liés par des indicateurs de liaison du premier type (R) qui indiquent chaque symbole mémorisé qui suit dans ladite liste.
15. Un procédé selon l'une quelconque des revendications 1 à 13 dans lequel chaque indicateur de liaison du deuxième type (D) indique un symbole quelconque d'une liste de variantes de symboles mémorisés de l'arbre de recherche, les symboles mémorisés dans ladite liste étant liés l'un à l'autre par des indicateurs du premier type (R) indiquant un sens et aussi par des indicateurs du premier type indiquant le sens opposé (L) de façon à permettre qu'il soit accédé à un symbole mémorisé quelconque de la liste.
16. Un procédé selon la revendication 14 ou la revendication 15 dans lequel des indicateurs de liaison du premier type (R) sont utilisés pour explorer la liste des variantes de symboles mémorisés afin de trouver une concordance pour le symbole d'entrée le plus récemment lu et pour obtenir, si une concordance est trouvée, l'indicateur du deuxième type (D) s'il est présent.
17. Un procédé selon une revendication précédente quelconque dans lequel des symboles mémorisés (a, b, c) existent initialement dans la mémoire avant que la donnée d'entrée ne soit traitée, chacun de ces symboles correspondant à une séquence de symboles dont l'apparition dans la donnée d'entrée est probable, lesdits symboles mémorisés existant initialement étant mémorisés en tant que noeuds dans



l'arbre de recherche.

18. Un procédé selon l'une quelconque des revendications 1 à 7 dans lequel le dictionnaire est initialisé de nouveau en réponse à un signal d'ordre reçu, en utilisation, d'un décodeur associé.

19. Un procédé selon l'une quelconque des revendications 1 à 17 dans lequel un total de contrôle du dictionnaire est calculé périodiquement, en utilisation, et un signal de sortie correspondant est engendré.

20. Un procédé selon l'une quelconque des revendications 1 à 17, dans lequel le dictionnaire est retenu en vue d'une utilisation ultérieure et qui inclut une exécution d'un calcul de total de contrôle sur le dictionnaire avant toute utilisation ultérieure de ce type, une réception d'un total de contrôle correspondant provenant d'un autre dictionnaire de ce type, une comparaison des totaux de contrôle et une nouvelle initialisation du dictionnaire si les totaux de contrôle ne sont pas identiques.

21. Un procédé de mémorisation de donnée comprenant l'étape consistant à comprimer la donnée selon un procédé conforme à l'une quelconque des revendications 1 à 17 et à mémoriser la donnée comprimée dans un milieu de mémoire de masse (1).

22. Un procédé de décodage d'une donnée comprimée qui a été comprimée par un procédé selon l'une quelconque des revendications 1 à 20 comprenant les étapes consistant à lire, à l'aide d'un processeur (μP) pourvu d'une mémoire (3), des caractères successifs de la donnée comprimée, mémoriser dans la mémoire d'un dictionnaire sous forme d'un arbre de recherche (Figures 1 à 3, 6, 8) de symboles qui est constitué de la donnée comprimée et utiliser l'arbre de recherche pour transmettre en une donnée décodée la donnée comprimée, des symboles de mémoire (S) de l'arbre de recherche étant reliés par des indicateurs de liaison de deux types distincts; un indicateur du premier type (R) entre symboles mémorisés indiquant que ces symboles sont associés à des chaînes décodées différentes de symboles pourvues du même nombre de symboles et du même préfixe et sont les derniers symboles respectifs de ces différentes chaînes et un indicateur du deuxième type (D) entre des symboles mémorisés indiquant que ces symboles sont des symboles successifs dans une chaîne de symboles décodés de sortie : caractérisé en ce que, lorsque la mémoire est pleine, des emplacements indexés séquentiels de mémoire de l'arbre de recherche (Figures 1 à 3, 6, 8) sont essayés et sont annulés s'ils contiennent un noeud (c, a; Figure 8(a)) de l'arbre de recherche qui ne possède pas d'indicateur de liaison du deuxième type (D) indiquant un autre noeud (Figure 9), et les emplacements libérés de mémoire qui en résultent sont rendus disponibles pour de nouvelles entrées du dictionnaire.

