5.1.1 Identify a situation that requires the use of recursive thinking

What is Recursive Thinking?

- Solving a problem by **breaking it into smaller instances** of itself.
- **Key features**:
  - **Base case**: when recursion stops.
  - **Recursive step**: method calls itself with a smaller input.

Example 1: Fractals (Snowflake Generation)

Fractals show **self-similarity**: parts look like the whole.

Recursive Idea:

- Draw a line segment.
- Subdivide into smaller segments.
- Repeat for each segment.

Pseudocode:

```
METHOD DrawFractal(level, length)
    IF level = 0 THEN
        Draw straight line of 'length'
    ELSE
        FOR each segment (usually 4 segments)
            DrawFractal(level - 1, length / 3)
            Turn by specific angle (e.g., 60 degrees)
        ENDFOR
    ENDIF
ENDMETHOD
```

Example 2: Calculating Factorial

Factorial of a number n (written as n!) is:

- n! = n × (n−1) × (n−2) × … × 1
- Special case: 0! = 1

Recursive Idea:

- n! = n × (n−1)!
- Base case: 0! = 1

Pseudocode:

```
METHOD Factorial(n)
    IF n = 0 THEN
```

```
        RETURN 1
    ELSE
        RETURN n × Factorial(n-1)
    ENDIF
ENDMETHOD
```

Example:

Calculate 3!

- Factorial(3) → 3 × Factorial(2)
- Factorial(2) → 2 × Factorial(1)
- Factorial(1) → 1 × Factorial(0)
- Factorial(0) → 1 (base case)

So:

Factorial(1) = 1

Factorial(2) = 2 × 1 = 2

Factorial(3) = 3 × 2 = 6

5.1.2 Identify recursive thinking in a specified problem solution

Example: Binary Tree Traversal

A binary tree is **naturally recursive**:

- Each node acts as a tree.

Recursive Idea:

- Traverse left subtree.
- Visit node.
- Traverse right subtree.

  (**Inorder Traversal**)

Pseudocode:

```
METHOD InorderTraversal(node)
    IF node ≠ null THEN
        InorderTraversal(node.left)
        OUTPUT node.value
        InorderTraversal(node.right)
    ENDIF
ENDMETHOD
```

5.1.3 Trace a recursive algorithm to express a solution to a problem

Example Trace: Inorder Traversal of a Binary Tree

Tree structure:

```
      4
     / \
    2   5
   / \
  1   3
```

Tracing Steps:

1. InorderTraversal(4)
2. → InorderTraversal(2)
3. → InorderTraversal(1)
   - Left null → OUTPUT 1
   - Right null
4. Back to 2 → OUTPUT 2
5. → InorderTraversal(3)
   - Left null → OUTPUT 3
   - Right null
6. Back to 4 → OUTPUT 4
7. → InorderTraversal(5)
   - Left null → OUTPUT 5
   - Right null

Final Output:

1 2 3 4 5

5.1.4 Describe the characteristics of a two-dimensional array

What is a Two-Dimensional Array?

- A **collection of data** organized in **rows and columns**.
- It is like a **table** or **matrix**.
- Each element is **accessed** using **two indices** (row and column).

Example:

An array of 3 rows and 4 columns:

```
[ [1, 2, 3, 4],
  [5, 6, 7, 8],
  [9, 10, 11, 12] ]
```

Access array[1][2] → value 7.

Link: 1D arrays are lists, 2D arrays extend lists into a grid.

5.1.5 Construct algorithms using two-dimensional arrays

Example 1: Initializing a 2D Array

Pseudocode:

```
METHOD InitializeArray(rows, cols)
    DECLARE array[rows][cols]
    LOOP FOR i FROM 0 TO rows-1
        FOR j FROM 0 TO cols-1
            array[i][j] ← 0
        ENDFOR
    ENDFOR
    RETURN array
ENDMETHOD
```

Example 2: Summing all elements in a 2D Array

Pseudocode:

```
METHOD SumElements(array, rows, cols)
    DECLARE total ← 0
    FOR i FROM 0 TO rows-1
        FOR j FROM 0 TO cols-1
            total ← total + array[i][j]
        ENDFOR
    ENDFOR
    RETURN total
ENDMETHOD
```

5.1.6 Describe the characteristics and applications of a stack

What is a Stack?

- **Last In, First Out (LIFO)** structure.
- Only the **top element** is accessible.

Applications:

- Running **recursive processes** (function calls are pushed onto stack).
- Storing **return memory addresses** during program execution.

5.1.7 Construct algorithms using the access methods of a stack

Stack Access Methods:

- **push(value):** Add a value on top.
- **pop():** Remove and return the top value.
- **isEmpty():** Check if the stack is empty.

