Parul® University NAAC A++ ACCREDITED UNIVERSITY

Centre for Distance and Online Education

Online MCA Program

Semester- I

(DATA STRUCTRE) ASSIGNMENT

Prepared by

Name of the Student :- Dev jitendrabhai kalwani

Enrollment no :- 2501j00600234

1. What is data Structure? Explain the Classification of Data Structure in detail.

A **data structure** is a specialized way of organizing, storing, and managing data so that it can be used efficiently. In simple terms, it provides a systematic format to arrange data in a computer's memory. The main purpose of using data structures is to perform operations like searching, inserting, deleting, and updating data quickly and efficiently.

1. Primitive Data Structures

Primitive data structures are the **basic building blocks** provided by programming languages. These are simple and cannot be broken down further. They are used to represent simple data values.

Common primitive data types include:

- **int (integer)** – stores whole numbers
- **float/double** – stores decimal numbers
- **char (character)** – stores a single character
- **boolean** – stores true/false values

These types provide the foundation on which non-primitive structures are built.

---

2. Non-Primitive Data Structures

Non-primitive data structures are more complex and are built using primitive data types. They are used to store large and structured data. They are further divided into **linear** and **non-linear** data structures.

A. Linear Data Structures

In linear structures, elements are arranged **sequentially**, one after another.

Examples:

- **Array** – collection of elements of the same type stored at contiguous memory locations
- **Linked List** – collection of nodes connected by pointers
- **Stack** – follows LIFO (Last In First Out) principle
- **Queue** – follows FIFO (First In First Out) principle

These structures are easy to implement and efficient for sequential data processing.

B. Non-Linear Data Structures

Non-linear structures do not store data sequentially. Instead, elements form a **hierarchical** or **network** relationship.

Examples:

- **Tree** – hierarchical structure with root and branches (e.g., binary tree)
- **Graph** – represents interconnected nodes (used in networks, maps, etc.)

These structures are powerful for representing complex relationships.

2. **Explain the Types of an Array. Write down syntax for declaration, initialization and how to access elements from Array in C.**

   In C programming, an array is a collection of elements of the same data type stored in contiguous memory locations. Arrays make it easier to manage large amounts of data using a single variable name with multiple indices. Based on the structure and dimensions, arrays in C are mainly of the following types:

---

1. One-Dimensional Array (1D Array)
A one-dimensional array is the simplest form of an array. It stores data in a single row. It is usually used to store lists like marks, salaries, or age of students.
Example:
int marks[5];
Here, 5 integer elements are stored sequentially.

---

2. Two-Dimensional Array (2D Array)
A two-dimensional array represents data in rows and columns, similar to a matrix or table. It is commonly used for storing tabular data like matrices, game boards, or pixel grids.
Example:
int matrix[3][3];
This creates a 3×3 matrix.

---

3. Multi-Dimensional Array
Arrays with more than two dimensions are called multi-dimensional arrays. They represent data in more complex structures such as 3D matrices.
Example:
int cube[3][3][3];
These are useful for scientific computations, 3D graphics, etc.

1D Initialization:

```
int arr[5] = {10, 20, 30, 40, 50};
```

2D Initialization:

```
int matrix[2][2] = {{1, 2}, {3, 4}};
```

Partial Initialization:

```
int arr[5] = {1, 2};
```

1D Access:

```
arr[2];  // Accesses 3rd element
```

2D Access:

```
matrix[1][0];  // Accesses element at row 1, column 0
```

3. What is stack? Explain the working of PUSH and POP operations in the stack with example

A stack is a linear data structure that follows the LIFO (Last In, First Out) principle. This means the last element inserted into the stack is the first element removed.

You can imagine a stack like a pile of plates:

- You add a new plate on the top,
- and when you need one, you remove the plate from the top.

A stack has one pointer called TOP, which always points to the most recently added (top-most) element.
Stacks are widely used in expression evaluation, backtracking, memory management, undo operations, and function call handling.

PUSH Operation (Insert an element)

PUSH means adding an element to the top of the stack.
Steps in PUSH:

1. Check if the stack is full (Overflow condition).
2. If not full, increment TOP by 1.
3. Insert the new element at the position indicated by TOP.

