Searching

Topics

- Sequential Search on an Unordered File
- Sequential Search on an Ordered File
- Binary Search

Searching

Searching : The goal of search is to find a particular object in this collection or to recognize that the object does not exist in the collection.

Often the objects have key values on which one searches and data values which correspond to the information one wishes to retrieve once an object is found. For example, a telephone book is a collection of names (on which one searches) and telephone numbers (which correspond to the data being sought

The collection of objects is often stored in a list or an array. Given a collection of n objects in an array A[1 . . n], the i-th element A[i] corresponds to the key value of the i-th object in the collection. Often, the objects are sorted by key value (e.g., a phone book), but this need not be the case.

Different algorithms for search are required if the data is sorted or not. The input to a search algorithm is an array of objects A, the number of objects n, and the k

Searching

- A question you should always ask when selecting a search algorithm is “How fast does the search have to be?” The reason is that, in general, the faster the algorithm is, the more complex it is.
- Let’s explore the following search algorithms, keeping speed in mind.

  A- Sequential (linear) search

    1- Unordered Linear Search

    2- ordered Linear Search

  B- Binary search

1. Unordered Linear Search

Suppose that the given array was not necessarily sorted. This might correspond, for example, to a collection exams which have not yet been sorted alphabetically. If a student wanted to obtain her exam score, how could she do so? She would have to search through the entire collection of exams, one-by-one, until her exam was found. This corresponds to the unordered

linear search algorithm.

Unordered Linear Search

Input: objects array A, the number of objects n, key value being sought x.

Output: if found, return position i, if not, return message “x not found”

a. Compare x with EACH element in array A from the every beginning.

b. If x = the ith element in A. Terminate the search and return position i.

c. If not, keep searching for the next element until the end of the array.

d. If no proper element was found in the array, return “x not found”.

Note that in order to determine that an object does not exist in the collection, one needs to search through the entire collection

Now consider the following array:

I	1	2	3	4	5	6	7	8
A	34	16	25	33	7	29	48	14

If we want to search for x = 33 in this array. We have to compare x with (34, 16, 25, 33), each element once. Then we find 33 in position 4. Therefore, we return 4. The total number of comparisons we have executed is 4.

If we want to search for x = 18 in this array. We have to compare x with (34, 16, 25, 33, 7, 29, 48, 14), each element once. After searching all the elements in this array, we don't find 18. So we return " not found". The total number of comparisons we have executed is 8.

In General, we want to search for x in an unordered object array A with n elements. In the worst case, we have to search through the entire array to get the final answer. That means we have to execute n comparisons. Now let's use an equation to represent the number of comparisons we have executed. That should be T(n) = n

2. Ordered Linear Search

Now suppose that the given array is sorted. A simple modification of the above algorithm yields the ordered linear search algorithm.

Ordered Linear Search

Input: ordered objects array A, the number of objects n, key value being sought x.

Output: if found, return position i, if not, return message “x not found”

a. From the every beginning of the array A, compare x with the element, say A[i], in A. See if they are EQUAL.

b. If x = A[i], terminate the search and return position i.

c. If not, compare x with that element AGAIN. See if x is GREATER than A[i].

d. If x > A[i], go on searching for the next element in array A.

e. If not, which means x < A[i], terminate the search and return “x not found”.

Now consider the following array, the sorted version of the array used in our previous example:

Now consider the following array, the sorted version of the array used in our previous example:

i	1	2	3	4	5	6	7	8
A	7	14	16	25	29	33	34	48

This time, if we still search for x = 33 in this array. We have to compare x with (7, 14, 16, 25, 29), each element twice, one for "=", another one for ">". Then we will compare x with (33) only once, for "=". And we will find 33 in position 6. Therefore, we return position 6. Total comparisons = 2*5 + 1 = 11.

If we search for x = 18 in this array. We have to compare x with (7, 14, 16, 25), each element twice, one for "=", another one for ">". And in then last comparison (if x > 25), we got an answer "NO"! That means all the elements after 25 in this array are greater than x. Therefore, we return "x not found". Total comparisons = 2*4 = 8.

In General, we want to search for x in an ordered object array A with n elements. In the worst case (x is greater than the last element in the array), we have to search through the entire array to get the final answer. That means we have to execute n*2 comparisons, n for "=", another n for ">". Now let's use an equation to represent the number of comparisons we have executed. That should be T(n) = 2n.

