UNIT-IV

1. Trees

1.1 Basic Tree Concepts

A **tree** is a non-linear data structure consisting of nodes connected by edges. It is used to represent hierarchical relationships.

Key Terms:

1. **Node:** A single element of a tree.
2. **Root:** The topmost node of a tree.
3. **Child:** Nodes directly connected to another node are its children.
4. **Parent:** A node connected directly above a given node is its parent.
5. **Leaf:** A node with no children.
6. **Height:** The number of edges from the root to the deepest leaf.
7. **Subtree:** A smaller tree derived from a parent node.

Characteristics:

- A tree has one root node.
- Every node has at most one parent.
- Acyclic: No loops or cycles exist in a tree.

1.2 Representation of Binary Tree in Memory

A **binary tree** is a tree where each node has at most two children: left and right. It can be represented in memory in two ways:

1.2.1 Array Representation

Each node is stored in an array, and the indices represent the node's position:

- Root is at index `0`.
- Left child of node at index `i` is at `2i + 1`.
- Right child of node at index `i` is at `2i + 2`.

Example: A binary tree with elements `[A, B, C, D, E]`:

```
        A
       / \
      B   C
     / \
    D   E
```

Array Representation: `[A, B, C, D, E]`

1.2.2 Linked Representation

Each node is represented as a structure with three fields:

1. Data (value of the node).
2. Pointer to the left child.
3. Pointer to the right child.

Example Code (C-like):

```c
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};
```

1.3 Binary Tree Traversals

Traversal is the process of visiting all nodes in a tree. There are three main types of binary tree traversal:

1.3.1 Inorder Traversal (Left, Root, Right)

Visit the left subtree, then the root, and finally the right subtree. **Example:** For the tree:

```
    A
   / \
  B   C
```

Traversal: `B, A, C`

Code (C-like):

```c
void inorder(Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}
```

1.3.2 Preorder Traversal (Root, Left, Right)

Visit the root first, then the left subtree, and finally the right subtree. **Example:** `A, B, C`

1.3.3 Postorder Traversal (Left, Right, Root)

Visit the left subtree, then the right subtree, and finally the root. **Example:** `B, C, A`

1.4 Binary Search Tree (BST)

A **Binary Search Tree** is a binary tree where:

1. The left subtree contains nodes with values smaller than the root.
2. The right subtree contains nodes with values larger than the root.

Operations:

- **Insertion:** Insert an element based on its value.
- **Search:** Search for an element by traversing left or right.
- **Deletion:** Remove a node and adjust the tree to maintain BST properties.

1.5 Heapsort

Heap: A binary tree that satisfies the **heap property**:

- **Max-Heap:** Parent node is greater than or equal to its children.
- **Min-Heap:** Parent node is smaller than or equal to its children.

Heapsort Algorithm:

1. Build a max-heap.

2. Swap the root with the last node.
3. Reduce the size of the heap and heapify.
4. Repeat until the heap size is 1.

---

2. Graphs

2.1 Basic Concepts

A **graph** is a collection of nodes (**vertices**) connected by edges. It is used to represent relationships or networks.

Types of Graphs:

1. **Directed Graph:** Edges have a direction.
2. **Undirected Graph:** Edges do not have a direction.
3. **Weighted Graph:** Edges have associated weights.

2.2 Representations of Graphs

2.2.1 Sequential Representation (Adjacency Matrix)

A 2D array where:

- Rows and columns represent vertices.
- An entry at `[i][j]` is `1` if there is an edge between vertex `i` and `j`.

2.2.2 Linked Representation (Adjacency List)

Each vertex points to a list of adjacent vertices.

Example: Graph:

```
A -- B
|    |
C -- D
```

Adjacency List:

```
A -> B, C
B -> A, D
C -> A, D
D -> B, C
```

2.3 Warshall's Algorithm

Warshall's algorithm finds the **transitive closure** of a graph (i.e., determines whether a path exists between any two vertices).

Steps:

1. Initialize a matrix `T` such that `T[i][j]` is `1` if there's a direct edge from `i` to `j`.
2. For each vertex `k`, update `T[i][j] = T[i][j] || (T[i][k] && T[k][j])`.

2.4 Operations on Graphs

1. **Insertion:** Add a vertex or an edge.
2. **Deletion:** Remove a vertex or an edge.
3. **Traversal:** Visit all vertices in the graph.

2.5 Traversing Graphs

2.5.1 Depth-First Search (DFS)

Start at a vertex and explore as far as possible along each branch before backtracking.

2.5.2 Breadth-First Search (BFS)

Start at a vertex and explore all its neighbors before moving to the next level.
