MODULE - II

Searching and Sorting

MODULE 2

Searching and Sorting

Module Description

This module introduces the problem of searching a list to find a particular entry. The discussion centres on two well-known algorithms: sequential search and binary search.

Most of this chapter assumes that the entire sort can be done in main memory, so that the number of elements is relatively small(less than a million).

Sorts that cannot be performed in main memory and must be done on disk or tape are also quite important. This type of sorting is known as external sorting and will be discussed in the second chapter of this module. It is assumed in our examples that the array contains only integers to simplify matters. At the same time, we have to understand that more complicated structures are possible.

Chapter Table of Contents

Chapter 2.1

Searching Techniques

Aim

To educate the students in searching and sorting techniques

Instructional Objectives

After completing this chapter, you should be able to:

- Explain searching and its types with its code snippet
- Describe sequential search using iterative and recursive
- Explain binary search
- Compare linear search and binary search

Learning Outcomes

At the end of this chapter, you are expected to:

- Elaborate searching techniques with example
- Compute time complexities for binary search and sequential search algorithm
- Outline the applications of linear and binary search
- Write code for iterative and recursive implementation for both the searching techniques

2.1.1 Introduction to Searching

This chapter focuses on how searching plays an important role in the concept of data structures. Searching helps to find if a particular element is part of a given list or not. In this chapter, we will focus on two types of searching techniques namely sequential or linear search and binary search. We will also come across iterative and recursive implementation for the two types of above mentioned searching techniques in this chapter.

Searching is a technique of determining whether a given element is present in a list of elements. We are given names of people and are asked for an associated telephone listing. We are given an employee names or codes and are asked for the personnel records of the employee. In these examples, we are given small piece of data or information, which we call as a **key,** and we are asked to find a record that has other information associated with the key. We shall allow both the possibility that there is more than one record with the same key and that there is no record at all with a given key. See Figure 2.1.1.

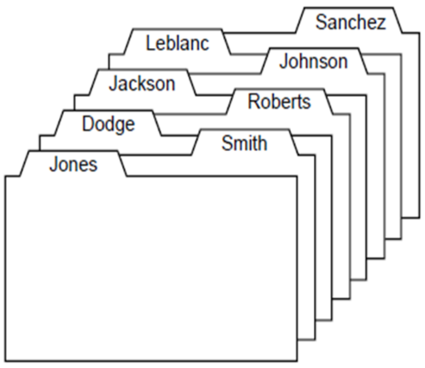


Figure 2.1.1: Sample records of employees

If the element we are searching is present in the list, then the searching technique should returns the index where that element is present in the list. If the search element is not present in the list then the searching technique should return NULL indicating that search element is not present in the list. Like sorting there are number of searching technique available in the literature and searching techniques vary based on its purpose suited for the application. Searching techniques can also be classified based on the data structures used to store the list. Searching techniques will vary for both linear as well as non-linear data structures. Among linear data structures such as arrays, linked lists, stacks and queues, the searching techniques will also vary. If array is used, then we must use different searching technique. Searching an element in a linked list requires different searching techniques. Also in case of non-linear data structures like trees will have different searching techniques. In this chapter we will introduce different types of searching algorithms and their implementations.

(i) Types of Searching

There are many different searching techniques and modern research is focused on advanced searching techniques using graphs like breadth first search (BFS) and depth first search (DFS) which will be discussed in the later chapters. In most common practice and very well-known there are two types of searching techniques, namely linear or sequential search and binary search algorithms.

1. Linear Search or Sequential Search

The simplest way to do a search in a given list is to begin at one end of the list and scan down it until the desired key is found or the other end is reached. This is our first method of searching which we call linear or sequential search.

Let A = [10 15 6 23 8 96 55 44 66 11 2 30 69 96] and searching element e = 11. Consider a pointer i, to begin with the process initialize the pointer i = 1. Compare the value pointed by the pointer with the searching element e= 11. As A (1) = 10 and it is not equal to element e increment the pointer i by i+1. Compare the value pointed by pointer i.e., A (2) = 15 and it is also not equal to element e. Continue the process until the search element is found or the pointer i reaches the end of the list.

