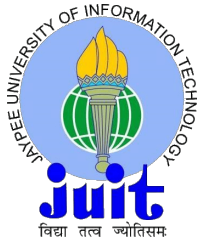


Data Structure

(Course Code: 25B11CI318)

Linear and Binary Search

Mr. Sandeep Kumar Patel

Assistant Professor

Jaypee University of Information Technology

Waknaghat, Solan, HP-173234

Linear Search

Definition

Linear Search is a simple searching algorithm that checks each element of a list sequentially until the required element is found or the end of the list is reached.

Linear Search:

- Starts from the first element.
- Compares each element with the search key.
- Stops when the key is found.
- If the key is not found, returns -1.

$$\boxed{\text{Sequential Search} \Rightarrow \text{Linear Search}}$$

Prerequisite of Linear Search

Important

Linear Search does **not require the data to be sorted**.

It can search in:

- Sorted arrays
- Unsorted arrays
- Linked lists

Example:

$$A = [45,\ 12,\ 78,\ 23,\ 9,\ 56]$$

Search for:

$$\boxed{23}$$

The elements are checked one by one.

Input and Output

Input

- An array A containing n elements.
- A search key *key*.

Output

- Index of the key if it is present.
- -1 if the key is not present.

Example:

$$A = [10,\ 20,\ 30,\ 40,\ 50]$$

Search:

Idea of Linear Search

The algorithm examines elements from left to right.

For:

$$A = [10,\ 25,\ 15,\ 30,\ 45]$$

and:

$$key = 30$$

the search proceeds as:

$$10 \rightarrow 25 \rightarrow 15 \rightarrow 30$$

At every position:

- If $A[i] = key$, return i.
- Otherwise, move to the next element.

Linear Search: Example

Consider:

$$A = [10,\ 25,\ 15,\ 30,\ 45,\ 50]$$

Search for:

$$\boxed{30}$$

1. Compare 30 with 10.

$$30 \neq 10$$

2. Compare 30 with 25.

$$30 \neq 25$$

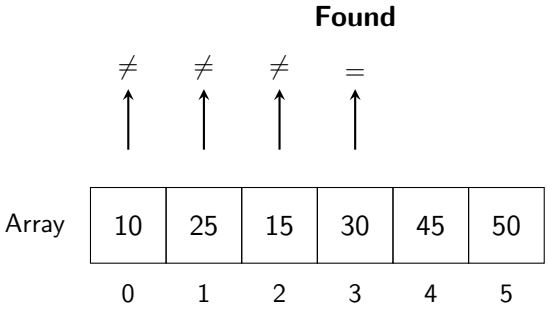
3. Compare 30 with 30.

$$30 = 30$$

Result

Element found at index 3

Linear Search: Visual Trace



$key = 30, \qquad index = 3$

Linear Search Algorithm

1. Start from the first element:
$$i = 0$$
2. Compare $A[i]$ with key.
3. If:
$$A[i] = key$$
return i.
4. Otherwise, increment:
$$i = i + 1$$
5. Repeat until:
$$i = n$$
6. If the loop ends, return:
$$-1$$

Linear Search: Pseudocode

```
LinearSearch(A, n, key)

for i = 0 to n-1
{
    if A[i] == key
        return i;
}

return -1;
```

Search sequentially from left to right

Linear Search: Element Not Found

Consider:

$$A = [10,\ 25,\ 15,\ 30,\ 45]$$

Search for:

$$\boxed{50}$$

Comparisons:

$$50 \neq 10$$

$$50 \neq 25$$

$$50 \neq 15$$

Time Complexity of Linear Search

Case	Complexity	Situation
Best Case	$O(1)$	Key is first element
Average Case	$O(n)$	Key is somewhere in the array
Worst Case	$O(n)$	Key is last or absent

Worst Case

The algorithm may need to compare the key with all n elements.

$$\boxed{T(n) = n \Rightarrow O(n)}$$

Linear Search: Best Case

Consider:

$$A = [10,\ 20,\ 30,\ 40,\ 50]$$

Search:

$$key = 10$$

The first comparison is successful:

$$A[0] = 10$$

Therefore, only one comparison is required.

$$T(n) = 1$$

Hence:

$$\boxed{T(n) = O(1)}$$

Best Case

The element is found at the first position.

Linear Search: Worst Case

Consider:

$$A = [10,\ 20,\ 30,\ 40,\ 50]$$

Search:

$$key = 50$$

The algorithm checks:

$$10 \rightarrow 20 \rightarrow 30 \rightarrow 40 \rightarrow 50$$

Therefore:

$$T(n) = n$$

Hence:

$$\boxed{T(n) = O(n)}$$

The same complexity occurs when the element is absent.

