Trie Memory*

EDWARD FREDKIN, *Bolt Beranek and Newman, Inc., Cambridge, Mass.*

Introduction

Trie memory is a way of storing and retrieving information.[1] It is applicable to information that consists of function-argument (or item-term) pairs—information conventionally stored in unordered lists, ordered lists, or pigeonholes.

The main advantages of trie memory over the other memory plans just mentioned are shorter access time, greater ease of addition or up-dating, greater convenience in handling arguments of diverse lengths, and the ability to take advantage of redundancies in the information stored. The main disadvantage is relative inefficiency in using storage space, but this inefficiency is not great when the store is large.

In this paper several paradigms of trie memory are described and compared with other memory paradigms, their advantages and disadvantages are examined in detail, and applications are discussed.

Many essential features of trie memory were mentioned by de la Briandais [1] in a paper presented to the Western Joint Computer Conference in 1959. The present development is essentially independent of his, having been described in memorandum form in January 1959 [2], and it is fuller in that it considers additional paradigms (finite-dimensional trie memories) and includes experimental results bearing on the efficiency of utilization of storage space.

Basic Paradigm of Trie Memory

Let us consider first a simple abstract form of trie memory. Suppose that we need to keep track of a set of words, a set of sequences of alphabetic characters. The words are of various lengths. From time to time, additions to the set and deletions from it must be made. What we have to remember is just which ones, of all the possible finite sequences of alphabetic characters, are currently in the set. That is, given a word, we must be able to determine whether or not it is at present a member. In this example, each word is an argument, and the corresponding function is simply the binary variable of which the admissible values are *member* and *nonmember*.

At the outset, before a storage is begun, the trie is merely a collection of *registers*. Except for two special registers, which we may call α and δ, every register has a *cell* for each member (type) of the ensemble of alphabetic characters. If we let that ensemble include a "space" to indicate the end of a word (argument), each register must have 27 cells.

Each cell has space for the address of any register in the memory. Cells in the trie that are not yet being used to represent stored information always contain the address of the special α register. A cell thus represents stored information if it contains the address of some register other than α. The information it represents is its own name, "A" for the A cell, "B" for the B cell, etc., and the address of the next register in the sequence.

Storage of words of alphabetic characters is illustrated in Fig. 1. For the sake of simplicity, the ensemble of characters has been restricted to the first five letters of the alphabet and ∇ for "space." Suppose that we want to store DAB, BAD, BADE, BE, BED, BEAD, CAB, CAD, and A. We may follow the procedure illustrated in Fig. 1, in which the rows represent registers, each one

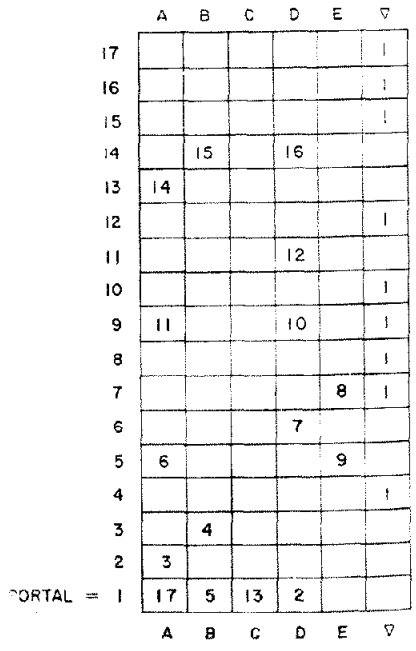

FIG. 1. Schematic representation of storage of words in trie memory

* The work reported here was begun at the MIT Lincoln Laboratory and completed at Bolt Beranek and Newman, Inc., with contractual support from the IBM Federal Systems Division. The author wishes to acknowledge his indebtedness to the many people—colleagues, friends—who have helped on this project. Especially, gratitude is expressed to Dr. J. C. R. Licklider for the most pervasive assistance.

[1] ED. NOTE: "Trie" is apparently derived from reTRIEval.

containing the six cells required by the fact that we are working with six character types. The bottom row represents a third special register, the portal register, through which we enter the memory system. Except for its use as an entrance, it is similar to the other, ordinary registers. The others are numbered in the order in which the α register (not shown) will select them. At the outset, that is, α is designating register 2.

In order to store DAB, we introduce the address "2" into the D cell of the portal register. Then we move to register 2 and introduce the address "3" into the A cell. Then we go to register 3 and put the address "4" into the B cell. And, finally, we go to register 4 and put the address "1" into the ∇, or end-or-argument, cell. That completes storage of the word DAB.

We turn to the second word, BAD. The B is represented by introducing the address "5" into the B cell of the portal register, the A by the address "6" in the fifth register, and the D by the address "7" in the sixth register. BAD is terminated by the address "1" in the seventh register.

When we start to store BADE, we find that the B, and A, and the D are already in the trie. We therefore follow the path already designated by BAD to the seventh register, and there introduce the address "8" in the E cell, next to the address "1" in the ∇ cell of the eighth register.

Following the same procedure, we store the other words of the list.

Besides illustrating the storage procedure, this example reveals that a small trie memory does not efficiently use the available storage space, for there are many unused cells. The question of efficiency of space utilization is one with which we shall be primarily concerned.

It may be evident from the illustration of storage in Fig. 1 that the procedure for reading from the trie is simply to follow the path designated by the addresses, advancing to the indicated level of the trie each time we move to a new character. At each level, we examine the cell that corresponds to the argument character to determine whether or not it contains an address. If it does, we move to that address. If we find ourselves in due course back in register 1, we know that the word is a member of the stored set. If we come to a cell that contains no address,[2] we know that the word is not a member of the stored set. DE... and A..., for example, do not have paths in Fig. 1.

The procedure for writing into the trie has, as its initial part, the procedure for reading. If the character is found by reading, it is not written. If the character is not found, it must be represented by the introduction of an address into the appropriate cell. This facet of the illustration suggests that a possibly important feature of trie memory is its capability of taking advantage of redundancy in the stored sequences. That point will be developed later in the paper.

