Comparison of search data structures and their performance evaluation with detailed analysis

S.V.SRIDHAR, P.RAVINDER RAO, E.PRIYADARSHINI,N.VAMSI KRISHNA

ABSTRACT:

In the field of computer science and information technology , and many other related areas where large volumes of data are present we need to do many operations on those sets of data which are part of processing , analyzing and producing results and related conclusions . so in this paper we present one of the most commonly used operations – SEARCHING in the fields related to information processing. Along with a detailed study on sequential and binary search we also present performance analysis and a detailed technical view on the method of operation of the above said searching methods along with programmatic approach

Index Terms— Minimum 7 keywords are mandatory, Keywords should closely reflect the topic and should optimally characterize the paper. Use about four key words or phrases in alphabetical order, separated by commas.

I. INTRODUCTION:

In computer science, a **search data structure** is any data structure that allows the efficient retrieval of specific items from a set of items, such as a specific record from a database or searching for the presence of an element among a list of elements.

23 14 8 55 71 31 11 43 12

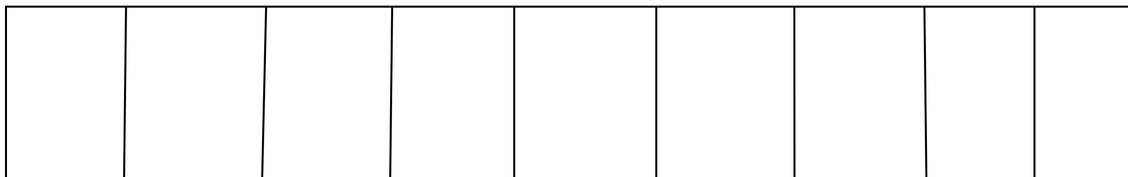


Figure 1

The above picture represents a general list of elements that are saved in memory. Assume that in a particular instance we need to know whether element 71 is present in the above list or not, if present we need to know the position of the element among the above, thereby enabling to access the element if needed , because in the field of computer science a storage operation always occurs by presence of a memory address and a mechanism to access the value present at that location and also a mechanism and store back the value after usage if needed. In general the “element to be searched” is referenced as KEY.

II. METHEDOLOGY:

So here we present the two most commonly used search procedures

1) SEQUENTIAL or LINEAR SEARCH
2) BINARY SEARCH

The first method is the sequential search which is relatively easy to understand and implement. it follows the general procedure that we use in our daily lives. First of all, knowing the KEY (target element) and comparing it with the first element in the list. if it is equal to the KEY then we will end the search as the KEY is present and also we came to know where it is present (in this case first position) , if not we will next move to the second and compare , this simple procedure follows till either KEY is found or END of list occurs.

The following fragment of programming code written in C gives a view of the methodology which is theoretically explained in the above lines

```
for ( element= 0 ; element < sizeoflist ; element++ )

  {

    if ( array[element] == search )     /* if required element found */

    {

      printf("target element is present at location %d.\n",       element+1);

    break;

    }

  }

  if ( element == endofelements)

    printf("target element is not present in list of elements");



}
```

Now we will look at the merits and demerits of the above procedure. As already said the great advantage of the above procedure is it is easy to understand and very easy to implement. And coming to demerits assume a situation where the size of elements among which target is to be searched is quite large say "n" , then if target is present in the last position then the procedure has to make a large number of comparisons i.e. "n" comparisons. In this case it consumes many CPU cycles and processing time for doing those unsuccessful (n-1) comparisons. This in turn has a drastic effect on the efficiency and overall thoughput and performance bottlenecks.

So from the big disadvantage which is presented in above linear search, a new search technique is developed which concentrates mainly on overcoming the above disadvantage, which is named as BINARY SEARCH.

in mathematics terminology the word BINARY refers to TWO. here we pre - assume that the list of elements to be in sorted order either ascending or descending before the procedure starts which is a compulsory pre-requisite for this procedure , because this procedure always tries to reduce the size of elements to be next searched to be nearly equal to half when compared to previous step .this situation occurs here because this procedure relies purely on finding out whether in reference with the middle element the left or right sides of the list is either greater or lower when compared with the KEY , there by confining the search procedure to any one-half , because as the list is already sorted there is no chance of presence of KEY among the wrong side (left or right) that means we cannot expect KEY to present below the values which are lower to it and also we cannot expect KEY to present above the values which are greater than it . The best case of this procedure is that the middle element itself is KEY.

