Volume 6, Issue 5, May 2016 ISSN: 2277 128X

International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

An Indexed Method for Improving the Efficiency of the Binary Search Algorithm

Ekenta Elizabeth Odokuma
Department of Computer Science,
University of Port Harcourt,
Port Harcourt, Nigeria

Olumide O. Owolabi
Department of Computer Science.
University of Abuja,
Abuja, Nigeria

Abstract: A new method for retrieving keys in a table of strings is implemented. The keys of the table are sorted according to first letter and an index list based on the first letter of the strings is built. With reference to this list, binary search is performed on only the list of strings with the same first letter. We conducted run time experiments to compare the performance of the traditional Binary Search (BS) and Indexed Binary Search (IBS). Results show that our approach reduces the search times and number of comparisons in binary search by an average of 30% and 46% respectively.

Keywords: Binary Search, String Array, Indexing, Search Algorithms

I. INTRODUCTION

Binary search is about the most efficient method for the search operation in an ordered sequentially mapped list (i.e. the elements are stored in a sorted order [1]. Binary search begins by comparing the middle element of the ordered list to the search key. This effectively splits the list into two halves - the lower half and the upper half. Since the list is ordered, the keys of each element in the lower half is smaller than the keys in elements of the upper half. If the search key equals the middle element, then we have found the desired element and the search is terminated. However if the search key is smaller than the key of the middle element, then the middle element of the lower half is examined otherwise the search key must be greater than the key of the middle element, and the middle element of the upper half is examined. The list is continually divided in this fashion until either the desired element is found or the entire list is searched. [2]

A. Searching

Searching a list for a particular item is a common task. In real applications, the list items are often records (e.g. student records), and the list is implemented as an array of objects. The goal is to find a particular record, identified by name or an ID number such as a student's matriculation number. Finding the matching list element provides access to target information in the record such as the student's address, phone number, payment detail, results, medical record, etc. For long lists and tasks like searching, that are repeated frequently, the choice among alternative algorithms becomes important because of efficiency considerations. Linear and binary search are two common algorithms used for this purpose [3]. Linear search has a complexity of O(n) while Binary search has a complexity of $O(\log_2 n)$ where n is the number of elements in the data set [4]

B. Linear Search

The simplest method for searching a list of elements is to check the first item, then the second item, and so on until we find the target item or reach the end of the list. This is the behavior of the linear (or sequential) search algorithm. The number of steps the algorithm will require from start to finish is a measure of its efficiency. The exact number of steps will depends on the input data. For the linear search algorithm, the number of steps depends on whether the target is in the list, and if so, where in the list, as well as on the length of the list [3].

C. Binary Search

If the list is sorted, then there is information about the location of the target even from comparisons that don't find it. They reveal whether the target is before or after the current position in the list, and that information can be used to narrow the search. Binary search begins by checking the middle item in the list. If it is not the target and the target is smaller than the middle item, the target must be in the first half of the list. If the target is larger than the middle item, the target must be in the last half of the list. Thus, one unsuccessful comparison reduces the number of items left to check by half! The search continues by checking the middle item in the remaining half of the list. If it's not the target, the search narrows to the half of the remaining part of the list that the target could be in. The splitting process continues until the target is located or the remaining list consists of only one item. If that item is not the target, then it's not in the list. Figure 1 shows the pseudocode for Binary Search

© 2016, IJARCSSE All Rights Reserved

```
location = -1;
first = 0;
last = number of items minus 1;
 while ((number of items left to search >= 1) and
                    (target not found))
            middle = position of middle item, halfway
                    between first and last
            if (item at middle position is target)
                    target found
            else
                    if (target < middle item)
                            search lower half of array next
                            last = middle - 1;
                    else
                            search upper half of array next
                            first = middle + 1;
end while
 if (target found) (i.e., middle item == target)
            location = position of target in array
                            (i.e., middle)
 return location as the result
```

Figure 1: Pseudocode for the binary search algorithm

D. Motivation

Searches of elements in sorted arrays are mostly done using binary search because it has been shown that it performs much better than linear search and some other search algorithms. Searches for identifier information in symbol tables during programming is one example that heavily rely on Binary search. Because of the importance of binary search in computer programming there is a need to improve its performance. Our approach is by reducing the number of comparison in the program thereby reducing the search time.

E. Related Work

Several authors have attempted to improve the performance of binary search. In [5], the authors attempted to eliminate the housekeeping instructions in Binary Search by arranging the elements of the string array in a suitable way. In [6], the input element is compared to the middle element in the manner of traditional binary search; it also compares it to the first and last elements of the data set. It does these with the aim of optimizing the worst case of the binary search algorithm.

