Algorithm-Analysis-Module-2-Important-Topics

> ⓘ ***For more notes visit***
>
> ***https://rtpnotes.vercel.app***

- Algorithm-Analysis-Module-2-Important-Topics
  - 1. AVL Trees
    - What is an AVL Tree?
    - Calculating Balancing Factor
    - Balancing AVL Tree
      - Left Rotation
      - Right Rotation
      - Left-RIght Rotation (LR Rotation)
      - Right-Left Rotation
    - Deletion in AVL Trees
      - Different cases in deletion
      - Case 1: Delete Leaf Node Example
      - Case 2: Delete a node with one child
      - Case 3: Delete an internal node (Having both left and right subtree)
    - Height of an AVL Tree
      - Problem
  - 2. Breadth First Search (BFS)
    - Algorithm BFS(G,S)
    - Analysis
    - Example
    - Applications
  - 3. Depth First Search
    - Algorithm
      - Algorithm DFS(g)

        - Algorithm DFSVISIT(G,u)
    - Analysis
    - Example
    - Applications
- 4. Strongly connected components
    - What are Connected Components?
    - What are strongly connected components?
    - Example
    - Strongly Connected Components - Complexity
    - Applications
- 5. Topological sorting
    - Topological sorting algorithm
    - Example
    - Topological sorting complexity

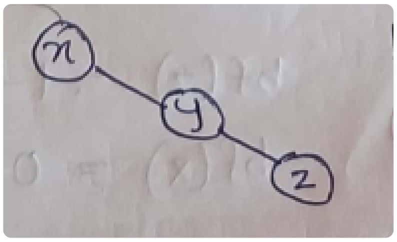
1. AVL Trees

What is an AVL Tree?

- Its a binary search tree
- Each node in the tree has a balancing factor {-1,0,1}
- The balancing factor has an equation
    - bf(node) = height of left subtree - height of right subtree

Calculating Balancing Factor

Consider this graph



- **The balancing Factor for Z**

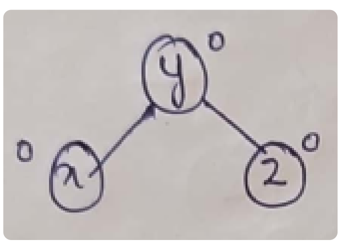
    - There is no left or right subtree for Z
    - So its 0-0 = 0
- **Balancing factor for Y**
    - No of Left subtree = 0
    - Right subtree height = 1
    - 0 - 1 = -1
- **Balancing factor for X**
    - No left subtree
    - Right subtree height = 2
    - 0 - 2 = -2
- Since balancing factor (x) is -2, **Its not a balanced AVL tree**

Balancing AVL Tree

We want to balance this graph, The method that we will use is Rotation.

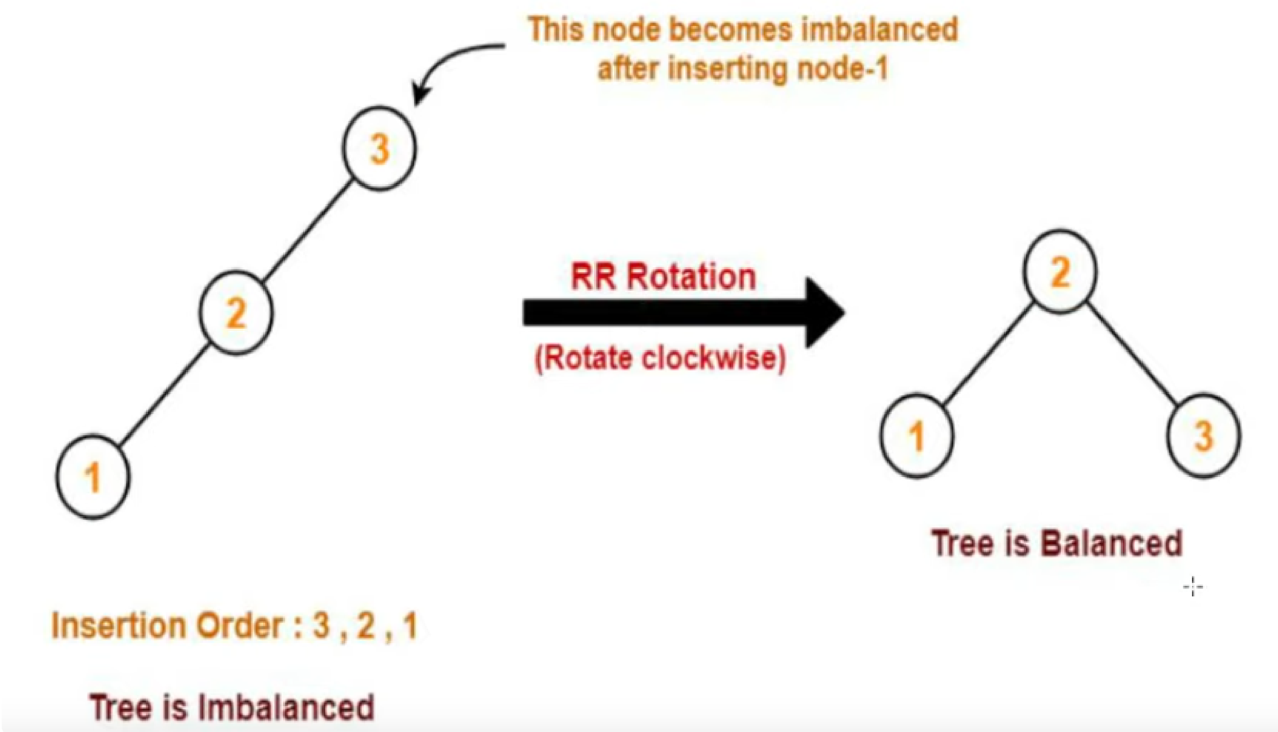
There are 4 types of Rotation

Left Rotation

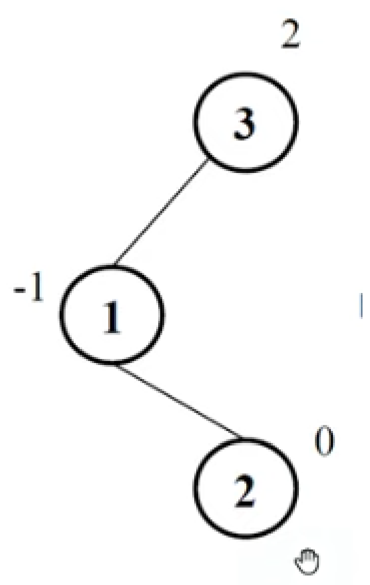
- In this rotation, median will become root. Here our median in y
- Unbalanced will become left child
- Here we are trying to balance x, So x is unbalanced
- We get the graph as
- 

Right Rotation

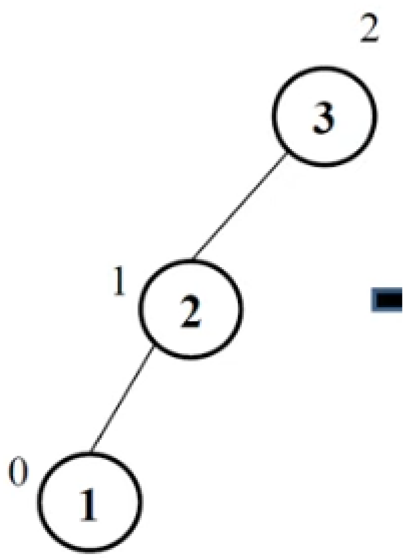
- In this rotation, Median will become root
- Unbalanced will become right child
- Here 3 is unbalanced, because its balancing factor = left - right = 2 - 0 = 2
- Rotating it to the right, The unbalanced Z will become the right child

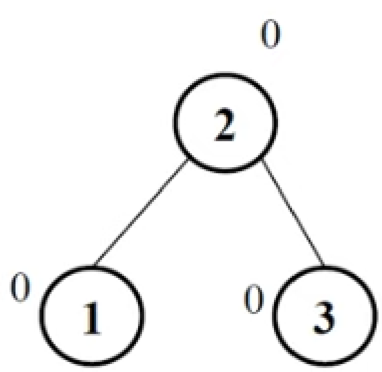
- 

Left-RIght Rotation (LR Rotation)



- Checking balancing factor for each vertex
  - 2
    - left - right = 0
  - 1
    - left - right = 0 - 1 = -1
  - 3
    - left - right = 2 - 0 = 2

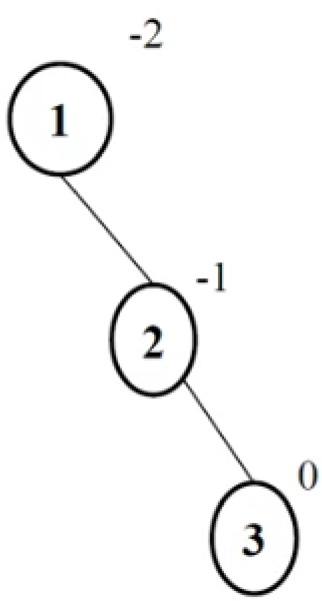
    - **3 is unbalanced, to balance it first straighten it**
- **Do LL(1) (Left Rotate 1)**
    - Applying left rotation
    - Unbalanced will become left child
    - 1 is left child and 2 is parent
    - 

- Here now 3 is unbalanced
    - Doing right rotation on 3
    - 

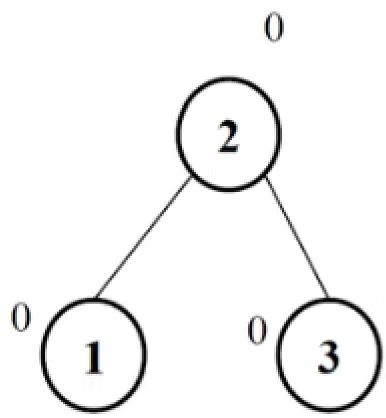
Here first we did a Left Rotation, and then a Right rotation, This is called LR Rotation

Right-Left Rotation

- **Straightening the graph, We have to Bring 2 to the right, so we do**
  - Right Rotate(3)
  - Unbalanced becomes the right child, Here its 3
  - 2 will be the parent



  -
- **Now we have to balance 1, Doing Left Rotate(1)**
  - Unbalanced vertex will be the left child, Here its 1