Pseudocode:

```
METHOD push(stack, value)
    stack.addToEnd(value)
ENDMETHOD

METHOD pop(stack)
    IF isEmpty(stack) THEN
        OUTPUT "Stack Underflow"
    ELSE
        RETURN stack.removeFromEnd()
    ENDIF
ENDMETHOD

METHOD isEmpty(stack)
    RETURN stack.size = 0
ENDMETHOD
```

5.1.8 and 5.1.9 Describe the characteristics and applications of a queue

What is a Queue?

- **First In, First Out (FIFO)** structure.
- Elements are inserted at **rear** and removed from **front**.

Applications:

- **Print queues**: Jobs printed in order of arrival.
- **Simulating physical queues**: Like at supermarket checkouts.

Implementations:

- **Linear queue:** Straightforward.
- **Circular queue:** Reuses space after elements are removed (efficient).

Construct algorithms using access methods of a queue

Queue Access Methods:

- **enqueue(value):** Insert value at rear.
- **dequeue():** Remove and return value from front.
- **isEmpty():** Check if the queue is empty.

Pseudocode:

```
METHOD enqueue(queue, value)
    queue.addToEnd(value)
ENDMETHOD

METHOD dequeue(queue)
    IF isEmpty(queue) THEN
        OUTPUT "Queue Underflow"
    ELSE
        RETURN queue.removeFromStart()
    ENDIF
ENDMETHOD

METHOD isEmpty(queue)
    RETURN queue.size = 0
ENDMETHOD
```

5.1.10 Explain the use of arrays as static stacks and queues

Using Arrays as Stacks:

- ***push***: Place element at the next available index.
- ***pop***: Remove element from the current top index.
- Need to **check for full** (before push) and **empty** (before pop).

Example (Stack using Array):

```
METHOD push(stack, top, value, maxSize)
    IF top = maxSize THEN
        OUTPUT "Stack Overflow"
    ELSE
        top ← top + 1
        stack[top] ← value
    ENDIF
ENDMETHOD

METHOD pop(stack, top)
    IF top = -1 THEN
        OUTPUT "Stack Underflow"
    ELSE
        value ← stack[top]
        top ← top - 1
        RETURN value
    ENDIF
ENDMETHOD
```

Using Arrays as Queues:

- ***enqueue***: *Insert element at rear.*
- ***dequeue***: *Remove element from front and shift others if linear.*

Example (Queue using Array):

```
METHOD enqueue(queue, rear, value, maxSize)
    IF rear = maxSize THEN
        OUTPUT "Queue Overflow"
    ELSE
        rear ← rear + 1
        queue[rear] ← value
    ENDIF
ENDMETHOD

METHOD dequeue(queue, front, rear)
    IF front > rear THEN
        OUTPUT "Queue Underflow"
    ELSE
        value ← queue[front]
        front ← front + 1
        RETURN value
    ENDIF
ENDMETHOD
```

- ***Circular queue*** *avoids shifting by wrapping around.*

5.1.11 Describe the features and characteristics of a dynamic data structure

Dynamic Data Structures:

- **Size is flexible**: Can grow or shrink during execution.
- **Efficient memory use**: Only allocates memory when needed.
- **Organized as nodes**: Each node stores **data** and a **pointer** (reference) to the next node.

Key Concepts:

- **Node**: A container that holds a **data value** and one (or more) **pointers**.
- **Pointer**: A reference/link to another node's memory address.

5.1.12 Describe how linked lists operate logically

Logical operation of linked lists:

- A linked list consists of a **series of nodes** connected by **pointers**.
- Each node points to the **next node** in the sequence.
- The **first node** is called the **head**.
- The **last node** points to **null** (meaning end of the list).

Main Operations:

- **Traversal**: Start from the head and follow pointers node by node.
- **Insertion**: Adjust pointers to add a new node without breaking the chain.
- **Deletion**: Redirect pointers to remove a node from the chain.
- **Searching**: Traverse the list to find a specific data item.

5.1.13 Sketch linked lists (single, double and circular)

Single Linked List

Each node points to the next node.

The last node points to null.

[Data|Next] → [Data|Next] → [Data|Next] → null

Adding an item:

- Create a new node.
- Set its pointer to the next node.
- Adjust the previous node's pointer.

Deleting an item:

- Redirect the pointer of the previous node to the next node.

Double Linked List

Each node has two pointers: one to the **next** node and one to the **previous** node.

null ← [Prev|Data|Next] ↔ [Prev|Data|Next] ↔ [Prev|Data|Next] → null

Adding an item:

- Update four pointers (new node's prev and next, adjacent nodes' next and prev).

Deleting an item:

- Adjust the previous and next node pointers to bypass the node.

Circular Linked List

The last node points back to the first node.

(Single Circular Linked List):

[Data|Next] → [Data|Next] → [Data|Next] ↺ (points back to first node)

(Double Circular Linked List):

(first node prev points to last node) ← [Prev|Data|Next] ↔ [Prev|Data|Next] ↔ [Prev|Data|Next] → (last node next points to first node)

Main Point:

Traversal never hits null — it keeps cycling through the list.

