PREPRINT. Software-Practice and Experience, 21(10):1125-1128, 1991.

A Note on Searching in a Binary Search Tree

Arne Andersson
Department of Computer Science, Lund University,
BOX 118, S-221 00 Lund, Sweden

SUMMARY

An algorithm for searching in a binary search tree using two-way comparisons is presented. The number of comparisons required by this algorithm is only one more than when using three-way comparisons. Since most high-level programming languages do not supply three-way comparisons, the number of comparisons used *de facto* **are reduced by a factor of two. We give experimental results to indicate the speedup that may be achieved by the presented algorithm.**

KEY WORDS: binary search tree, searching, two-way comparison, code optimization

Introduction

An operation which is often assumed to be present when designing algorithms is the three-way comparison with outcome $<$, $=$ or $>$. An example of an algorithm where this operation is used is when searching an element in a binary search tree. This is a fundamental algorithm and it is often presented in various text books and papers [1, 3, 4, 5, 7]. The basic algorithm uses one three-way comparison per visited node. Depending on the outcome of the comparison the algorithm terminates or continues in one of the subtrees. However, even though the three-way comparison may exist at the machine-level this operation is not available in most high-level programming languages. Thus, when the search algorithm is presented, it is coded as in Figure 1. In this procedure two two-way comparisons are used per visited node although it is claimed that one three-way comparison is made. However, so far the author has never observed a compiler which actually translates these two comparisons into one three-way comparison. Thus, when the search procedure is actually run, two comparisons are made per visited node in the tree.

Since binary tree searching is a common operation in computer programs, some care to optimize this operation seems worthwhile. A well-known technique to optimize the code is to use a *sentinel* [4, 6, 7]. In a binary search tree the sentinel is an extra node which is

```
PROCEDURE StandardSearch (x: Data; T: NodePtr): NodePtr;
BEGIN
    WHILE (T # NIL) & (T↑.key # x) DO
        IF x < T↑.key THEN
            T := T↑.left;
        ELSE
            T := T↑.right;
        END;
    END;
    RETURN T;
END StandardSearch;
```

Figure 1: *The search procedure with three-way comparisons. The code is given in MODULA-2.*

the common child of all leaves in the tree. By putting the searched value in the sentinel before the search starts we can guarantee that the value will always be found either in the tree or in the sentinel. If the algorithm terminates in the sentinel the search has been unsuccessful. The advantage of this algorithm compared to the original one is that we do not have to test for empty pointers. Thus the termination condition of the algorithm can be made simpler. The search algorithm using a sentinel is given in Figure 2.

In this paper we discuss another way of optimizing the search procedure. We present an algorithm for searching in a binary search tree using two-way comparisons only. The algorithm is very simple and may scarcely be called new (in [4] it is observed that binary search in an array, which can be made efficiently with two-way comparisons, corresponds to searching in an implicit tree). However, it seems that this way of improving binary tree search has been overlooked. With this algorithm we obtain the same simple termination condition as when using a sentinel but instead of removing the test for empty pointers we remove the extra comparison made at each node in the tree. In the likely case that element-comparisons are more expensive than pointer operations, our improvement is more significant than the improvement caused by using a sentinel. This makes it possible to code an efficient (in terms of the number of comparisons used) search procedure in

```
PROCEDURE SentinelSearch (x: Data; T: NodePtr): NodePtr;
BEGIN
    sentinel↑.key := x;
    WHILE T↑.key # x DO
        IF x < T↑.key THEN
            T := T↑.left;
        ELSE
            T := T↑.right;
        END;
    END;
    IF T # sentinel THEN
        RETURN T
    ELSE
        RETURN NIL;
    END;
END SentinelSearch;
```

Figure 2: *Searching with sentinel. The node* sentinel *is assumed to be the child of each leaf in the tree.*

high-level languages independently of the optimizing capability of the compiler employed. We also give experimental results to indicate the speedup that is achieved by the presented algorithm.

The Improved Algorithm

The number of two-way comparisons required in binary tree search may be decreased by a simple modification of the procedure in Figure 1. The price we have to pay is some extra pointer assignments.

Theorem 1 *A node in a binary search tree with height h can be found in $h + 1$ two-way comparisons.*

Proof: We use a global pointer *candidate* and the following algorithm: At each node we make one two-way comparison to decide whether the searched value is smaller than the node's value or not. If the searched value is smaller we turn left, otherwise we let

```
PROCEDURE TwoWaySearch (x: Data; T: NodePtr): NodePtr;
VAR candidate: NodePtr;
BEGIN
    candidate := NIL;
    WHILE T # NIL DO
        IF x < T↑.key THEN
            T := T↑.left;
        ELSE
            candidate := T;
            T := T↑.right;
        END;
    END;
    IF (candidate # NIL) & (candidate↑.key = x) THEN
        RETURN candidate
    ELSE
        RETURN NIL;
    END;
END TwoWaySearch;
```

Figure 3: *Efficient searching with two-way comparisons.*

candidate point at the current node and turn right. When an empty node is found we examine *candidate*. The search has been successful if and only if *candidate* points to a node containing the searched value.

