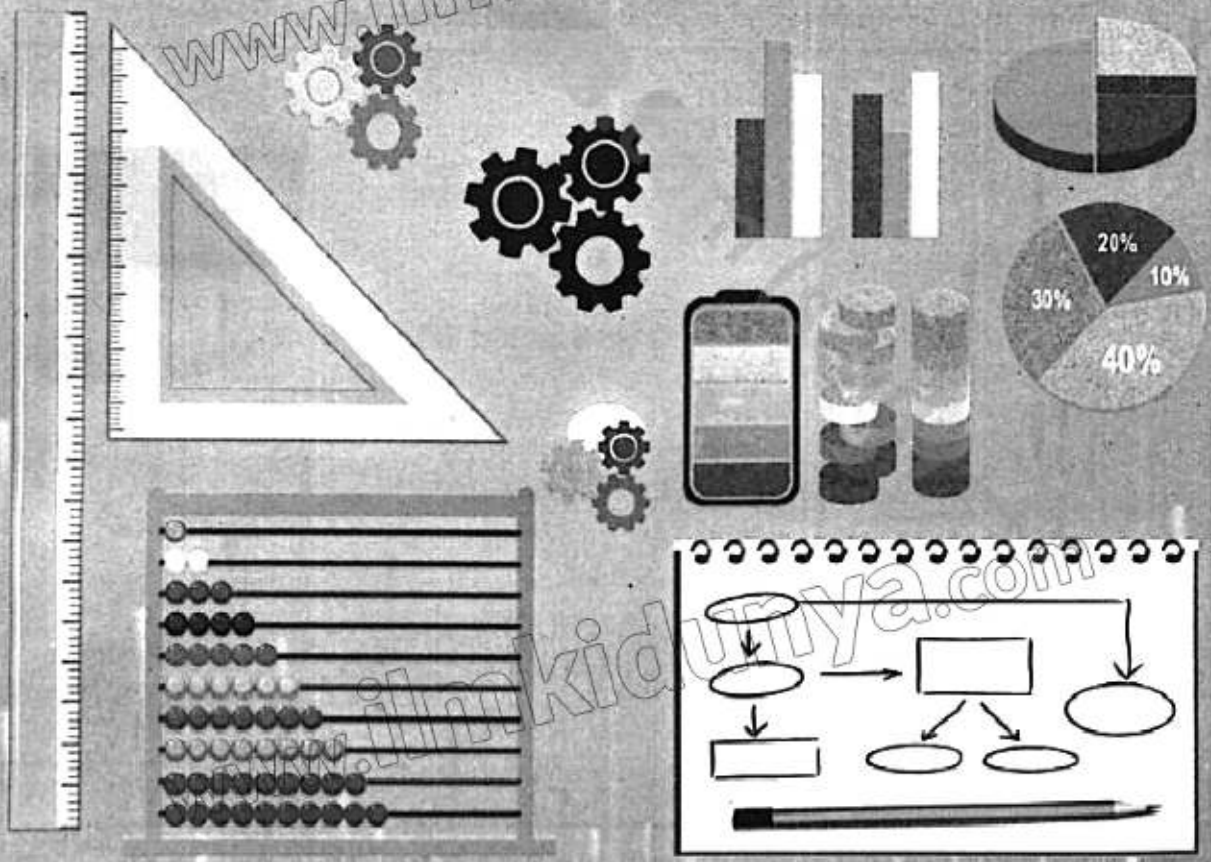
2 Computational Thinking & Algorithms

Learning Outcomes

At the end of this unit students will be able to:

- Identify and apply complex algorithms on data structures such as trees and binary search
- understand and evaluate the computational solutions in terms of efficiency , clarity and correctness



Introduction

Computational Thinking (CT) is a problem-solving process rooted in computer science principles. It enables individuals to tackle complex challenges by breaking them down into manageable parts and approaching them logically and systematically. Although CT is fundamental to computer science, its application extends far beyond – into science, engineering, business, medicine, and everyday life.

The core pillars of computational thinking include:

- **Decomposition:** breaking a problem into smaller, more manageable components.
- **Pattern Recognition:** identifying similarities or trends to simplify complex tasks.
- **Abstraction:** focusing on important information and ignoring irrelevant details.
- **Algorithm Design:** developing clear, step-by-step instructions to solve a problem.

Together, these elements form the foundation for designing effective and efficient computational solutions. At this advanced level, Computational thinking is not only applied to design algorithms but also is used to analyze and evaluate these solutions in terms of correctness, clarity, and efficiency. Understanding data structures is central to this process, as they greatly influence how well a solution performs.

