SNS COLLEGE OF ENGINEERING

Kurumbapalayam (Po), Coimbatore – 641 107

An Autonomous Institution

Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A' Grade

Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

COURSE NAME : 23ITT101- PROBLEM SOLVING & C PROGRAMMING

I YEAR /I SEMESTER

Unit II – ARRAYS AND STRINGS

Topic : Searching

Searching

- Searching techniques are algorithms used to find a specific element or value in a collection, such as an array, list, or database. There are several different searching techniques, and the choice of which one to use depends on the characteristics of the data being searched (e.g., whether it is sorted or unsorted, the size of the dataset, etc.).

Common searching techniques

- Linear Search

Best for: Unsorted data or when you need to check every element.

Algorithm: Starts from the first element and checks each one sequentially until the target element is found or the end of the collection is reached.

- Binary Search

Best for: Sorted data.

Algorithm: Divides the search interval in half. If the value of the target is less than the element in the middle, the search continues in the lower half. If the target is greater, the search continues in the upper half.

Common searching techniques

- Jump Search

Best for: Sorted data.

Algorithm: Jumps ahead by a fixed block size, and when the target is found, a linear search is performed to find the exact position.

- Exponential Search

Best for: Sorted data, especially large datasets.

Algorithm: First checks an exponentially increasing interval. Once a range is found, binary search is used within that range.

Common searching techniques

- Interpolation Search

Best for: Sorted arrays with uniformly distributed numeric data.

Algorithm: An extension of binary search that uses a heuristic to calculate the "mid" value based on the data values.

- Fibonacci Search

Best for: Sorted data.

Algorithm: Similar to binary search, but uses Fibonacci numbers to divide the array, making it more suitable for certain types of data.

Example: Linear search

```
#include <stdio.h>

int main() {
    int arr[] = {12, 34, 54, 2, 3};  // Sample array
  // Calculate the number of elements
  int n = sizeof(arr) / sizeof(arr[0]);
  int target = 54;  // Element to search for
    int i;
    int found = -1;  // Flag to indicate if the element is found

    // Linear search implementation in main
    for (i = 0; i < n; i++) {
        if (arr[i] == target) {
            found = i;  // Element found, store index
            break;  // Exit loop once the element is found
        }
    }
```

```
    // Output the result
    if (found != -1) {
     // Print the index
   printf("Element found at index %d\n", found);
} else {
      // If not found, print a message
  printf("Element not found\n");
}

    return 0;
}
```

Output:

Element found at index 2

Example: Binary Search

```
#include <stdio.h>
// Function to perform binary search
int binary_search(int arr[], int n, int target) {
    int low = 0;
    int high = n - 1;
    while (low <= high)
    {
        int mid = low + (high - low) / 2; // Find the middle index
        // If target is found at the middle
        if (arr[mid] == target) {
            return mid; // Return the index of the target
        }

        // If target is smaller, search the left half
        if (arr[mid] > target)
       {
            high = mid - 1;
        }
 // If target is larger, search the right half
        else
       {
            low = mid + 1;
        }
    }
    return -1; // Target not found
}
```

```
int main() {
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; // Sorted array
    int n = sizeof(arr) / sizeof(arr[0]); // Size of the array
    int target = 7; // Element to search for

    // Call binary search function
    int result = binary_search(arr, n, target);

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

    return 0;
}
```

Output:

Element 7 found at index 6.

Example : First and last position of an element in an array

```
// Function to find the last occurrence of a given element

int find_last_position(int arr[], int n, int target)
{
    int low = 0, high = n - 1, result = -1;

    while (low <= high)
    {
        int mid = low + (high - low) / 2;

 // If target is found, try to find a later occurrence
        if (arr[mid] == target)
       {
            result = mid;
            low = mid + 1;
// Move right to find the last occurrence
        } else if (arr[mid] < target)
        {
            low = mid + 1;
        } else
        {
            high = mid - 1;
        }
    }
    return result;
}
```

```
int main()
{
    int arr[] = {1, 2, 2, 2, 3, 3, 4, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 2;

    // Find the first and last position of the target
    int first_pos = find_first_position(arr, n, target);
    int last_pos = find_last_position(arr, n, target);

    if (first_pos != -1)
   {
        printf("First position of %d: %d\n", target, first_pos);
        printf("Last position of %d: %d\n", target, last_pos);
    } else
   {
        printf("Element %d not found in the array.\n", target);
    }

    return 0;
}
```

First position of 2: 1
Last position of 2: 3

Example : Remove All Occurrences of an Element in an array

```c
#include <stdio.h>

int remove_occurrences(int arr[], int n, int target)
{
    int index = 0;
 // Index for the new array without the target element

    for (int i = 0; i < n; i++)
   {
        if (arr[i] != target)
       {
            arr[index++] = arr[i];
 // If it's not the target, move it to the new position
        }
    }

    return index;
 // New size of the array after removal
}
```

```c
int main()
{
    int arr[] = {1, 2, 2, 3, 4, 2, 5, 6, 2};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 2;

    // Call the function to remove the target element
    int new_size = remove_occurrences(arr, n, target);

    // Print the array after removing the target element
    printf("Array after removing %d: ", target);
    for (int i = 0; i < new_size; i++)
   {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

First position of 2: 1
Last position of 2: 3

Choosing the Right Search Algorithm

- Linear Search: Best for small or unsorted datasets where the overhead of sorting is not justifiable.
- Binary Search: Best for sorted datasets where fast searching is required.
- Jump Search: An improvement over linear search, works well for sorted data where you can divide the search into blocks.

Choosing the Right Search Algorithm

- Exponential Search: Effective for large sorted arrays when you don't know the bounds of the dataset.
- Interpolation Search: Best for uniformly distributed data where values are spread evenly.
- Fibonacci Search: Similar to binary search but uses Fibonacci numbers, and is efficient for large sorted datasets.


Thank
you