Working of linear or sequential search is shown in the below figure 2.1.2

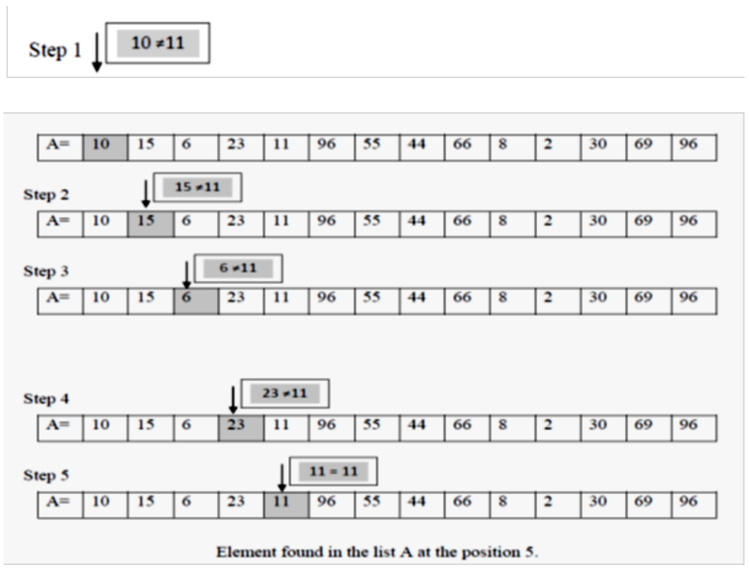

Step 1
10 ≠11
A= 10 15 6 23 11 96 55 44 66 8 2 30 69 96
Step 2
15 ≠11
A= 10 15 6 23 11 96 55 44 66 8 2 30 69 96
Step 3
6 ≠11
A= 10 15 6 23 11 96 55 44 66 8 2 30 69 96
Step 4
23 ≠11
A= 10 15 6 23 11 96 55 44 66 8 2 30 69 96
Step 5
11 = 11
A= 10 15 6 23 11 96 55 44 66 8 2 30 69 96
Element found in the list A at the position 5.

Figure 2.1.2: Pictorial representation of solution for sequential search

Characteristics and applications

In case of linear search, the searching happens sequentially. With this, if the element is present at end of the list or not at all present in the lists then this will lead to worst case scenario in case of linear search. In other words, for N elements in a list we will require (N-1) iterations for the above mentioned worst case scenario. This is a O(N) or Big O notation. The speed of linear search algorithms becomes directly proportional to the number of elements in the list. We should also note that, for linear search the list need not be in a sorted order. In some cases, we might place some frequently searched items or elements at the start of the list which will result into faster retrieval thereby increasing the performance irrespective of size of the item or element.

Despite of its worst case scenario which affects the performance of the searching technique, linear search technique is widely used in many applications. The built- in functions in programming languages like find index of Ruby, or of jQuery, completely depend on linear search techniques.

2. Binary search algorithm

Linear search is easy and efficient for short lists, but a worst for long ones. Just imagine trying to find the name “Carmel Fernandes” in a large directory by reading one name at a time starting at the front of the book! To find any record in a long list, there are far more efficient methods, provided that the keys in the list are already sorted into order.

A better methods for a list with keys in order is first to compare the key with one in the centre of the list and then restrict the search to only the first or second half of the list, depending on whether the key comes before or after the central one. With one comparison of keys we thus reduce the list to half its original size. Repeating this exercise, at each step, we reduce the length of the list to be searched by half. With only 20 search iterations, this method locates any required key in a list containing more than a million keys.

This method is called binary search. This approach requires that the keys in the list to be of a scalar or other type that can be regarded as having an order and that the list already completely in order.

Working of binary search algorithm

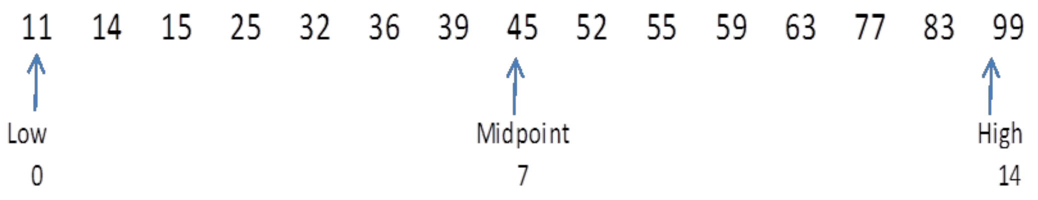
Consider an array A = [11, 14, 15, 25, 32, 36, 39, 45, 52, 55, 59, 63, 77, 83, 99] and the searching element e= 83. Let low be an integer containing first index of the array *i.e.,* 0 and high be the integer containing highest index of the array which is 14 in this case. Here we first compute mid by finding midpoint of high and low. In our case the midpoint is 7 (midpoint= [high (14) + low (0)]/2).

First we check if the search element is at midpoint. In our case search element is 83 and element at the midpoint is 45 which is not true.

Next step is we check if value at the midpoint is greater or smaller than the search element. If the value at midpoint is greater than the value of search element then it means our search element is in the left part of midpoint and hence the search must me restricted in the first half of the array. So we set high at midpoint and leave low unchanged. If the value at the midpoint is smaller than the search element, then it means our search element is present in right half of the array and hence search should be restricted in right side from midpoint. Therefore we set low to the midpoint and leave high unchanged.

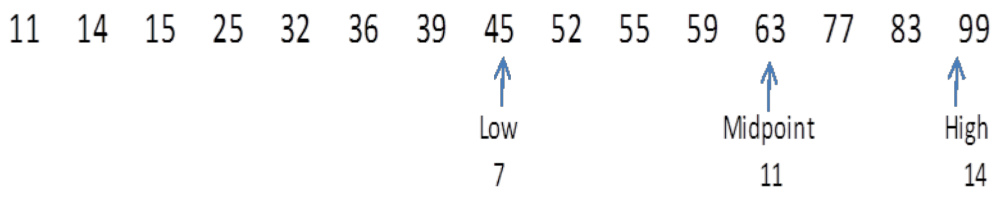
We repeat this process until the search element is found. The solution of the example taken above is solved below pictorially.