[2] To avoid the problem of defining "no address," and to anticipate a notation shortly to be introduced, we may say that the blank cells of Fig. 1 contain the address "α."

Nexuses and Nexus Chains

In order to permit more general description of trie memory, it is helpful to substitute, for the system of successive addresses used in connection with the illustrations in Fig. 1, a system of directed connections that we may call nexuses. A brief explication of nexuses and nexus chains will facilitate further discussion of trie memory.

A nexus is a path, or anything that designates a path, from one element of a set to another of the set. In Fig. 1, the address that was introduced into a cell constituted a nexus from that cell to the register identified by the address. An arrow running from the cell to the register would have fulfilled the same function. Since nexuses are directed, we may say (referring to Fig. 1) that cell A of register 1 "points to" register 17, or that register 17 is "designated" by cell A of register 1.

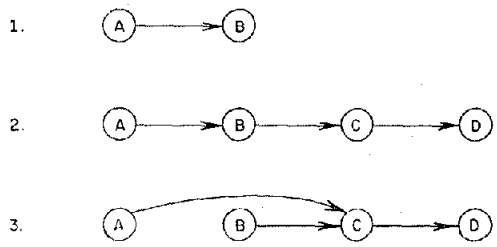

FIG. 2

In Fig. 2, line 1 represents a simple nexus, a directed connection between two members of a set, A and B. Line 2 represents a nexus chain, connecting A to B, B to C, and C to D. In line 3, we "project" the nexus AB one link ahead. The projection converts it into AC. When there is no chance of ambiguity, a nexus AB may be designated as A→ or →B. The effect of the projection is thus simply to move the tip of the nexus AB along BC to C and thereby to define the nexus AC, which we may represent by an arrow from A to C, not through B. With the aid of these simple concepts, we may set forth a rather general description of trie memories. We start out with a particular configuration of nexuses and, as we store information, progressively modify the structure by making a simple pattern of projections.

Storage, Retrieval, and Deletion Operations in Trie Memory

Figure 3 represents, in a form consonant with the concepts just defined, the operations involved in the storage of DAB, the first entry of Fig. 1. As we scan the page we see in succession an empty memory, a representation of the D, a representation of the DA, a representation of the DAB, and, finally, a representation of the entire word including the end-of-word mark (nexus to 1). The configuration of the empty memory (upper left-hand) illustrates the initial state of a memory medium from which a trie can be developed. At the left is the special register α. The lowermost register is the portal register.

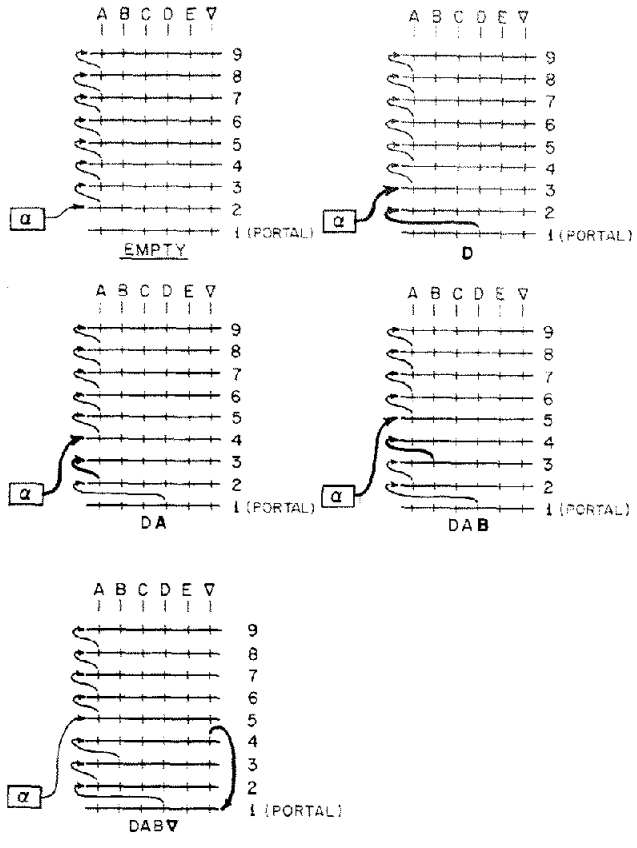

FIG. 3. Successive steps in the storage of a word

Registers 2 through 9 are ordinary registers. Note that, in the empty memory, all the cells of the portal register point to α. (Again, the convention is that blank cells point to α.) Register α points to the register that will be the terminus of the first nexus produced when we start to store information into the trie. In each of the registers 2 through 9, the A cell points to the next register in an ordered list, and all the other cells point to α.

Every cell is the origin of a nexus. Wherever no arrow is shown, the nexus is to α. In the second diagram, D is stored by projecting the (portal D) α nexus one link ahead and then projecting the $\alpha\rightarrow$ nexus one link ahead. The (2A)$\rightarrow$ nexus is then diverted to α, so that all the (2x)$\rightarrow$ nexuses are α nexuses. In the third diagram, A is stored by exactly the same procedure except, of course, that it is the (2A) α nexus that is projected. In the fourth and fifth diagrams, B and ∇ ("space") are stored by repetitions of the procedure.

Classes of Registers in Trie Memory

On the basis of the structural features just mentioned, we may divide the registers of the empty memory into four classes:

(1) the α (address) register, which designates the next address to be used in storing information;

(2) the δ (deletion) register, not yet discussed;

(3) the ν (next) register, into which information will next be stored (in the empty memory, it is the portal register); and

(4) the class χ (exterior) of all the registers that have not yet received stored information and that have not been designated as the depository next in line.

As soon as we begin to store information, we shall create a fifth class:

(5) the class o (occupied) of registers into which information is stored.

Writing into and Reading from Trie's

Of the foregoing classes, all but χ are "in the trie." The storage and reading operations can now be defined fairly simply.

