

Introduction and Preliminaries

Graph: A linear graph or simply a graph $G = (V, E)$ consists of a set of objects $V = \{v_1, v_2, \dots, v_n\}$ called vertices, and another set $E = \{e_1, e_2, \dots, e_m\}$, whose elements are called as edges, such that each edge e_k is identified with an unordered pair of vertices (v_i, v_j) of vertices. The vertices v_i, v_j associated with edge e_k is called end vertices of e_k .

A graph is denoted by $G = (V, E)$. A graph in this context is made up of vertices, nodes, or points which are connected by edges, arcs, or lines.

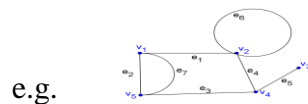


Figure 1-1

Here, $V = \{v_1, v_2, v_3, v_4, v_5\}$

$E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$ and $G = (V, E)$

Figure 1-1: Graph with five vertices and seven edges.

Adjacent vertices: Two vertices are said to be adjacent if they are end vertices of some edge.

In figure 1-1, v_1 and v_5 are adjacent.

Directed Edges and Directed Graph: In a graph, nodes are connected by edges which are bidirectional are called undirectional edges.

A graph in which all are directed is called a **directed graph**.

Here all edges indicate a one-way relationship, in that each edge can only be traversed in a single direction.

Undirected Edges and undirected Graph: In a graph, edges have an orientation, which is represented by an arrow are called directed edges.

Here head and tail of this arrow are the nodes representing the initiating and terminating points(nodes) of the edge.

A graph in which all edges are undirected is called undirected graph.

Loop: If an edge e_i has same end vertices than e_i is called a loop.

In figure 1-1: e_6 denotes a loop.

Parallel edges: Two edges e_i and e_j have same pair of end vertices than we say that e_i and e_j are parallel edges.

In figure 1-1: e_2 and e_7 are parallel edges.

Simple graph: A graph which has neither parallel edges nor loops is called a simple graph.
e.g.

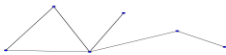
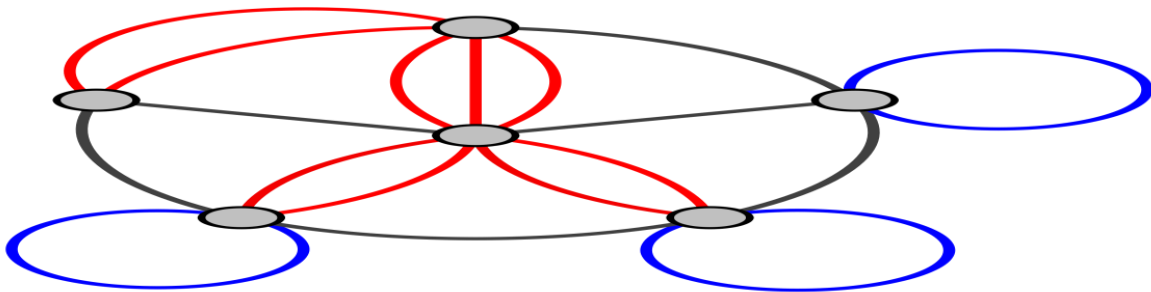
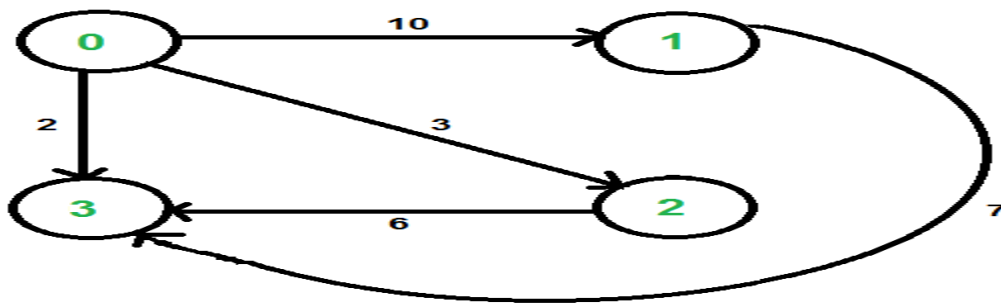


Figure 1-2

Multi-graph: a multigraph (in contrast to a simple graph) is a graph which is permitted to have multiple edges (also called parallel edges), that is, edges that have the same end nodes. Thus two vertices may be connected by more than one edge.



Weighted Graph: A weighted graph is a graph in which each branch is given a numerical weight. A weighted graph is therefore a special type of labeled graph in which the labels are numbers (which are usually taken to be positive).



Isolated Nodes: Node which is not connected with any edge.



In figure, G is isolated vertex.

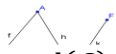
Null Graph: A graph whose every nodes are isolated is called a null graph.



Isomorphic Graphs: Two graphs G and H are said to be isomorphic if there is a one–one correspondence between their vertices which preserves the adjacency of the vertices.

In above figure, both graph are isomorphic.

Degree: If v_i is an end vertex of an edge e_j then we say that e_j and v_i are independent of each other at v_i . The total number of edges which are incident at vertex ' v ' is called the degree of the vertex ' v ' and is denoted by $d(v)$. In a multi-graph, loops are counted twice.



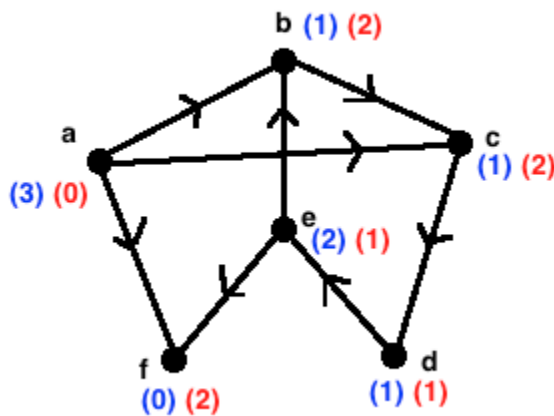
Here $d(C) = 4$

The degree sum formula states that, given a graph $G=(V,E)$,

$$\sum_{v \in V} \deg(v) = 2|E|$$

The formula implies that in any undirected graph, the **number of vertices with odd degree is even**. This statement (as well as the degree sum formula) is known as the **handshaking lemma**. The latter name comes from a popular mathematical problem, to prove that in any group of people the number of people who have shaken hands with an odd number of other people from the group is even.

In-degree & Out-degree: For a directed graph and a node, the Out-Degree of refers to the number of edges incident from. That is, the number of edges directed away from the nodes. The In-Degree of refers to the number of edges incident to. That is, the number of edges directed towards the node.



Blue labels indicate out degree of nodes and red labels indicate in degree of nodes.

For any graph sum of total in-degree should be equal to total out-degree

Total Degree of a node: In a directed graph the total degree of the node is the sum of its in-degree and out-degree.

In above graph total degree of node e is $1 + 2 = 3$.

Adjacent edges: Two non-parallel edges are said to be adjacent if they have a common end nodes.

In above figure, i and j are adjacent.

Pendant vertex: A vertex having degree one, is called a pendent vertex.

In above figure, E is pendant vertex.

Subgraph: A graph g is said to be sub graph of a graph G if all vertices and all the edges of g are also vertices and edges of G and each edge in g has the same end vertices as in G .

