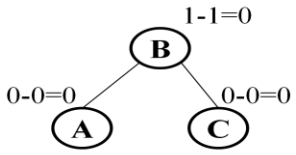
Module II

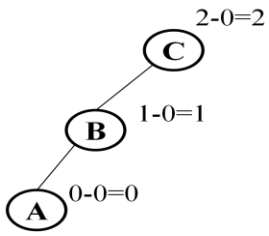
- **Advanced Data Structures and Graph Algorithms**
  - **Self Balancing Tree**
    - **AVL Trees : Insertion and deletion operations with all rotations in detail**
  - **Disjoint Sets**
    - **Disjoint set operations**
    - **Union and find algorithms**
  - **DFS and BFS traversals - Analysis**
  - **Strongly Connected Components of a Directed graph**
  - **Topological Sorting**

- **AVL Trees**
  - AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.
  - AVL Tree can be defined as **height balanced binary search tree** in which each node is associated with a balance factor.

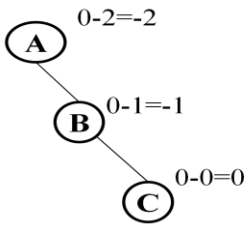
  - **Balance Factor**
    1. Balance Factor of a node = height of left subtree – height of right subtree
    2. In an AVL tree balance factor of every node is -1,0 or +1
    3. Otherwise the tree will be unbalanced and need to be balanced.



Tree is Balanced.

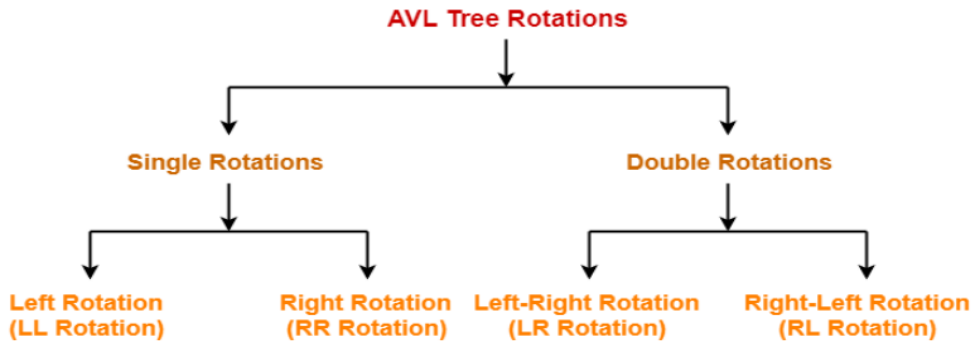


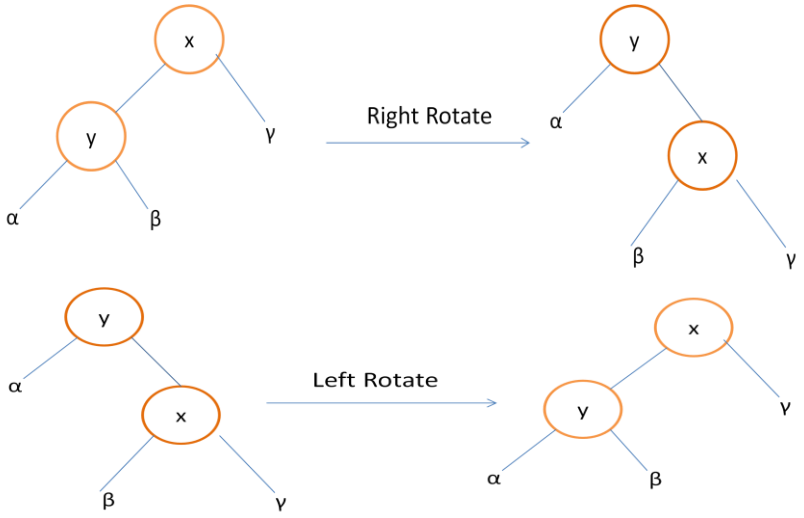
Tree is not Balanced.
Balance factor of C is 2



