Table look-up (with examples in COBOL)

A. D. Wilkie

Standard Life Assurance Company, 3 George Street, Edinburgh EH2 2XZ

Some straightforward methods of table look-up are described, suitable for routine use with moderately small tables in practical applications. Particular attention is paid to the different circumstances in which different methods are applicable, to the methods of constructing, coding, loading and updating of the table to be searched, and to the action to be taken when a suitably matching entry is not found.

(Received August 1975)

In an article 'Professional programmers?' in *Computer Bulletin* (Series 2, Number 3, March 1975, page 15), Professor D. W. Barron referred to

> 'an observation on table look-up that was made in the course of an article about APL in *Computing*. The author remarked that finding an item in a table was a frequent requirement in commercial data processing, and was usually 'attempted by a loop search through the table, testing each item for equality (or less usually, but equally laborious for the programmer, by a binary chop technique)'. Admittedly the binary chop (logarithmic search) is more complicated to program—perhaps eight lines of code instead of four—but as any despised computer science graduate knows, it is immensely superior in operation, being on average fifty times faster on a table of 1,000 entries. (Of course, a hash table technique can be even better, but let us not follow up that line.) One of the characteristics of a professional in any field should surely be a pride in doing a competent, efficient and workmanlike job, making best use of the available tools. Using linear search on anything but a trivial table is none of these.'

These remarks made me realise that there was a shortage of material on the practical aspects of table look-up, which my own experience has shown to be a substantially more varied problem than either Professor Barron or the author in *Computing* appears to recognise. The theoretical position is well discussed by Knuth (1973) and by Brooks and Iverson (1969), but they do not fully consider the practical convenience or applicability of the various methods. This article is a first attempt to fill that gap, but it cannot at this stage attempt to be comprehensive.

Table look-up is indeed frequently required in commercial data processing—or in fact in almost any form of computing. Input data fields may contain codes which must be checked for validity and which may determine subsequent processing, grouping for totals, etc. Operating systems too must look up names of files, modules, etc. in libraries; compilers must look up variable names in tables that are constructed as the program proceeds, and look up reserved words in tables that are established before the program begins. Random access systems must use an identifying number or name—customer name or number, policyholder, agent, part number, stock number—to look up a file and retrieve the appropriate record. There are many uses of table look-up and the appropriate solution (or solutions, for there may well be several almost equally good ways of doing a job) depends on the varied circumstances of the particular application.

The best solution is one which, as far as one's knowledge permits, optimises some weighted combination of factors of which at least the following are important:

(*a*) machine efficiency:	run time speed run time program size compiling efficiency
(*b*) programmer efficiency	ease and speed of writing first-time accuracy of program
(*c*) maintenance:	ease of changing the table.

In my view the last of these is often the most important; commercial systems do not remain static, and must respond quickly to changes in the practice of the organisation. Of course, changes may not be a problem in certain circumstances, as will be shown below. Programmer efficiency, to my mind, is next in importance. Getting programs right first time is vital to the rapid completion of a system, and this can be helped considerably if programmers adopt a number of standard methods for the solution of their problems. The more methods they have at their fingertips the better, but they must be able to use them quickly and accurately. For programs that are required quickly speed of actual writing and punching is not unimportant, and this is one of the serious disadvantages of COBOL. Machine efficiency has become progressively less important, and unless the solution is obviously grossly inefficient (like sequential search through a table with 1,000 entries) run time speed and program size are definitely secondary to the convenience of programmers.

Let us now introduce some definitions for a simple table. A *table* consists of a number of *entries*, each consisting of a *key*, x, and a *result*, y. Each entry is identified also by an *index*, i, so that to each $x(i)$ there corresponds one $y(i)$. Each key $x(i)$ is unique, so that given an arbitrary *input* X it is possible, first, to find whether or not there is a corresponding key, $x(i)$, and secondly, to extract the corresponding result $y(i)$ which becomes the *output* Y. The results, of course, need not be unique. Complex tables can also be constructed which contain alternative keys, any of which may be used to obtain the rest of the entry as the result.