In this chapter, it will be explored how computational thinking integrates with algorithms and data structures to create solutions that are not only correct but also optimized and scalable.

2.1 Data Structures

Data structure defines the way for organizing and storing data so that it can efficiently be accessed and modified. They provide a framework for managing and organizing data in computer programs, allowing for various operations to be performed effectively.

Data structures are essential for efficiently organizing and managing data in computer programs. They dictate how data is arranged in memory and impact the performance of various operations like insertion, deletion and retrieval. For instance, arrays provide quick access to elements by index, while linked lists offer flexibility in dynamic data manipulation. The choice of data structure e.g. stacks and queues, trees and graphs depends on the specific needs of the application and the operations it performs.

Following are some common data structures:

2.1.1 Arrays

Arrays are the simplest data structure used in programming that contain different elements of same size. Contiguous memory locations are used to store these elements. The memory locations are identified by indexes as shown in Fig.2.1.

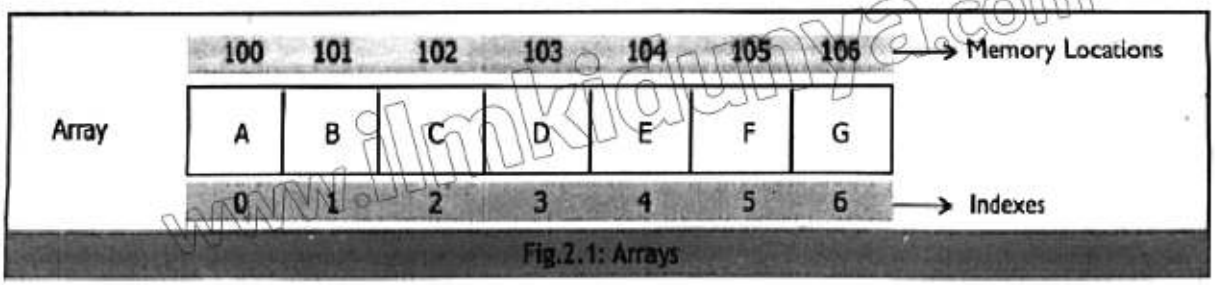


Fig.2.1: Arrays

The arrays are considered efficient because they do not require additional data (metadata) to store the elements. Additionally, because the memory locations are contiguous, therefore, it makes accessing the elements fast.

However, the arrays are fixed in size and it requires to declare its size at the start. At some later stage, when arrays need to be resized, it is costly to do so, because it may require creating new arrays with larger size and shifting of elements from old array to new. Similarly, the arrays could not be a good choice when elements are of different type.

The arrays are of two different types:

One-Dimentional (1-D) Array: In 1-D array, various elements (of same type) are stored in a single list. The element can be reached by knowing only a single index. One-Dimentional (1-D) Array example is shown in Fig.2.2.

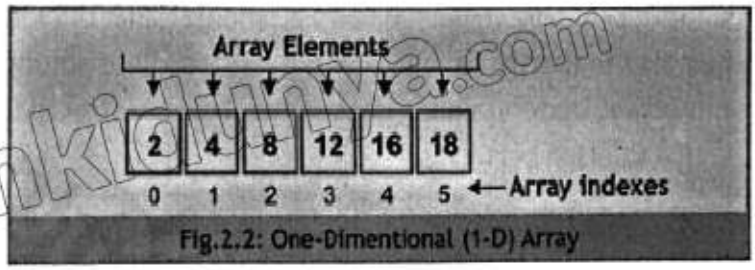


Fig.2.2: One-Dimentional (1-D) Array

Two-Dimentional (2-D) Array: The 2-D array is also known as matrix, where the elements are stored in the form of matrix or table. The elements can only be reached by knowing two indexes e.g. row and column. Fig.2.3 shows an example of Two-Dimentional (2-D) Arrays.

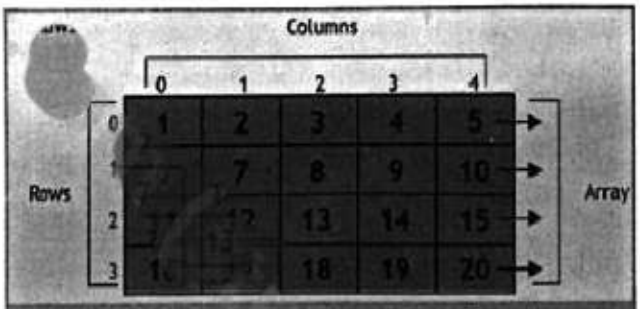


Fig.2.3: Two-Dimentional (2-D) Array

2.1.2 Linked Lists

Linked Lists are basic data structures in which elements (called nodes) are connected to each other using the concept of pointers. The nodes contain the values to be stored and pointer is the reference (memory address) of the next node in the sequence as shown in Fig.2.4.

