Student Name: _____ Roll No: _____ Section: _____

# Experiment No. 10

*Lab 10 – Trees and Graphs in Data Structures using JAVA.*

## Graphs and Trees

In this lab, we will learn about the constructing Graphs and Trees.

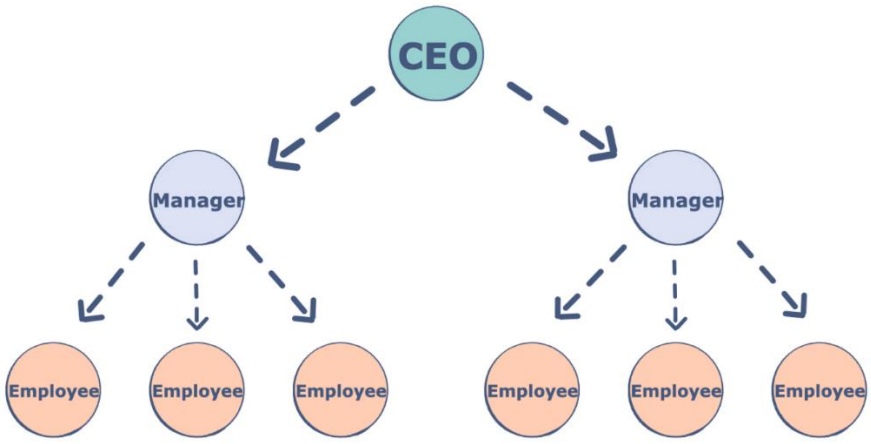
Data structures are an important part of programming and coding interviews. These skills show your ability to think complexly, solve ambiguous problems, and recognize coding patterns.

Programmers use data structures to organize data, so the more efficient your data structures are, the better your programs will be.

## What is a Tree?

Data Structures are used to store and organize data. We can use algorithms to manipulate and use our data structures. Different types of data are organized more efficiently by using different data structures.

Trees are non-linear data structures. They are often used to represent hierarchical data. For a real-world example, a hierarchical company structure uses a tree to organize.




## Components of a Tree

Trees are a collection of nodes (vertices), and they are linked with edges (pointers), representing the hierarchical connections between the nodes. A node contains data of any type, but all the nodes must be of the same data type. Trees are similar to graphs, but a cycle cannot exist in a tree. What are the different components of a tree?

Root: The root of a tree is a node that has no incoming link (i.e. no parent node). Think of this as a starting point of your tree.

Student Name: _____ Roll No: _____ Section: _____

**Children:** The child of a tree is a node with one incoming link from a node above it (i.e. a parent node). If two children nodes share the same parent, they are called siblings.

**Parent:** The parent node has an outgoing link connecting it to one or more child nodes.

**Leaf:** A leaf has a parent node but has no outgoing link to a child node. Think of this as an endpoint of your tree.

**Subtree:** A subtree is a smaller tree held within a larger tree. The root of that tree can be any node from the bigger tree.

**Depth:** The depth of a node is the number of edges between that node and the root. Think of this as how many steps there are between your node and the tree's starting point.

**Height:** The height of a node is the number of edges in the longest path from a node to a leaf node. Think of this as how many steps there are between your node and the tree's endpoint. The height of a tree is the height of its root node.

