ASSIGNMENT-JULY 2023
Sub. Name: - Data Structure
MCA – 103: Data Structure

Q-1 What do you mean by data and data structure? In how many ways you can categorize data structure?

Data:
Data refers to facts, information, or details that are raw and unorganised. It can take various forms, including numbers, text, images, and more. In computing, data is often processed and manipulated to extract meaningful information.

Data Structure:

A data structure is a way of organising and storing data to perform operations efficiently. It defines the relationship between data and the operations that can be performed on the data. The choice of a data structure depends on the nature of the data and the types of operations that need to be performed.

Ways to Categorise Data Structures:

Primitive Data Structures:
- These are basic and fundamental data structures that directly operate upon the machine instructions.
- Examples include integers, floating-point numbers, characters, and pointers.

Non-primitive Data Structures:
- These are more complex structures that are derived from primitive data structures.
- Examples include arrays, linked lists, stacks, queues, trees, and graphs.

Linear Data Structures:
- Data elements are arranged in a linear sequence or a linear fashion.
- Examples include arrays, linked lists, stacks, and queues.

Non-linear Data Structures:
- Data elements are not arranged in a sequential manner.
- Examples include trees and graphs.

Homogeneous Data Structures:
- All elements are of the same type.
- Example: Array of integers.

Heterogeneous Data Structures:

- Elements can be of different types.
- Example: Structure in C or class in C++.

Static Data Structures:

- Size and structure are fixed at compile time.
- Example: Arrays.

Dynamic Data Structures:

- Size can be changed during runtime.
- Examples include linked lists, trees, and graphs.

Abstract Data Types (ADTs):

- Data types whose behaviour is defined by a set of operations, but the implementation details are hidden.
- Examples include stacks, queues, and lists.

Concrete Data Types:

- Actual implementation of an abstract data type.
- Example: A linked list implementing the list ADT.

Q.2 What are the concepts of arrays in data structure? What is two-dimensional array?

Arrays in Data Structures:

An array is a data structure that stores a fixed-size sequential collection of elements of the same type. Each element in an array is identified by an index or a key. The main concepts associated with arrays are:

Declaration:

- An array is declared by specifying the type of its elements and its name, followed by square brackets that denote the size of the array.
- Example in C: `int myArray[5];z`

Initialization:

- The array elements can be initialized at the time of declaration or later in the program.
- Example in C: `int myArray[5] = {1, 2, 3, 4, 5};`

Indexing:

- Elements in an array are accessed using their index. The index starts from 0.
- Example: In `myArray[2]`, the index is 2, and the value is the third element of the array.

Size:

- The size of an array is fixed at the time of declaration and cannot be changed during runtime.

Contiguous Memory Allocation:

- Array elements are stored in contiguous memory locations, making it efficient to access elements using index calculations.

Homogeneous Elements:

- All elements in an array must be of the same data type.

Random Access:

- Array elements can be accessed directly using their index, allowing for constant-time access.

Two-Dimensional Array:

A two-dimensional array is an array of arrays, essentially creating a matrix-like structure. It is often used to represent tables of values with rows and columns. In memory, a 2D array is stored as a contiguous block of memory, and the elements are accessed using two indices (row and column).

Declaration and Initialization:

```
int matrix[3][4];  // Declaration of a 3x4 matrix
int matrix[3][4]  = { {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12} }; // Initialization
```

Accessing Elements:

```
int element = matrix[1][2]; // Accessing the element in the second row and third column
```

Memory Representation:

matrix[0][0]	matrix[0][1]	matrix[0][2]	matrix[0][3]
matrix[1][0]	matrix[1][1]	matrix[1][2]	matrix[1][3]
matrix[2][0]	matrix[2][1]	matrix[2][2]	matrix[2][3]

Key Points:

- A 2D array is like an array of arrays, where each element is identified by two indices.
- It is often used to represent grids, tables, or matrices in programming.
- The memory is allocated in a contiguous block, and the elements are accessed using row and column indices.
- Initialization can be done at the time of declaration or later in the program.

Q3. What are the basic concepts of linked list in data structure?

A linked list is a fundamental data structure in computer science that consists of a sequence of elements, where each element points to the next one in the sequence. The basic concepts of a linked list include:

1. **Node:**
    - A node is the fundamental building block of a linked list.
    - Each node contains two components: data (or payload) and a reference (or link) to the next node in the sequence.

