Binary Search

Siem Korteweg
J. v. Stolberglaan 16
3931 KA Woudenberg
the Netherlands

Hans Nieuwenhuÿzen
Sterrewacht "Sonnenborgh"
Zonnenburg 2
3512 NL Utrecht
the Netherlands

Abstract

This paper describes a new dictionary structure supporting binary search. This dictionary structure can be implemented without a penalty in memory usage. It appears in this implementation that vocabularies can increase the searching time. Therefore, the new dictionary structure does not support vocabularies.

Objectives and Strategy

It is our goal to speed up compilation. During compilation, every definition can be generated, i.e., inserted in the dictionary, only once. It can also be FORGETten (removed from the dictionary) only once, but it can be interpreted, requiring it to be searched in the dictionary, any number of times. A definition that calls 'd' other definitions or numbers requires one insertion and 'd+2' searches, one for the colon and one for the semicolon.

One could try to optimize the insert or delete operations, but the reduction of the compilation time will only be marginal, as these operations require little time in classical dictionary structures. They are also applied far less frequently than the theoretically slow linear search operation. Therefore we have chosen to optimize the search operation and ignore minor effects on the insertion and deletion times.

Searching Structures

Hashing is a technique that transforms an identification key directly to an address in a suitable table. Traditionally, this table has a fixed size, leading to a waste of memory when the table is too large, or to time-expensive rehashing when the table overcrowds. However, new techniques have been developed to overcome these drawbacks [FAGI78]. Forth systems using hashing exist, see [DOWL81] and [MCNE81]. It remains to be studied to what extent the new hashing techniques can meet the specific demands of a Forth dictionary structure.

Trees support searching mechanisms with theoretically optimal O(logN) time characteristics. [REIN77] gives examples of trees, and [CURR80] shows a Forth implementation of a dictionary structure based on an AVL-tree. Trees, however, require much overhead, caused by many pointers

in the nodes of the trees or by elaborate algorithms to update the tree after insertions and deletions.

Therefore a new dictionary structure has been developed, offering the same time characteristics for the searching mechanism, but demanding less memory overhead.

Binary search algorithms, working on a row of objects in some order, offer the same time characteristics as searching mechanisms supported by trees. When headers are placed in alphabetical order in a row, much data movement is introduced to preserve the alphabetical order after insertions and deletions. To avoid this, index pointers can be used in the row. When these pointers give the alphabetical order of the headers, the headers and the code can remain in the usual historical order in the dictionary. Note that only the index pointers are used to search a header; this implies that a link field is no longer needed in the headers. Therefore a binary search can be implemented without memory overhead compared with the usual linear dictionary structure.

Modified Binary Search

Normal binary search algorithms stop when the searched object is found. As this new dictionary structure has to support redefinitions, the new search algorithm cannot use this stop-criterion. Using the normal stop-criterion depends upon the number of entries in the dictionary where redefinition is found. Note that the index pointers to redefinitions reside consecutively in the row of index pointers. To avoid this, we adopt the following stop-criterion. The binary search is continued in the lower/upper part of the searched interval until it contains only one index pointer. This index pointer is the leftmost/rightmost index pointer of the redefinitions, depending on the choice for the lower/upper part of the interval. By inserting a new index pointer of a redefinition before/after the other index pointers of redefinitions, a historical order of the index pointers of redefinitions is maintained. This enables one to find the most recent redefinition by means of binary search.

Insertion consists of the addition of a new index pointer, requiring some index pointers to be moved, and the creation of a new header. This implies that the new insert routine will be somewhat slower than usual dictionary structures caused by moving the index pointers.

`FORGET` consists of the removal of index pointers corresponding to entries in the dictionary that were added after the routine to be deleted, and the simple removal of headers and code by means of an adjustment of the dictionary pointer. As the order of the index pointers and the historical order of the headers and the code will differ, in general a linear scan of the entire row of index pointers is used to select the index pointers to be removed. This is similar to the linear scan of a usual dictionary in the case of a `FORGET`.

In some implementations the routine `VLIST` will list all the information in the headers. As the headers no longer contain a link field, the index pointers are used for this, resulting in an alphabetical order of information. The routine `ALIST` is defined to list the header information in alphabetical order.

