Using Binary Search On A Linked List

Firooz Khosraviyani
Department of Mathematics and Computer Science
The University of Texas of the Permian Basin
4901 East University Boulevard
ODESSA, TX 79762-8301

Abstract

In this article a variation of binary search applicable to a linked list structure is examined. There are no additional data structure properties imposed on the list; that is the list may be singly or multiply linked, but it is not required that the list be represented as a binary tree. The process can be used with improved efficiency in place of linear search on a linked list where introducing additional data structures complexity on the list is not desirable.

This procedure seems appropriate for inclusion in data structures courses where searching and sorting are discussed. The texts on data structures cover the binary search on a fixed length sequence, but the general consensus is that there is no advantage in trying to implement the binary search process on linked lists. One possible exception in this regard, where no details are provided, is [2, page 147].

1 Introduction

All the discussions in this article require sorted lists, and this is assumed without further reference.

The binary search principle requires and is naturally applicable to binary tree structures and structures which very easily imitate the mathematical structure of a binary tree. Natural candidates for the latter are fixed length record lists with contiguous storage allocation which are normally represented by linear array structures or finite sequences. With finite sequences the binary search is effectively achieved through a simple arithmetic calculation which detects the middle of the list and hence makes it possible to partition the list into two (nearly) equal halves, permitting a binary decision tree construct. In a simple linked list we do not have the ability to easily detect the middle of the list, hence we are forced to use a sequential search or restructure the list into a binary tree where the underlying structure of the list, the binary tree, permits binary search. If care is not taken, especially if many insertions and deletions can occur, in the worst case the binary tree created would become a linear list! But algorithms are available that help to make complete or approximately complete and hence balanced or nearly balanced binary trees which optimize the search process (for details and references see [1,3]).

As the remainder of this article will provide, there is a middle ground between:

- using the simple linked list with sequential search which makes the search, hence insertions and deletions, slow for large lists;

and

- using the binary tree structure of the linked list, that is when the binary tree structure is not required for other purposes, and hence may not be desirable because of the overhead introduced to manage the tree structure and to keep it optimized.

2 Algorithm Design

To enable a binary search principle to be applied to a list, we need to determine the middle of the list. With a linked list this is not at all trivial. The only way available is to begin from one end of the list and traverse it, sequentially, half way through. Therefore we have a sequential method!

Two separate tasks contribute to the time required for a search on a linked list:

1. pointer assignments, indicating the search position in the list; and

2. comparisons, made to indicate the status of search.

There is no remedy available to speed up the pointer assignment process, other than restructuring the list as a binary tree. But the comparison process can be sped up. This is achieved by reducing the more expensive, in terms of execution time, key field comparisons and replacing them with the less expensive loop structure comparisons for a counter controlled loop. This loop provides a fast and convenient way of traveling half way through the list and then performing the key field comparison for this (near the) middle record. This becomes more apparent with the discussion in the next paragraph.

To perform binary search on a fixed length sequence, pointers to (indices of) the two ends of a list, or sublist thereof, are kept which not only indicate the two ends of the list but also provide information as to the number of records in the list and the (approximate) middle of the list. The equivalent of this for a linked list amounts to the maintenance of a length counter and two pointers pointing to the head and tail of the list. Then the (approximate) middle of the list under search can be obtained by using a counter controlled loop. The overhead caused by this technique requires tail pointer and length counter maintenance operations, which are of constant time for each deletion or insertion.

The *Binary Search* algorithm presented here makes the above modifications to the usual binary search used for a fixed length sequence. The algorithm uses sequential search to complete the search on small enough sublists, in this case sublists of seven or less records. The sublist to apply the binary search step to can be as small as two in length, but applying linear search is essential to avoid extra overhead to the search process which could defeat the purpose; after all this strategy is recommended even if the binary search process could run into completion.

3 The Algorithm

Algorithm *Binary Search*

Input:

List(·): a linked list; each record in the list, among other possible fields, has key and link fields denoted by Key(·) and Link(·) indicating the field name and the record they belong to;

Head, Tail: head and tail pointers to the list List(·);

n: the number of records in List(·);

Key: the key of the record of interest.

Output:

Found: a Boolean variable indicating if the record of interest is in the list;

Ptr: a pointer pointing to the record of interest if the record was found or pointing to the record whose key is the predecessor of the one being searched for if the record was not in the list.