These definitions leave some loose ends that may vary in different circumstances. The key that corresponds to the input may be given by equality: $X = x(i)$; or may be determined by subranges in an ordered sequence of $x(i)$s: $x(i-1) < X \leqslant x(i)$, as for instance in the calculation of stamp duty on life assurance policies, stockbrokers' commission, or customers' discounts, where the method of calculation (rounding, rate, etc.) depends on the amount of the appropriate input. In the former case it is not essential that the keys be in any specific order in the table, but in the latter case it is clearly necessary for the keys to be ordered, so that: $x(i) < x(i+1) < \ldots$. In this case too there is the question of the open ends. Are all values of X valid, so that we can conveniently have an implied $x(0)$ = low values, and an actual $x(n)$ = high values? Or is the valid range of X restricted so that we have at some stage to treat $X < x$ (low) or $X > x$ (high) as errors?

Next one must consider the action to be taken if X is not found to be within the range of the table so that there is no corresponding $x(i)$. It may be required:

(*a*) to indicate simply 'invalid X' by, say, setting an error

Downloaded from https://academic.oup.com/comjnl/article/20/3/202/751653 by guest on 19 April 2024

marker, or performing an error routine

(*b*) to take a default action, by providing, say, an output y (default) appropriate to non-matching inputs

(*c*) to insert a new entry in the table with key x (new) $= X$ and a result y (new) that corresponds appropriately.

In the last case the table has to be dynamically altered during the run, and some quite different considerations start to apply that are not relevant to static tables.

1. The direct method

Now let us introduce a variety of methods of table look-up. A first and simple method is to incorporate the table *directly into the program*, e.g.

```
IF INPUT-SEX = 'M' THEN ADD 1 TO NO-MALES,
   ELSE
IF INPUT-SEX = 'F' THEN ADD 1 TO NO-FEMALES,
   ELSE
MOVE 'X' TO ERROR-MARK PERFORM
    ERROR-ROUTINE.
```

Here we see the keys 'M', 'F' as literal constants in the IF-conditions, the results incorporated as instructions following the THENs and the error action following the last ELSE. This method involves a sequential search through the respective IFs until equality is found, and so it is unsuitable for large tables. For very short tables it is probably the most efficient method on all counts. At run time it is faster than a sequential search using an indexed entry to the table, but of course it produces a larger run time program as soon as the number of entries gets above half a dozen or so. For tables with more than say 25 entries a more efficient method than direct sequential search might be wanted anyway.

Provided the IF and THEN can be restricted to a single line of code the direct method makes for easy writing, with ditto marks, and easy punching, by partial duplication from the previous card. If more than one line is required, careful duplicating and then interleaving of cards is possible, though less convenient. Alternatively, the THEN statement can give the value of an index which is used to extract the result from a separate table organised as an array.

Altering the direct table must be done by altering and recompiling the program, but an alteration can be easily effected by changing or inserting only one card or group of cards in the appropriate sequence. There is no need to keep a note of the table size as a constant elsewhere in the program.

The direct method is particularly advantageous if the key is of an irregular form—perhaps where correspondence is tested for partly by equality and partly by subrange of values of the input, or for example, where different results apply to males, married females and other females, and where sex and marital status are given in two separate input fields. This sort of flexibility is not infrequently required in practice.

2. Input of index

A second simple method of arranging table look-up is to make the input, X, *equal to the required index*, i.e. $x(i) = i$. After checking the input for range, so as to indicate an error for $X < i$ (low) or $X > i$ (high), the required result can be extracted directly from a table. If the valid range for i is not consecutive it may nevertheless be convenient to make an entry in the table of results for every value of i and include also a validity marker as one field in the result.

An obvious disadvantage of this method is that it restricts the form of the input. While it is convenient enough to code months as 01, 02, . . ., 12 instead of JAN, FEB, . . ., DEC—the result being the number of days in the month for instance—it is usually easier for clerical staff to remember mnemonic letters (e.g. S, M, W, D for single, married, widowed, divorced) rather than arbitrary numbers. Letters can of course be changed into numeric equivalents, but only by using machine dependent conversions and code.