23. Un procédé selon la revendication 22 dans lequel tous les emplacements indexés de mémoire formant le dictionnaire sont essayés avant que le symbole suivant à mémoriser ne soit mémorisé dans un emplacement libéré de mémoire.

24. Un procédé selon la revendication 22 ou la revendication 23 modifié par la stipulation que des noeuds récemment créés (b (NOUVELLE ENTREE), Figure 8(a)) de l'arbre de mémoire sont protégés contre une annulation.

25. Un procédé selon l'une quelconque des revendications 22 à 24, dans lequel les emplacements indexés de mémoire qui contiennent pas de noeuds de l'arbre de recherche sont reliés à des indicateurs afin de former une liste libre (Figure 1(b), Figure 2(b)) et des noeuds de l'arbre de recherche sont annulés en restaurant des indicateurs afin de contourner les noeuds annulés et de les lier à la liste libre, grâce à quoi l'arbre de recherche est maintenu sous forme d'un ensemble lié.

26. Un procédé selon l'une quelconque des revendications 22 à 25 dans lequel chaque noeud de l'arbre de recherche est associé à un compteur respectif qui est incrémenté chaque fois que le noeud associé est utilisé, les mots de code les plus courts sont alloués aux noeuds les plus fréquemment utilisés, et la donnée comprimée à décoder comprend des mots de code parmi lesquels les mots de code les plus courts représentent les noeuds les plus fréquemment utilisés de l'arbre de recherche d'un encodeur correspondant.



27. Un procédé selon l'une quelconque des revendications 22 à 25 dans lequel les noeuds de l'arbre de recherche sont mémorisés dans l'ordre dans la mémoire et l'ordre dans lequel ils sont mémorisés est agencé de nouveau en élevant la valeur ordinale d'un noeud lorsqu'il a été utilisé de sorte que des noeuds rarement utilisés acquièrent une faible valeur ordinale, et sont annulés.

28. Un procédé selon la revendication 27 dans lequel la valeur ordinale du noeud utilisé est augmentée de un, et la valeur ordinale du noeud situé immédiatement au-dessus du noeud utilisé est diminuée de un, de sorte que ces deux noeuds échangent des valeurs ordinales.

29. Un procédé selon la revendication 27 dans lequel la valeur ordinale du noeud utilisé est augmentée à la valeur maximale, et les valeurs ordinales de tous les noeuds situés au-dessus du noeud utilisé sont diminuées de un.

30. Un procédé selon l'une quelconque des revendications 27 à 29 dans lequel chaque noeud de l'arbre de recherche est pourvu d'un index de longueur qui lui est associé d'une manière telle que les mots de code les plus courts représentent les noeuds dont les valeurs ordinales sont les plus élevées et la donnée comprimée à décoder comprend des mots de code dont les longueurs sont liées aux indices de longueur d'une manière telle que les mots de code les plus courts représentent les noeuds de l'arbre de recherche, d'un encodeur correspondant, dont les valeurs ordinales sont les plus élevées.

31. Un procédé selon la revendication 22 dans lequel chaque indicateur de liaison du deuxième type (D) indique un premier symbole mémorisé d'une liste ordonnée (S2, S3, S4) de variantes de symboles mémorisés de l'arbre de recherche, et des symboles mémorisés successifs dans ladite liste ordonnée sont liés par des indicateurs de liaison du premier type (R) qui indiquent chaque symbole mémorisé qui suit dans ladite liste.

32. Un procédé selon la revendication 22 dans lequel chaque indicateur de liaison du deuxième type (D) indique un symbole quelconque d'une liste de variantes de symboles mémorisés de l'arbre de recherche, les symboles mémorisés dans ladite liste étant liés l'un à l'autre par des indicateurs du premier type (R) indiquant un sens et aussi par des indicateurs du premier type indiquant le sens opposé (L) de façon à permettre qu'il soit accédé à un symbole mémorisé quelconque de la liste.