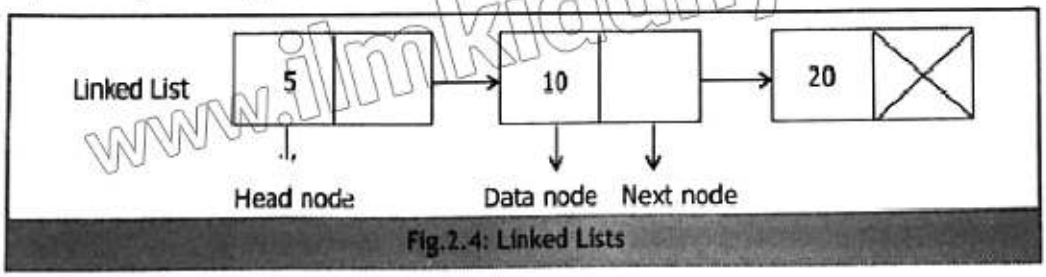


Fig.2.4: Linked Lists

The linked lists are capable to store different types of data and it also makes insertion and deletion operations easy as compared to arrays. The insertion/deletion could be performed at

any position in the sequence of linked list. Another advantage of linked lists is that they are dynamic in size (no need to specify the size of linked list at the start, rather they can grow and shrink as per requirement.

Singly Linked List

Singly linked list is basic and simple type of list, it maintains two parts in each node.

- data to be stored
- reference to next node in the sequence

Fig. 2.5 shows an example scenario:

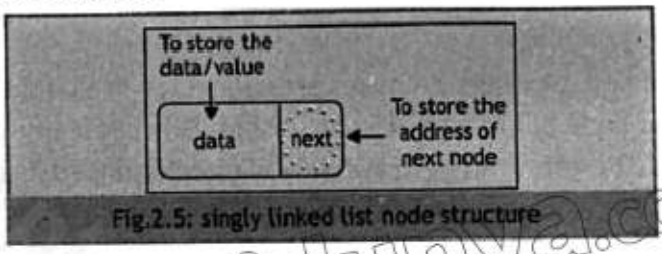


Fig.2.5: singly linked list node structure

The last node of the linked list contains NULL because it does not refer to any other node as shown in Fig.2.6.

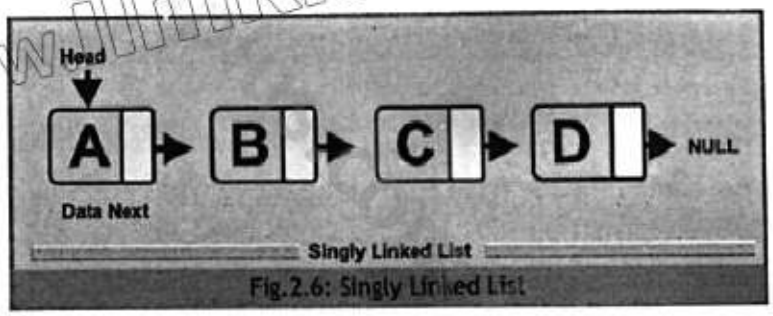


Fig.2.6: Singly Linked List

Doubly Linked List

The doubly linked list is complex as compared to singly linked list but it also provides more advantages. The major advantage is that it allows traversal in both directions and for that purpose it needs to maintain pointers for both directions e.g. reference to previous node and reference to next node. Therefore, the node in doubly linked list have three parts (Fig.2.7):

- reference to previous node in the sequence
- data to be stored
- reference to next node in the sequence

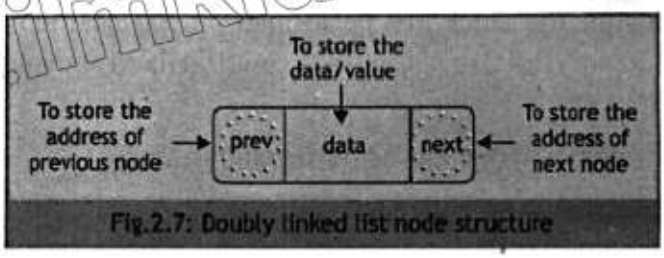


Fig.2.7: Doubly linked list node structure

Fig.2.8 shows an example scenario:

Fig.2.8: Doubly linked list

The first node and the last node in the sequence of linked list contains NULL because it does not refer to any other nodes.

