**Task 1:**

**Create a node for a tree and include a constructor.**

// TreeNode.java

public class TreeNode {

int data;

TreeNode left;

TreeNode right;

// Constructor

public TreeNode(int value) {

this.data = value;

this.left = null;

this.right = null;

}

}

Task 2:

Create a class named Binarty Search tree in which you have 2 insert operations

1 insert —----> for inserting if the tree is empty

1 insert —----> for inserting if the tree has 1 or more nodes

By 11.55 to 12.13

Task 3:

Ionorder travel of the above code snippets from task 1 and Task 2

12.14 to 12.19

Task 4:

Create a main method Task 1, 2 and 3

**Task 1– 4**

**class TreeNode {**

**int value;**

**TreeNode left, right;**

**TreeNode(int item) {**

**value = item;**

**left = right = null;**

**}**

**}**

**class BinarySearchTreeOp {**

**TreeNode root;**

**// Insert value into BST**

**void insert(int value) {**

**root = insertVal(root, value);**

**}**

**TreeNode insertVal(TreeNode node, int value) {**

**if (node == null) {**

**return new TreeNode(value);**

**}**

**if (value < node.value) {**

**node.left = insertVal(node.left, value);**

**} else if (value > node.value) {**

**node.right = insertVal(node.right, value);**

**}**

**return node;**

**}**

**// Inorder Traversal (Left, Root, Right)**

**void inorder() {**

**System.*out*.print("Inorder traversal: ");**

**inorderVal(root);**

**System.*out*.println();**

**}**

**void inorderVal(TreeNode node) {**

**if (node != null) {**

**inorderVal(node.left);**

**System.*out*.print(node.value + " ");**

**inorderVal(node.right);**

**}**

**}**

**// Preorder Traversal (Root, Left, Right)**

**void preorder() {**

**System.*out*.print("Preorder traversal: ");**

**preorderVal(root);**

**System.*out*.println();**

**}**

**void preorderVal(TreeNode node) {**

**if (node != null) {**

**System.*out*.print(node.value + " ");**

**preorderVal(node.left);**

**preorderVal(node.right);**

**}**

**}**

**// Postorder Traversal (Left, Right, Root)**

**void postorder() {**

**System.*out*.print("Postorder traversal: ");**

**postorderVal(root);**

**System.*out*.println();**

**}**

**void postorderVal(TreeNode node) {**

**if (node != null) {**

**postorderVal(node.left);**

**postorderVal(node.right);**

**System.*out*.print(node.value + " ");**

**}**

**}**

**}**

**public class BinarySearchTree {**

**public static void main(String[] args) {**

**BinarySearchTreeOp bstobj = new BinarySearchTreeOp();**

**// Inserting values**

**bstobj.insert(10);**

**bstobj.insert(50);**

**bstobj.insert(40);**

**bstobj.insert(70);**

**bstobj.insert(5);**

**// Traversals**

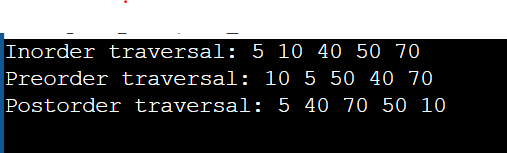
**bstobj.inorder();**

**bstobj.preorder();**

**bstobj.postorder();**

**}**

**}**

****

**class TreeNode {**

**int value;**

**TreeNode left, right;**

**TreeNode(int item) {**

**value = item;**

**left = right = null;**

**}**

**}**

**class BinarySearchTreeOp {**

**TreeNode root;**

**void insert(int value) { // 10**

**root = insertVal(root, value); //root = null**

**}**

**TreeNode insertVal(TreeNode node, int value) { // null, 10 //**

**if (node == null) {**

**node = new TreeNode(value);**

**return node;**

**}**

**if (value < node.value) {**

**node.left = insertVal(node.left, value);**

**} else if (value > node.value) {**

**node.right = insertVal(node.right, value);**

**}**

**return node;**

**}**

**void inorder() {**

**inorderVal(root);**

**}**

**void inorderVal(TreeNode node) {**

**if (node != null) {**

**inorderVal(node.left);**

**System.*out*.println("Node: " + node.value +**

**" | Left: " + (node.left != null ? node.left.value : "null") +**

**" | Right: " + (node.right != null ? node.right.value : "null"));**

**inorderVal(node.right);**

**}**

**}**

**}**

**public class BinarySearchTree2 {**

**public static void main(String[] args) {**

**BinarySearchTreeOp bstobj = new BinarySearchTreeOp();**

**bstobj.insert(50);**

**bstobj.insert(10);**

**bstobj.insert(400);**

**bstobj.insert(70);**

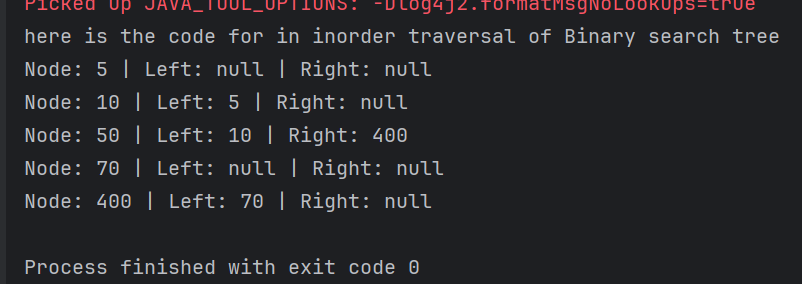
**bstobj.insert(5);**

**System.*out*.println("here is the code for in inorder traversal of Binary search tree ");**

**bstobj.inorder();**

**}**

**}**

****

**Task 5:**

**Applications of Trees**

Here are the key applications of Trees in computer science and real-world scenarios:

1. File System Organization

- Directory structures in operating systems

- Folder hierarchies on computers

- File explorers in operating systems

2. Database Systems

- B-Trees and B+ Trees for indexing

- Binary Search Trees for efficient data retrieval

- Decision trees for data classification

3. HTML DOM (Document Object Model)

- Web browsers use tree structures to represent HTML pages

- CSS selectors navigate this tree structure

- XML parsing

4. Artificial Intelligence

- Decision Trees for prediction models

- Game Trees for strategic planning (like chess)

- Mini-max trees in game theory

5. Network Routing

- Network broadcast algorithms

- Spanning trees in network topology

- Routing tables in networking

6. Compiler Design

- Expression trees

- Abstract Syntax Trees (AST)

- Parse trees for programming languages

7. Organization Charts

- Company hierarchies

- Family trees

- Organizational structures

8. Computer Graphics

- Scene graphs

- Spatial partitioning (BSP trees, Quad trees)

- Animation hierarchies

9. Search Algorithms

- Binary Search Trees for efficient searching

- Trie structures for string searching

- Auto-complete features

10. Mathematics

- Expression evaluation

- Mathematical formula representation

- Probability trees

11. Machine Learning

- Random Forest algorithms

- Decision tree learning

- Hierarchical clustering

12. Operating Systems

- Process trees (parent-child relationships)

- Memory allocation (buddy system)

- File system implementation

These applications demonstrate why trees are one of the most important and widely used data structures in computer science and software development.