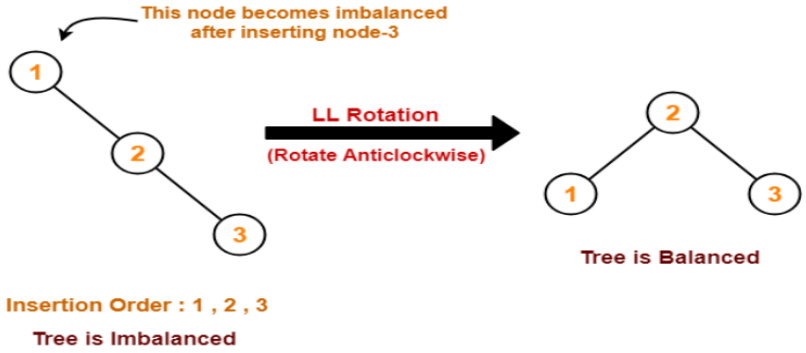
Tree is not Balanced.
Balance factor of A is -2

  - **Why AVL Tree?**
    1. Most of the Binary Search Tree(BST) operations (eg: search, insertion, deletion etc) take O(h) time where h is the height of the BST.
    2. The minimum height of the BST is log n
    3. The height of an AVL tree is always O(log n) where n is the number of nodes in the tree.
    4. So the time complexity of all AVL tree operations are O(log n)
  - An AVL tree becomes imbalanced due to some insertion or deletion operations
  - We use rotation operation to make the tree balanced.
  - There are 4 types of rotations





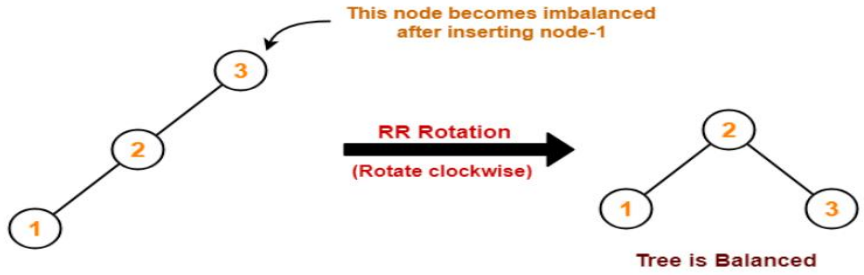
- **Single Left Rotation(LL Rotation)**
  - In LL rotation every node moves one position to left from the current position



Insertion Order : 1 , 2 , 3

Tree is Imbalanced

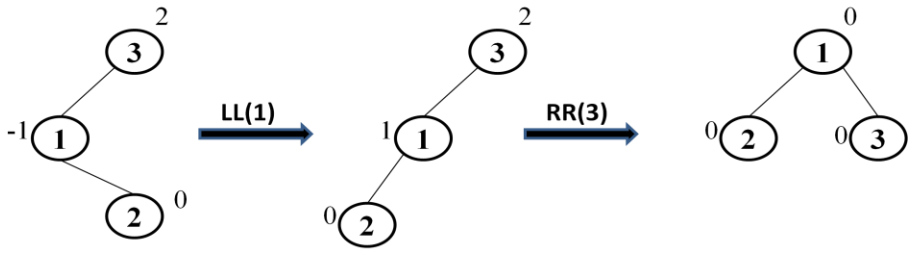
- **Single Right Rotation(RR Rotation)**
  - In RR rotation every node moves one position to right from the current position



Insertion Order : 3 , 2 , 1

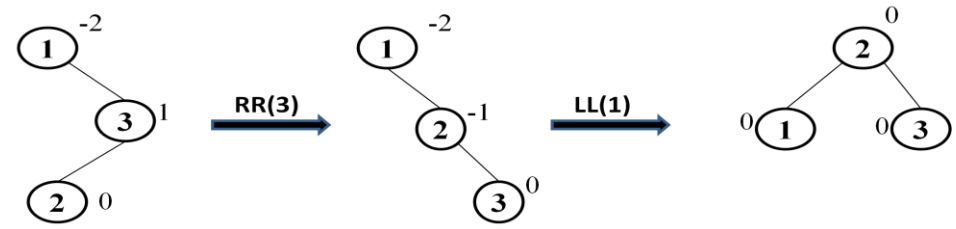
Tree is Imbalanced

- **Left-Right Rotation(LR Rotation)**
  - The LR rotation is the combination of single left rotation followed by single right rotation.



Insertion Order: 3,1,2

- **Right-Left Rotation(RL Rotation)**
  - The RL rotation is the combination of single right rotation followed by single left rotation.