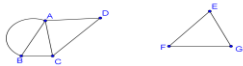


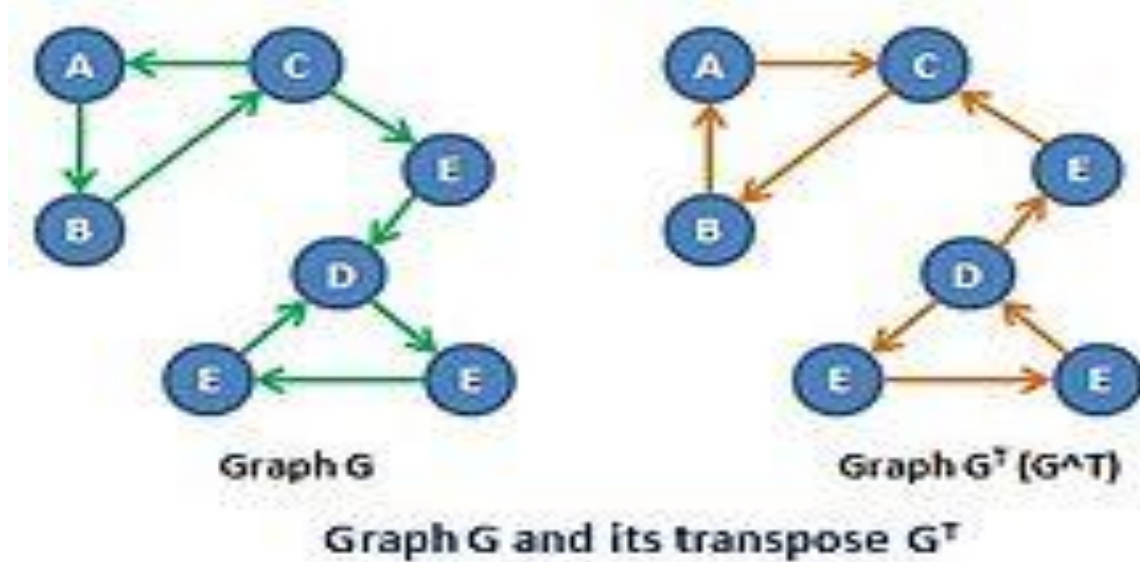
Figure 1-3



Figure 1-4

Graph in figure 1-4 is sub graph of graph shown in figure 1-3.

Converse of a digraph(Directional dual) : the converse, transpose or reverse of a directed graph G is another directed graph on the same set of vertices with all of the edges reversed compared to the orientation of the corresponding edges in G . That is, if G contains an edge (u, v) then the converse/transpose/reverse of G contains an edge (v, u) and vice versa.



Path

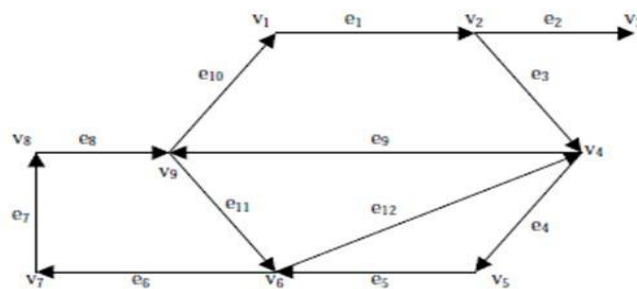
1) In a directed graph a sequence of edges $e_1, e_2, e_3, \dots, e_n$ where e_i is associated to (x_i, x_{i+1}) for $i = 0, 1, 2, \dots, n$ where $x_0 = u$ and $x_{n+1} = v$.

Length of a path: length of the path is the number of edges traversed.

Simple Path & Elementary Path:

Elementary or Simple Path

- In a directed graph, a path is a sequence of edges ($e_1, e_2, e_3, \dots, e_n$) such that the edges are connected with each other
- A **path** is said to be **elementary** if it does not meet the same vertex twice
- A **path** is said to be **simple** if it does not meet the same edges twice



27

Circuit A closed path is called a circuit.

Reachability: It refers to the ability to get from one vertex to another within a graph. A vertex can reach a vertex (and is reachable from) if there exists a sequence of adjacent vertices (i.e. a path) which starts with and ends with.

Geodesic: The distance between two vertices in a graph is the number of edges in a shortest path (also called a graph geodesic) connecting them. This is also known as the geodesic distance. Notice that there may be more than one shortest path between two vertices.

Triangle inequality: If the distance $d(u, v)$ between two vertices u and v that can be connected by a path in a graph is defined to be the length of the shortest path connecting them, then prove that the distance function satisfies the triangle inequality: $d(u, v) + d(v, w) \geq d(u, w)$.

Proof: If you simply connect the paths from u to v to the path connecting v to w you will have a valid path of length $d(u, v) + d(v, w)$. Since we are looking for the path of minimal length, if there is a shorter path it will be shorter than this one, so the triangle inequality will be satisfied.

Connectedness:

An undirected graph is said to be **connected** if there is a path between every two vertices, and is said to be disconnected otherwise.

A directed graph is said to be **connected** if the undirected graph derived from it by ignoring the directions of the edge is connected and is said to be disconnected otherwise.

It follows that a disconnected graph consists of two or more components each of which is a connected graph.

Strongly connected graph: A directed graph is said to be strongly connected if for every two vertices a and b in the graph there is a path from a to b as well as a path from b to a .

Weakly connected graph: A directed graph is called weakly connected graph if replacing all of its directed edges with undirected edges produces a connected (undirected) graph.

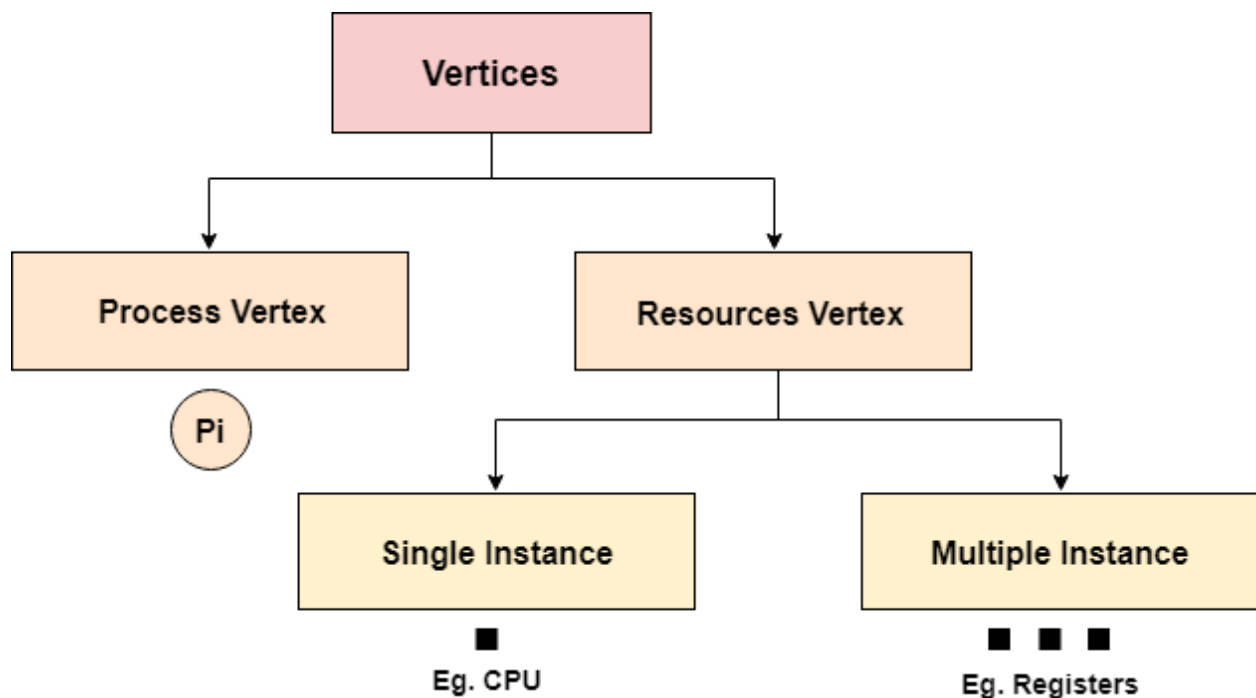
Unilaterally connected graph: A directed graph is said to be unilaterally connected if for any pair of nodes, at least one of the nodes is reachable from the other.