**Task 6:**

**Create a binary search operation on tree**

**class TreeNode2 {**

**int value;**

**TreeNode2 left, right;**

**TreeNode2(int value) {**

**this.value = value;**

**left = right = null;**

**}**

**}**

**class BinarySearchTreeOperations {**

**TreeNode2 root;**

**int i = 0;**

**// Constructor**

**public BinarySearchTreeOperations() {**

**this.root = null;**

**}**

**// Insert operations**

**void insert(int value) {**

**root = insertVal(root, value);**

**}**

**TreeNode2 insertVal(TreeNode2 node, int value) {**

**if (node == null) {**

**node = new TreeNode2(value);**

**return node;**

**}**

**if (value < node.value) {**

**node.left = insertVal(node.left, value);**

**} else if (value > node.value) {**

**node.right = insertVal(node.right, value);**

**}**

**return node;**

**}**

**// Search operation**

**public TreeNode2 search(int key) {**

**TreeNode2 current = root;**

**while (current != null) {**

**if (key == current.value) {**

**return current;**

**} else if (key < current.value) {**

**current = current.left;**

**} else {**

**current = current.right;**

**}**

**}**

**return null;**

**}**

**// Traversal operations**

**void inorder() {**

**inorderVal(root);**

**}**

**void inorderVal(TreeNode2 node) {**

**if (node != null) {**

**inorderVal(node.left);**

**System.