**Degree:** The degree of a node refers to the number of sub-trees.

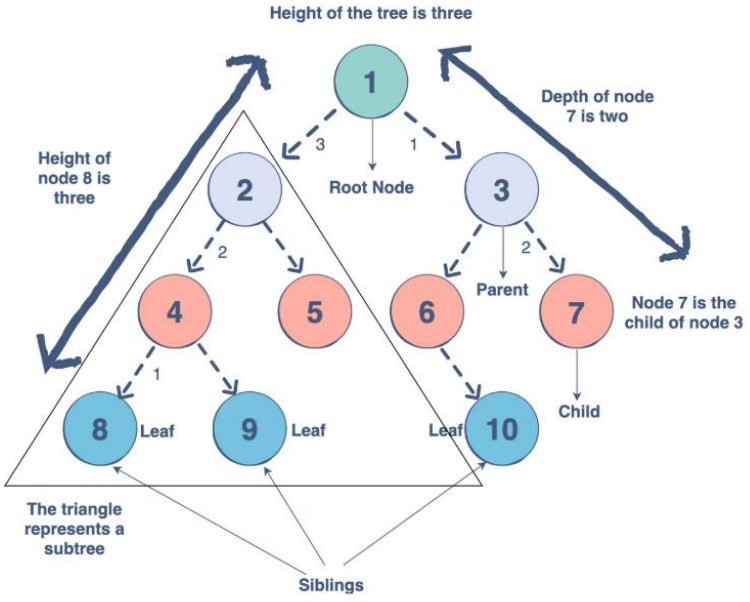



***Figure 10.1 : Concept of Graphs and Trees***

## Why do we use trees?

Trees can be applied to many things. The hierarchical structure gives a tree unique properties for storing, manipulating, and accessing data. Trees form some of the most basic organization of computers. We can use a tree for the following:

Student Name: _____ Roll No: _____ Section: _____

- Storage as hierarchy. Storing information that naturally occurs in a hierarchy. File systems on a computer and PDF use tree structures.
- Searching. Storing information that we want to search quickly. Trees are easier to search than a Linked List. Some types of trees (like AVL and Red-Black trees) are designed for fast searching.
- Inheritance. Trees are used for inheritance, XML parser, machine learning, and DNS, amongst many other things.
- Indexing. Advanced types of trees, like B-Trees and B+ Trees, can be used for indexing a database.
- Networking. Trees are ideal for things like social networking and computer chess games.
- Shortest path. A Spanning Tree can be used to find the shortest paths in routers for networking.
- and much more

## How to make a Tree?

We start with the root node.

```
Node<String> root = new Node<>("root");Queue<String> student1 = new LinkedList<>();
```

Once we have our root, we can add our first child node using addChild, which adds a child node and assigns it to a parent node. We refer to this process as insertion (adding nodes) and deletion (removing nodes).

```
NoNode<String> node1 = root.addChild(new Node<String>("node 1"));
```

We continue adding nodes using that same process until we have a complex hierarchical structure. In the next section, let's look at the different kinds of trees we can use.

There are many types of trees that we can use to organize data differently within a hierarchical structure. The tree we use depends on the problem we are trying to solve. Let's take a look at the trees we can use in Java. We will be covering:

- N-ary trees
- Balanced trees
- Binary trees
- Binary Search Trees
- AVL Trees
- Red-Black Trees
- 2-3 Trees
- 2-3-4 Trees

## N-ary Tree

In N-ary tree, a node can have child nodes from 0-N. For example, if we have a 2-ary tree (also called a Binary Tree), it will have a maximum of 0-2 child nodes.

12346378

## N-ary tree

Note: The balance factor of a node is the height difference between the left and right subtrees.

## Balanced Tree

A balanced tree is a tree with almost all leaf nodes at the same level, and it is most commonly applied to sub-trees, meaning that all sub-trees must be balanced. In other words, we must make the tree height balanced, where the difference between the height of the right and left subtrees do not exceed one. Here is a visual representation of a balanced tree.

1234567

## Binary tree

There are three main types of binary trees based on their structures.

## 1. Complete Binary Tree

A complete binary tree exists when every level, excluding the last, is filled and all nodes at the last level are as far left as they can be. Here is a visual representation of a complete binary tree.

125346

Complete Binary Tree (left side is completely filled)

## 2. Full Binary Tree

A full binary tree (sometimes called proper binary tree) exits when every node, excluding the leaves, has two children. Every level must be filled, and the nodes are as far left as possible. Look at this diagram to understand how a full binary tree looks.

1234567

a full binary tree

## 3. Perfect Binary Tree

A perfect binary tree should be both full and complete. All interior nodes should have two children, and all leaves must have the same depth. Look at this diagram to understand how a perfect binary tree looks.

1234567

a perfect binary tree

Note: You can also have a skewed binary tree, where all the nodes are shifted to the left or right, but it is best practice to avoid this type of tree in Java, as it is far more complex to search for a node.

## Binary Search Trees

A Binary Search Tree is a binary tree in which every node has a key and an associated value. This allows for quick lookup and edits (additions or removals), hence the name "search". A Binary Search Tree has strict conditions based on its node value. It's important to note that every Binary Search Tree is a binary tree, but not every binary tree is a Binary Search Tree.