Deletion in AVL Trees

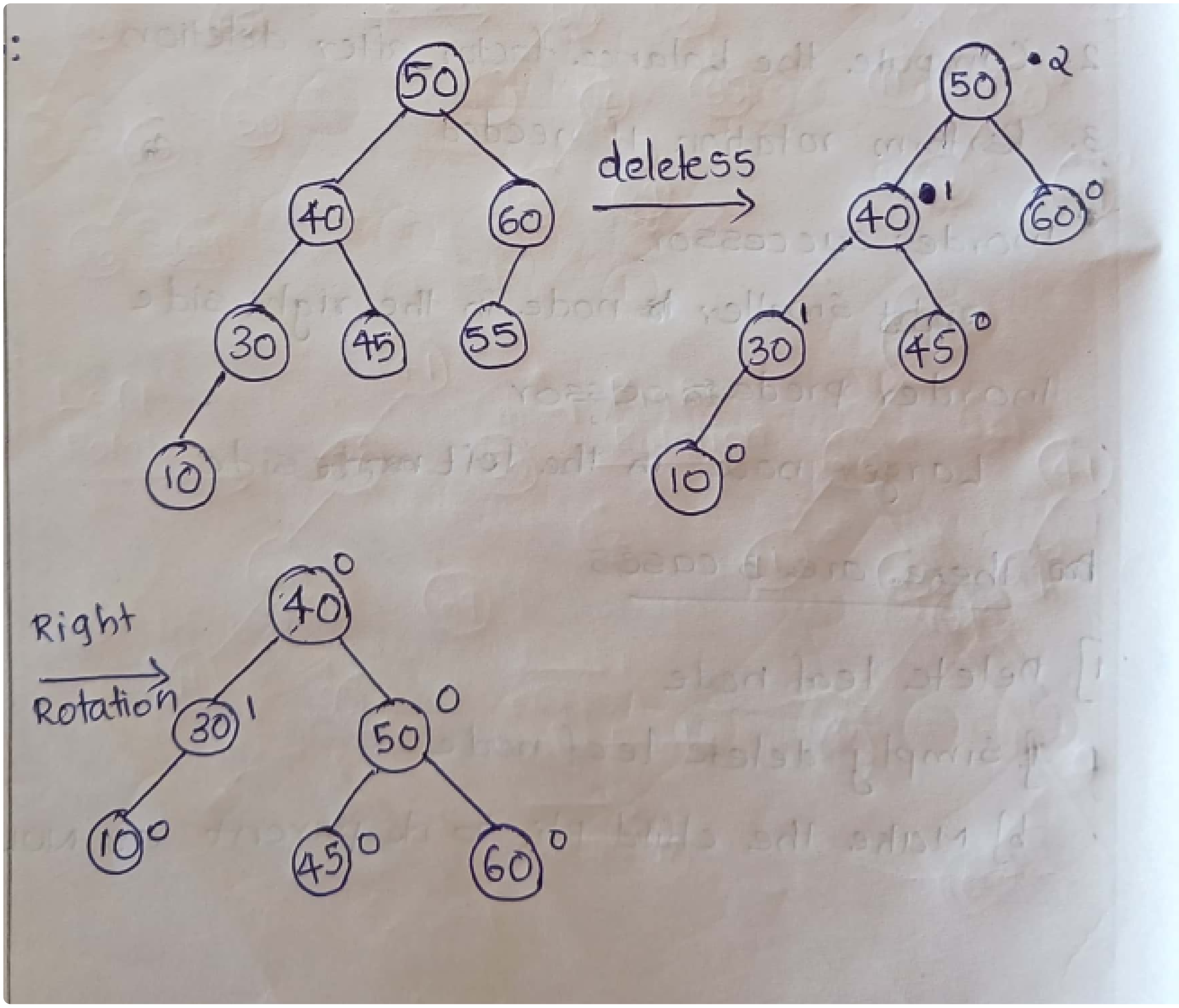
- Perform deletion as in binary search tree
- Compute the balance factor after deletion
- Perform rotation if needed
- Inorder successor
  - Smaller node in right side
- Inorder predecessor
  - Larger node in left side

Different cases in deletion

1. **Delete Leaf Node**
   1. Simply delete leaf node
   2. Make the child ptr to its parent null
2. **Delete a node with one child**
   1. Swap the values of child and parent node
   2. Delete the leaf node
3. **Delete an internal node (Having both left and right subtree)**
   1. Replace node with inorder predecessor or successor
   2. Check whether resulting case is case 1 or 2

Case 1: Delete Leaf Node Example

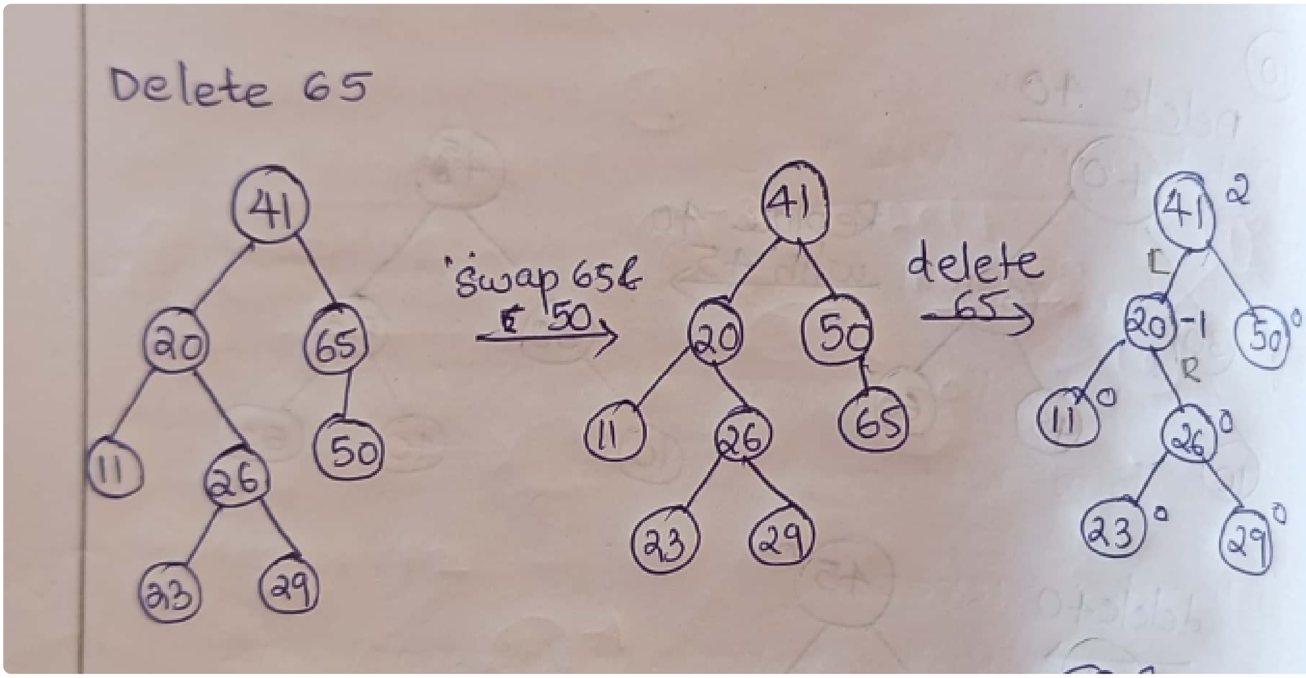
Delete 55



- Here 55 is a leaf node, so simply deleting it
- Calculating the balance factor
  - 50
    - 3 - 1 = 2
  - 60
    - 0 - 0 = 0
  - 40 = 1
- Here 50 is unbalanced
- We need more nodes to the right, so performing Right rotation RR(50)
  - Taking 40 as median
  - Taking unbalanced as right child, here 50 is unbalanced
  - Putting 45 in the appropriate place