Insertion Order: 1,3,2

- **AVL Tree Insertion Algorithm**
  1. Insert the node as the leaf node. Use BST insertion procedure
  2. After insertion check the balance factor of every node
  3. If the tree is imbalanced, perform the suitable rotation. Let z be the newly inserted node. x be the first unbalanced node on the path from z to root. y be the child of x on the path from z to root
     - If y is the left child of x and z is in the left subtree of y, then perform RR Rotation with respect to x.
     - If y is the right child of x and z is in the right subtree of y, then perform LL Rotation with respect to x.
     - If y is the left child of x and z is in the right subtree of y, then perform LR Rotation
     - If y is the right child of x and z is in the left subtree of y, then perform RL Rotation

- Complexity of AVL tree insertion = **O(log n)**
  Where log n is the height of the tree.

- **Examples:**
  1. Insert 14,17,11,7,53,4 and 13 in to an empty AVL tree
  2. Insert numbers from 1 to 8 into an empty AVL tree
  3. Insert 10,20,15,25,30,16,18 and 19 in to an empty AVL tree
  4. Find minimum and maximum height of any AVL tree with 7 nodes? Assume that the height of a tree with a single node is 0.
  5. Find the minimum and maximum height of any AVL-tree with 11nodes. Assume that height of the root is 0.

- **AVL Tree Deletion Algorithm**

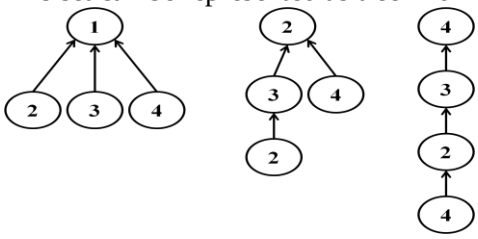
Let w be the node to be deleted
1. Delete w using BST deletion procedure
2. Starting from w, travel up and find the first unbalanced node. Let x be the first unbalanced node.
3. If balance factor(x)>1 then y=leftchild(x)
4. Else y=rightchild(x)
5. If y is the leftchild of x
   1. If balance factor(y)≥ 0 then z=leftchild(y)
   2. Else z=rightchild(y)
6. Else
   1. If balance factor(y) ≤ 0 then z=rightchild(y)
   2. Else z=leftchild(y)
7. If y is the left child of x and z is the left child of y, then perform **RR Rotation** with respect to x
8. If y is the right child of x and z is the right child of y, then perform **LL Rotation** with respect to x

9. If y is the left child of x and z is the right child of y, then perform **LR Rotation** with respect to x
10. If y is the right child of x and z is the left child of y, then perform **RL Rotation** with respect to x

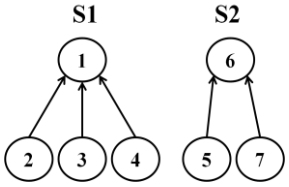
- Complexity of AVL tree deletion = **O(log n)**
  Where log n is the height of the tree.

- **Disjoint Sets**
  - Two or more sets with nothing in common are called disjoint sets.
  - Example: S1 = {1, 2, 3, 4} S2 = {5, 6, 7} S3 = {8, 9}
  - Two sets S1 and S2 are said to be disjoint if S1-S2= ϕ
  - The disjoint set data structure is also known as union-find data structure and merge-find set.

  - **Disjoint set representations**
    - **Linked list representation(Tree representation)**
      - Example: S1 = {1, 2, 3, 4}
        This set can be represented as tree in different ways



      - Example: S1 = {1, 2, 3, 4} S2 = {5, 6, 7}



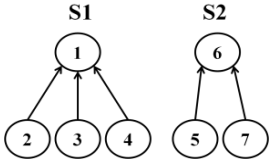
    - **Array representation**
      - Example: S1 = {1, 2, 3, 4} S2 = {5, 6, 7}

i	1	2	3	4	5	6	7
p	-1	1	1	1	6	-1	6

  - **Disjoint Set operations**
    - Make set
    - Union
    - Find

  - **Make Set Operation**
    - Make-set(x) function will create a set that containing one element x.

  - **Find Operation**
    - Determine which subset a particular element is in.
    - This can be used for determining if two elements are in the same subset.