Local variables:

Left, Mid, Right: pointer type variables pointing to the two ends and middle of the sublist under search;

m: an integer variable used to keep the number of records in the sublist under search.

k: an integer variable indicating the minimum length of sublist on which binary search is to be performed.

```
BEGIN ( Binary Search )
$m := n$                    (Make the entire list the
                             sublist under search.)
Left := Head
Right := Tail
k := 7                      (This check value, $k$, for
                             the length $m$ of the sublist
                             under binary search must
                             be 2 or greater.)
WHILE $m > k$ DO            (The sublist
                             is long enough, so perform
                             a binary search step.)
  $m:=[m/2]$                (integer part of $m/2$)
  Mid:=Left
  FOR i:=1 TO $m$ DO
    Mid:=Link(Mid)          (One reason to do sequen-
                             tial search on small sublists
                             is that we may run into
                             difficulties for the value of
                             $m$=1 here.)
  END FOR
  IF Key < Key(Mid)
    THEN Right:=Mid
    ELSE Left:=Mid
  END IF
END WHILE                   (end of binary search step)
Sequential Search(List, Left, Right, Key; Found, Ptr)
    (Input list; Output list) for the routine.
END ( Binary Search )
```

4 Analysis of the Algorithm

The iteration of the WHILE loop is of order $O(\log n)$. This makes the key field comparisons of order $O(\log n)$. Although this makes the number of key field comparisons as efficient as is expected from the binary search, the complexity of the algorithm introduces another set of comparisons, namely the FOR loop control variable comparisons. On each iteration of the WHILE loop, there are m iterations of the FOR loop. This results in a total of $n-1$ iterations of the FOR loop, if $k = 1$, as:

$$[n/2] + [n/4] + \cdots + 1 = (2[n/2] - 1)/(2-1) = n - 1.$$

Therefore, the control variable comparisons of the FOR loop is of order $O(n)$. This also provides that there are $O(n)$ pointer assignments in the search process since there are $O(n)$ pointer assignments within the FOR loop and $O(\log n)$ pointer assignments outside the FOR loop.

5 Experimental Results

A simple pseudo random number generator is used to produce a list of integers. In an effort to minimize outside interference with the search/sort process, different sized sublists of this integer list are fed into sorting modules utilizing sequential, binary, and binary tree search routines and the resulting sort execution times are summarized in the tables in Appendix B.

The linked lists were constructed using Modula-2/Pascal pointers. The integers are 32-bit integers for microcomputer and mini/mainframe implementations, where 46-bit integers were used on the super computer implementation. For the virtual memory machine (DEC/VAX), the sort times reported in table 2 in Appendix B were obtained with few or no memory page swapping (page faults) as a result of enlarging the working space for the program.

It should be noted that the data provided by the linear congruential pseudo random number generator is approximately uniformly distributed. This provides for the average case for sequential search and, at the same time, for the best case for binary search tree algorithms. The binary search algorithm, similar to its implementation on fixed length record sequences, performs its optimum in all situations.

There are no tree balancing algorithms employed to prevent the binary search tree from degrading to a linear list, as this is assured by the supplied data. As such, the execution times reported are for an optimized situation for the binary search tree algorithm. In practice, a more realistic comparison of binary search tree and binary search algorithms would require applying a tree balancing algorithm to the binary search tree, which increases the search/sort times for the implementation, though it prevents the occurrence of the worst case.

6 Conclusion

The binary search tree implementation of a linked list provides a search time of order $O(\log n)$, whereas the binary and sequential searches of a linked list yield a search time of order $O(n)$.

A brief observation of the execution times from tables 1 and 3 in Appendix B shows that employing the binary search process improves the sort times by about 58% over sequential search implementation. The theoretical discussions indicate that this improvement rate is an ascending function of the list size. However, this expectation is not observed on the results obtained on the virtual memory machine (DEC/VAX), although there were no page swapping to corrupt the system reported CPU execution times for the sorting process.

Therefore, the binary search tree implementation of a linked list should be used whenever possible. But, if the binary search tree implementation of a linked list is not used, and the list is not sufficiently small, the binary search of a linked list provides substantial improvement over the sequential search. This improvement comes with almost no overhead cost, in memory usage and complexity of the algorithm.

A Auxiliary Algorithms

For ease of reference and to make this article self contained, sequential and binary tree search algorithms are included. The algorithms are coded similarly and provide compatible interface. Also, the binary search algorithm, as presented earlier in this article, calls upon sequential search to complete the search process.

