Experiment No.11
Aim: Application of Binary Search Technique.
Name:Nikhil Kamalaji Shingade
Roll no: 57
Date of Performance:
Date of Submission:

Experiment No. 11: Application of Binary Search Technique

Aim: Application of Binary Search Technique.

Objective:

1) Understand the optimal search algorithm in terms of time.

2) Understand the method searching technique.

Theory:

Binary search is the search technique that works efficiently on sorted lists. Hence, to search an element into some list using the binary search technique. 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 of the middle element is returned. Otherwise, we search into either of the halves depending upon the result produced through the match.

Binary search is much faster than linear search for large datasets, with a time complexity of O(log n). The iterative version of binary search uses O(1) additional space, making it memory efficient.

Algorithm:

Binary_Search(a, lower_bound, upper_bound, val) // 'a' is the given array,

'lower_bound' is the index of the first array element, 'upper_bound' is the index of the last array element, 'val' is the value to search

Step 1: set beg = lower_bound, end = upper_bound, pos = - 1

Step 2: repeat steps 3 and 4 while beg <=end

Step 3: set mid = (beg + end)/2

Step 4: if a[mid] = val set pos

= mid

print pos go to step

6 else if a[mid] >

val set end = mid -

1

else

set beg = mid + 1

[end of if]

[end of loop]

Step 5: if pos = -1

print "value is not present in the array"

[end of if]

Step 6: exit

Code:

```
#include <stdio.h>

// Function to perform binary search
int binarySearch(int array[], int size, int target)
{
   int left = 0;
   int right = size - 1;

   while (left <= right)
 {
      int mid = left + (right - left) / 2;
// To avoid overflow
      // Check if target is present at mid
      if (array[mid] == target)
 {
         return mid;
      }

      // If target is greater, ignore left half
      if (array[mid] < target)
 {
         left = mid + 1;
      }
      // If target is smaller, ignore right half
      else {
         right = mid - 1;
      }
   }
   return -1;
}
```

```
int main()
 {
   int array[] = {2, 3, 4, 10, 40, 50, 70, 80, 90, 100};
   int size = sizeof(array) / sizeof(array[0]);
   int target = 10;
   int result = binarySearch(array, size, target);
    if (result != -1)
 {
      printf("Element found at index %d\n", result);
   } else
{
      printf("Element not found in the array\n");
   }
   return 0;
}
```

Output:

Element found at index 3

Conclusion:

1. What is searching?

Ans: Searching is the process of finding a specific element or value within a data set. Whether you're looking for a word in a document, a number in an array, or a name in a database, searching algorithms help streamline this task

2. Differentiate between binary search and linear search.

Linear Search:

How it Works:

- Sequentially checks each element in the list until the desired element is found or the list ends.

Efficiency:

- Time Complexity: O(n)
- Worst-case scenario: Searches through the entire list.

Usage:

- Works on both sorted and unsorted lists.
- Simple to implement.

Example:

- Searching for a number in an unsorted list like [3, 5, 1, 4, 2].

Binary Search:

How it Works:

- Divides the search interval in half each time, comparing the middle element to the target.
- Continues in the lower or upper half based on the comparison.

Efficiency:

- Time Complexity: O(log n)
- Worst-case scenario: Cuts the search space in half with each step.

Usage:

- Only works on sorted lists.
- More complex but much faster for large datasets.

Example:

- Searching for a number in a sorted list like [1, 2, 3, 4, 5].