When the searched node is reached the algorithm does not terminate but continues in the right subtree. Since all elements in that subtree are larger than the one searched for, the algorithm will make no more right turns after passing that node. Thus if the element searched for is to be found in the tree it has to be in the node where the last right turn was made, which is the node referred to by *candidate*.

The maximal number of comparisons required by this algorithm equals the height of the tree plus one, the extra comparison being made when testing the value pointed to by *candidate*. The resulting algorithm is illustrated in Figure 3. □

The search algorithm can also be implemented recursively and it can be used in insert and delete procedures. As an example we give a recursive deletion algorithm in Figure 4.

```
PROCEDURE Delete (x: Data; VAR T: NodePtr);
VAR temp: NodePtr;
BEGIN
    IF T # NIL THEN
        IF x < T↑.key THEN
            IF T↑.left # NIL THEN
                Delete (x, T↑.left)
            ELSIF (candidate # NIL) & (candidate↑.key = x) THEN
                candidate↑.key := T↑.key;
                temp := T;
                T := T↑.right;
                DISPOSE (temp);
            END
        ELSIF T↑.right # NIL THEN
            candidate := T;
            Delete (x, T↑.right);
        ELSIF x = T↑.key THEN
            temp := T;
            T := T↑.left;
            DISPOSE (temp);
        END
    END
END Delete;
```

Figure 4: *Recursive deletion with two-way comparisons. The global pointer candidate is assumed to be initialized to NIL.*

Usually deletion is implemented as two procedures, one to find the node to be deleted and one to find an external node that can replace the deleted one. When using two-way comparisons we only need one procedure. This is because the search will always continue down to the bottom of the tree. The node to be deleted is pointed to by the global pointer and the last visited node during the search is the one to replace it with.

Experimental Results

In order to illustrate the speedup that may be achieved by the described algorithm, some experimental results are given. The procedures in Figures 1, 2, and 3 were run on a

Sun 3/180 using the Sun Modula-2 2.2 compiler with maximum optimization. The same (corresponding) procedures were also run on a VAX/780 with the Digital Ada-compiler and on a IBM PS/2 55X (with math co-processor) with Turbo Pascal 5.5.

To measure the cost of successful search the following experiment was made:

1. A tree containing 5000 elements (of type REAL when running MODULA-2, DOUBLE in Turbo-Pascal, and DIGITS(18) in Ada) is constructed and the elements in the tree are copied into an array.

2. The timer is started.

3. 100 000 calls of the search procedure are made; all elements in the array are searched for the same number of times.

4. The timer is stopped.

5. Step 1 to 4 are repeated five times and the average and standard deviation of the measured CPU-times are computed.

To measure the cost of unsuccessful search the same procedure was followed, except that the ith element in the array is chosen as the average between the ith and $(i+1)$th element in the tree, and the last element in the array is made larger than all elements in the tree.

The measured CPU-times are given in Table 1. All standard deviations were less than 5 percent for random trees and less than 1 percent for perfectly balanced trees.

As can be seen in the table, our experiments indicate a significant speedup (26 - 44 percent) by the use of two-way comparisons. As a matter of fact, using the new search algorithm helps more (on average) than balancing the tree! Note also that the use of a sentinel does not seem to give any particular speedup.

The experiment above was also made for other sizes of the trees, resulting in the same speedup.

tree generated from random insertions

compiler	successful search			unsuccessful search		
	standard	sentinel	two-way	standard	sentinel	two-way
Sun Modula-2	180	169	122	202	185	117
Digital Ada	4646	4826	2886	5125	5442	2882
Turbo-Pascal	518	510	364	572	577	365

perfectly balanced tree

compiler	successful search			unsuccessful search		
	standard	sentinel	two-way	standard	sentinel	two-way
Sun Modula-2	136	130	98	158	146	93
Digital Ada	3504	3666	2307	3962	4263	2312
Turbo-Pascal	400	397	295	454	464	295

Table 1: *Average search time (in microseconds) for searching in a tree containing 5000 elements.*

Comments

The algorithm presented here allows us to code efficient search procedures in programming languages where only two-way comparisons are available.

The search algorithm may easily be generalized to work also for multidimensional search trees, or k-d-trees [2]. In that case we need as many *candidate* pointers as the number of dimensions.

Acknowledgements

I would like to thank Dr. Svante Carlsson, Dr. Ola Peterson, Christian Collberg, and the referees for valuable comments on this paper.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, Reading, Massachusetts, 1983.

[2] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[3] G. H. Gonnet. *Handbook of Algorithms and Data Structures*. Addison-Wesley, Reading, Massachusetts, 1983. ISBN 0-201-0023-7.

[4] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, Reading, Massachusetts, 1973. ISBN 0-201-03803-X.

[5] E. M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice-Hall, Englewood Cliffs, New Jersey, 1977. ISBN 0-13-152447-X.

[6] R. Sedgewick. *Algorithms*. Addison-Wesley, Reading, Massachusetts, 1988. ISBN 0-201-06673-4.

[7] N. Wirth. *Algorithms and Data Structures*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986. ISBN 0-13-022005-1.