For a table where both the key and the result can be contained in a single byte, extraction of the result can be performed by this method in IBM/360/370 Assembler by a single Translate instruction.

3. Sequential search

The simple conventional method is, as the author in *Computing* states, a *sequential search* through the table, testing each table key for equality with the input key.

```
A1. MOVE 1 TO I.
A2. IF I GREATER THAN TABLE-LENGTH MOVE 'X'
      TO ERROR-MARK GO TO A4.
    IF INPUT-X = KEY-X (I) MOVE RESULT-Y (I) TO
    OUTPUT-Y GO TO A3.
    ADD 1 TO I.
    GO TO A2
A3. Continue if found, with I and OUTPUT-Y given.
A4. Continue if not found.
```

This routine can be further simplified in other languages by the use of a DO-loop or FOR-loop; it may be necessary to be careful about whether the index used for incrementing through the loop is available or not on exit from that loop.

A neater method described by Knuth (1973, page 395) is to reserve a spare entry at the end of the table and to place the input key in that location initially:

```
A1. MOVE INPUT-X TO KEY-X (ILAST)
    MOVE 1 TO I.
A2. IF INPUT-X = KEY-X(I) GO TO A3.
    ADD 1 TO I.
    GO TO A2.
A3. IF I = ILAST GO TO A4.
    Continue if found . . .
A4. Not found . . . .
```

It may also be convenient to test for the end of the table by inserting in the last entry some key which it is impossible for the input to equal (not one which is invalid but possible); the searching routine then does not need to know the length of the table.

While the program for a sequential search is fairly trivial, the construction of the table in a COBOL data division is not (assuming that the table is to be included in the compiled program at all—the alternatives are discussed later). The table must be able to be accessed by index; the entries must be inserted by value clauses. The restrictions of COBOL require these to be inserted the right way round—values first, redefined by an occurs clause. The obvious way to start writing such a table is:

```
01 TABLE.
 03 TABLE-ENTRIES-BY-VALUE.
  05 ENTRY-1.
   07 KEY-1 PIC X(3) VALUE 'JAN'.
   07 RESULT-1 USAGE COMP PIC S9(2) VALUE +31.
  05 ENTRY-2.
   07 KEY-2 PIC X(3) VALUE 'FEB'.
   07 RESULT-2 USAGE COMP PIC S9(2) VALUE +28.
   . . .
   . . .
 03 TABLE-ENTRIES-BY-INDEX REDEFINES
          TABLE-ENTRIES-BY-VALUE.
  05 ENTRY OCCURS 12.
    10 KEY-X PIC X(3).
    10 RESULT-Y USAGE COMP PIC S9(2).
 03 TABLE-LENGTH USAGE COMP PIC S9(2)
          VALUE +12.
```

Downloaded from https://academic.oup.com/comjnl/article/20/3/202/751653 by guest on 19 April 2024

It can readily be seen that this is appallingly laborious for coding and punching. It is certainly worth while keeping the result in character form, even though more work is done by the program in converting to decimal or binary. We can then simplify the value part to:

```
05 ENTRY-1 PIC X(5) VALUE 'JAN31'.
05 ENTRY-2 PIC X(5) VALUE 'FEB28'.
...
...
```

and rewrite the appropriate line:

```
10 RESULT-Y PIC 9(2).
```

If the table is likely to remain unchanged—as perhaps in this example—the values can be run together into one or a few long literals:

```
    05 ENTRIES PIC X(60) VALUE
'JAN31FEB28 . . . DEC31'.
```

This, if it can be done, saves a great deal of tedious coding. But if the table requires moderately frequent updating, perhaps with the insertion of new entries, then at least one line for each entry is desirable, both for clarity and to reduce errors. The omission of a single character in a long string will put the whole table out of step—and produce no diagnostic message. If the values are arranged as a table on separate lines and in the same columns, then errors are easily spotted.