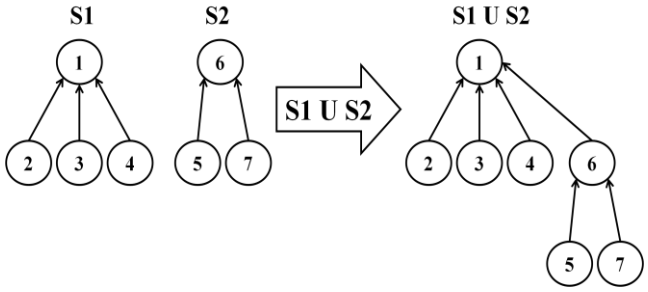


Find(3) will return 1, which is the root of the tree that 3 belongs
Find(6) will return 6, which is the root of the tree that 6 belongs

- **Find Algorithm**

```
Algorithm Find(i)
{
        while p[i] > 0 do
                i = p[i]
        return i
}
```

- **Union Operation**
  - Join two subsets into a single subset.
  - Here first we have to check if the two subsets belong to same set. If no, then we cannot perform union



i	1	2	3	4	5	6	7
P	-1	1	1	1	6	1	6

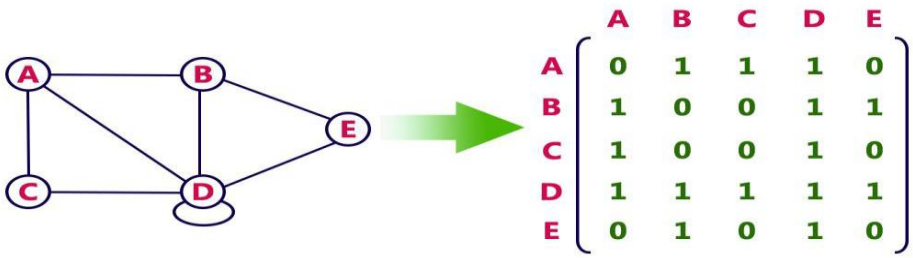
- **Union Algorithm**

```
Algorithm Union(i, j)
{
        X =Find(i)
        Y = Find(j)
        If X != Y then
                p[Y] = X
}
```

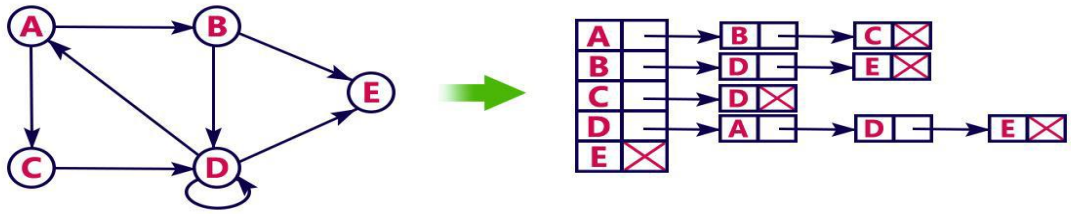
- **Applications of disjoint sets**
  - It is easier to check whether the given two elements are belongs to the same set.
  - Used to merge 2 sets into one

- **Graphs**
  - Graph is a data structure that consists of following two components:
    - A finite set of vertices (nodes).
    - A finite set of edge(ordered pair of the form (u, v))
  - A graph **G = (V , E)**, where **V is a set of vertices** and **E is a set of edges**.
  - Representations of graph.
    - Adjacency Matrix
    - Adjacency List
  - **Adjacency Matrix:**
    - Adjacency Matrix is a 2D array(say adj[][]) of size |V| x |V| where |V| is the number of vertices in a graph.
    - If adj[i][j] = 1, then there is an edge from vertex i to vertex j.
    - Adjacency matrix for undirected graph is always symmetric.
    - Adjacency Matrix is also used to represent weighted graphs.

- If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.



- **Adjacency List:**
  - An array of linked lists is used.
  - Size of the array is equal to number of vertices.
  - An entry array[i] represents the linked list of vertices adjacent to the **ith** vertex.
  - This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists.



- **Types of graph:**
  - Undirected Graph: A graph with only undirected edges.
  - Directed Graph: A graph with only directed edges.
  - Directed Acyclic Graphs(DAG): A directed graph with no cycles.
  - Cyclic Graph: A directed graph with at least one cycle.
  - Weighted Graph: It is a graph in which each edge is given a numerical weight.
  - Disconnected Graphs: An undirected graph that is not connected.