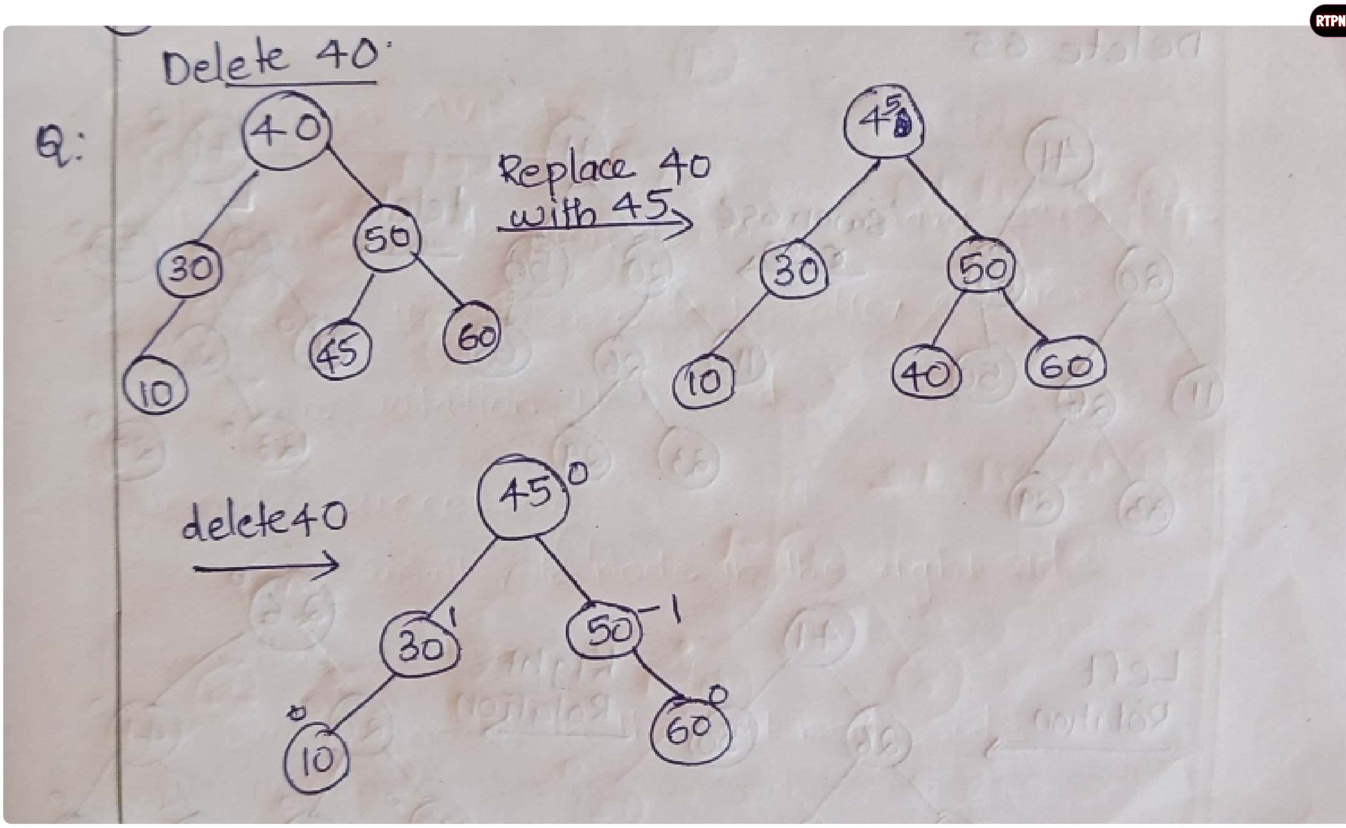
Case 2: Delete a node with one child

Delete 65



- **Swap 65 and 50**
- **Delete 65**

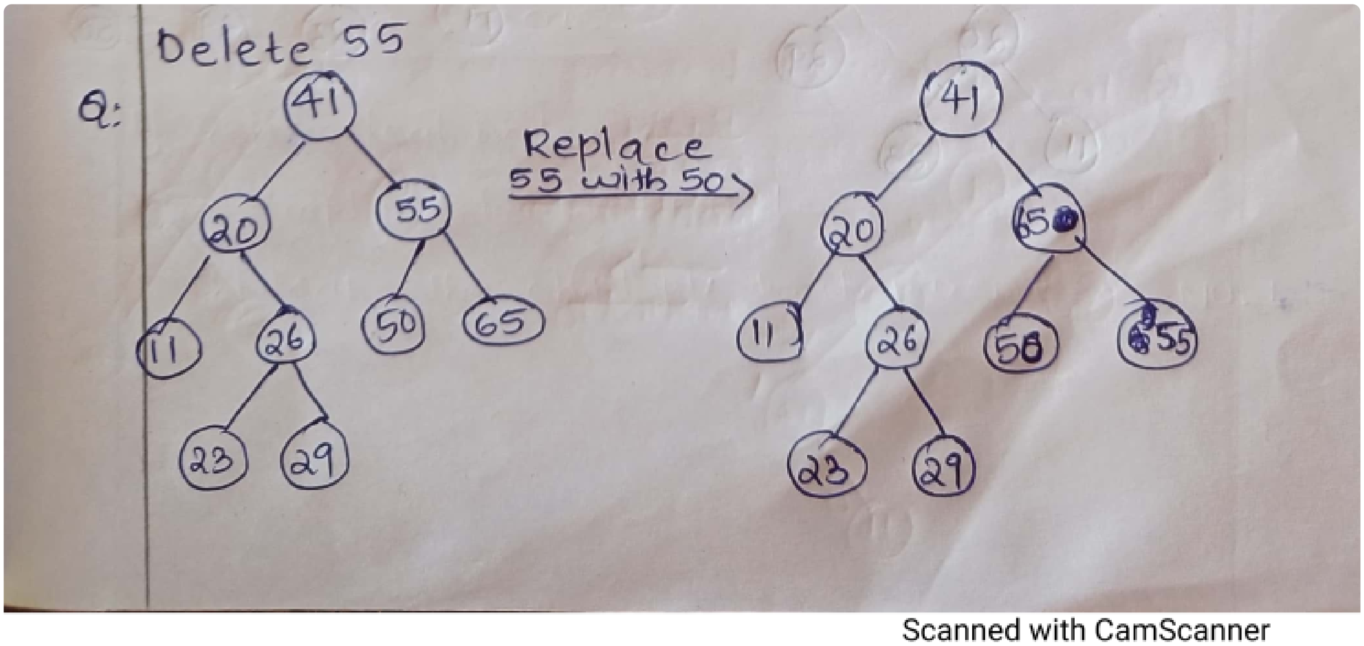
Case 3: Delete an internal node (Having both left and right subtree)

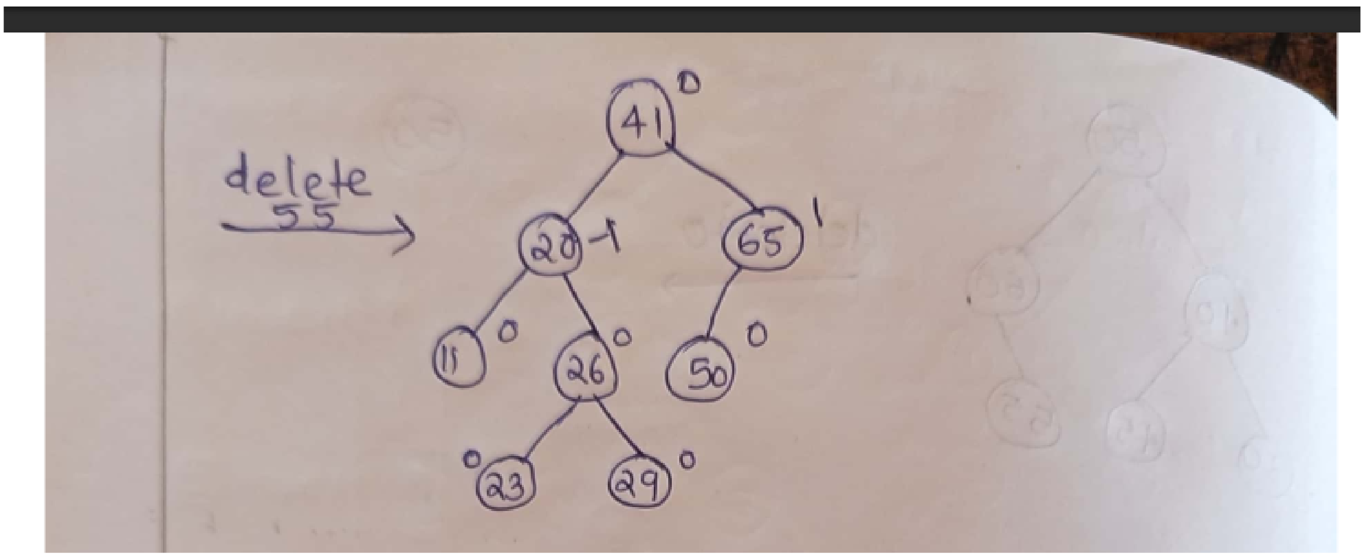


Delete 40

- **40 has both left and right subtree**
- **Inorder successor of 40**
  - Inorder successor of 40, is the smallest node which is greater than 40,
  - Here it is 45
- **Replace 40 with 45**
- **Delete 40**
- Check balance
  - All are balanced

Delete 55





- **Here 55 has 2 subtrees**
- **Inorder successor of 55 is 65**
- **Swapping 65 and 50**
- **Deleting 55**
- The tree is balanced

Height of an AVL Tree

- Let $N(h)$ denotes the minimum number of nodes in an AVL Tree of height h
- if $h = 0, N(0) = 1$
- $h = 1, N(1) = 2$

- $N(h) = N(h-1) + N(h-2) + 1$
- Minimum height = $\lfloor \log_2 n \rfloor$

Problem

Q What is the maximum height of any AVL tree with 7 nodes

- Given Nodes = N(h) = 7
- We need to find h
- Minimum height = $\lfloor log_2 n \rfloor$
- Log 7 = 2

---

2. Breadth First Search (BFS)

Algorithm BFS(G,S)

1. For each vertex u in V, except Source
    1. color[u] <- white
    2. d[u] <- ∞
    3. predecessor[u] <- Nil
2. color[s] = gray # setting color of source as gray
3. d[s] <- 0
4. predecessor[s] = Nil
5. Q <- φ
6. Enqueue(Q,S)
7. While Q != φ
    1. u <- Dequeue(Q)
    2. for each V ∈ Adjacent[u]
        1. if color[v] = WHITE do
            1. color[v] = GRAY
            2. d[v] <- d[u] + 1
            3. predecessor[v] <- u
            4. ENQUEUE(Q,V)

8. color[u] <- BLACK

- Here we basically take all the vertices, set their color to white, set their distance to infinity and their predecessor to nil
- We mark color gray, distance as 0 and predecessor as nil for the Source node
- We initialize the queue as null
- We start by enqueueing the source vertex to the queue
- While the queue is not empty, we need to dequeue the queue and store it it variable u
- Now We will take each of u's adjacent vertices one by one
    - If their color is white, change it to gray, update distance and predecessor and Enqueue that node
- After all vertices are processed, Change the color of vertex to black, indicating the vertex is completely processed
- What the colors mean
    - White for unvisited vertices.
    - Gray for vertices that are currently in the queue and are being processed.
    - Black for vertices that have been fully processed.