To WRITE:

(1) Enter the next (ν) register with the ith character of the argument. For the first character, this is the portal register.

(2) Select the cell that corresponds to the character. If the ith character of the argument is the jth letter of the alphabet, select cell j.

(3) Examine the nexus originating from the jth cell.

(4) If that nexus leads to the α register:

(a) Project the nexus through the α register one link ahead, thus storing the character.

(b) Then project the nexus originating in the α register one link ahead to create a new "next" register (ν).

(c) Finally make all the nexuses from ν terminate upon α.

(5) If the nexus originating from the jth cell leads to a non-α register, go to that register:

(a) If it is register 1 (which we may select, for example, as a "terminal" register until we come to the problem of storing non-binary functions), the argument is a member of the stored set (end of procedure)

(b) If it is not register 1, increment i by 1 and return to step 2.

The foregoing procedure is illustrated in Fig. 4. In the uppermost diagram, we intend to use the marked cell of register 2 to store a character. It points to α, which points to the next register (4) in the waiting line (4, 5, 1, 3). To effect the storage we project (2x) α to 4, as shown in the middle diagram, then project $\alpha\rightarrow$ to 5 and point all the (4x)$\rightarrow$ at α, as shown in the lowermost diagram.

TO READ: Follow the same procedure, except for the projections. Do not project any nexuses. If the path leads to register 1, the argument is a member of the stored set. If at any point a nexus leads to the α register, on the other hand, the argument is not a member of the stored set.

Deleting Information from Trie Memories

Information may be deleted in either of two ways. One way removes the information completely. The other way renders the entry inoperative but leaves most of it in the memory to facilitate re-storage.

The first operation of either kind of deletion is to establish, through reading, that the entry to be deleted

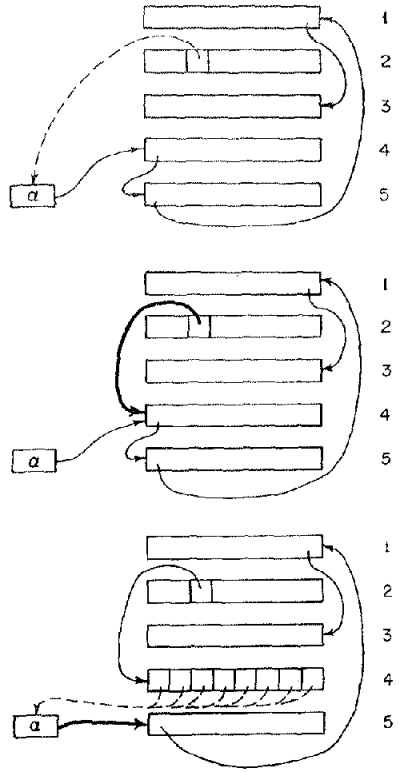

FIG. 4. Storage procedure for trie memory

actually in the memory. Suppose that the argument of the entry to be deleted is a sequence of k characters. As we read our way up through the corresponding k levels of the trie, we examine not only the cells that correspond to the successive characters of the argument, but also all the other cells of every register through which we pass. We examine each such cell to determine whether or not it points to a register other than α. If it does, it contains stored information that shares the path along which we have come. We must not delete the shared path. To keep track of the portion of the path that is shared, we make use of the special cell δ. From it, we create a nexus to the most recently encountered active register. We up-date that record as we ascend the path corresponding to the argument to be deleted. The nexus from δ is different from others we have considered in that it terminates not just upon the register but upon the particular cell of the register through which passes the path to be deleted.

When we come to the kth level of the trie, we find that the cell corresponding to the last character of the argument to be deleted points to register 1. We examine the other cells, as before, to determine whether any of them points to a register other than α. If it does, we note that the argument of the entry being deleted happens to be the same as the initial part of a longer argument that is stored in the trie. Since we do not want to delete the longer argument, we merely delete the end-of-argument marker. That is to say, we make the nexus go to α instead of to register 1. If, as we have been assuming, we are working with binary functions, that completes the deletion. If we find no other cell in the kth-level register that contains stored information, we must decide whether to make a complete deletion or to render the entry nonfunctional. To make the complete deletion, we must return all the registers, down to but not including the one designated by the δ register, to the exterior class χ.

The procedure for returning the registers to the exterior class χ is:

(1) Connect any cell of the register at level k to α and then project it one link ahead;

(2) Connect all the other cells of the kth-level register to α; and

(3) Connect α to δ and then project the $\alpha\delta$ nexus two links ahead.

The procedure for functional deletion—for rendering the entry nonfunctional but leaving its trace in the trie—is simply to remove the end-of-argument mark. This is done by changing the terminal of the nexus that was on register 1 to make it terminate upon α.

De la Briandais' Paradigm

The main shortcomings of full alphabetic and alphanumeric trie's stem from the fact that many cells may be left unused. Registers are assigned only when they are needed, but all the cells of a register are assigned whenever any cell is needed. De la Briandais handled this problem by applying the principle of Newell, Simon, and Shaw's list structure [5]. He provided space in each register for only one cell of the type we have been considering. However, to each register he added a special cell that established a nexus to another register for use in the event of overflow. That procedure increases efficiency of utilization of storage space. However, since it makes it necessary to designate in each cell the character that the cell represents, the scheme is inherently of limited efficiency.

Binary Trie's

Another approach to the problem of correcting the inefficiency that stems from unused cells is to decrease the number of cells per register that are required in principle. Since the smallest usable number is two, binary trie's occupy a fundamental position.[3]

To represent alphanumeric or other nonbinary information in a binary trie of course requires coding, but coding introduces no fundamental problems. Using the binary

TABLE 1

Code Used to Transform Characters of Fig. 1 into Binary Sequences

A	000	D	011
B	001	E	100
C	010	▽	111

[3] For storing a string of symbols without any branches and considering constant size cells, the optimum number of cells per register is e (2.718 . . .). Trie's with four cells per register and two cells per register are equally efficient and three is better than either. Since the number of bits per cell is determined by the log of the number of registers and since there will be branches in symbol strings, it may well be that four branches per level (cells per register) will be closest to optimum, and most convenient to use.

