

DATA STRUCTURES

NAME : B.INDRA

REG NO : 192365047

COURSE CODE : CSA0312

**TOPIC : LINEAR & BINARY SEARCH, COMPARISON
B/W THEM.**

#LINEAR SEARCH

~Algorithm:-

1. **Start** with the first element in the list.
2. Compare the current element with the target (key).
3. If the current element matches the target, return the index of the element.
4. If the current element does not match, move to the next element.
5. Repeat steps 2–4 until you either find the target or reach the end of the list.
6. If you reach the end of the list and do not find the target, return -1 (indicating that the element is not present).

LinearSearch(arr, n, key)

Input: arr is an array of n elements, key is the value to search for

Output: Return the index of the key if found, otherwise -1

1. For $i = 0$ to $n-1$:
 - a. If $arr[i] == key$:
 - i. Return i (index of the found key)
2. If the key is not found after checking all elements:
 - a. Return -1

EXAMPLE:-

- Input Array: [10, 24, 5, 78, 2]
- Target Key: 78
- Process:
 - Move to index 1, compare 24 with 78 → no match
 - Move to index 2, compare 5 with 78 → no match
 - Move to index 3, compare 78 with 78 → match found
- Output: Index 3

~Advantages and disadvantages

Advantages:-

1. Simplicity: Linear search is very simple to implement.
2. No Need for Sorting: It works on unsorted data, unlike binary search which requires a sorted array.

Disadvantages:-

1. Lack of Optimizations: No shortcuts to find the element; it examines every element one by one.
2. Not Efficient for Large Datasets: As the size of the dataset grows, linear search becomes slower and inefficient.

~ COMPILER EXECUTION:-

#INPUT:-

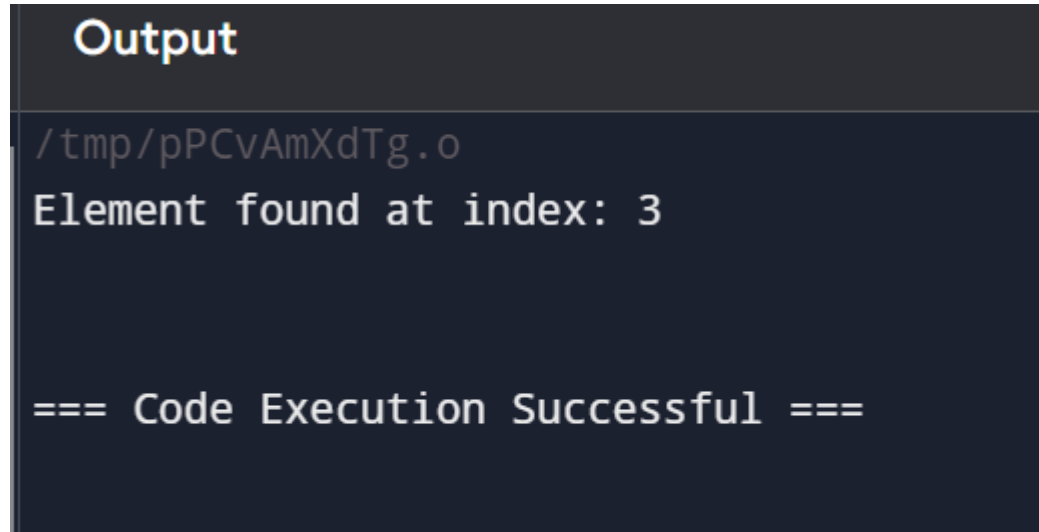
```
#include <stdio.h>
```

```
int linearSearch(int arr[], int size, int key) {  
    for (int i = 0; i < size; i++) {  
        if (arr[i] == key) {  
            return i;  
        }  
    }  
    return -1;  
}
```

```
int main() {  
    int arr[] = {2, 4, 6, 8, 10, 12};  
    int size = sizeof(arr) / sizeof(arr[0]);  
    int key = 8;  
  
    int result = linearSearch(arr, size, key);  
  
    if (result != -1) {  
        printf("Element found at index: %d\n", result);  
    } else {  
        printf("Element not found\n");  
    }  
}
```

```
    return 0;  
}
```

#OUTPUT:-

A screenshot of a terminal window with a dark background. The title bar at the top says 'Output'. Below the title bar, the text '/tmp/pPCvAmXdTg.o' is visible. The main output text reads 'Element found at index: 3' followed by '=== Code Execution Successful ===' on a new line.

```
Output  
/tmp/pPCvAmXdTg.o  
Element found at index: 3  
  
=== Code Execution Successful ===
```

#BINARY SEARCH

~Algorithm:-

1. **Start** with two pointers: low pointing to the first element and high pointing to the last element of the sorted array.
2. Find the middle index: $\text{mid} = (\text{low} + \text{high}) / 2$.
3. Compare the target (key) with the middle element:
 - If the middle element is equal to the target, return the index mid.
 - If the target is less than the middle element, set $\text{high} = \text{mid} - 1$ (search the left half).
 - If the target is greater than the middle element, set $\text{low} = \text{mid} + 1$ (search the right half).
4. Repeat steps 2 and 3 until low is greater than high.