Initially low =0, high =14, midpoint=7 and we have element to be searched e= 83.



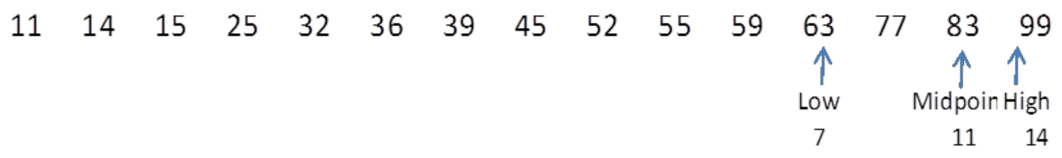
Since A[mid] is 45 which is smaller than 83 we set low =mid for our next iteration. So calculate the next midpoint using new low and high.

Iteration 2



Again the element is not present at midpoint and midpoint element is greater than 83 hence, the new we set low =11 and leave high unchanged. Our new midpoint will be,

Iteration 3



Now, first we check if the value at midpoint is our search element; which is true. Hence, our search is complete. The algorithm will return the value of the midpoint which is 11 in our case.

Characteristics and applications

In case of binary search technique, the given list of elements will be divided into two parts and one part gets eliminated in each iteration which is not so in case of linear search technique. This feature makes binary search more efficient and more powerful compared to that of linear search irrespective of the number of elements present in a list. Binary search technique is implemented either iteratively or recursively.

For binary search, the elements need to be in a sorted order which is not so in case of linear search. With this, the list is already in a sorted order before we start with searching. Since binary search is more suitable for larger sets of elements, its performance degrades if the list frequently gets updated for which again sorting takes place for the updated list.

Self-assessment Questions

1) Searching is important function because ________________.
   a) Information retrieval is most important part of the computer system
   b) Data in the computer system is unorganized
   c) It allows validating data present in the computer's memory
   d) It allows computer to validate information

2) What factor degrades the performance of binary search technique?
   a) Number of iterations   b) Re-sorting
   c) Size of the element   d) Size of the list

3) ____________ search algorithm begins at one end of the list and scans down it until the desired key is found or the other end is reached.
   a) Sequential   b) Binary
   c) BFS   d) DFS

2.1.2 Basic Sequential Searching

As we have already discussed working of sequential search algorithm, let us now focus on building a logical algorithm and implementing a program for the same. There are two ways of implementing this algorithm. First one is iterative method of implementation and the second one is recursive implementation. Figure 2.3 below shows a flowchart for implementing basic sequential search algorithm.

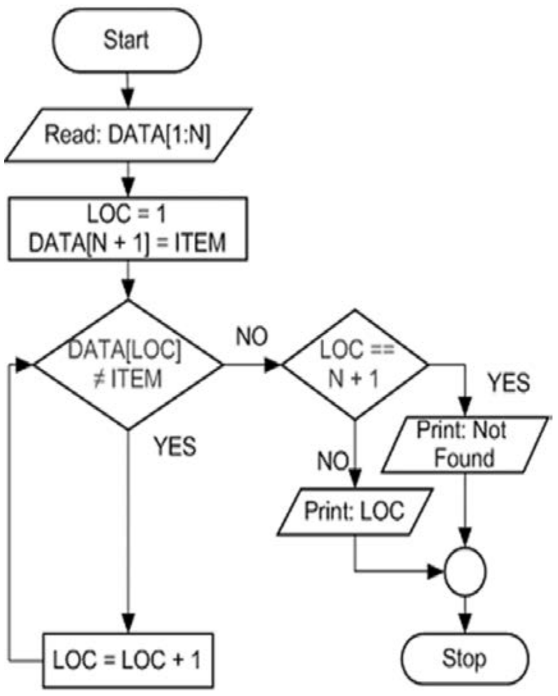


Figure 2.1.3: Flowchart for linear search algorithm

Types of implementation:

A search can be implemented by two methods:

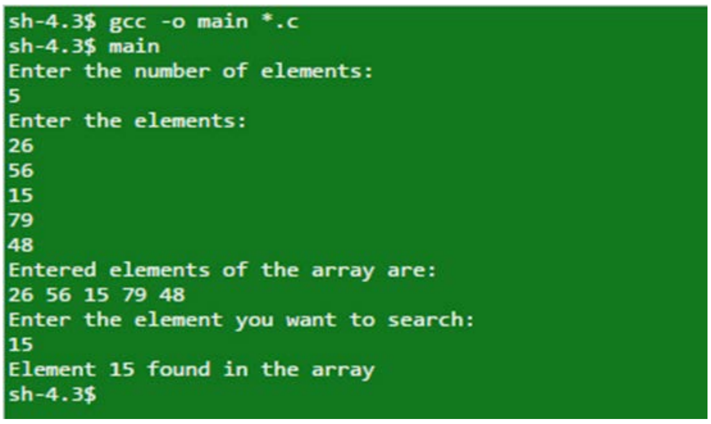
1. Iterative implementation
2. Recursive implementation