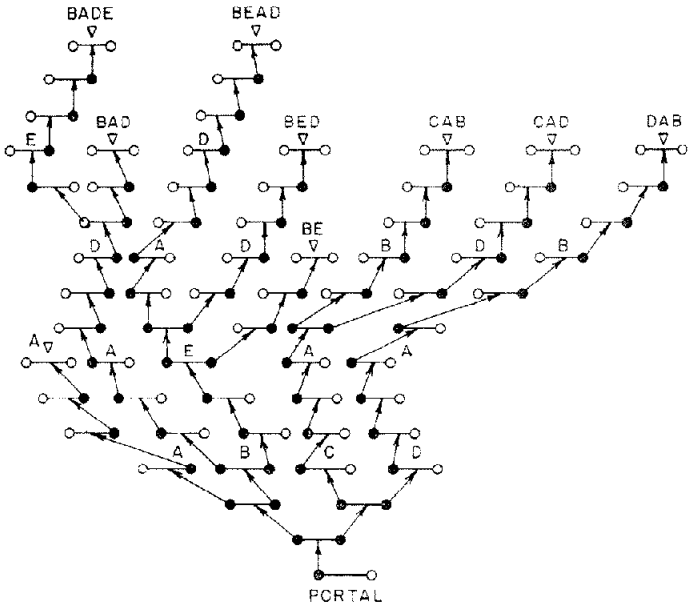

FIG. 5. Storage of words in a character-coded binary tree

representation of alphanumeric characters that is given in Table 1, we may store the information of Fig. 1 in the binary trie of Fig. 5. For convenience, we may represent the memory in the form of a *tree*. Note, however, that the branches are constructed as needed. In this respect, a trie memory tree is quite different from a system of pigeonholes accessible through a predetermined decision tree.

Each memory register has only two cells, 0 and 1. Three levels are therefore required to store a character from the 6-character ensemble. Reading and writing are accomplished by exactly the same general procedure described in connection with multi-cell memories. It is important to note that the branching, tree-like structure is a result of an attempt to simplify the diagram; in principle, directly connected registers need not be physically near each other.

N-Dimensional Binary Trie's

As a memory is made larger, the number of bits required to specify one register (location) out of the entire set of registers increases, and we would like to have a way of minimizing the effect of that increase.

One approach is to impose a constraint on the set of registers at level $i + 1$ that are accessible from a register at level i. Such a constraint would let us move from the present register, not to any register selected from the entire set, but only to a neighboring register. The number of "neighbors" must be fairly large, of course, or we shall be too likely to be trapped, to find that all the neighbors have already been pre-empted by other storage paths previously laid down. What we need is a simple way of defining neighbors that will yield a fairly large number of neighbors and yet facilitate coding their locations.

We may think of a memory facility as being organized in N dimensions. For the sake of simplicity, we may restrict ourselves to move in only the positive direction along only one dimension at a time. Then, if the present register has the location $i, j, k, \cdots$, its neighbors are the registers with locations $i + 1, j, k, \cdots ; i, j + 1, k, \cdots ; i, j, k + 1, \cdots$; etc. This is illustrated for a three-dimensional structure in Fig. 6. The number of neighbors is equal to the dimensionality of the structure, and the number of bits required to specify a relative location is only $\log_2 N$.

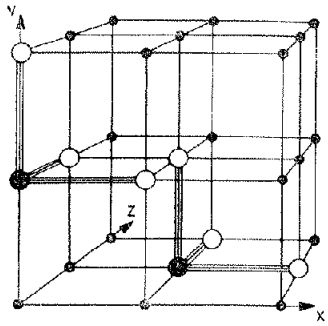

FIG. 6. Three-dimensional memory structure

In these terms, the unconstrained trie's discussed in previous sections were $(n - 1)$-dimensional trie's, n being the number of registers. When we refer to N-dimensional trie's, however, we shall imply that N is considerably smaller than $n - 1$.

Registers are represented by nodes. If one of the large solid nodes is the present register, the next register must be one of the large open nodes connected to it by heavy lines. When a new register is needed during the storing phase, we may examine those nodes, one at a time, in ordered sequence or in random sequence. As soon as we find an empty one, we use it, recording its relative location (dimension) at the large solid node. If all three (or, in general, all N) nodes have been pre-empted, we are blocked, and we must use some procedure outside the system.

In one sense, since we can make it any number we choose, N is independent of memory size. However, to make a useful memory, we must choose N in such a way as to keep low the probability that all N of the registers that are neighbors of a given register will have been pre-empted before that register is used. N must therefore be allowed to increase as some function of memory size.

Analysis of the Space Utilization Problem

The problem of space utilization can be approached in two steps. First, we can develop a formula that will structure the problem. Second, we can conduct a Monte Carlo simulation to test the formulas and to provide an empirical measure of the efficiency of utilization.

After many sequences have been stored in a trie, at what level during storage of a new sequence will we cease to find the characters already represented in the trie and have to store the location of a register hitherto not employed? Suppose, to make the problem specific, that we want to store S sequences of b binary digits each, drawn at random without replacement from the 2^b such sequences. What is the probability of having to shift, at a

given level, from reading and finding that the symbol is already stored to finding that it is not yet stored and storing it?

This probability is a function of the position i of the bit in the sequence, or the level i of the register in the trie. It is closely related to the function Y_i, the number of sequences that change, at level i, from merely following nexuses to projecting them.

