A Faster Binary Search

An important technique results in faster-running applications programs and shorter response times.

Most applications of computer processing involve searching data tables of one form or another. The process is used in compilers, language interpreters, command processors, assemblers, database processors, and word processors. The regularity with which table searching is used makes the choice of searching techniques vital. A reduction in search time usually results in faster-running applications programs and shorter response times.

Although many techniques exist for searching tables in storage and on external media, the three principal ones are linear, series, and binary. The linear search examines each item, starting with the first, and proceeds sequentially. The series search, based on a mathematical series such as the power series or the Fibonacci series, works by subdividing the table of data in accordance with successive smaller numbers in the series. The binary search divides the table of data into two parts, rejecting one part and repeating the process on the other part until the item in question is found. (“Hashing” can be used to search by address calculation, but it sometimes yields the same key for more than one different field, which often reduces it to one of the three principal techniques.)

A discussion of a method of enhancing the binary search would not be complete without some background on the binary search itself. The binary search is appropriate for tables whose entries are in some order. Based on the concept of dividing a large problem into smaller parts, this technique involves dividing a list into two parts of equal size. None of the entries in one part meets the search criteria value (low), while an entry in the other part does meet that value (not low). The binary search divides the not-low part again, and the process of division continues until only one entry remains. The remaining entry, of course, matches the search item.

Usually, the midpoint of a table is computed by dividing the sum of the left and right indexes by two. Initial -
code modules with handwritten code.

The task of including internal documentation in programs is the bane of all programmers. This mechanical task has now been taken over by Quickcode. The programs it writes all contain detailed comments in English, which not only eases the job of modifying the generated code, but also assists the less-experienced user in learning the dBASE II language.

Evaluating the performance of any written material often becomes the personal judgment of an individual's style. Two conflicting styles of programming are in common practice. One style involves the use of all possible tricks and shortcuts in a language in order to optimize the speed of the running programs. Advocates of this method (often C and FORTH programmers) call it tight programming. Critics often refer to it as write-only code, because of the difficulty in reading it at a later date. Quickcode takes the opposite approach and generates clean, standard code. The resulting programs could run faster if shortcuts were taken, and some programmers might prefer to modify the code to take advantage of a personal speedup technique. I prefer a slow program that I can later enhance, instead of a fast but cryptic mess.

One area of performance where Quickcode clearly shines is in the elimination of programming bugs. A great deal of programming time is usually spent tracking down and removing these pesky critters. Because the code is being generated from prewritten text stored within the Quickcode program, syntax errors and improper use of commands are eliminated.

Overall, I would say that the quality of the programs produced by Quickcode is equal to that of a very methodical programmer with more than one year's experience with dBASE II.

Documentation

The 130-page manual that accompanies Quickcode is fairly easy to follow. A preliminary tutorial section is designed for overly anxious users who need their applications finished two weeks before buying the product. This is followed by detailed instructions on each section of the program.

Although there is a table of contents, the manual lacks an index. I hope that Fox & Geller finds the time to add one, even though it might seem to fly in the face of tradition.

The manual also needs more detailed application examples. Although the basic operation of Quickcode is clearly described, a sample inventory or accounts payable system would be helpful.

Limitations

While I am obviously pleased with most aspects of Quickcode, it has some limitations that should be made clear. A major weakness is the inability to create programs that access more than one data file. Also, some Fox & Geller advertisements claim that a complete accounting system could be "knocked out in a weekend." Typically, accounting systems consist of several modules that share data files. For example, a receivable module must be able to access the files of an inventory module. And although adequate inventory and receivable systems could be written with Quickcode, the necessary integration of the two systems would require a fair amount of programming knowledge. The other major weakness is the lack of any sophisticated report-writing facilities. I hope that Fox & Geller will be able to address these limitations in a later version.

Conclusions

Quickcode is a well-written, easy-to-use program generator for the dBASE II programming language, which allows the user to describe an application by simply filling in screens.

A large amount of code (more than 30K bytes) can be generated in less than two minutes. The code produced is modular, easily modified, and runs at an acceptable speed.

The manual included with the program is clearly written, but lacks an index and sufficient application examples. The limitations of Quickcode include the inability to access more than one data file and a weakness in the report-writing functions.

The major audiences for Quickcode are dBASE II users with little programming background and programmers who need to produce large amounts of standard code quickly.
A search tree representing the binary-search process. The search begins at the top with the root node and proceeds down the tree to the leaf or terminal nodes. This process continues until the search argument is found or the table is exhausted.

ly, the left and right indexes are the two extremes of the table. Comparing the search argument to the table entry at the midpoint determines whether the right or left index is replaced. The process continues until the matching entry is found or the table has been reduced to an empty state.

