

**Symbiosis Institute of Technology**

**Faculty of Engineering**

**CSE- Academic Year 2024-25**

**Data Structures – Lab Batch 2023-27**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Lab Assignment No:- 1** | | | | | |  |
|  |  | | | | | |
| **Name of Student** | Laxmi kumari sah | | | | | |
| **PRN No.** | 23070122125 | | | | | |
| **Batch** | 2023-27 | | | | | |
| **Class** | CS-B1 | | | | | |
| **Academic Year &**  **Semester** | 2024, SEM-III | | | | | |
| **Date of Performance** | 23-07-2024 | | | | | |
|  |  | | | | | |
| **Title of Assignment:** | 1. Implement following searching algorithm: Linear search with multiple occurrences 2. Implement following searching algorithms in menu:    1. Binary search with iteration    2. Binary search with recursion | | | | | |
| **Theory Questions:** | 1. Prepare table for following 5 different searching algorithms for their best case, average case and worst case time complexities. | | | | | | |
|  | Algorithm | Best Case Time Complexity | Average Case  Time  Complexity | Worst Case  Time  Complexity |  | |
| Linear Search | O(1) | O(n) | O(n) |
| Binary Search | O(1) | O(log n) | O(log n) |
| Jump Search | O(√n) | O(√n) |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 2. |  |  |  |  |  |
| Interpolation Search | O(1) | O(log log n) | O(n) |
| Exponential Search | O(1) | O(log n) | O(log n) |
| Apply binary search on the input data set. Show all steps. | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 3. | Compare linear search and binary search | |  | |
| **Aspect** | **Linear Search** | **Binary Search** |  |
| **Algorithm** | Check each element sequentially. | Repeatedly divide the search interval in half. |
| **Time Complexity** | **Best Case**: O(1) | **Best Case**: O(1) |
|  | **Average case:** O(n) | **Average case:** O(log n) |
|  | **Worst Case**: O(n) | **Worst Case**: O(log n) |
| **Space Complexity** | O(1) | O(1) |
| **Array Condition** | Works on both sorted and unsorted arrays. | Requires the array to be sorted. |
| **Simplicity** | Simple and easy to implement. | Slightly more complex but still straightforward. |
| **Use Case** | Effective for small or unsorted datasets. | Efficient for large, sorted datasets. |
| **Advantages** | - No sorting required. | - Much faster for large datasets. |
|  | - Minimal setup. | - Predictable performance on sorted arrays |
| **Disadvantages** | * Inefficient for large datasets. * Slow for large data | - Requires sorted array. - Additional overhead to maintain sorted array. |

|  |  |
| --- | --- |
|  | 4. Why is the complexity of Binary search O(log n)?  The complexity of binary search is O(log n) because it operates by repeatedly dividing the search interval in half. Starting with the entire sorted array, binary search compares the target value with the middle element. If the target is not found, it eliminates half of the array from consideration, focusing on either the left or right half depending on whether the target is less than or greater than the middle element.  This halving process continues until the target is found or the interval is empty. Since each comparison cuts the problem size in half, the number of steps required is proportional to the logarithm of the number of elements, leading to a time complexity of O(log n). |

|  |  |
| --- | --- |
| **Source**  **Code/Algorithm/Flow Chart:** |  |

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
| **Output Screenshots** |  |

|  |  |  |
| --- | --- | --- |
| **Practice questions** | 1. | What is Fibonacci Search explain in detail  Fibonacci Search is a comparison-based search algorithm that uses Fibonacci numbers to divide the array and determine the probe positions. It is particularly useful for searching in a sorted array and is similar to both binary search and jump search, leveraging Fibonacci numbers to optimize the search process. |
|  | 2. | Write algorithm for Fibonacci search  1. Initialize Fibonacci Numbers:   * `fibMMm2 = 0` (m-2)th Fibonacci number. * `fibMMm1 = 1` (m-1)th Fibonacci number. * `fibM = fibMMm1 + fibMMm2` mth Fibonacci number.   2. Find the Smallest Fibonacci Number Greater Than or Equal to `n`:   * While `fibM < n`, do: * `fibMMm2 = fibMMm1` * `fibMMm1 = fibM` * `fibM = fibMMm1 + fibMMm2`   3. Initialize the Offset:  - `offset = -1`  4. Search While `fibM` > 1:   * While `fibM > 1`: * `i = min(offset + fibMMm2, n - 1)` * If `data[i] < x`: * `fibM = fibMMm1` * `fibMMm1 = fibMMm2` * `fibMMm2 = fibM - fibMMm1` * `offset = i` * Else if `data[i] > x`: * `fibM = fibMMm2` * `fibMMm1 = fibMMm1 - fibMMm2` * `fibMMm2 = fibM - fibMMm1` * Else: |

|  |  |  |
| --- | --- | --- |
|  |  | - Return `i` (element found)  5. Check the Last Element:  - If `fibMMm1` and `data[offset + 1] == x`, return `offset + 1`  6. Element Not Found:  - Return -1 |
|  | 3. | Implement Fibonacci Search |

|  |  |
| --- | --- |
|  | 4. o/p screenshot |
| **Conclusion** | Thus we have studied different sorting algorithms and their time complexities. |