Let us divide the set of already-stored sequences of length b into subsets 0 and 1, as their first digits are 0 and 1 respectively—and then into sub-subsets 00, 01, 10, and 11, as their first two digits are 00, 01, 10, and 11 respectively—and then into sub-sub-subsets 000, 001, $\cdots$, 111, $\cdots$ etc. If S is less than 2^b, we must come in due course upon empty subsets (i.e., sub- $\cdots$ -subsets) as we continue the progressive subdivision. Let the level at which the number of empty subsets becomes significant be q. (The value of q will be defined exactly by the intersection of slope lines, one for the lower part of the trie and the other for the upper part of the trie. This intersection will not in general fall at an integral "level" or bit position.)

We can state that as we develop the trie from level 1 to the vicinity of q, there is a high probability that two sequences will store first at level 1. One will be the first-encountered sequence of the set 0; the other will be the first-encountered sequence of the set 1. Two more sequences will (almost surely) start to store at level 2. These will be the first-encountered sequence of the subset 00 or the subset 01 (whichever did not begin to store at level 1) and the first-encountered sequence of the subset 10 or the subset 11. At level 3, the number will be 4. At level 4, it will be 8. And in general (except at level 1) the number Y_i will be, with high probability,

$$Y_i \cong 2^{i-1} \quad \text{for} \quad i < q. \tag{1}$$

That will be the case until we approach q. At levels just short of q, (2) will turn into a rough approximation. As we pass q, Y_i will diverge sharply from 2^{i-1}.

As we pass the level q and enter the upper part of the trie, we must focus attention on the number (k) of subsets of sequences that are not empty. The number k must be a decreasing function of i. The probability that a sequence will start to store at level $i + 1$ is the probability $p(i + 1)$ that it matches no previously stored sequence in digits 1 through $i + 1$. Since there must be many empty subsets when $i > q$, $p(i + 1) \cong p(i)/2$. Therefore k_{i+1} must approximate $k_i/2$, and

$$Y_i \cong 2^{2m-i} \quad \text{for} \quad i > q, \tag{2}$$

in which m is the integer nearest q.

According to the reasoning just outlined, the number Y_i of sequences that start to store at level i increases approximately along a line y_i of sequences that start to store at level $i = q$, and then, beyond that vicinity, falls off approximately along the line $y_i = 2^{2m-i}$. If the approximations are good, we can estimate how many sequences of length b may be stored in a trie of R registers and thus determine the efficiency of space utilization. Since Y_i is the number of sequences that start to store at level i, the sum of Y_i over i is the number S of sequences in the trie.

$$S = \sum_{i=1}^{b} Y_i$$

$$S = Y_1 + \sum_{i=2}^{m} Y_i + \sum_{i=m+1}^{b} Y_i$$

$$S = 2 + \sum_{i=2}^{m} 2^{i-1} + \sum_{i=m+1}^{b} 2^{2m-i}$$

$$S = 2^m + (2^m - 1) - 2^{2m-b}$$

$$S \cong 2^{m+1} - 2^{2m-b} \tag{3}$$

If $2m < b$, then the second term is unimportant and $S \cong 2^{m+1}$.

How many registers were required to store the S sequences? Each sequence that started to store at level i required $b - i$ new registers. The number (R) of registers is therefore

$$R = \sum_{i=1}^{b} (Y_i)\,(b - i) \tag{4}$$

$$= 2(b - 1) + \sum_{i=2}^{m} (2^{i-1})\,(b - i) + \sum_{i=m+1}^{b} (2^{2m-i})\,(b - i).$$

Since each register had to have space for two addresses, and each address required $\log_2 R$ bits, the overall requirement for storing S sequences each of length b bits, or $S \cdot b$ bits, was $B = 2R \cdot \log_2 R$ bits. The efficiency E of utilization was therefore

$$E = S \cdot b/B - (2^{m+1}b)/(2R \cdot \log_2 R). \tag{5}$$

Simulation Study of Space Utilization in *N*-Dimensional Trie Memories

In order to test the accuracy of the general trie formulas developed in the preceding section and to determine the characteristics of N-dimensional trie's, a Monte Carlo simulation study was carried out.

The test program was written in FORTRAN for the IBM 709. The program is composed of 42 subroutines, of which 19 were coded specially for this program and 23 were taken from the library. The program is capable of simulating trie's with dimensionality as high as 16.

All trie's had approximately the same number of registers, but the number of dimensions was varied from one trie to the next, the number of dimensions running from 4 to 16. The items stored in various "runs" were 16-, 30-, and 36-digit binary numbers, generated by a random number generator. Each run of the simulated trie memory was continued until trapping occurred, i.e., until the storing operation encountered a register that had no unfilled neighbors. The fraction of the space filled at the end of each test run was determined. That fraction increased with the number of dimensions. In the case of