In the quest for enhancement, many different techniques exist for analyzing the process time of an algorithm. In searching, for example, the usual technique is to examine the number of comparisons required to locate an item in a table. Because my enhancement of the binary-search algorithm involves moving a portion of the midpoint-calculation code after a comparison is made, the analysis that I will present focuses on the number of comparisons and the number of required iterations of the midpoint-calculation code.

The binary-search process I devised is a traversal of an implicitly defined binary-search tree that is a complete binary tree as well. Like all traversals, it begins with the root node and proceeds down the tree to the leaf or terminal node. Figure 1 shows a representation of the search process as a search tree.

In the binary-search process, if it takes one unit of time to locate the third element in a table of seven entries, then the computation time necessary to locate the third entry does not double until the table is expanded to 31 entries. In other words, the binary search resembles a logarithmic pattern despite the use of the division process. It is this logarithmic performance that has led to the false conclusion that little can be done to improve the binary search.

A long-standing rule of thumb about random access to data files is that 80 percent of the activity is concerned with only 20 percent of the file. The implication is that after a data argument has been seen, the probability of seeing it on the next request is 3.25 times that for the total random case.

Files and tables share an important characteristic: both can be viewed as linear-ordered representations of the records to be inspected and retrieved. Extending the 80/20 rule to tables, then, suggests a means for improving the performance of a binary search.

Analysis of Enhancement

Figure 2 shows the implicit tree used for the enhanced-search process. In this example, a prior search returned the eighth entry of the table (P represents the node returned by a...
prior search). The subtrees to either side represent the search path used as a result of the first comparison. Although the table's 11 elements require a complete binary-search tree to a depth of 4, the enhanced search has two trees to consider: the tree shown on the left has a depth of 3, and the one on the right has a depth of 2. The reduction in depth indicates a reduction in the number of iterations required by the search.

Figure 3 illustrates the depth of each node in a full binary-search tree and the total number of accesses required to inspect every node in the tree. A binary search of a table of seven elements would require an average of 2.43 (17/7) accesses if the likelihood for all cases were equal.

Table 1 illustrates all of the possible cases for a table of four elements. The left half of the table shows the number of comparisons required for each element in each configuration. The right half of the table shows the number of iterations through the midpoint-calculation code if the comparison is moved to the beginning of the loop and the previous search information is used. The average number of comparisons in the example shown is 2.13 (34/16), but the number of iterations is 1.13 (18/16). Traditional implementations would have required 2.00 iterations of the comparison code and the midpoint-calculation code.

Table 2 summarizes the possible cases for a table of eight elements and the number of table interrogations required to inspect every entry in every subtable for every case. The subtable “weights” reflect the number of iterations required in each case. If the weights are added and the equal-likelihood assumption is applied, the result is an average of 2.84 table accesses and 1.84 iterations.

For a full binary tree of depth D, there are $2^{(D-1)}$ nodes at that depth. In general, at depth K there are $2^{(K-1)}$ nodes at the level of K in the tree. When the tree is full (meaning all nodes are present at a level), the average number of comparisons (C) to locate a node, assuming equal likelihood, is the sum of the levels for each node divided by the number of nodes. Thus

$$C = \sum_{1}^{n} \frac{I(2^{r-1})}{2^{D-1}}$$

To extend to the case for the complete, but not full, binary tree, the average becomes

$$C = \sum_{1}^{n} \frac{I(2^{r-1})+R(D+1)}{N}$$
where \( N \) is the number of nodes in the tree satisfying the relation

\[ N = 2^R + R - 1 \]

The solution for the general case becomes

\[ C = \frac{(D-1)(2^R) + R(D+1)}{N} \]

Although the formulas imply a method to calculate the average number of comparisons as a function of \( N \) by solving for \( D \) and \( R \), the round-off errors in the calculations of \( \log(2, N) \) produce incorrect results. Thus the best way to calculate the average number of comparisons is through an iterative process that determinates the depth of the complete tree and adds the remaining weights.