33. Un procédé selon l'une quelconque des revendication 22 à 32 incluant les étapes consistant à détecter quand une donnée comprimée reçue correspond à des emplacements de mémoire vides ou libres et à engendrer un signal de sortie correspondant.

34. Un procédé selon l'une quelconque des revendication 22 à 32 dans lequel le dictionnaire est retenu pour une utilisation ultérieure et incluant les étapes consistant à réaliser sur le dictionnaire un calcul de total de contrôle avant toute utilisation ultérieure de ce type, recevoir un total de contrôle correspondant d'un autre dictionnaire de ce type, comparer les totaux de contrôle et initialiser de nouveau le dictionnaire si les totaux de contrôle ne sont pas identiques.

35. Un procédé de transmission de donnée comprenant les étapes consistant à encoder la donnée par un procédé selon l'une quelconque des revendications 1 à 20, transmettre la donnée comprimée qui en résulte à un emplacement situé à distance et décoder la donnée comprimée par un procédé correspondant selon la revendication appropriée parmi les revendications 22 ou 23.

36. Un encodeur de compression d'une donnée d'entrée comprenant un microprocesseur (μ P) susceptible de recevoir des symboles successifs (a, b, c) de donnée d'entrée, une mémoire (3) à emplacement indexés de mémoire, un moyen de mémorisation, dans la mémoire, d'un arbre de recherche (Figures 1 à 3, 6, 8) de symboles pourvu de trajets représentatifs de chaînes de symboles de la donnée d'entrée, et un moyen prévu pour mettre en concordance des chaînes de symboles de la donnée d'entrée avec des trajets mémorisés précédemment de l'arbre de recherche, et pour engendrer à partir des trajets mémorisés une donnée de sortie comprimée correspondant à la donnée d'entrée, des symboles mémorisés (S) de l'arbre de recherche (Figures 1 à 3, 6, 8) étant reliés de façon à former lesdits trajets en reliant des indicateurs de deux types distincts; un indicateur (R) du premier type entre des symboles mémorisés indiquant que ces symboles mémorisés sont des variantes possibles de symboles à une position donnée dans une séquence de symboles d'entrée et un indicateur (D) du deuxième



type entre des symboles mémorisés indiquant que ces symboles mémorisés apparaissent l'un et l'autre, dans l'ordre, dans une séquence éventuelle de symboles d'entrée : caractérisé en ce que le processeur (μP) est agencé, en utilisation, de manière à déterminer quand la mémoire (3) est pleine, à essayer des emplacements indexés séquentiels de mémoire de l'arbre de recherche, à annuler un emplacement de mémoire s'il contient un noeud (c, a; Figure 8(a)) de l'arbre de recherche qui n'est pas pourvu d'un indicateur de liaison du deuxième type (D) indiquant un autre noeud et à rendre disponible pour de nouvelles entrées de dictionnaire les emplacements mémorisés de mémoire qui en résultent.