Algorithm *Sequential Search*

Input:

List(·): a linked list; each record in the list, among other possible fields, has key and link fields denoted by Key(·) and Link(·) indicating the field name and the record they belong to;

Head, Tail: head and tail pointers to the list List(·);

Key: the key of the record of interest.

Output:

Found: a Boolean variable indicating if the record of interest is in the list;

Ptr: a pointer pointing to the record of interest if the record was found or pointing to the record whose key is the predecessor of the one being searched for if the record was not in the list.

Local variables:

Prev: pointer type variable pointing to the record in the list preceding the record under examination;

EndSrch: a Boolean variable indicating the status of search.

```
BEGIN ( Sequential Search )
Found := False          (Assume record not in list.)
EndSrch := False        (Search is not over yet!)
Prev := Λ               (There is no previous node.)
Ptr := Head             (Start with first node of list.)
WHILE NOT EndSrch DO
In Case
  Key > Key(Ptr) : (not found yet)
                   Prev := Ptr
                   IF Ptr = tail
                      THEN EndSrch := True
                      ELSE Ptr := Link(Ptr)
                   END IF
  Key = Key(Ptr) : Found := True
                   EndSrch := True
  Key < Key(Ptr) : (not in list)
                   Ptr := Prev
                   EndSrch := True
END Case
END WHILE             (end of search)
END ( Sequential Search )
```

Algorithm *Tree Search*

Input:

List(·): a linked binary tree structured list; each record in the list, among other possible fields, has key and left and right link fields denoted by Key(·), LLink(·), and RLink(·) indicating the field name and the record they belong to;

Root: pointer to the root of the tree List(·);

Key: the key of the record of interest.

Output:

Found: a Boolean variable indicating if the record of interest is in the list;

Ptr: a pointer pointing to the record of interest if the record was found or pointing to the record whose key is the predecessor of the one being searched for if the record was not in the list.

Local variables:

EndSrch: a Boolean variable indicating the status of search.

```
BEGIN ( Tree Search )
Found := False    (Assume record not in list.)
EndSrch := False  (Search is not over yet!)
Ptr := Root       (Start with first node of list.)
WHILE NOT EndSrch DO
In Case
  Key > Key(Ptr) :
        IF RLink(Ptr) = Λ
           THEN EndSrch := True    (not in list)
           ELSE (not found yet, look to the right)
                  Ptr := RLink(Ptr)
        END IF
  Key = Key(Ptr) :
        Found := True
        EndSrch := True
  Key < Key(Ptr) :
        IF LLink(Ptr) = Λ
           THEN EndSrch := True    (not in list)
           ELSE (not found yet, look to the left)
                  Ptr := LLink(Ptr)
        END IF
END Case
END WHILE       (end of search)
END ( Tree Search )
```

B Tables

These tables provide the sort times using the indicated search methods for lists of indicated lengths. The time is reported in the format

"Hours:Minutes:Seconds.Fraction of a second",

where the insignificant parts may be omitted, and represents the average when multiple runs were available.

List Length	Search Method		
	Sequential	Binary	Tree
250	1.33	0.56	0.24
1000	20.41	6.92	1.15
2500	2:04.73	40.59	3.22
10000	33:20.45	10:27.62	15.01

Table 1: Sort Times — PC/AT compatible

List Length	Search Method		
	Sequential	Binary	Tree
250	0.39	0.21	0.10
1000	5.50	2.63	0.44
2500	35.43	15.92	1.18
10000	9:37.26	5:02.81	5.42
25000	1:03:50.01	39:36.80	14.37
100000	18:13:39.09	12:13:31.17	4:48.63

Table 2: Sort Times — DEC/VAX 8200

List Length	Search Method		
	Sequential	Binary	Tree
250	0.055	0.023	0.011
1000	0.773	0.286	0.053
2500	4.780	1.666	0.150
10000	1:14.941	25.848	0.678
25000	7:51.801	2:39.291	1.871
100000	2:04:34.804	42:46.139	8.332

Table 3: Sort Times — Cray/XMP

References

[1] *The Art of Computer Programming, Volume 3: Searching and Sorting*, by D.E. Knuth, Addison-Wesley, Reading, Massachusetts, 1973.

[2] *Principles of Data Structures and Algorithms with Pascal*, by Robert R. Korfhage and Norman E. Gibbs, Wm. C. Brown Publishers, Dubuque, Iowa, 1987.

[3] *Data Structure Techniques*, by Thomas A. Standish, Addison-Wesley, Reading, Massachusetts, 1980.