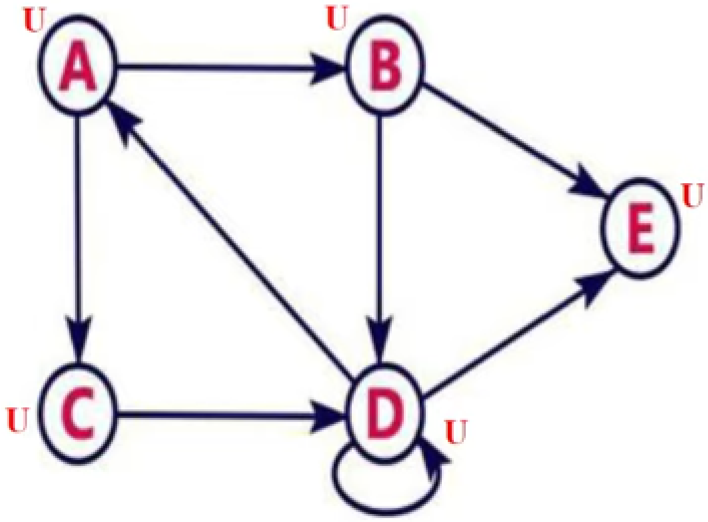
Analysis

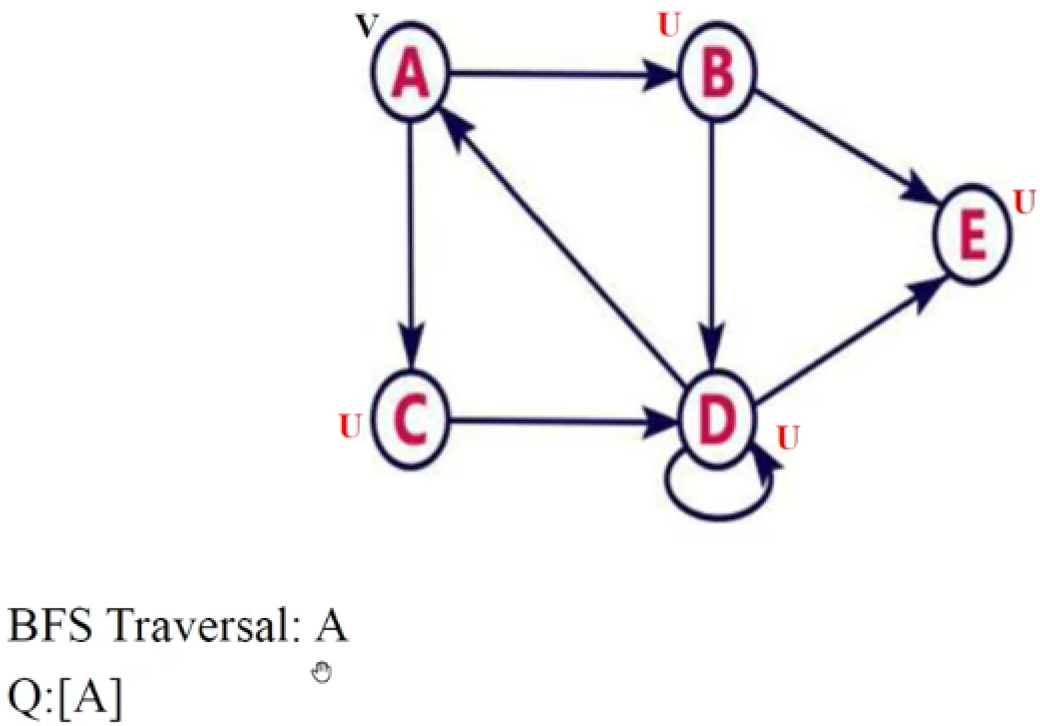
If the graph is represented as an adjacency list

- **Each vertex is enqueued and dequeued atmost once**
- **Each queue operation takes O(1) time**
- Number of vertices is V, **so time devoted to 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 spent in scanning adjacency list is O(E)**
- **Time complexity of BFS = O(V) + O(E) = O(V+E)**
- In a Dense Graph (Maximum no of edges)
    - $E = O(V^2)$
    - Time complexity = $O(V) + O(V^2) = O(V^2)$
- If the graph is represented as an adjacency matrix
    - There are $|V^2|$ Entries in the adjacency matrix

- Time complexity = $O(V^2)$

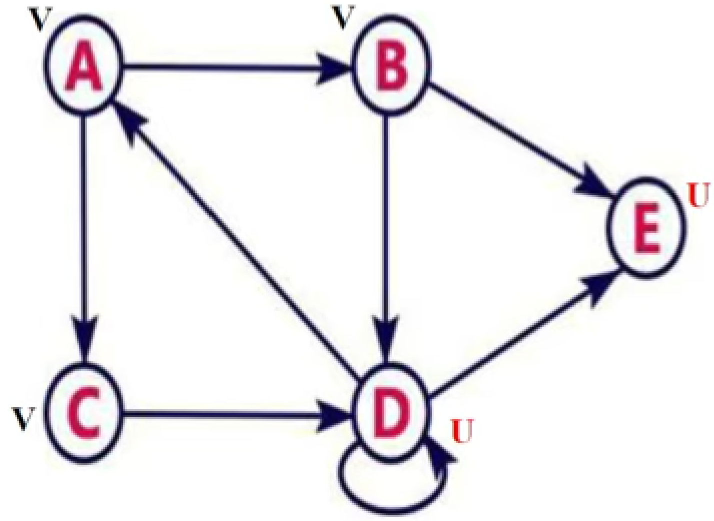
Example

- Consider this graph
- 

- **All nodes are unvisited (White)**
- Queue is empty
- **Starting with Node A, marking as Visited (Gray)**
  - 

- **Dequeueing A**
  - Checking for unvisited neighbours
  - B and C

- Adding to Queue, mark as visited

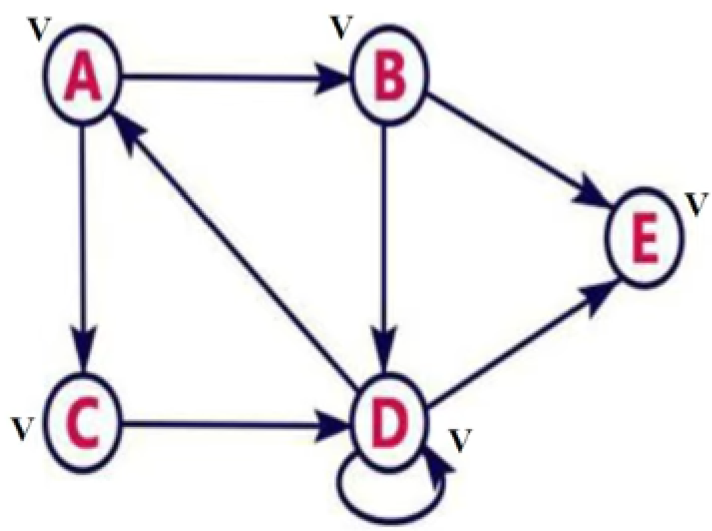


BFS Traversal: A B C

Q:[B,C]

v = A

- **Dequeuing B**
  - Neighbours, D and E



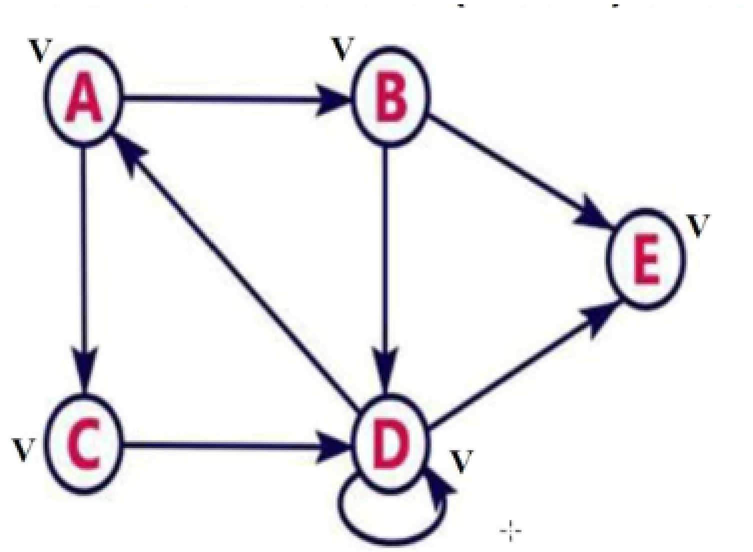
BFS Traversal: A B C D E

Q:[C,D,E]

v = B

- **Dequeueing C**

    - Neighbours, D, already visited

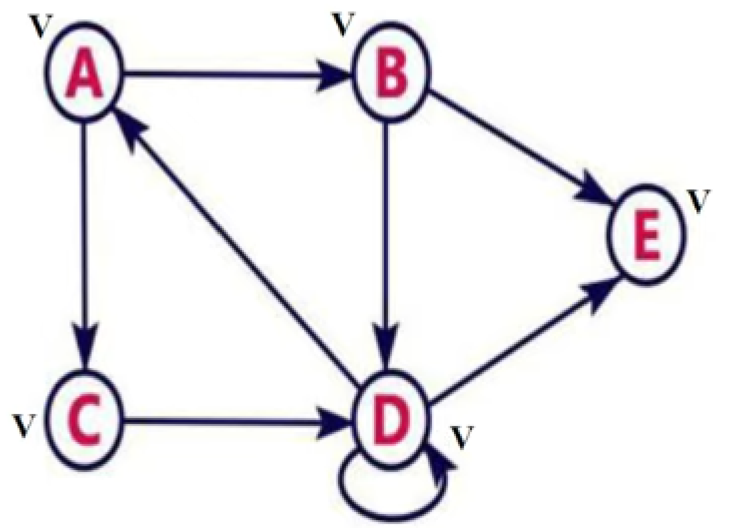


BFS Traversal: A B C D E

Q:[D,E]

v = C

- **Dequeuing D**
    - Neighbours, E,D and A, already visited



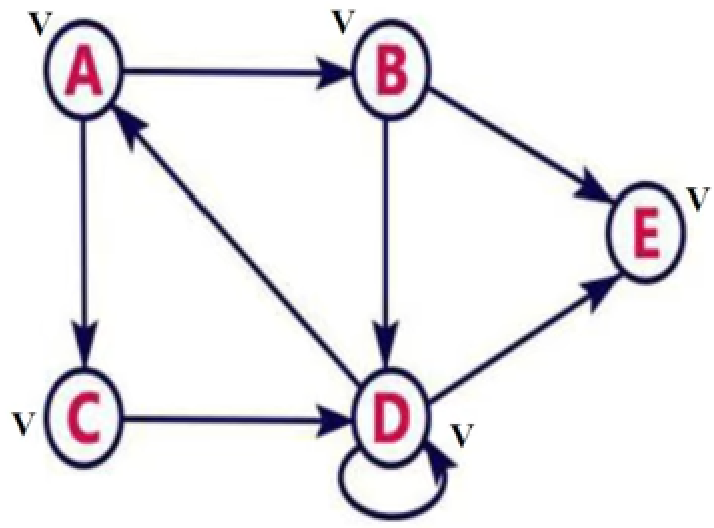
BFS Traversal: A B C D E

Q:[E]

v = D

- **Dequeueing E**

- Neighbours, none



- BFS Traversal: A B C D E
  Q:[]
  v = E

- The resultant BFS Traversal Order is
  - **A B C D E**

Applications

- Finding shortest path between 2 nodes u and v
- Minimum spanning tree for unweighted graph
- Finding nodes in any connected component of a graph

3. Depth First Search

Algorithm

Each vertex has 3 parameters

- d[v] = first visit time
- f[v] = finish time
- π[v] = predecessor of V in traversal