Circular Linked List

It is a special type of linked list where all nodes are connected in a circle - forming a loop. Unlike other types, where last node points to NULL, the last node in circular linked linked is connected back to the first node. Therefore, while traversing in circular linked linked, it never reached to NULL. The structure of Circular Link List is depicted in Fig.2.9.

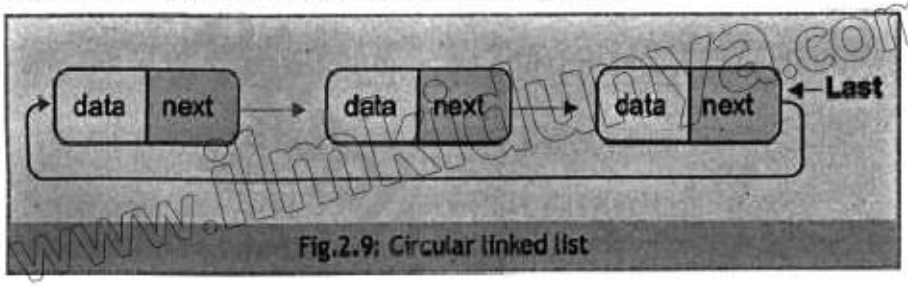


Fig.2.9: Circular linked list

The circular linked list could be made from both of the above types e.g. Singly linked list or doubly linked list. Fig.2.10 shows formation of a circular doubly linked list.

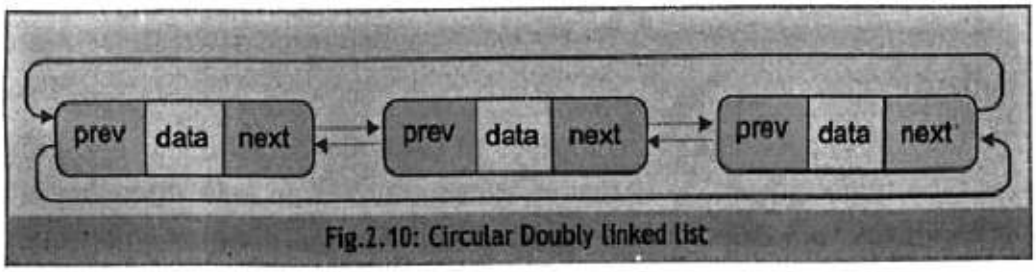


Fig.2.10: Circular Doubly linked list

2.1.3. Stacks

The stacks use the LIFO (Last-In, First-Out) principle, where the most recently added element (last element in the atack) is the first to be removed. This is important for understanding algorithms that need to manage tasks in a specific order. The LIFO principle makes it simple because the operations on the stack are performed on top position of the stack. Stack are considered memory efficient, the size of the stack could be fixed or dynamic, it depends upon the implementation.

Following are the basic operation that can be performed on stack:

- Push: Add a new element at the top of stack
- Pop: remove existing element from the top of stack

- Top: return the top element without removing it
- IsEmpty: Check, if stack is empty
- IsFull: Check, if stack is full

The working of Stack with core operations is shown Fig.2.11.