F. Aim and Objectives

Our aim is to develop an algorithm that reduces the search time of the binary search algorithm by reducing the number of comparisons it does.

The objectives are as follows:

1. To develop a modification if the binary search algorithm that uses an index based on the first letters of the strings of the data set.
2. To perform run time experiments with different sets of data and compare the performance of Indexed Binary Search to the traditional Binary Search.

II. DESCRIPTION OF INDEXED BINARY SEARCH (IBS)

We can improve the performance of Binary search by introducing an index array. An index array is similar to a database index in that it lets us visit records by an ordering different from the physical order of the records in memory or on disk.

We propose an index of first letter for the strings in the ordered sequentially mapped list i.e. the string array. Indexed binary search begins by locating the index of the first and last string in the string array beginning with the first letter of the search string through the index array, then binary search is performed within these bounds

The steps taken by indexed Binary search for “fancy” in the array of fifteen strings is shown in figure 1.

A. The Pseudocode for the pre processing algorithm

ALGORITHM buildIndex

Description: Builds the first letter index list of the string array

Input: A string array: b, the size of b: m

Output: A 27 element array of integers (indexArray) that indicate the location of the first string that begin each letter (a-z) of the English alphabet. If there is no string beginning with a particular letter, the location of the next sting is indicated where such a string exist, otherwise the size of stringArray is indicated which means there are no more strings. The 27th element (indexArray[26]) contains the size of stringArray

```
Declare an array (d) of 27 elements and initialize all the elements to -1
char s ← '\n'
int i, j, ma, x, c
for i ← 0 to stringArraySize do
   char bfl ← b[i][0] //bfl = first character of current string
   if bfl > s do
         x ← bfl-97 //convert first letter to integer between 0 and 25
            d[x] = i
            j ← ma ← x
            s ← bfl
            while (d[j-1] = -1)
                        d[j-1] ← i
if ma != 25//where first letter of the last string is not z
            for c ← ma + 1 to 25
                        d[c] ← stringArraySize
d[26] ← stringArraySize  // the last element = size of string array
output index array to the file index.res
return
```

Index Array	
a	0
b	4
c	6
d	8
e	9
f	10
g	11
h	11
i	11
j	11
k	11
l	11
m	11
n	12
o	13
p	14
q	14
r	14
s	14
t	14
u	14
v	14
w	14
x	14
y	15
z	15
	15

0	Abandon
1	Able
2	About
3	Again
4	Before
5	Better
6	Carry
7	Could
8	Done
9	Every
10	Fancy
11	Mango
12	Nanny
13	Orange
14	Yellow

Figure 1: Showing the number of basic operations per line of Indexed Binary search

B. Pseudocode for Indexed Binary Search (IBS)and the number of primitive operations executed

The Indexed Binary search algorithm and the number of basic operation per line of the algorithm is shown in figure 2

Algorithm: IndexedBinarySearch (stringArray, indexArray, searchKey, begin index, end index)
Input: (1) a sorted array of strings (stringArray),
(2) a 27 element array of integers (indexArray)
(3) a key (searchKey)
Output: 1 if searchKey is found, -1 if not found

Line No.	Algorithm: Indexed Binary Search	No. of Basic ops
1	firstLetter ← searchKey[0]	2
2	**switch**(firstLetter){	1

© 2016, IJARCSSE All Rights Reserved

```
3       case (α):      // α = a or b or .. or z                 1
4           pos ← #   // # =   0,1,2..25 for α =                1
5       break          //       a,b,c..z respectively           1
6       default:                                                0
7           pos ← -2                                            1
8           break                                               1
9     } // end switch                                           0
10    if pos != -2 {                                            1
11      low ← indexArray[pos]                                   2
12      high ←indexArray[pos+1] – 1                             4
13      while low <= high do                                    1
14        int middle ← (low + high) / 2                         3
15        if (stringArray[middle] < searchKey)                  2
16          low ← middle + 1                                    2
17        else if (stringArray[middle] > searchKey)             2
18          high ← middle - 1                                   2
19        else                                                  0
20          return 1                                            1
21    // couldn't find the key                                  0
22          return -1                                           1
23          }                                                   0
                                                                29
```

Figure 2: Showing the steps required to locate the string “fancy”. Only one comparison required.

III. EXPERIMENTAL RESULTS

In the following sections we describe the experimental setup and the results obtained.

A. Description of the Data File

The data file used to test the programs is a file containing 25,500 words one word per line. The word distribution statistics for the data file is shown in table 1. Different sets of strings were randomly extracted from this file and used to test the programs.

B. The Program Modules

The program consists of the following modules

1. main
2. sortTheRand
3. readNameFile
4. readSortFile
5. prinList
6. timeall
7. buildIndex
8. searchIBS
9. binarySearch
10. countComparisons

C. Program Construction

C++ is the programming language used to develop these algorithms and the C++ standard Template Library (STL) containers: list and vector [7].