Algorithm DFS(g)

1. for each vertex 'u' in v do
    1. u.colour = WHITE
    2. π[u] = NIL # predecessor is Nil
2. time = 0
3. for each vertex u in v do
    1. if u.colour == WHITE then
        1. DFS_VISIT(G,u) # visiting each unvisited node

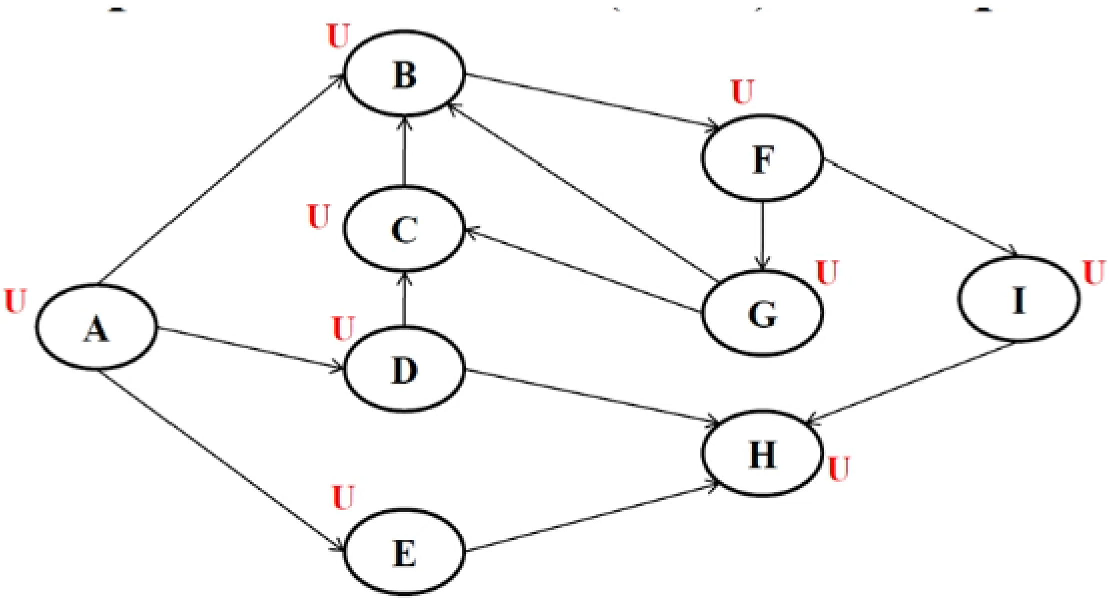
Algorithm DFS_VISIT(G,u)

1. time = time + 1
2. u.d = time
3. u.colour = GREY # Marking as visited
4. for each vertex v in Adj(u) do
    1. if v.colour == white then
        1. π[v] = u
        2. DFS_VISIT(G,V) # Visiting the adjacent vertices
5. u.colour = black # Marking vertex as fully visited
6. time = time+1
7. u.f = time # Storing finish time

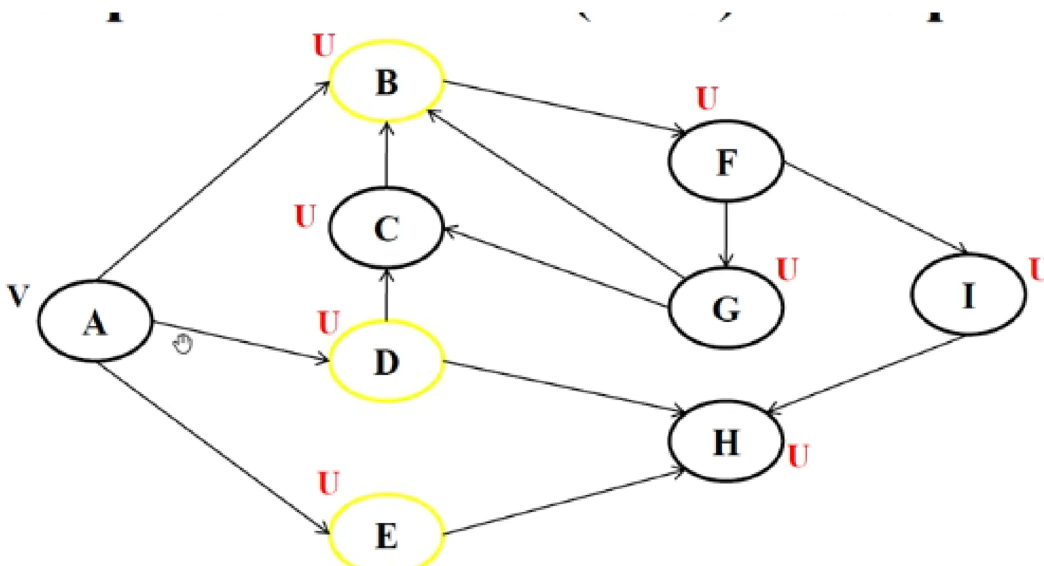
Analysis

- If graph is represented as adjacency list
    - Each vertex is visited atmost once, There are V vertices
    - **Time taken is O(V)**
    - Each adjacency list is scanned atmost once.
    - **So time devoted is O(V+E)**
- If the graph is represented as an adjacency matrix
    - There are $|V^2|$ Entries in the adjacency matrix
    - Time complexity = $O(V^2)$

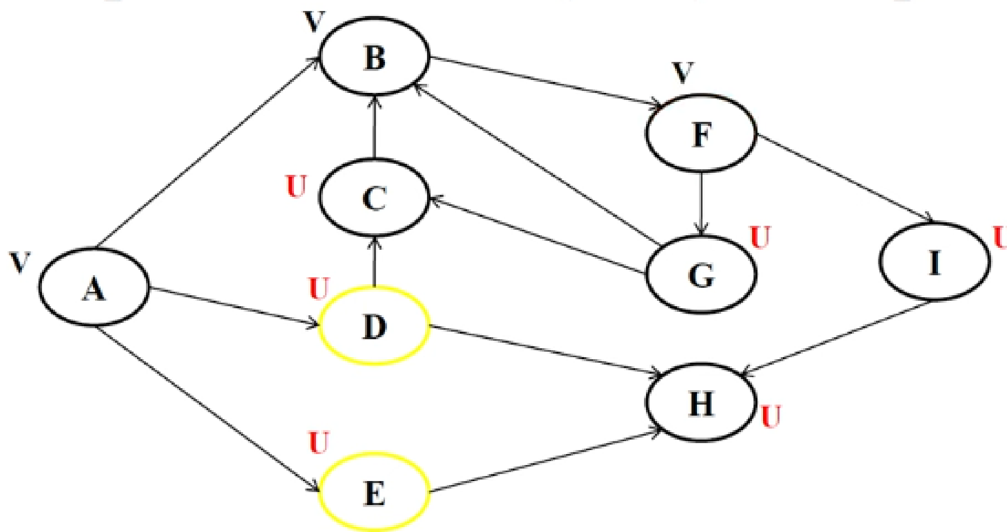
Example



- All nodes are unvisited
- Suppose A is the starting Node
- Starting Traversal
  - **Starting with A**
  - Neighbours of A are, B, D and E

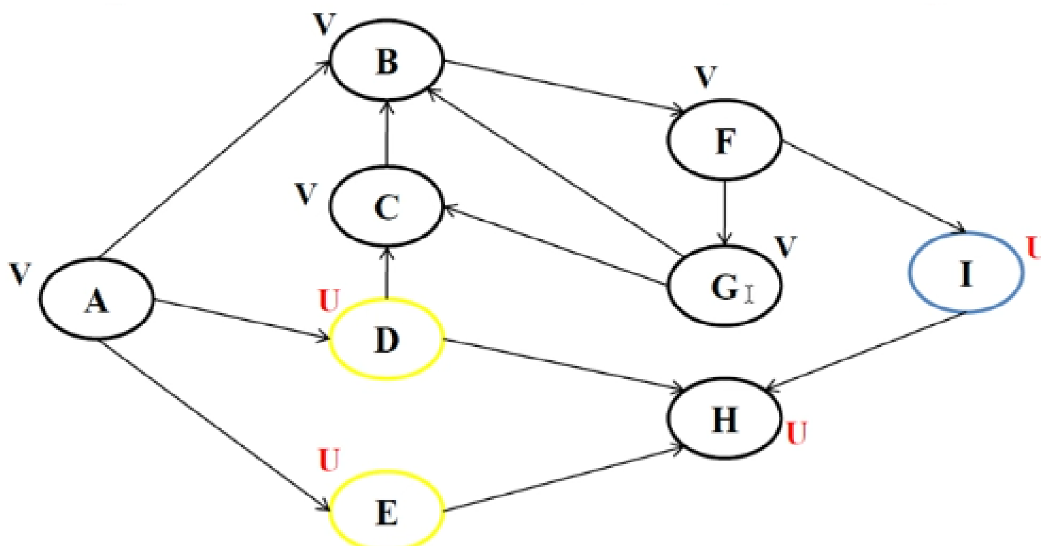


  - **DFS Traversal: A**
  - **All are unvisited, Taking B**
  - Neighbour of B is F, Unvisited

