DSC IA1

M1Q1) What is a data structure? Explain the different types of data structure and data operation with an example.

A **data structure** is a way of organizing, storing, and managing data in a computer so that it can be used efficiently. It provides a systematic format for data organization, making it easier to access, manipulate, and store information.

Types of Data Structures

Data structures are broadly categorized into **Primitive** and **Non-Primitive** data structures.

1. **Primitive Data Structures**

   These are the most basic data types available in programming languages. They operate directly upon the data and are supported by the system.

   - **Examples:**
     - Integer (`int`)
     - Float (`float`)
     - Character (`char`)
     - Boolean (`bool`)

2. **Non-Primitive Data Structures**

   These are more complex data structures built using primitive data types. They are further classified into two types:

   a) Linear Data Structures

   In linear data structures, elements are arranged in a sequential manner, where each element has a successor and predecessor (except the first and last elements).

   - **Examples:**
     - Arrays
     - Linked Lists

- Stacks
- Queues

b) Non-Linear Data Structures

In non-linear data structures, elements are connected in a hierarchical or interlinked manner.

- **Examples:**
  - Trees
  - Graphs

Data Operations

1. **Insertion** - Adding an element to a data structure.
2. **Deletion** - Removing an element from a data structure.
3. **Traversal** - Accessing each element of the data structure sequentially.
4. **Searching** - Finding an element within the data structure.
5. **Sorting** - Arranging data in ascending or descending order.

M1Q2) Explain how two-dimensional arrays are represented in memory with an example.

A **two-dimensional (2D) array** is essentially an array of arrays, where data is stored in a matrix format with rows and columns. In memory, a 2D array is stored in a contiguous block of memory, and it can be represented in two main ways:

1. **Row-Major Order**
   - In **row-major order**, the elements of the array are stored row by row.
   - All elements of the first row are stored first, followed by the elements of the second row, and so on.

   Formula to calculate the address of an element:

   $$\text{Address}(A[i][j]) = \text{Base Address} + ((i \times \text{Number of Columns}) + j) \times \text{Size of Data Type}$$

   - i = Row index

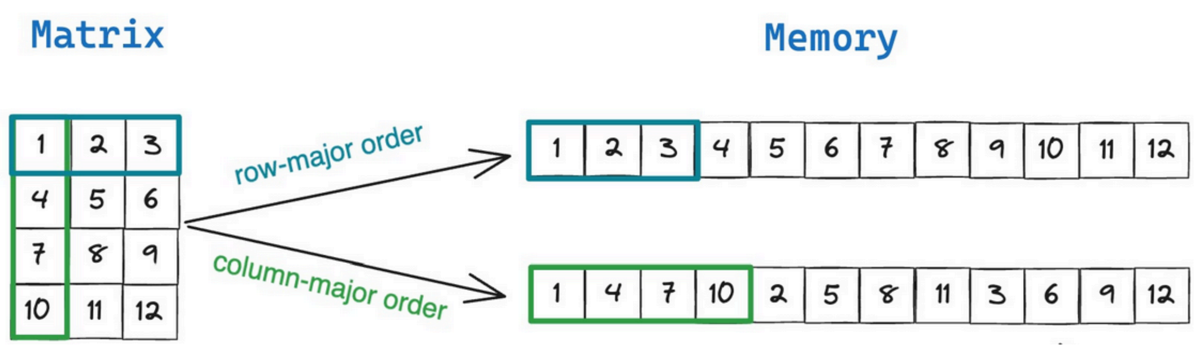
- j = Column index
- **Base Address** = Address of the first element of the array
- **Number of Columns** = Total columns in the array
- **Size of Data Type** = Size of each element (e.g., `int` usually takes 4 bytes)

2. **Column-Major Order**

- In **column-major order**, the elements are stored column by column.
- All elements of the first column are stored first, followed by the elements of the second column, and so on.
- This is commonly used in languages like **Fortran** or **MATLAB**.

Formula to calculate the address of an element:

$$\text{Address}(A[i][j]) = \text{Base Address} + ((j \times \text{Number of Rows}) + i) \times \text{Size of Data Type}$$



M1Q3) What is a structure in C. Give its syntax? Differentiate it from an array. Give example.

