File Searching Using Variable Length Keys

RENE DE LA BRIANDAIS†

MANY computer applications require the storage of large amounts of information within the computer's memory where it will be readily available for reference and updating. Quite commonly, more storage space is required than is available in the computer's high-speed working memory. It is, therefore, a common practice to equip computers with magnetic tapes, disks, or drums, or a combination of these to provide additional storage. This additional storage is always slower in operation than the computer's working memory and therefore care must be taken when using it to avoid excessive operating time.

This paper discusses techniques for use in locating records stored within a low-speed memory medium where they are identifiable by a key word or words of variable length on a machine not equipped to accomplish this automatically. The technique is also applicable to the conversion of variable word length information into fixed length code words.

When records can be stored in a slower memory medium in such a fashion that their exact location may be determined from the nature of their designation, reasonably efficient handling procedures can be established. However, as is often the case, the records cannot be so easily located and it becomes necessary to examine each entry in order to locate a particular record. Sequential examination of the key words of each record, until the desired record is located, is not a satisfactory approach on machines not having automatic buffered searching facilities, and may not be satisfactory on machines so equipped, if, for instance, reels are searched which need not be because it is not known in advance that they are not needed. Because the average search time for the desired records is proportional to the number of records stored in this slower memory, the total operating time of a program is proportional to the product of the number of records stored and the number of records for which search is instigated. This product may approach the square of the number of records involved. This relationship between operating time and the number of records stored places a definite limitation on the number of records which may reasonably be stored by any particular program. Fortunately, if the records can be stored with the key words in some ordered arrangement, an educated guess can then be made as to the location of a particular record, and a better system will result. However, records cannot always be arranged in such a fashion.

When records are large compared to the key word or words, a useful technique is to form an index having in it just the key words and the location on the corresponding record. A particular record is then located by searching the index to determine the record's location and then taking the most rapid approach in arriving at the record. Since only the key words and the locations of the corresponding records are stored in the index this technique reduces the amount of information which must be handled during a search. With the smaller amount of information involved it is often possible to utilize the computer's high-speed memory for the storage and searching of the index. Furthermore, this index can now be ordered or otherwise subjected to speed-up techniques. This index approach often can greatly improve the operating efficiency of record handling programs. In many instances this improvement is sufficient but there are also many cases where a further increase in efficiency is necessary. In particular, the time required to perform the search when consulting the index may still be objectionably large. If this is true then it is necessary to apply a speed-up technique to the searching operation. Of course these techniques can be applied to any table lookup problem where the nature of the key word or words does not lead directly to the desired entry.

Peterson[1] has suggested a method of arranging such an index which greatly reduces the lookup time when it can be applied. This method, referred to as the "bucket method," calls for randomizing the digits of the key word to produce a number which indicates that point in the available memory where a particular key and its corresponding record location should be stored. If this particular space is not available it is stored in the next highest (or lowest) available space. When seeking a particular record, the exact randomization process is repeated producing the same indicated point and a search is begun from that point in memory and in the previously used direction. When using this method, facility must be provided to continue from the other end when one limit of the available space is reached. During the process of placing an entry in this table a record is kept of the number of steps which must be taken before finding space to store the entry. This number is then compared with and, if necessary, replaces the previously occurring maximum. This maximum can then be used to limit the operation when a search is undertaken for an item for which there is no entry.

The object of this procedure is to distribute the records evenly throughout the available space in spite of uneven characteristics which occur because of similarities in the structure of the keys. A limitation of this

† U. S. Naval Ordnance Lab., Corona, Calif.

[1] W. W. Peterson, "Addressing for random-access storage," *IBM J. Res. Dev.*, vol. 1, pp. 130–146; April, 1957.

method is that it is necessary to store the key as a part of the entry in order to identify an item positively during the searching phase. This increases the storage space used and, as the memory fills up, the average number of spaces which must be examined before an entry is located increases. This saturation effect greatly decreases the operating efficiency. Furthermore, this method becomes far too involved from the bookkeeping standpoint when the keys are variable word-length words which exceed the length of one computer word when working with a fixed word-length machine.