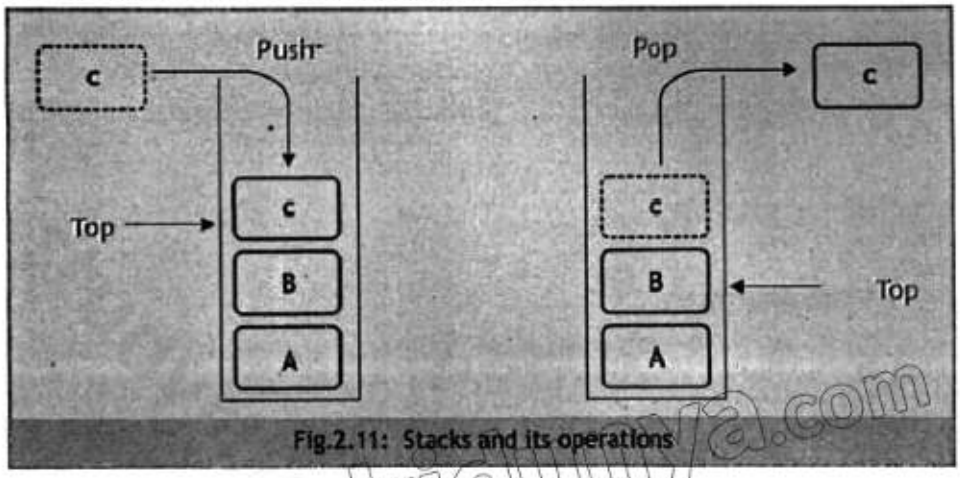


Fig.2.11: Stacks and its operations

2.1.4. Queues

The Queues use the FIFO (First-In, First-Out) principle, where the first element added is the first to be removed. This is useful for managing tasks that need to be processed in the order they arrive. Unlike stack, the operations on queues occur at two ends. The new elements are added at the end of queue, whereas the existing elements are removed from the start of the queue.

Following are the basic operation that can be performed on queues:

- Enqueue: Add a new element at the end(back/tail/rear) of the queue
- Dequeue: remove existing element at the start (front/head) of the queue.
- Front: return the start element without removing it
- IsEmpty: Check, if queue is empty
- IsFull: Check, if queue is full

The working of Queues with core operations is shown Fig.2.12.

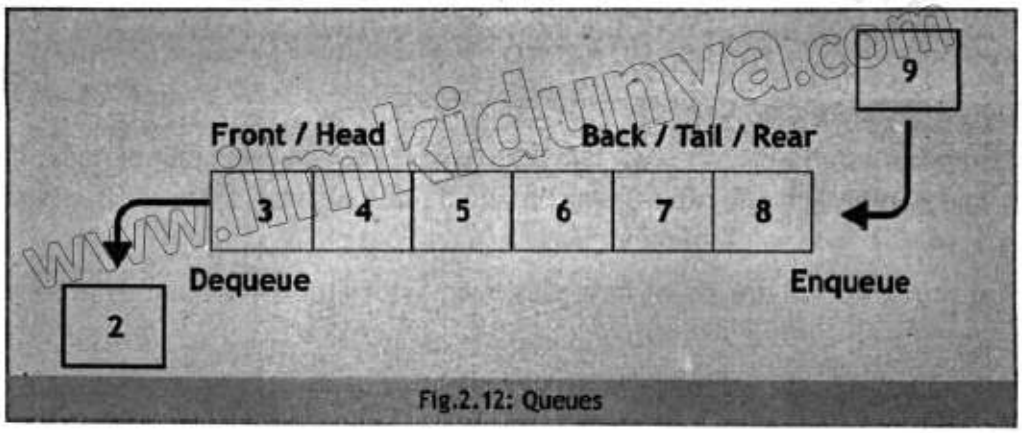


Fig.2.12: Queues

They are key consideres key data structure in algorithms like breadth-first search (BFS) and in various real-world scenarios like scheduling and buffering.

2.1.5. Graphs

The graph data structure also consist of nodes (called vertices) connected to each other by edges.

The graphs model complex relationships and networks. Graphs help in understanding problems related to social networks, routing and resource management. They are essential for algorithms related to network analysis, shortest paths (e.g., Dijkstra's algorithm) and connectivity as shown in Fig.2.13.

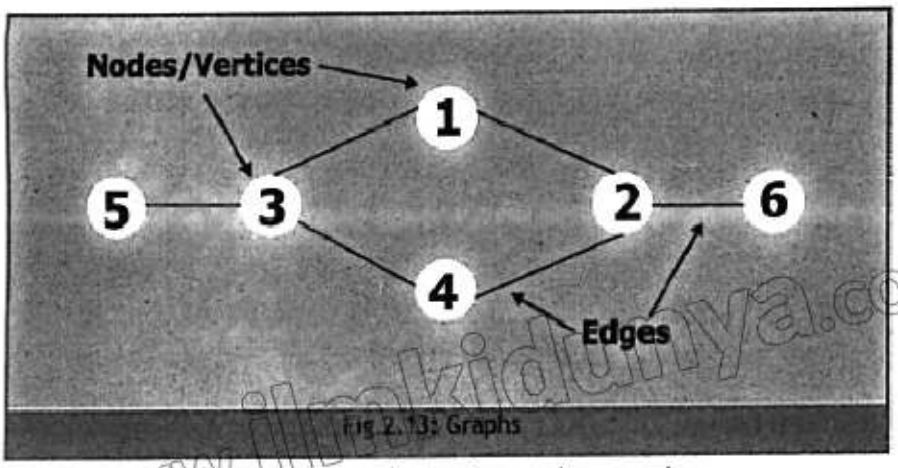


Fig.2.13: Graphs

Following are the basic operation that can be performed on graphs:

- Add Vertex: Add a new node to the graph
- Add Edge: Create connection between two nodes
- Remove Vertex/Edge: Delete node or edge from the graph
- Traversal: Visiting graph in specific order.