Example:
Suppose the stack initially is:
Stack: [10, 20, 30]
TOP = 2

PUSH(40):

- TOP becomes 3
- Stack becomes: [10, 20, 30, 40]

POP Operation (Remove an element)

POP means removing the top-most element from the stack.
Steps in POP:

1. Check if the stack is empty (Underflow condition).
2. Retrieve the element at TOP.
3. Decrement TOP by 1.

Example:
If the stack is:
Stack: [10, 20, 30, 40]
TOP = 3

POP():

- Removes 40
- TOP becomes 2
- New stack: [10, 20, 30]

4. Give the difference between Simple Queue V/S Circular Queue.

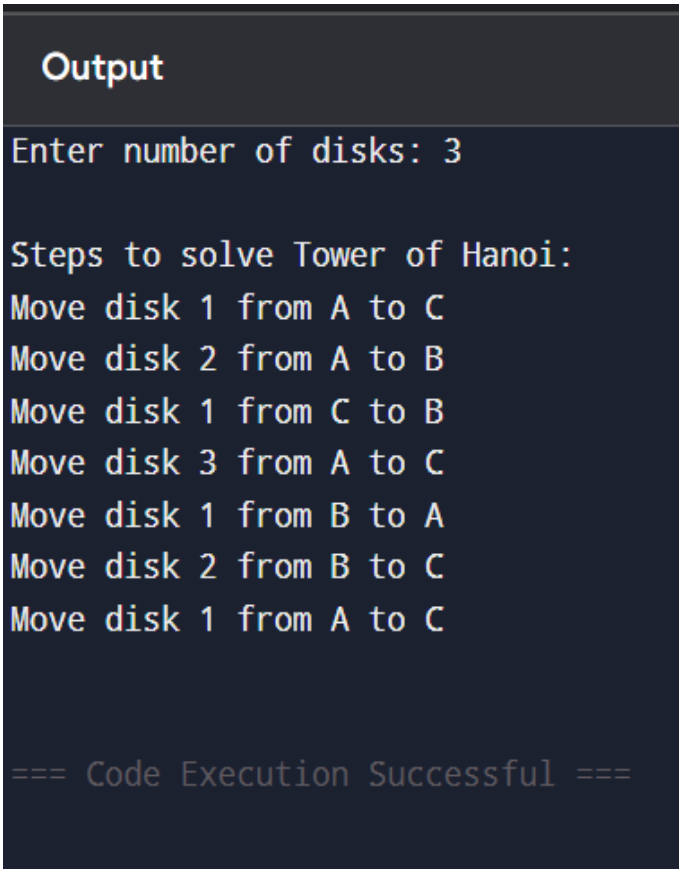
Simple Queue	Circular Queue
A Simple Queue is a linear data structure where insertion happens at the rear and deletion happens at the front.	A Circular Queue connects the last position back to the first position, forming a circular structure.
It follows the FIFO (First In, First Out) principle.	It also follows FIFO, but manages memory more efficiently due to circular linking.
The queue has a fixed linear end. When the rear reaches the end of the array, no more insertions are possible, even if space exists at the front.	The queue has no fixed end. When the rear reaches the end, it wraps around to the front if space is available.
Results in wastage of memory because free spaces created by deletion at the front cannot be reused.	Memory is fully utilized because empty spaces at the front are reused due to circular rotation.
Overflow occurs even when the queue has empty spaces at the front.	Overflow occurs only when completely full.
Uses two pointers: front and rear, which move only in forward direction.	Uses front and rear pointers, but they move in a circular manner (modulus operation).
Implementation is easier but less efficient in terms of memory management.	Implementation is slightly more complex but more efficient.
Example representation: [10, 20, 30, ---, ---] If rear reaches last index, insertion stops.	Example representation: [30, ---, ---, 10, 20] Rear wraps around to index 0.
Useful for simple, small applications like linear task scheduling.	Useful in scenarios requiring efficient memory use, such as operating system process scheduling, buffering, and network queues.

5. **How do singly and doubly linked lists differ from each other? Explain with Proper Example**

Singly Linked List (SLL)	**Doubly Linked List (DLL)**
Each node contains data and one pointer (next).	Each node contains data and two pointers (prev and next).
Traversal is only in forward direction.	Traversal is possible both forward and backward.
Requires less memory because it stores only one pointer.	Requires more memory because it stores two pointers.
Implementation is simple due to a single pointer.	Implementation is complex due to maintaining two pointers.
Deleting a node is difficult because there is no previous pointer.	Deleting a node is easier since previous pointer is available.
Cannot move backward once you move to the next node.	Can move backward easily using the prev pointer.
Useful in applications where memory is limited.	Useful where frequent insertions and deletions are required.
Example: 10 → 20 → 30 → NULL	Example: NULL ← 10 ⇄ 20 ⇄ 30 → NULL