Note that in the example above the number of entries in the table was included as another variable as part of the whole 01 entry. If the table may ever change in length it is obviously convenient for the length to appear in the program as seldom as possible, and those cases to be near one another—in this case twice: 'OCCURS 12' and 'VALUE +12'. When a change is made there is then a better chance of both being altered together.

4. Improved sequential search

For both the direct search method and the sequential search through a table with N entries the average number of tests before finding equality is $N/2$, assuming all keys are equally frequent and there are no errors. Errors of course require N tests. If errors are infrequent, and keys are not evenly distributed, then two improvements are possible. First, the table (or the program tests in the direct method) can be *arranged in frequency order* so that the commonly occurring keys come first. So put a table of maritial status as: Married, Single, Widowed, Divorced, in that sequence. Put a table of life assurance policy classes so that the frequent endowment assurances come first and the rare multiple life cases come last. If there is sufficient concentration in the early entries then even quite a long table, say over 100 entries, can be satisfactorily dealt with sequentially.

A second possible improvement is worth while if the keys are likely to be bunched together in the input so that runs of similar cases are found together. In life assurance a bundle of assurances may be followed by a bundle of annuities. In other applications a bundle of orders of a similar type may come together. In these cases it may be worth while treating the table *dynamically*; after each entry is found it moves up the table one place, changing with the one previously just above it (unless it was already at the top). The process is similar to that of a squash 'ladder'. In this way the locally frequent entry makes its way to the first position in the table. Unless direct exchange is possible in the language, a spare location is needed to hold a complete table entry and the relevant bit of program is:

```
IF INPUT-X = KEY-X (I) THEN
   MOVE ENTRY (I) TO TEMP-ENTRY
   IF I = 1 GO TO FOUND
   ELSE MOVE ENTRY (I − 1) TO ENTRY (I)
          MOVE TEMP-ENTRY TO ENTRY (I − 1)
          GO TO FOUND.
```

Note that with this method the position of any entry in the table changes, so the value of the index, I, cannot be preserved for later re-entry to the table to obtain the same result. The whole result is extracted at once (and in this case placed in TEMP-ENTRY, even for I = 1), and if an index is needed at a later stage, for example for the accumulation of totals in a totals table, then the totals index must be an explicit part of the result.

Clearly the dynamic table method requires some additional program on each occasion that a successful match is found, so it is not worth while unless sufficient bunching does occur. But even with a table of months bunching may occur for any one run, since the input data may relate to transactions in the current or in neighbouring months. The current months rapidly move to the top of the table, and commonly the matching month will be the first entry in the table. However, if keys occur in some specifically cyclical pattern the dynamic method may be less satisfactory. Consider what happens when two keys occur alternately! If a field in the result is allocated to a count of occurrences then alternative promotion strategies are of course possible.

Yet a further variant really eliminates searching altogether. If the input keys are sorted initially into the same sequence as the table, the problem becomes the different one of matching one file against the other. The table key entries must be in sequence and each search begins at the point where the last finished. This method is, of course, restricted to one key per program run, and this is normally taken as the key requiring the largest 'table' to be looked-up—such as the master policy file or customer file itself.

Another variant of the method, particularly useful if the keys occur clearly in runs in sequence, is to test each key first against the entry for the previous key, which is left in a fixed, and unindexed, location such as TEMP-ENTRY. If a match is found, much time is saved; if no match is found, only a little time is lost. Note that the very first key may require special treatment—such as by initialising TEMP-ENTRY either with the value of any valid table entry or with a conspicuously invalid key that cannot match.

5. Binary search

When we get beyond a comparatively short table, the sequential search, even improved, becomes intolerably slow, and better methods must be used. The *binary search* method is simple to use, but it relies on the keys in the table being in sequence ($x(1) < x(2) < \ldots < x(N)$).

```
A1. MOVE 1 TO ILO.
    COMPUTE IHI = TABLE-LENGTH + 1.
A2. COMPUTE I = (ILO + IHI)/2.
    IF X = KEY-X (I) GO TO A3-FOUND.
    IF I = ILO GO TO A4-NOT-FOUND.
                 (Since ILO was the only possibility left).
    IF X LESS THAN KEY-X (I) MOVE I TO IHI,
    ELSE MOVE I TO ILO. (Since X > KEY-X (I)).
    GO TO A2.
A3-FOUND. . . .
A4-NOT-FOUND. . . .
```

This assumes that the result of integer division is truncated, so if ILO and IHI are only 1 apart the resulting I is equal to ILO, and can never equal IHI.