Linear Search: Average Case

Assume the element is equally likely to occur at any position.
Number of comparisons:

$$1, 2, 3, \ldots, n$$

Average number of comparisons:

$$\frac{1 + 2 + 3 + \cdots + n}{n}$$

Using:

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

we get:

$$\frac{n+1}{2}$$

Space Complexity of Linear Search

The iterative Linear Search uses only a few variables:

- i
- key
- Temporary variables

No additional array or data structure is required.
Therefore:

$$\boxed{S(n) = O(1)}$$

Important

Linear Search is memory efficient because its iterative implementation requires constant auxiliary space.

Can Linear Search Work on Sorted Data?

Yes.

Consider:

$$A = [10,\ 20,\ 30,\ 40,\ 50]$$

Search for:

$$key = 25$$

We compare:

$$25 > 10$$

$$25 > 20$$

$$25 < 30$$

Linear Search vs Binary Search

Feature	Linear Search	Binary Search
Data Requirement	Unsorted/Sorted	Sorted Only
Technique	Sequential	Divide and Conquer
Best Case	$O(1)$	$O(1)$
Average Case	$O(n)$	$O(\log n)$
Worst Case	$O(n)$	$O(\log n)$
Iterative Space	$O(1)$	$O(1)$

Key Difference

Linear Search eliminates **one element at a time**, while Binary Search eliminates approximately **half of the search space** at every step.

Linear Search vs Binary Search: Intuition

For $n = 16$ elements:

Linear Search:

$$16 \to 15 \to 14 \to \cdots \to 1$$

Worst case:

$$\boxed{16 \text{ comparisons}}$$

Binary Search:

$$16 \to 8 \to 4 \to 2 \to 1$$

Worst case:

$$\boxed{4 \text{ comparisons}}$$

Applications of Linear Search

Linear Search is useful when:

- The dataset is small.
- The data is unsorted.
- Searching is performed only occasionally.
- The data structure does not support random access.
- Simplicity is more important than search speed.

Examples:

- Searching an unsorted array.
- Searching a linked list.
- Finding a particular record in a small dataset.

Advantages and Limitations

Advantages

- Very simple to implement.
- Does not require sorted data.
- Works with arrays and linked lists.
- Requires only $O(1)$ auxiliary space.

Limitations

- Slow for large datasets.
- Worst-case time is $O(n)$.
- May require checking every element.

Linear Search: Key Takeaways

1. Linear Search examines elements sequentially.

Linear Search: Key Takeaways

1. Linear Search examines elements sequentially.
2. It does not require sorted data.

Linear Search: Key Takeaways

1. Linear Search examines elements sequentially.
2. It does not require sorted data.
3. Best-case complexity:

$$O(1)$$

Linear Search: Key Takeaways

1. Linear Search examines elements sequentially.
2. It does not require sorted data.
3. Best-case complexity:

$$O(1)$$

4. Average-case complexity:

$$O(n)$$

Linear Search: Key Takeaways

1. Linear Search examines elements sequentially.
2. It does not require sorted data.
3. Best-case complexity:

$$O(1)$$

4. Average-case complexity:

$$O(n)$$

5. Worst-case complexity:

$$O(n)$$

Linear Search: Key Takeaways

1. Linear Search examines elements sequentially.
2. It does not require sorted data.
3. Best-case complexity:

$$O(1)$$

4. Average-case complexity:

$$O(n)$$

5. Worst-case complexity:

$$O(n)$$

6. Iterative space complexity:

$$O(1)$$

Binary Search

Definition

Binary Search is an efficient searching algorithm used to find an element in a **sorted array**.

Instead of checking every element sequentially, Binary Search:

- Compares the search key with the middle element.
- Eliminates half of the search space.
- Repeats the process on the remaining half.

Key Idea

Binary Search follows the **Divide and Conquer** strategy.

$$\text{Search Space} \longrightarrow \frac{n}{2} \longrightarrow \frac{n}{4} \longrightarrow \frac{n}{8} \longrightarrow \cdots$$

Prerequisite of Binary Search

Important

Binary Search works only when the data is arranged in **sorted order**.

