MODULE 1: INTRODUCTION TO DATA STRUCTURES

What is a Data Structure?

A data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently. For example, suppose you are building a contact book. If you just store names in a random way, it will take more time to search someone. But if you organize them properly (say in alphabetical order), you can search faster. That's what data structures help with — efficient storing and accessing of data.

Data structures are important in programming because they allow us to perform operations like:

- Searching quickly (e.g., Google search),
- Sorting items (e.g., product price low to high),
- Inserting or deleting data without breaking things,
- Managing memory usage properly.

Types of Data Structures

1. Primitive Data Structures

These are the basic types of data that are already provided by the programming language. Examples include:

- Integer: To store numbers like 1, 2, 3
- Float: To store decimal numbers like 3.14
- Character: To store single letters like 'A', 'b'
- Pointer: Stores memory address

2. Non-Primitive Data Structures

These are more complex and can hold a collection of values. These include:

- **Linear Structures**: Data is arranged in a line, one after the other.
- **Non-Linear Structures**: Data is arranged like a tree or a graph — not in one straight line.

Linear Data Structures

These store data in a sequence. Each element has a next and maybe a previous one. You can move step-by-step through the data.

Examples:

- **Arrays**
- **Stacks**
- **Queues**
- **Linked Lists** (although often discussed in a separate module)

Operations like insert, delete, and search are easy to understand in these structures because of their order.

Non-Linear Data Structures

In these, the data is not arranged in a straight line. One element can be connected to multiple elements.

Examples:

- **Trees**
- **Graphs**

These are useful for representing hierarchical relationships (e.g., a family tree, folders in a computer) or network-based data (e.g., social networks, maps).

Arrays

An array is a collection of similar data items stored in **contiguous (side-by-side)** memory locations.

For example, if you want to store 5 student marks, you can do it like this:

```
c
Copy code
int marks[5] = {80, 90, 75, 60, 85};
```

Here, all 5 values are integers, and they are stored next to each other in memory. You can access any mark using an index, e.g., `marks[2]` will give 75.

Operations on Arrays:

1. **Traversal** – Going through all elements one by one using a loop.
2. **Insertion** – Adding a new element at a specific position. To do this, you might have to shift other elements.

3. **Deletion** – Removing an element. Again, you may need to shift elements to fill the gap.
4. **Searching** – Checking if a value exists and finding its position.
5. **Sorting** – Rearranging elements in order (like ascending or descending).

Arrays are **static**, meaning their size is fixed once declared. If you want to store more data than planned, you'll run into a problem.

Stacks

A **stack** is a linear data structure where the last item added is the first to be removed. This is called **LIFO – Last In, First Out**.

Think of a stack of plates. You add plates to the top and remove from the top only.

Basic Operations:

- **Push**: Add an element to the top.
- **Pop**: Remove the element from the top.
- **Peek**: Look at the top element without removing it.

Stack Overflow: Happens when the stack is full and you try to push a new element.

Stack Underflow: Happens when the stack is empty and you try to pop an element.

Applications of Stack:

- Reversing a string
- Checking balanced brackets (like `{[()]}`)
- Converting expressions (infix to postfix/prefix)
- Evaluating postfix expressions

Queues

A **queue** is another linear structure but works on **FIFO – First In, First Out**.

Imagine a queue at a movie ticket counter — the first person in line gets the ticket first.

Operations:

- **Enqueue**: Add an element to the rear (end).
- **Dequeue**: Remove an element from the front.

Types of Queues:

1. **Simple Queue**: Basic queue – insert at rear, delete from front.
2. **Circular Queue**: Rear wraps around to front when it reaches the end.
3. **Deque (Double Ended Queue)**: You can insert or delete from both ends.
4. **Priority Queue**: Elements are processed based on priority, not position.

Expression Conversion and Evaluation

We often write expressions like: `a + b * c`

This is called **infix notation**, where the operator is in between the operands.

Other notations:

- **Postfix**: `abc*+` → Operator comes after operands
- **Prefix**: `+a*bc` → Operator comes before operands

Why convert? Because postfix/prefix expressions don't need brackets and are easier for computers to evaluate using stacks.