5. If the element is not found, return -1.

BinarySearch(arr, n, key)

Input: arr is a sorted array of n elements, key is the value to search for

Output: Return the index of the key if found, otherwise -1

1. Set low = 0 and high = n-1
2. While low <= high:
 - a. Set mid = (low + high) / 2
 - b. If arr[mid] == key, return mid
 - c. If arr[mid] < key, set low = mid + 1
 - d. Else, set high = mid - 1
3. If the key is not found, return -1

Advantages:-

1. **Efficiency:** Binary search has a time complexity of $O(\log_{10} n)$ or $O(\log n)$, making it much faster than linear search for large datasets.
2. **Fewer Comparisons:** It significantly reduces the number of comparisons needed to find an element.
3. **Works Well on Large Datasets:** Especially useful for large, sorted arrays where linear search becomes inefficient.
4. **Recursive or Iterative:** It can be implemented in both recursive and iterative approaches.

Disadvantages:-

1. **Requires Sorted Data:** Binary search only works on sorted datasets, which may add overhead if sorting is needed first.
2. **Not Suitable for Small Datasets:** For very small datasets, linear search might be more efficient due to its simplicity.
3. **Fixed Data Structures:** It is less efficient for dynamic data structures like linked lists where accessing the middle element takes linear time.

4. **Not Adaptive:** Unlike some other algorithms, binary search does not adapt well to changes in the data unless the array remains sorted.

~ COMPILER EXECUTION:-

#INPUT:-

```
#include <stdio.h>
```

```
int binarySearch(int arr[], int size, int key) {
```

```
    int low = 0, high = size- 1;
```

```
    while (low <= high) {
```

```
        int mid = low + (high- low) / 2;
```

```
        if (arr[mid] == key) {
```

```
            return mid;
```

```
        }
```

```
        if (arr[mid] < key) {
```

```
            low = mid + 1;
```

```
        } else {
```

```
            high = mid- 1;
```

```
        }
```

```
    }
```

```
    return -1;
```

```
}
```

```
int main() {
```

```
    int arr[] = {2, 4, 6, 8, 10, 12};
```

```
    int size = sizeof(arr) / sizeof(arr[0]);
```

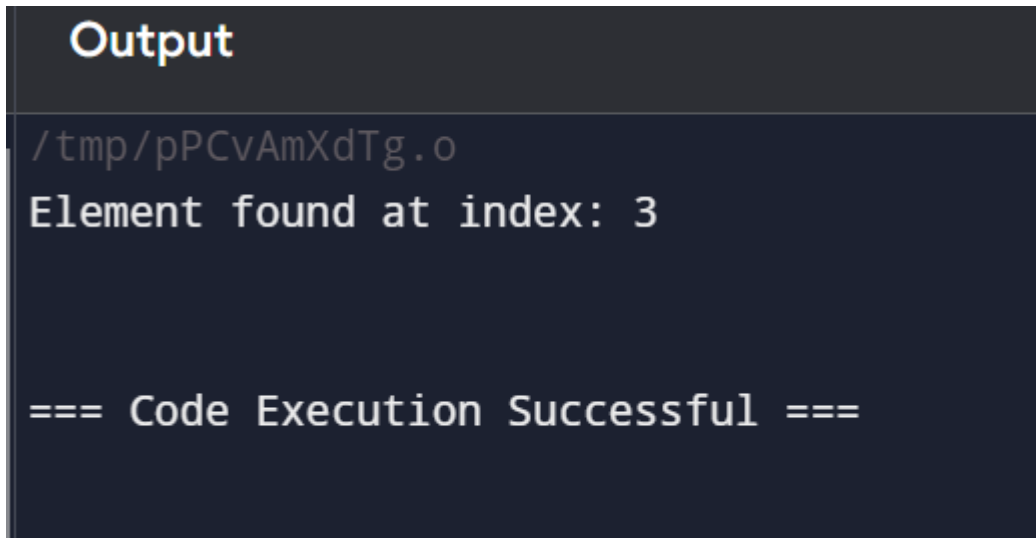
```
int key = 8;

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

if (result != -1) {
    printf("Element found at index: %d\n", result);
} else {
    printf("Element not found\n");
}

return 0;
}
```