Example of valid input:

$$[2,\ 5,\ 8,\ 12,\ 16,\ 23,\ 38,\ 45,\ 56,\ 72]$$

Ascending order:

$$A[0] \leq A[1] \leq A[2] \leq \cdots \leq A[n-1]$$

If the array is unsorted:

$$[23,\ 5,\ 45,\ 12,\ 8]$$

Binary Search cannot be directly applied

Input and Output

Input

- A sorted array A containing n elements.
- A search key key.

Output

- Index of the search key if it is present.
- -1 if the element is not present.

For example:

$$A = [10,\ 20,\ 30,\ 40,\ 50]$$

Search:

Idea of the Algorithm

Binary Search compares the target with the **middle element**.

1. Find the middle element.
2. If:
$$key = A[mid]$$
the element is found.
3. If:
$$key < A[mid]$$
search only the **left half**.
4. If:
$$key > A[mid]$$
search only the **right half**.
5. Repeat until the element is found or the search space becomes empty.

How Binary Search Reduces the Search Space

Suppose we have:

$$[2,\ 5,\ 8,\ 12,\ 16,\ 23,\ 38,\ 45]$$

Initially:

$$n = 8$$

After first comparison:

$$\frac{8}{2} = 4$$

After second comparison:

$$\frac{4}{2} = 2$$

After third comparison:

Binary Search: Example

Consider:

$$A = [2,\ 5,\ 8,\ 12,\ 16,\ 23,\ 38,\ 45,\ 56,\ 72]$$

Search for:

$$\boxed{23}$$

Initially:

$$low = 0, \qquad high = 9$$

$$mid = \frac{0 + 9}{2} = 4$$

Therefore:

$$A[mid] = A[4] = 16$$

Binary Search: Iteration 2

Current search space:

$$[23,\ 38,\ 45,\ 56,\ 72]$$

Indices:

$$5,\ 6,\ 7,\ 8,\ 9$$

Therefore:

$$low = 5, \qquad high = 9$$

$$mid = \frac{5+9}{2} = 7$$

$$A[7] = 45$$

Since:

Binary Search: Iteration 3

Current search space:

$$[23,\ 38]$$

Indices:

$$5,\ 6$$

Therefore:

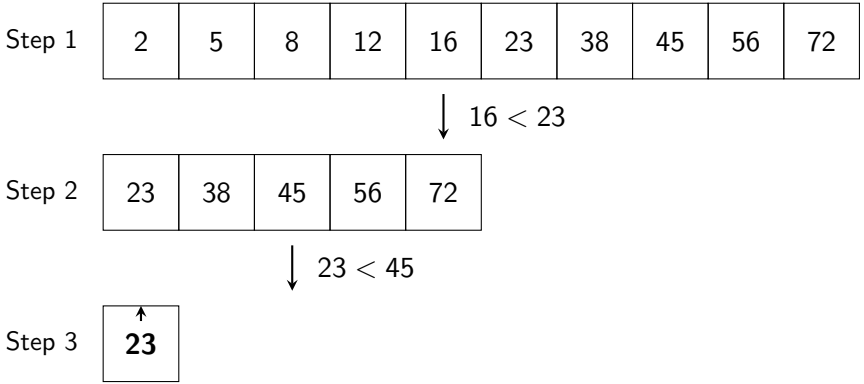
$$mid = \frac{5+6}{2} = 5$$

$$A[5] = 23$$

Since:

$$A[5] = key$$

Binary Search: Visual Trace



Found at index 5

Binary Search Algorithm

1. Set:
$$low = 0, \qquad high = n - 1$$
2. Repeat while:
$$low \leq high$$
3. Calculate:
$$mid = low + \frac{high - low}{2}$$
4. Compare key with $A[mid]$.
5. If:
$$key = A[mid]$$
return mid.
6. If:
$$key < A[mid]$$
set:

Iterative Binary Search

```
BinarySearch(A, key)
low = 0
high = length(A) - 1
while low <= high
{
    mid = low + (high - low) / 2;
    if(A[mid] == key)
        return mid;
    else if(key < A[mid])
        high = mid - 1;
    else
        low = mid + 1;
}
return -1;
```

Calculating the Middle Index

The traditional formula is:

$$mid = \frac{low + high}{2}$$

However, in some programming languages, $low + high$ can cause integer overflow.
Example:

$$low = 5, \qquad high = 9$$

$$mid = 5 + \frac{9 - 5}{2}$$

$$mid = 5 + 2 = 7$$

Programming Practice

Using $low + (high - low)/2$ is generally preferred in implementation.