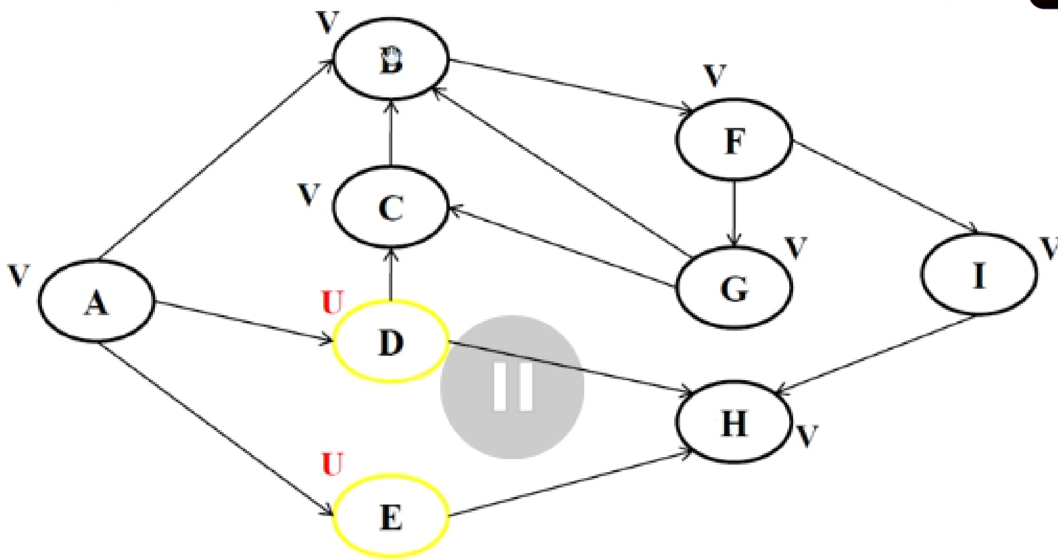


DFS Traversal: A B F

- Neighbours of F are G and I
- Neighbour of G is C

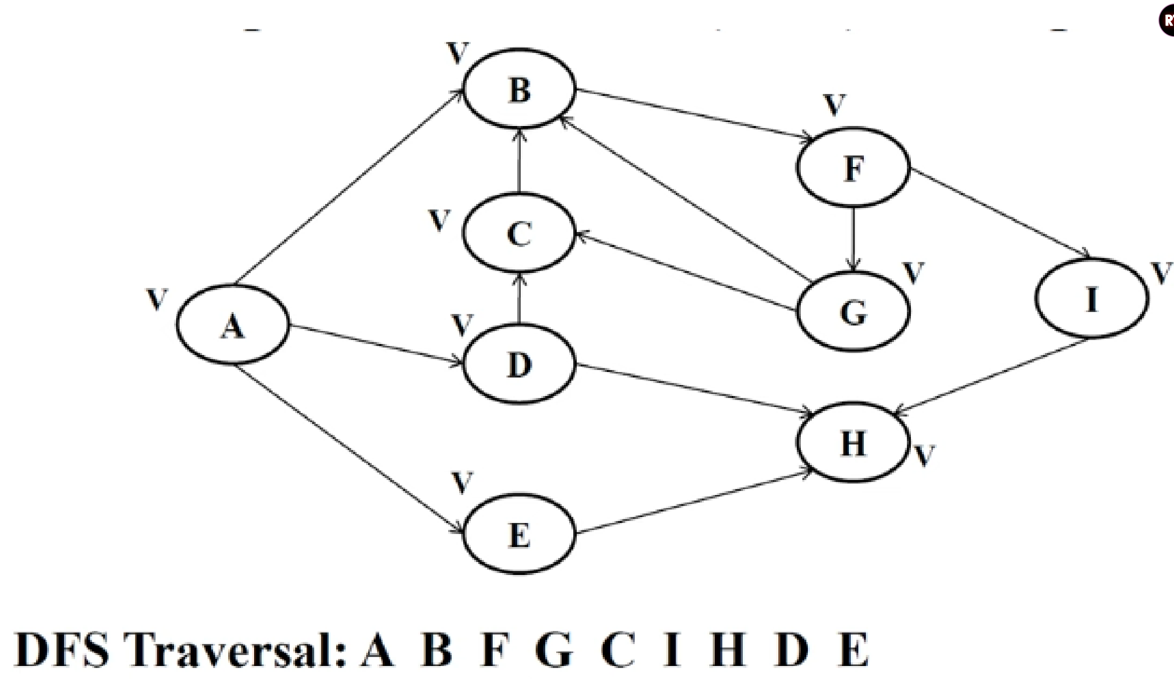


- **DFS Traversal: A B F G C**
- C has no other unvisited Neighbours
- Going back, G has no unvisited Neighbours
- Going Back, F has Unvisited Neighbour I
- I has neighbour H



DFS Traversal: A B F G C I H

-
- H has no neighbours
  - Going back, I has no other neighbours
  - Going back to F, no neighbours
  - Going back to B, no neighbours
  - Going back to A, there are D and E
- Visiting D
  - D has no other unvisited neighbours
- Going back, A has E remaining
- E has no other unvisited neighbour



- The resultant DFS Traversal is
  - A B F G C I H D E

Applications

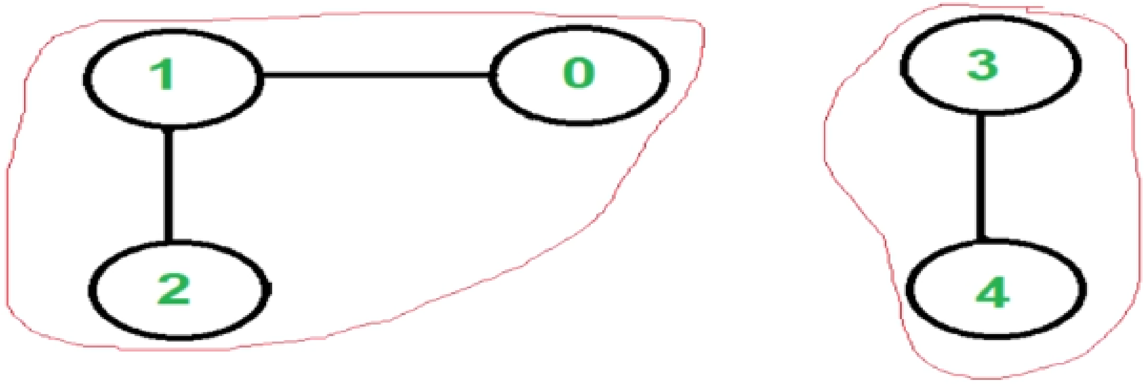
- Scheduling Problems
- Cycle detection in graphs
- Solving puzzles with only one solution, like mazes

---

4. Strongly connected components

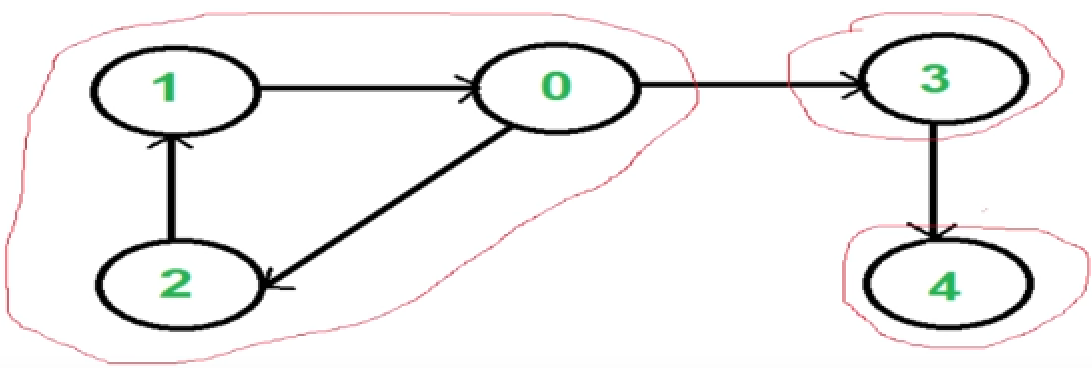
What are Connected Components?

- Connected component of graph G is a connected subgraph G of maximum size
- A graph may have more than one connected components
- For example

- 

- Here the graph has 2 connected components
    - 0,1,2
    - 3,4

What are strongly connected components?

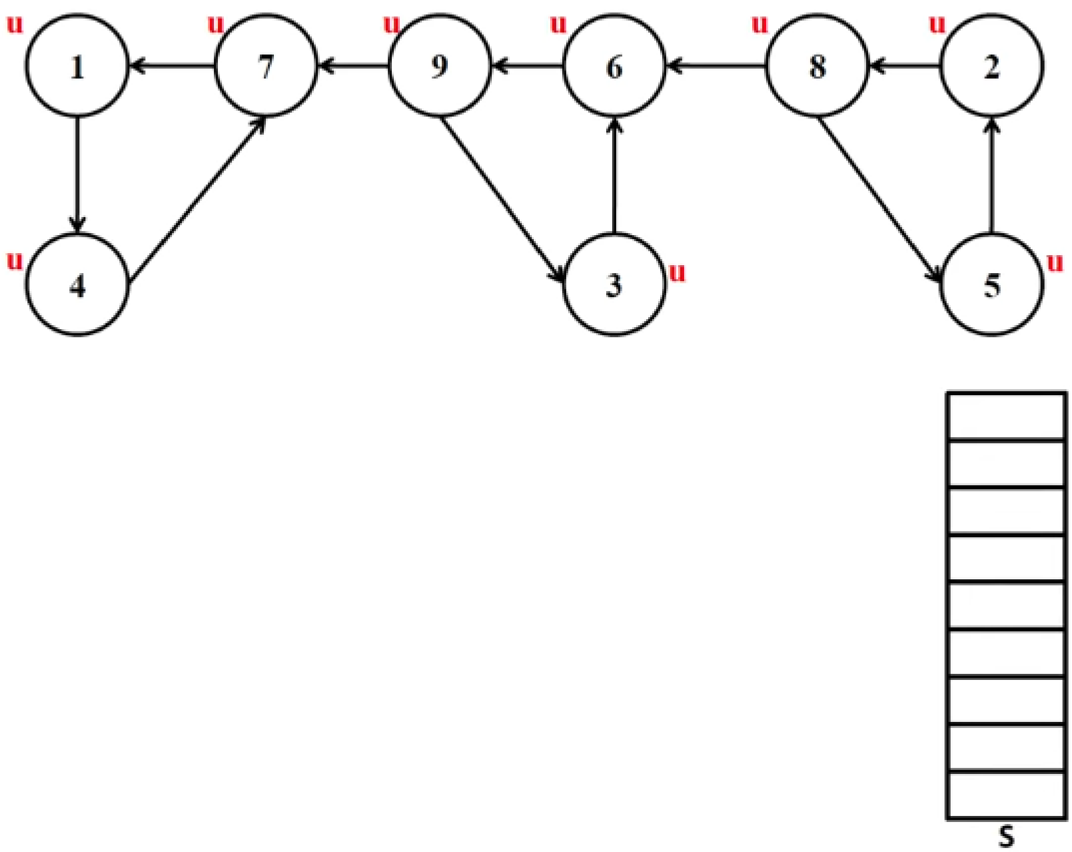
- Strong Connectivity applies only to directed graphs
- In strongly connected component of a directed graph **all the vertices in that component is reachable from every other vertex in that component**
- For example check this
    - 