D. Data sizes used

To measure the effectiveness of indexed binary search, we performed run time experiments on 16 data sets randomly selected and sorted from the original data file of 25500 words (datastrings.dat). Data sets of 10, 50, 80, 100, 500, 800, 1000, 1500, 2000, 5000, 8000, 10000, 15000, 18000, 20000, 25000 were used.

E. Timing of experiments

The C++ clock() function was used in the function timeall() to determine the elapsed time for each of the different data sets using the two different algorithms.

Since we are more interested in the difference in search times instead of the search time itself, we performed 80 searches of all the strings in each of the 16 different data sets. This is because fewer searches produced 0 seconds for the datasets with strings less than 500 as search times for the algorithms. We then performed 10 runs of each experiment and took the average.

F. Running Time

Table 2 shows the running times of IBS and BS for the 16 different data sets while figure 4 clearly depicts the results.

© 2016, IJARCSSE All Rights Reserved

G. Comparison

We used the countCompare algorithm, (which is a modification of binary search that includes a counter), to count the number of comparisons done in binary search for IBS and BS and obtained the result shown in Table 3. Figure 4 is an X-Y scatter chart, depicting the results.

Table 1: Word distribution statistics for the data file.

First Letter	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**
Frequency	1774	1690	2547	1374	1061	1037	935	1129	1051	304	315	909	1501
Percent freq	6.96	6.63	9.99	5.39	4.16	4.07	3.67	4.43	4.12	1.19	1.24	3.56	5.89
First Letter	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
Frequency	545	509	1991	140	965	2808	1376	205	424	722	13	113	62
Percent freq	2.14	2	7.81	0.55	3.78	11.01	5.4	0.8	1.66	2.83	0.05	0.44	0.24
Total	**25,500**												

Table 2: Run times in seconds of Indexed Binary search and Binary Search

N	IBS(sec)	BS(sec)	% gain of IBS
10	0.05	0.02	-150.00
50	0.16	0.22	27.27
80	0.33	0.39	15.38
100	0.38	0.5	24.00
500	2.25	2.96	23.99
800	3.73	4.83	22.77
1000	4.78	6.15	22.28
1500	7.58	9.72	22.02
2000	13.78	25.44	45.83
5000	29.77	39.49	24.61
8000	47.4	57.67	17.81
10000	61.73	73.66	16.20
15000	91.45	113.69	19.56
18000	120.56	138.85	13.17
20000	136.82	159.34	14.13
25000	169.01	200.81	15.84

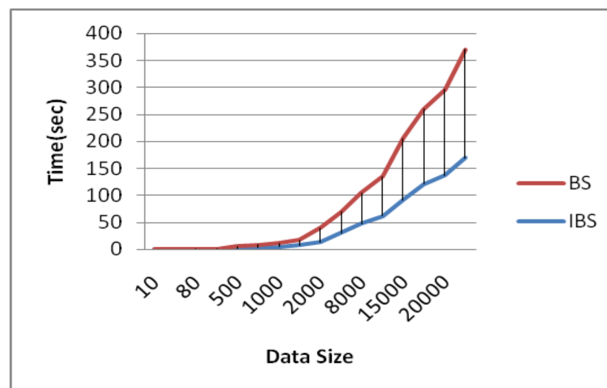

Figure 3: Showing the relative performance in terms of running time of IBS and BS with increase in the size of the data sets

Table 4: Characteristics of the test systems

	Processor	Ram size	HDD	OS
Machine I	Intel P4	1GB	250	Windows P
Machine II	Intel Core 3	4GB	500	Windows 8

H. Hardware Used

To show that the actual search time depends on the machine used, we ran the programs in two different machines. Table 4 shows the characteristics of the two different machines used for the experiment. Both systems were running Visual C++ under the Windows Operating System. Table 5 and Figure 5 shows the results obtained.

© 2016, IJARCSSE All Rights Reserved

Table 3: Showing the number of comparisons done in binary search for IBS and BS for the data sets, the searches are repeated 80 times each.