The following fragment of programming code written in C gives a view of the methodology which is theoretically explained in the above lines (assumpted sort order: ascending)

```
first = 0;

  last = n - 1;

  middle = (first+last)/2;


  while( first <= last )

  {

    if ( list[middle] < search )

      first = middle + 1;

    else if ( list[middle] == search )

    {

      printf("target found at location %d.\n", search, middle+1);

      break;

    }

    else
```

```
      last = middle - 1;


    middle = (first + last)/2;

  }

  if ( first > last )

    printf("target element is not present in the list.\n", search);



  return 0;
```

the below image presents the pictorial view of the procedure where target is assumed to be 76 .

in first phase low = 0 high = 15 , so mid will be 8 , after this the search progresses by confining to either one side depending on comparisons and there after proceeding recursively till target id found or end of elements .

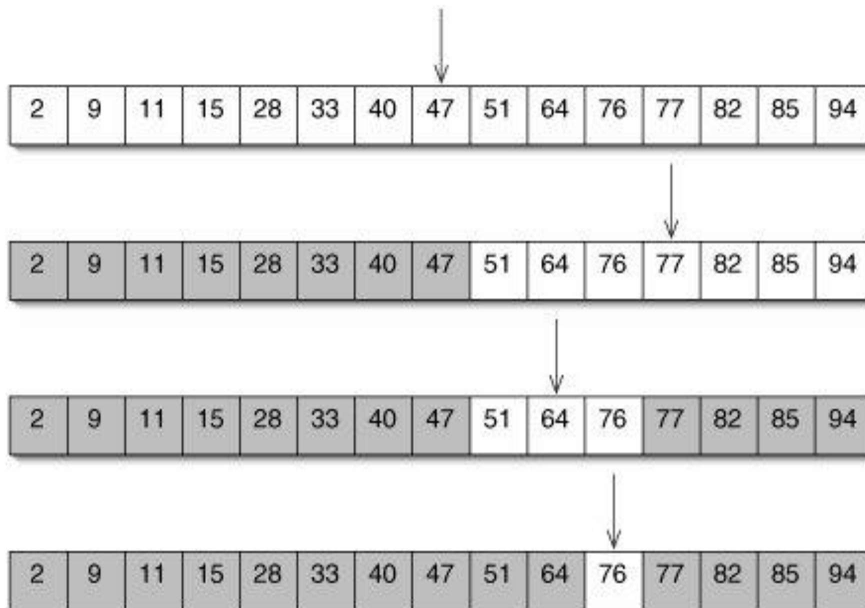


Figure 2

III. PERFORMANCE AND COMPLEXITY ANALYSIS:

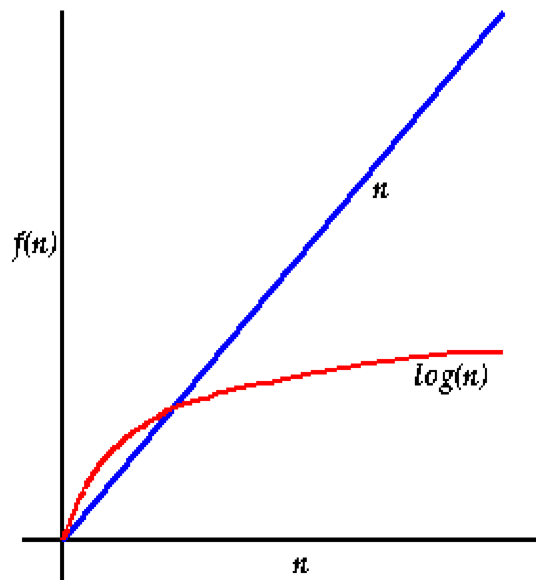
Each step of the algorithm divides the block of items being searched in half. We can divide a set of ***n*** items in half at most $\log_2 n$ times.

Thus the running time of a binary search is proportional to **log *n*** and we say this is an ***O* (log *n*)** algorithm.