When information from a prior search is available, you compute the number of cases by solving for \( \log_2(N) \). The number of comparisons for the nodes in the subtables, you can simply divide the sum by the number of cases \((N-1)\). If \( K \) is set to

\[ K = \sum_{m=1}^{N-1} \text{subtable weight}, \]

the result after simplification is

\[ C = 1 + \frac{2K}{N^2} \]

The number of iterations of the midpoint-calculation code is

\[ I = \frac{2K}{N^2} \]

The above derivations apply to the case of equal likelihood; however, it is possible to have the case of never-equal likelihood. Changing the probabilities for the never-equal case produces

\[ C_{\text{NEVER}} = 1 + \frac{2K}{N(N-1)} \]

The number of iterations of the midpoint-calculation code is

\[ I_{\text{NEVER}} = \frac{2K}{N(N-1)} \]

A general formula relating the probability of a match with a prior search argument (M) and the size of the table (N) is

\[ C = 1 + \frac{2K(1-M)}{N(N-1)} \]

and the number of iterations becomes

\[ I = \frac{2K(1-M)}{N(N-1)} \]

The possibility of the 80/20 rule applying in an example requires that we compute the probability of the occurrence of a duplicate argument. The rule divides the members of the table into two sets: high activity (H) and low activity (L). A duplicate occurrence can exist only if the prior and current arguments are members of the same set. If \( X \) represents the prior argument and \( Y \) the current argument, the probability of duplication can be computed by

\[ P(X = Y) = A \cdot B \cdot C \cdot + D \cdot E \cdot F \]

where

\[ A = P(X = Y | X, Y \text{ in } H) = \frac{1}{0.2N} \]
\[ B = P(X \text{ in } H) = 0.8 \]
\[ C = P(Y \text{ in } H) = 0.8 \]
\[ D = P(X = Y | X, Y \text{ in } L) = \frac{1}{0.8N} \]
\[ E = P(X \text{ in } L) = 0.2 \]

and

\[ F = P(Y \text{ in } L) = 0.2 \]

The resulting simplifications produce

\[ P(X = Y) = 3.25/N \]

and

\[ P(X \neq Y) = (N - 3.25)/N \]

Thus

\[ C_{\text{RULE}} = 1 + \frac{2K(N-3.25)}{N^2(N-1)} \]

As before, the number of iterations of the midpoint-calculation code is

\[ I_{\text{RULE}} = \frac{2K(N-3.25)}{N^2(N-1)} \]

The results of these equations are shown in table 3, which compares a pure binary search for tables of dif-

---

**ENHANCE YOUR COLOR COMPUTER WITH THESE GREAT PRODUCTS!**

**MACRO-BGC DISK BASED EDITOR/ASSEMBLER**
This is a powerful macro assembler, screen oriented editor and machine language monitor. It features local labels, conditional assembly, printer formatting and cross reference listings. Assemble multiple files. Program comes on Radio Shack compatible disk with extensive documentation. Price: $99.95

**MICROTEXT COMMUNICATIONS**
Make your computer an intelligent printing terminal with off-line storage! Use Microtext for timesharing interactions, printing what is received as it is received and saving text to cassette. and more! Price: $59.95

**PI80C PARALLEL PRINTER INTERFACE**
Use a parallel printer with your Color Computer! Serial Parallel converter plugs into the serial port and allows use of Centronics compatible printers. You supply the printer cable. Price: $69.95

**PI80C PARALLEL PRINTER INTERFACE**

**THE MICRO WORKS COLOR FORTH**
Color Forth is easier to learn than assembly language, executes in less time than Basic and is faster to program in than Basic. Rompack comes with 112-page manual containing glossary of system-specific words, full standard F1G glossary and complete source. A fascinating language designed for the Color Computer! Price: $109.95

**SDS-BGC SOFTWARE DEVELOPMENT SYSTEM**
SDS-BGC is a Rompack containing a complete editor, assembler and monitor. It allows the user to write, assemble and debug assembly language programs with no reloading, object patching or other hassles. Supports full 6809 instruction set. Price: $89.95

**80C DISASSEMBLER**
Runs on the Color Computer and generates your own source listing of the Basic interpreter ROM. Documentation includes useful ROM entry points, complete memory map, I/O hardware details and more. Cassette requires 16K system. Price: $49.95

**THE MICRO WORKS**
Also Available: Machine Language Monitor  Book Memory Upgrade Kits Parts and Services Call or write for more information California Residents add 6% Tax Master Charge/VISA and COD Accepted

---

P.O. BOX 1110 DEL MAR, CA 92014  619-942-2400

Circle 278 on inquiry card.
### Table 3: A comparison of the binary search and the enhanced binary search.