Infix to Postfix Conversion Algorithm (using Stack):

1. Add ')' at the end and push '(' to the stack at the beginning.
2. Read the expression left to right.
3. If it's an operand, add it to the result.
4. If it's '(', push to stack.
5. If it's ')', pop until you find '('.
6. If it's an operator, pop operators from stack that have higher or equal priority, then push the new operator.
7. Repeat until end of expression.

Postfix Evaluation:

- Read left to right.
- If operand → push to stack.
- If operator → pop 2 elements, apply operation, push result.
- At the end, the result is on top of the stack.

MODULE 2: LINKED LISTS

What is a Linked List?

A **linked list** is a type of linear data structure, like an array, but with a major difference: elements in a linked list are **not stored in consecutive memory locations**. Instead, each element (called a **node**) contains:

1. **Data** (the actual value)
2. **Link/Pointer** (which points to the next node)

In other words, each node knows where the next node is, and they're "linked" together — that's why it's called a linked list.

Unlike arrays, linked lists are **dynamic** in size — you can add or remove elements as needed during program execution. This makes them flexible and memory-efficient when the number of elements is not known in advance.

Structure of a Node

In C programming, a node in a linked list is usually defined like this:

```
c
Copy code
struct node {
    int data;
    struct node* next;
};
```

- `data` stores the actual information.
- `next` is a pointer to the next node in the list.

We keep track of the entire list using a pointer called `start` or `head`, which stores the address of the first node. If `start` is `NULL`, it means the list is empty.

Advantages of Linked Lists Over Arrays

Arrays	Linked Lists
Fixed size (static)	Can grow/shrink dynamically

Arrays	Linked Lists
Elements stored continuously	Elements stored anywhere in memory
Insert/delete is costly	Insert/delete is easier
Fast random access (indexing)	No direct access, only sequential

So, if you often need to insert/delete elements, **linked list is better**. But for quick access using an index, **arrays are faster**.

Types of Linked Lists

1. Singly Linked List

Each node has one pointer — to the next node. The last node points to `NULL`.

Example:
`10 → 20 → 30 → NULL`

You can only move **forward**, not backward.

---

2. Doubly Linked List

Each node has two pointers:

- One to the **next** node
- One to the **previous** node

So you can move in both directions: forward and backward.

Example:
`NULL ← 10 ⇄ 20 ⇄ 30 → NULL`

Useful when you want **bidirectional traversal**.

---

3. Circular Linked List

Here, the **last node** points back to the **first node**, forming a circle.

Example (Singly circular):
`10 → 20 → 30 → 10 → ...`

This allows looping through the list endlessly.

Operations on Singly Linked List

We can perform the following operations:

1. Traversal

Visiting each node, one by one, from start to end.

Steps:

- Set a pointer `ptr` to the start of the list.
- While `ptr` is not `NULL`, print `ptr->data` and move `ptr` to `ptr->next`.

Purpose: Used to display or process the elements in the list.

---

2. Insertion

You can insert a new node:

- At the **beginning**
- At the **end**
- **After** a given node

a. Insert at Beginning

- Create a new node.
- Make its `next` point to the current `start`.
- Update `start` to point to this new node.

b. Insert at End

- Create a new node with `next = NULL`.
- Traverse to the last node.
- Set last node's `next` to point to the new node.

c. Insert After a Given Node

- Find the node after which you want to insert.
- Create a new node.
- Set new node's `next` to that node's `next`.
- Set that node's `next` to the new node.

---

3. Deletion

You can delete:

- The **first** node
- The **last** node
- A **specific node** (based on value)

a. Delete First Node

- Just set `start = start->next`.
- Free memory of the old first node.

b. Delete Last Node

- Traverse to the second-last node.
- Set its `next` to `NULL`.
- Free the last node.

c. Delete Specific Node

- Traverse and keep track of current and previous nodes.
- If node found:
    - Set `prev->next = current->next`.
    - Free `current`.

---

4. Searching in Linked List

- Start from the `start` node.
- Compare each node's data with the value.
- If found, print its position.
- Else, say it's not found.

