Riphah International University
I-14 Main Campus
Faculty of Computing

Class:	Fall-2024	**Subject:**	Data Structures & Algorithms
Course Code:	CS 2124	**Lab Instructor:**	Zeeshan Ali

Learning Objective:

- Binary Search
- Binary Search Algorithm
- Working of Binary Search
- Binary Search Example in C++
- Lab Tasks

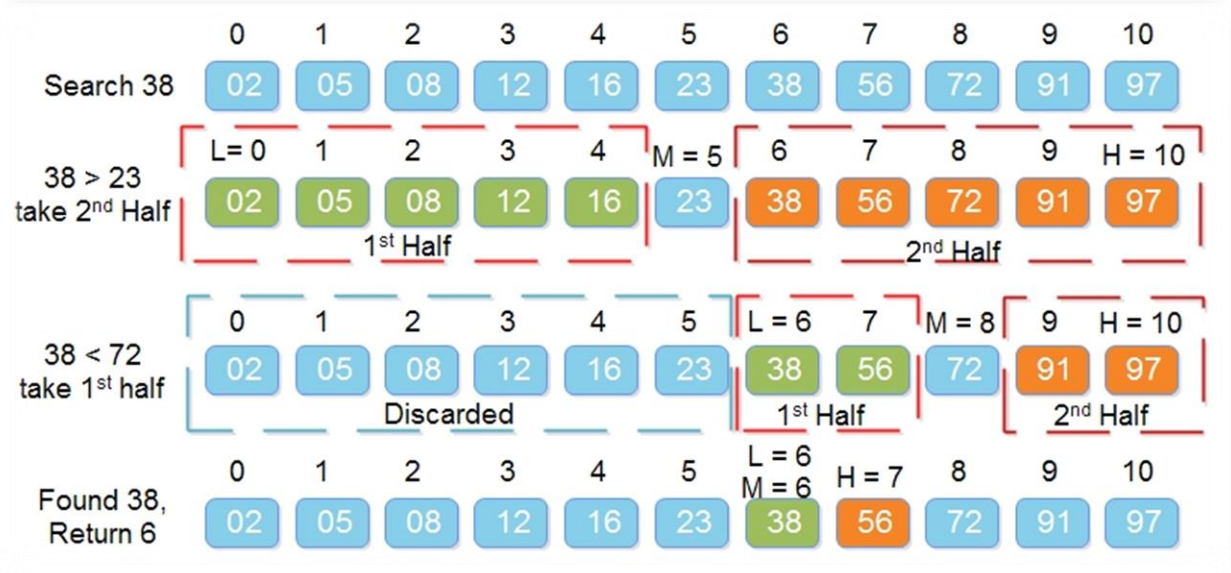
Binary Search

Binary search is an **efficient algorithm** for finding an item from a **sorted list** of items. It works by repeatedly dividing in half the portion of the list that could contain the item, until you've narrowed down the possible locations to just one.

Binary search follows the **divide and conquer approach** in which the list is divided into **two halves**, and the **item** is compared with the **middle element** of the list.

If the match is found then, the location/index of the middle element is returned.

Otherwise, we search into either of the **halves** depending upon the result produced through the match.



Binary Search Algorithm

- Binary search algorithm is **efficient** if the array is sorted.
- A binary search is used whenever the list starts to become **large**.
- Consider to use binary searches whenever the list contains more than **16** elements.
- The binary search starts by testing the data in the element at the **middle** of the array to determine if the **target** is in the first or **second half** of the list.
- If it is in the **first half**, we do not need to check the **second half**. If it is in the **second half**, we do not need to test the **first half**. In other words we eliminate half the list from further consideration. We repeat this process until we find the target or determine that it is not in the list.
- To find the middle of the list, we need three variables, one to identify the beginning of the list, one to identify the middle of the list, and one to identify the end of the list.