37. Un encodeur selon la revendication 36 dans lequel le processeur est en outre agencé de manière à essayer tous les emplacements de mémoire indexés formant le dictionnaire avant que le symbole suivant à mémoriser ne soit mémorisé dans un emplacement libéré de mémoire.
38. Un encodeur selon la revendication 36 dans lequel le processeur est en outre agencé de manière à protéger, contre une annulation, des noeuds récemment créés (b (NOUVELLE ENTREE), Figure 8(a)) de l'arbre de mémoire.
39. Un encodeur selon la revendication 36 dans lequel le processeur est en outre agencé de manière à relier par des indicateurs les emplacements de mémoire qui ne contiennent pas de noeuds de l'arbre de recherche afin de former une liste libre (Figure 1(b), Figure 2(b)) et à annuler des noeuds de l'arbre de recherche en restaurant des indicateurs afin de contourner les noeuds annulés et de les lier à la liste libre, grâce à quoi l'arbre de recherche est maintenu sous forme d'un ensemble lié.
40. Un encodeur selon la revendication 36 dans lequel chaque noeud de l'arbre de recherche est associé à un compteur respectif qui est incrémenté chaque fois que le noeud associé est utilisé, et le processeur est agencé de manière à produire la donnée comprimée de sortie comprenant des mots de code dont les longueurs sont en relation avec les contenus des compteurs d'une manière telle que les mots de code les plus courts représentent les noeuds les plus fréquemment utilisés.
41. Un encodeur selon la revendication 36 dans lequel les noeuds de l'arbre de recherche sont mémorisés dans l'ordre dans la mémoire et le processeur est agencé de manière à agencer de nouveau l'ordre dans lequel ils sont mémorisés, en élevant la valeur ordinale d'un noeud lorsqu'il a été utilisé de sorte que des noeuds rarement utilisés acquièrent une faible valeur ordinale, et à annuler les noeuds dont la valeur ordinale est la plus faible.
42. Un encodeur selon la revendication 41 dans lequel la valeur ordinale du noeud utilisé est augmentée de un, et la valeur ordinale du noeud situé immédiatement au-dessus du noeud utilisé est diminuée de un, de sorte que ces deux noeuds échangent des valeurs ordinales.
43. Un encodeur selon la revendication 41 dans lequel la valeur ordinale du noeud utilisé est augmentée à la valeur maximale, et les valeurs ordinales de tous les noeuds situés au-dessus du noeud utilisé sont diminuées de un.
44. Un encodeur selon l'une des revendication 41 à 43 dans lequel un index de longueur est associé à chaque noeud de l'arbre de recherche et le processeur est agencé de manière à produire la donnée comprimée de sortie comprenant des mots de code dont les longueurs sont en relation avec les indices de longueurs de sorte que les mots de code les plus courts représentent des noeuds dont les valeurs ordinales sont les plus élevées.
45. Un encodeur selon l'une quelconque des revendications 36 à 44 destiné à encoder des données d'entrée incluant des espaces entre des séquences de symboles et dans lequel le processeur est agencé de manière à terminer le processus de génération d'un nouveau trajet mémorisé de l'arbre de recherche lorsqu'un tel espace est détecté.
46. Un encodeur selon l'une quelconque des revendications 36 à 45 destiné à encoder des données d'entrée consistant en un train de chiffres binaires (bits) représentant des caractères, où les symboles (S) mémorisés dans l'arbre de recherche sont constitués chacun de séquences de chiffres binaires (bits), et dans lequel le processeur est agencé de manière à choisir le nombre de bits par séquence de



manière qu'il soit différent du nombre de bits par caractère de la donnée d'entrée, si celui-ci est connu.

47. Un encodeur selon la revendication 46 dans lequel le processeur (μP) est agencé de manière à effectuer le choix en réponse à un signal extérieur d'ordre venant d'un utilisateur.

48. Un encodeur selon la revendication 46 ou la revendication 47 dans lequel on fait varier initialement le nombre de bits par séquence choisi par le processeur et le rapport de compression qui en résulte entre la donnée d'entrée et la donnée de sortie est mesuré, et le nombre de bits, par séquence, des symboles mémorisés de l'arbre de recherche est choisi sur la base du rapport de compression mesuré.

49. Un encodeur selon l'une quelconque des revendications 36 à 48 dans lequel chaque indicateur de liaison du deuxième type (D) indique un premier symbole mémorisé d'une liste ordonnée (S2, S3, S4) de variantes de symboles mémorisés de l'arbre de recherche, et des symboles mémorisés successifs de ladite liste ordonnée sont liés par des indicateurs de liaison du premier type (R) qui indiquent chaque symbole mémorisé suivant dans ladite liste.

50. Un encodeur selon l'une quelconque des revendications 36 à 48 dans lequel chaque indicateur de liaison du deuxième type (D) indique l'un quelconque des symboles d'une liste de variantes de symboles mémorisés de l'arbre de recherche, les symboles mémorisés de ladite liste étant liés entre eux par des indicateurs du premier type (R) indiquant un sens et aussi par des indicateurs du premier type indiquant le sens opposé (L) de façon à permettre qu'il soit accédé à un symbole mémorisé quelconque de la liste.

