Al-Mustaqbal University
College of Sciences
Intelligent Medical System Department

جامعة المستقبل
AL MUSTAQBAL UNIVERSITY

كلية العلوم

قسم الانظمة الطبية الذكية
Intelligent Medical Systems Department

Subject: Data Structure

Class: Second

Lecturer: Prof.Dr. Mehdi Ebady Manaa

Lecture: (9)

Searching Algorithms

Searching Algorithms

Searching

The task of searching is one of most frequent operations in computer programming. It also provides an ideal ground for application of the data structures so far encountered. There exist several basic variations of the theme of searching, and many different algorithms have been developed on this subject. The basic assumption in the following presentations is that the collection of data, among which a given element is to be searched, is fixed. We shall assume that this set of N elements is represented as an array, say as

A: ARRAY N OF Item

Typically, the type item has a record structure with a field that acts as a key. The task then consists of finding an element of A whose key field is equal to a given search argument x. The resulting index i, satisfying A[i].key = x, then permits access to the other fields of the located element. Since we are here interested in the task of searching only, and do not care about the data for which the element was searched in the first place, we shall assume that the type *Item* consists of the key only, i.e. *is* the key.

1-Linear Search

When no further information is given about the searched data, the obvious approach is to proceed sequentially through the array in order to increase step by step the size of the section, where the desired element is known not to exist. This approach is called *linear search*. There are two conditions which terminate the search:

1. The element is found, i.e. a[i] = x.
2. The entire array has been scanned, and no match was found.

This results in the following algorithm:

Seq-search Algorithm

```
 i=1
 While (i<=n) and (a[i] ≠ x) do
    Increment (i)
 If i=n   then   element is found
 Else            element is not found
End
```

2- Binary search:

Generally, to find a value in unsorted array, we should look through elements of an array one by one, until searched value is found. In case of searched value is absent from array, we go through all elements. In average, complexity of such an algorithm is proportional to the length of the array.

Situation changes significantly, when array is sorted. If we know it, random access capability can be utilized very efficiently to find searched value quick. Cost of searching algorithm reduces to binary logarithm of the array length. For reference, $\log_2(1\ 000\ 000) \approx 20$. It means, that **in worst**

case, algorithm makes 20 steps to find a value in sorted array of a million elements or to say, that it doesn't present it the array.

Algorithm

Algorithm is quite simple. It can be done either recursively or iteratively:

1. get the middle element;
2. if the middle element equals to the searched value, the algorithm stops;
3. otherwise, two cases are possible:
    - Searched value is less, than the middle element. In this case, go to the step 1 for the part of the array, before middle element.
    - Searched value is greater, than the middle element. In this case, go to the step 1 for the part of the array, after middle element.

Now we should define, when iterations should stop. First case is when searched element is found. Second one is when subarray has no elements. In this case, we can conclude, that searched value doesn't present in the array.

Examples

Example 1. Find 6 in {-1, 5, 6, 18, 19, 25, 46, 78, 102, 114}.

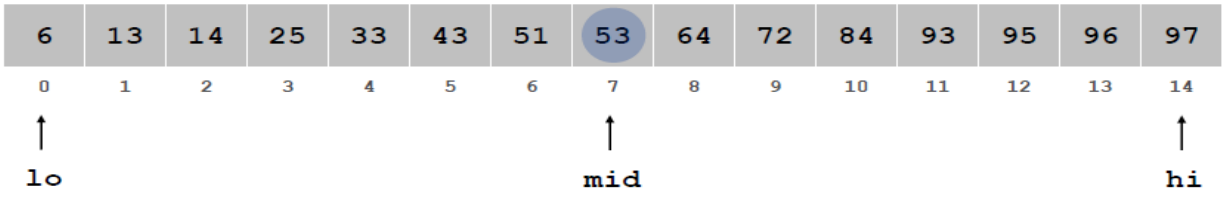
```
Step 1 (middle element is 19 > 6):      -
1 5 6 18 19 25   46   78   102   114

Step 2 (middle element is 5 < 6):       -
1   5   6   18   19   25   46   78   102   114

Step 3 (middle element is 6 == 6):      -
1   5   6   18   19   25   46   78   102   114
```