Note: As there's no index, we must check each node one by one.

Memory Allocation in Linked List

In C, we create new nodes **dynamically** using memory allocation functions from `<stdlib.h>`.

`malloc()`

Allocates memory but does not initialize it.

```
c
Copy code
struct node* ptr = (struct node*)malloc(sizeof(struct node));
```

`calloc()`

Allocates and **initializes** memory with zeros.

```
c
Copy code
int* a = (int*)calloc(10, sizeof(int));
```

`free()`

Releases memory back to the system.

```
c
Copy code
free(ptr);
```

`realloc()`

Changes the size of previously allocated memory.

```
c
Copy code
a = (int*)realloc(a, newSize);
```

Real-Life Applications of Linked Lists

- Music playlist apps (where you can move next/previous)
- Browsers (forward/backward history navigation)
- Dynamic memory allocation (OS level)
- Implementation of stacks and queues

MODULE 3: TREES

1. What is a Tree?

A **tree** is a **non-linear** data structure that is used to represent **hierarchical relationships** among elements. Unlike arrays or linked lists (which are linear), trees have a branching structure like a real tree (inverted).

A tree is made up of **nodes** connected by **edges**. The topmost node is called the **root**, and each node can have **zero or more child nodes**. Trees are very useful in situations like:

- Storing folder structures on computers,
- Representing family trees,
- Searching and sorting large datasets quickly.

---

2. Basic Terminologies in Trees

a. Node

Each element in a tree is called a **node**. It contains:

- A data part (the value),
- Zero or more pointers/links to its child nodes.

b. Root

The **first node** or topmost node of a tree. It has no parent.

c. Child

A node that comes directly under another node is called a child. For example, if B is under A, then B is a child of A.

d. Parent

If a node has a child, it is called the **parent** of that child.

e. Leaf

A node that has **no children** is called a **leaf node** or **terminal node**.

f. Siblings

Nodes that share the **same parent** are called **siblings**.

g. Level

Level tells us how deep a node is in the tree. The root is at **level 0**, its children are at **level 1**, and so on.

h. Height of Node

It is the number of edges on the **longest path** from that node to a leaf.

i. Height of Tree

The height of a tree is the height of the **root** node, i.e., the longest path from the root to any leaf node.

j. Degree of Node

The number of children a node has is called its **degree**.

k. Degree of Tree

The highest degree of any node in the tree.

l. Ancestor

All the nodes that are above a particular node along the path to the root.

m. Path

A sequence of nodes and edges connecting a node with a descendant.

n. Path Length

The number of nodes or edges in a path.

---

3. Types of Trees

a. Binary Tree

A tree in which **each node has at most two children** — a **left child** and a **right child**. It is called a binary tree because each node can have only two branches.

b. Full Binary Tree

A binary tree where **every node has either 0 or 2 children**. No node has only one child.

c. Complete Binary Tree

A binary tree where **all levels are completely filled**, except possibly the last, and the nodes in the last level are filled **from left to right**.

d. Perfect Binary Tree

A binary tree in which **all internal nodes have two children** and **all leaf nodes are at the same level**.

---

4. Linked Representation of Binary Tree

Each node in a binary tree contains:

- Data
- Pointer to the left child
- Pointer to the right child

```
c
Copy code
struct node {
    int data;
    struct node* left;
    struct node* right;
};
```

We keep a global pointer called `root` to store the address of the first node (the root of the tree).

---

5. Tree Traversals

Traversal means visiting all the nodes in a tree in a specific order. There are three main types of traversal in binary trees:

a. Pre-order Traversal (NLR)

- Visit the node,
- Then the left subtree,
- Then the right subtree.

Example: For a tree with root A, left child B, and right child C, the pre-order is:
`A B C`

b. In-order Traversal (LNR)

- Traverse the left subtree,
- Visit the node,
- Then the right subtree.

This is very useful because in **Binary Search Trees (BSTs)**, **in-order traversal gives sorted output**.

c. Post-order Traversal (LRN)

- Traverse the left subtree,
- Then the right subtree,
- Finally visit the node.