2. **Head:**
    - The head is the reference to the first node in the linked list.
    - It serves as the starting point for traversing the list.

3. **Tail:**
    - The tail is the last node in the linked list.
    - The reference in the tail points to a null or a special value to indicate the end of the list.

4. **Next (or Link) Field:**
    - Each node contains a reference to the next node in the sequence.
    - This reference is often called the "next" field, and it holds the memory address of the next node.

5. **Singly Linked List:**
    - In a singly linked list, each node points only to the next node in the sequence.

- Traversal is only possible in the forward direction.

6. Doubly Linked List:

- In a doubly linked list, each node has references to both the next and the previous nodes.
- Traversal can be done in both forward and backward directions.

7. Circular Linked List:

- In a circular linked list, the last node points back to the first node, forming a circle.
- Useful in cases where cyclic access is needed.

8. Linked List Operations:

- **Insertion:**
    - Adding a new node to the linked list.
    - Insertion can occur at the beginning, end, or at a specific position.

- **Deletion:**
    - Removing a node from the linked list.
    - Deletion can occur at the beginning, end, or at a specific position.

- **Traversal:**
    - Visiting each node in the linked list to perform an operation.

- **Search:**
    - Finding a specific element in the linked list.

9. Dynamic Size:

- Linked lists can dynamically grow or shrink in size during program execution.
- Memory is allocated as needed, and nodes can be easily inserted or removed.

10. Memory Efficiency:

- Unlike arrays, linked lists do not require contiguous memory allocation.
- Memory is allocated on-demand for each node.

11. No Fixed Size:

- Linked lists can change in size without the need to specify the size at the time of declaration.

Linked lists are used in situations where dynamic memory allocation and flexibility in size are important, but they may have higher overhead compared to arrays in terms of memory usage and access times. The choice between arrays and linked lists depends on the specific requirements of the application.

Q-4 What are the advantages of Linked List over Arrays? Write an algorithm to insert an element at given position in the Single Linked list.

Advantages of Linked Lists over Arrays:

1. **Dynamic Size:**
    - Linked lists can dynamically grow or shrink in size, as memory is allocated as needed. In contrast, arrays have a fixed size, and resizing them can be inefficient.

2. **Efficient Insertion and Deletion:**
    - Inserting or deleting elements in a linked list can be more efficient than in arrays, especially in the middle of the list. In arrays, shifting elements is required after insertion or deletion.

3. **No Wasted Memory:**
    - Linked lists don't require a fixed, contiguous block of memory. Memory is allocated as needed for each node, avoiding wasted space.
    -

4. **Ease of Implementation:**
    - Implementing certain data structures and algorithms is more straightforward with linked lists. For example, implementing a stack or a queue is often simpler with linked lists.

5. **No Pre-allocation of Space:**
    - Unlike arrays, linked lists don't require pre-allocation of space. Memory is allocated on-demand, making them suitable for situations with uncertain data size.

Algorithm to Insert an Element at a Given Position in a Single Linked List:

Assuming you have a singly linked list structure with nodes having a "data" field and a "next" pointer, here's an algorithm to insert an element at a given position:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def insert_at_position(self, position, data):
        new_node = Node(data)

        # If position is 0, insert at the beginning
        if position == 0:
            new_node.next = self.head
            self.head = new_node
            return

        current = self.head
        count = 0
        # Traverse to the node before the desired position
        while current is not None and count < position - 1:
            current = current.next
            count += 1

        # Check if position is valid
        if current is None:
            print("Invalid position")
            return

        # Perform the insertion
        new_node.next = current.next
        current.next = new_node

    def display(self):
        current = self.head
        while current is not None:
            print(current.data, end=" -> ")
            current = current.next
        print("None")