What makes them different? In a Binary Search Tree, the left subtree of a subtree must contain nodes with fewer keys than a node's key, while the right subtree will contain nodes with keys greater than that node's key. Take a look at this visual to understand this condition.

Node XNode YSubTree 3SubTree 1SubTree 2

In this example, the node Y is a parent node with two child nodes. All nodes in subtree 1 must have a value less than node Y, and subtree 2 must have a greater value than node Y.

## AVL Trees

AVL trees are a special type of Binary Search tree that are self-balanced by checking the balance factor of every node. The balance factor should either be +1, 0, or -1. The maximum height difference between the left and right sub-trees can only be one.

If this difference becomes more than one, we must re-balance our tree to make it valid using rotation techniques. These are most common for applications where searching is the most important operation. Look at this visual to see a valid AVL tree.

64835207

## Red-Black Trees

A red-black tree is another type of self-balancing Binary Search Tree, but it has some additional properties to AVL trees. The nodes are colored either red or black to help re-balance a tree after insertion or deletion. They save you time with balancing. So, how do we color our nodes?

- The root is always black
- Two red nodes cannot be adjacent (i.e. a red parent cannot have a red child)
- A path from the root to a leaf should contain the same number of black nodes
- A null node is black

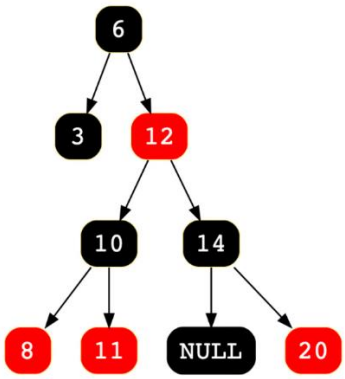



***Figure 10:2: A valid red-black tree***

Student Name: _____ Roll No: _____ Section: _____

## 2-3 Trees

A 2-3 tree is very different from what we've learned so far. Unlike a Binary Search Tree, a 2-3 Tree is a self-balancing, ordered, multiway search tree. It is always perfectly balanced, so every leaf node is equidistant from the root. Every node, other than leaf nodes, can be either a 2-Node (a node with a single data element and two children) or a 3-node (a node with two data elements and three children). A 2-3 tree will remain balanced no matter how many insertions or deletions occur.

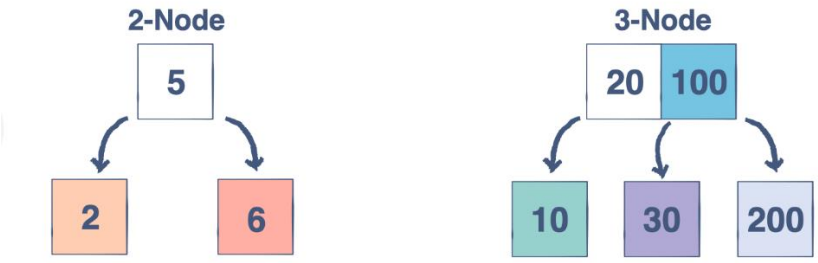



***Figure 10.3 : 2 & 3 Node Trees***

## 2-3-4 Trees

A 2-3-4 tree is a search tree that can accommodate more keys than a 2-3 tree. It covers the same basics as a 2-3 tree, but adds the following properties:

- 2-Node has two child nodes and one data element
- 3-Node has three child nodes and two data elements
- 4-Node has four child nodes and three data elements
- Each internal node has a max of 4 children
- For three keys at an internal node, all keys at the LeftChild node are smaller than the left key
- All keys at the LeftMidChild are smaller than the mid key
- All keys at the RightMidChild are smaller than the right key
- All keys at the RightChild are greater than the right key

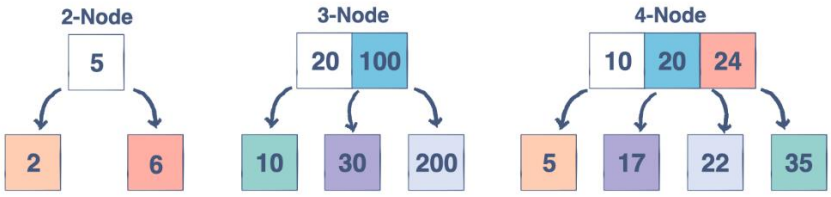



***Figure 10.4: 2, 3 ,& 4 node Trees***

## Intro to Tree Traversal and Searching

To use trees, we can traverse them by visiting/checking every node of a tree. If a tree is "traversed", this means every node has been visited. There are four ways to traverse a tree. These four processes fall into one of two categories: breadth-first traversal or depth-first traversal.

