Data Structures Lab 4 (B)

Course: Data Structures (CL2001) **Semester:** Fall 2024
Instructor: Muhammad Nouman Hanif

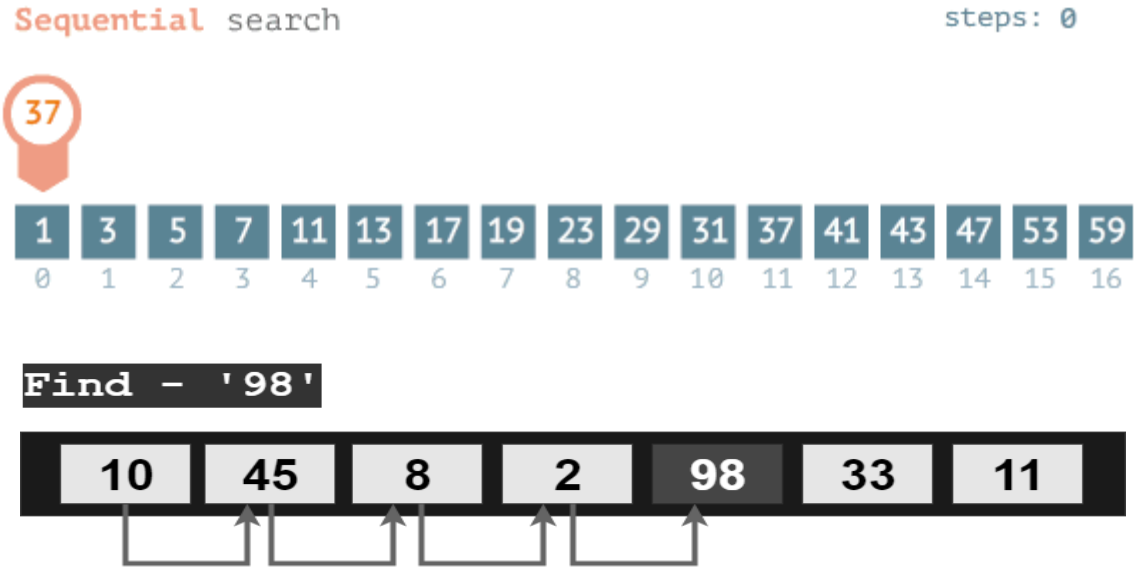
Note:

- Lab manual cover following below searching algorithms
  {Linear Searching, Binary Searching, Interpolation Search}
- Maintain discipline during the lab.
- Just raise your hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session.

Searching Algorithms

Linear Search

Linear search algorithm or **sequential search** is a method for finding an element within a list. It sequentially checks each element of the list until a match is found or the whole list has been searched.



Pseudocode:

```
LinearSearch(arr[], n, target)
{
    for (i = 0 to n-1)
    {
        if (arr[i] == target)
            return i
    }

    return -1
}
```

➔ **Every element is considered as a potential match** for the key and checked for the same.

➔ If any **element is found equal to the key**, the search is successful and the index of that element is returned.

➔ If no element is found equal to the key, the search yields “No match found”.