*out*.println(node.left + " " + node.value + " " + node.right + " ===> " + i++);**

**inorderVal(node.right);**

**}**

**}**

**}**

**public class task6 {**

**public static void main(String[] args) {**

**BinarySearchTreeOperations bst = new BinarySearchTreeOperations();**

**// Insert operations**

**bst.insert(10);**

**bst.insert(50);**

**bst.insert(400);**

**bst.insert(70);**

**bst.insert(5);**

**// Inorder traversal**

**System.*out*.println("Inorder traversal of Binary Search Tree:");**

**bst.inorder();**

**// Search operations**

**int searchKey = 70;**

**TreeNode2 result = bst.search(searchKey);**

**if (result != null) {**

**System.*out*.println("\nFound " + searchKey + " in the tree");**

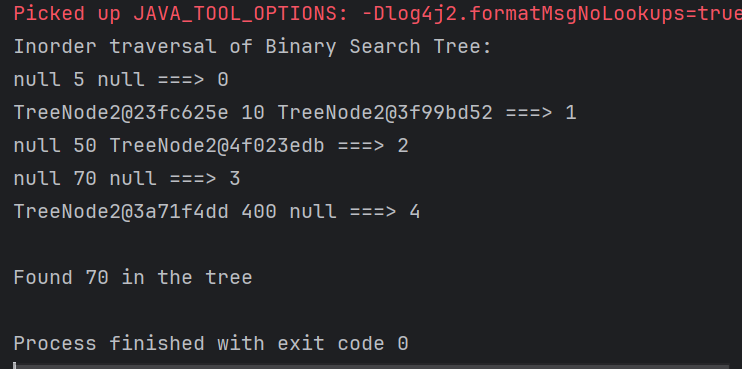
**} else {**

**System.*out*.println("\n" + searchKey + " not found in the tree");**

**}**

**}**

**}**

****

**Task 7:**

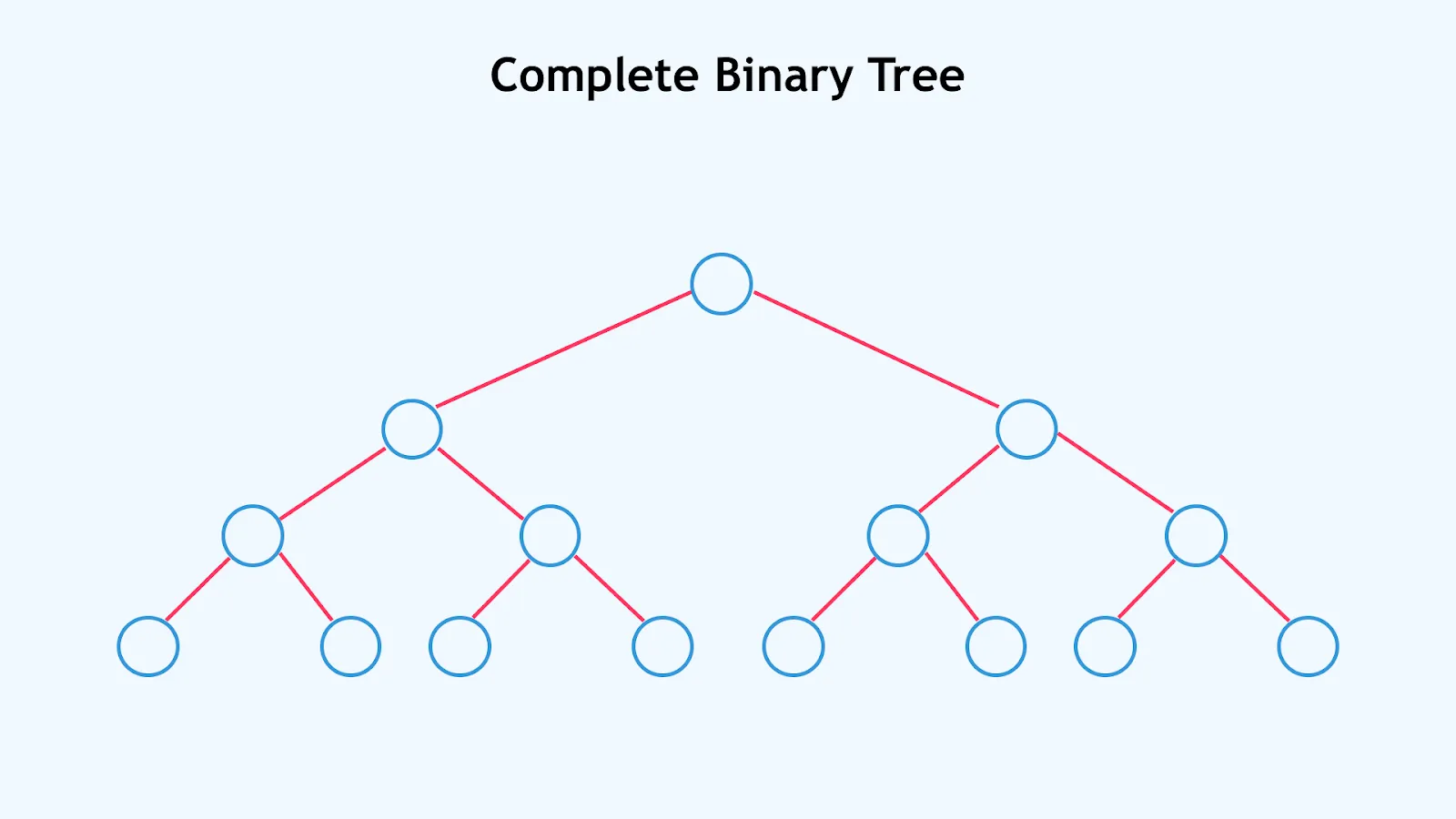
**Types of binary trees:**

**Full Binary Tree (or Strict/Proper Binary Tree):**

In a full binary tree, every node has either zero or two children. No node has only one child.