Student Name: _____ Roll No: _____ Section: _____

- **Inorder:** Think of this as moving up the tree, then back down. You traverse the left child and its sub-tree until you reach the root. Then, traverse down the right child and its subtree. This is a depth-first traversal.

- **Preorder:** This of this as starting at the top, moving down, and then over. You start at the root, traverse the left sub-tree, and then move over to the right sub-tree. This is a depth-first traversal.

- **Postorder:** Begin with the left-sub tree and move over to the right sub-tree. Then, move up to visit the root node. This is a depth-first traversal.

- **Levelorder:** Think of this as a sort of zig-zag pattern. This will traverse the nodes by their levels instead of subtrees. First, we visit the root and visit all children of that root, left to right. We then move down to the next level until we reach a node that has no children. This is the left node. This is a breadth-first traversal.

So, what's the difference between a breadth-first and depth-first traversal? Let's take a look at the algorithms Depth-First Search (DFS) and Breath-First Search (BFS) to understand this better.
Note: Algorithms are a sequence of instructions for performing certain tasks. We use algorithms with data structures to manipulate our data, in this case, to traverse our data.

## Depth-First Search

Overview: We follow a path from the starting node to the ending node and then start another path until all nodes are visited. This is commonly implemented using stacks, and it requires less memory than BFS. It is best for topographical sorting, such as graph backtracking or cycle detection.

The steps for the DFS algorithm are as follows:

1. Pick a node. Push all adjacent nodes into a stack.
2. Pop a node from that stack and push adjacent nodes into another stack.
3. Repeat until the stack is empty or you have reached your goal. As you visit nodes, you must mark them as visited before proceeding, or you will be stuck in an infinite loop.

## Breadth-First Search

Overview: We proceed level-by-level to visit all nodes at one level before going to the next. The BFS algorithm is commonly implemented using queues, and it requires more memory than the DFS algorithm. It is best for finding the shortest path between two nodes.

The steps for the BFS algorithm are as follows:

1. Pick a node. Enqueue all adjacent nodes into a queue. Dequeue a node, and mark it as visited. Enqueue all adjacent nodes into another queue.
2. Repeat until the queue is empty of you have met your goal.
3. As you visit nodes, you must mark them as visited before proceeding, or you will be stuck in an infinite loop.

Student Name: _____ Roll No: _____ Section: _____

## Search in Binary Search Trees

It's important to know how to perfom a search in a tree. Searching means we are locating a specific element or node in our data structure. Since data in a Binary Search Tree is ordered, searching is quite easy. Let's see how it's done.

1. Start at the root.
2. If the value is less than the value of the current node, we traverse the left sub-tree. If it is more, we traverse the right sub-tree.
3. Continue this process until you reach a node with that value or reach a leaf node, meaning that the value doesn't exist.

In the below example, we are searching for the value 3 in our tree. Take a look.

5216-43

Here is our tree. Since 3 is less than 5, we traverse the left sub-tree.

5216-43

Since 3 is more than 2, we move to the right child.

5216-43

The searched value has been found.

**Program 1:** Write a simple Binary Search Tree Program

```
public class BinarySearchTree {


	public boolean search(int value) {
		if (root == null)
			return false;
		else
			return root.search(value);
	}
}
public class BSTNode {
	…
	public boolean search(int value) {
		if (value == this.value)
			return true;
		else if (value < this.value) {
			if (left == null)
				return false;
			else
				return left.search(value);
		} else if (value > this.value) {
			if (right == null)
				return false;
			else
				return right.search(value);
		}
```

Student Name: _____ Roll No: _____ Section: _____

```
            return false;
        }
}
```