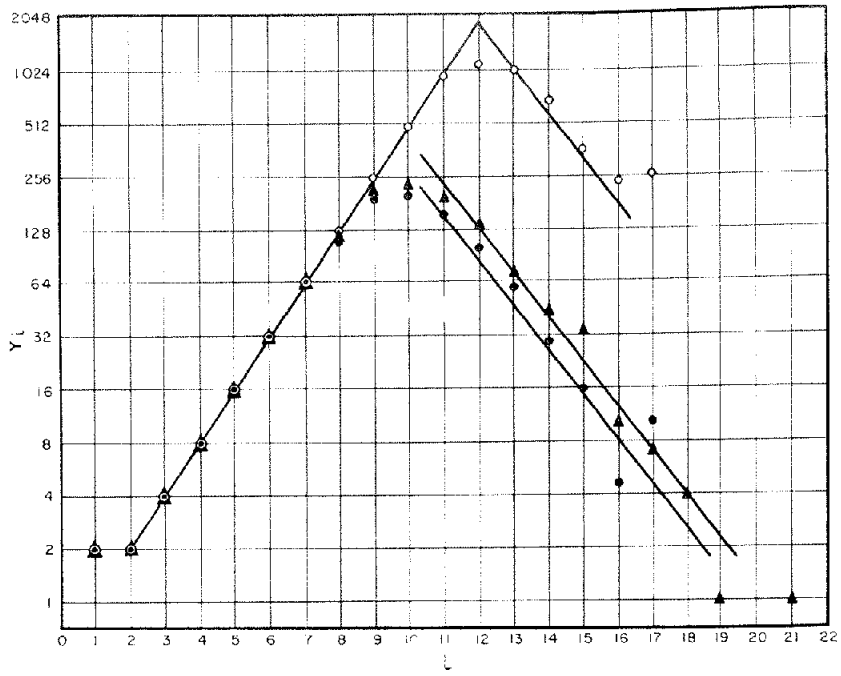

FIG. 7A. Experimental determination of the parameters q and m

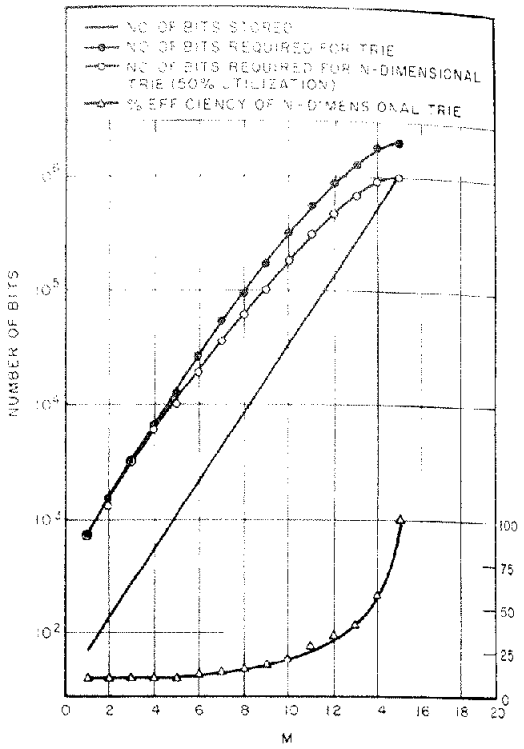

FIG. 7B. Relative efficiency in storing random 16-bit word

the 16-dimensional tries with 32-bit words, it exceeded one half.

Because of the limitation of computer memory, we did not attempt to simulate trie's with more than 16 dimensions. However, extrapolation from the data suggests that trie's with more than 16 dimensions will be even more efficient: more than half the total space will be usable. The percentage of the space utilized also appears to increase with the number of digits in the sequences stored.

Three different schemes for selecting the "next adjacent register" were tested. Selecting the next adjacent register is the same thing as selecting the dimension along which to move next, movement being restricted to a single step in the positive direction along any one dimension. One of the schemes was to pick neighboring registers at random. Another was to pick them in order. The third, slightly more complicated but also algorithmic, involved directed search, a search guided by simulated repelling forces from the locations of previously used registers. The random scheme was clearly far better than either of the other two.

The simulation test verified the analytical model of the behavior of trie memories and indicated that with memories of high dimensionability and long argument sequences at least one half of the total memory facility can be used to store information.

The results of three runs of 709 simulation program are shown in Fig. 7A to illustrate the determination of q and of m. The i-axis is the level, and Y_i is the number of sequences that change at level i from merely following nexuses to projecting them. The parameter q is equal to the value of i at the point of intersection of the two slopes. For the curve plotted in triangles this would be about 9.9. Here m is equal to the integer nearest to that value In the example, m for the triangles is 10.

In Fig. 7B the straight line is an evaluation of $S \cdot b = 2^{m+1}b$. In this case $b = 16$, and $S \cdot b$ is therefore the tota number of bits in the words stored. The curves provide a evaluation of the models developed in the previous sectio They show the increase in efficiency realizable as th amount of information stored increases.

Thirty runs of the 709 simulation program were mad to derive Fig. 8. The number of registers in the space wa maintained at approximately 60,000 while the number dimensions was varied. The points plotted are the pe centages of the registers in the space that were in the tr when the first trapping occurred. One can conclude tha in N-dimensional trie's the percentage of registers tha may be used prior to trapping is an increasing function the number of dimensions of the space (obviously reachin 100 when the number of dimensions equals the number registers) and that the percentage is an increasing functio also of the number of bits per word.

Thus, N-dimensional trie memories are efficient if th trie's contain many dimensions and many registers. Fo them to be highly efficient, the stored words must be lon Trie memory of small dimensionality may be useful, course, if—as may well be the case—the N-dimension addressing scheme can be implemented at lower cost tha conventional addressing schemes.

Storage of Non-binary Functions in Trie Memories

Thus far, we have considered arguments of arbitrar lengths, but we have restricted attention to the simple of functions: whether or not the argument is a member