Resource Allocation Graph

The resource allocation graph is the pictorial representation of the state of a system. As its name suggests, the resource allocation graph is the complete information about all the processes which are holding some resources or waiting for some resources.

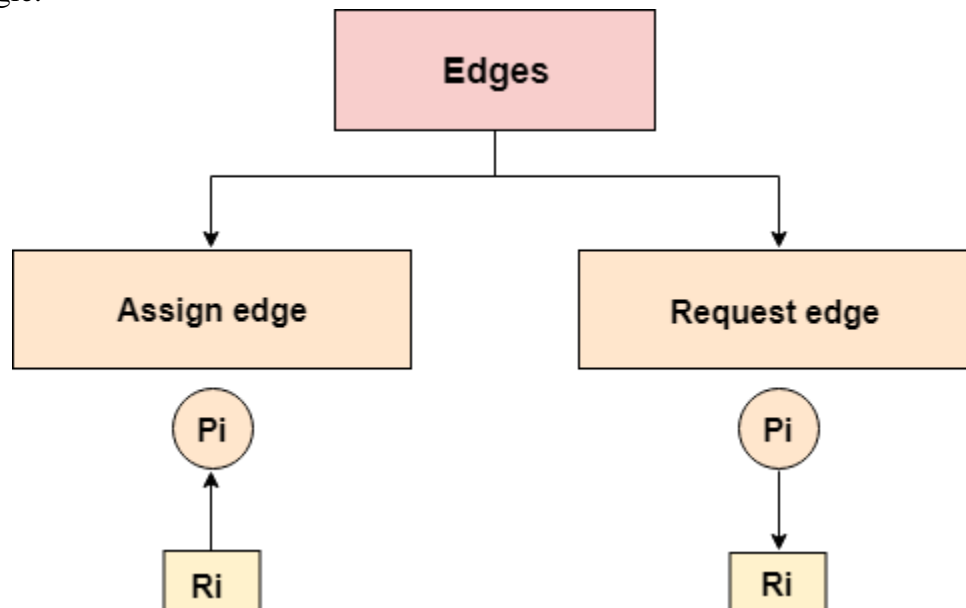
It also contains the information about all the instances of all the resources whether they are available or being used by the processes.

In Resource allocation graph, the process is represented by a Circle while the Resource is represented by a rectangle. Let's see the types of vertices and edges in detail.



Vertices are mainly of two types, Resource and process. Each of them will be represented by a different shape. Circle represents process while rectangle represents resource.

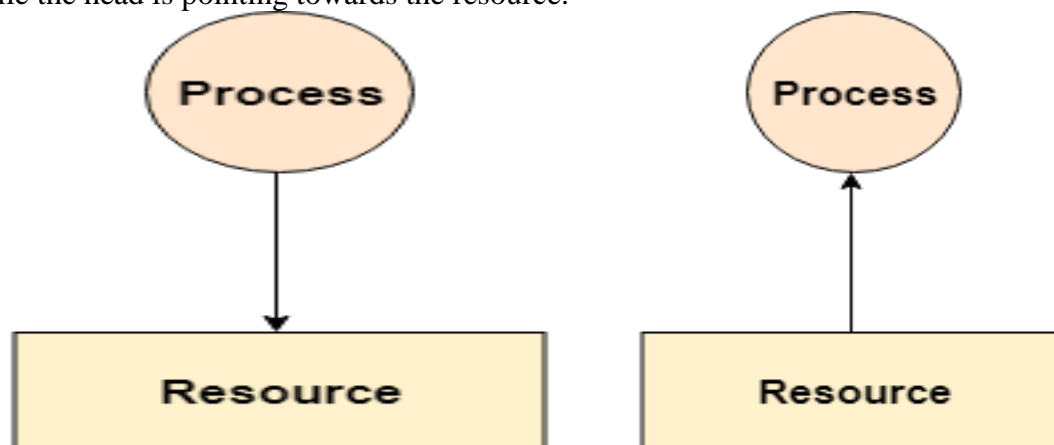
A resource can have more than one instance. Each instance will be represented by a dot inside the rectangle.



Edges in RAG are also of two types, one represents assignment and other represents the wait of a process for a resource. The above image shows each of them.

A resource is shown as assigned to a process if the tail of the arrow is attached to an instance to the resource and the head is attached to a process.

A process is shown as waiting for a resource if the tail of an arrow is attached to the process while the head is pointing towards the resource.



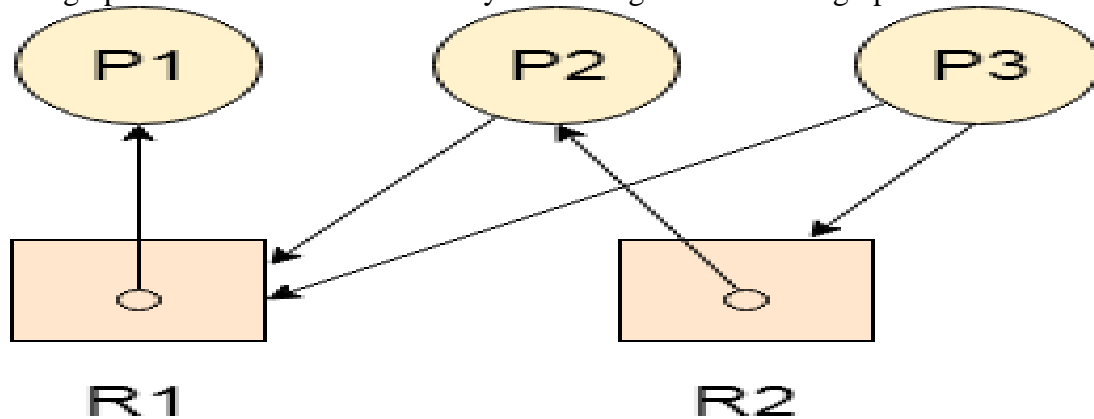
**Process is requesting Resource is assigned
for a resource to process**

Example

Let's consider 3 processes P1, P2 and P3, and two types of resources R1 and R2. The resources are having 1 instance each.

According to the graph, R1 is being used by P1, P2 is holding R2 and waiting for R1, P3 is waiting for R1 as well as R2.