    - The ones circled in red are strongly connected Components
    - Component 1 {0,1,2}
        - 0 has
            - direct edge from 1
            - A path from 2-1-0
            - Hence 0 can be accessed from both 1 and 2
        - 1 has
            - a direct edge from 2
            - A path from 0-2-1
            - Hence 1 can be accessed from both 2 and 0

        - Similarly 2 also can be accessed
        - But when we check 3, 3 cant access other numbers like 1, 0 and 2. So it cant be included in this component
    - Component {3} and {4} are individual so its strongly connected

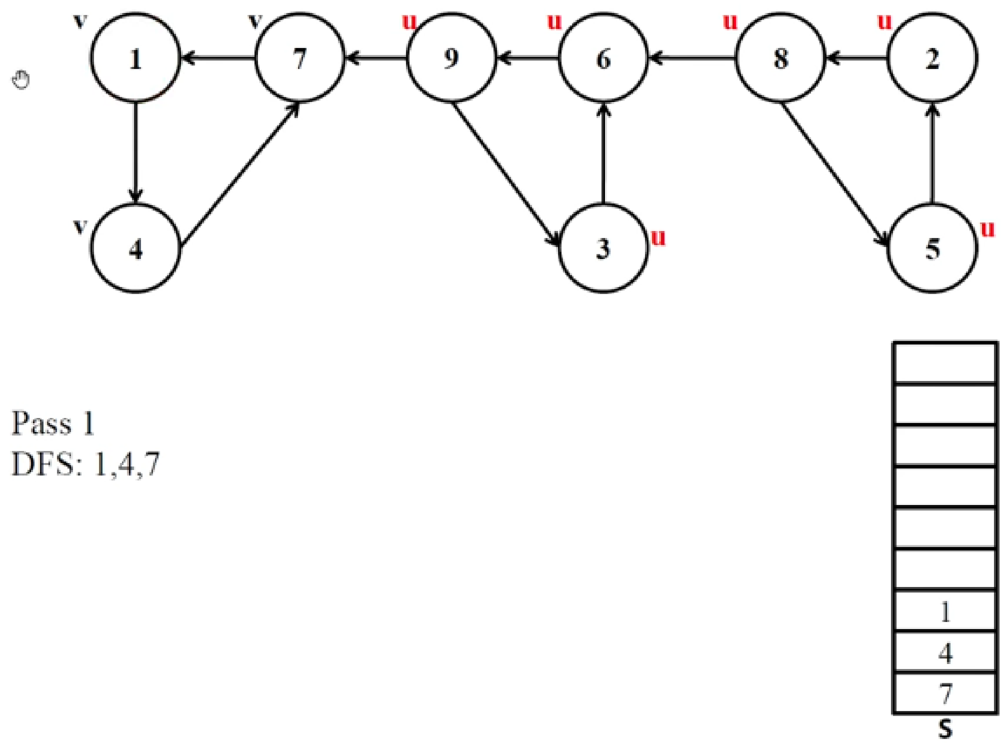
Example

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

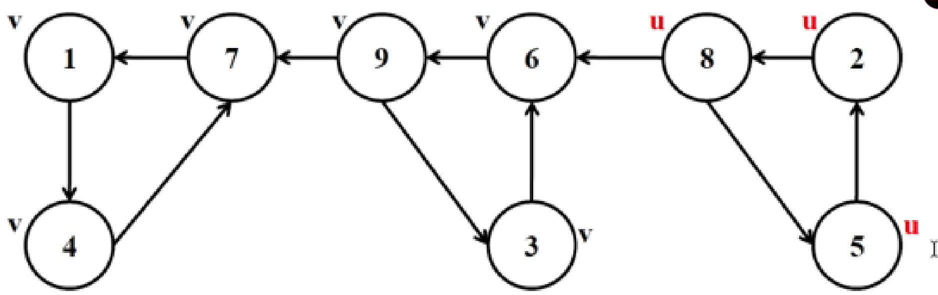


- **Pass 1**
    - All nodes are unvisited
    - Stack is empty
    - Starting DFS Traversal
        - **Starting from 1, mark as visited**
            - Its unvisited neighbour is 4
            - 4's neighbour is 7
            - 7 has no neighbour
                - Push 7 to the stack
            - Going back, 4 has no other neighbour

    - Push 4 to stack
  - Going back, 1 has no other neighbour
    - Push 1 to stack

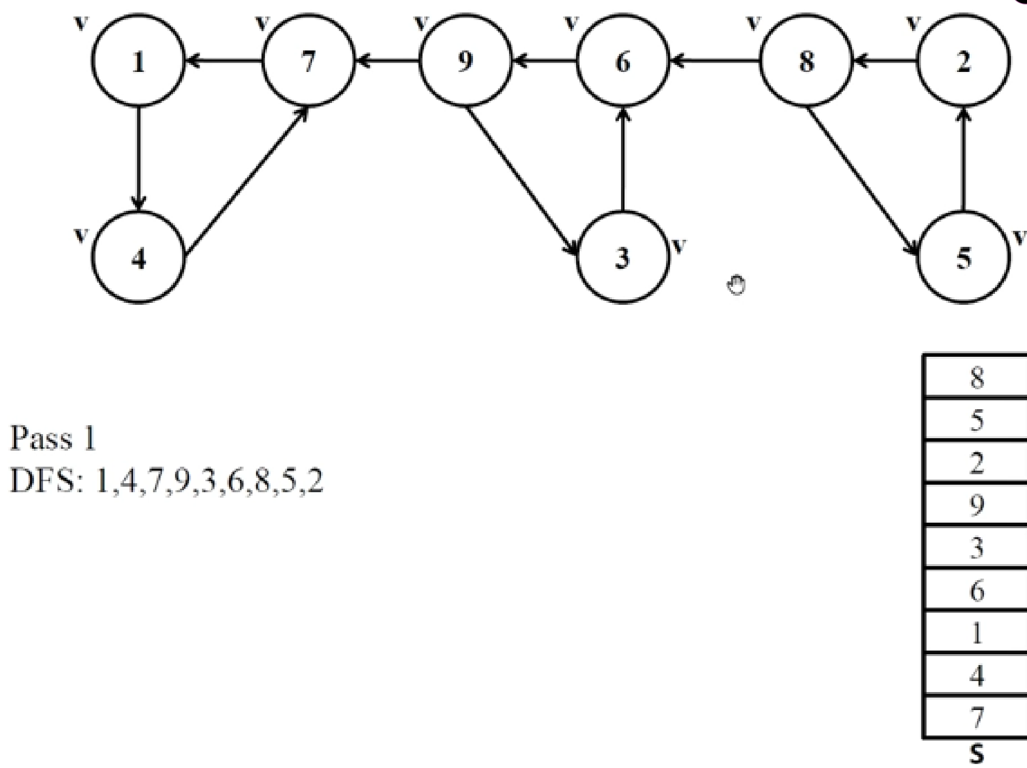


- **Starting with 9, visiting**
  - Neighbour is 3, visiting
  - Neighbour of 3 is 6, visiting
  - 6 has no neighbour
    - Pushing 6 to stack
  - Going back, 3 has no neighbour
    - Pushing 3 to the stack
  - Going back 9, no neighbour
    - Pushing 9 to stack

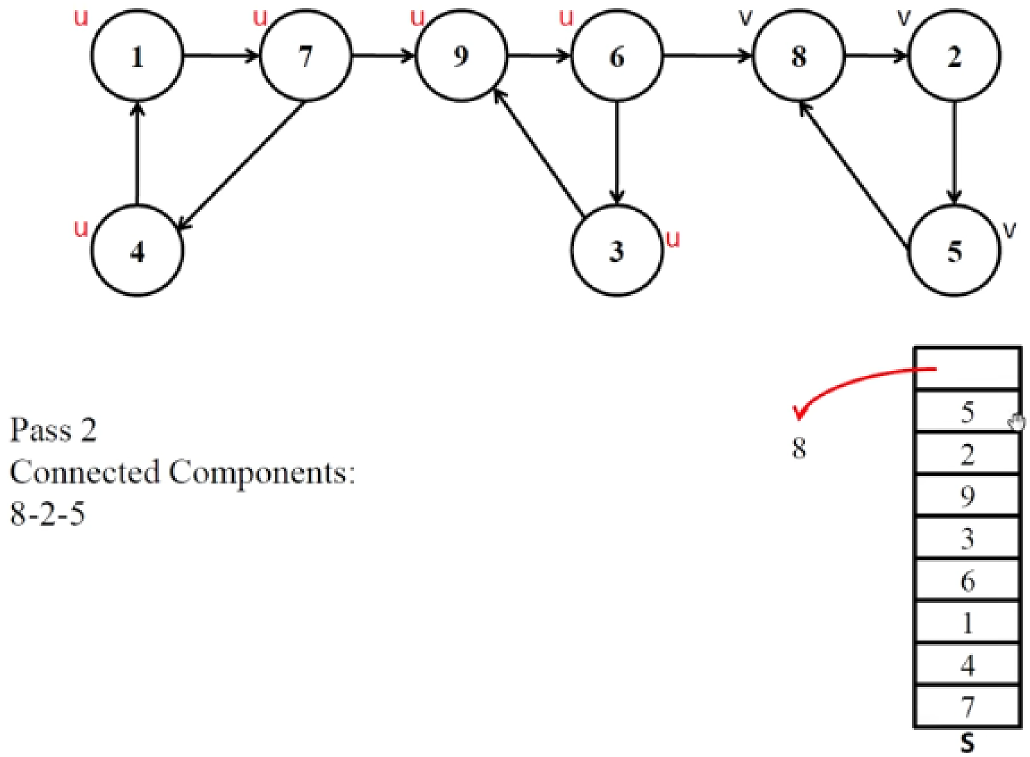


Pass 1
DFS: 1,4,7,9,3,6

- **Starting with 8, visiting**
  - Neighbour of 8 is 5, visiting 5
  - Neighbour of 5 is 2, visiting 2
  - 2 has no other unvisited neighbour
    - Push 2 to stack
  - Going back, 5 has no other neighbour
    - Push 5 to stack
  - Going back, 8 has no other neighbour
    - Pushing 8 to stack



- **Pass 2**
  - Now we do DFS, Pop each element from stack one by one
  - **Popping 8,**
    - 8 has a neighbour 2
    - 2 has a neighbour 5
    - 5 has no other neighbour
      - going back 2 has no other neighbout
      - going back, 8 has no other neighbour
    - **This forms 8-2-5, which is a connected component**