- **Graph Traversal Algorithms**:
  - Graph traversal algorithms visit the vertices of a graph, according to some strategy. Different graph traversal algorithms are:
    - Breadth First Search(BFS)
    - Depth First Search(DFS)

  - **Breadth First Search(BFS)**
    - **Algorithm BFS(G, u)**
      1. Set all nodes are unvisited
      2. Mark the starting vertex u as visited and put it into an empty Queue Q
      3. While Q is not empty
         3.1 Dequeue v from Q
         3.2 While v has an unvisited neighbor w
             3.2.1 Mark w as visited
             3.2.2 Enqueue w into Q
      4. If there is any unvisited node x
         4.1 Visit x and Insert it into Q. Goto step 3

- **Complexity**
  - If the graph is represented as an adjacency list
    - Each vertex is enqueued and dequeued atmost once. Each queue operation take O(1) time. So the time devoted to the queue operation is **O(V).**
    - The adjacency list of each vertex is scanned only when the vertex is dequeued. Each adjacency list is scanned atmost once. Sum of the lengths of all adjacency list is |E|. Total time spend in scanning adjacency list is **O(E).**
    - Time complexity of BFS = O(V) + O(E) = **O(V+ E).**
    - **In a dense graph:**
      - $E=O(V^2)$
      - Time complexity= $O(V) + O(V^2) = \mathbf{O(V^2)}$
  - If the graph is represented as an adjacency matrix
    - There are V^2 entries in the adjacency matrix. Each entry is checked once.
    - Time complexity of BFS = $\mathbf{O(V^2)}$

- **Applications of BFS**
  - Finding shortest path between 2 nodes u and v, with path length measured by number of edges
  - Testing graph for bipartiteness
  - Minimum spanning tree for unweighted graph
  - Finding nodes in any connected component of a graph
  - Serialization/deserialization of a binary tree
  - Finding nodes in any connected component of a graph

- **Depth First Search(DFS)**

Algorithm DFS(G, u)

1. Mark vertex u as visited
2. For each adjacent vertex v of u
   2.1 if v is not visited
      2.1.1 DFS(G, v)

Algorithm main(G,u)

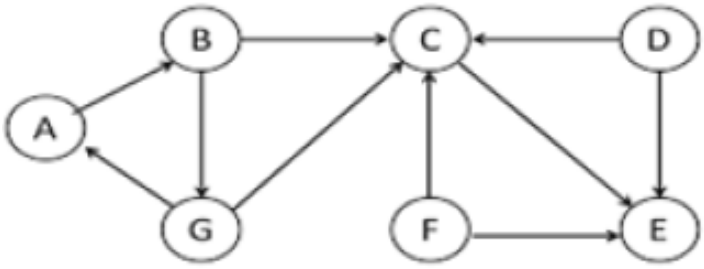
1. Set all nodes are unvisited.
2. DFS(G, u)
3. For any node x which is not yet visited
   3.1 DFS(G, x)

- **Complexity**
  - If the graph is represented as an adjacency list
    - Each vertex is visited atmost once. So the time devoted is **O(V)**
    - Each adjacency list is scanned atmost once. So the time devoted is **O(E)**
    - Time complexity of DFS = **O(V + E)**.
  - If the graph is represented as an adjacency matrix
    - There are V^2 entries in the adjacency matrix. Each entry is checked once.
    - Time complexity of DFS = $\mathbf{O(V^2)}$

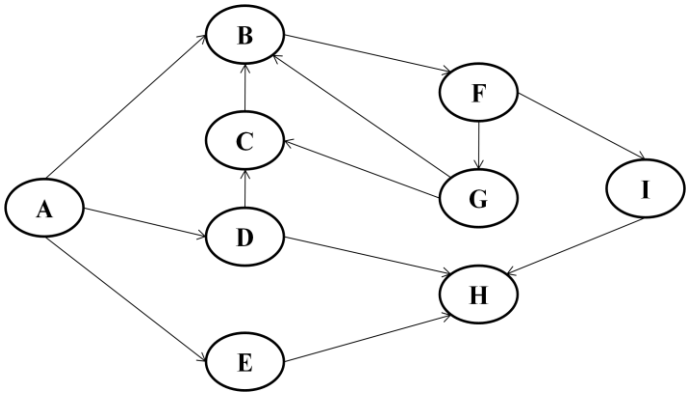
- **Applications of DFS**
  - Finding connected components in a graph
  - Topological sorting in a DAG
  - Scheduling problems
  - Cycle detection in graphs
  - Finding 2-(edge or vertex)-connected components