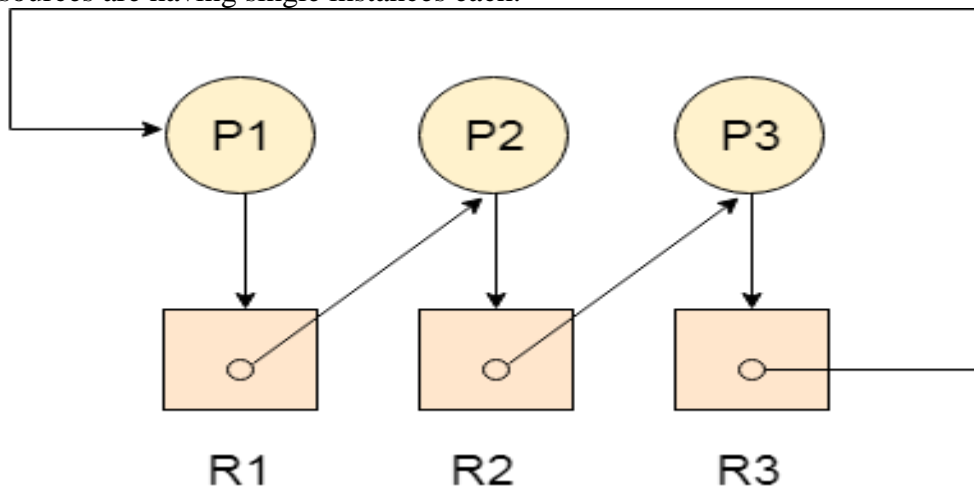
The graph is deadlock free since no cycle is being formed in the graph.

**Deadlock Detection using RAG**

If a cycle is being formed in a Resource allocation graph where all the resources have the single instance then the system is deadlocked.

In Case of Resource allocation graph with multi-instanced resource types, Cycle is a necessary condition of deadlock but not the sufficient condition.

The following example contains three processes P1, P2, P3 and three resources R1, R2, R3. All the resources are having single instances each.



If we analyze the graph then we can find out that there is a cycle formed in the graph since the system is satisfying all the four conditions of deadlock.

Allocation Matrix

Allocation matrix can be formed by using the Resource allocation graph of a system. In Allocation matrix, an entry will be made for each of the resource assigned. For Example, in the following matrix, an entry is being made in front of P1 and below R3 since R3 is assigned to P1

Process	R1	R2	R3
P1	0	0	1
P2	1	0	0
P3	0	1	0

Request Matrix

In request matrix, an entry will be made for each of the resource requested. As in the following example, P1 needs R1 therefore an entry is being made in front of P1 and below R1.

Process	R1	R2	R3
P1	1	0	0
P2	0	1	0
P3	0	0	1

Avial = (0,0,0)

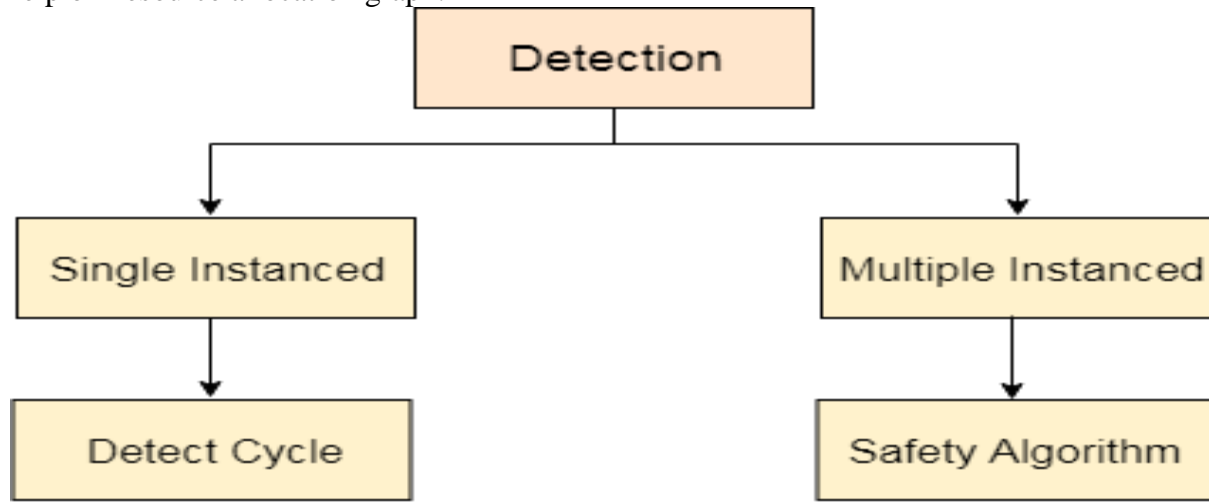
Neither we are having any resource available in the system nor a process going to release. Each of the process needs at least single resource to complete therefore they will continuously be holding each one of them.

We cannot fulfill the demand of at least one process using the available resources therefore the system is deadlocked as determined earlier when we detected a cycle in the graph.

Deadlock Detection and Recovery

In this approach, The OS doesn't apply any mechanism to avoid or prevent the deadlocks. Therefore the system considers that the deadlock will definitely occur. In order to get rid of deadlocks, The OS periodically checks the system for any deadlock. In case, it finds any of the deadlock then the OS will recover the system using some recovery techniques.

The main task of the OS is detecting the deadlocks. The OS can detect the deadlocks with the help of Resource allocation graph.



In single instanced resource types, if a cycle is being formed in the system then there will definitely be a deadlock. On the other hand, in multiple instanced resource type graph, detecting a cycle is not just enough. We have to apply the safety algorithm on the system by converting the resource allocation graph into the allocation matrix and request matrix.

In order to recover the system from deadlocks, either OS considers resources or processes.

For Resource

Preempt the resource

We can snatch one of the resources from the owner of the resource (process) and give it to the other process with the expectation that it will complete the execution and will release this resource sooner. Well, choosing a resource which will be snatched is going to be a bit difficult.

Rollback to a safe state

System passes through various states to get into the deadlock state. The operating system can rollback the system to the previous safe state. For this purpose, OS needs to implement check pointing at every state.

The moment, we get into deadlock, we will rollback all the allocations to get into the previous safe state.

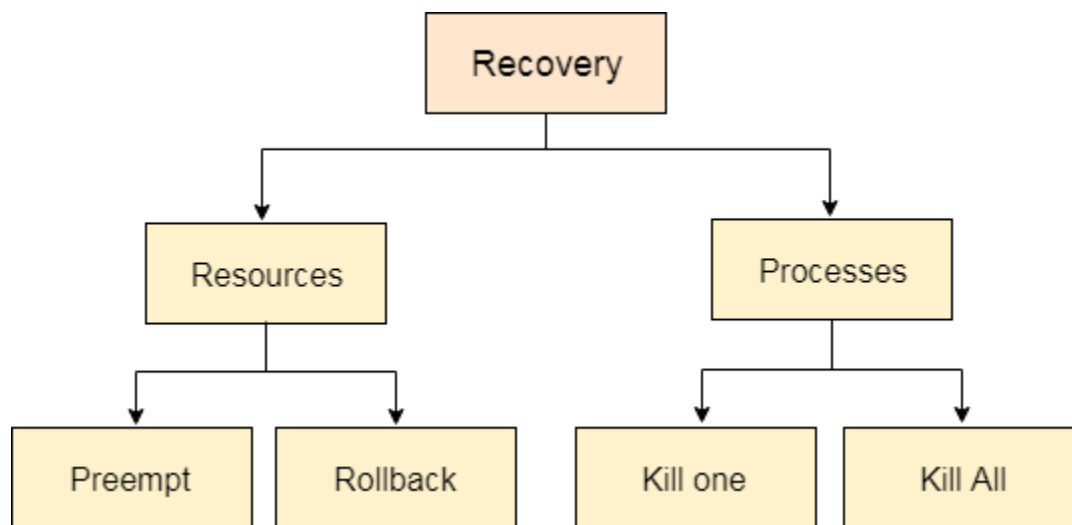
For Process

Kill a process

Killing a process can solve our problem but the bigger concern is to decide which process to kill. Generally, Operating system kills a process which has done least amount of work until now.

Kill all process

This is not a suggestible approach but can be implemented if the problem becomes very serious. Killing all process will lead to inefficiency in the system because all the processes will execute again from starting.