1. Iterative Implementation

Below program code demonstrates iterative method of implementation of sequential search algorithm.

```
#include <stdio.h>
void main()
{
    int arr[20];
    int x, n, key, flag = 0;
    printf("Enter the number of elements: \n");
    scanf("%d", &n);
    printf("Enter the elements: \n");
    for (x = 0; x < n; x++)
    {
        scanf("%d", &arr[x]);
    }
    printf("Entered elements of the array are:\n");
    for (x = 0; x < n; x++)
    {
        printf("%d ", arr[x]);
    }
    printf("\nEnter the element you want to search: \n");
    scanf("%d", &key);
    /*  Linear search logic */
    for (x = 0; x < n ; x++)
    {
        if (key == arr[x] )
        {
            flag = 1;
            break;
        }
    }
    if (flag == 1)
        printf("Element %d found in the array\n",key);
    else
        printf("Element %d not found in the array\n",key);
}
```

Output:



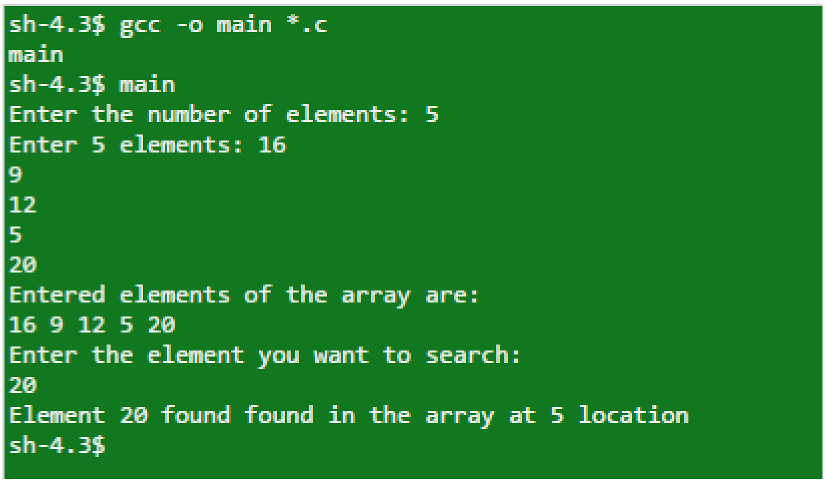
```
sh-4.3$ gcc -o main *.c
sh-4.3$ main
Enter the number of elements:
5
Enter the elements:
26
56
15
79
48
Entered elements of the array are:
26 56 15 79 48
Enter the element you want to search:
15
Element 15 found in the array
sh-4.3$
```

2. Recursive Implementation

Below program code demonstrates recursive method of implementation.

```
#include<stdio.h>
int line_search(int[],int,int);
int main()
{
    int arr[100],n,x,key;
    printf("Enter the number of elements: ");
    scanf("%d",&n);
    printf("Enter %d elements: ",n);
    for(x=0;x<n;x++)
    scanf("%d",&arr[x]);
    printf("Entered elements of the array are:\n");
    for(x=0;x<n;x++)
    printf("%d ",arr[x]);
    printf("\nEnter the element you want to search: \n");
    scanf("%d",&key);
    x=line_search(arr,n-1,key);
    if(x!=-1)
    printf("Element %d found in the array at %d location\n",key,x+1);
    else
    printf("Element %d is not found in the array\n",key);
}
int line_search(int s[100],int n,int key)
{
    if(n<=0)
    return -1;
    if(s[n]==key)
    return n;
    else
    return line_search(s,n-1,key);
}
```

Output:



```
sh-4.3$ gcc -o main *.c
main
sh-4.3$ main
Enter the number of elements: 5
Enter 5 elements: 16
9
12
5
20
Entered elements of the array are:
16 9 12 5 20
Enter the element you want to search:
20
Element 20 found found in the array at 5 location
sh-4.3$
```

Analysis of linear search algorithm

Worst Case Analysis (Usually Done)

In the worst case complexity, the upper bound on running time of an algorithm is calculated. In this case, we understand which case causes maximum number of operations to be executed. For sequential or linear search, the worst case is when the element to be searched is not present in the array. When number to be searched is not present, the algorithm compares it with all the elements of array one by one. Therefore, the worst case time complexity of linear search would be Θ (n).