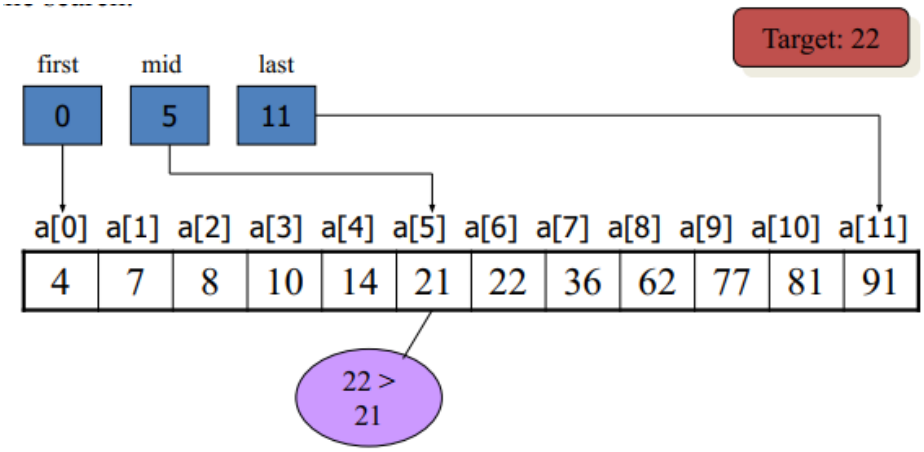
Working of Binary Search

For a binary search to work, it is mandatory for the target array to be **sorted**.

Target found case: Assume we want to find key = **22** in a sorted list as follows:

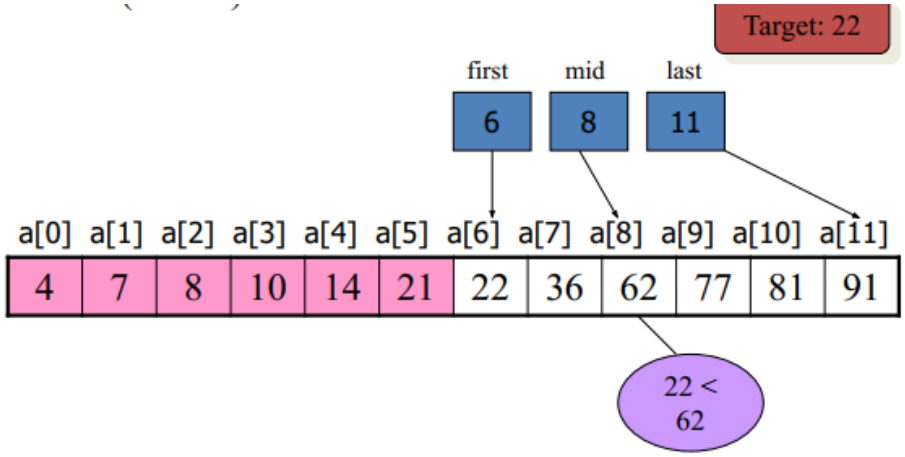
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]
4	7	8	10	14	21	22	36	62	77	81	91

The three indexes are first, mid and last. Given **first** as 0 and **last** as 11, **mid** is calculated as follows:
mid = (first + last) / 2
mid = (0 + 11) / 2 = 11 / 2 = 5 (**Integer**)
At index location **5**, the target is greater than the list value (22 > 21).