Matrix representation of Graph:

Carrying out graph algorithms using the representation of graphs by list of edges is cumbersome if there are many edges in the graph. To simplify computation, graphs can be represented using matrices.

Adjacency matrix: Suppose that $G = (V, E)$ is a simple graph where $|V| = n$. Suppose that the vertices of G are listed arbitrarily as v_1, v_2, \dots, v_n . The **adjacency matrix** A of G , with respect to this listing of the vertices, is the $n \times n$ zero-one matrix with 1 as its (i, j) th entry when v_i and v_j are adjacent, and 0 as its (i, j) th entry when they are not adjacent. In other words, if its adjacency matrix is $A = [a_{ij}]$, then

$$a_{ij} = \begin{cases} 1, & \text{if } \{v_i, v_j\} \text{ is an edge of } G \\ 0, & \text{otherwise} \end{cases}$$

Zero-One Matrix: A matrix with entries either 0 or 1 is called a zero-one matrix. Zero-one matrices are often used to represent discrete structures.

Boolean or Bit Matrix: Algorithms using these structures are based on Boolean arithmetic with zero-one matrices. This arithmetic is based on the Boolean operations \wedge and \vee , which operate on pairs of bits, defined by

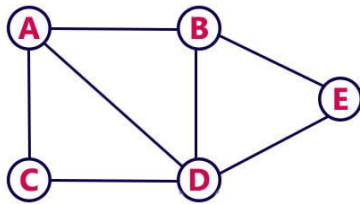
$$b_1 \wedge b_2 = \begin{cases} 1, & \text{if } b_1 = b_2 = 1 \\ 0, & \text{otherwise} \end{cases}$$

$$b_1 \vee b_2 = \begin{cases} 1, & \text{if } b_1 = 1 \text{ or } b_2 = 1 \\ 0, & \text{otherwise} \end{cases}$$

Adjacency matrix of Diagraph: Matrix $A = [a_{ij}]$ with entries

$$a_{ij} = \begin{cases} 1, & \text{if } G \text{ has a directed edge } (i, j) \\ 0, & \text{else} \end{cases}.$$

Finding paths of length n in a graph: Suppose you have a non-directed graph, represented through its adjacency matrix. How would you discover how many paths of length link any two nodes?



For example, in the graph aside there is one path of length 2 that links nodes A and B (A-D-B). How can this be discovered from its adjacency matrix?

It turns out there is a beautiful mathematical way of obtaining this information! Although this is not the way it is used in practice, it is still very nice. In fact, Breadth First Search is used to find paths of any length given a starting node.

PROP. $[A^n]_{ij}$ holds the number of paths of length n from node i to j node .

Let's see how this proposition works. Consider the adjacency matrix of the graph above:

$$[A]_{ij} = \begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 1 & 0 \\ B & 1 & 0 & 0 & 1 & 1 \\ C & 1 & 0 & 0 & 1 & 0 \\ D & 1 & 1 & 1 & 0 & 1 \\ E & 0 & 1 & 0 & 1 & 0 \end{array}$$

With $n = 2$ we should find paths of length 2. So we first need to square the adjacency matrix:

$$[A^2]_{ij} = \begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 3 & 1 & 1 & 2 & 2 \\ B & 1 & 3 & 2 & 2 & 1 \\ C & 1 & 2 & 2 & 1 & 1 \\ D & 2 & 2 & 1 & 4 & 1 \\ E & 2 & 1 & 1 & 1 & 2 \end{array}$$

Back to our original question: how to discover that there is only one path of length 2 between nodes A and B? Just look at the value $[A^2]_{12}$, which is 1 as expected! Another example: $[A^2]_{22} = 3$, because there are 3 paths that link B with itself: B-A-B, B-D-B and B-E-B.

This will work with any pair of nodes, of course, as well as with any power to get paths of any length.

Transitive closure of a graph: Given a directed graph, find out if a vertex j is reachable from another vertex i for all vertex pairs (i, j) in the given graph. Here reachable mean that there is a path from vertex i to j . The reach-ability matrix is called transitive closure of a graph.

Section V.6: Warshall's Algorithm to find Transitive Closure

Definition V.6.1: Let S be the finite set $\{v_1, \dots, v_n\}$, R a relation on S . The **adjacency matrix** A of R is an $n \times n$ Boolean (zero-one) matrix defined by

$$A_{i,j} = \begin{cases} 1 & \text{if the digraph } D \text{ has an edge from } v_i \text{ to } v_j \\ 0 & \text{if the digraph } D \text{ has no edge from } v_i \text{ to } v_j \end{cases}$$

(This is a special case of the adjacency matrix M of a directed graph in Epp p. 642. Her definition allows for more than one edge between two vertices. But the digraph of a relation has at most **one** edge between any two vertices).

Warshall's algorithm is an efficient method of finding the adjacency matrix of the transitive closure of relation R on a finite set S from the adjacency matrix of R . It uses properties of the digraph D , in particular, walks of various lengths in D .

The definition of walk, transitive closure, relation, and digraph are all found in Epp.

Definition V.6.2: We let A be the adjacency matrix of R and T be the adjacency matrix of the transitive closure of R . T is called the **reachability matrix** of digraph D due to the property that $T_{i,j} = 1$ if and only if v_j can be reached from v_i in D by a sequence of arcs (edges).

Digraph Implementation

Definition V.6.3: If $a, v_1, v_2, \dots, v_n, b$ is a walk in a digraph D , $a \neq v_1, b \neq v_n, n > 2$, then v_1, v_2, \dots and v_n are the **interior vertices** of this walk (path).

In **Warshall's algorithm** we construct a sequence of Boolean matrices $A = W^{[0]}, W^{[1]}, W^{[2]}, \dots, W^{[n]} = T$, where A and T are as above. This can be done from digraph D as follows.

$[W^{[1]}]_{i,j} = 1$ if and only if there is a walk from v_i to v_j with elements of a subset of $\{v_1\}$ as interior vertices.

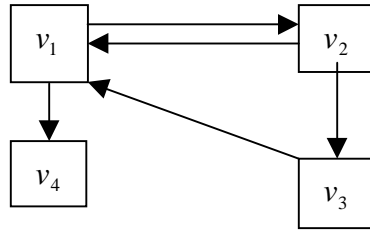
$[W^{[2]}]_{i,j} = 1$ if and only if there is a walk from v_i to v_j with elements of a subset of $\{v_1, v_2\}$ as interior vertices.

Continuing this process, we generalize to

$[W^{[k]}]_{i,j} = 1$ if and only if there is a walk from v_i to v_j with elements of a subset of $\{v_1, v_2, \dots, v_k\}$ as interior vertices.

Note: In constructing $W^{[k]}$ from $W^{[k-1]}$ we shall either keep zeros or change some zeros to ones. No ones ever get changed to zeros. Example V.6.1 illustrates this process.

Example V.6.1: Get the transitive closure of the relation represented by the digraph below. Use the method described above. Indicate what arcs must be added to this digraph to get the digraph of the transitive closure, and draw the digraph of the transitive closure.