Average Case Analysis (Sometimes done)

In average case complexity, we take every possible input and calculate computing time for all of the inputs. All the calculated values are summed and are divided by the sum of total number of inputs. In this way, we understand distribution of cases. For the linear search problem, consider that all cases are uniformly distributed including the worst case. So we sum all the cases and divide the sum by (n+1). Below is the example of average case time complexity.

$$\text{Average Case Time} = \frac{\sum_{i=1}^{n+1} \theta(i)}{(n+1)}$$

$$= \frac{\theta((n+1)*(n+2)/2)}{(n+1)}$$

$$= \Theta(n)$$

Best Case Analysis (Ideal)

Linear search algorithm performs best if the element is found to be the first element in the list during a searching process. That is, the time required for searching will be very less. Hence, the best or ideal case of linear search will be Θ (1).

Self-assessment Questions

4) Which is the worst case for linear search algorithm?

   a) The element to be searched is present at the first position in the array

   b) The element to be searched is present at the last position in the array

   c) The element to be searched is present at the middle position in the array

   d) The element to be searched is not present in the array

5) Best or ideal case complexity for linear search algorithm is _______.

   a) O(log n) b) O(n)

   c) O(1) d) O(n log n)

6) The average case complexity of linear search algorithm is ___________.

   a) O(log n) b) O(n)

   c) O(1) d) O(n log n)

2.1.3 Binary search

Similar to the implementation of linear search algorithm, binary search algorithm can also be implemented using linear and recursive methods of implementation. First, consider the below flowchart in figure 2.1.4 to understand the working of the algorithm. The flowchart is implemented using the same logic as discussed in the previous section.

The flowchart begins with input key (element) from the user. It then finds the mid-point of the list using the values from low and high index of the list. After finding the mid-point, a comparison will be done so as to check the key value with that of the element present at the mid-point of the list. If the key matches the element at the mid-point, the search is successful. If it does not match, another comparison is done to check if the key value is lesser than that of the element at the mid-point. If it is successful then the left sub-array will be considered and if not then right sub-array will be considered. The whole process is repeated until we find the key value matching the element of the list and if not then we can conclude that the key value is not present in the list.