# Example usage:
```

```
linked_list = LinkedList()

linked_list.insert_at_position(0, 5)

linked_list.insert_at_position(1, 10)

linked_list.insert_at_position(1, 8)

linked_list.insert_at_position(3, 12)

linked_list.display()
```

In this example, `insert at position` inserts a new node with the given data at the specified position in the linked list. The `display` method is used to print the elements of the linked list.

Q-5 Explain difference between Stack and Queue

A stack and a queue are both abstract data types that represent collections of elements, but they differ in how elements are added and removed.

Stack:

Ordering Principle: Follows the Last In, First Out (LIFO) principle. The last element added to the stack is the first one to be removed.

Operations:

- **Push:** Adds an element to the top of the stack.
- **Pop:** Removes the element from the top of the stack.
- **Peek (or Top):** Retrieves the element from the top of the stack without removing it.

Real-world Analogy: Think of it like a stack of plates. You add a plate to the top and remove the topmost plate.

Queue:

Ordering Principle: Follows the First In, First Out (FIFO) principle. The first element added to the queue is the first one to be removed.

Operations:

- E**nqueue:** Adds an element to the back (or rear) of the queue.
- **Dequeue:** Removes the element from the front (or head) of the queue.
- **Front:** Retrieves the element from the front without removing it.
- **Rear (or Back):** Retrieves the element from the back without removing it.

Real-world Analogy: Think of it like people waiting in line. The person who arrived first is the first one to proceed.

Q-6 What are the advantages and disadvantages of stack and queue implemented using linked list over array?

Implementing a stack or a queue using linked lists has its own set of advantages and disadvantages compared to using arrays.

Advantages of using Linked Lists:

1. **Dynamic Size:** Linked lists allow for dynamic memory allocation, meaning the size of the stack or queue can change during runtime. This is in contrast to arrays, which have a fixed size.

2. **Ease of Insertion and Deletion:** Insertions and deletions at both ends of a linked list (front for a queue, and front or end for a stack) are typically faster and more efficient compared to arrays. This is because linked lists don't require shifting of elements, unlike arrays.

3. **No Wasted Memory:** Linked lists use memory more efficiently since they allocate memory only as needed. In contrast, arrays may need to allocate a larger chunk of memory than necessary, leading to potential wasted space.

Disadvantages of using Linked Lists:

1. **Increased Overhead:** Linked lists have additional overhead due to the storage of pointers/references for each element, leading to increased memory consumption compared to arrays.

2. **Random Access Limitation:** Unlike arrays, linked lists do not provide constant-time random access to elements. Accessing an element in a linked list requires traversing the list from the beginning, which can be inefficient for certain option.

3. **Cache Locality:** Arrays have better cache locality compared to linked lists. Accessing elements in an array is more cache-friendly, as they are stored in contiguous memory locations, reducing cache misses. Linked lists, with their scattered memory locations, may result in more cache misses.

4. **Complexity of Implementation:** Implementing a linked list-based stack or queue involves managing pointers and dynamic memory allocation, which can add complexity to the code compared to the simplicity of array-based implementations.

Q-7 Explain the Preorder, Inorder and Postorder traversal techniques of the binary tree with suitable

Traversal techniques in binary trees refer to the order in which we visit or process the nodes of the tree. There are three main types of binary tree traversals: preorder, inorder, and postorder.

Consider the following binary tree for illustration:

```
    A

   / \

  B   C

 / \

D   E
```

1. Preorder Traversal:

In preorder traversal, we visit the root node before its children. The order is Root, Left, Right.

For the example tree:

Preorder: A, B, D, E, C

2. Inorder Traversal:

In inorder traversal, we visit the left child, then the root, and finally the right child. The order is Left, Root, Right.

For the example tree:

Inorder: D, B, E, A, C

3. Postorder Traversal:

In postorder traversal, we visit the children before the root. The order is Left, Right, Root.

For the example tree:

Postorder: D, E, B, C, A

These traversal techniques are fundamental in tree-based algorithms and are used for various applications, such as expression tree evaluation, tree-based searches, and constructing postfix expressions. Understanding these traversals helps in analyzing and manipulating binary trees efficiently.

Q-8 Consider the following arithmetic expression P, written in postfix notation. Translate it in infix notation and evaluate.

To translate the given postfix expression P into infix notation, we can use a stack to keep track of operands. The expression is processed from left to right, and operands are pushed onto the stack. When an operator is encountered, the necessary operands are popped from the stack, and the result is pushed back onto the stack. Here's the step-by-step translation:

Postfix expression P: 12, 7, 3, -, /, 2, 1, 5, +, *, +

Process each element from left to right:

- Push 12 onto the stack.
- Push 7 onto the stack.
- Push 3 onto the stack.
- Encounter the subtraction operator (-):
    - Pop two operands (3 and 7) from the stack.
    - Perform the operation: 7 - 3 = 4.
    - Push the result (4) back onto the stack.

- Encounter the division operator (/):
    - Pop two operands (12 and 4) from the stack.
    - Perform the operation: 12 / 4 = 3.
    - Push the result (3) back onto the stack.
- Push 2 onto the stack.
- Push 1 onto the stack.
- Push 5 onto the stack.
- Encounter the addition operator (+):
    - Pop two operands (5 and 1) from the stack.
    - Perform the operation: 1 + 5 = 6.
    - Push the result (6) back onto the stack.
- Encounter the multiplication operator (*):
    - Pop two operands (2 and 6) from the stack.
    - Perform the operation: 2 * 6 = 12.
    - Push the result (12) back onto the stack.
- Encounter the final addition operator (+):
    - Pop two operands (3 and 12) from the stack.
    - Perform the operation: 3 + 12 = 15.
    - Push the result (15) back onto the stack.

The final result on the stack is 15. Therefore, the infix expression is:

$$\frac{12-7+3\times(2+1+5)15}{15}$$

And the evaluated result is 1.

Q-9 What do you mean by traversal of any Graph?

Graph traversal is the process of visiting all the vertices (nodes) and edges of a graph in a systematic way. It involves systematically exploring each vertex and its neighboring vertices in a graph. The traversal can start from any arbitrary vertex, and during the process, all the vertices and edges are visited exactly once, ensuring that the entire graph is explored.

There are two common methods for graph traversal:

1. Depth-First Search (DFS): In DFS, the algorithm explores as far as possible along each branch before backtracking. It uses a stack (either explicitly or through recursion) to keep track of the vertices to be visited.
2. Breadth-First Search (BFS): In BFS, the algorithm explores all the neighbors of a vertex before moving on to the next level of vertices. It uses a queue to keep track of the vertices to be visited, ensuring that vertices are visited in the order of their distance from the starting vertex.

Graph traversal is a fundamental operation in graph theory and is used in various applications, such as finding paths between two vertices, checking for connectivity, detecting cycles, and solving network flow problems. The choice of traversal method depends on the specific requirements of the problem and the characteristics of the graph.

Q-10 What is the basic of recursion in data structure? What are the rules of recursion in data structure?
Recursion is a programming and problem-solving technique where a function calls itself in its own definition. In the context of data structures, recursion is often used to solve problems that can be broken down into smaller instances of the same problem. The basic idea of recursion involves breaking a problem into smaller sub-problems and solving each sub-problem recursively until a base case is reached.

Here are the fundamental principles and rules of recursion in the context of data structures:

1. **Base Case:**
    - Every recursive algorithm must have one or more base cases.
    - The base case defines the simplest scenario for which the solution is known without further recursion.
    - The base case ensures that the recursion eventually terminates.

2. **Divide and Conquer:**
    - Recursion involves breaking down a problem into smaller, more manageable sub-problems.
    - Each recursive call should work on a smaller instance of the original problem.

3. **Self-Call:**

- The function calls itself to solve a smaller instance of the same problem.
- The function should make progress toward reaching the base case during each recursive call.

4. **Progress Toward Base Case:**
    - In each recursive call, the state of the problem should move closer to the base case.
    - This ensures that the recursion eventually reaches the base case and terminates.

5. **Save State:**
    - When making a recursive call, the current state of the function (local variables, parameters) is saved.
    - The saved state is restored when the recursive call returns, allowing the function to continue its execution.

6. **Example:**

    - Consider the classic example of factorial calculation: n!=n×(n−1)! with the base case 0 ! = 1 0!=1.

    - The recursive function in this case would be like:

```
def factorial(n):
   # Base case
   if n == 0:
      return 1
   # Recursive call
   else:
      return n * factorial(n - 1)
```

Recursion is a powerful concept in computer science and programming, allowing elegant solutions to certain types of problems. However, it should be used judiciously, and excessive recursion can lead to stack overflow errors. Understanding the base case and ensuring progress toward it is crucial for the correct and efficient implementation of recursive algorithms.