**Output:**

**Program 2:** Implement Graph Data Structure

```
class Graph {

  // inner class
  // to keep track of edges
  class Edge {
    int src, dest;
  }

  // number of vertices and edges
  int vertices, edges;

  // array to store all edges
  Edge[] edge;

  Graph(int vertices, int edges) {
    this.vertices = vertices;
    this.edges = edges;

    // initialize the edge array
    edge = new Edge[edges];
    for(int i = 0; i < edges; i++) {

      // each element of the edge array
      // is an object of Edge type
      edge[i] = new Edge();
    }
  }

  public static void main(String[] args) {

    // create an object of Graph class
    int noVertices = 5;
    int noEdges = 8;
    Graph g = new Graph(noVertices, noEdges);

    // create graph
    g.edge[0].src = 1;   // edge 1---2
```

Student Name: _____ Roll No: _____ Section: _____

```
    g.edge[0].dest = 2;

    g.edge[1].src = 1;   // edge 1---3
    g.edge[1].dest = 3;

    g.edge[2].src = 1;   // edge 1---4
    g.edge[2].dest = 4;

    g.edge[3].src = 2;   // edge 2---4
    g.edge[3].dest = 4;

    g.edge[4].src = 2;   // edge 2---5
    g.edge[4].dest = 5;

    g.edge[5].src = 3;   // edge 3---4
    g.edge[5].dest = 4;

    g.edge[6].src = 3;   // edge 3---5
    g.edge[6].dest = 5;

    g.edge[7].src = 4;   // edge 4---5
    g.edge[7].dest = 5;

    // print graph
    for(int i = 0; i < noEdges; i++) {
      System.out.println(g.edge[i].src + " - " + g.edge[i].dest);
    }

  }
}
```

**Output:**

**Program 3:** Program to detect cycle in an undirected graph

```
import java.io.*;
import java.util.*;
// This class represents a directed graph using adjacency list representation
class Graph
{

      // No. of vertices
```

Student Name: _____ Roll No: _____ Section: _____

```
	private int V;

	// Adjacency List Representation
	private LinkedList<Integer> adj[];

	// Constructor
	Graph(int v)
	{
		V = v;
		adj = new LinkedList[v];
		for(int i=0; i<v; ++i)
			adj[i] = new LinkedList();
	}

	// Function to add an edge
	// into the graph
	void addEdge(int v,int w)
	{
		adj[v].add(w);
		adj[w].add(v);
	}

	// A recursive function that uses visited[] and parent to detect cycle in
subgraph reachable
	// from vertex v.
	Boolean isCyclicUtil(int v,
				Boolean visited[], int parent)
	{
		// Mark the current node as visited
		visited[v] = true;
		Integer i;

		// Recur for all the vertices adjacent to this vertex
		Iterator<Integer> it =
				adj[v].iterator();
		while (it.hasNext())
		{
			i = it.next();

			// If an adjacent is not
			// visited, then recur for that
			// adjacent
			if (!visited[i])
			{
				if (isCyclicUtil(i, visited, v))
					return true;
			}
```

Student Name: _____ Roll No: _____ Section: _____

```
                    // If an adjacent is visited and not parent of current vertex,
then there is a cycle.
                    else if (i != parent)
                          return true;
             }
             return false;
      }

      // Returns true if the graph contains a cycle, else false.
      Boolean isCyclic()
      {

             // Mark all the vertices as not visited and not part of recursion stack
             Boolean visited[] = new Boolean[V];
             for (int i = 0; i < V; i++)
                    visited[i] = false;

             // Call the recursive helper function to detect cycle in different DFS
trees
             for (int u = 0; u < V; u++)
             {

                    // Don't recur for u if already visited
                    if (!visited[u])
                          if (isCyclicUtil(u, visited, -1))
                                return true;
             }

             return false;
      }


      // Driver method to test above methods
      public static void main(String args[])
      {

             // Create a graph given
             // in the above diagram
             Graph g1 = new Graph(5);
             g1.addEdge(1, 0);
             g1.addEdge(0, 2);
             g1.addEdge(2, 1);
             g1.addEdge(0, 3);
             g1.addEdge(3, 4);
             if (g1.isCyclic())
                    System.out.println("Graph
```