A problem involving variable word-length information confronted us in writing a FORTRAN type compiler for the Datatron 205. In this version of FORTRAN we allow the names of quantities to be of any length and they may consist of any number of separate words provided there are no intervening special characters. For reasons of simplicity in the internal handling, each of these external names must be converted to a code word of specific length such that it may be stored with other information within one cell. Further, it is necessary that the external name be preserved and made easily available for annotating the finished program. To accomplish this, each external word is placed in a file which, for future reference, is written on tape as it is formed. The position of each entry in this file is then used as the code word for the name. Knowing the position of a particular external word makes it very simple to recover for annotation purposes.

When it is necessary to have some method of determining whether any given word has occurred previously, sequentially scanning the previous entries is impractical because of the limitations mentioned previously. A more desirable situation would be a scheme that in no way depended upon the amount of previously stored information in the file. If this could be accomplished, the operating time would then be more nearly proportional to the number of items for which a search was performed rather than the product of this number and the total number of entries in the file.

The technique we have developed accomplishes the goal. The operating time is related to each letter of the external word and is, therefore, proportional to the number of letters in each word for which a search is performed. The size of the file has little effect on the operating time. Total operating time is proportional to the number of external words multiplied by the time required for a word of average length.

In our particular application each external name becomes a record which is stored on tape. The location of the first word of each record is the code for that external word. The external word itself is the key. In the computer's main (drum) memory we form an index of the key and its corresponding code. The organization of this index is the reason for this method's efficiency. We call this the "letter tables" method. The index consists of a set of tables with the number of tables as well as the number of entries in each table varying with each running of the program. An address, usually the lowest numbered available cell, is assigned as the starting point of the table of first letters. The key words are examined letter by letter and each first letter which occurs is entered in the table of first letters if such an

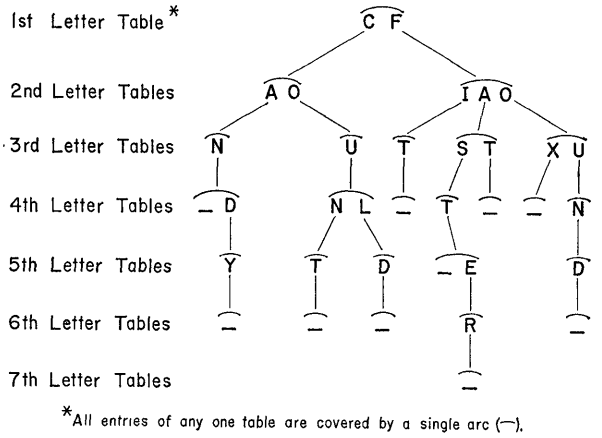


Fig. 1—Formation of a set of tables.

entry does not already exist (Fig. 1). A new table is assigned to each of these letters. It is formed by assigning it a starting address. Each second letter which occurs is entered in the table assigned to the letter which it follows. To each of these second letter entries a new table is assigned as before. Each third letter is placed in the table assigned to the letter pair which it follows. Thus, there will be a table for the letters which follow "CA" and a different table for those letters which follow "CB" if these combinations occur. To each third letter a table is assigned and the technique is repeated until all letters of the key have been taken care of.

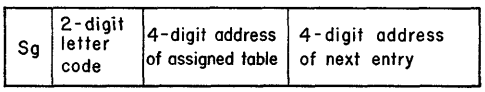
In our program the words which make up a key are compressed to eliminate blank spaces before being placed in the table as one word. A blank space is then used to signify the end of the word. This blank space is stored in the set of tables as a signal that the entry is complete and the code word which was previously determined is stored with this blank instead of an indication of a table assigned for the next letter. If preservation of the blanks is necessary, a special unique mark may be used to signal the end of a key.

When attempting to determine the previous occurrence of a particular word the procedure is first to scan the table of first letters until the desired letter is found. The second letter is then compared with the various entries in the assigned table until agreement is found for the second letter. The third letter is compared in a similar manner with the indicated table and the process is continued until comparison occurs with a blank. When this occurs the desired code may be found in the remainder of the entry with that blank. In the event that the desired letter does not occur in a particular list, it is known that this word is not in the index. If this word is to be added, it is now necessary to store the remaining

letters of the word in the index using the previously described technique. The first letter to be added will be the one which did not occur. Once a letter has been added to the tables there are no entries in the newly formed table so no further searching is necessary, and it is only necessary to add each letter remaining in the word to the new tables.