Recursive Binary Search

```
BinarySearch(A, low, high, key)
{
    if(low > high)
        return -1;
    mid = low + (high-low)/2;
    if(A[mid] == key)
        return mid;
    if(key < A[mid])
        return BinarySearch(A,
                            low, mid-1, key);
    return BinarySearch(A,
                        mid+1, high, key);
}
```

Binary Search Recursion

At every recursive call:

$$n \longrightarrow \frac{n}{2}$$
$$\frac{n}{2} \longrightarrow \frac{n}{4}$$
$$\frac{n}{4} \longrightarrow \frac{n}{8}$$

Continue until:

$$\frac{n}{2^k} = 1$$

Therefore:

$$n = 2^k$$

Taking logarithm:

$$k = \log_2 n$$

Hence:

$$\boxed{T(n) = O(\log n)}$$

Recurrence Relation

For recursive Binary Search:

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

Expanding:

$$T(n) = T\left(\frac{n}{4}\right) + O(1) + O(1)$$

After k levels:

$$T(n) = T\left(\frac{n}{2^k}\right) + O(k)$$

At the base case:

$$\frac{n}{2^k} = 1$$

Therefore:

$$k = \log_2 n$$

Hence:

$$\boxed{T(n) = O(\log n)}$$

Time Complexity of Binary Search

Case	Complexity	Reason
Best Case	$O(1)$	Key is at middle
Average Case	$O(\log n)$	Half eliminated each time
Worst Case	$O(\log n)$	Repeated halving

Key Result

Binary Search reduces the search space by half at every step.

$$\boxed{\text{Worst Case Time} = O(\log n)}$$

Space Complexity

Implementation	Space Complexity
Iterative Binary Search	$O(1)$
Recursive Binary Search	$O(\log n)$

Why?

- Iterative version uses only a few variables.
- Recursive version creates function-call stack frames.

Linear Search vs Binary Search

Feature	Linear Search	Binary Search
Data	Unsorted/Sorted	Sorted
Technique	Sequential	Divide and Conquer
Best Case	$O(1)$	$O(1)$
Average Case	$O(n)$	$O(\log n)$
Worst Case	$O(n)$	$O(\log n)$
Iterative Space	$O(1)$	$O(1)$

Conclusion

For large **sorted** datasets, Binary Search is significantly faster than Linear Search.

Applications of Binary Search

Binary Search is useful in:

- Searching in sorted arrays.
- Dictionary and phone-book lookups.
- Database indexing.
- Searching symbol tables.
- Finding lower and upper bounds.
- Finding insertion positions.
- Searching a monotonic answer space.

Important Extension

The idea of Binary Search can also be applied to problems where the answer space is **monotonic**, even when we are not directly searching for an array element.

Limitations of Binary Search

- Data must be sorted.
- Sorting an unsorted array requires additional time.
- Insertion and deletion in a sorted array may be expensive.
- Not efficient for linked lists because random access is not available.

Remember

Binary Search is highly efficient when the data is sorted and supports efficient random access.

Binary Search: Key Takeaways

1. Binary Search requires **sorted data**.

Binary Search: Key Takeaways

1. Binary Search requires **sorted data**.
2. It follows the **Divide and Conquer** strategy.

Binary Search: Key Takeaways

1. Binary Search requires **sorted data**.
2. It follows the **Divide and Conquer** strategy.
3. Each comparison eliminates approximately half of the search space.

Binary Search: Key Takeaways

1. Binary Search requires **sorted data**.
2. It follows the **Divide and Conquer** strategy.
3. Each comparison eliminates approximately half of the search space.
4. Best-case time complexity:

$$O(1)$$

Binary Search: Key Takeaways

1. Binary Search requires **sorted data**.
2. It follows the **Divide and Conquer** strategy.
3. Each comparison eliminates approximately half of the search space.
4. Best-case time complexity:

$$O(1)$$

5. Average and worst-case time complexity:

$$O(\log n)$$

Binary Search: Key Takeaways

1. Binary Search requires **sorted data**.
2. It follows the **Divide and Conquer** strategy.
3. Each comparison eliminates approximately half of the search space.
4. Best-case time complexity:

$$O(1)$$

5. Average and worst-case time complexity:

$$O(\log n)$$

6. Iterative space complexity:

$$O(1)$$

Binary Search: Key Takeaways

1. Binary Search requires **sorted data**.
2. It follows the **Divide and Conquer** strategy.
3. Each comparison eliminates approximately half of the search space.
4. Best-case time complexity:

$$O(1)$$

5. Average and worst-case time complexity:

$$O(\log n)$$

6. Iterative space complexity:

$$O(1)$$

7. Recursive space complexity:

Thank You

Questions?