- Finding 3-(edge or vertex)-connected components
- Finding the bridges of a graph
- Finding strongly connected components
- Solving puzzles with only one solution, such as mazes
- Finding biconnectivity in graphs

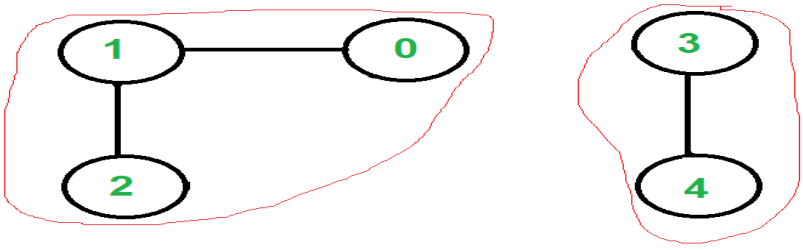
- **Examples:**
  1. Perform BFS traversal on the below graph starting from node A. If multiple node choices may be available for next travel, choose the next node in alphabetical order



  2. Perform DFS traversal on the bellow graph starting from node A. Where multiple node choices may be available for next travel, choose the next node in alphabetical order.

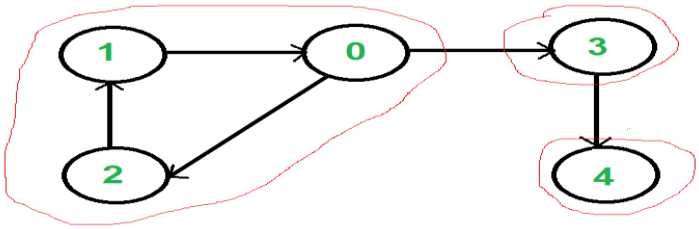


- **Connected Components:**
  - Connected component of a graph *G* is a connected subgraph of *G* of maximum size
  - A graph may have more than one connected components.



There are two connected components in above undirected graph
0 1 2
3 4

- **Strongly Connected Components(SCC)**
  - Strong Connectivity applies only to directed graphs.
  - A strongly connected component of a directed graph is a compliment such that all the vertices in that component is reachable from every other vertex in that component.



Here we have 3 SCCs. {0,1,2}, {3}, {4}

- **Kosaraju's Algorithm**
  - It is a 2 Pass algorithm. Steps 1-4 are Pass1. Steps 5-7 are Pass2.

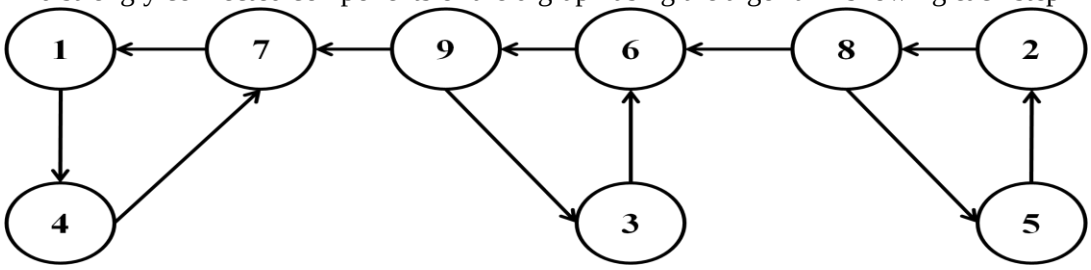
1. Set all vertices of graph G are unvisited.
2. Create an empty stack S.
3. Do DFS traversal on unvisited vertices and set it as visited. If a vertex has no unvisited neighbor, push it in to the stack.
4. Perform the above step until all vertices are visited
5. Reverse the graph G.
6. Set all nodes are unvisited.
7. While S is not Empty
   - 7.1 POP one vertex v'
   - 7.2 If v' is not visited
     - 7.2.1 Set v' as visited
     - 7.2.2 Call DFS(v'). It will print strongly connected component of v'.