6. Binary tree

Since the keys in the table for a binary search must be ordered, it is not a convenient method to use when a new key may be inserted into the table during the course of the program. For this a *binary tree* is useful. Consider a program to print the number of occurrences of each identical word in a text, or of

Downloaded from https://academic.oup.com/comjnl/article/20/3/202/751653 by guest on 19 April 2024

policies of each specific sum assured in a file, or of orders for each specific part number, the final print to be in alphabetic (numeric) sequence. The number of possible keys is very large, but the number of actual different keys in any one run is small enough to be included in a table in core. Various ways of organising the tree are possible; an example follows of a simple 'right threaded' tree. The data entry needed is:

```
01 TABLE.
      03 TABLE-ENTRY OCCURS 1000 (say).
         05 KEY-X          PIC . . .
         05 RESULT-Y       PIC . . . USAGE COMP.
         05 IL             PIC S9(4) USAGE COMP.
         05 IR             PIC S9(4) USAGE COMP.
      03 TABLE-LENGTH PIC S9(4) USAGE COMP
                           VALUE + 1000.
      03 INEXT PIC S9(4) USAGE COMP VALUE + 1.
      03 IP PIC S9(4) USAGE COMP.
      03 IQ PIC S9(4) USAGE COMP.
```

The table, initially empty, will become organised as a list, with IL and IR as 'left' and 'right' or low and high pointers, in each case containing the index to some other entry in the table (or zero, as a null index), according to a scheme such as in **Fig. 1**. $x(2) < x(5) < x(1) < x(6) < x(3) < x(4)$, i.e. all keys down the table to the left of any entry are lower in value, and keys to the right are higher. Where there is no left branch the left pointer is zero; where there is no right branch the right pointer contains a negative pointer to the next sequential entry.

To find or insert an entry in the table we use the following routine:

```
A1. IF INEXT = 1 MOVE 0 TO IQ GO TO A3.
              (For first entry in table)
    MOVE 1 TO IQ.
A2. MOVE IQ TO IP.
    IF X = KEY-X (IP) ADD 1 TO RESULT-Y (IP)
    GO TO A4. (Found).
    IF X LESS THAN KEY-X (IP)
      MOVE IL (IP) TO IQ
      IF IQ = 0 MOVE INEXT TO IL (IP)
                COMPUTE IQ = -IP
                GO TO A3, (Add entry to the left)
      ELSE GO TO A2. (Go down to the left)
    (So now X > KEY-X (IP))
    MOVE IR (IP) TO IQ
    IF IQ GREATER THAN 0 GO TO A2,
              (Go down to the right)
    ELSE MOVE INEXT TO IR (IP).
              (Add entry to the right)
```

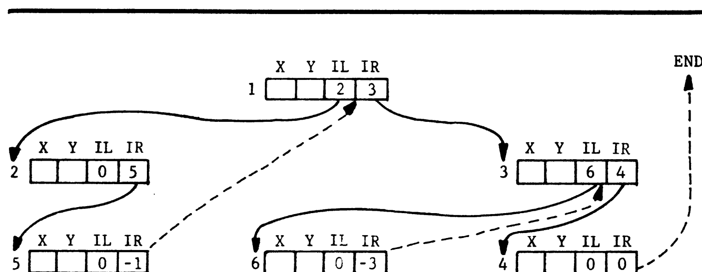

Fig. 1

```
A3. (To set up new entry).
    IF INEXT GREATER THAN TABLE-LENGTH GO
              TO ERROR-TABLE-FULL.
    MOVE ZERO TO IL (INEXT)
    MOVE IQ TO IR (INEXT)
    (If this entry was added to the left of IP, IQ contains -IP:
    if it was added to the right, IQ contains whatever was the
    previous right link of IP)
    MOVE X TO KEY-X (INEXT)
    MOVE 1 TO RESULT-Y (INEXT)
    ADD 1 TO INEXT.
A4. . . .
```