A **structure** in C is a user-defined data type that allows you to group related variables of different data types under a single name. It is particularly useful when you need to store information about an entity that has multiple attributes.

```
struct StructureName {
    dataType member1;
    dataType member2;
```

```
    dataType member3;
    ...
};
```

- `struct` → Keyword used to declare a structure.
- `StructureName` → Name of the structure.
- `member1, member2...` → Variables of different data types inside the structure.

```
//example

struct Book {
    char title[50];
    char author[50];
    float price;
};

int main() {
    struct Book b1 = {"C Programming", "E Balagurusamy", 499.50};

    printf("Book Title: %s\n", b1.title);
    printf("Author: %s\n", b1.author);
    printf("Price: %.2f\n", b1.price);

    return 0;
}

//output
Book Title: C Programming
Author: E Balagurusamy
Price: 499.50
```

Difference Between Structure and Array

Aspect	Structure	Array
Data Type	Can store variables of different data types	Can store only variables of the same data type

Aspect	Structure	Array
Storage	Uses memory according to the size of individual members	Uses contiguous memory for elements
Access	Accessed using the dot `.` operator	Accessed using index values

M1Q4) What is a pointer? What are the uses of pointers? How do you declare and initialize the pointers?

A **pointer** is a variable that stores the memory address of another variable. Instead of storing a direct value, it stores the location of the value in memory.

- It is called a pointer because it "points" to the address of a variable.

Uses of Pointers

Pointers are widely used in C for various purposes:

1. **Accessing Memory Directly:**
   - Pointers can access and manipulate memory directly using memory addresses.
2. **Dynamic Memory Allocation:**
   - Pointers are used to allocate memory dynamically using functions like `malloc()` and `calloc()`.
3. **Function Arguments (Call by Reference):**
   - Pointers allow functions to modify the actual values of variables by passing memory addresses instead of copies.
4. **Data Structures:**
   - Pointers are essential for implementing data structures like linked lists, trees, and graphs.
5. **File Handling:**
   - File pointers are used to read, write, and manipulate files.

```
//declaration
dataType *pointerName;
```

```
//initialization
pointerName = &variableName;
```

M1Q5) How do you access the value Pointed by the Pointers?

To access the value stored at the memory address pointed to by a pointer, you use the **dereference operator (`*`)**.

- **Dereferencing a Pointer** means accessing the value stored at the memory location to which the pointer points.
- The operator * is also called the **indirection operator**.

```
*pointerName  // gives value stored at address
```

M1Q6) Explain with examples how structure members can be accessed with pointers.

In **C programming**, structure members can be accessed using pointers. To achieve this, the following concepts are used:

1. **Dot Operator (`.`)** → Used when accessing structure members using a structure variable.
2. **Arrow Operator (`>`)** → Used when accessing structure members using a pointer to a structure.

```
struct Student {
    char name[50];
    int rollNumber;
    float marks;
};

void main() {
    struct Student s1 = {"Alice", 102, 92.3};
    struct Student *ptr = &s1;  // Pointer to structure

    // Access using arrow operator
```

```
    printf("Name: %s\n", ptr→name);
    printf("Roll Number: %d\n", ptr→rollNumber);
    printf("Marks: %.2f\n", ptr→marks);
}
```

Explanation

- `struct Student *ptr = &s1;` → Pointer `ptr` stores the address of structure `s1`.
- `ptr→name` → Accesses the `name` member using pointer.
- `ptr→rollNumber` → Accesses the `rollNumber` member.
- `ptr→marks` → Accesses the `marks` member.

> Note: ptr→member is equivalent to (*ptr).member
>
> The `→` operator is a shorthand for dereferencing the pointer and accessing the member using the dot operator.

M1Q7) What are the various memory allocation techniques? Explain how dynamic memory is allocated for arrays with C program.

Memory allocation in C can be classified into two main categories:

1. **Static Memory Allocation**
    - Memory is allocated **at compile time**.
    - The size of the memory is **fixed** and cannot be changed during program execution.
    - Example: Arrays declared using standard methods (`int arr[10];`)
    - Memory is allocated in the **Stack**.