<table>
<thead>
<tr>
<th>Table Size</th>
<th>Binary Search</th>
<th>Enhanced Search</th>
<th>80/20</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Equal Likelihood</td>
<td>Never Equal</td>
<td>Rule</td>
</tr>
<tr>
<td>1.00</td>
<td>1.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>2.00</td>
<td>1.50</td>
<td>0.50</td>
<td>1.00</td>
</tr>
<tr>
<td>3.00</td>
<td>1.67</td>
<td>0.89</td>
<td>1.33</td>
</tr>
<tr>
<td>4.00</td>
<td>2.00</td>
<td>1.13</td>
<td>1.50</td>
</tr>
<tr>
<td>5.00</td>
<td>2.20</td>
<td>1.36</td>
<td>1.70</td>
</tr>
<tr>
<td>6.00</td>
<td>2.33</td>
<td>1.55</td>
<td>1.87</td>
</tr>
<tr>
<td>7.00</td>
<td>2.43</td>
<td>1.71</td>
<td>2.00</td>
</tr>
<tr>
<td>8.00</td>
<td>2.63</td>
<td>1.84</td>
<td>2.11</td>
</tr>
<tr>
<td>9.00</td>
<td>2.78</td>
<td>1.98</td>
<td>2.22</td>
</tr>
<tr>
<td>10.00</td>
<td>2.90</td>
<td>2.10</td>
<td>2.33</td>
</tr>
<tr>
<td>30.00</td>
<td>4.13</td>
<td>3.44</td>
<td>3.58</td>
</tr>
<tr>
<td>50.00</td>
<td>4.86</td>
<td>4.12</td>
<td>4.20</td>
</tr>
<tr>
<td>100.00</td>
<td>5.80</td>
<td>5.06</td>
<td>5.11</td>
</tr>
<tr>
<td>300.00</td>
<td>7.33</td>
<td>6.59</td>
<td>6.61</td>
</tr>
<tr>
<td>500.00</td>
<td>8.00</td>
<td>7.32</td>
<td>7.34</td>
</tr>
<tr>
<td>1000.00</td>
<td>8.99</td>
<td>8.31</td>
<td>8.32</td>
</tr>
<tr>
<td>5000.00</td>
<td>11.36</td>
<td>10.62</td>
<td>10.62</td>
</tr>
<tr>
<td>10000.00</td>
<td>12.36</td>
<td>11.62</td>
<td>11.62</td>
</tr>
</tbody>
</table>

The average number of iterations of the midpoint-calculation code that are required to find the search argument are given for different sizes of the table being searched. For the enhanced binary search, the number of iterations is given for three different assumptions about the table being searched: (1) that each element in the table is equally likely to match the search argument, (2) that no two elements in the table are equally likely to match the search argument, and (3) that after a data argument has been seen, the probability of seeing it again after the next iteration is 3.25 times greater than the probability for the random case (the 80/20 rule).

Different sizes to the enhanced binary search in the cases of equal likelihood, never-equal likelihood, and the 80/20 rule.

While a binary search can be implemented in many ways, traditional implementations require the initialization of local variables (five PL/I statements) followed by a loop composed of the midpoint calculation (five PL/I statements) and a comparison of the search argument with an entry in the table (three PL/I statements). The enhanced search is similar in structure, but its midpoint calculation follows the comparison. If the processor that executes the searches requires one instruction cycle per PL/I style statement, the binary-search time (BT) can be expressed as

\[ BT = 5 + 8C \]

and the enhanced-search time (ET) can be expressed as

\[ ET = 5 + 3C + 5I = 8 + 8I \]

in which C represents the number of comparisons and I represents the number of iterations needed to satisfy the search.

If in the two preceding equations we substitute the number of comparisons and the number of iterations indicated in table 3, a comparison of the data indicates that the enhanced search is usually better than a pure binary search. If the tables contain approximately 300 entries and an equal likelihood applies, the enhanced search results in an advantage of approximately 6 percent. A higher probability of duplication increases the reduction-in-time advantage of the enhanced search. If your processor takes a long time to perform a divide or shift, the advantage approaches 10 percent.