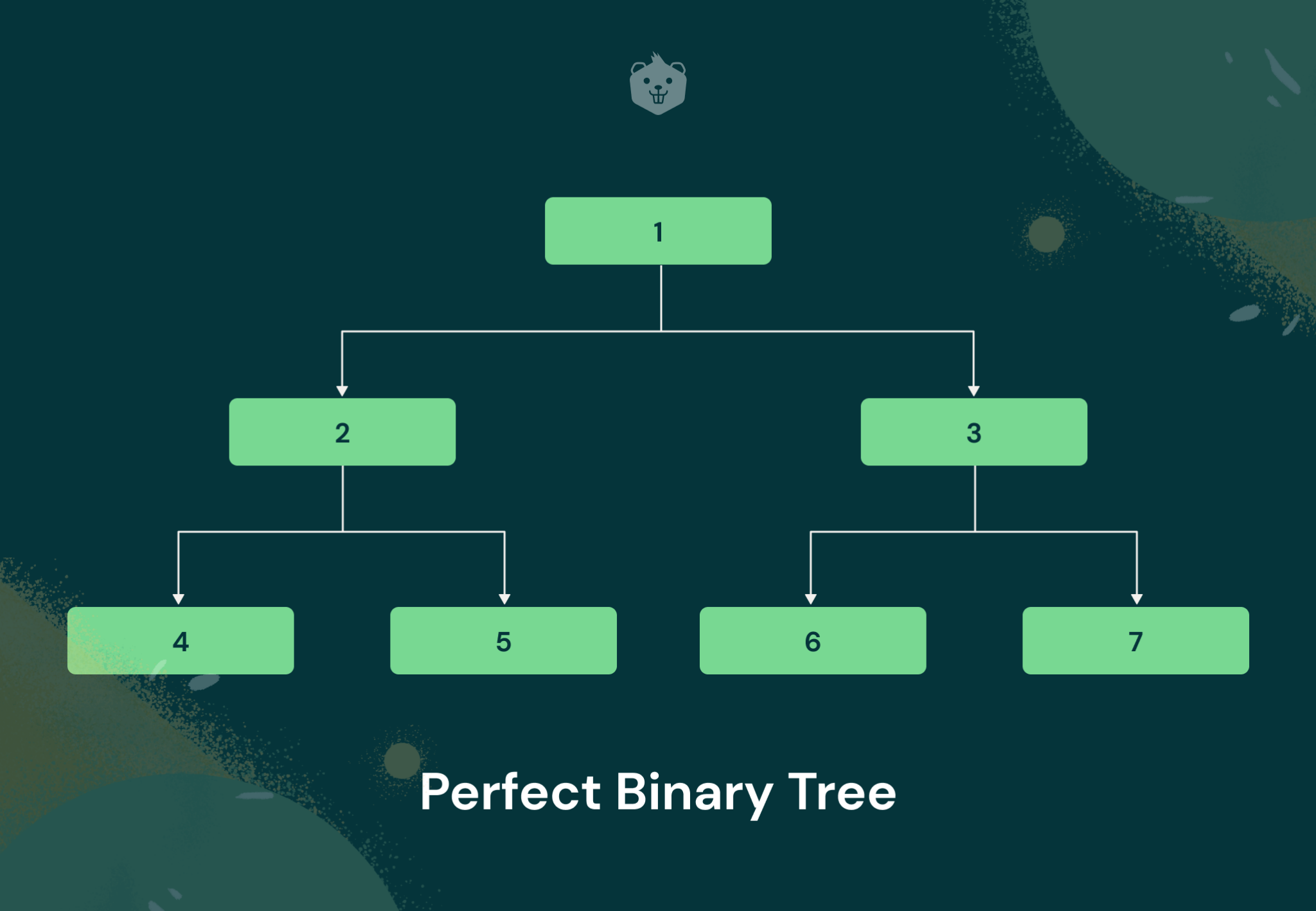
**Complete Binary Tree:**

A complete binary tree is a binary tree where all levels are completely filled except possibly the last level, and in the last level, all nodes are as far left as possible.



**Perfect Binary Tree:**

A perfect binary tree is a special case that is both a full and a complete binary tree. All internal nodes have two children, and all leaf nodes are at the same level (depth).



**Degenerate (or Pathological) Binary Tree:**

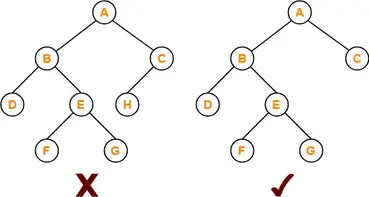
In a degenerate binary tree, each parent node has only one child node. This results in a tree that resembles a linked list, either left-skewed (all nodes lean left) or right-skewed (all nodes lean right).

**Balanced Binary Tree:**

A balanced binary tree is one where the heights of the left and right subtrees of any node differ by at most one. This property helps maintain efficient search, insertion, and deletion operations. Examples include AVL trees and Red-Black trees.

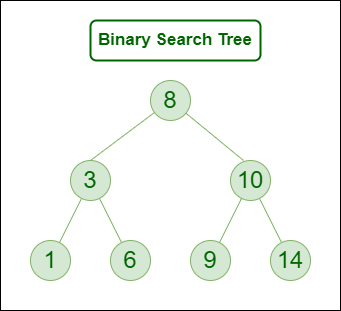
**Skewed Binary Tree:**

This is a specific type of degenerate tree where all nodes are either on the left side (left-skewed) or the right side (right-skewed).



**Binary Search Tree (BST):**

While not a structural type in the same sense as the above, a BST is a widely used type of binary tree where nodes are ordered. For every node, all values in its left subtree are less than the node's value, and all values in its right subtree are greater than the node's value.