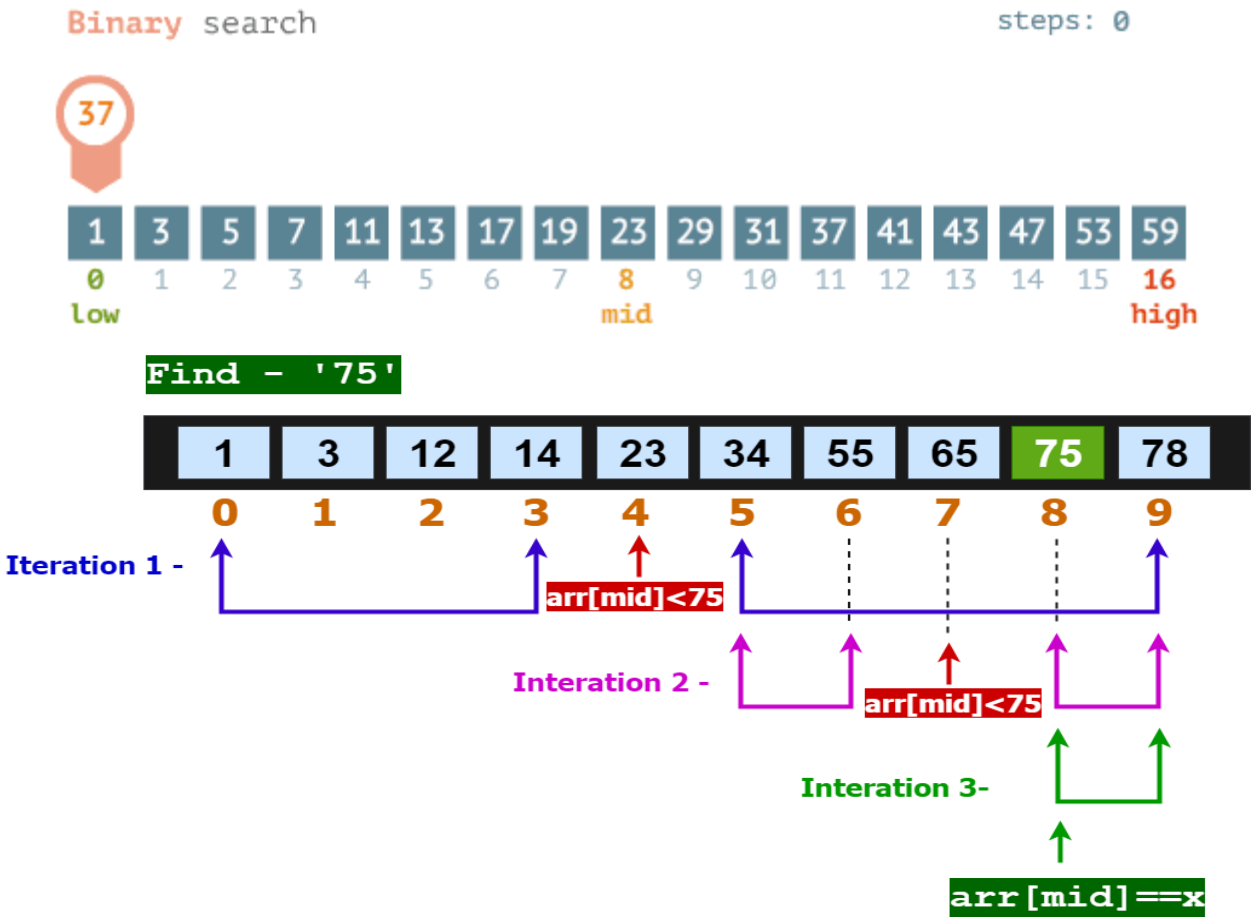
Complexity Analysis of Linear Search:

➔ **Time Complexity:**

- Worst Case - O(N)
- Average Case - O(N)
- Best Case - O(1)

Binary Search

Binary search algorithm falls under the category of interval search algorithms (Sorted Data). This algorithm is much more efficient compared to linear search algorithm. Binary search only works on sorted data structures. This algorithm repeatedly targets the center of the sorted data structure & divide the search space into half till the match is found.



Pseudocode:

```
BinarySearch(arr[], n, target)
{
    // Initialize the low and high indices
    low = 0
    high = n - 1

    // Continue searching while there is a valid range
    while (low <= high)
    {
        // Find the middle index
        mid = low + (high - low) / 2

        // Check if the target is at the middle
        if (arr[mid] == target)
            return mid

        // If target is smaller, discard the right half
        else if (arr[mid] > target)
            high = mid - 1

        // If target is larger, discard the left half
        else
            low = mid + 1
    }
    return -1
}
```

➔ Divide the search space into **two halves by finding the middle index** "mid".
➔ **Compare the middle element** of the search space with the key.
➔ If the key is **found at middle** element, the process is terminated.
➔ If the key is **not found at middle** element, choose which half will be used as the next search.
➔ If the key is **smaller than the middle** element, then the left side is used for next search.
➔ If the key is **larger than the middle** element, then the right side is used for next search.
➔ This process is continued until the key is found.

Complexity Analysis of Binary Search:

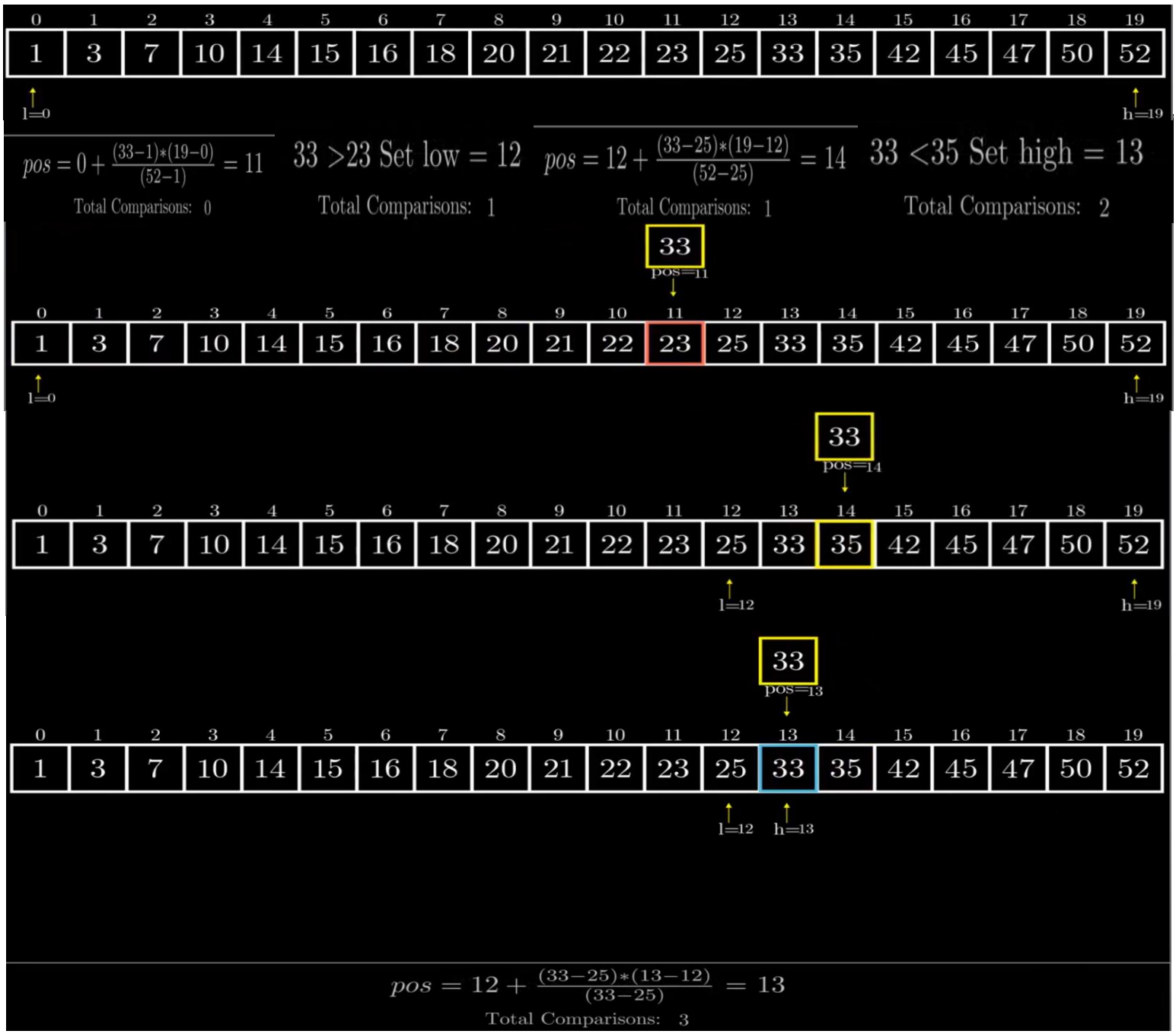
➔ **Time Complexity:**
- Worst Case - O(Log N)
- Average Case - O(Log N)
- Best Case - O(1)

Interpolation Search

Interpolation search is an improvement over binary search. Binary Search always checks the value at middle index. But interpolation search may check at different locations based on the value of element being searched. For interpolation search to work efficiently the **array elements/data should be sorted and uniformly distributed**.

Search key =33

$$\textbf{Position} = start\ index + \frac{(element - Arr[start\ index]) * (end\ index - start\ index)}{Arr[end\ index] - Arr[start\ index]}$$



Pseudocode:

```
InterpolationSearch(arr[], n, target)
{
    // Initialize the low and high indices
    low = 0
    high = n - 1

    // Continue searching while the target is within the search range
    while (low <= high and target >= arr[low] and target <= arr[high])
    {
        // Estimate the position of the target using interpolation
formula
        pos = low + ((target - arr[low]) * (high - low)) / (arr[high]
- arr[low])

        // Check if the target is at the estimated position
        if (arr[pos] == target)
            return pos

        // If target is larger, discard the left half
        else if (arr[pos] < target)
            low = pos + 1

        // If target is smaller, discard the right half
        else
            high = pos - 1
    }

    // If the target is not found, return -1
    return -1
}
```

➔ In a loop, calculate the value of "pos" using the probe **position formula**.

➔ If it is a match, **return the index of the item**, and exit.

➔ If the item is **less than** arr[pos], calculate the probe position of the **left sub-array**. Otherwise, calculate the same in the **right sub-array**.

➔ Repeat until a match is found or the **sub-array reduces to zero**.

Complexity Analysis of Interpolation Search:

➔ **Time Complexity:**

- Worst Case - O(N)
- Average Case - O(Log (Log N))
- Best Case - O(1)a

Lab Exercises

1. Write a C++ program to implement Linear Search. Your program should take an array of integers and a target value as input from the user. The program should search for the target value in the array using the linear search algorithm and output the index at which the target is found. If the target value is not found in the array, the program should display an appropriate message.

2. You've been given an array of numbers representing employee IDs. Your task is to identify the employee whose ID matches the last two digits of your roll number. If your roll number's last two digits are not present in the array, insert the missing value in its correct position within the array. You must use binary search to locate the position of that value within the array.

3. Your team has been given a large dataset (input by user) of sorted, uniformly distributed account balances. If the data is not sorted, you have to sort it first. If the data is not uniformly distributed after you apply sorting (if necessary) you can prompt an error. Your manager has asked you to implement Interpolation Search because it estimates the position of the target value based on the data distribution. This will allow the search to "jump" closer to the target in fewer iterations.