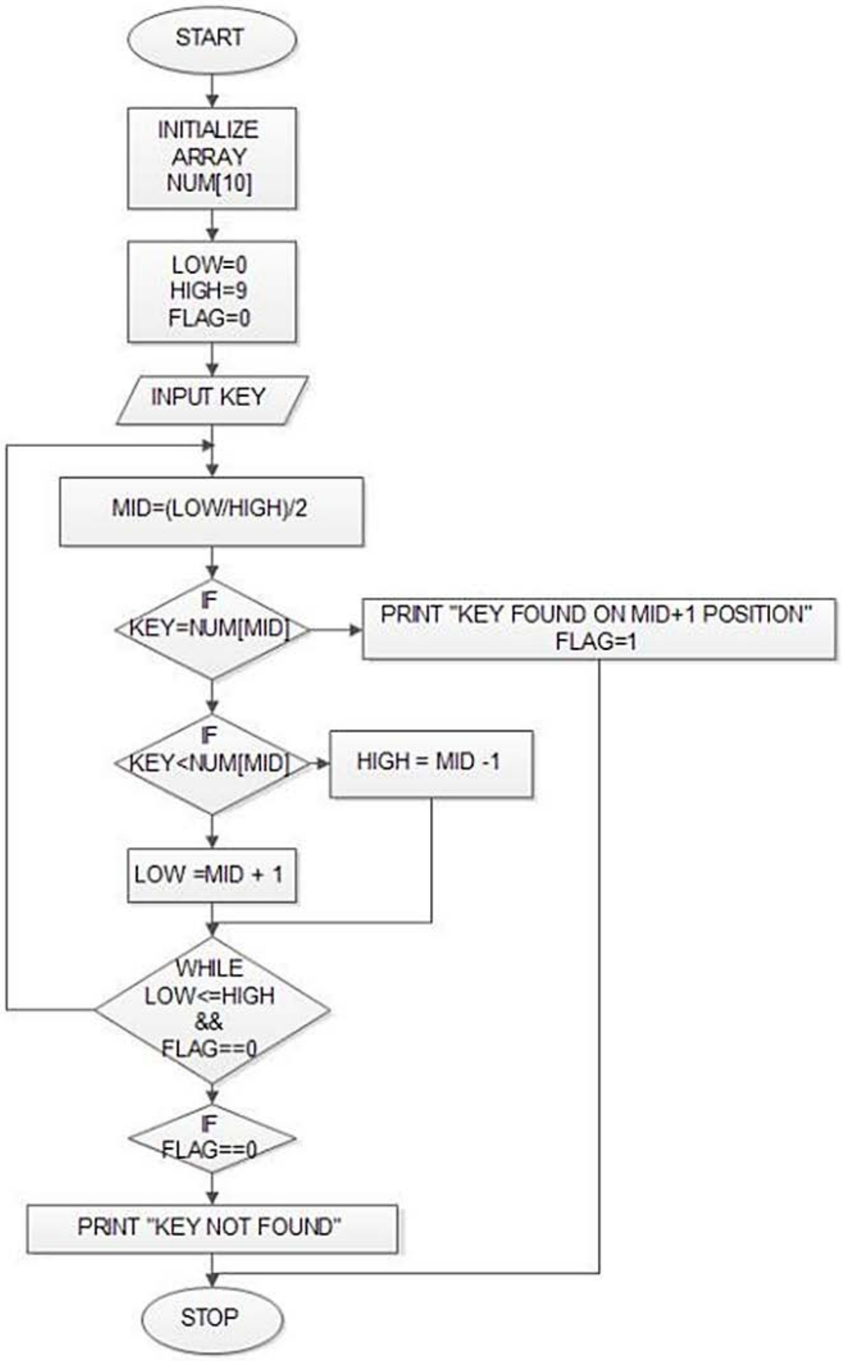


Figure 2.1.4: Flowchart for binary search algorithm

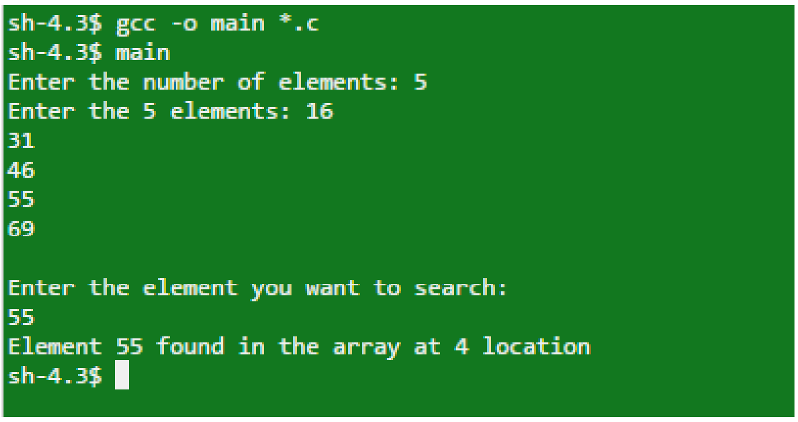
(i) Iterative implementation

Iterative implementation of binary search is based on the explanation in the above flowchart. The limitation of this algorithm is seen at the termination of this algorithm for an unsuccessful search. That is, when the search is not successful the low convolutes over to the right of high, low > high and this terminates the while loop.

The Algorithm below implements iterative method of binary search.

```
#include <stdio.h>
int main()
{
   int x, low, high, mid, n, key, arr[50];
   printf("Enter the number of elements: ");
   scanf("%d",&n);
   printf("Enter the %d elements: ", n);
   for (x = 0; x < n; x++)
       scanf("%d",&arr[x]);
   printf("\nEnter the element you want to search: \n");
   scanf("%d", &key);
   low = 0;
   high = n - 1;
   mid = (low+high)/2;
   while (low <= high) {
      if (arr[mid] < key)
         low = mid + 1;
      else if (arr[mid] == key) {
         printf("Element %d found in the array at %d location\n", key,
mid+1);
         break;
      }
      else
         high = mid - 1;
      mid = (low + high)/2;
   }
   if (low > high)
      printf("Element %d is not found in the array\n", key);
   return 0;
}
```

Output:



```
sh-4.3$ gcc -o main *.c
sh-4.3$ main
Enter the number of elements: 5
Enter the 5 elements: 16
31
46
55
69

Enter the element you want to search:
55
Element 55 found in the array at 4 location
sh-4.3$ █
```

(ii) Recursive implementation

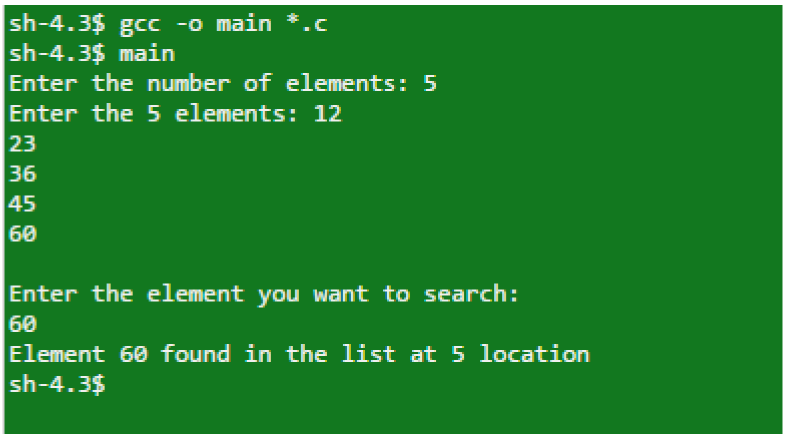
Recursive implementation of binary search overcomes the limitation that was seen in iterative method of binary search. Here it checks for condition, if (low>high) then it returns -1. This is similar to while condition in the previous method. This terminates the recursion. If this is not successful then the recursive function gives new values into parameters of a recursive call.