Solution:
$$A = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Walks $\{v_2, v_1, v_2\}$, $\{v_2, v_1, v_4\}$ and $\{v_3, v_1, v_2\}$ have elements of $\{v_1\}$ as interior vertices. Therefore $[W^{[1]}]_{2,2} = 1$, $[W^{[1]}]_{2,4} = 1$, and $[W^{[1]}]_{3,2} = 1$

$$W^{[1]} = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & \underline{1} & 1 & \underline{1} \\ 1 & \underline{1} & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \text{The new "ones" are underlined.}$$

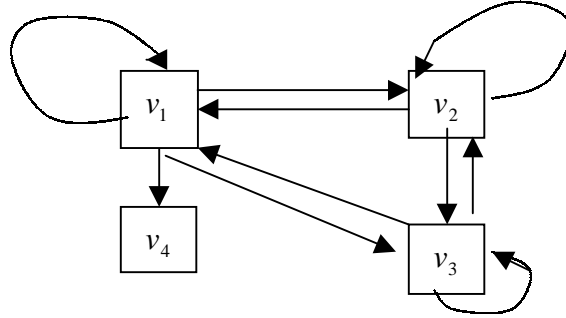
We need only consider walks with v_2 or v_1 and v_2 as interior vertices since walks with v_1 as an interior vertex have already been considered. Walks $\{v_1, v_2, v_3\}$, $\{v_1, v_2, v_1\}$, and $\{v_3, v_1, v_2, v_3\}$ have elements of $\{v_1, v_2\}$ as interior vertices. Therefore $[W^{[2]}]_{1,1} = 1$, $[W^{[2]}]_{1,3} = 1$, and $[W^{[2]}]_{3,3} = 1$. There are other walks with elements of subsets of $\{v_1, v_2\}$ as interior vertices, but they do not contribute any new "ones" to $W^{[2]}$.

$$W^{[2]} = \begin{bmatrix} 0 & 1 & \underline{1} & \underline{1} \\ 1 & 1 & 1 & 1 \\ 1 & \underline{1} & \underline{1} & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \text{The new “ones” are underlined.}$$

There are walks with elements of subsets of $\{v_1, v_2, v_3\}$ as interior vertices. We need only consider walks with v_3 (possibly along with v_1 or v_2 or both) as interior vertices since walks with v_1, v_2 (but not v_3) as interior vertices have already been considered. However, none of these walks create any new “ones” in $W^{[3]}$. We continue this process to obtain $W^{[4]}$. However, any walks we construct with v_4 as an interior vertex contributes no new “ones”. Therefore, $T = W^{[4]} = W^{[3]} = W^{[2]}$. Therefore,

$$T = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

One must add arcs from v_1 to v_1 , v_1 to v_3 , v_2 to v_2 , v_3 to v_2 , and v_3 to v_3 . The graph of the transitive closure is drawn below.



PseudoCode Implementation

No algorithm is practical unless it can be implemented for a large data set. The following version of Warshall’s algorithm is found in Bogart’s text (pp. 470-471). The algorithm immediately follows from definition V.6.4.

Definition V.6.4: If A is an $m \times n$ matrix, then the **Boolean OR operation** of row i and row j is defined as the n -tuple $x = (x_1, x_2, \dots, x_n)$ where each $x_k = a_{ik} \vee a_{jk}$. We do componentwise OR on row i and row j .

Notation: Let a_i and a_j denote the i -th and j -th rows of A , respectively. Then we say $x = a_i \vee a_j$. (Italic x represents an n -tuple)

Algorithm Warshall**Input:** Adjacency matrix A of relation R on a set of n elements**Output:** Adjacency matrix T of the transitive closure of R .**Algorithm Body:** $T := A$ [initialize T to A]**for** $j := 1$ **to** n **for** $i := 1$ **to** n **if** $T_{i,j} = 1$ **then** $a_i := a_i \vee a_j$ [form the Boolean OR of row i and row j , store it in a_i]**next** i **next** j **end Algorithm Warshall****Note:** The matrix T at the end of each iteration of j is the same as $W^{[j]}$ in the digraph implementation of Warshall's algorithm.**Example V.6.2:** Let $A = T = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$ Trace the pseudocode implementation of Warshall's algorithm on A , showing the details of each Boolean OR between rows**Solution:** $j = 1 \quad i = 1 \quad T_{i,j} = 0 \quad \text{no action}$ $i = 2 \quad T_{i,j} = 0 \quad \text{no action}$ $i = 3 \quad T_{i,j} = 0 \quad \text{no action}$ Therefore $W^{[1]} = T = A$ $j = 2 \quad i = 1 \quad T_{i,j} = 1 \quad (0 \ 1 \ 0) \text{ OR } (0 \ 1 \ 1) = (0 \vee 0, 1 \vee 1, 0 \vee 1) = (0 \ 1 \ 1)$ row 1 of T becomes $(0 \ 1 \ 1)$ $i = 2 \quad T_{i,j} = 1 \quad \text{row 2 OR row 2 is computed and put into row 2}$

however, row 2 OR row 2 = row 2

 $i = 3 \quad T_{i,j} = 0 \quad \text{no action}$ At this stage we have $T = W^{[2]} = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$

$j = 3 \quad i = 1 \quad T_{i,j} = 0 \quad \text{no action}$
 $i = 2 \quad T_{i,j} = 1 \quad (0 \ 1 \ 1) \text{ OR } (0 \ 0 \ 0) = (0 \vee 0, 1 \vee 0, 1 \vee 0) = (0 \ 1 \ 1)$
 $\text{result is put into row 2, which is unchanged}$
 $i = 3 \quad T_{i,j} = 0 \quad \text{no action}$

At this stage $T = W^{[3]} = W^{[2]}$ above. We now have the transitive closure.

Exercises:

(1) For each of the adjacency matrices A given below, (a) draw the corresponding digraph and (b) find the matrix T of the transitive closure using the digraph implementation of Warshall's algorithm. Show all work (see example V.6.1). (c) Indicate what arcs must be added to the digraph for A to get the digraph of the transitive closure, and draw the digraph of the transitive closure.

$$(i) \ A = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (ii) \ A = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

$$(iii) \ A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

(2) For the matrix A in example V.6.1, compute all the Boolean OR operations that occur in the pseudocode version of Warshall's algorithm. Write separately the matrices that result from a OR operation in the inner loop. Also convince yourself that the matrix T at the end of each iteration of j is the same as $W^{[j]}$ in the digraph implementation of Warshall's algorithm.

TREES

Tree: It is undirected connected graph that contains no simple circuit.

Branch node (internal node): A vertex of degree larger than 1 is called a branch node or an internal node.

Leaf (Terminal node): A vertex of degree 1 in a tree is called a leaf or terminal node.

Properties of Trees:

- 1) There is a unique path between every two vertices in a tree.
- 2) The number of vertices is one more than the number of edges in a tree. ($E = V - 1$)
- 3) A tree with two or more vertices has at least two leaves.

Rooted Trees: A directed graph is said to be a **directed tree** if it becomes a tree when the directions of the edges are ignored.

A directed tree is called a **rooted tree** if there is exactly one vertex whose incoming degree is 0 is called the **root** of the rooted tree.

In a rooted tree, a vertex whose outgoing degree is nonzero is called branch node or an internal node and a vertex whose outgoing degree is 0 is called a leaf or terminal node.

Different Representations of Tree: There are many occasions when we encounter structures that can be represented as rooted trees.

For examples organizational chart of a corporation in below figure can be represented immediately by a rooted tree.

Let a be a branch node in a rooted tree. A vertex b is said to be a son of a if there is an edge from a to b . Also, a is said to be the father of b . Two vertices are said to be brothers if they are sons of the same vertex. A vertex c is said to be a descendant of a if there is a directed path from a to c . Also, a is said to be an ancestor of c . These terms indeed remind us what we commonly call family trees are indeed rooted trees.

NIP-SPECT

m-ary Tree: A rooted tree is called an m – ary tree if every internal vertex has no more than m - children.

