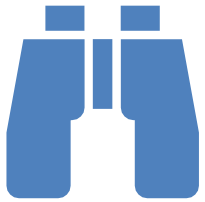
Data Structures & Algorithms



Searching

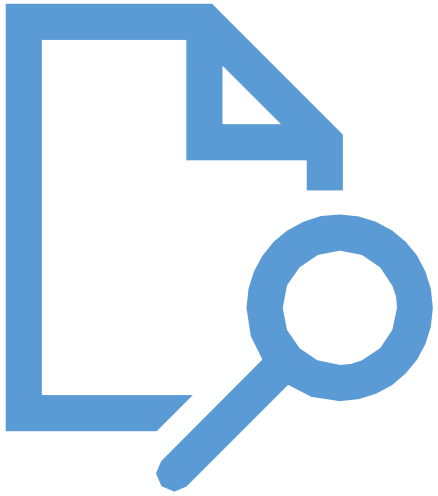
Searching

By searching, we mean to find the position/location of a desired element stored in some data structure.

In this course we will learn Internal search techniques. [All data can be loaded into RAM (main memory) while the searching is being carried out]

A search can be either successful (if the search item present) or unsuccessful (search fails - if the search item is not present).

Assume that array contains unique elements.



The Sequential Search

The most direct and simplest but slowest approach is known as **sequential/linear** searching.

Given unsorted sequence of numbers stored in an array.

The method

A linear search sequentially checks each element of the array until it finds an element that matches the target value.

If the algorithm reaches the end of the list, the search terminates unsuccessfully.

Linear Search

// This algorithm searches for a ‘key/target’ element in an array A of size n.

// Successful search returns the index of search element, if the search is unsuccessful -1 is returned.

```
linear_search(A[], n, Target)
 {
    for (i = 0; i < n; i++)
    {
            if (A[i] == Target)
                return i; // Found the position of the target element requested
     }
     return -1; // The target element to be searched is not in the array
}
```

Analysis of Sequential Search

There are actually three different scenarios that can occur.

Worst-case - O(n) // Since every element is checked in the worst case

// If the search(or target) element is not present in the array at all (or occurs at the last cell of the array)

Best-case - O(1) // when the search element is found in the beginning(first cell) of the array

Average case - O(n)

// we will find the search element about halfway into the array; that is, we will compare against n/2 items.

[assumption is that there is an equal chance for the number to be found in any cell of the array; that is, the probability distribution is uniform (probability =1/n for all cell)]