**Heap (Min-Heap/Max-Heap):**

Heaps are tree-based data structures that satisfy the heap property, which dictates a specific ordering between parent and child nodes. In a max-heap, the parent node's value is greater than or equal to its children's, while in a min-heap, it's less than or equal to its children's.

**Task 8:**

**Applications of Graphs**

Here are the key applications of Graphs in various fields:

1. Social Networks

* Representing friendships and connections
* Analyzing social relationships
* Influence mapping
* Community detection

1. Transportation Systems

* Road networks and navigation
  + Flight routes optimization
  + Public transit planning
  + Traffic flow analysis

1. Computer Networks

* Internet topology
* Network routing protocols
* Data center architecture
* Network security analysis

1. Biology and Chemistry

* Molecular structures
* Protein interaction networks
* Gene regulatory networks
* Ecosystem food webs

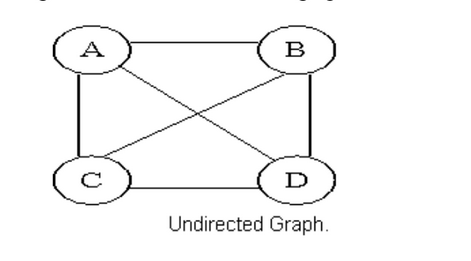
**Task 9:**

**Types of Graphs:**

Types of Graphs

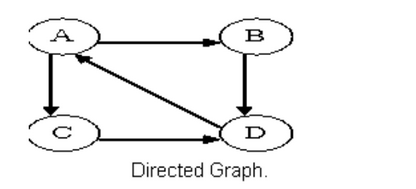
**1.Undirected Graph**

A graph with only undirected edges is said to be undirected graph.



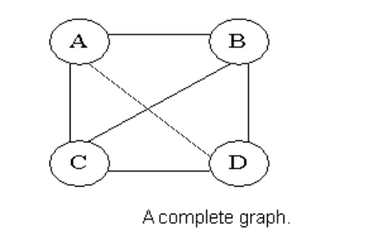
**2.Directed Graph**

A graph with only directed edges is said to be directed graph.