Full m-ary Tree: A rooted tree is called a full m – ary tree if every internal vertex has exactly m - children.

Binary Tree: An m –ary tree with $m = 2$ is called a binary tree.

Converting a m-ary tree (general tree) to a binary tree:

There is a one-to-one mapping between general ordered trees and binary trees. So, every tree can be uniquely represented by a binary tree. Furthermore, a forest can also be represented by a binary tree.

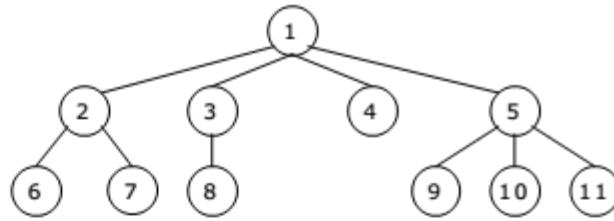
Conversion from general tree to binary can be done in two stages.

Stage 1:

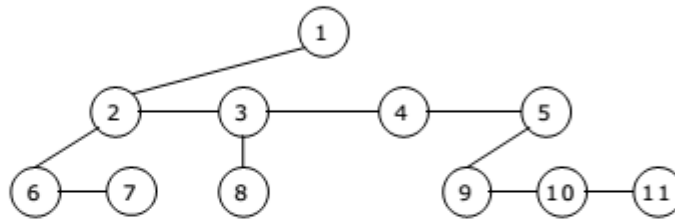
- As a first step, we delete all the branches originating in every node except the left most branch.
- We draw edges from a node to the node on the right, if any, which is situated at the same level.

Stage 2:

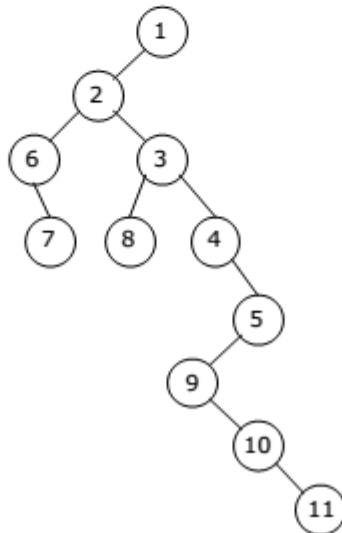
- Once this is done then for any particular node, we choose its left and right sons in the following manner:
- The left son is the node, which is immediately below the given node, and the right son is the node to the immediate right of the given node on the same horizontal line. Such a binary tree will not have a right subtree.
- **Example 1:**
- Convert the following ordered tree into a binary tree:

**Solution:**

Stage 1 tree by using the above mentioned procedure is as follows:

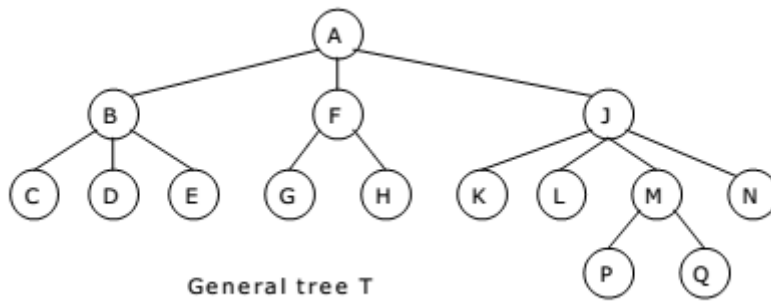


Stage 2 tree by using the above mentioned procedure is as follows:

**Example 2:**

For the general tree shown below:

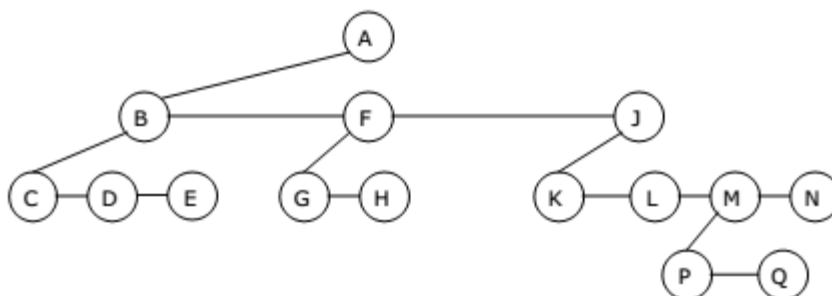
1. Find the corresponding binary tree T' .
2. Find the preorder traversal and the postorder traversal of T .
3. Find the preorder, inorder and postorder traversals of T' .
4. Compare them with the preorder and postorder traversals obtained for T' with the general tree T .



Solution:

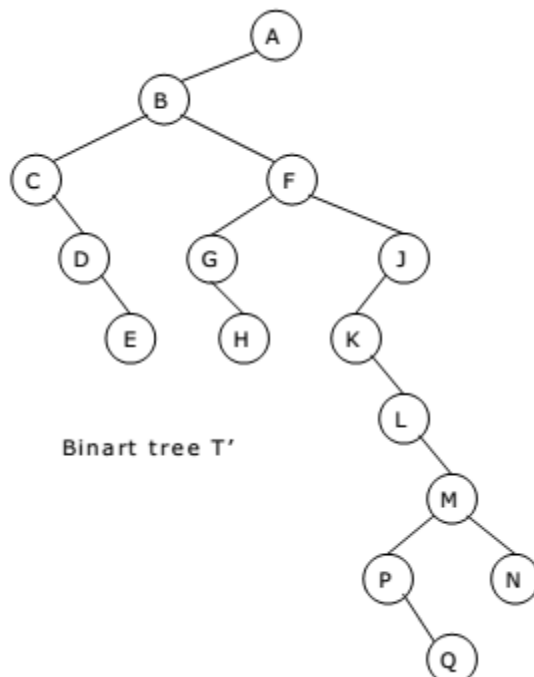
1. Stage 1:

The tree by using the above-mentioned procedure is as follows:



Stage 2:

The binary tree by using the above-mentioned procedure is as follows:



Binary Tree representation:

Representing Binary Tree in memory:

Let **T** be a Binary Tree. There are two ways of representing **T** in the memory as follow

1. **Sequential (Array) Representation of Binary Tree.**
2. **Link Representation of Binary Tree.**

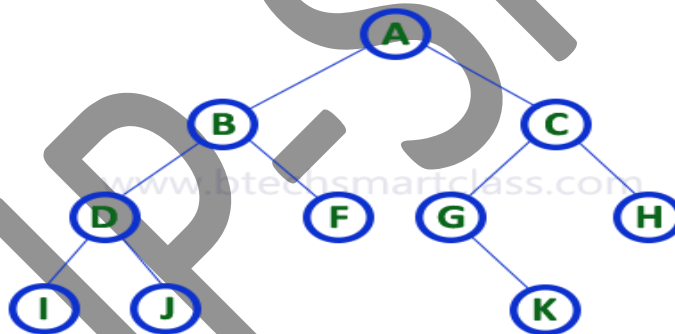
Construct Complete Binary Tree from its Linked List Representation:

Given Linked List Representation of Complete Binary Tree, construct the Binary tree. A complete binary tree can be represented in an array in the following approach.

If root node is stored at index i , its left, and right children are stored at indices $2*i+1$, $2*i+2$ respectively.

Suppose tree is represented by a linked list in same way, how do we convert this into normal linked representation of binary tree where every node has data, left and right pointers? In the linked list representation, we cannot directly access the children of the current node unless we traverse the list.

Consider the following binary tree...



1. Array Representation of Binary Tree

In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.

Consider the above example of a binary tree and it is represented as follows...