5.1.14 Describe how trees operate logically (both binary and non-binary)

Trees (General):

- A **hierarchical** dynamic data structure.
- Made up of **nodes** connected by **pointers**.
- Each node may have **zero or more child nodes**.
- Top node is called the **root**.

Binary Trees:

- A **special type** of tree where **each node has at most two children**:
  - **Left child**
  - **Right child**

Non-Binary Trees:

- Nodes can have **more than two children** (sometimes unlimited).

Logical Operations on Trees:

- **Traversal**: Visit all nodes in a specific order (e.g., inorder, preorder, postorder).
- **Insertion**: Add a new node while maintaining tree rules.
- **Deletion**: Remove a node and reconnect the tree properly.
- **Searching**: Find a specific node by following paths.

Link to Recursive Thinking:

Tree operations are naturally recursive, because each subtree is itself a smaller tree.

5.1.15 Define the terms: parent, left-child, right-child, subtree, root and leaf

Term	Definition
Parent	A node that has one or more child nodes.
Left-child	The child node connected on the left side.
Right-child	The child node connected on the right side.
Subtree	A tree structure consisting of a node and its descendants.
Root	The topmost node in the tree (no parent).
Leaf	A node with **no children**.

(These definitions apply specifically to **binary trees**.)

5.1.16 State the result of inorder, postorder and preorder tree traversal

Inorder Traversal (Left → Root → Right)

- Visit the left subtree.
- Visit the root node.
- Visit the right subtree.

➡️ **Result**: Nodes are visited in **sorted order** (for binary search trees).

Preorder Traversal (Root → Left → Right)

- Visit the root node.
- Visit the left subtree.
- Visit the right subtree.

➡️ **Result**: The root node is always visited **first**.

Postorder Traversal (Left → Right → Root)

- Visit the left subtree.
- Visit the right subtree.
- Visit the root node.

➡️ **Result**: The root node is visited **last**.

5.1.17 Sketch binary trees

You must be able to **sketch**:

- A **binary tree** from a given sequence.
- Resulting tree after **adding** nodes.
- Resulting tree after **removing** nodes.

Example: Binary Tree Sketch

Suppose we insert the values: 7, 4, 9, 2, 5

Step-by-Step Insertion:

Insert 7 → becomes the root.

Insert 4 → smaller than 7 → goes to the left of 7.

Insert 9 → greater than 7 → goes to the right of 7.

Insert 2 → smaller than 4 → goes to the left of 4.

Insert 5 → greater than 4 → goes to the right of 4.

Resulting Tree:

```
    7
   / \
  4   9
 / \
2   5
```

Example: Removing a Node

Remove 4:

- 4 has two children.
- Find the **inorder successor** (smallest node in the right subtree → 5).
- Replace 4 with 5.

Resulting Tree:

```
      7
     / \
    5   9
   /
  2
```

5.1.18 Define the term dynamic data structure

Dynamic Data Structure:

- **A data structure whose size can change during program execution.**
- **Memory allocation is done at runtime** based on the need.
- Can grow or shrink in response to operations like adding or removing elements.

Examples:

- **Linked Lists**: Can dynamically add or remove nodes.
- **Stacks** and **Queues** (implemented with arrays or linked lists).

5.1.19 Compare the use of static and dynamic data structures

Aspect	**Static Data Structures**	**Dynamic Data Structures**
Size	Fixed size; cannot change during runtime.	Size can change dynamically at runtime.
Memory Allocation	Memory allocated at compile-time.	Memory allocated at runtime.
Efficiency	Faster access as memory is contiguous.	Slower access due to non-contiguous memory allocation.
Flexibility	Less flexible; predefined size.	More flexible; can grow and shrink as needed.

Example:

- **Static Structure**: An array with a fixed size.
- **Dynamic Structure**: A linked list that grows as nodes are added.

5.1.20 Suggest a suitable structure for a given situation

Situation 1: Implementing a Stack

- **Recommended Structure**: **Dynamic (Linked List or Array-based Stack)**.
    - **Reason**: Stacks require constant push and pop operations, which benefit from dynamic memory allocation to prevent overflow.
    - **Dynamic Structure**: A linked list (where the stack grows/shrinks dynamically) or a resizable array.

Situation 2: Queue for Print Jobs

- **Recommended Structure**: **Dynamic (Queue using Linked List or Array-based Queue)**.
    - **Reason**: Queues are FIFO (First In, First Out), so memory should grow or shrink as print jobs are added or removed.
    - **Dynamic Structure**: A queue implemented with a linked list or a circular array.

Situation 3: Database Management (Fixed Size Table)

- **Recommended Structure**: **Static (Array)**.
    - **Reason**: The table size is predefined, so a static structure is suitable because memory allocation is known in advance.
    - **Static Structure**: A static array or a 2D array for storing database records.