As previously mentioned, the code is stored with the blank which signifies the end of the word. This code is the next available location for a record in the external language file. As soon as the index is complete the external word, which is this next record, is placed in the file. The location indicator is adjusted to indicate the next available space in this file and this determines what the next code word will be.

Since the amount of space required for any of the tables in this type of operation depends upon the manner in which the letters happen to follow each other, it becomes necessary to assign space to each of the tables as it is needed. This is best done by assigning the next available space to whichever table is being expanded. The programming principles involved in this type of operation were first described by Newell and Shaw.[2] However, since in our application it is not necessary to remove entries from the tables as was the case in their application, a less involved method than the one they described can be applied. If a continuous portion of memory can be devoted to the storage of the tables it can be utilized in a sequential fashion with the next available space being the next cell. A simple counter can then be used to keep track of this next available space. This operation results in the storage of the various tables in an overlapping fashion and, therefore, it is necessary that each entry in a table have an indication of the location of the next entry in that table.



(a)

(b)

Fig. 2—(a) Format of a letter entry. (b) Format of a word-terminating entry.

Fig. 2 shows the two types of entries in these tables. The letter entry shown in Fig. 2(a) has the letter in the two digits on the left end of the word, and the four digits on the right end are used for the address of the next entry in this table. The remaining four digits specify the address of the first word in the assigned table.

[2] A. Newell and J. C. Shaw, "Programming the logic theory machine," *Proc. WJCC*, pp. 230–240; February, 1957.

Fig. 2(b) shows the configuration used for the storage of a blank which signifies the end of a word. In this word, the first two digits are blank, the next four digits are the code, and as in the previous case, the last four digits indicate the address of the next entry. The end of a particular list is signified by the absence of an address indicating the next entry.

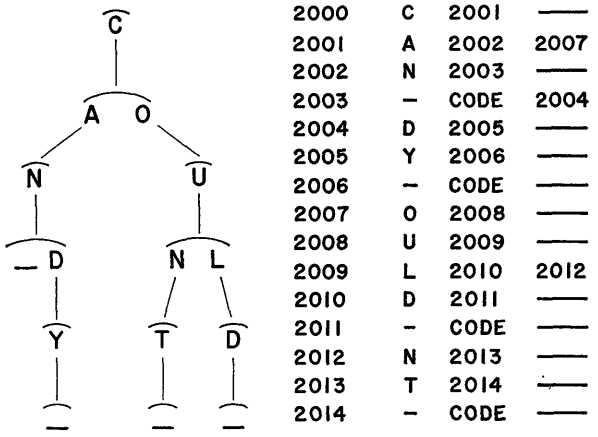


Fig. 3—Memory distribution of a set of tables.

Fig. 3 shows a memory distribution which would result from the storage of the four words: can, candy, count, and could. In this example there are a total of twelve separate tables stored in the fifteen spaces. Although the system may appear more complicated, no more bookkeeping is required than in a system of sequentially searching a list of entries where the different entries require different numbers of words in memory for their storage.

An additional time-saving feature that can be applied with a slight additional cost of memory space is the establishment of a full set of possible first letters with the corresponding second letter tables assigned starting points. By this we mean that if the first letter is "E" the first entry in the assigned second letter table will be in the fifth cell with respect to the beginning of the tables. Those first letters which do not occur are then wasting one memory word each.

Since in this scheme there is only one letter stored in each word, it requires far more space than other schemes where several letters are stored in one word. However, the advantage in scanning speed makes up for this disadvantage and it becomes practical to form several tables of this type, storing them in a slower memory medium until needed. When doing this, difficulty can arise if care is not taken to avoid excessive transfers to and from this slower memory. Methods of overcoming this problem depend upon the particular application. In our application each set of tables is no longer needed after one complete pass of the input, and when overflow of space occurs, during input, those words which cannot be converted are marked. After completion of the first

input pass the set of letter tables is erased and a second input pass begins at that point in the input data where overflow occurred. A new set of tables is formed to convert the marked words and the process can be repeated as often as necessary.