    ```
    int arr[5];  // Static memory allocation for an array
    ```

2. **Dynamic Memory Allocation**
    a. Using `malloc()`

- `malloc()` stands for **Memory Allocation**.
- It allocates memory of the specified size and returns a pointer of type `void*`, which needs to be typecast.

  ```
  ptr = (dataType*) malloc(size_in_bytes);
  ```

b. Using `calloc()`

- `calloc()` stands for **Contiguous Allocation**.
- It allocates memory for an array of elements and **initializes** all the elements to **0**.

```
ptr = (dataType*) calloc(number_of_elements, size_of_each_element);
```

c. Using `realloc()`

- `realloc()` is used to **resize** the previously allocated memory using `malloc()` or `calloc()`.

  ```
  ptr = (dataType*) realloc(ptr, new_size_in_bytes);
  ```

M1Q8) Define a structure. Write a program to calculate the subject wise and student wise totals and store them as a part of a structure.

```
#include <stdio.h>

// Define structure to store student data
struct Student {
    char name[50];
    int marks[5]; // Assuming 5 subjects
    int total;
};

int main() {
    int n, i, j;
```

```
    printf("Enter the number of students: ");
    scanf("%d", &n);

    struct Student students[n];
    int subjectTotal[5] = {0}; // Initialize subject totals to 0

    // Input student details
    for (i = 0; i < n; i++) {
        students[i].total = 0;
        printf("\nEnter name of student %d: ", i + 1);
        scanf("%s", students[i].name);

        printf("Enter marks of %s for 5 subjects: ", students[i].name);
        for (j = 0; j < 5; j++) {
            scanf("%d", &students[i].marks[j]);
            students[i].total += students[i].marks[j]; // Calculate student-wise to
tal
            subjectTotal[j] += students[i].marks[j];  // Calculate subject-wise tot
al
        }
    }

    // Display Student Wise Totals
    printf("\nStudent Wise Totals:");
    for (i = 0; i < n; i++) {
        printf("\n%s: Total Marks = %d", students[i].name, students[i].total);
    }

    // Display Subject Wise Totals
    printf("\n\nSubject Wise Totals:");
    for (j = 0; j < 5; j++) {
        printf("\nSubject %d Total: %d", j + 1, subjectTotal[j]);
    }

    return 0;
}
```

M1Q9) Differentiate between malloc() and calloc().

Both `malloc()` and `calloc()` are functions in C used for **dynamic memory allocation**. However, they have some key differences in their behavior.

Comparison Table

Feature	`malloc()`	`calloc()`
Full Form	**Memory Allocation**	**Contiguous Allocation**
Initialization	Does **not initialize** the memory (contains garbage values).	Initializes the allocated memory to **0**.
No. of Arguments	Takes **1 argument** (Total memory size in bytes).	Takes **2 arguments** (Number of blocks and size of each block).
Syntax	`ptr = (int*) malloc(size_in_bytes);`	`ptr = (int*) calloc(num_elements, size_of_element);`
Speed	Faster because it doesn't initialize memory.	Slightly slower due to memory initialization.
Use Case	Suitable when memory initialization is **not required**.	Preferred when memory needs to be initialized to **0** and contigous memory req.
Return Value	Returns a `void*` pointer to allocated memory.	Returns a `void*` pointer to allocated memory.
Example	`ptr = (int*) malloc(5 * sizeof(int));`	`ptr = (int*) calloc(5, sizeof(int));`

When to Use `malloc()` vs `calloc()`

- **Use `malloc()`** if memory initialization is **not required** and performance is a priority.
- **Use `calloc()`** if you need the memory to be initialized to **zero** (e.g., working with sensitive data or ensuring clean memory).

M1Q10) Write a program to find the largest of n numbers using pointer.

```
#include <stdio.h>

int main() {
    int n, i, arr[100];
```

```
    int *ptr;

    printf("Enter the number of elements (up to 100): ");
    scanf("%d", &n);

    printf("Enter %d numbers: ", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    ptr = arr; // Pointer points to the first element
    int largest = *ptr; // Initialize with first element

    for (i = 1; i < n; i++) {
        if (*(ptr + i) > largest) {
            largest = *(ptr + i); // Access using pointer arithmetic
        }
    }

    printf("The largest number is: %d\n", largest);

    return 0;
}
```

M1Q11) Differentiate between static and dynamic memory allocation.

Aspect	Static Memory Allocation	Dynamic Memory Allocation
Definition	Memory is allocated **at compile time**.	Memory is allocated **at runtime**.
Flexibility	Fixed size; cannot be resized during execution.	Flexible size; can be resized using `realloc()` if required.
Memory Management	Managed by the compiler.	Managed manually using `malloc()`, `calloc()`, `realloc()`, and `free()`.
Speed	Faster because allocation is done at compile time.	Slower due to runtime allocation and memory management overhead.