51. Un encodeur selon l'une quelconque des revendications 36 à 50 dans lequel le processeur est sensible, en utilisation, à un signal d'ordre reçu d'un décodeur associé pour initialiser de nouveau son dictionnaire.

52. Un encodeur selon l'une quelconque des revendications 36 à 50 dans lequel le processeur, en utilisation, calcule périodiquement un total de contrôle du dictionnaire et engendre un signal de sortie correspondant.

53. Un encodeur selon l'une quelconque des revendications 36 à 50 dans lequel le dictionnaire est retenu en vue d'une utilisation ultérieure et le processeur est agencé de manière à effectuer un calcul de total de contrôle du dictionnaire avant toute utilisation ultérieure de ce type et à engendrer un signal de sortie correspondante, à comparer ce total de contrôle avec un total de contrôle correspondant reçu, en utilisation, d'un décodeur associé, et à initialiser de nouveau le dictionnaire à un état initial si les totaux de contrôle ne sont pas identiques.

54. Un décodeur de décodage d'une donnée comprimée comprenant un microprocesseur (μP) susceptible de recevoir des symboles successifs de donnée comprimée, une mémoire (3) à emplacements indexés de mémoire et un moyen de mémorisation, dans la mémoire, d'un dictionnaire sous forme d'un arbre de recherche (Figures 1 à 3, 6, 8) de symboles, le processeur étant agencé de manière à constituer l'arbre de recherche à partir de la donnée comprimée et à utiliser l'arbre de recherche pour traduire la donnée comprimée en une donnée décodée, des symboles mémorisés (S) de l'arbre de recherche étant reliés par des indicateurs de liaison de deux types distincts; un indicateur du premier type (R) entre symboles mémorisés indiquant que ces symboles sont associés à des chaînes décodées différentes de symboles pourvues du même nombre de symboles et du même préfixe et sont les derniers symboles respectifs de ces différentes chaînes et un indicateur du deuxième type (D) entre des symboles mémorisés indiquant que ces symboles sont des symboles successifs dans une chaîne de symboles décodés de sortie : caractérisé en ce que le processeur (μP) est agencé, en utilisation, de manière à déterminer quand la mémoire (3) est pleine, à essayer des emplacements indexés séquentiels de mémoire de l'arbre de recherche, à annuler un emplacement de mémoire s'il contient un noeud (c, a; Figure 8(a)) de l'arbre de recherche qui n'est pas pourvu d'un indicateur de liaison du deuxième type (D) indiquant un autre noeud et à rendre disponible pour de nouvelles entrées de dictionnaire les emplacements mémorisés de mémoire qui en résultent.