Vocabularies

Vocabularies can be implemented in the following way: every vocabulary consists of a row of index pointers and a link to another vocabulary that is to be searched entirely when the first vocabulary does not contain the searched entry. This is not equivalent with usual vocabularies, as in that case the search in the second vocabulary might start anywhere in that vocabulary.

The rows of index pointers are relocatable, but the aforementioned implementation of vocabularies requires a memory-manager-like algorithm to update the rows in case one of the vocabularies overcrowds. Therefore we looked to see whether vocabularies are still useful in the new dictionary structure. Usual implementations use vocabularies to:

- Allow words in different contexts. Depending upon the context, we see different parts of the dictionary.
- Decrease the searching time by decreasing the number of headers to be checked, as only a part of the dictionary is seen.

Obviously, the first advantage will hold for the new structure, but does the second also hold? Suppose there are three vocabularies, named A, B, and C, where vocabulary C contains all the headers of both A and B, and suppose that A, B, and C require #A, #B, and #C sweeps of binary search respectively. Then the following relationship holds:

$$\text{MAX}(\#A,\#B) <= C <= \text{MAX}(\#A,\#B) + 1$$

Using vocabularies A and B instead of C, a search is started for a header in A, and when it is not found, B is also searched. Suppose we search X times for a header in A, and Y times for a header residing in B. When A contains only a few very frequently used routines, the search time is reduced. The comparison of searching in A and B, opposed to searching in C only, is denoted by the fraction of the binary search sweeps required in both cases:

$$\frac{X\#A / Y(\#A+\#B)}{\#C(X+Y)} = \frac{\#A}{\#C} + \frac{\#B/\#C}{X/Y+1} \rightarrow \frac{\#A}{\#C} + \frac{1}{X/Y+1} \rightarrow \frac{\#A}{\#C}$$

The approximation holds when it is assumed that #B > #A and x >> Y. It is obvious that the required conditions are not always met. The savings are at most 1-#A/#C. For instance, when A and B contain 30 and 210 headers respectively, #A=5, #B=8, and #C=8, and the savings are at most 3/8, or 35%.

Using vocabularies in the binary search dictionary structure, the savings for the searching time are not so dramatic. This can be seen from the argument that a reduction of the number of headers in the binary search structure with a factor 'r', does not mean a reduction of the searching time by a factor 'r', as in the case of the usual, linear dictionary structure, but by a term log(r).

From the above relation, it can be seen that there will be many situations (e.g. #A=#C) in which the use of vocabularies will slow down the binary search. This implies that vocabularies should not be implemented in the new dictionary except when most applications use the concept of different contexts.

Test Results

In the following tests we have put identifiers from the FYS FORTH kernel and the identifiers from the source of the new structure together in a test dictionary.

We have made the following assumption for the usual dictionary: the average time required to search an entry is ½*n*c, where n is the number of entries in the dictionary, and c the time required to compare two identifiers. By searching the (2*n)th entry in a usual dictionary, we find the average searching time for a linear dictionary containing n entries.

The number of sweeps done by binary search drops from 9 to 7 as the number of entries ranges from 400 to 100. The recorded times also give a decrease of about 20%, which is consistent with the above reduction. The classical implementation against which the tests have been made is FYS FORTH on APPLE][(cf. [JOOS81]).

number of headers in the dictionary	classical structure	new structure
	time (msec)	
100	3	2.3
200	6	2.5
300	8	2.8
400	11	2.8

Table 1: The time required to search for one identifier.

Note that for this implementation the crossover point for the mean time of identifier searching is approximately a dictionary containing X entries, where X*3/100 = (2log(2.5))*2log(X), so X=70 entries.

It appears that the time required to list the contents of the headers using the screen output dominates the time required for a sequential search for all the headers. This means that the time required by VLIST is practically independent of the searching structure.

We have not tested the searching times for a classical implementation, but the possible check against redefinitions in that case requires a linear search of the whole dictionary, requiring 6-22 msec, as the number of headers ranges from 100 to 400. The time required to insert 25 headers in the new dictionary containing 100-400 entries is 1 sec. Some tests with large numbers of inserted entries also show an insertion time of approximately 40 msec per header.