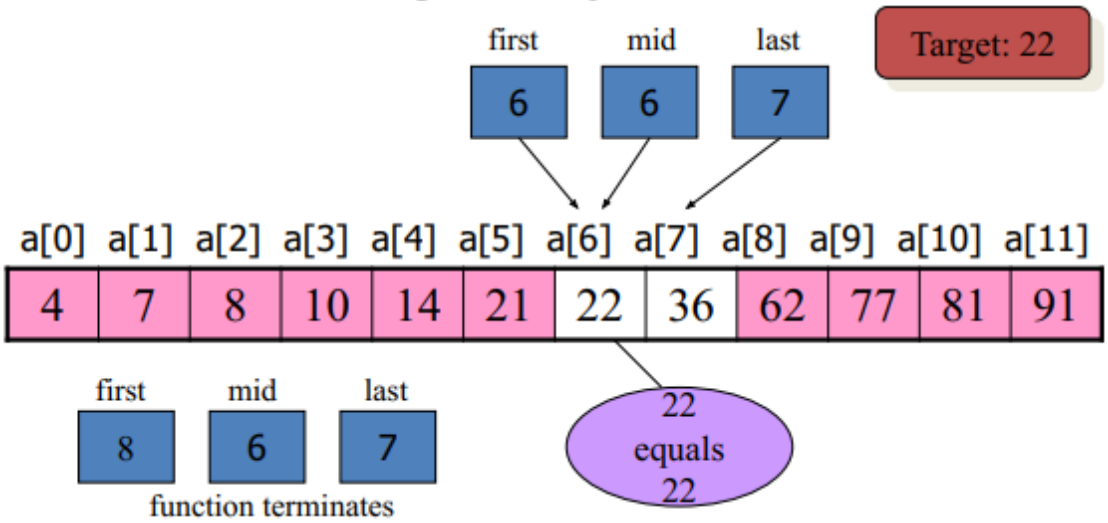
Therefore, **eliminate** the array locations 0 through 5 (mid is automatically Eliminated).
To narrow our search, we assign mid + 1 to first and repeat the search (last would be same 11).
first = mid +1 = 5+1 =6

mid = (6 + 11) / 2 = 17 / 2 = 8



end= mid-1;

When we test the target to the value at mid a second time, we discover that the target is less than the list value (22 < 62). This time we adjust the end of the list by setting last to mid – 1 and recalculate mid. This step effectively eliminates elements 8 through 11 from consideration. We have now arrived at index location 6, whose **value matches** our **target**. This stops the search.



Binary Search in C++

```
6  int binary(int arr[], int n, int k)
7  {
8      int start = 0;    // Start of array (Lower Bound)
9      int end = n - 1;  // End of array (Upper Bound)
10     // Continue while the lower bound is less than or equal to the
           upper bound
11     while (start <= end)
12     {
13         int mid = (start + end) / 2;  // Finding the middle index
14         if (arr[mid] == k)  // If the middle element is equal to the
               key
15         {
16             return mid;  // Return the index of the key
17         }
18         else if (arr[mid] > k)  // If the middle element is greater
               than the key
19         {
20             end = mid - 1;  // Search the lower portion of the array
21         }
```

```
22          else
23          {
24              start = mid + 1;  // Search the upper portion of the
                    array
25          }
26     }
27     return -1;  // Return -1 if the key is not found
28 }
```

```
int main()
{
    int n, k;  // Declare variables for size of array and key to
        search

    // Get array size from the user
    cout << "Enter the size of array: ";
    cin >> n;

    // Dynamically allocate memory for the array
    int* arr = new int[n];

    // Get elements of the array from the user
    for (int i = 0; i < n; i++)
    {
        cout << "Enter Element No " << i + 1 << ": ";
        cin >> arr[i];
    }
```

```
    // Get the key to search for
    cout << "Enter the value you want to search: ";
    cin >> k;

    // Perform binary search and print the result
    int result = binary(arr, n, k);
    if (result != -1)
    {
        cout << "Item found at index: " << result << endl;
    }
    else
    {
        cout << "Item not found in the array." << endl;
    }

    // Free the dynamically allocated memory
    delete[] arr;
    return 0;
}
```

```
/tmp/GvtoMtz98a.o
Enter the size of array: 5
Enter Element No 1: 23
Enter Element No 2:
4
Enter Element No 3: 5
Enter Element No 4: 6
Enter Element No 5: 7
Enter the value you want to search: 6
Item found at index: 3


=== Code Execution Successful ===
```

Guessing number game

Lab Tasks.

1. Given a sorted array of n integers and a target value, find the index of the target value in the array. Show the all active items [divided half] at every stage. Handle the situation when the target value is not found.
2. Given a sorted array of n integers and a target value, find the first occurrence of the target value in the array.
3. Given a sorted array of n integers and a target value, find the last occurrence of the target value in the array.
4. Given a sorted array of n integers and a target value, find the number of occurrences of the target value in the array.