It is then easy to extract the entries from the table in sequence, perhaps for printing:

```
B1. IF INEXT = 1 GO TO B4. (No entries in the table)
    MOVE 1 TO IP.
B2. IF IL (IP) NOT = 0 MOVE IL (IP) TO IP
                       GO TO B2.
                       (Go down to left)
B3. PERFORM . . . print data for entry (IP) . . .
    MOVE IR (IP) TO IQ.
    IF IQ GREATER THAN 0 MOVE IQ TO IP
                         GO TO B2.
                         (Go down to right)
    IF IQ = 0 GO TO B4. (At the end)
    COMPUTE IP = -IQ GO TO B3.
    (Use a backwards thread)
B4. . . . .
```

It is not convenient to insert initial values into such a table, mainly because of the difficulty of getting the values of the pointers right, or of expecting clerical staff to understand and update correctly such a table. It may be useful to insert a few common keys in the first few entries of the table in an introductory paragraph; but if the table requires a large number of initial entries—perhaps values for the results—then these entries should be read from an initial file and inserted by program into the binary tree table. If frequently occurring keys are inserted in the first entries in the tree this method can gain in speed over the pure binary search, in which the frequent keys may unluckily be placed in positions that occur late in the search. It may also be desirable to reorganise the tree from time to time to keep it balanced; if, for example, input keys were already in sequence, the right hand branch would build up to a great length and the method would degenerate to a rather slow sequential search; but reorganising a tree is not such an easy procedure so it is fortunate that, if the keys are tolerably random, the tree is likely to be fairly well balanced.

7. More elaborate methods

When the table of entries becomes much larger then more elaborate methods of searching become necessary. It is also generally necessary to construct the table by program rather than by direct coding of values. It may not be possible to hold the whole table in core, and we then have the problems of accessing a random access or an indexed sequential type of structured file on backing store. The methods fall into two general classes corresponding to some extent to these two access methods—though they can be used for tables wholly in core just as well.

Hash tables are the natural means of dealing with large random access files. They require the transformation of each key by some randomising or hash function, $f(X)$, such that the transformed value can act like an index. The values of $f(X)$ must be integral, the range must be from 1 to some number larger but not too much larger than the number of entries in the table, and most values of X must produce a unique value for $f(X)$. The methodology of hash tables has been fully discussed in the literature, as can be seen from almost any recent volume of *The Computer Journal*. (An extensive bibliography is given by Knott (1975) and there is a useful introduction to hash tables by Maurer and Lewis (1975) in ACM *Computing Surveys*).

Downloaded from https://academic.oup.com/comjnl/article/20/3/202/751653 by guest on 19 April 2024

It is a convenient method for larger tables—say 300 or more entries in core, or as many as you like on disc. It has the disadvantage that it is impracticable to load the table except by program, and that the table cannot readily be processed sequentially. On the other hand, it is easy to alter the table dynamically during the program. However, it may be necessary to know something about the distribution of actually occurring keys in order to choose a suitable transformation function. Further the transformation must produce an integral result from a key which is possibly in character form; this depends on the internal representation of characters, and is not necessarily machine- or compiler-independent. Thus the method requires more care than is desirable for routine use in varied circumstances.

Both *indexed sequential* and *dictionary* look-up methods are multistage ones. In both cases the result of the first table to be entered is the location of an appropriate second table, and so on until the level of the appropriate entry is found. The searches of each table may be any of the previous methods—directly indexed, sequential search, binary search—and need not be the same at each stage. In an indexed sequential search the whole table is partitioned into roughly equal sized subtables and the effect is similar to a binary search in successive stages. With a dictionary search the key is partitioned into subfields and each subfield is used in turn as the entry for each stage of the search—equivalent to finding a word in a dictionary by search on each letter in turn. This may appear less efficient than the more balanced indexed search, but it may be useful when the separate subfields themselves are of significance, such as British Postcodes, or a list of personnel within department, within branch, within company say. The organisation of later subfields may even be different for different values of the earlier fields, so different processing may be necessary—consider STD telephone numbers or words in an inflected language like Latin. A dictionary search may be of use for quite a small table in core; an indexed sequential search is really only appropriate for a disc file—if the whole table were in core a binary search would be simpler.