Why is the insertion time almost independent of the number of entries in the dictionary? The time required for the search is at most 3 msec, the time to relocate some index pointers at most 12 msec (empirically found for our implementation). The time characteristics of these two routines depend upon the number of entries in the dictionary. They require, at most, some 30% of the total time for insertion. The other 70% is overhead, independent of the number of entries in the dictionary.

`FORGET` in our test system and `FORGET` in FYS FORTH require almost the same amount of time. This is consistent with the remark made in the discussion of the implementation of the `FORGET` operation.

Conclusions

The traditional linear search is replaced by binary search without penalties in memory usage. Some test data has been given for the binary search implemented in a hybrid system. The crossover point is a dictionary containing some 70 headers. When the dictionary contains some 400 entries, the binary search is four times faster than the linear search algorithm.

Insertion and deletion (i.e. `FORGET`) operations have become somewhat slower, but these operations are used far less frequently than the optimized search.

Listing header information (`VLIST` in some implementations) is easily adapted without a penalty in execution time. As the headers no longer contain a link field, it will require some effort to implement an historical list of headers. Using the index pointers, the headers are listed in alphabetical order.

Vocabularies can still be useful, as they allow words to be used in different contexts. However, due to the O(logN) time for binary search, they might even increase the searching time for the binary search dictionary structure. One should, therefore, consider carefully before implememting vocabularies in the new dictionary structure.

Acknowledgements

We thank prof. Dr. J. V. Leeuwen of the department of Computer Science of the State University of Utrecht for giving Siem Korteweg the opportunity to work a year at the observatory for the final phase of his study. We also thank Rieks Joosten and Frans Cornelis for the time they spent discussing and analyzing this concept and its use, and Hans v. Koppen for the use of his room and his Apple II computer.

We thank the reviewers of this paper for their meticulous reading and many important suggestions.

Bibliography

[BART79] P. Barthodi. "The TO Solution," *Forth Dimensions,* Vol. 1 No. 4, pp 38-41, 1979.

[CURR80] D. H. Currie. "Balanced Tree Search", *Forth Dimensions,* Vol. 2 No. 4, 1980.

[DOWL81] T. Dowling. "Hash Encoded Forth Name Fields," *1981 Rochester Forth Standards Conference,* Institute for Applied Forth Research, Inc., Rochester, NY, 1981.

[FAGI78] R. Fagin, J. Nievergelt, N. Pippenger and H. R. Strong. *Extendible Hashing – A Fast Access Method for Dynamic Files,* Research Report, 1978.

[JOOS81] R. Joosten. *FYS FORTH 0.2/0.3 Preliminary User Manual,* State University Utrecht, Utrecht, the Netherlands, 1981.

[JOOS83] R. Joosten and H. Nieuwenhuyzen. "Vectoring Arrays of Structures," *The Journal of Forth Application and Research,* Vol. 1 No. 2, pp 35-53, 1983.

[KNUT73] R. Knuth. Sorting and Searching, Vol. II of the *Art of Computer Programming,* Addison-Wesley, Reading, MA, 1973.

[MCNE72] M. McNeil. "Hashed Dictionary Searching," *FORML 1981,* Forth Interest Group, San Jose, CA, 1981.

[REIN77] E. M. Reingold, J. Nievergelt and N. Deo. *Combinatoral Algorithms,* Prentice-Hall, 1977, chapter 6.

Manuscript received April 1984.

To obtain a copy of the FYS FORTH manual of FYS FORTH 0.3 please contact: Hans Nieuwenhuÿzen, Sterrewacht "Sonnenborgh", Zonneburg 2, 3512 NL Utrecht, the Netherlands.