#OUTPUT:-

A screenshot of a terminal window with a dark background. The title bar at the top says "Output" in white. The terminal shows the path "/tmp/pPCvAmXdTg.o" in a light gray font. Below it, the text "Element found at index: 3" is displayed in white. At the bottom, the text "=== Code Execution Successful ===" is shown in white.

```
Output
/tmp/pPCvAmXdTg.o
Element found at index: 3

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

comparision between binary search and linear serach

Aspect	Binary Search	Linear Search
Algorithm Type	Divide and conquer	Sequential search
Time Complexity	$O(\log n)$	$O(n)$
Best Use Case	Works best on sorted arrays or lists	Works on unsorted or sorted arrays or lists
Efficiency	Highly efficient for large datasets	Inefficient for large datasets
Data Structure Requirement	Requires sorted data	Does not require sorted data
Number of Comparisons	Reduces comparisons by half after each iteration	Compares each element one by one
Applicability	Only works on random-access data structures (e.g., arrays) 99	Works on any sequential data structure (arrays, linked lists)

~ COMPILER EXECUTION:-

#INPUT:-

```
#include <stdio.h>
```

```
int linearSearch(int arr[], int size, int key) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == key) {
            return i;
        }
    }
    return -1;
}
```

```
int binarySearch(int arr[], int size, int key) {
    int low = 0, high = size - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
```



```

        if (arr[mid] == key) {
            return mid;
        }
        if (arr[mid] < key) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return -1;
}

```

```

int main() {
    int arr[] = {2, 4, 6, 8, 10, 12, 14};
    int size = sizeof(arr) / sizeof(arr[0]);
    int key = 10;

    int linearResult = linearSearch(arr, size, key);
    if (linearResult != -1) {
        printf("Linear Search: Element found at index %d\n", linearResult);
    } else {
        printf("Linear Search: Element not found\n");
    }

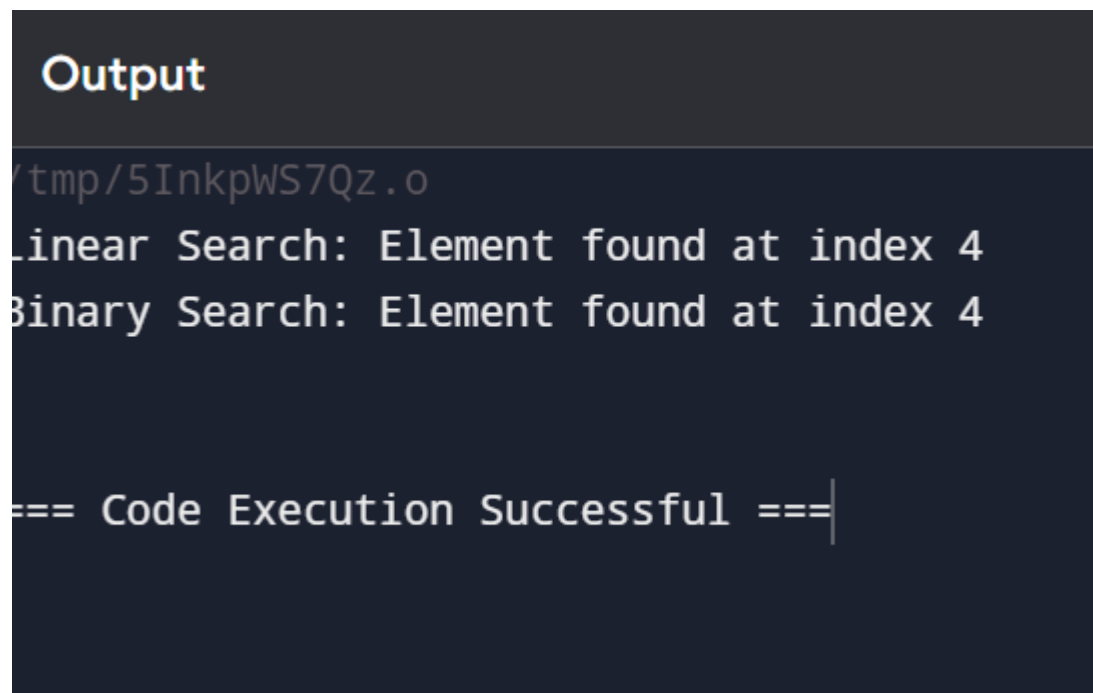
    int binaryResult = binarySearch(arr, size, key);
    if (binaryResult != -1) {
        printf("Binary Search: Element found at index %d\n", binaryResult);
    } else {

```

```
        printf("Binary Search: Element not found\n");
    }

    return 0;
}
```

#OUTPUT:-



```
Output
/tmp/5InkpWS7Qz.o
Linear Search: Element found at index 4
Binary Search: Element found at index 4

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