-

- **Popping 5**
  - Already visited
- **Popping 2**
  - Already visited
- **Popping 9**
  - Visiting 9
    - Neighbour 6
      - Neighbour 3
  - **9-6-3 is the connected component**
- **Popping 3**
  - Already visited
- **Popping 6**
  - Already visited
- **Popping 1**
  - Unvisited
  - Visiting 1
    - Visiting neighbour 7
      - Visiting neighbour 4
  - **1-7-4 is the connected component**

So the strongly connected components are

- **8-2-5**
- **9-6-3**
- **1-7-4**

Strongly Connected Components - Complexity

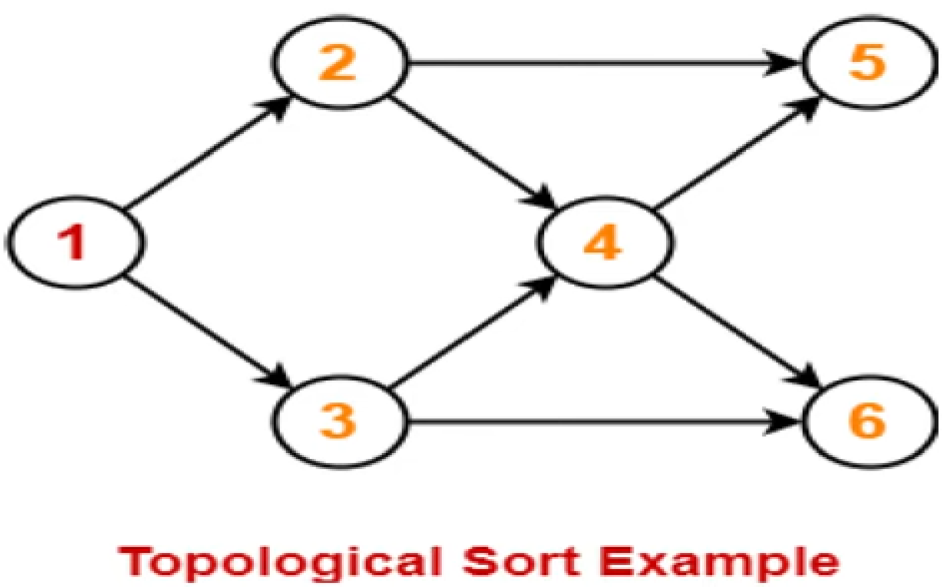
- First pass we did DFS, Time complexity = O(V+E)
- Pass 2 takes another O(V+E)
- Total time complexity = O(V+E)

Applications

- Social networks

---

5. Topological sorting

- Topological sorting is applicable for Direct Acyclic Graph (DAG)
  - The graphs should not have a cycle
  - 

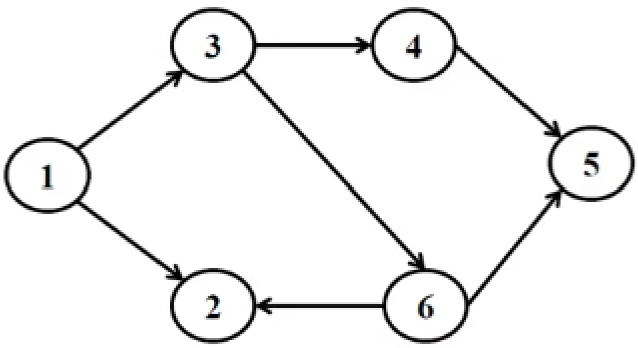
  - One graph can have many topological ordering

Topological sorting algorithm

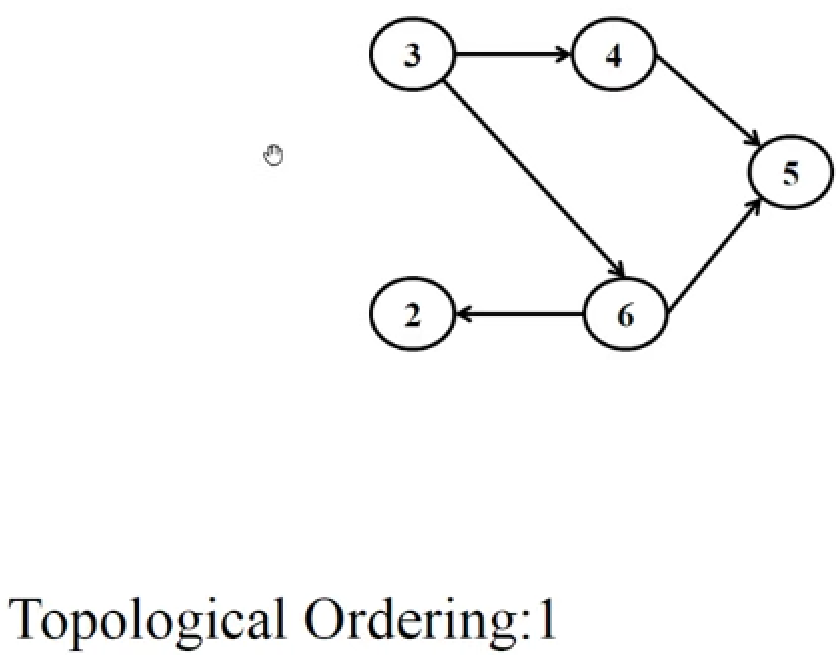
- Identify a node with no incoming edges(indegree = 0)
- Add this node to the topological ordering
- Remove this node and all its outgoing edges from the graph
- Repeat step 1 to 3 until graph becomes empty

Example

Write the topological sorting for the DAG given below

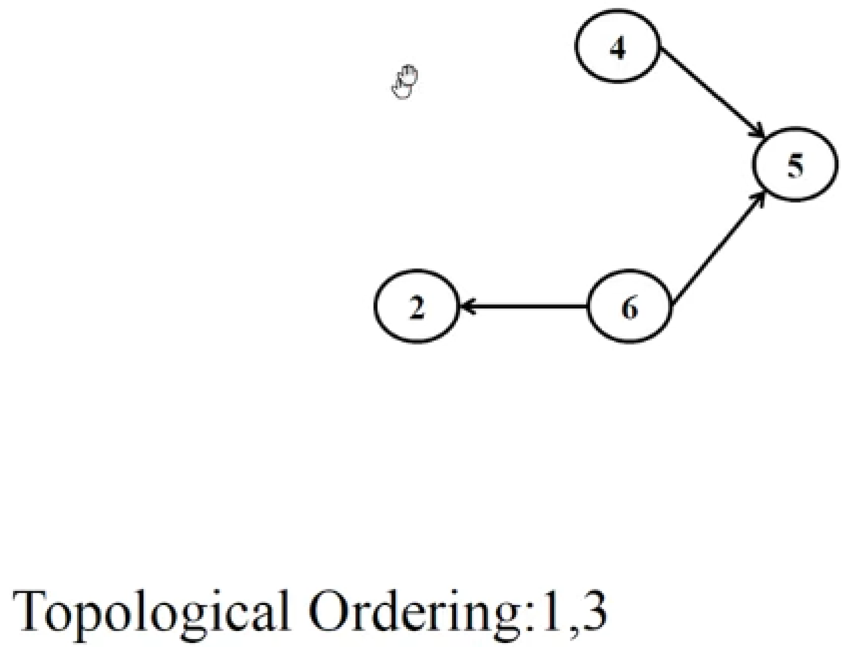


- Here 1 has no incoming edges, its indegree is 0
    - Adding 1 to topological ordering and removing it from graph



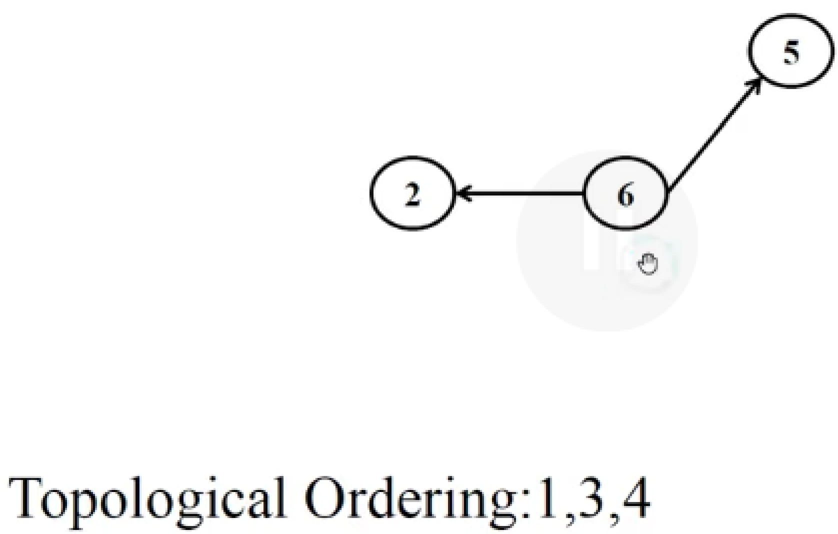
    -
- Now, 3 has no incoming edges
    - Adding 3 to topological ordering

    - Removing 3 from graph



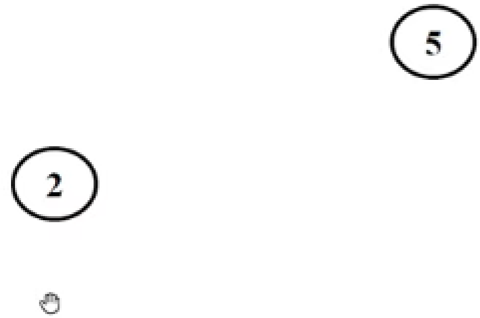
    -

- 4 has no incoming edges
    - Adding 4 to topological ordering
    - Removing 4 from graph



    -

- 6 has no incoming edges



- Topological Ordering:1,3,4,6

- 2 and 3 remains, select in any order
  - Selecting 2
    - Adding to ordering
    - Removing
  - Selecting 3
    - Adding to ordering
    - Removing
- The topological ordering
  - Topological Ordering:1,3,4,6,2,5

Topological sorting complexity

- Time to determine indegree for each node = O(E) time
- Time to determine nodes with no incoming edges = O(V)
- Step 1 complexity = O(E+V)
- Add nodes until we run out of nodes with no incoming edges
  - Takes O(V) times
- **Total time complexity = O(V+E)**