Data Structures & Algorithms (C++)

Searching Algorithms: Sequential Search & Binary Search

1. Sequential Search (Linear Search)

Introduction to Sequential Search

Sequential search (also called linear search) is the simplest searching technique in which each element of the array is checked one by one until the desired element is found.

It does not require the data to be sorted, making it very flexible and easy to implement.

It is widely used in small datasets or unsorted collections where sorting is not possible or unnecessary.

Definition and Purpose

Sequential search is a method of searching where each element is compared with the target value in order.

Its main purpose is to locate an element in a list by scanning each element sequentially.

It ensures correctness but may take more time for large datasets.

When to Use Sequential Search

- When data is unsorted
- When dataset is small
- When simplicity is more important than speed
- When frequent updates happen in data

Pseudocode for Sequential Search

```
SEQUENTIAL_SEARCH(arr, n, key)

for i = 0 to n-1
   if arr[i] == key
      return i
return -1
```

Working Principle

Sequential search starts from the first element and compares each element with the target.

If a match is found, the index is returned immediately.

If no match is found after checking all elements, the search fails.

Time Complexity

Case	Complexity
Best Case	O(1)
Worst Case	O(n)
Average Case	O(n)

Space Complexity

- O(1) (constant space)
- No extra memory required

Advantages

- Simple to implement
- Works on unsorted data
- No preprocessing required
- Suitable for small datasets

Disadvantages

- Very slow for large datasets
- Inefficient compared to binary search
- Needs scanning of all elements in worst case

C++ Implementation (Sequential Search)

```
#include <iostream>
using namespace std;

int linearSearch(int arr[], int n, int key) {
   for (int i = 0; i < n; i++) {
      if (arr[i] == key)
         return i;
```

```
    }
    return -1;
}

int main() {
    int arr[] = {10, 25, 30, 45, 60};
    int n = 5;
    int key = 30;

    int result = linearSearch(arr, n, key);

    if (result != -1)
        cout << "Element found at index: " << result << endl;
    else
        cout << "Element not found" << endl;

    return 0;
}
```

2. Binary Search

Introduction to Binary Search

Binary search is a highly efficient searching algorithm used on sorted arrays.

It works by repeatedly dividing the search interval into half until the target value is found.

It is much faster than sequential search but requires sorted data.

Definition and Use Cases

Binary search is a divide-and-conquer algorithm that compares the target value with the middle element of the array.

It is commonly used in databases, dictionaries, and large sorted datasets.

It significantly reduces search time compared to linear scanning.

Requirements

- Data must be sorted (ascending or descending)
- Random access to elements (array or vector)

Pseudocode for Binary Search

```
BINARY_SEARCH(arr, left, right, key)

while left <= right
   mid = (left + right) / 2

   if arr[mid] == key
      return mid
   else if arr[mid] < key
      left = mid + 1
   else
      right = mid - 1

return -1
```

Working of Binary Search

Binary search compares the key with the middle element.

If the key is smaller, it searches the left half.

If the key is larger, it searches the right half.

This process continues until the element is found or the range becomes empty.

Why Binary Search is Faster

Binary search reduces the search space by half at each step.

This makes its complexity **O(log n)**, which is much faster than linear search O(n).

For large datasets, this difference becomes extremely significant.

Time Complexity

Case	Complexity
Best Case	O(1)
Worst Case	O(log n)
Average Case	O(log n)

Space Complexity

- Iterative: O(1)

- Recursive: O(log n) (call stack)

Advantages

- Very fast for large datasets
- Efficient (logarithmic time)
- Reduces number of comparisons
- Ideal for sorted data

Disadvantages

- Requires sorted data
- Not useful for linked lists (inefficient access)
- More complex than linear search

C++ Implementation (Binary Search)

```
#include <iostream>
using namespace std;

int binarySearch(int arr[], int n, int key) {
   int left = 0, right = n - 1;

   while (left <= right) {
      int mid = (left + right) / 2;

      if (arr[mid] == key)
         return mid;

      else if (arr[mid] < key)
         left = mid + 1;

      else
         right = mid - 1;
   }

   return -1;
}

int main() {
   int arr[] = {10, 20, 30, 40, 50, 60};
   int n = 6;
   int key = 40;

   int result = binarySearch(arr, n, key);

   if (result != -1)
      cout << "Element found at index: " << result << endl;
```

```
    else
        cout << "Element not found" << endl;

    return 0;
}
```

Comparison: Sequential vs Binary Search

Feature	Sequential Search	Binary Search
Data Requirement	Unsorted	Sorted
Time Complexity	O(n)	O(log n)
Speed	Slow	Very fast
Implementation	Easy	Moderate
Best Use	Small data	Large sorted data

Key Takeaways

- Sequential search checks elements one by one.
- Binary search divides the search space into halves.
- Sequential search works on unsorted data.
- Binary search requires sorted data.
- Binary search is significantly faster for large datasets.
- Sequential search is simple but inefficient for big data.
- Binary search is a classic **divide-and-conquer algorithm**.