- **Time Complexity**
  - First pass we did DFS. So the time complexity = **O(V+E)**
  - Reversal of a graph will take **O(V+E)** time
  - Pass 2 will take another **O(V+E)** time
  - Total time complexity = **O(V+E)**

- **Applications**
  - In social networks, a group of people are generally strongly connected (For example, students of a class or any other common place). Many people in these groups generally like some common pages or play common games. The SCC algorithms can be used to find such groups and suggest the commonly liked pages or games to the people in the group.

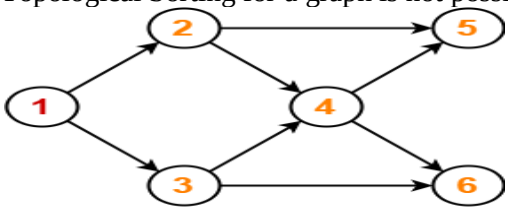
- **Example**

1. Find strongly connected components of the digraph using the algorithm showing each step



- **Topological Sorting**
  - Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge (u,v), vertex u comes before v in the ordering.
  - A topological sort of a graph is an ordering of its vertices along a horizontal line so that all directed edges go from left to right.
  - If the graph contains a cycle, then no linear ordering is possible.

- Topological Sorting for a graph is not possible if the graph is not a DAG.



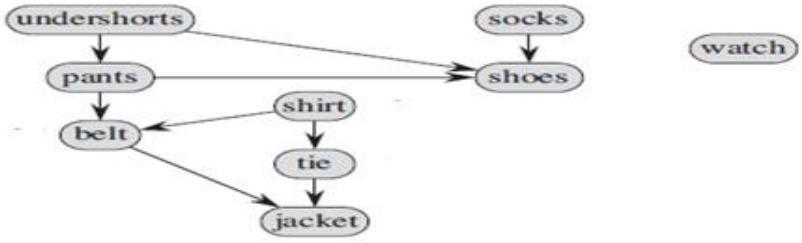
Topological Sort Example One possible Topological Sort=[1,2,3,4,5,6]

- **Algorithm**
  1. Identify a node with no incoming edges(indegree=0)
  2. Add this node to the ordering.
  3. Remove this node and all its outgoing edges from the graph.
  4. Repeat step 1 to 3 until the graph becomes empty

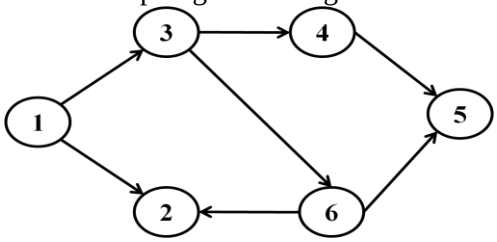
- **Complexity**
  - Suppose |E| is the number of edges and |V| is the number of nodes of the graph G.
  - Time to determine the indegree for each node = **O(E)** time. This involves looking at each directed edge in the graph once.
  - Time to determine the nodes with no incoming edges = **O(V)** time
  - Add nodes until we run out of nodes with no incoming edges. This loop could run once for every node—**O(V)** times
    - Constant-time operations to add a node to the topological ordering.
    - Decrement the indegree for each neighbor of the node we added. Over the entire algorithm, we'll end up doing exactly one decrement for each edge, making this step **O(E)** time.
  - Check if we included all nodes or found a cycle. This is a fast **O(1)** comparison
  - All together, the time complexity is **O(V+E)**

- **Applications**
  - Scheduling jobs from the given dependencies among jobs
  - Instruction Scheduling
  - Determining the order of compilation tasks to perform in makefiles
  - Data Serialization

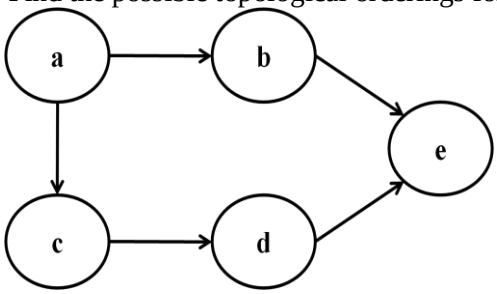
- **Examples**
  1. Write the topological sorting for the DAG given below



  2. Write the topological sorting for the DAG given below



3. Find the possible topological orderings for the following graph



Protect pdf from copying with Online-PDF-No-Copy.com