Name: SATYAM PANDEY

FE B1 AIDS

2401074

Experiment No. 1

AIM:

With the help of methods for inserting the elements in an sorted array, method for deleting an element from array, method for sorting an array, method for merging two arrays, method for searching in an array. Write a menu driven program (Switch Case) to repeatedly execute the one or more of this methods .

Theory:

1. Arrays

An array is a collection of elements, all of the same data type, stored in contiguous memory locations. Arrays are widely used because of their simplicity and efficiency in accessing elements by index.

- **Characteristics**:
    - Fixed size (in this case, arr1[100] and arr2[100]).
    - Random access to elements using indices.
    - Elements are stored in sorted order for specific operations like binary search or efficient merging.

---

2. Basic Array Operations

Insertion:

- The program inserts elements into a sorted array. The insertion is done in a way that maintains the sorted order.
- **Algorithm**:
    - Compare the new element with existing elements.
    - Shift larger elements one position to the right.
    - Insert the new element at its correct position.
- **Complexity**: O(n) in the worst case, where n is the number of elements in the array.

Deletion:

- Removes an element from the array by finding it and shifting subsequent elements to the left.
- **Algorithm**:

    - Find the element to delete.
    - Shift all elements after the target element one position to the left.
    - Reduce the size of the array.
- **Complexity**: O(n) for searching and shifting elements.

Sorting:

- The program uses **Bubble Sort**, a simple comparison-based sorting algorithm.
- **Algorithm**:
    - Repeatedly compare adjacent elements and swap them if they are out of order.
    - Continue until no swaps are needed.
- **Complexity**: $O(n^2)$ in the worst case, O(n) for already sorted arrays.

Searching:

- A linear search is implemented to find the index of an element in the array.
- **Algorithm**:
    - Iterate through the array and compare each element with the target.
    - Return the index if found; otherwise, return -1.
- **Complexity**: O(n) in the worst case.

Merging:

- Combines two sorted arrays into a single sorted array using a two-pointer approach.
- **Algorithm**:
    - Compare the elements of both arrays.
    - Append the smaller element to the result array.
    - Continue until all elements from both arrays are added.
- **Complexity**: O(n + m), where n and m are the sizes of the two arrays.

Display:

- Prints the array elements in order.
- Handles the case of an empty array by notifying the user.

---

3. Menu-Driven Approach

The program uses a **menu-driven approach**, allowing users to choose operations interactively. This is a common design in simple programs to facilitate user input and output.

---

4. Limitations

1. **Static Arrays**:
    - The fixed-size arrays limit scalability.
    - Dynamic memory allocation (malloc, calloc) or dynamic arrays (like std::vector in C++) are recommended for larger datasets.
2. **Sorting Algorithm**:
    - Bubble Sort is not efficient for large datasets. More efficient algorithms like **Quick Sort** or **Merge Sort** are preferred for real-world applications.
3. **Search Algorithm**:
    - The program uses **Linear Search**. When the array is sorted, **Binary Search** would be more efficient with a time complexity of O(log n).

Applications

1. **Data Management**:
    - This program can be used in basic inventory or database systems for operations like insertion, deletion, and search.
2. **Learning Tool**:
    - It is an excellent program for beginners to learn array manipulation, sorting, merging, and searching.
3. **Foundation for Complex Programs**:
    - This program provides the building blocks for more advanced data structures like linked lists, trees, and hash tables.

Program:

```
#include <stdio.h>


void insertElement(int arr[], int *size, int element);

void deleteElement(int arr[], int *size, int element);

void sortArray(int arr[], int size);

void mergeArrays(int arr1[], int size1, int arr2[], int size2, int result[], int *resultSize);

int searchElement(int arr[], int size, int element);

void displayArray(int arr[], int size);
```

```
int main() {
    int arr1[100], arr2[100], result[200];
    int size1 = 0, size2 = 0, resultSize = 0;
    int choice, element, index;

    while (1) {
        printf("\nMenu:");
        printf("\n1. Insert Element into Sorted Array");
        printf("\n2. Delete Element from Array");
        printf("\n3. Sort an Array");
        printf("\n4. Merge Two Arrays");
        printf("\n5. Search for an Element");
        printf("\n6. Display Array");
        printf("\n7. Exit");
        printf("\nEnter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the element to insert: ");
                scanf("%d", &element);
                insertElement(arr1, &size1, element);
                printf("Element inserted. Array: ");
                displayArray(arr1, size1);
                break;

            case 2:
                printf("Enter the element to delete: ");
                scanf("%d", &element);
                deleteElement(arr1, &size1, element);
```