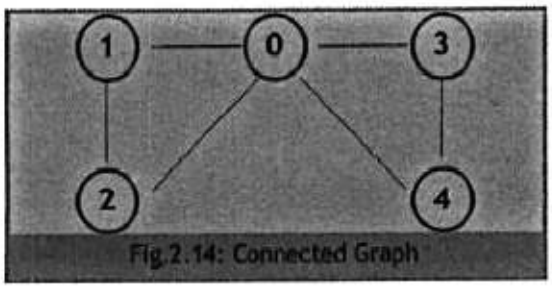


Fig.2.14: Connected Graph

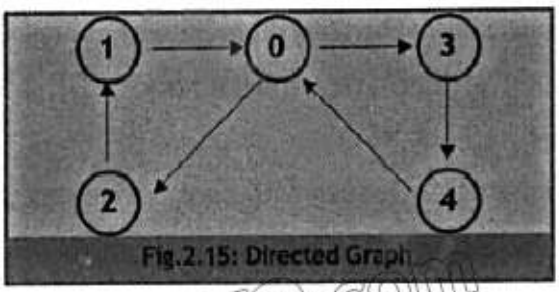


Fig.2.15: Directed Graph

Primarily, there are two types of graphs:

Connected Graph: A graph where at least one path exists between every pair of node. If you can start at any node and reach any other node (in any direction), it is connected. An example is shown in Fig.2.14.

Directed Graph: A graph where edges have direction associated with them as represented in Fig.2.15.

2.1.6 Trees

The trees represent hierarchical structures consisting of nodes. The top most node is called root. The directly connected nodes make a relationship of parent-child. The higher node in hierarchy is parent, whereas, lower is child. The node which do not have child is referred as leaf node. The relationship between nodes is referred as edge. Each node consist of two things: value and pointer to its child. An example is shown in Fig.2.16.

The trees are crucial for understanding complex relationships between data. Trees are used in various algorithms such as traversal (in-order, pre-order, post-order) and are foundational for understanding more advanced data structures.

In computer Science, Directory structure in an operating system or Tag structure in HTML Language are best examples to understand trees.

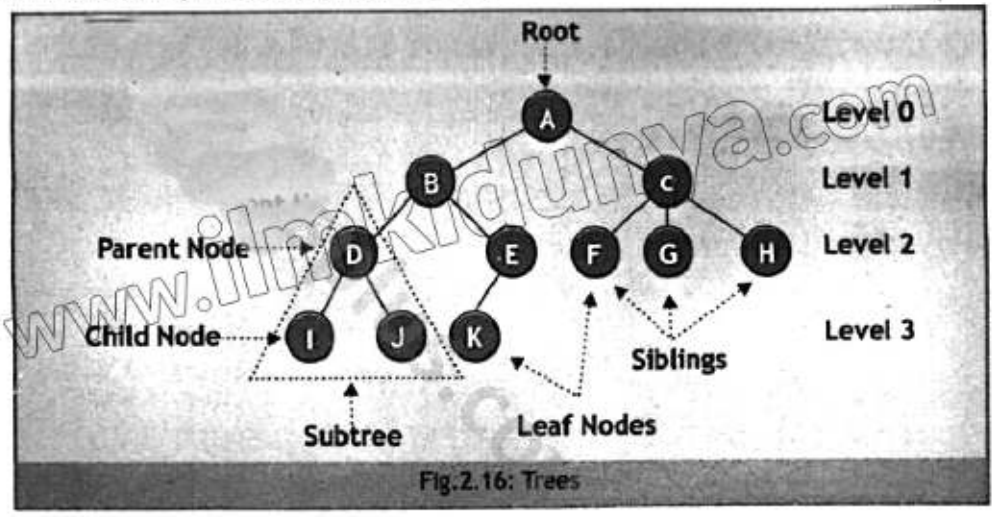


Fig.2.16: Trees

Following are the basic operation that can be performed on trees:

- Insertion: Add a new node at specified location in the tree
- Deletion: Remove existing node
- Search: Find specific node in the tree
- Traversal: Visiting tree in specific order

Tree traversal

A tree traversal is always done in a systematic way and it involves a mechanism to visit all the nodes. There are several methods for tree taversal and each have different purpose. Following is a brief overview of some common tree traversal methods:

- **In-Order Traversal (for Binary Trees) - see Fig.2.17:**
  1. Traverse the left subtree.
  2. Visit the root node.
  3. Traverse the right subtree.