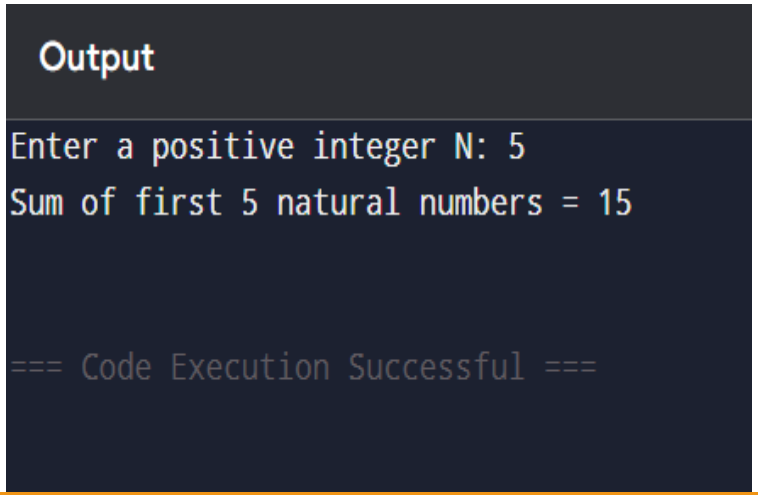
6. Write a Program to Perform Tower of Hanoi Problem.

```
// Function to perform Tower of Hanoi
void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod) {
    if (n == 1) {
        printf("Move disk 1 from %c to %c\n", from_rod, to_rod);
        return;
    }

    // Move top n-1 disks from source to auxiliary rod
    towerOfHanoi(n - 1, from_rod, aux_rod, to_rod);

    // Move nth disk from source to destination
    printf("Move disk %d from %c to %c\n", n, from_rod, to_rod);

    // Move n-1 disks from auxiliary to destination
    towerOfHanoi(n - 1, aux_rod, to_rod, from_rod);
}

int main() {
    int n;

    printf("Enter number of disks: ");
    scanf("%d", &n);

    printf("\nSteps to solve Tower of Hanoi:\n");
    towerOfHanoi(n, 'A', 'C', 'B'); // A = source, C = destination, B = auxiliary

    return 0;
}
```


Output
Enter number of disks: 3
Steps to solve Tower of Hanoi:
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
=== Code Execution Successful ===

7. **Write a Program to calculate the sum of the first N natural numbers using Recursive function.**

```
#include <stdio.h>

// Recursive function to compute sum of first N natural numbers
int sumN(int n) {
    if (n == 1) {
        return 1;   // Base case
    } else {
        return n + sumN(n - 1);  // Recursive case
    }
}

int main() {
    int n;

    printf("Enter a positive integer N: ");
    scanf("%d", &n);

    printf("Sum of first %d natural numbers = %d\n", n, sumN(n));

    return 0;
}
```



8. **Write a program to implement Insertion (Push) & deletion (Pop) operation using linked list in c.**

```
#include <stdio.h>
#include <stdlib.h>

struct node {
   int data;
   struct node *next;
};

struct node *top = NULL;

// Function to push an element into the stack
void push(int value) {
   struct node *newNode = (struct node*) malloc(sizeof (struct node));
   if (newNode == NULL) {
      printf("Stack Overflow (Memory not allocated)\n");
      return;
   }

   newNode->data = value;
   newNode->next = top;
   top = newNode;

   printf("%d pushed into the stack\n", value);
}

// Function to pop an element from the stack
void pop() {
   if (top == NULL) {
      printf("Stack Underflow (No elements to pop)\n");
      return;
   }

   struct node *temp = top;
   printf("%d popped from the stack\n", top->data);
   top = top->next;
   free(temp);
}
```

```
// Function to display the stack
void display() {
   if (top == NULL) {
      printf("Stack is empty\n");
      return;
   }

   struct node *temp = top;
   printf("Stack elements: ");
   while (temp != NULL) {
      printf("%d ", temp->data);
      temp = temp->next;
   }
   printf("\n");
}

int main() {
   int choice, value;

   while (1) {
      printf("\n--- Stack Menu ---\n");
      printf("1. Push\n");
      printf("2. Pop\n");
      printf("3. Display\n");
      printf("4. Exit\n");
      printf("Enter your choice: ");
      scanf("%d", &choice);

      switch (choice) {
         case 1:
            printf("Enter value to push: ");
            scanf("%d", &value);
            push(value);
            break;

         case 2:
            pop();
            break;
```

```
        case 3:
            display();
            break;

        case 4:
            exit(0);

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

}
```