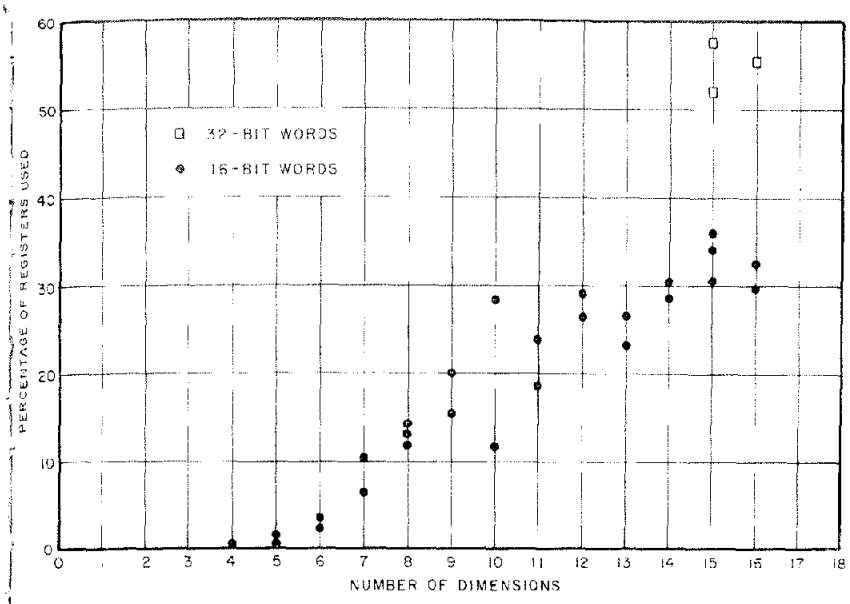

FIG. 8. Results of Monte Carlo simulation tests

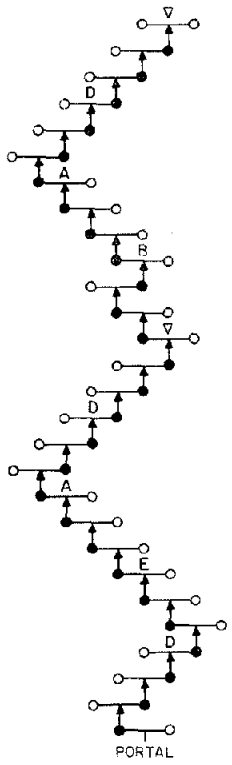

FIG. 9. Storage of a function-argument pair in a character-coded binary tree.

the stored set. Let us examine, now, the problem of storing other arguments, focusing attention upon binary tries.

The key concept is the use of the end-of-argument marker to effect transition between argument and function. For the sake of simplicity, we may waste the right-hand cell of each end-of-argument-marking register and use only the left-hand cell. In the left-hand cell we store the location of the register in which the first character of the function will begin. This procedure is illustrated in Fig. 9, in which the word BAD is stored as the function of DEAD. The procedure for the function is entirely similar except that we proceed beyond the end-of-argument mark (∇ = 111) instead of returning to the portal. The character code is the same as the one given in the legend of Fig. 5.

For the storage of the function, the same trie-building procedure can be employed as was used for the argument, except for the difference in starting points. Alternatively, one of the conventional storage plans can be used for the function. If each argument has only one function, and the functions are of fixed length, there is no reason to continue using trie memory as we store the functions. However, if the functions are of various lengths, or of lengths not known in advance, then either trie memory or list structure is likely to be advantageous.

The ideas just mentioned apply to storage of functions in N-dimensional trie's as well as to storage of functions in tries of the type represented in Figs. 5 and 9. In the case of N-dimensional trie's a special possibility arises. If the dimensionality is not too high, it is possible to find our way back down the trie by searching through the set of possible reverse paths until a register is found that contains a location just one less than the location contained in the present register.

Capability of Trie Memory to Remove Redundancies from the Information Stored

Redundancy is evident in sequential arguments and functions whenever particular n-grams occur with frequencies higher than would be expected if the sequences were random. Such redundancy is pronounced in English words and phrases. Measurements made by Shannon [4] and others [3] suggest that the relative redundancy of printed English is approximately 0.75. Computer programs appear to be not less redundant.

Trie memory eliminates all initial n-gram redundancy in the process of storing arguments. Any sequence that has an initial part in common with any other sequence already stored does not require new storage space for the common part.

However, it would further increase the storage efficiency of trie memories if we could eliminate duplication of sequences of letters located elsewhere than at the beginnings of arguments. Consider, for example, the group of letters "ing" that is so frequently encountered in English: wing, ring, wringing, etc. Can we not replace all occurrences of each such sequence of letters by a compact indicator?

One way of doing this is to replace each occurrence of such an n-gram by a nexus to a "model" n-gram, one occurrence of which would suffice for the storage of all occurrences of the corresponding sequences in input material. The model would start at the portal as do all words. We may point unambiguously to the model by pointing to its end-of-word marker. Therefore, we may

"abstract" an n-gram by substituting for it a register with three items:

1. An indicator bit, called the "abstract" bit, which would distinguish ordinary registers from those that represent references to "abstracted" models.
2. A nexus to the end-of-word marker of the model.
3. A nexus to the continuation of the word represented (the word from which the part was abstracted).

Let us assume, by way of example, that the trigram "ing" is stored as a separate sequence in a trie—a sequence beginning at the portal and ending with its own end-of-argument marker. When we have to store a word containing "ing", we may find the model by entering the portal and reading the three letters in succession. We can then abstract "ing" by replacing its occurrence in this word by a single register. In this register we set the "abstract" bit to one, point one nexus to the end-of-word marker of the model "ing", and point the other nexus to the continuation of the word—the part of the word that followed the "ing".

To read a word of which a part has been removed and replaced by a model, we read in the normal fashion until we reach a register identified by the "abstract" bit. We then re-enter the trie at the portal and read up the trie in the normal fashion until we reach the end-of-argument marker pointed at by the first nexus. We then return to the register with the abstract bit, and follow the second nexus in order to proceed with the reading of the word.

The contribution to storage efficiency of thus "abstracting" sequences of characters that occur frequently and representing them by models depends upon several factors. Even if a trigram occurs only twice in a trie, the abstracting procedure will save some storage space. However, the advantage of abstracting is bound to be an inverse function of the probability that a branch will be formed, during the storage of some future material, in the middle of the n-gram. If that probability is high, then it is best to divide the n-gram into subsequences with low branch probabilities and to store the subsequences as models. Although an explicit procedure for abstracting is easy to formulate for unconstrained trie's, some difficulties arise in N-dimensional trie's. The ends of the model and of the remainder of the word from which the modeled sequence was abstracted—the ends that must be connected together—are not likely to be neighbors in an N-dimensional trie.

Advantages of Trie Memory

The main advantages of trie memory that have been discussed are

(1) facility in handling information sequences of diverse lengths,

(2) ease with which addition and deletion can be accomplished,

(3) speed of storage and access (provided that the basic trie operations are available as instructions),

(4) elimination of n-gram redundancies, and

(5) inherent symbolic addressing.

The last-mentioned advantage is particularly interesting because it suggests a way of cutting through much of the complication presently involved in using assembly and compiling programs. In a computer designed around the trie memory principle, one would have only to name the procedure and the data to which it should be applied. The name of the procedure would serve as the argument that would retrieve the main "section headings" of the description.