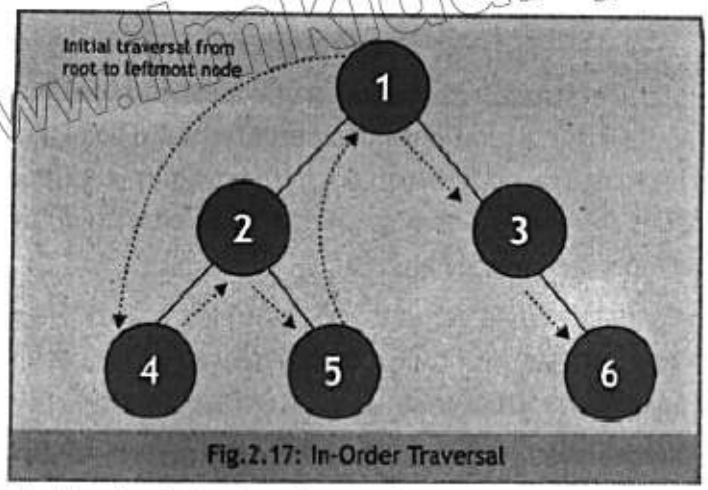


Fig.2.17: In-Order Traversal

So the order of traversal of nodes is 4 -> 2 -> 5 -> 1 -> 3 -> 6.

This traversal is often used with binary search trees (BSTs) because it visits nodes in ascending order.

- **Pre-Order Traversal - see Fig.2.18:**
  1. Visit the root node.
  2. Traverse the left subtree.
  3. Traverse the right subtree.

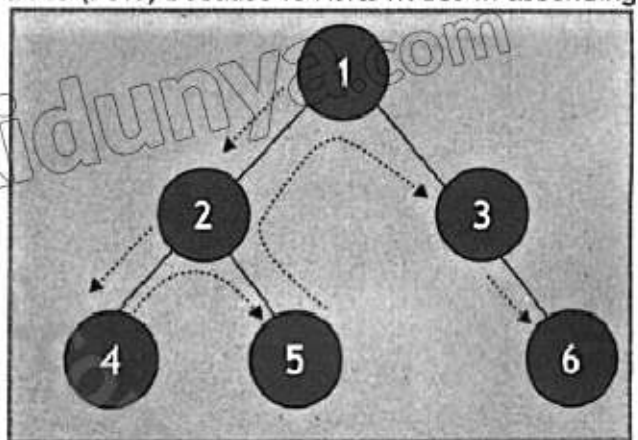


Fig.2.18: Pre-Order Traversal

So the order of traversal of nodes is 1 -> 2 -> 4 -> 5 -> 3 -> 6.

This is useful when we need to work with the root node before its children.

- **Post-Order Traversal - see Fig.2.19:**
  1. Traverse the left subtree.
  2. Traverse the right subtree.
  3. Visit the root node.

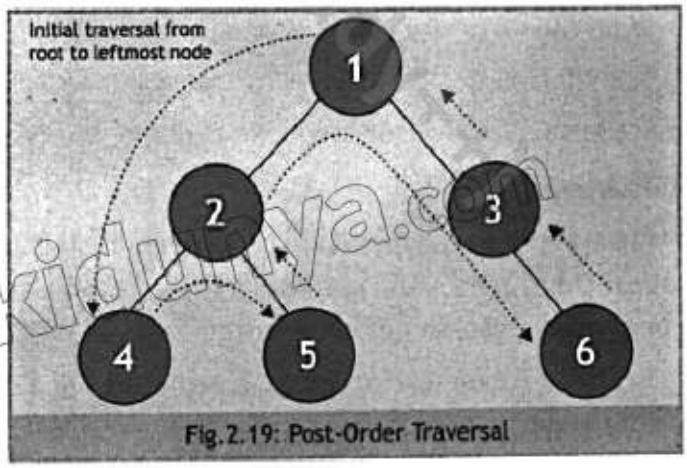


Fig.2.19: Post-Order Traversal

So the order of traversal of nodes is 4 -> 5 -> 2 -> 6 -> 3 -> 1.

This is often used for deleting nodes or freeing memory, as it processes children before their parent.