Output

```
--- Stack Menu ---
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 2
Stack Underflow (No elements to pop)

--- Stack Menu ---
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
Stack is empty
```

9. Write a C Program to Implement a queue using array

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 5   // size of queue

int queue[MAX];
int front = -1, rear = -1;

// Function to insert (Enqueue) an element
void enqueue(int value) {
   if (rear == MAX - 1) {
      printf("Queue Overflow! Cannot insert %d\n", value);
      return;
   }
   if (front == -1) front = 0; // first insertion

   rear++;
   queue[rear] = value;
   printf("%d inserted into the queue\n", value);
}

// Function to delete (Dequeue) an element
void dequeue() {
   if (front == -1 || front > rear) {
      printf("Queue Underflow! No elements to delete\n");
      return;
   }

   printf("%d deleted from the queue\n", queue[front]);
   front++;
}

// Function to display queue elements
void display() {
   if (front == -1 || front > rear) {
      printf("Queue is empty\n");
      return;
   }
```

```
    printf("Queue elements: ");
    for (int i = front; i <= rear; i++) {
        printf("%d ", queue[i]);
    }
    printf("\n");
}

int main() {
    int choice, value;

    while (1) {
        printf("\n--- Queue Menu ---\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                enqueue(value);
                break;

            case 2:
                dequeue();
                break;

            case 3:
                display();
                break;

            case 4:
                exit(0);

            default:
                printf("Invalid choice! Try again.\n");
```

```
        }
    }

    return 0;
}
```

Output

```
--- Queue Menu ---
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter value to insert: 12
12 inserted into the queue

--- Queue Menu ---
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 3
Queue elements: 12
```

10. Write a program to implement Singly Linked List

```
#include <stdio.h>
#include <stdlib.h>

struct node {
   int data;
   struct node *next;
};

struct node *head = NULL;

// Insert node at beginning
void insertAtBeginning(int value) {
   struct node *newNode = (struct node*) malloc(sizeof(struct node));
   newNode->data = value;
   newNode->next = head;
   head = newNode;

   printf("%d inserted at beginning.\n", value);
}

// Insert node at end
void insertAtEnd(int value) {
   struct node *newNode = (struct node*) malloc(sizeof(struct node));
   newNode->data = value;
   newNode->next = NULL;

   if (head == NULL) {
      head = newNode;
   } else {
      struct node *temp = head;
      while (temp->next != NULL)
         temp = temp->next;
      temp->next = newNode;
   }

   printf("%d inserted at end.\n", value);
}
```

```
// Delete a node by value
void deleteNode(int value) {
   struct node *temp = head, *prev = NULL;

   if (temp != NULL && temp->data == value) {
      head = temp->next;
      free(temp);
      printf("%d deleted from the list.\n", value);
      return;
   }

   while (temp != NULL && temp->data != value) {
      prev = temp;
      temp = temp->next;
   }

   if (temp == NULL) {
      printf("Value %d not found in the list.\n", value);
      return;
   }

   prev->next = temp->next;
   free(temp);

   printf("%d deleted from the list.\n", value);
}

// Display Linked List
void display() {
   struct node *temp = head;

   if (temp == NULL) {
      printf("Linked List is empty.\n");
      return;
   }

   printf("Linked List: ");
   while (temp != NULL) {
      printf("%d → ", temp->data);
      temp = temp->next;
```

```
    }
    printf("NULL\n");
}

int main() {
    int choice, value;

    while (1) {
        printf("\n--- Singly Linked List Menu ---\n");
        printf("1. Insert at Beginning\n");
        printf("2. Insert at End\n");
        printf("3. Delete Node\n");
        printf("4. Display List\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value: ");
                scanf("%d", &value);
                insertAtBeginning(value);
                break;

            case 2:
                printf("Enter value: ");
                scanf("%d", &value);
                insertAtEnd(value);
                break;

            case 3:
                printf("Enter value to delete: ");
                scanf("%d", &value);
                deleteNode(value);
                break;

            case 4:
                display();
                break;
```

```
            case 5:
                exit(0);

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

Output

```
--- Singly Linked List Menu ---
1. Insert at Beginning
2. Insert at End
3. Delete Node
4. Display List
5. Exit
Enter your choice: 1
Enter value: 3
3 inserted at beginning.

--- Singly Linked List Menu ---
1. Insert at Beginning
2. Insert at End
3. Delete Node
4. Display List
5. Exit
Enter your choice: 5
```