N	IBS	BS	Comparisons Saved	Percentage Comp saved
10	16	29	13	45
30	50	124	74	60
50	95	243	148	61
80	182	438	256	58
100	241	578	337	58
500	2052	3970	1918	48
800	3796	6952	3156	45
1000	5017	8936	3919	44
1500	8365	14348	5983	42
2000	11947	19783	7836	40
5000	35596	55775	20179	36
10000	79037	119466	40429	34
15000	124699	185067	60368	33
18000	152489	225324	72835	32
20000	171098	252771	81673	32
23000	199734	293057	93323	32
25000	218204	320016	101812	32
Average no. of comparisons saved				**43%**

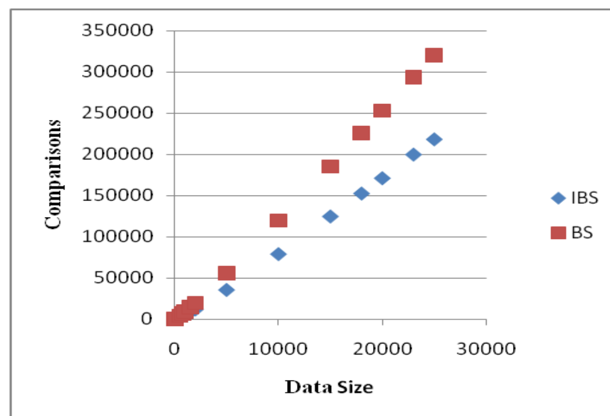

Figure 4: Showing a scatter plot of the comparisons done in binary search by IBS and BS for the 16 data sets.

Table 5: Showing search times for 5 different data sets using Intel Pentium 4 (32 bits) and a Core i3(64bits) computers

	P4 (32Bits)		CORE I3 (64Bits)	
	-IBS-	-BS-	-IBS-	-BS-
10	0.05	0.02	0.007	0.003
100	0.38	0.5	0.053	0.069
1000	4.78	6.15	0.693	0.891
10000	61.73	73.66	9.353	11.161
20000	136.82	159.34	20.92	24.364

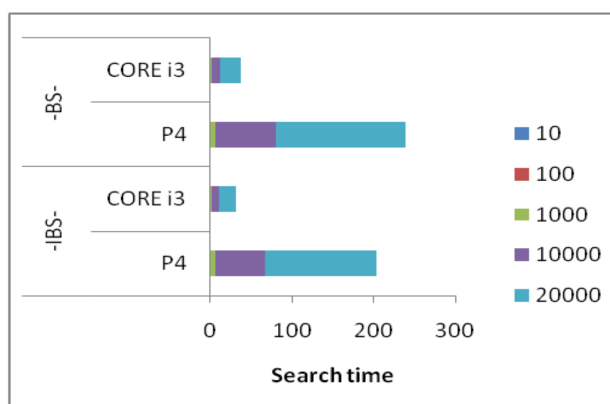

Figure 5: Chart Showing the difference in run times using 2 different machines

© 2016, IJARCSSE All Rights Reserved

IV. DISCUSSION OF RESULTS

The results in Table 2 show that for lower values of n (data size < 50), Binary Search performs better than Indexed Binary Search. This may be due to indexing overhead. For n > 50, Indexed Binary Search performs better than Binary Search.

The results in Table 2 and Figure 3 show that an average of 30% of the runtime is saved by using Indexed Binary Search instead of Binary Search.

The results in Table 3 and Figure 4 show that an average of 42% (30-60) comparisons was saved using Indexed Binary Search instead of Binary Search.

The results of Table 5 and Figures 5 show that the nature of hardware used influenced the actual runtime of these algorithms. The programs ran faster in the Core i3 system.

V. CONCLUSION

We have considered a variant of binary searching which reduces search time by reducing the number of comparisons done by Binary Search. Our algorithm require that we build an index of first letter which we can achieve in O(n) time. This algorithm is more suited for a static table, but may be useful in a dynamic table that does not require frequent revision since it does not require much time to build the index.

REFERENCES

[1] Deitel, H. M. and Deitel, P. J. (2014). *C++ how to program*. Patience hall. Upper saddle River. 9ed.

[2] Wikipedia contributors (2016b). "Binary search algorithm." *Wikipedia, The Free Encyclopedia*. Wikipedia, The Free Encyclopedia, 19 May. 2016. Web. 20 May. 2016

[3] Rodger, J. (2016). Searching Arrays: Algorithm and efficiency. http://www.cs.queensu.ca/home/CISC1011winter2015/webnotes/search.html.

[4] Knuth, D. E. (1975). *The art of Computer Programming Volume 3: Sorting and Searching*. Addison-Wesley, Reading MA (second printing).

[5] Merlili, D., Sprugnolis R. and Verri, M. C. (2002). Modified Binary Searching for Static Tables, Theoretical Computer Science 285 (1), 73–88

[6] Ankit R. C., Rishikesh M., Tanaya M. (2014) Modified Binary Search Algorithm. *International Journal of Applied Information Systems* Volume 7 (2), 37-40

[7] Wikipedia contributors(2016a) “Standard Template Library”. (2016, April 12). In *Wikipedia, The Free Encyclopedia*. Retrieved 23:44, May 20, 2016

© 2016, IJARCSSE All Rights Reserved