Aspect	Static Memory Allocation	Dynamic Memory Allocation
Memory Utilization	May lead to memory wastage if extra memory is allocated.	Efficient as memory is allocated as per the need.
Example	`int arr[10];`	`int *arr = (int*)malloc(10 * sizeof(int));`
Reallocation	Not possible once declared.	Possible using `realloc()` .
Lifetime	Memory exists throughout the program execution.	Memory exists until explicitly deallocated using `free()` .
Storage Location	Stored in the **Stack**.	Stored in the **Heap**.
Use Case	Suitable for programs where memory requirements are fixed.	Suitable for programs where memory needs may change during execution.

M2Q1) Write C program to search for an item using binary search.

```
#include <stdio.h>

// Function to perform binary search
int binarySearch(int arr[], int size, int target) {
    int low = 0, high = size - 1;

    while (low <= high) {
        int mid = low + (high - low) / 2; // Calculate mid to avoid overflow

        if (arr[mid] == target) {
            return mid; // Target found, return index
        }
        else if (arr[mid] < target) {
            low = mid + 1; // Search in the right half
        }
        else {
            high = mid - 1; // Search in the left half
        }
    }
    return -1; // Target not found
}
```

```
// Function to print result
void printResult(int index, int target) {
    if (index != -1) {
        printf("Element %d found at index %d.\n", target, index);
    } else {
        printf("Element %d not found in the array.\n", target);
    }
}

int main() {
    // Example sorted array
    int arr[] = {10, 20, 30, 40, 50, 60, 70, 80, 90};
    int size = sizeof(arr) / sizeof(arr[0]);

    // Search for different elements
    printResult(binarySearch(arr, size, 30), 30);
    printResult(binarySearch(arr, size, 100), 100); // Not present

    return 0;
}
```

M2Q2) Write C programs to search for an item using linear search.

```
#include <stdio.h>

// Function to perform linear search
int linearSearch(int arr[], int size, int target) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) {
            return i; // Return the index if the element is found
        }
    }
    return -1; // Return -1 if the element is not found
}
```

```
int main() {
    // Example array
    int arr[] = {12, 45, 67, 23, 89, 54, 38};
    int size = sizeof(arr) / sizeof(arr[0]);

    // Items to search for
    int target = 89;

    // Perform linear search and display results
    int result = linearSearch(arr, size, 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;
}
```

M2Q3) Differentiate between linear search and Binary search.

Aspect	Linear Search	Binary Search
Definition	Searches for an element sequentially, one by one.	Divides the array into two halves and searches in the appropriate half.
Array Type	Works on both **sorted** and **unsorted** arrays.	Works only on **sorted** arrays.
Time Complexity	**O(n)** in the worst case.	**O(log n)** in the worst case.
Best Case Complexity	**O(1)** if the element is the first element.	**O(1)** if the middle element is the target.
Efficiency	Less efficient for large datasets.	Highly efficient for large datasets.
Memory Usage	No extra memory required.	No extra memory required.
Working Principle	Compares elements one by one until the match is found.	Compares the middle element and decides the half to continue the search.

Aspect	Linear Search	Binary Search
Use Case	Suitable for small arrays or unsorted data.	Suitable for large sorted datasets.
Example Use	Searching in small datasets like student records.	Searching in phone books or large databases.

M2Q4) Write a C function for insertion sort. Sort the following list using insertion sort: 50, 30, 10, 70, 40, 20, 60.

- **Insertion Sort** is a simple sorting algorithm that builds the sorted array one element at a time.
- It compares the current element with the previous elements and inserts it into its correct position.
- It works similarly to how we sort playing cards in our hands.

```
#include <stdio.h>

// Function for Insertion Sort
void insertionSort(int arr[], int n) {
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        // Move elements greater than key to one position ahead
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key; // Insert key at the correct position
    }
}

// Function to print the array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
```

```
        printf("%d ", arr[i]);
    }
    printf("\n");
}

// Main function
int main() {
    int arr[] = {50, 30, 10, 70, 40, 20, 60};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original Array: ");
    printArray(arr, n);

    // Sorting the array using insertion sort
    insertionSort(arr, n);

    printf("Sorted Array using Insertion Sort: ");
    printArray(arr, n);

    return 0;
}
```

Step-by-Step Execution

Let's see how the array `{50, 30, 10, 70, 40, 20, 60}` is sorted using Insertion Sort.