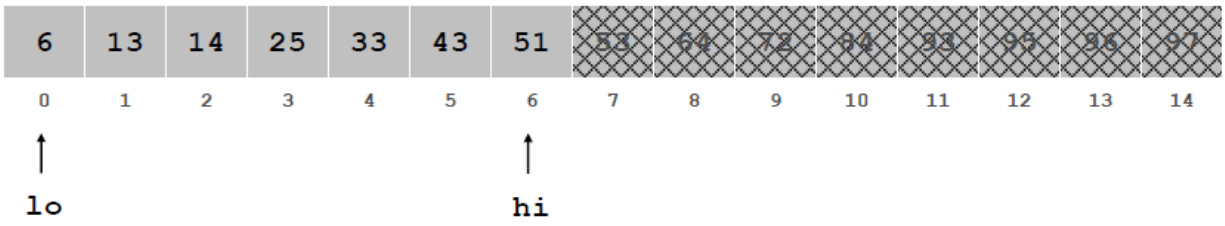
Example 2. Find 103 in {-1, 5, 6, 18, 19, 25, 46, 78, 102, 114}.

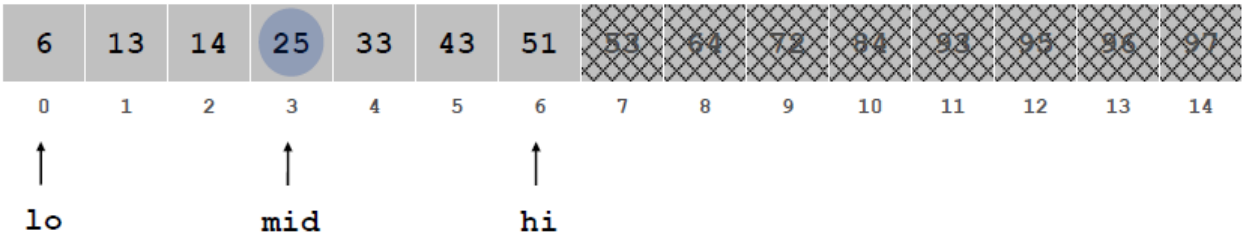
```
Step 1 (middle element is 19 < 103):    -
1   5   6   18   19   25   46   78   102   114

Step 2 (middle element is 78 < 103):    -
1   5   6   18   19   25   46   78   102   114

Step 3 (middle element is 102 < 103):   -
1   5   6   18   19   25   46   78   102   114
```

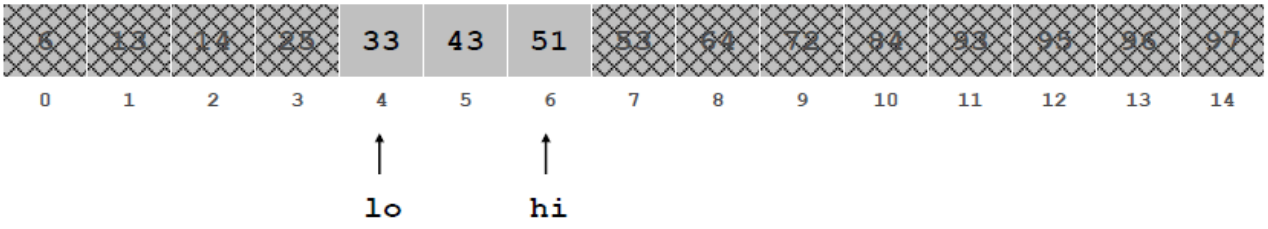
```
Step 4 (middle element is 114 > 103):   -
1   5   6   18   19   25   46   78   102   114

Step 5 (searched value is absent):      -
1   5   6   18   19   25   46   78   102   114
```