Binary search requires a more complex program than our original search and thus for *small* ***n*** it may run slower than the simple linear search. However, for large ***n***,

$$\lim_{n \to \infty} \frac{\log n}{n} = 0$$

Thus at large ***n***, **log *n*** is *much* smaller than ***n***, consequently an ***O*(log *n*)** algorithm is *much* faster than an ***O*(*n*)** one.



Plot of ***n*** and **log *n*** *vs.* ***n***.

Given a list of **length n** for a **sequential** (linear) **search Algorithm** we find that the **number of comparisons** is **n**, in the case *where the item to be found is not in the list*. We use the notation **O (n)** to describe this time complexity. It reads "**Order of n**". This is called the **Big-O notation.** For a list of length 1000, we would make 1000 comparisons. If we double the list length we double the number of Comparisons. But sometimes (**on average**) we are likely to find the

item in the list. Sometimes near the beginning, sometimes near the end. On average, we will have to search **half of the list**. So, on average we would make **O(n / 2)** comparisons. This is called the **average case complexity**. We have seen, that if the item is not in the list we make n comparisons, this is called the **worst case complexity**. Note that we still regard the complexity of a linear search algorithm as O (n) despite the fact that on average we find the item in n/2 comparisons. This is because as n grows very large, the difference between n and n/2 may will become less significant i.e. they are of the same order (e.g. 2 billion versus 1 billion). **When comparing algorithms it is important to know both the average and worst case complexity.** In **binary search algorithm** where the number of comparisons is substantially less. It is **O(log n**), using base-2 logs. Thus if the list length is 1000, using binary search we will find the item in at worst 10 comparisons (as opposed to 1000 for linear search). But consider a list length of **1 billion**. With a binary search algorithm we will find an item with **at worst 30 comparisons**. It is obvious that the binary search algorithm is far superior from a time complexity viewpoint to a linear search algorithm.

There is a major drawback with the binary search algorithm. It **can only be used** when the list is in a particular order i.e. **when the list is sorted**.

It is the method we use to search for a number in phone directory. We open the directory in the middle, compare the entry there with what we are looking for, and depending on the outcome we know if the number is in the lower half of the directory, the upper half or on the page we have opened. We thus eliminate half of the directory with 1 comparison!!

We repeat this procedure until we find the number or know it is not present. With each comparison we reduce the list length (number of pages of directory) by a factor of 2. Thus with 10,000 pages after 1 comparison we have 5,000 left to search, then 2500, then 1250, then 625 etc.

The maximum number of comparison is log(10,000) (to base 2) i.e. the number of times we can divide 2 into 10,000.

We can only use this method, because the list is sorted.

Consider a list **A** of **n** integers, sorted in increasing order. The binary search algorithm to search **A** for an element **e** takes the form:

while not finished and found == false do

begin

compute middle of list

if e == middle item then found = true

else if e < middle item search lower half

else if e > middle item search upper half end

IV. ADDITIONAL WORK

Our discussion of the data structures is on a very abstract conceptual level, and we have ignored many problems that arise in actual applications of range searching. In this section we briefly examine some of those problems and the solutions that have been proposed to handle them. We went through and discussed only static and fixed kind of implementation for these searching methodologies.

V.CONCLUSION

In the process of writing this paper we went through and studied a number of data structures and methodologies for the range searching problem. Knuth was able to write that "no really nice data structures seem to exist" for the problem of range searching. In this paper we have tried to show that this situation has changed in the interim, and that these changes can have a substantial impact on both the theory and practice of searching methodologies. More precise research and application leads to development and implementation of more effective and reliable searching methodologies.

REFERENCES:

Adamson Iain T.	Data Structures and Algorithms: A First Course
Aho Alfred V., et al.	Data Structures and Algorithms
Aho Alfred V., et al.	The Design and Analysis of Computer Algorithms
Bucknall Julian	The Tomes of Delphi: Algorithms and Data Structures
Cormen Thomas H.,	Introduction to Algorithms

Drozdek Adam	Data Structures and Algorithms in C++
Flamig Bryan	Practical Data Structures in C++
Ford William, **Topp** William R.	Data Structures with C++ Using STL
Gilberg Richard F., **Forouzan** Behrouz A.	Data Structures: A Pseudo code Approach with C++
Goodrich Michael T., **Tamassia** Roberto	Data Structures and Algorithms in Java
Harrington Jan L.	Object-Oriented C++ Data Structures for Real Programmers