### A Description of the Process

We can express the process for the improved binary search in several ways. Table 4 is an example of a decision table that represents a looping process. The first row of entries...
Table 4: A decision table for the enhanced binary search specifying the various actions to be performed under various combinations of conditions. The labels shown in the gray areas are external to the decision table. The decision table itself is divided both horizontally and vertically. The upper part is called the “condition” portion; the lower part, shaded in blue, is the “action” portion. The left portion of the table, called the “stub,” identifies the tests to be performed and the actions to be taken (in this case, data transformations). The right portion is divided into six columns, each of which expresses a decision rule. The first row of each column shows the condition under which a decision rule applies, and the lower rows show the actions to be performed if those conditions are true. For example, if M is not equal to 0, we must select one of columns 2 through 6. Moreover, if ARG is greater than or equal to TABARG(M), we can narrow our choice to columns 2 through 4. If L is also less than R, then all columns except the third are ruled out. Therefore that column expresses the relevant decision rule. Looking down that column to its action portion, you can see that two actions are selected: L is to be set equal to M+1, and M is to be set equal to (L+R)/2. All the statements in the stub are from the PL/1 program shown in listing 1. The variables represent the following: ARG, the search argument (the value being searched for); TABARG, the function argument (the value at the current midpoint address); M, the midpoint address; L, the left (or low) extreme address; R, the right (or high) extreme address. The programming language used in the decision-table stubs is PL/1, but converting the statements to APL, Pascal, BASIC, or machine codes would not be difficult.

A brief description of the enhanced binary-search process provides an understanding of the procedure that is employed when the searching process uses the prior search results and completes the search using the reduced implicit-search tree. For the

describes the tests that have to be performed for the process to work correctly. The next row indicates the various data transformations that will be applied. The third specifies when and how the loop will be terminated. The fourth row describes the initial steps that are required, and the fifth row provides instructions for terminating the execution process. The YN-column entries specify the results of the condition tests that must be satisfied to select a column. The numbers in the column identify the actions to be selected and their sequence. The X values select the loop-termination criteria. The decision table presents, in an abstract manner, all of the information that is required for a program without requiring a unique implementation.

The programming language used in the decision-table stubs is PL/1, but converting the statements to APL, Pascal, BASIC, or machine codes would not be difficult.
following situations we will assume that the table is an ascending linear list in an array data structure. The calling sequence takes for granted a call parameter that contains the prior index returned for a prior search of the entries in the table. The prior index value returned is initialized to 0 if there is no prior search data available and then updated by the searching process:

- If the current index is 0, the midpoint address is recalculated for the next iteration and the process continues.
- If the search argument is not less than the function argument and the low address is less than the high address, the low address is replaced with the midpoint-plus-one entry. The midpoint address is recalculated for the next iteration.
- If the search argument is equal to the function argument in the table, the current midpoint is the value returned to the calling program.
- If the search argument is less than the function argument in the table and the low address is less than the high address, then the high address is replaced with the midpoint address. The midpoint address is recalculated for the next iteration.
- If no entry is found, the current midpoint is set to 0.
- The current midpoint is the value returned to the calling program when all iterations have been completed.

The decision table (table 4) illustrates how to use the enhanced-search technique. One of the many possible implementations is illustrated in listing 1.

**Listing 1:** A PL/1 procedure that carries out an enhanced binary search. The first line identifies the procedure and its variables and states that it will return a fixed value. The second line declares the variables so the computer can arrange appropriate storage for the kind of values that each variable can assume. ARG represents the search argument (the value being searched for); TABARG, the function argument (the value at the current midpoint address); M, the midpoint address; L, the left (or low) extreme address; and R, the right (or high) extreme address. The procedure works by repeatedly setting the value of one of the extremes (R or L) to the previous midpoint value and then calculating a new midpoint by adding the extremes and dividing by two. Statements between "*" and "*" are comments.

SEARCH: PROCEDURE (ARG, TABARG, LEFT, RIGHT, M) RETURNS (FIXED);
DCL (ARG, TABARG(*), LEFT, RIGHT, L, R, M) FIXED;
L=LEFT;
R=RIGHT;
ENHANCE='0'B;
DO WHILE (ENHANCE='0'B);
  IF M=0 THEN
    DO;
     M=(L+R)/2;
     /\ LOOP *;*
    END;
  ELSE
    DO;
     IF ARG>=TABARG(M) THEN
       DO;
        IF ARG=TABLEGARG(M) THEN
          DO;
           RETURN(M);
          END;
        ELSE
          DO;
            IF L<R THEN
              DO;
               L=M+1;
               M=(L+R)/2;
              /\ LOOP *;/
            END;
          ELSE
            DO;
             M=0;
             RETURN(M);
          END;
         END;
     END;
  ELSE
    DO;
     IF L<R THEN
      DO;
       R=M;
       M=(L+R)/2;
      /\ LOOP *;/
      END;
    ELSE
      DO;
       M=0;
       RETURN(M);
     END;
     END;
   END;
/* NO SPECIAL TERMINATIONS */;
END;

**References**