Each traversal method is used in different situations based on application — for example, expression trees, file systems, etc.

---

6. Binary Search Tree (BST)

A **Binary Search Tree** is a special kind of binary tree where the values are arranged such that:

- All values in the **left subtree** of a node are **less** than the node's value.
- All values in the **right subtree** of a node are **greater or equal** to the node's value.

This rule applies to every node in the tree.

Why BST is Useful?

- Searching is very fast (O(log n) in average cases).
- In-order traversal gives sorted order.
- BST is widely used in search engines, dictionaries, database indexing, etc.

Operations in BST:

1. **Insertion**: Find the correct position and add the node without breaking the BST rule.
2. **Searching**: Check left or right subtree based on comparison.
3. **Deletion**:
    - Case 1: Node has no child (just delete it).
    - Case 2: Node has one child (replace it with the child).
    - Case 3: Node has two children (replace with inorder successor or predecessor).
4. **Traversal**: In-order, pre-order, post-order.
5. **Counting Nodes**: Count total number of nodes.
6. **Height**: Calculate depth of tree.

---

7. Expression Trees

An **expression tree** is a binary tree that represents an arithmetic expression. Operands (like numbers or variables) are stored in leaf nodes, and operators (+, –, *, /) are stored in internal nodes.

Example: Expression: `a + b * c`
Expression Tree:

```
css
Copy code
     +
    / \
   a   *
      / \
     b   c
```

Traversals on expression trees are used to convert between:

- Infix (human readable)
- Prefix (for evaluation)
- Postfix (for stack-based evaluation)

---

8. Threaded Binary Trees

In a normal binary tree, many pointer fields are `NULL`, especially in leaf nodes. Threaded Binary Trees are special trees where **these NULL pointers are replaced with links to in-order predecessors or successors**.

This helps to traverse the tree **without recursion or stack**, making traversal faster and memory-efficient.

Real-Life Applications of Trees

- **BST**: Fast searching, auto-suggestions.
- **Heaps**: Priority queues, scheduling.
- **Tries**: Dictionary words, autocomplete.
- **Expression Trees**: Compilers, calculators.
- **Decision Trees**: AI, Machine Learning.

MODULE 4: GRAPHS

1. What is a Graph?

A **graph** is a **non-linear data structure** made up of:

- **Vertices (or nodes)** — the data points.
- **Edges** — connections between these vertices.

Graphs are used to represent **relationships** or **connections** between elements. For example:

- A **map** can be a graph: cities are vertices, roads are edges.
- In a **social network**, people are vertices and their friendships are edges.

Graphs are **more flexible** than trees because any node can be connected to any other node.

---

2. Basic Terminologies in Graphs

a. Vertex

Each data point in the graph is called a vertex (or node). Denoted as A, B, C, etc.

b. Edge

An edge is a link or connection between two vertices.

c. Adjacent Vertices

Two vertices are said to be **adjacent** if there is an edge connecting them.

d. Self-loop

An edge that connects a vertex to **itself**.

e. Parallel Edges

Two or more edges that connect the same pair of vertices.

f. Degree of a Vertex

The **number of edges** connected to that vertex.

- For directed graphs:

    - **In-degree**: number of edges coming **into** the vertex.
    - **Out-degree**: number of edges going **out** from the vertex.
- For undirected graphs: just count the total edges connected.

g. Isolated Vertex

A vertex with **no connections** at all (degree 0).

h. Pendant Vertex

A vertex connected by **only one edge** (degree 1).

i. Path

A sequence of vertices connected by edges.

j. Cycle

A **path that starts and ends at the same vertex** without repeating edges.

k. Simple Path

A path where **all vertices (except start and end)** are different.

l. Subgraph

A graph made using a **subset of vertices and edges** from another graph.

---

3. Types of Graphs

a. Directed Graph (Digraph)

In this graph, edges have a **direction** (represented by arrows).

- If A → B exists, you can go from A to B, but not from B to A unless B → A also exists.

b. Undirected Graph

Edges do **not** have direction. A connection between A and B means you can travel both ways.

c. Simple Graph

- No self-loops.
- No multiple edges between same pair of nodes.

d. Complete Graph

A graph where **every pair of vertices is connected** by exactly one edge.