Binary Search

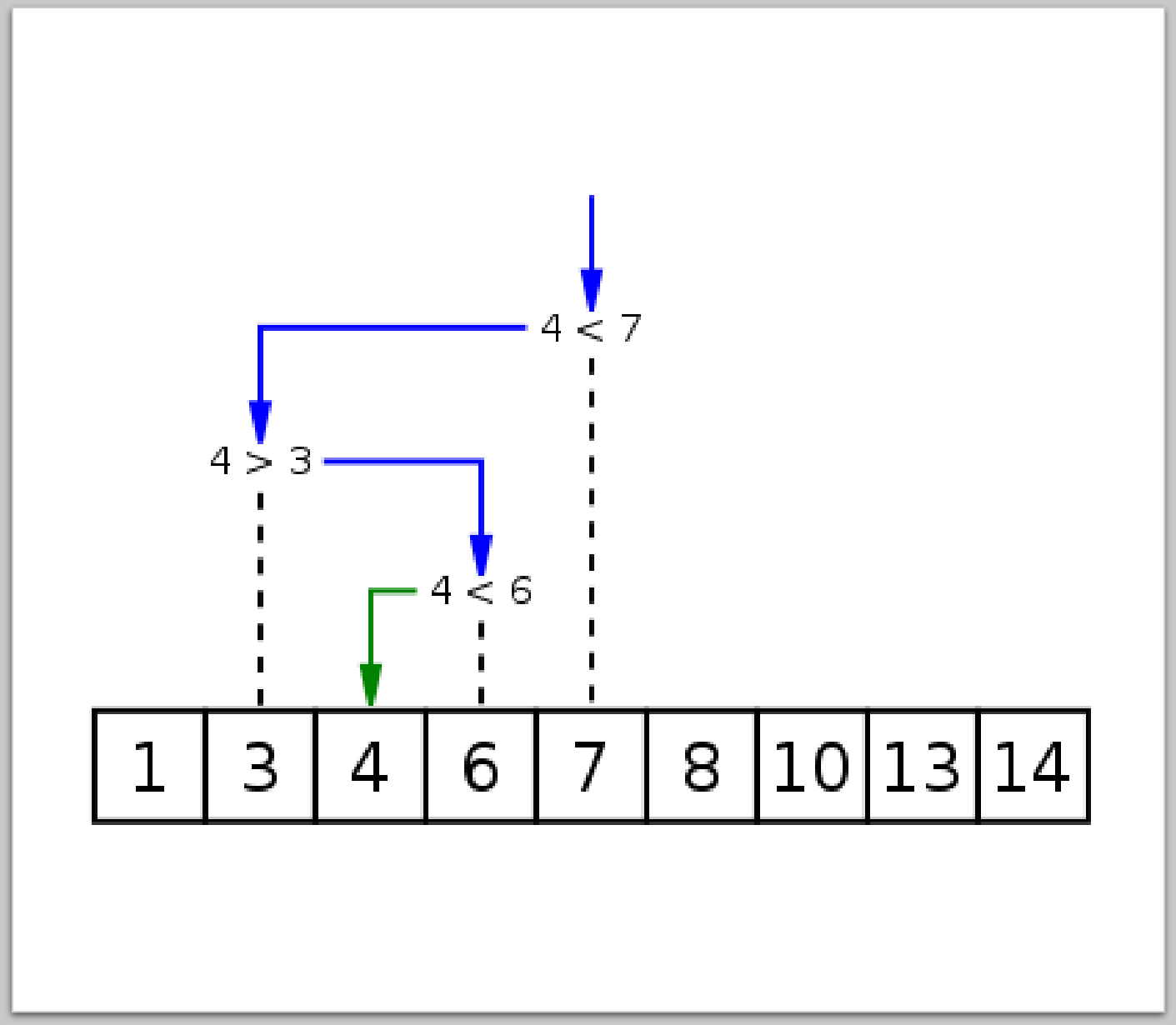
When the elements in the array are **sorted**, we can search faster using Binary search technique.

Binary search compares the **target value** to the **middle element** of the array with the target value. If the target value matches the element, its position in the array is returned.

If they are not equal, the **half in which the target cannot lie is eliminated** and the **search continues on the remaining half**, again taking the middle element to compare with the target and repeating this until the target is found.

If the search ends with the remaining half being empty, the target is not in the array.

Binary Search



Iterative Binary Search

/* left = 0, right = n-1; It is invoked with initial left and right values of 0 and n-1 for a zero-based array of size n*/

```
Binary_Search (arr[], left, right,  target)
 {
  while (left <= right)
   {
        mid = ⌊(left +right)/2⌋;          // or   mid = left + (right - left) / 2;
        if  (target == arr[mid])
              return mid; // success : return the index of the cell occupied by target value;
        if (target < arr[mid])
              right = mid - 1;
        else
              left = mid + 1; // if (arr[mid] < target)
   }
   return -1; // failure(unsuccessful search): key/target is not present in the array;
}
```

Analysis of Binary Search

Worst-case – O(log n)

// when either the target element does not exist in the array or the search has reduced to one element

Best-case - O(1)

//when the target element is the middle element of the array or find the target element in the first comparison

Average case - O(log n)

// when the target value is anywhere in the array, except in the middle element of the array or the search has not reduced to one element