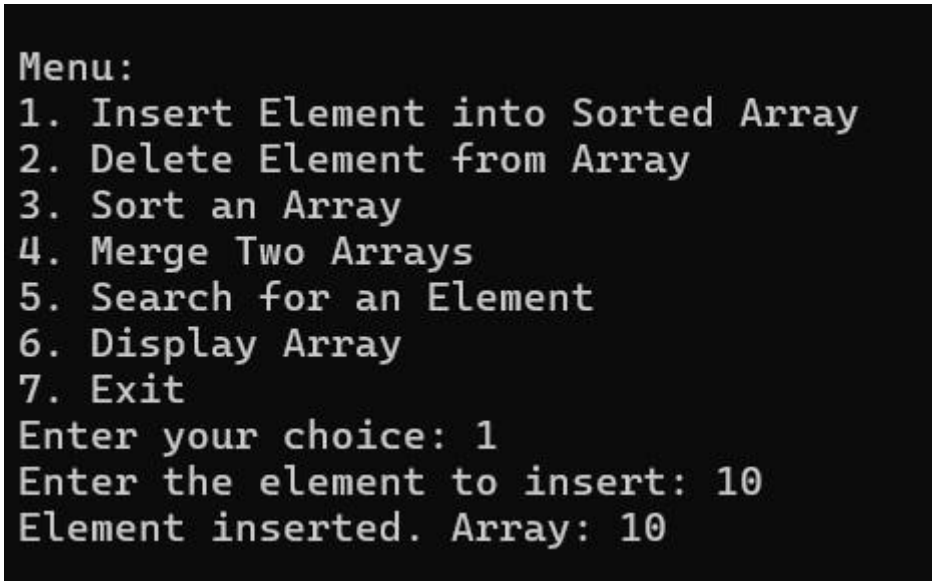
```
    printf("Element deleted (if existed). Array: ");
    displayArray(arr1, size1);
    break;

case 3:
    printf("Sorting the array...\n");
    sortArray(arr1, size1);
    printf("Array sorted: ");
    displayArray(arr1, size1);
    break;

case 4:
    printf("Enter the size of the second array: ");
    scanf("%d", &size2);
    printf("Enter elements of the second array:\n");
    for (int i = 0; i < size2; i++) {
        scanf("%d", &arr2[i]);
    }
    mergeArrays(arr1, size1, arr2, size2, result, &resultSize);
    printf("Merged Array: ");
    displayArray(result, resultSize);
    break;

case 5:
    printf("Enter the element to search for: ");
    scanf("%d", &element);
    index = searchElement(arr1, size1, element);
    if (index != -1)
        printf("Element found at index %d.\n", index);
    else
        printf("Element not found.\n");
```

```
            break;

        case 6:
            printf("Current array: ");
            displayArray(arr1, size1);
            break;

        case 7:
            printf("Exiting program.\n");
            return 0;

        default:
            printf("Invalid choice! Try again.\n");
      }
   }
   return 0;
}

void insertElement(int arr[], int *size, int element) {
   int i = *size - 1;
   while (i >= 0 && arr[i] > element) {
      arr[i + 1] = arr[i];
      i--;
   }
   arr[i + 1] = element;
   (*size)++;
}

void deleteElement(int arr[], int *size, int element) {
   int i, pos = -1;
   for (i = 0; i < *size; i++) {
```

```
        if (arr[i] == element) {
            pos = i;
            break;
        }
    }
    if (pos != -1) {
        for (i = pos; i < *size - 1; i++) {
            arr[i] = arr[i + 1];
        }
        (*size)--;
    } else {
        printf("Element not found.\n");
    }
}

void sortArray(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

void mergeArrays(int arr1[], int size1, int arr2[], int size2, int result[], int *resultSize) {
    int i = 0, j = 0, k = 0;
    while (i < size1 && j < size2) {
        if (arr1[i] < arr2[j])
```

```
            result[k++] = arr1[i++];
        else
            result[k++] = arr2[j++];
    }
    while (i < size1)
        result[k++] = arr1[i++];
    while (j < size2)
        result[k++] = arr2[j++];
    *resultSize = k;
}

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

void displayArray(int arr[], int size) {
    if (size == 0) {
        printf("Array is empty.\n");
        return;
    }
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```

Output:



Conclusion:

1. **Functionality**:
    - The program provides a comprehensive set of array operations: insertion, deletion, sorting, merging, searching, and display. These are common tasks in basic data structure management.
    - The array is kept sorted automatically when inserting elements, simplifying operations like searching and merging.
2. **Efficiency**:
    - Sorting uses the bubble sort algorithm, which is not the most efficient for large datasets but suffices for small-scale educational purposes.
    - Merging is implemented using a two-pointer approach, which is efficient and works seamlessly with sorted arrays.
3. **User-Friendly Design**:
    - The menu-driven interface is intuitive and easy to follow.
    - Error handling ensures invalid inputs are managed gracefully (e.g., searching or deleting nonexistent elements).
4. **Scalability**:
    - The use of a static array with a fixed size (100) limits scalability. In real-world applications, dynamic memory allocation or dynamic arrays (e.g., std::vector in C++) should be considered.
    - Str**u**ctured approach to writing modular, reusable code with functions.