The algorithm below implements the recursive method for performing binary search.

```
#include<stdio.h>
#include<stdlib.h>

int bin_rsearch(int[], int, int, int);

int main() {
   int n, x, key, pos;
   int low, high, arr[20];
   printf("Enter the number of elements: ");
   scanf("%d", &n);
   printf("Enter the %d elements: ");
   for (x = 0; x < n; x++)
    {
      scanf("%d", &arr[x]);
    }
   low = 0;
   high = n - 1;
   printf("\nEnter the element you want to search: \n");
   scanf("%d", &key);

   pos = bin_rsearch(arr, key, low, high);

   if (pos != -1) {
      printf("Element %d found in the list at %d location\n", key, (pos +
1));
   } else
      printf("Element %d is not found in the array\n", key);
   return (0);
}

// Binary Search function
int bin_rsearch(int s[], int i, int low, int high)
{
   int mid;

   if (low > high)
      return -1;

   mid = (low + high) / 2;
```

```
    if (i == s[mid]) {
       return (mid);
    } else if (i < s[mid]) {
       bin_rsearch(s, i, low, mid - 1);
    } else {
       bin_rsearch(s, i, mid + 1, high);
    }
}
```

Output:



```
sh-4.3$ gcc -o main *.c
sh-4.3$ main
Enter the number of elements: 5
Enter the 5 elements: 12
23
36
45
60

Enter the element you want to search:
60
Element 60 found in the list at 5 location
sh-4.3$
```

Analysis of Binary search algorithm

Worst Case Analysis (Usually Done)

Similar to linear search, the worst case of binary search is when the element to be searched is not present in the array. When number to be searched is not present in each iteration of the binary search algorithm, the size of the permissible array is halved. And this having goes up to O(log n) times. Therefore, the worst case time complexity of recursive binary search algorithm is O(log n) and the worst case complexity of iterative binary search algorithm is O(1).

Average Case Analysis (Sometimes done)

To calculate the average case complexity of binary search algorithm, we take the sum over all the elements of the product of number of comparisons required to find each element and the probability of searching that element.

For simplicity of analysis, consider no item which is not in A will be searched for, and the probabilities for searching each element uniform. Therefore the time complexity of binary search algorithm is O (log n).

Best Case Analysis (Ideal)

In the sequential search problem, the best case is the element to be searched is present at the middle of the array. The number of operations in the best case is constant i.e. it is independent on n. So time complexity in the best case would be O (1).

Did you Know?

The difference between O(log(N)) and O(N) is extremely significant when size of the array N is large: for any practical problem it is crucial that we avoid O(N) searches.

Self-assessment Questions

7) To calculate midpoint in binary search algorithm we ___________.

a) Divide lowest index by highest index

b) First add lowest index and highest index and then divide the sum by 2

c) First subtract highest index from lowest index and divide the result by 2

d) Just subtract highest index and lowest index

8) Best case for binary search algorithm is when the element to be searched is,

a) In the beginning of the array

b) At the end of the array

c) At the middle of the array

d) At any position in the array

9) Average case complexity for binary search algorithm is O(log n),

a) True b) False

2.1.4 Comparison between sequential and binary search

As already discussed in the previous section of this chapter, it is understood that both binary search and sequential search algorithms have several differences. In this section we will compare these algorithms based on various parameters.

1. Implementation requirements

First and the foremost thing, the most obvious difference between the two algorithms lies in input requirements. Sequential search can be done even over an unsorted array elements as the comparison is done in sequential manner. Whereas, for binary search algorithm to work, the input array of elements must be sorted. This is because the fundamental way of working of algorithm, which is based on array indices.

2. Efficiency

Linear search algorithms works best for small array sizes. However, as the size of the array increases the performance goes down. However, binary search technique works at its best for any array size as the array size is halved in every iteration or recursion, so does the number of comparisons.

The efficiency also depends on location of the search element. If for linear search algorithm, the element is present at the starting location of the array, then it becomes the best case and if present at the last position then it becomes worst case. Binary search is most efficient when the element to be searched is at the middle of the array. For any other location other than the middle, the efficiency doesn’t affect much.

3. Complexities

For linear search has an average case complexity of O(n) which makes it very slow and inefficient for huge array sizes. However, binary search an average case complexity of O (log n) which makes it a better search algorithm even for large array sizes.

4. Data structure

The binary search algorithm works best for arrays but not for linked lists because of the very fundamental structure of arrays having regular indexing and contiguous memory allocation unlike liked lists. On the other hand, sequential searching works well for both arrays and linked lists.

Self-assessment Questions

10) Linear search algorithm requires array to be sorted before it start searching.

    a) True b) False

11) Efficiency of linear search algorithm does not depend on the position of the element to be searched.

    a) True b) False

12) In general, binary search algorithm is best when the array size is big.

    a) True b) False

Summary

- Searching is one of the primary function of the computer system, information retrieval being increasingly important.
- Though there are various other algorithms for searching, two very famous and important algorithms are sequential search and binary search
- Linear search is simplest of the two, which involves searching elements from one of the ends of the array until the search element is found. As the array size grows big, this algorithm proves inefficient as it consumes lot of time for carrying out search.
- Binary search overcomes the disadvantage of sequential search as it reduces number of iterations taken for finding out the search element. It halves the array size and hence the number of comparisons in each iteration.
- Unlike linear search, binary search requires the input array to be sorted because of very fundamental working of its algorithm which is based on index assignment to lower to upper side of the array.
- In general case, time complexity of sequential search algorithm is O(n) which makes it slower and less efficient as compared to binary search which has time complexity of O(log n).