Appendix: High Level Implementation of the Algorithms

The source code resides on file and is written in FYS FORTH (cf. [JOOS81]). For the sake of clarity, the definitions will use many VALUEs, see [BART79]. For deviations of FYS FORTH from the FORTH-79 Standard, see [JOOS83].

```
BASE HEX
CREATE BASE.I 400 ALLOT                          ( index pointer space )

: INDEX 2* BASE.I + ;        ( transform indices into abs. addresses)

0 VALUE BOI                                    ( beginning of indices )
0 VALUE EOI                                        ( end of indices )
0 VALUE LEFT                                ( pointer for binary search )
0 VALUE RIGHT                               ( pointer for binary search )
0 VALUE MIDDLE                              ( pointer for binary search )
0 VALUE PTR                                 ( pointer for binary search )
0 VALUE ?FOUND                                 ( flag for binary search )
0 VALUE FORGET.ADR                              ( help value for FORGET )
0 VALUE FIRST                                     ( pointer for FORGET )
0 VALUE FOLLOW                                    ( pointer for FORGET )
0 VALUE #DEL                                    ( help value for FORGET )

( INITIALIZATION OF THE DICTIONARY )
( due to choice of the lower/upper interval in the binary search   )
( algorithm, the first/last entry in the dictionary cannot be      )
( found. To avoid this, an index pointer is added to a dummy       )
( header to the row of index pointers.                             )

0 TO BOI                                  ( initialization, the dictionary )
0 TO EOI                                  ( contains only one dummy header )

-WORD X  BUILDHEADER                                ( build dummy header )
DUP 1+                                                 ( 1+: skip count )
0 !                          ( this identifier is first in alph. order )
BOI INDEX !                                          ( add index pointer )
```

```
: COMPARE                          ( beg.adr1,beg.adr2,#chars -> order )
   ( this routine has to be optimized in low level and is          )
   ( implementation dependent.                                     )
   ( order=-1 if STR1 > STR2 )
   ( order= 0 if STR1 = STR2 )
   ( order= 1 if STR1 < STR2 ) ;

: <= > 0= ;

: BINSEARCH                              ( beg.adr.str -> index,?found )
   ( invariant: header at LEFT  INDEX @ <  searched header )
   (            header at RIGHT INDEX @ >= searched header )
   TO PTR
   BOI TO LEFT
   EOI TO RIGHT
   0 TO ?FOUND
BEGIN RIGHT LEFT - 1 <>
WHILE RIGHT LEFT + 2/ TO MIDDLE
      PTR COUNT
      MIDDLE INDEX @ COUNT
      ROT MIN
      COMPARE
      DOCASE
        -1 CASE MIDDLE TO LEFT       ELSE
         0 CASE MIDDLE TO RIGHT
                1 to ?FOUND          ELSE
         1 CASE MIDDLE TO RIGHT
      ENDCASE
REPEAT
RIGHT ?FOUND ;

: FIND                         ( beg.adr.str -> PFA,1 when found     )
   BINSEARCH                   (                 0 when not found    )
   IF   INDEX @ PFA 1                              ( 1: found        )
   ELSE DROP 0                                     ( 0: not found    )
   THEN ;

: BUILD.HEADER                ( beg.adr.ident -> beg.adr.header )
   ( as usual, but no link field is generated    )  ;

: ID.                         ( beg.adr.header -> -- )
   ( list information in header, implementation dependent ) ;
```

```
: INSERT                        ( -- -> -- )
   DUP BUILD.HEADER
   SWAP BINSEARCH
   WARNING -1 <> AND
   IF CR ." "REDEFINING :" DUP INDEX @ ID. THEN
   TO PTR
   PTR INDEX DUP 2+          ( 2+: prepare location for one pointer )
   EOI PTR - 2*             (2*: index pointers consist of two bytes )
   CMOVE                                                ( make room )
   PTR INDEX !                                 ( add new index pointer )
   1 +TO EOI ;

: ALIST                         ( -- -> -- )
   EOI 1                                      ( 1: ignore dummy header )
   ?DO ?KEY Ø=                                ( until a key is pressed )
   WHILE I INDEX @ ID.
   LOOP ;

: FORGET                        ( -- -> -- )
   -WORD FIND Ø= 4 ?ERROR                                  ( cannot find )
   TO FORGET.ADR
   Ø TO #DEL
   BOI INDEX DUP TO FIRST TO FOLLOW
   EOI Ø
   DO FORGET.ADR FIRST @.DICT <=              ( header to be removed? )
      IF   1 +TO #DEL
      ELSE FIRST @ FOLLOW !                      ( keep ptr in array  )
           2 +TO FOLLOW                             ( advance FOLLOW )
      THEN
      2 +TO FIRST                                   ( advance FIRST  )
   LOOP
   #DEL -TO EOI              ( adjust bound of row of index pointers )
   FORGET.ADR HERE - ALLOT ;             ( adjust dictionary pointer )

TO BASE
```