Student Name: _____ Roll No: _____ Section: _____

```
                                        contains cycle");
            else
                  System.out.println("Graph
                                        doesn't contains cycle");

            Graph g2 = new Graph(3);
            g2.addEdge(0, 1);
            g2.addEdge(1, 2);
            if (g2.isCyclic())
                  System.out.println("Graph
                                        contains cycle");
            else
                  System.out.println("Graph
                                        doesn't contains cycle");
      }
}
```

**Output:**

**Program 4:** Program to find number of triangles in an Undirected Graph. The program is for adjacency matrix representation of the graph

```
import java.io.*;


class Directed
{
      int V = 4;
void multiply(int A[][], int B[][],int C[][])
{
      for (int i = 0; i < V; i++)
      {
            for (int j = 0; j < V; j++)
            {
                  C[i][j] = 0;
                  for (int k = 0; k < V;
                                          k++)
```

Student Name: _____ Roll No: _____ Section: _____

```
                        {
                                C[i][j] += A[i][k]* B[k][j];
                        }
                }
        }
}
int getTrace(int graph[][])
{
        int trace = 0;

        for (int i = 0; i < V; i++)
        {
                trace += graph[i][i];
        }
        return trace;
}
int triangleInGraph(int graph[][])
{
        // To Store graph^2
        int[][] aux2 = new int[V][V];

        // To Store graph^3
        int[][] aux3 = new int[V][V];

        // Initialising aux matrices
        // with 0
        for (int i = 0; i < V; ++i)
        {
                for (int j = 0; j < V; ++j)
                {
                        aux2[i][j] = aux3[i][j] = 0;
                }
        }
```

Student Name: _____ Roll No: _____ Section: _____

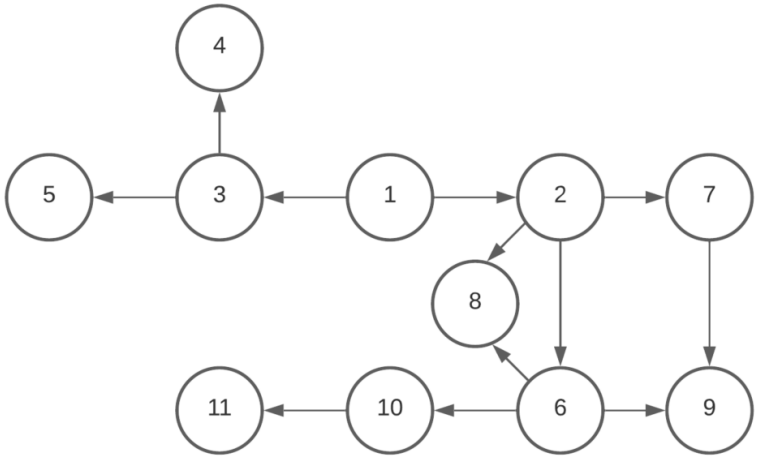
```
        multiply(graph, graph, aux2);

        // after this multiplication aux3
        // is graph^3 printMatrix(aux3)
        multiply(graph, aux2, aux3);

        int trace = getTrace(aux3);

        return trace / 6;
}

// Driver code
public static void main(String args[])
{
        Directed obj = new Directed();

        int graph[][] = { {0, 1, 1, 0},
                                        {1, 0, 1, 1},
                                        {1, 1, 0, 1},
                                        {0, 1, 1, 0}
                                };

        System.out.println("Total number of Triangle in Graph : "+
                        obj.triangleInGraph(graph));
}
}
```

**Output:**

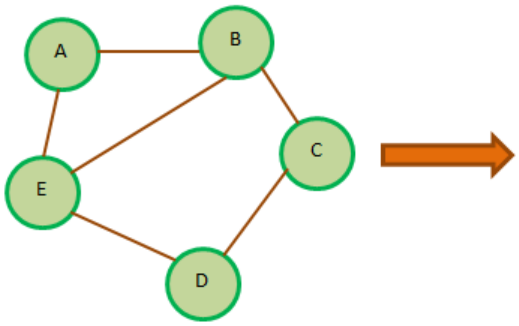
Student Name: _____ Roll No: _____ Section: _____

## Programming Exercise

1. Write a code to check weather a graph is connected or not?
2. Write a code to differentiate between trees and a graph.
3. Implement the following graphs and show how you can find that it has cycles in it or not. Calculate its vertices and edges, in degrees and out degrees. Then convert your program into generic code.