8. Updating

The frequency and method for updating the table is of major importance in deciding on the appropriate search method. Allied to this is the method of constructing the table and of loading it into the appropriate part of the program. A number of methods will be described in roughly increasing sequence of 'dynamicness'.

First, the values in the table may be permanently fixed—such as the names of the months—and can therefore be an integral part of the program, either as direct literals in program statements or as values given by VALUE clauses in the data division entries (or their equivalents in other languages). Almost as static is the case where changes are sufficiently infrequent for it to be convenient for the *programmer* to update the source program and recompile.

If the table is normally unchanged from one program execution to the next, but still changes frequently enough for it to be worth avoiding recompilation—or perhaps because changes can be effected by a *clerical department* independently of programmers—then it may be convenient for the table values to be coded directly in an Assembler language module and compiled into a library ready to be overlaid into the area defined for the table in a COBOL program. Many installations have an overlay subroutine that can be called from a COBOL program for this purpose. It is necessary to ensure that the table does not become too big for the space reserved for it. If the look-up method requires the keys in the table to be in the correct sequence it may be necessary to check this after the table is loaded—an inconvenience of the binary search method. However the overlay method is particularly useful when several programs use the same table and it is important that they are all updated at the same time—separately coded tables that require independent recompilation are all too likely to become divergent. It may also be more convenient to code (IBM/360/370 Assembler) DC statements than COBOL VALUE clauses where the table entries are of varied form. And it may be easy for a nonprogramming clerical department to code and compile a fixed pattern of DC statements where they would not be allowed to alter full scale programs.

An Assembler overlay is not the only way of effecting the same result. If the table entries are complex and the table rather long—say over 50 entries with 10 varied format fields each—then it may be easier to code the values on data cards and use a program to construct a file on disc, which in turn is loaded into core at the beginning of the (COBOL) program that is to access it.

If the table values are likely to be altered somewhat for each execution of the program then the values can be held permanently on data cards, which are altered as required and read in to fill the table locations at the start of the program. But card handling is a perpetual source of error, so if the card file is at all large, if errors are costly and if the file is not changed totally from one run to the next, then it may be better to hold the file on disc and use special updating programs to effect the alterations.

If the table entries are wholly different from one run to the next then a direct loading of the table from cards is unavoidable, but by now the data cards have become an ordinary file of input data that requires verifying and processing like any other input data. This is almost indistinguishable from the wholly dynamic table structure which is constructed by the program from its own input data as it goes along—as a compiler does with the variable names of a source program. For these, the binary tree or hash table techniques are convenient, since the insertion of new entries may be as important as the retrieval of existing entries.

Finally, a program may look up a table that is a file on disc—organised as a direct random file or an indexed sequential file—without any responsibility for updating it; another set of programs deal with the updating and maintenance of that file. The form of file is then determined externally to the program doing the look-up. The problems of updating such a file are beyond the scope of this article.

Downloaded from https://academic.oup.com/comjnl/article/20/3/202/751653 by guest on 19 April 2024

References

BROOKS, F. D. and IVERSON, K. E. (1969). *Automatic Data Processing System/360 Edition*, John Wiley.
KNOTT, G. D. (1975). Hashing Functions, *The Computer Journal*, Vol. 18, No. 3, pp. 265-278.
KNUTH, D. E. (1973). *The Art of Computer Programming*, Vol. 3: Sorting and Searching, Addison-Wesley.
MAURER, W. D. and LEWIS, T. G. (1975). Hash Table Methods, ACM *Computing Surveys*, Vol. 7, No. 1, pp. 5-19.