If there are `n` vertices, the number of edges is `nC2 = n(n-1)/2`.

e. Cyclic Graph

A graph that contains **at least one cycle**.

f. Acyclic Graph

A graph that **does not** contain any cycle.

g. Bipartite Graph

A graph where the set of vertices can be **divided into two groups**, such that:

- Edges only connect vertices from one group to the other.
- No edges exist between vertices in the same group.

h. Complete Bipartite Graph

Every vertex in one group is connected to **every vertex in the other group**.

i. Connected Graph

There is **at least one path** between **every pair of vertices**. You can travel between any two nodes.

j. Disconnected Graph

At least one pair of vertices **cannot be reached** from one another.

k. Regular Graph

All vertices have the **same number of edges**. If every node has degree `k`, it's called a **k-regular graph**.

---

4. Representation of Graphs

Graphs can be stored in a computer using:

a. Adjacency Matrix

- A 2D array where the row and column indices represent vertices.
- If an edge exists between vertex `i` and `j`, then `matrix[i][j] = 1`, else 0.
- For **undirected graphs**, the matrix is **symmetric**.
- For **directed graphs**, the matrix may not be symmetric.

Example (5 vertices):

```
mathematica
Copy code
   A B C D E
A  0 1 0 1 0
B  1 0 1 1 0
C  0 1 0 0 1
D  1 1 0 0 1
E  0 0 1 1 0
```

b. Adjacency List

- Each vertex stores a list of all the vertices it is connected to.
- Efficient for **sparse graphs** (with fewer edges).

Example:

```
mathematica
Copy code
A → B → D
B → A → C → D
C → B → E
D → A → B → E
E → C → D
```

c. Adjacency Set

- Similar to an adjacency list but uses a **set** instead of a list.
- Useful when you want **fast checking** if an edge exists between two nodes.

---

5. Graph Traversal Algorithms

To process a graph (like searching or checking connectivity), we need to **traverse** it — i.e., visit all the vertices. Two standard methods are:

A. Breadth-First Search (BFS)

Idea: Visit all **neighboring nodes** first (go wide), then go deeper.

Data structure used: **Queue**

Steps:

1. Choose a starting vertex, mark it as visited.
2. Add it to the queue.
3. While the queue is not empty:
    - Remove vertex from the front.
    - Visit all its unvisited neighbors.
    - Mark them as visited and add to the queue.

Result: Visits nodes level-by-level.

Example BFS Order: A → B → D → E → C → F → G

Applications:

- Finding the shortest path in unweighted graphs.
- Checking if a graph is connected.

B. Depth-First Search (DFS)

Idea: Visit a vertex and **go as deep as possible**, then backtrack.

Data structure used: **Stack** (can also be done using recursion)

Steps:

1. Choose a starting vertex, mark it as visited.
2. Push it to the stack.
3. While the stack is not empty:
    - Look at the top of the stack.
    - If it has unvisited neighbors, visit one and push it.
    - If not, pop the top.

Result: Visits nodes deeply before backtracking.

Example DFS Order: A → B → C → E → D → F → G

Applications:

- Solving puzzles like mazes.
- Detecting cycles.
- Topological sorting.

---

6. Warshall's Algorithm for Shortest Path

Purpose: To find the **shortest path between all pairs of vertices** in a **weighted directed graph**.

It uses **Dynamic Programming** to build a matrix that stores the shortest distances step-by-step.

Steps:

1. Create an initial distance matrix from the adjacency matrix.
2. Update the distance matrix by checking if a shorter path exists using an intermediate node `k`:

   ```
   lua
   Copy code
   distance[i][j] = min(distance[i][j], distance[i][k] + distance[k][j])
   ```

3. Repeat for all nodes as intermediate points.

Result: The matrix will show the shortest distances between all pairs of vertices.

Applications of Graphs

- **Social networks** (friend suggestions)
- **Google Maps** and GPS (shortest paths)
- **Airline routes**
- **Web page linking**
- **Network routing**
- **Game AI**