Example

Fig.2.20 shows examples of Inorder, Preorder and Postorder Traversal

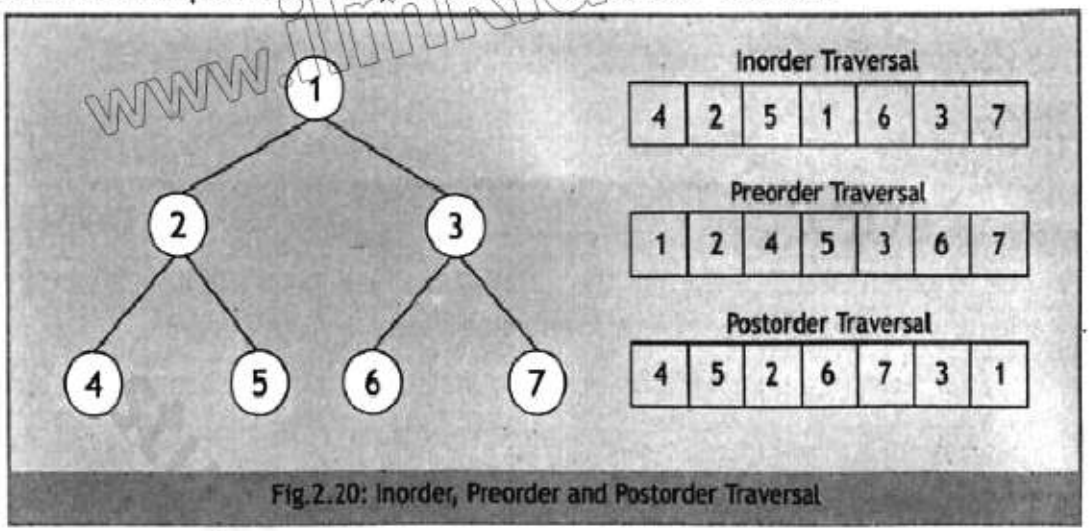


Fig.2.20: Inorder, Preorder and Postorder Traversal

Data Structures Support Computational Thinking:

- **Problem Decomposition:** Different data structures help break down problems into manageable components, making it easier to design and implement algorithms.
- **Efficiency Considerations:** They illustrate trade-offs between different operations (e.g., access time vs. insertion time), helping in choosing the most appropriate structure for a given problem.
- **Algorithmic Thinking:** Understanding these structures aids in developing and analyzing algorithms, improving problem-solving skills by selecting the right tool for each task.

Overall, data structures are not just tools for storing data; they shape the way we think about solving problems and designing algorithms. Mastery of these concepts is crucial for effective computational thinking and developing efficient algorithms.

2.2 Evaluating Computational Solutions

When evaluating computational solutions, it is essential to consider the following criteria:

2.2.1 Correctness

An algorithm is considered correct if it generates the expected output for all valid inputs. To verify correctness, multiple test cases should be used that cover various scenarios, including edge cases.

Example 1: Sorting Algorithm using Bubble Sort Algorithm

Test Case: Given an unsorted array [5, 2, 9, 1, 5, 6], Bubble Sort should correctly sort to [1, 2, 5, 5, 6, 9].

If the algorithm consistently produces the sorted array for various test cases, including edge cases like an empty array or an array with one element, it can be considered correct.

Example 2: Search Algorithm using Binary Search Algorithm

Case: Given a sorted array [1, 2, 3, 4, 5] and a search key 3, Binary Search should return the index 2 (assuming zero-based indexing).

To verify correctness, ensure Binary Search works for various search keys (existing and non-existing) and for arrays of different sizes.

2.2.2 Clarity

An algorithm is clear if its steps are logically structured and easily understandable. For this purpose, multiple things should be taken care of, for example using descriptive names for variables and functions and ensuring that the steps are ordered logically. Similarly, documentation and comments in code can also enhance clarity.

Example: Consider two python implementations of a function that calculates the factorial of a number.

Case 1: Clear Code

```python
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Case 2: Less Clear Code

```python
def fct(n):
    return 1 if n == 0 else n * fct(n - 1)
```

The first case is easily readable and understandable because the function name (factorial) represents what logic inside the function will be doing. Similarly, the conditional statement (if n == 0) is also very clear that what it means in the code. However, the second case uses a shorter function name and a ternary operator, which, while valid, may be less immediately understandable to someone new to the code.

2.2.3 Efficiency

Efficiency refers to how well an algorithm uses computer resources, such as time and memory. It is