The name of the data would call forth the parameters of the data. The section subheadings and data parameters would then retrieve procedure subheadings, sub-subheadings, etc. In due course, the entire procedure program would appear in working sequence. It would then call forth elements or sections of the data by name, and the computer would perform the indicated operations upon the data. Subscripts and subscripted subscripts would be handled as parts of names, or as names of names. The retrieval of variables would require neither calculation of subscripts nor information concerning the sizes of the arrays. This scheme, which goes considerably beyond the concept of trie memory itself, is being developed under the name "trie memory computer."

In addition to the listed advantages, there are others to be mentioned briefly: In searching files stored in trie memory, the objective may be not simply to find the function of a given argument (which may not be in the file at all) but to find the functions most nearly relevant. If the argument is ordered in such a way that the most basic information comes first, then it is necessary only to climb to the top of the argument, retrieve the function if it is stored, then retrace the path down the trie, retrieving any other functions encountered and exploring upward along neighboring branches. This concept, which is being developed under the name "associative memory," of course depends heavily upon working out a system for ordering arguments and criteria for determining the relevance of neighboring branches. It is not essential to have perfect ordering of arguments in an associative memory. Since every function can serve as an argument, and since trie memory can store functions of functions . . . of arguments, information sequences can be switched around like freight cars, and trains of sequences can be made up in one supersequence after another. This opens a way to the use, in digital computers, of search plans equivalent to those employed in cross-filing systems. In the trie memory scheme, however, the information is filed only once not in multiple.

Finally, trie memory provides a natural way of achieving in computers something similar to "apperceptive set" in human memory. In very large storage systems, in frequently used material must be stored away in slow access parts of the over-all memory. If the system is organized on a hierarchical trie plan, details being repre

sented by functions of functions . . . of the key argument, then the whole domain of a key argument can be alerted by an operation involving almost nothing but reading. As soon as this alerting moves beyond the boundaries of the fastest part of the storage system, it can proceed on its own, as it were, without holding up operation of the main computer, and it can initiate transfers of probably relevant information from slow-access to faster-access units. An inverse process is required to send the information back to limbo after the time of its probable use has passed. A plan for a memory system of this type is being developed under the name "apperceptive memory."

Summary

Trie memory is a paradigm, or set of related paradigms, for organization of information storage and retrieval systems, including computer memories. It facilitates handling of sequences of diverse lengths and makes it easy to add or delete information. It offers high speed in both storage and access. It provides inherent symbolic addressing and indefinitely deferred addressing. The main question concerning the feasibility of trie memories is the question of efficiency in utilizing storage space. The results of analysis and simulation indicate that this efficiency will exceed 0.5 in large trie memories if they are organized on the multidimensional plan described.

APPENDIX I

Storage of Functions in Purely Binary Tries

Storage of functions in purely binary trie's is less simple than it is in character-coded binary trie's. In purely binary trie's, one cannot assign one member of the character ensemble to the role of end-of-argument marker and still retain any capacity to store arguments. One could associate an extra binary digit with each binary storage register and let the value 1 indicate end of argument or end of function. That, however, would be very inefficient. The following scheme is slightly cumbersome, but better from the point of view of storage efficiency.

To mark the end of each argument, put the same address in both cells of the register that is specified in the last cell of the argument. Since no argument or function register will ever have that configuration, it is an unambiguous designator of "end-of-argument." The location to write into both cells of the special register is the next one in the location list. It will specify a second special register. Now, write into the left-hand cell of this second special register the location of the register in which the representation of the function will begin. That leaves the right-hand side for continuation of the argument sequence if an argument appears that is longer than the one just stored but identical with the one just stored throughout its length.

If the argument under consideration itself required no writing (if it turned out to be identical with the first part of some longer argument, already stored), it is necessary to use a detour procedure. Note and hold for re-use the locations previously entered in the relevant cell of the last register of the present argument. Replace it by a new location, the location of a "first-special-register." Write the same "second-special-register" location into both cells of the first-special-register. Read the location of the next register in the previously stored argument from temporary storage and write it into the right-hand cell of the second-special-register. Then write the location of the first new function register into the left-hand side of the second-special-register.

The procedure just described requires revision of the reading procedure, and it calls for two extra registers per argument. For long arguments, however, the relative increments in space and time required are small.

APPENDIX II

Storage and Access Speeds of Trie Memories

The lengths of time required to store and retrieve functions from a trie memory can be calculated from specifications of the material stored, the trie structure, and the parameters of the computer.

Experience obtained in the simulation tests described in the body of the report indicates that trie memory is competitive with conventional storage plans even when the trie operations have to be programmed. The inherent potentiality of trie memory for rapid storage and access can be realized, however, with only a moderate amount of circuitry. For truly effective application, a "store-in-trie" instruction and a "read-from-trie" instruction are required. If those instructions were available in an IBM 709, for example, the storage and access times would be approximately 432 microseconds per 36-bit word, and the corresponding speeds would be 80,000 bits per second. In general, one may move one level every memory reference cycle.

REFERENCES

1. R. DE LA BRIANDAIS, File searching using variable length keys. *Proceedings, Western Joint Computer Conference, 1959*, pp. 295–298.
2. E. FREDKIN, Trie memory. Informal Memorandum, Bolt Beranek and Newman Inc., Cambridge, Mass., 23 January 1959.
3. J. C. R. LICKLIDER AND N. BURTON, Long range constraints in the statistical structure of printed English. *Am. J. Psychol. 68* (1955), 650–653.
4. C. E. SHANNON, Prediction and entropy in printed English. *Bell System Tech. J. 30* (1951), 50–64.
5. J. C. SHAW, A. NEWELL, AND H. A. SIMON, A command structure for complex information processing. *Proceedings, Western Joint Computer Conference, 1958*, pp. 119–128.