Pass	Key	Compared Values	Array State
1	30	50 > 30 → Swap	{30, 50, 10, 70, 40, 20, 60}
2	10	50 > 10, 30 > 10 → Swap	{10, 30, 50, 70, 40, 20, 60}
3	70	70 > 50 → No Swap	{10, 30, 50, 70, 40, 20, 60}
4	40	70 > 40, 50 > 40 → Swap	{10, 30, 40, 50, 70, 20, 60}
5	20	70 > 20, 50 > 20, 40 > 20, 30 > 20 → Swap	{10, 20, 30, 40, 50, 70, 60}

Pass	Key	Compared Values	Array State
6	60	70 > 60 → Swap	{10, 20, 30, 40, 50, 60, 70}

M2Q5) Write a C function for Selection sort. Sort the following list using insertion sort: 50, 30, 10, 70, 40, 20, 60.

- **Selection Sort** is a simple sorting algorithm that works by repeatedly selecting the **smallest (or largest)** element from the unsorted part of the array and swapping it with the first unsorted element.
- It is called **Selection Sort** because it repeatedly **selects** the smallest element.

```
#include <stdio.h>

// Function for Selection Sort
void selectionSort(int arr[], int n) {
    int i, j, minIndex, temp;

    for (i = 0; i < n - 1; i++) {
        minIndex = i; // Assume the current index has the minimum value

        // Find the minimum element in the unsorted part
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }

        // Swap the minimum element with the first unsorted element
        temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }
}
```

```
// Function to print the array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

// Main function
int main() {
    int arr[] = {50, 30, 10, 70, 40, 20, 60};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original Array: ");
    printArray(arr, n);

    // Sorting the array using Selection Sort
    selectionSort(arr, n);

    printf("Sorted Array using Selection Sort: ");
    printArray(arr, n);

    return 0;
}
```

Step-by-Step Execution Using Selection Sort

Let's see how the array `{50, 30, 10, 70, 40, 20, 60}` is sorted using Selection Sort.

Pass	Array State	Minimum Element	Swap
1	**10**, 30, 50, 70, 40, 20, 60	10	10 ↔ 50
2	10, **20**, 50, 70, 40, 30, 60	20	20 ↔ 30
3	10, 20, **30**, 70, 40, 50, 60	30	No Swap
4	10, 20, 30, **40**, 70, 50, 60	40	40 ↔ 70
5	10, 20, 30, 40, **50**, 70, 60	50	No Swap
6	10, 20, 30, 40, 50, **60**, 70	60	No Swap

M2Q6) Write a C function for Bubble sort. Sort the following list using insertion sort: 50, 30, 10, 70, 40, 20, 60.

```
#include <stdio.h>

// Function to perform Bubble Sort
void bubbleSort(int arr[], int n) {
    int i, j, temp;
    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap adjacent elements if they are in the wrong order
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

// Function to print the array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

// Main function
int main() {
    int arr[] = {50, 30, 10, 70, 40, 20, 60};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original Array: ");
    printArray(arr, n);

    bubbleSort(arr, n);
```

```
    printf("Sorted Array using Bubble Sort: ");
    printArray(arr, n);

    return 0;
}
```

Step-by-Step Execution Table for Insertion Sort

Pass	Key	Comparison	Shifting	Array After Pass
1	30	30 < 50	Shift **50** to the right	**30**, 50, 10, 70, 40, 20, 60
2	10	10 < 50 → Shift 50	10 < 30 → Shift 30	**10**, 30, 50, 70, 40, 20, 60
3	70	70 > 50 → No Change	No Shifting	10, 30, 50, **70**, 40, 20, 60
4	40	40 < 70 → Shift 70	40 < 50 → Shift 50	10, 30, **40**, 50, 70, 20, 60
5	20	20 < 70 → Shift 70	20 < 50 → Shift 50	20 < 40 → Shift 40
6	60	60 < 70 → Shift 70	60 > 50 → No More Shifting	10, 20, 30, 40, 50, **60**, 70

*********************************** EOF ***********************************