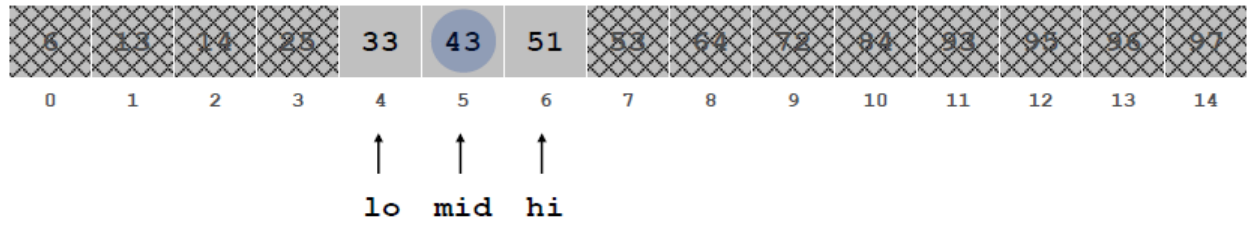
Given value and sorted array a[], find index I such that a[i] = value, or report that no such index exists. Invariant. Algorithm maintains a [lo] ≤ value ≤ a[hi].
Ex. Binary search for 33.

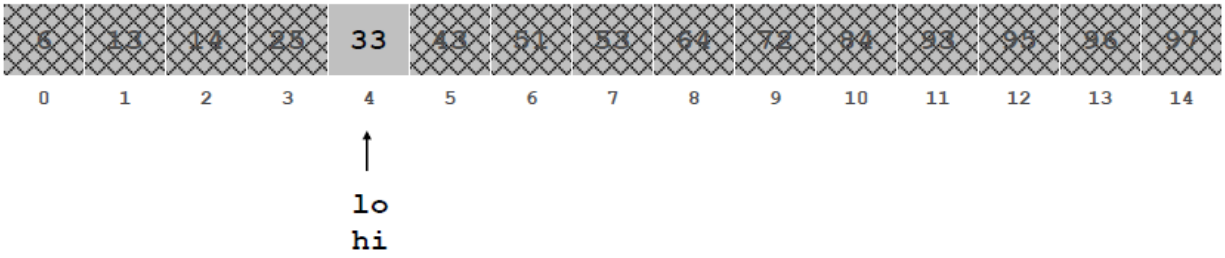


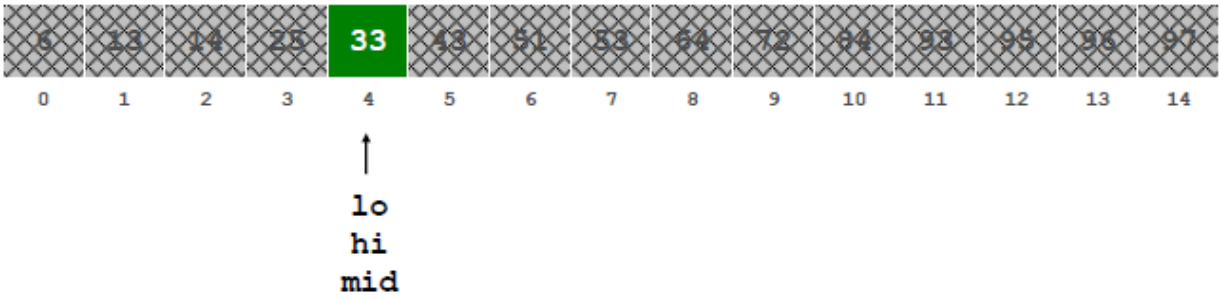







6 13 14 25 33 43 51 53 64 72 84 93 95 96 97
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
lo mid hi


6 13 14 25 33 43 51 53 64 72 84 93 95 96 97
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
lo
hi


6 13 14 25 33 43 51 53 64 72 84 93 95 96 97
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
lo
hi
mid