To represent a binary tree of depth ' n ' using array representation, we need one dimensional array with a maximum size of $2n + 1$.

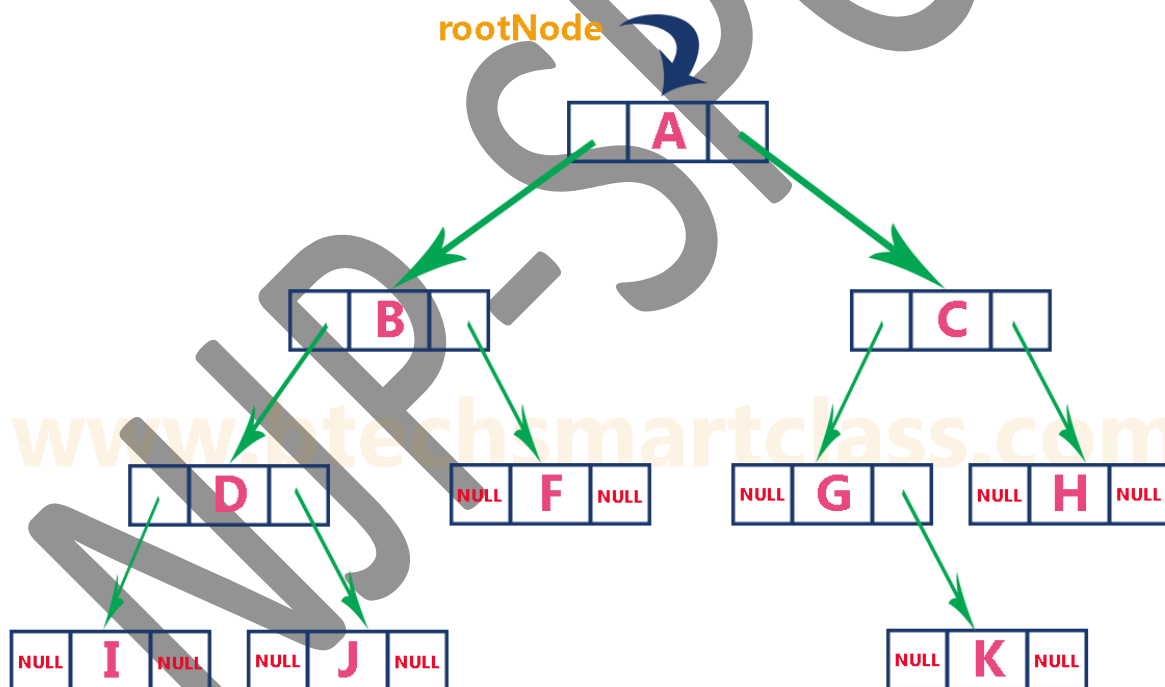
2. Linked List Representation of Binary Tree

We use a double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.

In this linked list representation, a node has the following structure...



The above example of the binary tree represented using Linked list representation is shown as follows...



Binary Tree Traversals:

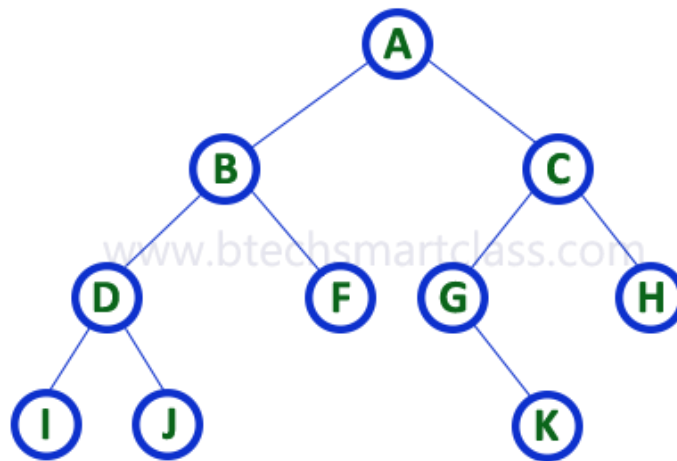
When we wanted to display a binary tree, we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree, displaying order of nodes depends on the traversal method.

Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal.

There are three types of binary tree traversals.

1. In - Order Traversal
2. Pre - Order Traversal
3. Post - Order Traversal

Consider the following binary tree...



1. In - Order Traversal (leftChild - root - rightChild)

In In-Order traversal, the root node is visited between the left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting the right child node. This in-order traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.

In the above example of a binary tree, first we try to visit left child of root node 'A', but A's left child 'B' is a root node for left subtree. so we try to visit its (B's) left child 'D' and again D is a root for subtree with nodes D, I and J. So we try to visit its left child 'I' and it is the leftmost child. So first we visit 'I' then go for its root node 'D' and later we visit D's right child 'J'. With this we have completed the left part of node B. Then visit 'B' and next B's right child 'F' is visited. With this we have completed left part of node A. Then visit root node 'A'. With this we have completed left and root parts of node A. Then we go for the right part of the node A. In right of A again there is a subtree with root C. So go for left child of C and again it is a subtree with root G. But G does not have left part so we visit 'G' and then visit G's right child K. With this we have completed the left part of node C. Then visit root node 'C' and next visit C's right child 'H' which is the rightmost child in the tree. So we stop the process.

That means here we have visited in the order of **I - D - J - B - F - A - G - K - C - H** using In-Order Traversal.

In-Order Traversal for above example of binary tree is

Discrete Mathematics(3140708) Sem:4 Branch: IT/COMP
I - D - J - B - F - A - G - K - C - H

2. Pre - Order Traversal (root - leftChild - rightChild)

In Pre-Order traversal, the root node is visited before the left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all subtrees in the tree.

In the above example of binary tree, first we visit root node 'A' then visit its left child 'B' which is a root for D and F. So we visit B's left child 'D' and again D is a root for I and J. So we visit D's left child 'I' which is the leftmost child. So next we go for visiting D's right child 'J'. With this we have completed root, left and right parts of node D and root, left parts of node B. Next visit B's right child 'F'. With this we have completed root and left parts of node A. So we go for A's right child 'C' which is a root node for G and H. After visiting C, we go for its left child 'G' which is a root for node K. So next we visit left of G, but it does not have left child so we go for G's right child 'K'. With this, we have completed node C's root and left parts. Next visit C's right child 'H' which is the rightmost child in the tree. So we stop the process.

That means here we have visited in the order of **A-B-D-I-J-F-C-G-K-H** using Pre-Order Traversal.

Pre-Order Traversal for above example binary tree is

A - B - D - I - J - F - C - G - K - H

3. Post - Order Traversal (leftChild - rightChild - root)

In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.

Here we have visited in the order of **I - J - D - F - B - K - G - H - C - A** using Post-Order Traversal.

Post-Order Traversal for above example binary tree is

I - J - D - F - B - K - G - H - C - A

Applications of linked list in computer science –

1. Implementation of stacks and queues
2. Implementation of graphs : Adjacency list representation of graphs is most popular which is uses linked list to store adjacent vertices.
3. Dynamic memory allocation : We use linked list of free blocks.
4. Maintaining directory of names
5. Performing arithmetic operations on long integers
6. Manipulation of polynomials by storing constants in the node of linked list
7. representing sparse matrices

Applications of linked list in real world-

8. Image viewer – Previous and next images are linked, hence can be accessed by next and previous button.
9. Previous and next page in web browser – We can access previous and next URL searched in web browser by pressing back and next button since, they are linked as linked list.
10. Music Player – Songs in music player are linked to previous and next song. you can play songs either from starting or ending of the list.

NIP.SPCE