55. Un décodeur selon la revendication 54 dans lequel chaque indicateur de liaison du deuxième type, en utilisation, indique un premier symbole mémorisé d'une liste ordonnée (S2, S3, S4) de variantes de symboles mémorisés de l'arbre de recherche, et des symboles mémorisés successifs de ladite liste ordonnée sont liés par des indicateurs de liaison du premier type (R) qui indiquent chaque symbole mémorisé suivant de ladite liste.
56. Un décodeur selon la revendication 54 dans lequel chaque indicateur de liaison du deuxième type (D), en utilisation, indique l'un quelconque des symboles d'une liste de variante de symboles mémorisés de l'arbre de recherche, les symboles mémorisés de ladite liste étant liés entre eux par des indicateurs du premier type (R) indiquant un sens et aussi par des indicateurs du premier type indiquant le sens opposé (L) de façon à permettre qu'il soit accédé à un symbole mémorisé quelconque de la liste.
57. Un décodeur selon l'une quelconque des revendications 54 à 56 dans lequel le processeur est agencé de manière à essayer tous les emplacements de mémoire indexés formant le dictionnaire avant que le symbole suivant à mémoriser ne soit mémorisé dans un emplacement libéré de mémoire.
58. Un décodeur selon l'une quelconque des revendications 54 à 56 dans lequel le processeur est agencé de manière à protéger, contre une annulation, des noeuds récemment créés (b (NOUVELLE ENTREE), Figure 8(a)) de l'arbre de mémoire.
59. Un décodeur selon l'une quelconque des revendications 54 à 58 dans lequel le processeur est agencé de manière à relier par des indicateurs les emplacements de mémoire qui ne contiennent pas de noeuds de l'arbre de recherche afin de former une liste libre (Figure 1(b), Figure 2(b)) et à annuler des noeuds de l'arbre de recherche en restaurant des indicateurs afin de contourner les noeuds annulés et de les lier à la liste libre, grâce à quoi l'arbre de recherche est maintenu sous forme d'un ensemble lié.
60. Un décodeur selon l'une quelconque des revendications 54 à 59 dans lequel chaque noeud de l'arbre de recherche est associé à un compteur respectif qui est incrémenté chaque fois que le noeud associé est utilisé, et le processeur est agencé de manière à produire la donnée comprimée de sortie comprenant des mots de code dont les longueurs sont en relation avec les contenus des compteurs d'une manière telle que les mots de code les plus courts représentent les noeuds les plus fréquemment utilisés.
61. Un décodeur selon l'une quelconque des revendications 54 à 59 dans lequel les noeuds de l'arbre de recherche sont mémorisés dans l'ordre dans la mémoire et le processeur est agencé de manière à agencer de nouveau l'ordre dans lequel ils sont mémorisés, en élevant la valeur ordinale d'un noeud lorsqu'il a été utilisé de sorte que des noeuds rarement utilisés acquièrent une faible valeur ordinale, et à annuler les noeuds dont la valeur ordinale est la plus faible.
62. Un décodeur selon la revendication 61 dans lequel la valeur ordinale du noeud utilisé est augmentée de un, et la valeur ordinale du noeud situé immédiatement au-dessus du noeud utilisé est diminuée de un, de sorte que ces deux noeuds échangent des valeurs ordinales.
63. Un décodeur selon la revendication 61 dans lequel la valeur ordinale du noeud utilisé est augmentée à la valeur maximale, et les valeurs ordinales de tous les noeuds situés au-dessus du noeud utilisé sont diminuées de un.
64. Un décodeur selon l'une des revendications 61 à 63 dans lequel un index de longueur est associé à chaque noeud de l'arbre de recherche et le processeur est agencé de manière à produire la donnée comprimée de sortie comprenant des mots de code dont les longueurs sont en relation avec les indices de longueurs de sorte que les mots de code les plus courts représentent des noeuds dont les valeurs ordinales sont les plus élevées.
65. Un décodeur selon l'une quelconque des revendications 54 à 64 dans lequel le processeur est agencé, en utilisation, de manière à détecter quand une donnée reçue comprimée correspond à des emplacements de mémoire vides ou libres et à engendrer un signal de sortie correspondant en vue d'une transmission à un encodeur associé.



66. Un décodeur selon l'une quelconque des revendications 54 à 65 dans lequel le dictionnaire est retenu en vue d'une utilisation ultérieure et le processeur est agencé de manière à effectuer un calcul de total de contrôle du dictionnaire avant toute utilisation ultérieure de ce type et à engendrer un signal de sortie correspondant en vue d'une transmission à un encodeur associé, à comparer ce total de contrôle avec un total de contrôle correspondant reçu, en utilisation, dudit encodeur, et à initialiser de nouveau le dictionnaire à un état initial si les totaux de contrôle ne sont pas identiques.

67. Appareil de traitement de donnée comprenant un encodeur (2) selon l'une quelconque des revendications 36 à 53, un décodeur correspondant (5) selon l'une quelconque des revendications 54 à 66 et une liaison de donnée pour transmettre la donnée comprimée entre l'encodeur et le décodeur.

68. Appareil de traitement de donnée comprenant un encodeur (2) selon l'une quelconque des revendications 36 à 50, un décodeur correspondant (5) selon l'une quelconque des revendications 54 à 57 et un milieu de mémoire de masse (1) accessible par l'encodeur et le décodeur.