4. Write a code for the Adjacency Matrix. An Adjacency Matrix is simple technique to use 2-dimensional array into matrix format.

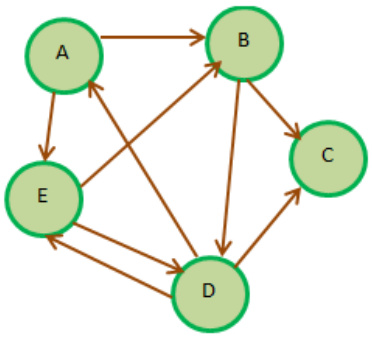



*Undirected graph*

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 1 | 0 | 0 | 1 |
| B | 1 | 0 | 1 | 0 | 1 |
| C | 0 | 1 | 0 | 1 | 0 |
| D | 0 | 0 | 1 | 0 | 1 |
| E | 1 | 1 | 0 | 1 | 0 |

*Adjacency* Matrix

Student Name: _____ Roll No: _____ Section: _____

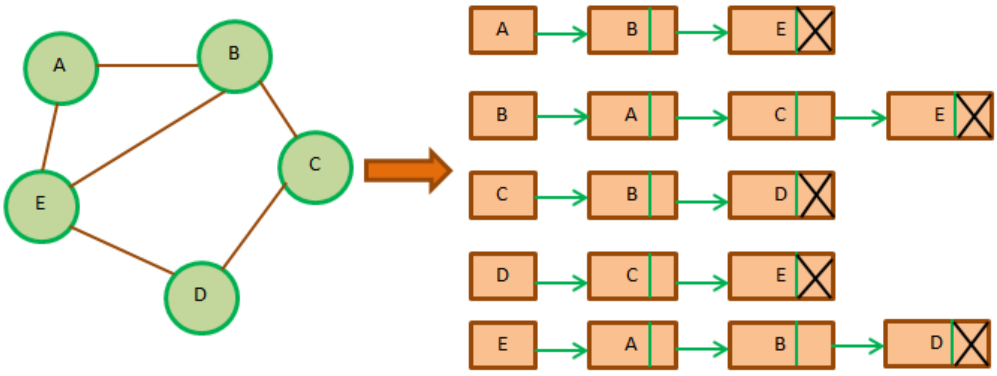



Directed graph

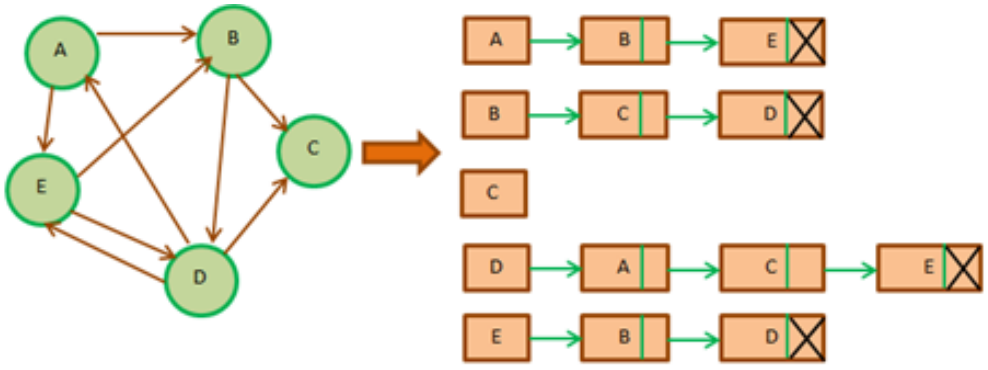
| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 1 | 0 | 0 | 1 |
| B | 0 | 0 | 1 | 1 | 0 |
| C | 0 | 0 | 0 | 0 | 0 |
| D | 1 | 0 | 1 | 0 | 1 |
| E | 0 | 1 | 0 | 1 | 0 |

*Adjacency* Matrix

5. Now instead of using array what if, we use linked list. Now convert the code to show how it works using linked list.




*Undirected graph* Adjacency list




*Directed graph* *Adjacency list*