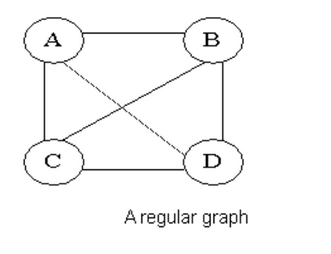
**3.Complete Graph**

A graph in which any V node is adjacent to all other nodes present in the graph is known as a complete graph. Anundirected graph contains the edges that are equal to edges = n(n-1)/2 where n is the number of vertices present inthe graph. The following figure shows a complete graph.



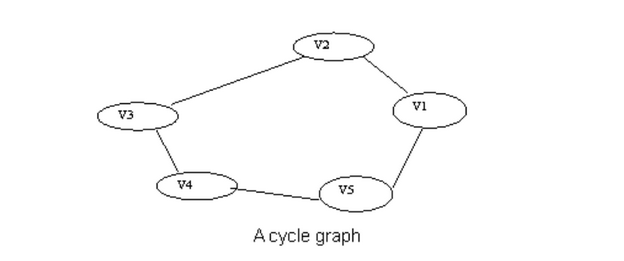
**4.Regular Graph**

Regular graph is the graph in which nodes are adjacent to each other, i.e., each node is accessible from any other Node.



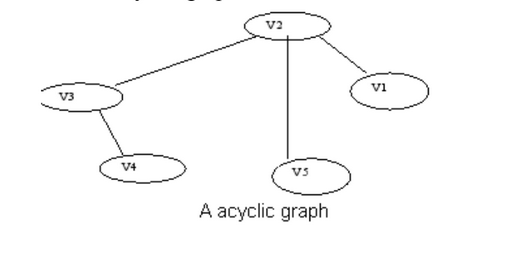
**5.Cycle Graph**

A graph having cycle is called cycle graph. In this case the first and last nodes are the same. A closed simple path is a cycle.



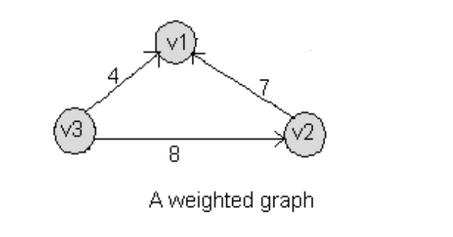
**6.Acyclic Graph**

A graph without cycle is called acyclic graphs.



**7. Weighted Graph**

A graph is said to be weighted if there are some non negative value assigned to each edges of the graph. Thevalue is equal to the length between two vertices. Weighted graph is also called a network.



Task10:

Wap to display a graph edges .in the below order no od edges 8 and no of vertex 5

1 - 2

1 - 3

1 - 4

2 - 4

2 - 5

3 - 4

3 - 5

4 - 5

class Graph {

// Edge class to represent graph edges

class Edge {

int src; // source vertex

int dest; // destination vertex

// Constructor

Edge(int src, int dest) {

this.src = src;

this.dest = dest;

}

}

int vertices; // number of vertices

int edges; // number of edges

Edge[] edgeArray; // array to store edges

// Constructor

Graph(int vertices, int edges) {

this.vertices = vertices;

this.edges = edges;

edgeArray = new Edge[edges];

for(int i = 0; i < edges; i++) {

edgeArray[i] = new Edge(0, 0);

}

}

// Method to create the graph

public void createGraph() {

// Adding edges as per the given structure

edgeArray[0] = new Edge(1, 2);

edgeArray[1] = new Edge(1, 3);

edgeArray[2] = new Edge(1, 4);

edgeArray[3] = new Edge(2, 4);

edgeArray[4] = new Edge(2, 5);

edgeArray[5] = new Edge(3, 4);

edgeArray[6] = new Edge(3, 5);

edgeArray[7] = new Edge(4, 5);

}

// Method to display the graph

public void displayGraph() {

System.*out*.println("Graph Edges:");

for(int i = 0; i < edges; i++) {

System.*out*.println(edgeArray[i].src + " - " + edgeArray[i].dest);

}

}

// Main method to test the graph

public static void main(String[] args) {

// Create graph with 5 vertices and 8 edges

Graph graph = new Graph(5, 8);

// Create the graph structure

graph.createGraph();

// Display the graph

graph.displayGraph();

}

}