Now let us look more closely at the technique in an effort to determine the operating time. For the sake of the following discussion, we shall assume each character to be one of 40 possibilities. This list of possibilities could include the 26 letters of the alphabet, ten decimal digits, a blank, and three special characters. The maximum number of comparisons necessary to determine a word then comes to 40 for each character in the word including the blank which terminates the word. We find however that this maximum is seldom reached. To show this, let us assume a file contains 1000 words. If the first characters of these words are evenly distributed among the 40 possibilities there will be approximately 25 words starting with each character. Since 25 words can provide only five-eighths of the possible entries in the second letter tables we can expect that to determine a word, the average number of comparisons needed will be 20 to determine the first letter, 13 to determine the second letter, and thereafter only one per letter. Thus a nine-letter word including the blank might require an average of approximately 40 comparisons.

Now we increase the number of words to 10,000 and we find that we have an average of 250 words per character in the first letter table, $62\frac{1}{2}$ per character in the second letter tables, and approximately one and one-half in the third letter tables. The average number of comparisons for a nine-letter word now comes to 20 for each of the first three letters and one for each of the remaining letters. This new total of 66 is 1.65 times greater than the previous average.

When the words stored in such a fashion are taken from some formal language such as English, the number of words beginning with certain characters tends to increase and, therefore, the possibility of having all of the various characters occur in the table of first letters is decreased. This decreases the average number of comparisons needed to determine the first letter. Furthermore, the number of letters which normally might follow a particular letter is limited so that the average number of comparisons is reduced for subsequent letters also.

For example we normally expect "U" to follow "Q" and one of the vowels or the letters "H," "L," "R," or "Y" to follow the letter "C." Thus we find we are able to adjust favorably the averages we determined previously due to bunching, a phenomenon which usually leads to decreased efficiency in other methods. In the instance of the 10,000-word file we might expect the averages to be more like 16, 12, 8, 3, and one thereafter which would be 44 for the nine-letter word, an improvement of one-third.

We shall now attempt to compare this technique with Peterson's "bucket method." The bunching which we described as useful to us must be overcome when using the "bucket method." This is usually done by generating a number which is influenced by all characters of the word and yet appears to be random with respect to them. This is extremely difficult when dealing with variable word length information, especially with long words where only one letter differs or where the letters are the same but two have been interchanged. Other difficulties encountered with the "bucket method" include terminating the search when enough entries have been examined to know that the word is not in the file and then finding suitable space to insert the word. The resultant bookkeeping can actually consume many times more operating time than the actual comparison operation requires. Thus, although fewer comparisons may be required when using the "bucket method," due to the variable word length problems, the operating time is pushed up into the same range as that of the "letter tables method" which we have described.

Another way in which the two methods must be compared is with regard to the amount of memory required for the storage of similar amounts of information. In a fixed word length machine up to all but one character might be wasted with each word stored when using the "bucket method." Also, when a minimum of two adjacent cells are used, one for the word and one for the code, an occasional word will be lost due to storage of a word requiring an odd number of cells in such a position as to leave only one unused cell between itself and an adjacent entry. The amount of memory space required for the storage of a particular amount of information in the "letter tables method" cannot be specifically determined because it is quite dependent upon the number of repetitions of letter sequences which occur. The number of cells required will always be greater than the number of words and may in some instances approach the total number of characters stored. This means that the "letter tables method" will probably require from two to six times as much memory space as the "bucket method" for a similar amount of information.

The amount of code required by the "bucket method" may run three to five times as much as for the "letter tables method" depending on the application. Also, this latter method could quite probably be written as a subroutine or by a generator in a compiling routine with much greater ease than could the "bucket method." The final choice of method depends on details of the specific problem and also on operating characteristics of the machine on which it is to be run. It may in some cases be necessary to program and run tests before a final determination can be made. Both methods are an order of magnitude faster than the simple sequential search and we have found them both to be of value in different parts of the FORTRAN for Datatron Project.