Binary Search

Binary Search is a more efficient option for searching arrays.

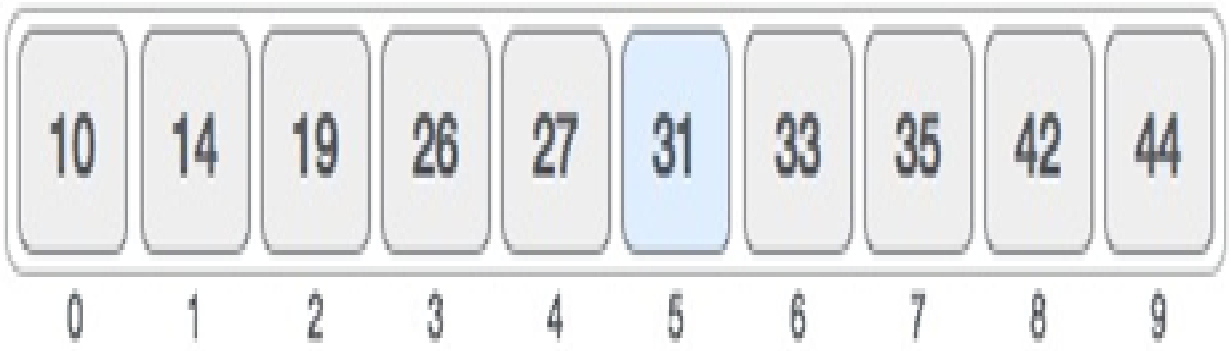
There are two tricks to using Binary Search:

First, the array must be sorted. (Before you use Binary Search, you might first sort the array.)

Second, Binary Search works by dividing the search area in half after each comparison (more on this soon.)

.

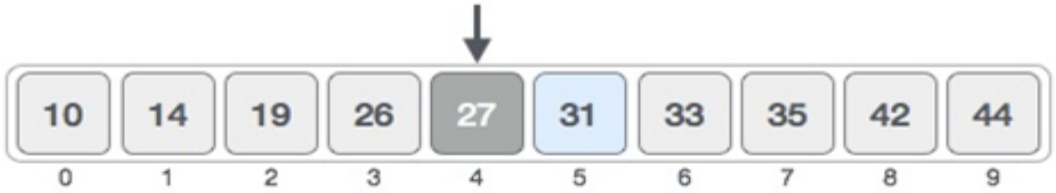
```
Procedure binary_search
  A ← sorted array
  n ← size of array
  x ← value to be searched
   Set lowerBound = 1
   Set upperBound = n
   while x not found
    if upperBound < lowerBound
       EXIT: x does not exists.
       set midPoint = lowerBound + ( upperBound - lowerBound ) / 2
         if A[midPoint] < x
                set lowerBound = midPoint + 1
         if A[midPoint] > x
               set upperBound = midPoint - 1
         if A[midPoint] = x
              EXIT: x found at location midPoint
   end while
  end procedure
```



First, we shall determine half of the array by using this formula –

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

Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

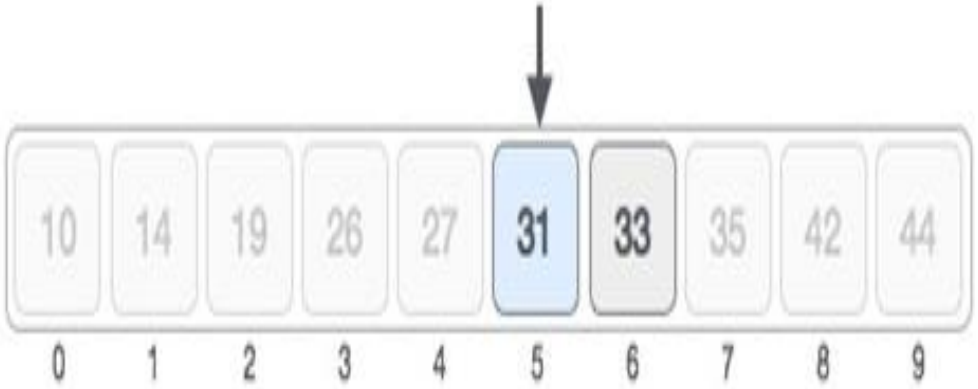
We change our low to mid + 1 and find the new mid value again.

```
low = mid + 1
mid = low + (high - low) / 2
```

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.

We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.