Terminal Questions

1. Explain in brief different types of searching algorithms.

2. Consider the following array A = {23, 26, 32, 35, 39, 42, 44, 47, 50, 55, 58, 62, 66, 88, 99} and search for element e=26 using binary search technique. (Solution needs to be demonstrated pictorially with solution for each iteration).

3. Provide an algorithm for recursive implementation of linear search.

4. Draw a neat flowchart for binary search algorithm.

5. Explain in brief the difference between linear and binary search algorithm.

Answer Keys

Self-assessment Questions	
Question No.	**Answer**
1	a
2	b
3	a
4	d
5	c
6	b
7	b
8	c
9	a
10	b
11	b
12	a

Activity

1. Activity Type: Offline **Duration: 20 Minutes**

Description:

Stack 10 reference books in ascending order of their titles.

Ask the students to write a program for binary search to search books by their title.

Case Study: Alphabetizing Papers

Consider the example of a human alphabetizing a couple dozen papers. If we think about it for a while it's basically a sorting algorithm. If one tries to understand the process or working of this algorithm, following questions are needed to be asked.

1. How are the papers alphabetized?
2. How are the papers arranged?

Basically all papers with names starting with A are put in pile named 'A', similarly names starting with B are put in pile names 'B' and so on. The groups (pile) range varies based on the number of other factors which are chosen as per convenience. Once the grouping is done, next each pile or a group is scanned letter by letter and a new algorithm is used for further working. In 90 per cent of the cases, humans unknowingly use insertion sort algorithm.

It is well known that the quicksort is the best and fastest way to sort. The question is then, why don't humans use quicksort? Human brain doesn't do all comparisons equally. It's just "easier" for our brains to quickly apply insertion sort. Splitting into letter groups makes each smaller problem more manageable.

In reality, humans use an algorithm called a bucket sort. A bucket sort followed by individual insertion sorts (exactly what humans tend to do) is a linear time sorting algorithm. When we have some notion of the distribution of the items to be sorted, we can break through that boundary and do linear time sorting. The requirement with linear time sorting is that the input must follow some known distribution. This is the reason why humans instinctively break the piles into various types of groupings. If there are many papers, it is required to reduce group ranges. Furthermore, the ideal bucket setup would distribute the papers roughly evenly. The letter S might need its own bucket, but we can put all the letters up through F in their own bucket. Humans have many of experiences with both the general problem and their specific problem (***For example,*** the peculiarities of a particular class' name distribution) and so they try to optimize the algorithm given the known distribution. They are setting up the parameters of the linear time sort (number of buckets, bucket ranges, etc.) exactly as they should to optimize the sort time.

The main disadvantage to these linear sort algorithms is that they require lot of extra memory space (versus comparison-based sorting). We need to have an auxiliary bookkeeping array on the order of the original problem to do them. This isn't a problem in real life problem, where in we just need a large table to arrange papers. In a very real sense, this supposedly "naive" algorithm that humans use is among the very best possible.

Questions:

1. Explain the process followed by humans for sorting papers, as described in above case study. What is the method called technically and what is the supported sorting algorithm?
2. Why do you think humans cannot think of sorting using quicksort?
3. Why it is not advised to use bucket sort for implementing computer based sorting algorithm?
4. Do you agree with the author in the case study that the process followed by humans for sorting applications in real life is fastest?

Bibliography

e-References

- interactivepython.org, (2016). *Problem solving in Data structures: The Binary Search.* Retrieved on 19 April 2016, from http://interactivepython.org/runestone/static/pythonds/SortSearch/TheBinarySearch.html

- pages.cs.wisc.edu, (2016). *Searching and Sorting.* Retrieved on 19 April 2016, from http://pages.cs.wisc.edu/~bobh/367/SORTING.html

External Resources

- Kruse, R. (2006). *Data Structures and program designing using 'C' (2nd ed.).* Pearson Education.

- Srivastava, S. K., & Srivastava, D. (2004). *Data Structures Through C in Depth (2nd ed.).* BPB Publications.

- Weiss, M. A. (2001). *Data Structures and Algorithm Analysis in C (2nd ed.).* Pearson Education

Video Links

Topic	Link
For Introduction to searching, and types of searching techniques	https://www.youtube.com/watch?v=mqixr2wdLqg
Implementation of linear search iterative method	https://www.youtube.com/watch?v=AqjVd6FVFbE
Implementation of binary search iterative method	https://www.youtube.com/watch?v=g9BKw_TobpI
Implementation of binary search recursive method	https://www.youtube.com/watch?v=-bQ4UzUmWe8
Comparison of linear and binary search	https://www.youtube.com/watch?v=u3v-vh2t9FE

Notes: