Data Structures and Algorithms Design

DSECLZG519

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Parthasarathy

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

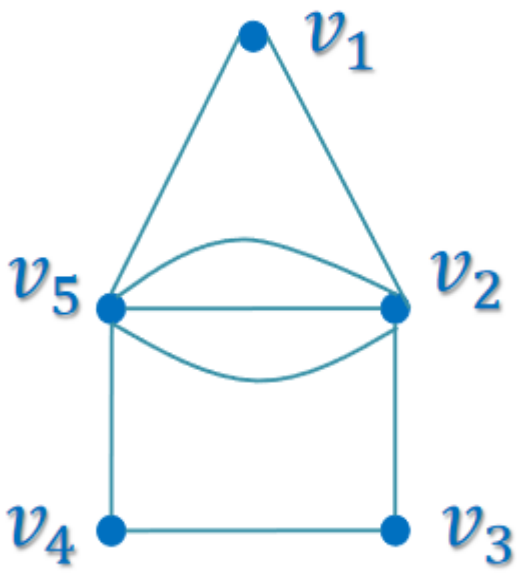
Contact Session #7

DSECLZG519 – Graph Traversals & BST

Agenda for CS #7

1) Recap and continuation of CS#6
2) Graph Traversals:
   - BFS
   - DFS
3) BST
   - Motivation
   - Insertion
   - Deletion
4) Exercises
5) Q&A

Adjacency Matrix Examples



$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 3 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 3 & 0 & 1 & 0 \end{pmatrix}$$

Adjacency Matrix: Pros and Cons

Advantages

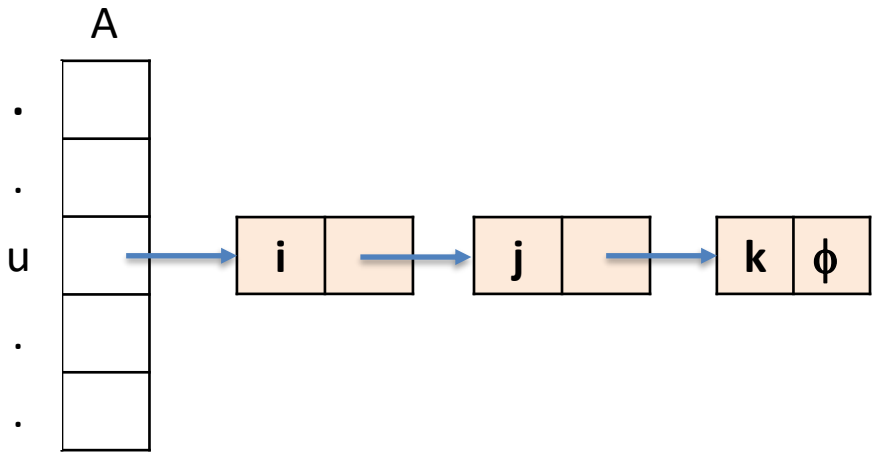
- Fast to tell whether edge exists between any two vertices *i* and *j* (and to get its weight)

Disadvantages

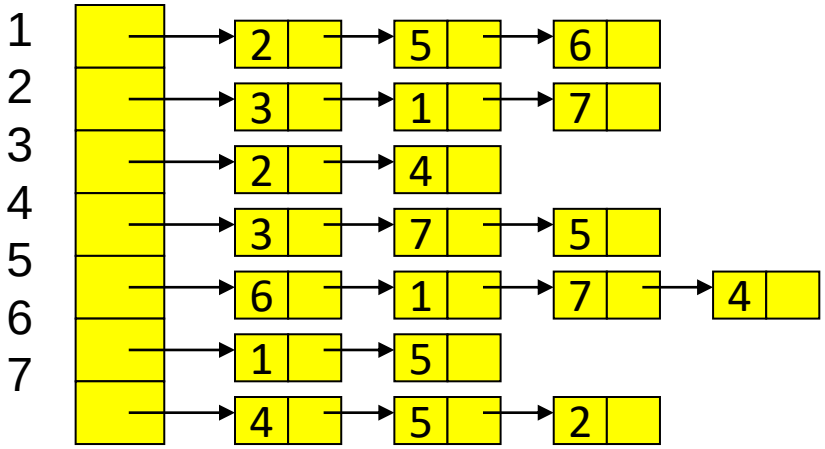
- Consumes a lot of memory on **sparse** graphs (ones with few edges)
- Redundant information for undirected graphs

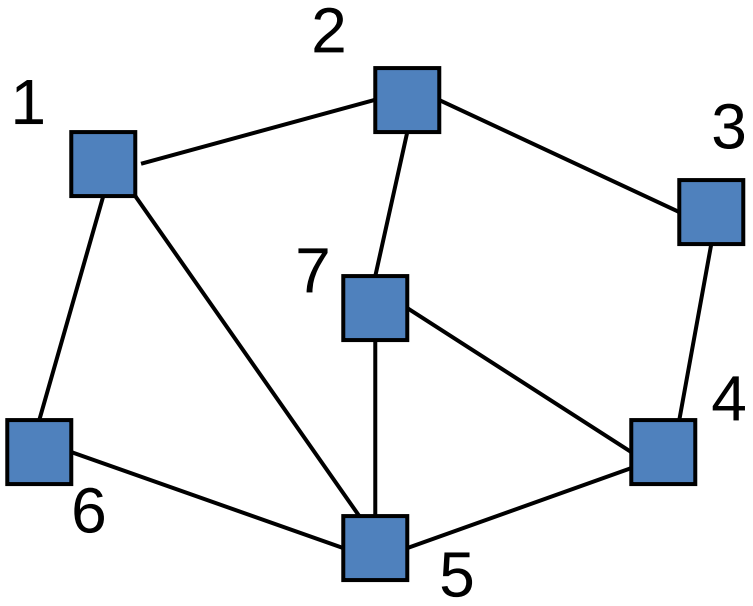
Adjacency Lists

- An adjacency linked list is an array of n linked lists where n is the number of vertices in graph G. Each location of the array represents a vertex of the graph. For each vertex u ∈ V, a linked list consisting of all the vertices adjacent to u is created and stored in A[u]. The resulting array A is an adjacency list.
- Note: It is clear from the def. that if i,j and k are the vertices adjacent to the vertex u, then i, j and k are stored in a linked list and starting address of linked list is stored in A[u] as shown below:



Adjacency List Example





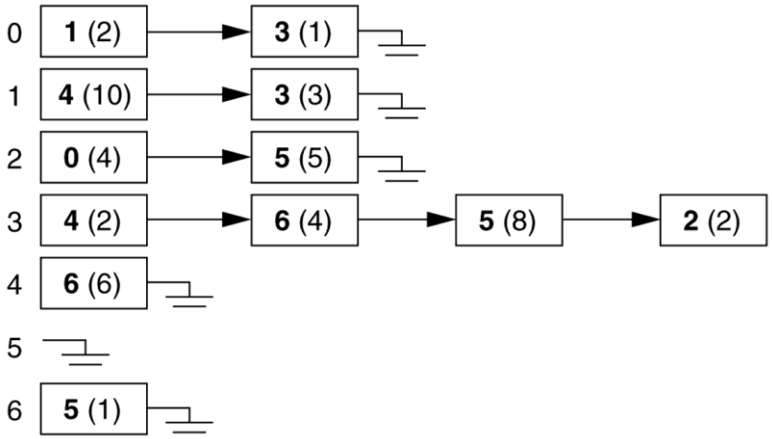
Adjacency List: Pros and Cons

Advantages:

- new nodes can be added easily
- new nodes can be connected with existing nodes easily
- "who are my neighbors" easily answered

Disadvantages:

- determining whether an edge exists between two nodes: O(average degree)



Breadth First Search (BFS)

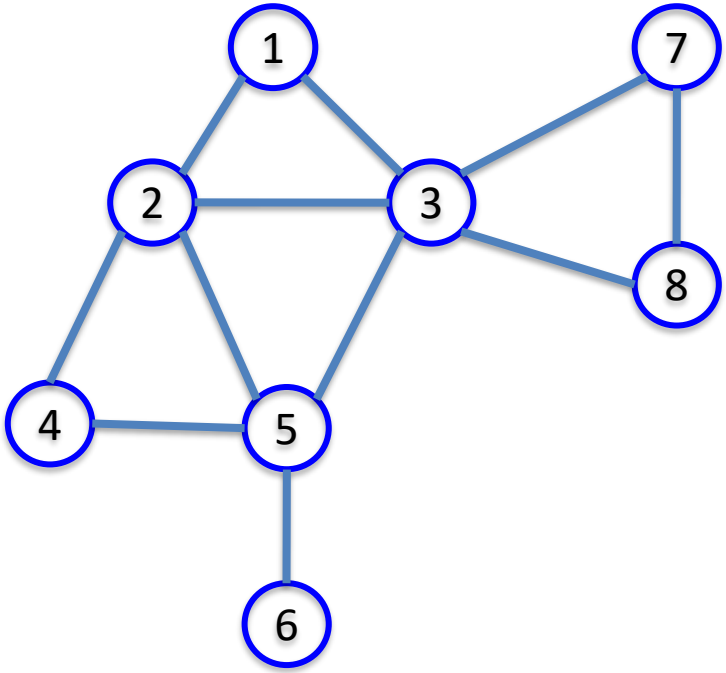
- Breadth-first search (BFS) is a general technique for traversing a graph.
- A BFS traversal of a graph G:
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G

- BFS on a graph with n vertices and m edges takes $O(n + m)$ time
- BFS can be further extended to solve other graph problems
  - Find and report a path with the minimum number of edges between two given vertices
  - Find a simple cycle, if there is one

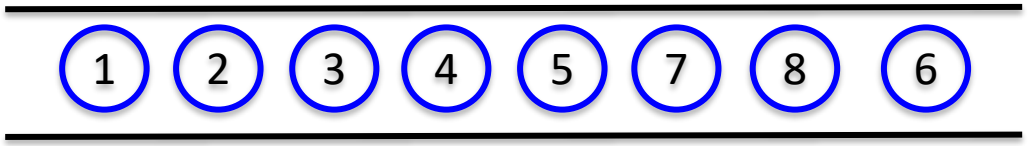
Breadth First Search (BFS)

- It is very common to use a *queue* to keep track of:
  - nodes to be visited next, or
  - nodes that we have already visited.
- Typically, use of a queue leads to a *breadth-first* visit order.
- Breadth-first visit order is "cautious" in the sense that it examines every path of length i before going on to paths of length i+1.
- Dotted lines depict *Cross edges* and solid lines depict the *discovery edges*.

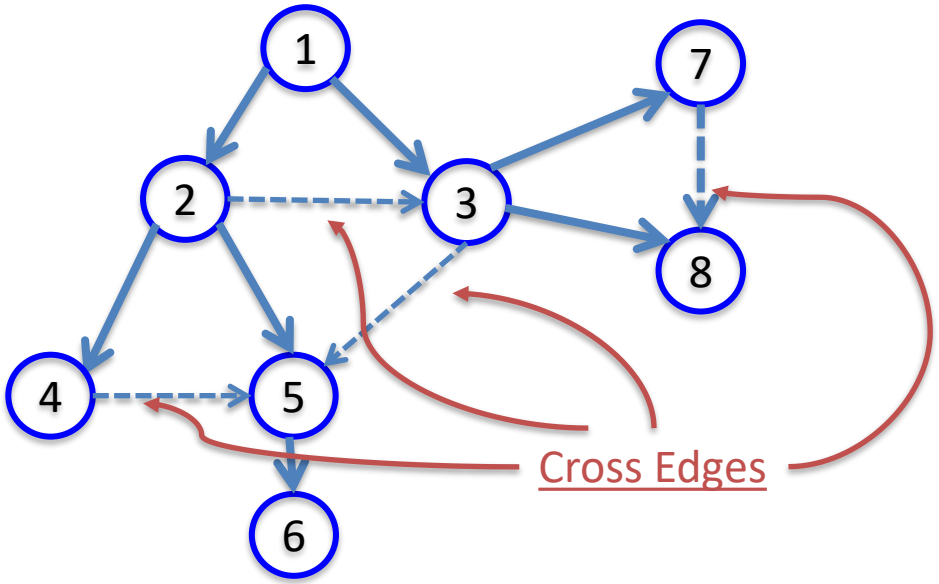
BFS - Demo



Queue - FIFO

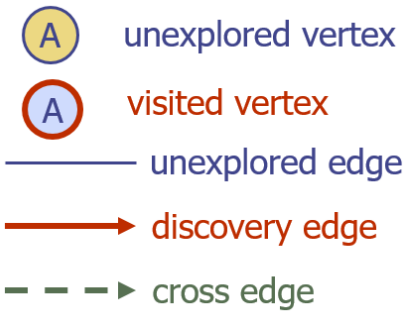


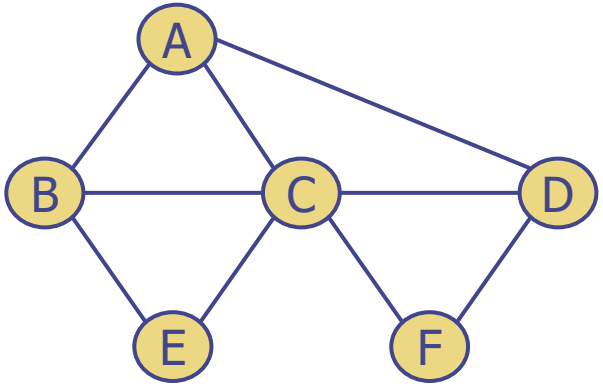
BFS Spanning Tree

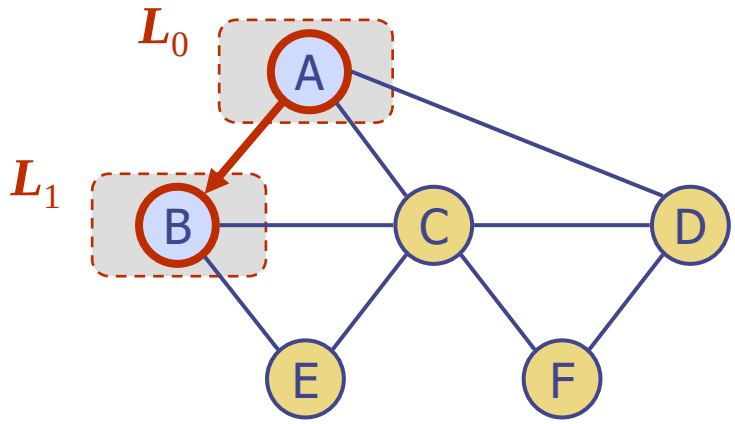


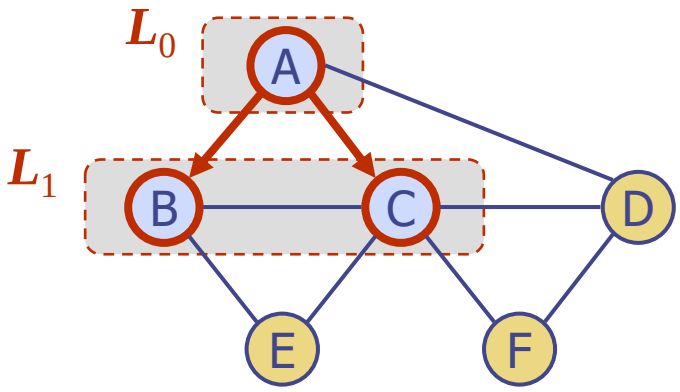
BFS: **1,2,3,4,5,7,8,6**

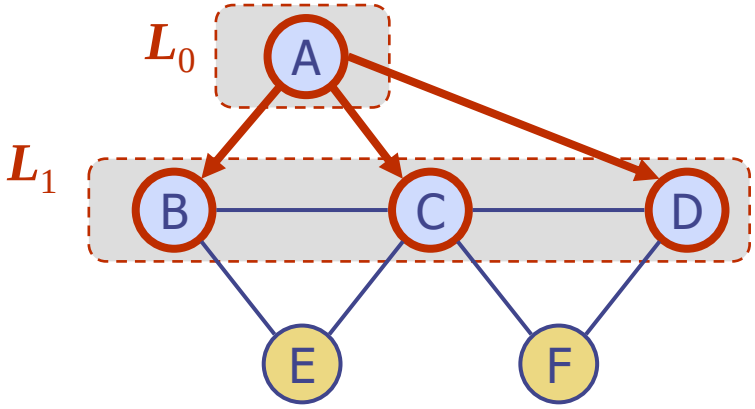
BFS – Example 2



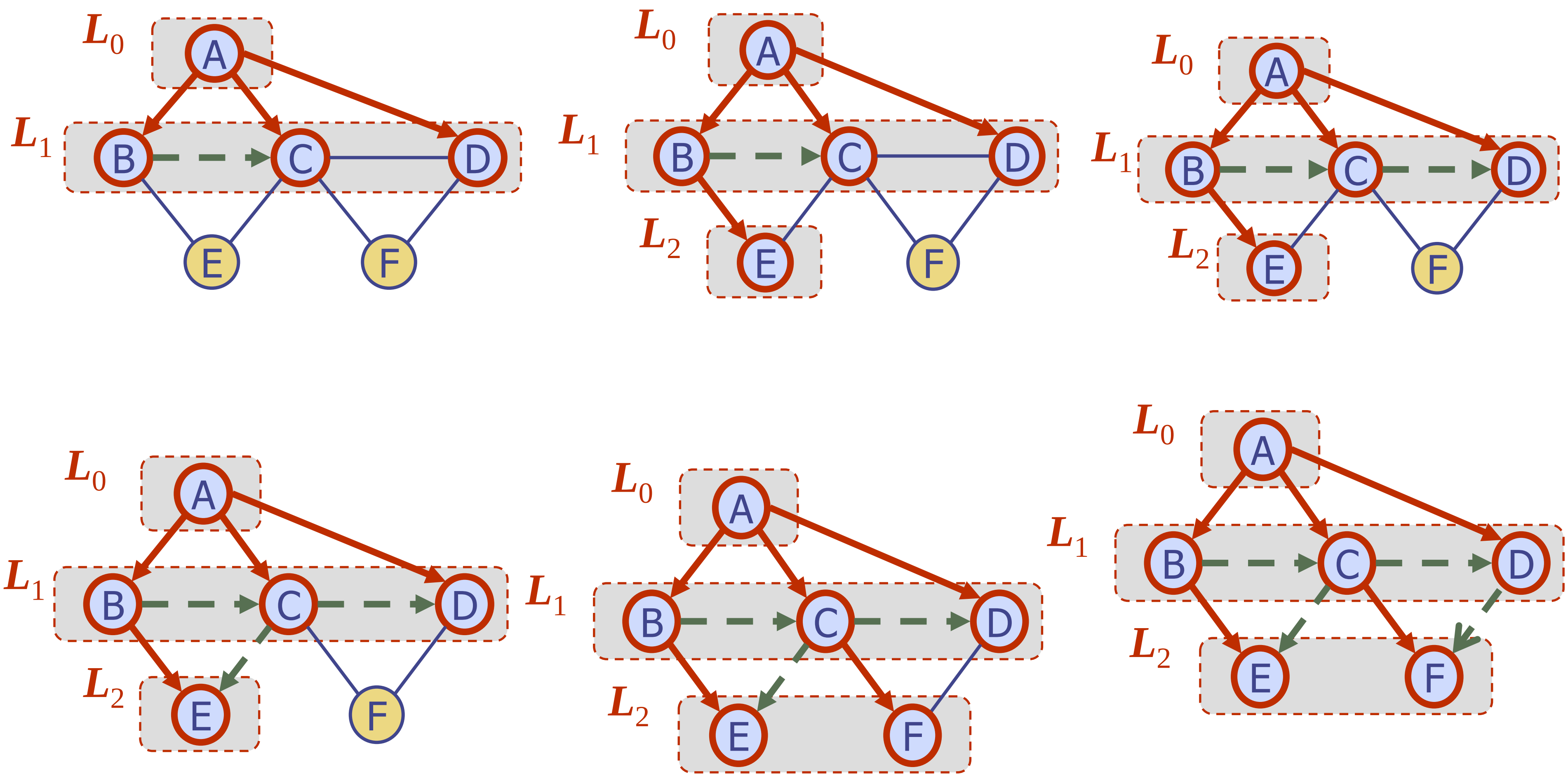








BFS – Example 2



BFS- Pseudocode

```
BFS (G, s)                          //Where G is the graph and s is the source node
      let Q be queue.
      Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices are marked.

      mark s as visited.
      while ( Q is not empty)
            //Removing that vertex from queue,whose neighbour will be visited now
            v  =  Q.dequeue( )

          //processing all the neighbours of v
          for all neighbours w of v in Graph G
                if w is not visited
                          Q.enqueue( w )              //Stores w in Q to further visit its neighbour
                          mark w as visited.
```

The time complexity of BFS is **O(V+E)** where V is the number of nodes and E is the number of edges.

BFS Applications

- We can specialize the BFS traversal of a graph G to solve the following problems in $O(n + m)$ time
  - Compute the connected components of G
  - Compute a spanning forest of G
  - Find a simple cycle in G, or report that G is a forest
  - Given two vertices of G, find a path in G between them with the minimum number of edges, or report that no such path exists

Good Design exercise! Try the above problems using BFS traversal

Depth First Search (DFS)

- Depth-first search (DFS) is a general technique for traversing a graph
- A DFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G

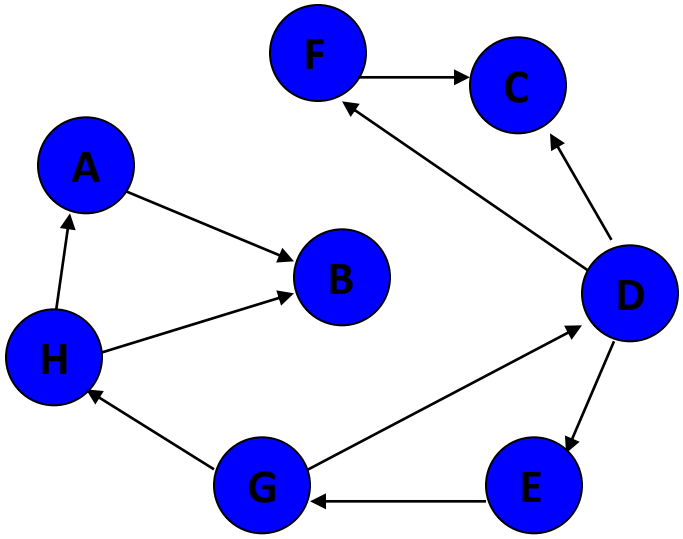
- DFS on a graph with n vertices and m edges takes $O(n + m)$ time
- DFS can be further extended to solve other graph problems
  - Find and report a path between two given vertices
  - Find a cycle in the graph

DFS

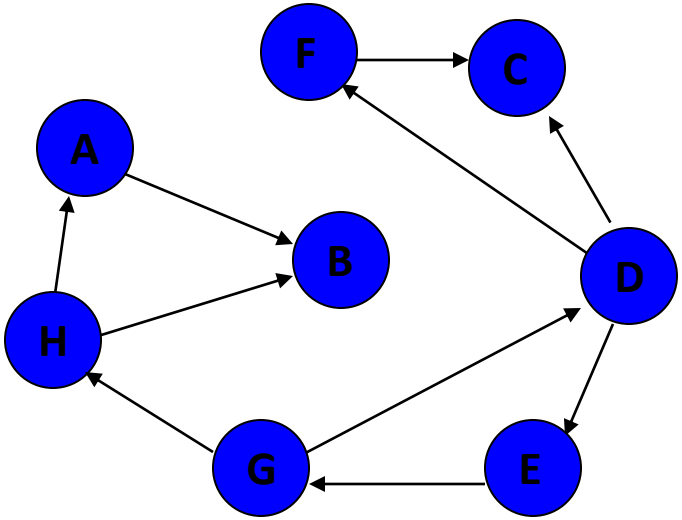
- It is very common to use a *stack* to keep track of:
  - nodes to be visited next, or
  - nodes that we have already visited.
- Typically, use of a stack leads to a *depth-first* visit order.
- Depth-first visit order is “aggressive” in the sense that it examines complete paths.
- Solid lines depict the *discovery edges* and dotted lines depict the *Back edges*.

DFS-Demo

Consider this graph below and perform DFS with D as start node.

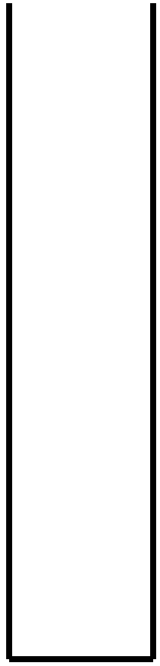


Walk-Through



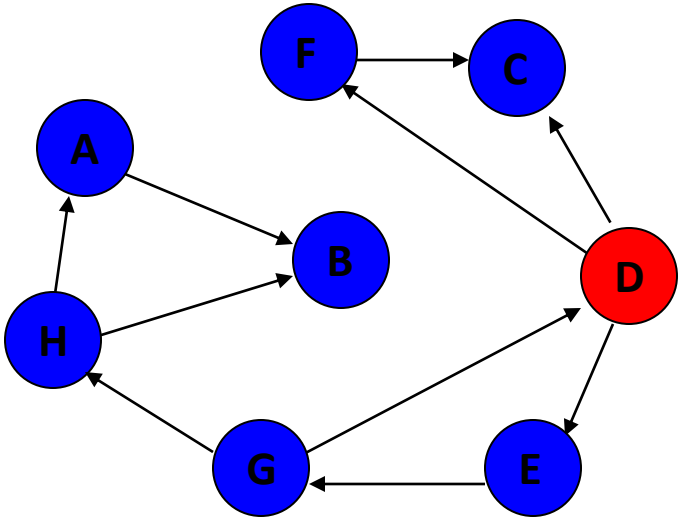
Visited Array

A	
B	
C	
D	
E	
F	
G	
H	



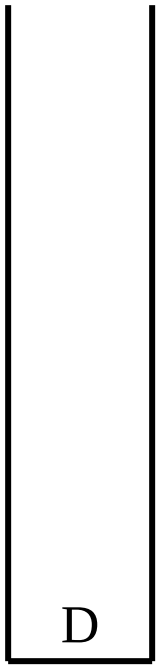
Task: Conduct a depth-first search of the graph starting with node D

Walk-Through



Visited Array

A	
B	
C	
D	√
E	
F	
G	
H	

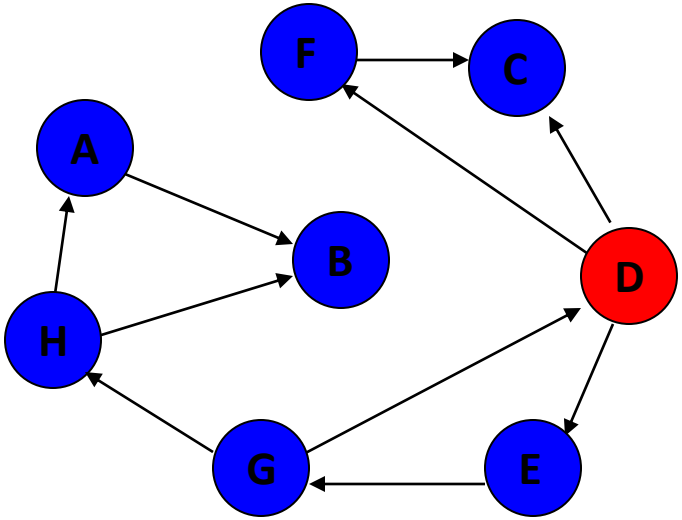


The order nodes are visited:

D

Visit D

Walk Through



Visited Array

A	
B	
C	
D	√
E	
F	
G	
H	

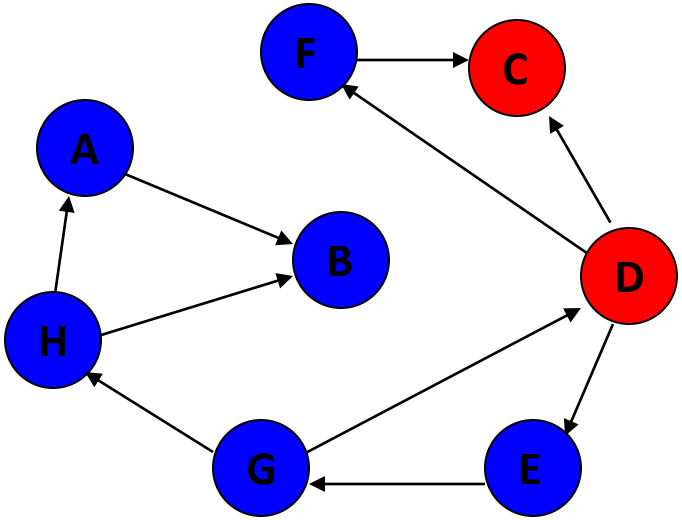
D

The order nodes are visited:

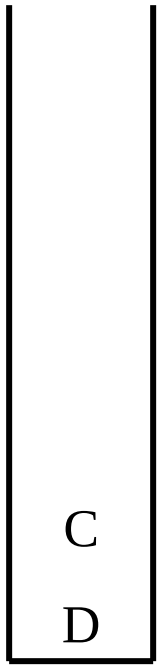
D

Consider nodes adjacent to D, decide to visit C first (Rule: visit adjacent nodes in alphabetical order)

Walk Through



A	
B	
C	√
D	√
E	
F	
G	
H	

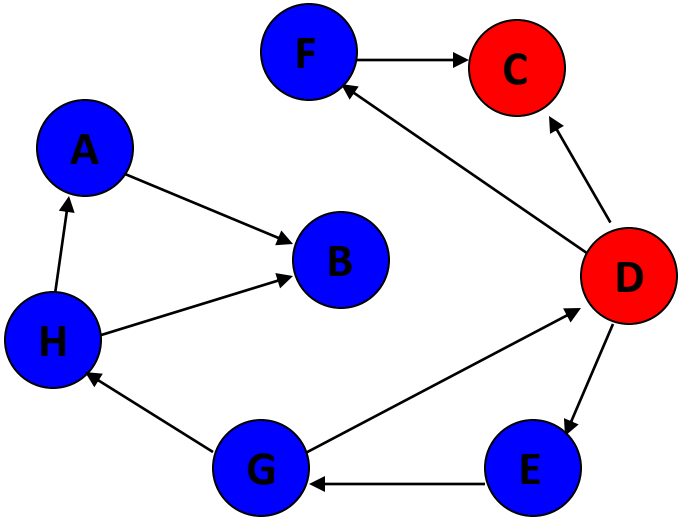


Visit C

The order nodes are visited:

D, C

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	
F	
G	
H	

C

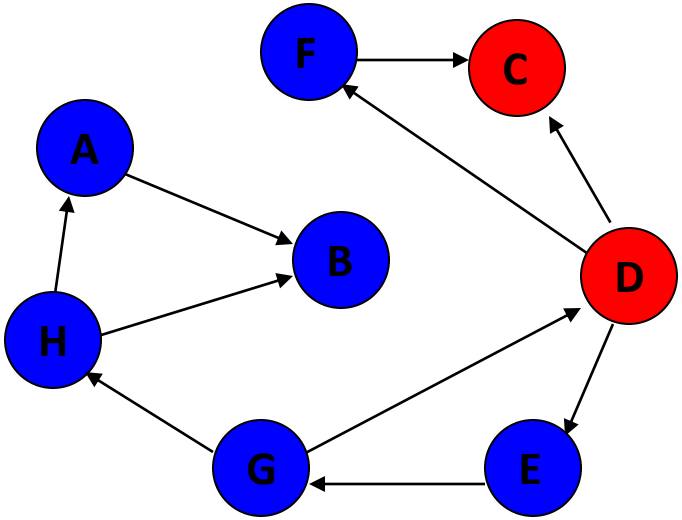
D

The order nodes are visited:

D, C

No nodes adjacent to C; cannot continue → *backtrack*, i.e., pop stack and restore previous state

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	
F	
G	
H	

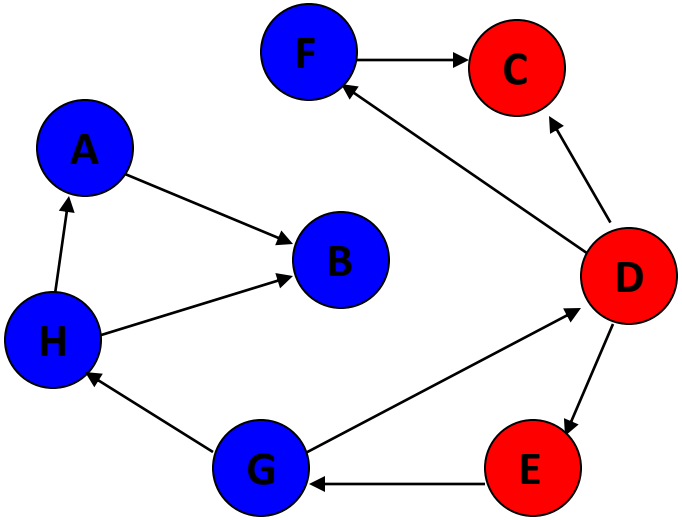
D

The order nodes are visited:

D, C

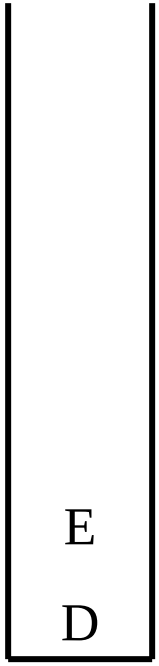
Back to D – C has been visited, decide to visit E next

Walk Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	
H	

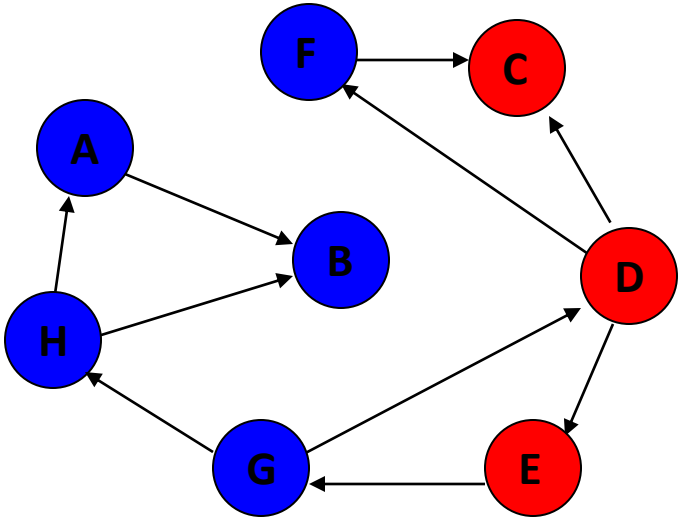


The order nodes are visited:

D, C, E

Back to D – C has been visited, decide to visit E next

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	
H	

E

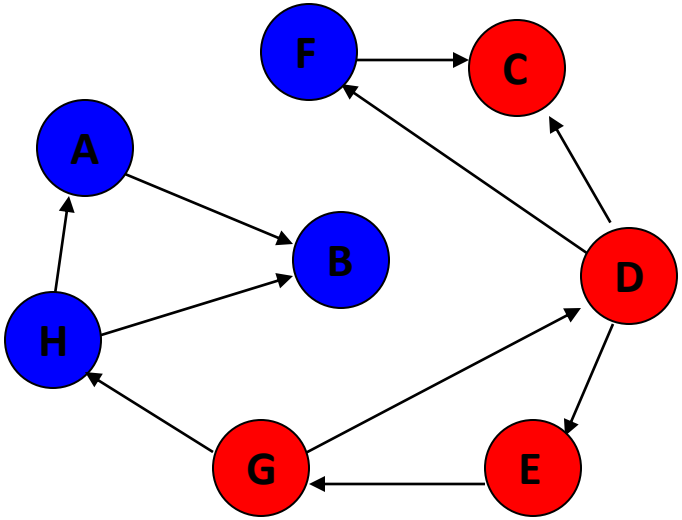
D

The order nodes are visited:

D, C, E

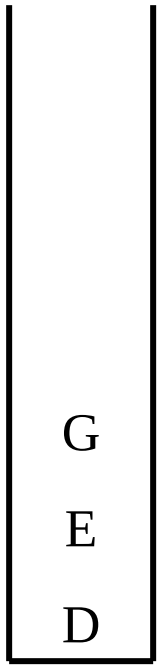
Only G is adjacent to E

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	√
H	

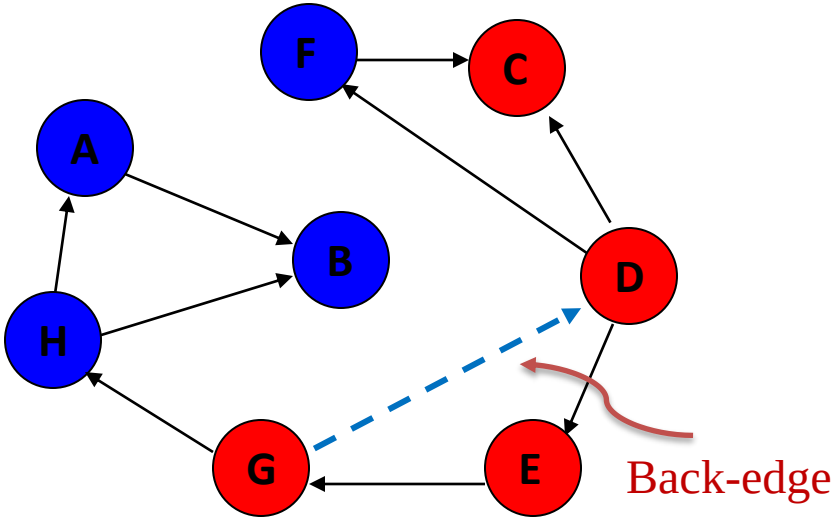


Visit G

The order nodes are visited:

D, C, E, G

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	√
H	

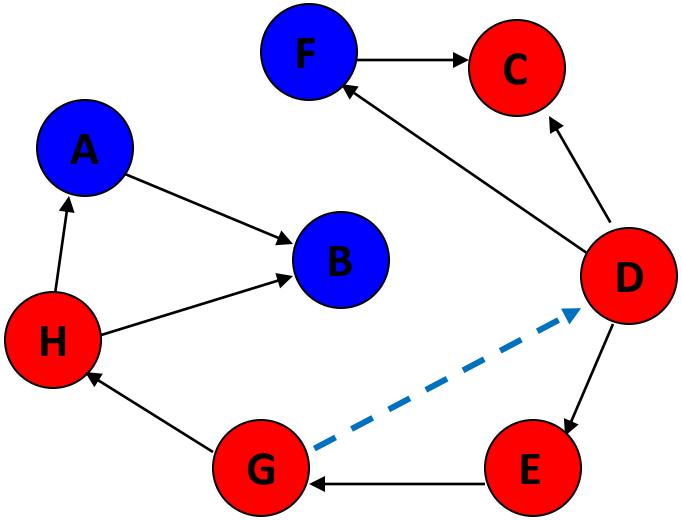
G
E
D

The order nodes are visited:

D, C, E, G

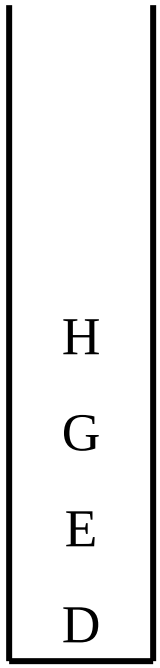
Nodes D and H are adjacent to G. D has already been visited. Decide to visit H.

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	√
H	√

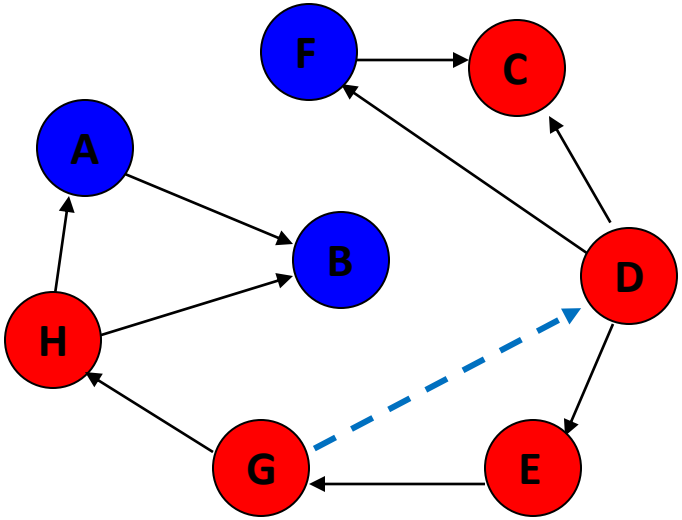


Visit H

The order nodes are visited:

D, C, E, G, H

Walk-Through



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	√
H	√

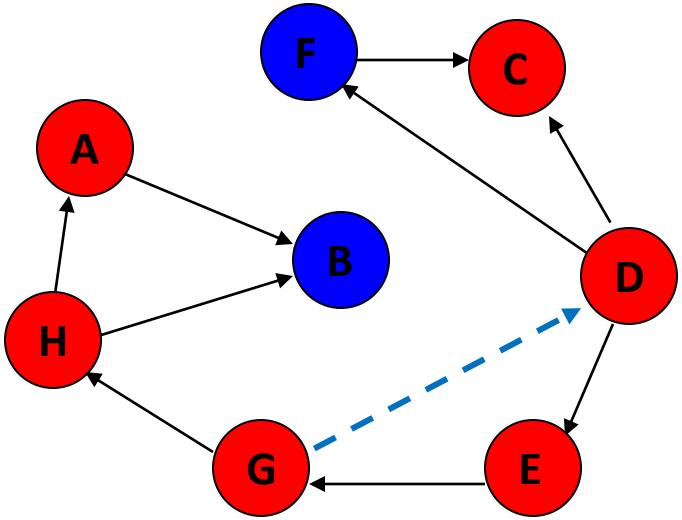
H
G
E
D

The order nodes are visited:

D, C, E, G, H

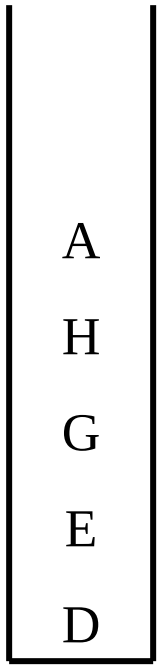
Nodes A and B are adjacent to F. Decide to visit A next.

Walk-Through



Visited Array

A	√
B	
C	√
D	√
E	√
F	
G	√
H	√

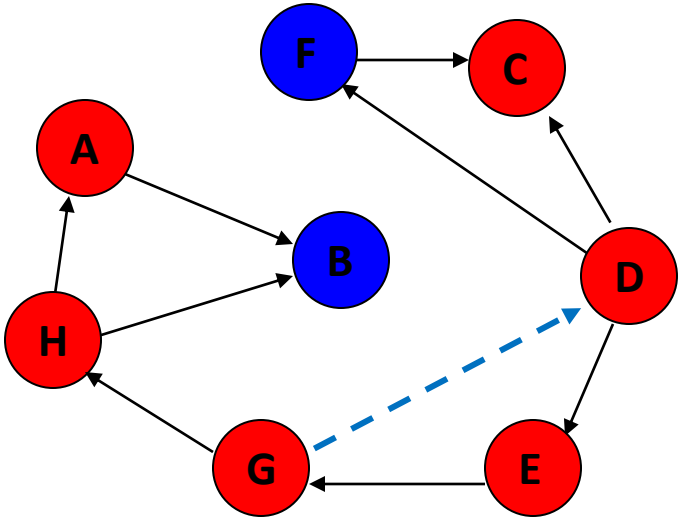


Visit A

The order nodes are visited:

D, C, E, G, H, A

Walk-Through



Visited Array

A	√
B	
C	√
D	√
E	√
F	
G	√
H	√

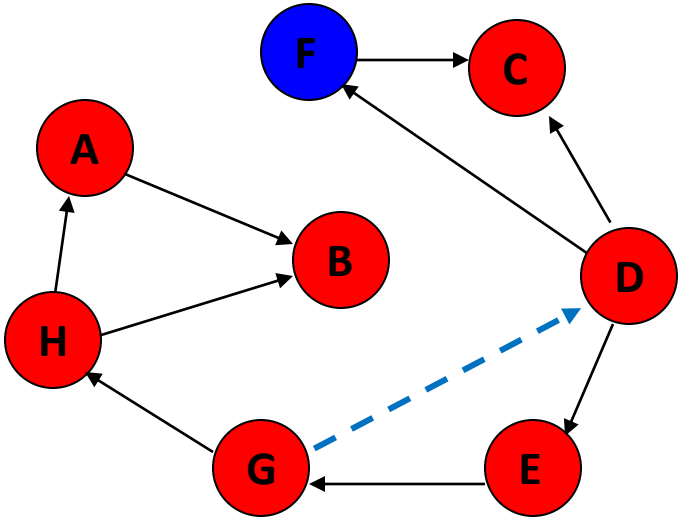
A
H
G
E
D

The order nodes are visited:

D, C, E, G, H, A

Only Node B is adjacent to A. Decide to visit B next.

Walk Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

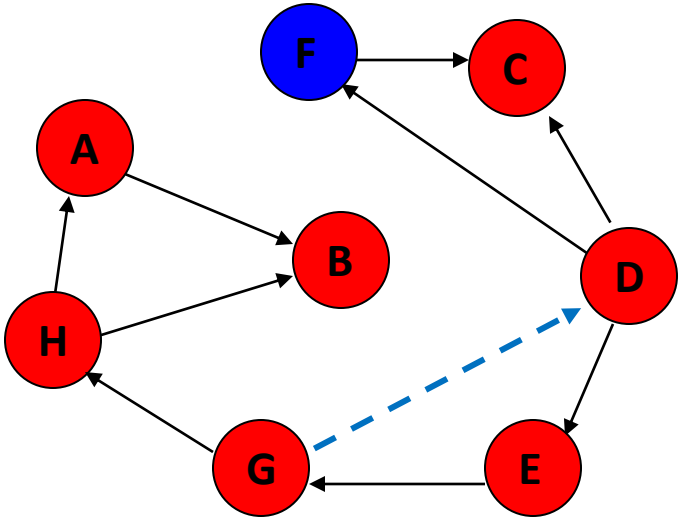
B
A
H
G
E
D

Visit B

The order nodes are visited:

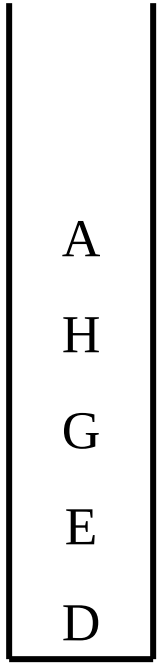
D, C, E, G, H, A, B

Walk Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

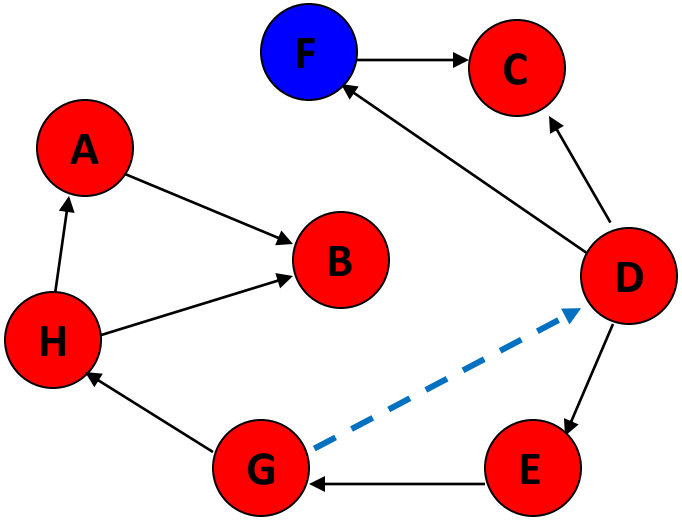


The order nodes are visited:

D, C, E, G, H, A, B

No unvisited nodes adjacent to B. Backtrack (pop the stack).

Walk Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

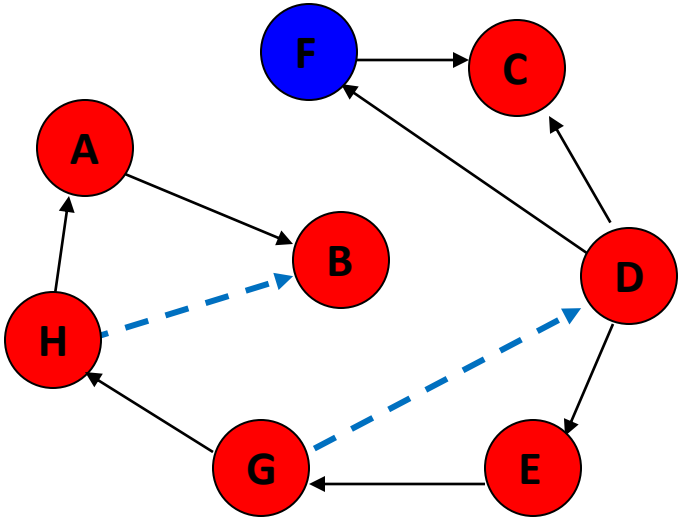
H
G
E
D

The order nodes are visited:

D, C, E, G, H, A, B

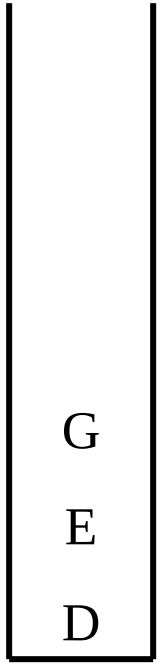
No unvisited nodes adjacent to A. Backtrack (pop the stack).

Walk Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

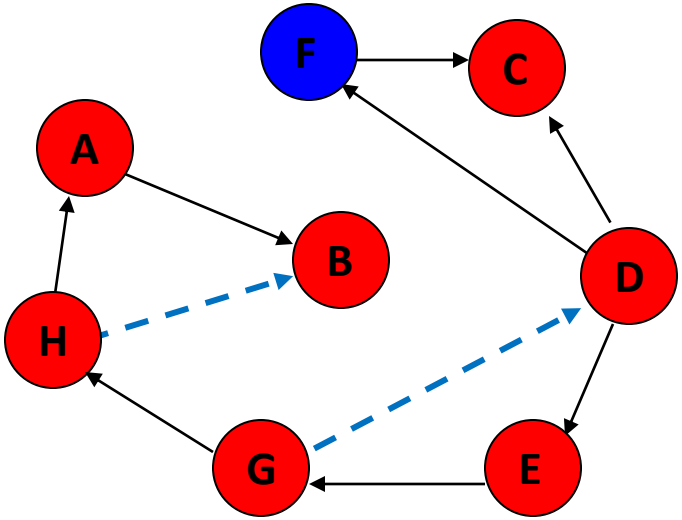


The order nodes are visited:

D, C, E, G, H, A, B

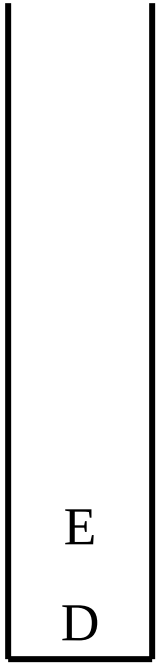
No unvisited nodes adjacent to H. Backtrack (pop the stack).

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

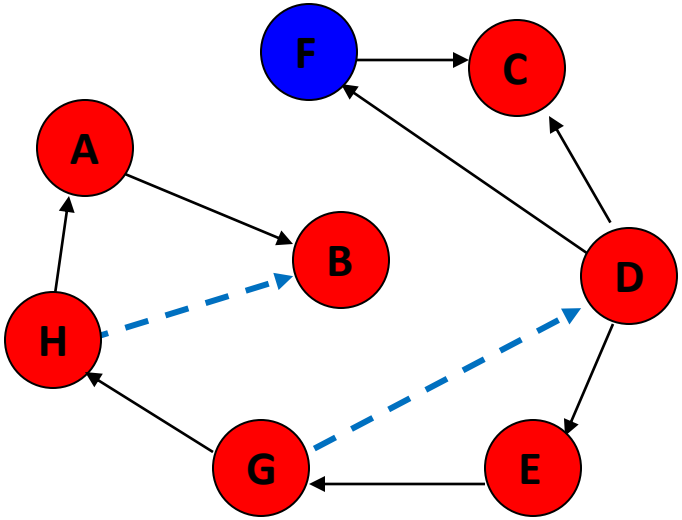


The order nodes are visited:

D, C, E, G, H, A, B

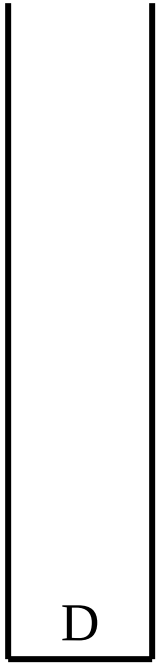
No unvisited nodes adjacent to G. Backtrack (pop the stack).

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

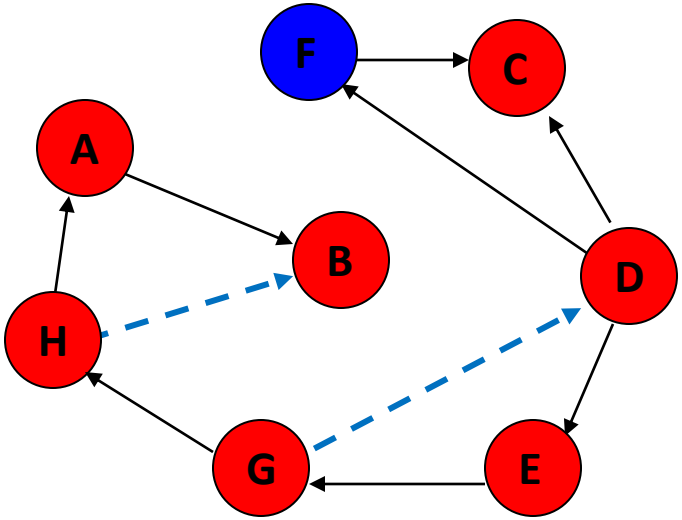


The order nodes are visited:

D, C, E, G, H, A, B

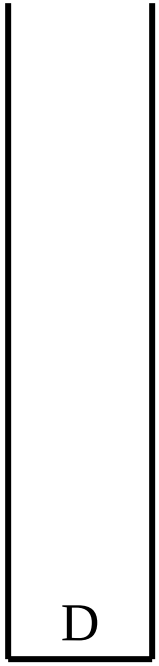
No unvisited nodes adjacent to E. Backtrack (pop the stack).

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

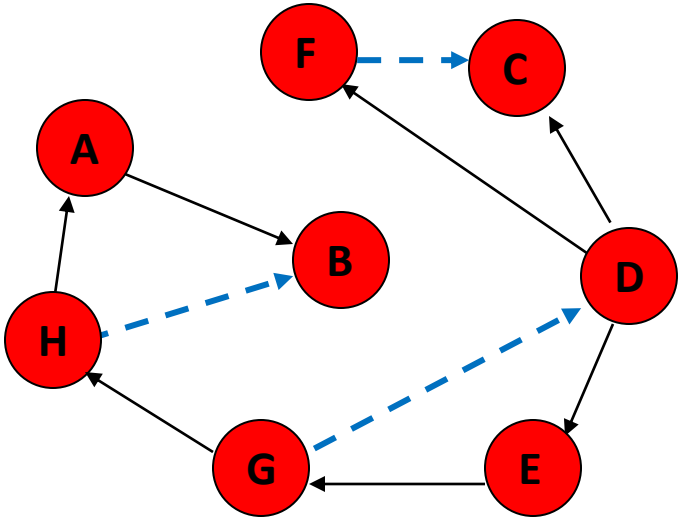


The order nodes are visited:

D, C, E, G, H, A, B

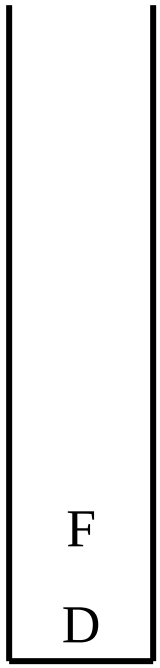
F is unvisited and is adjacent to D. Decide to visit F next.

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

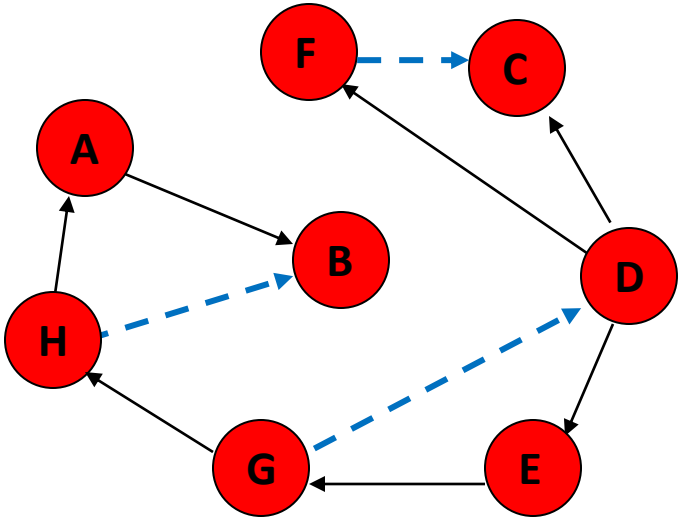


Visit F

The order nodes are visited:

D, C, E, G, H, A, B, F

Walk-Through



A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

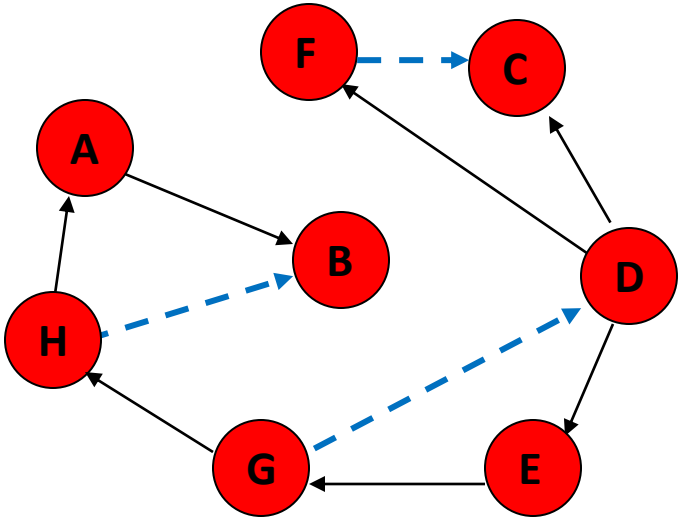
D

The order nodes are visited:

D, C, E, G, H, A, B, F

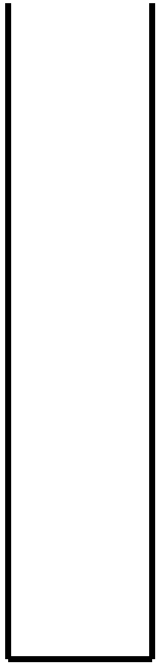
No unvisited nodes adjacent to F. Backtrack.

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

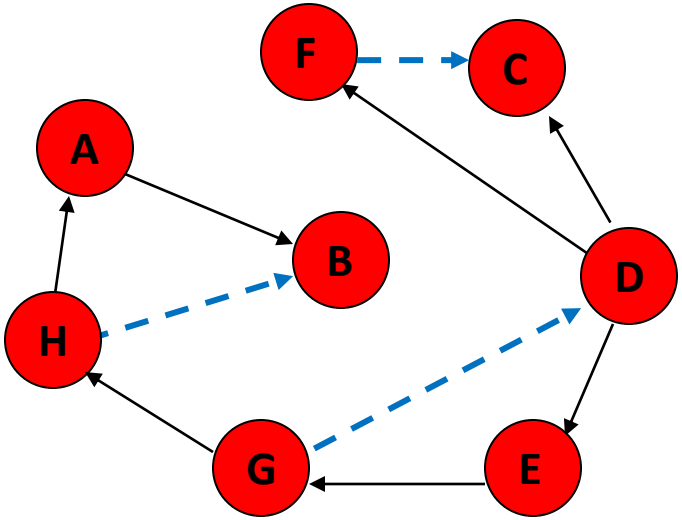


The order nodes are visited:

D, C, E, G, H, A, B, F

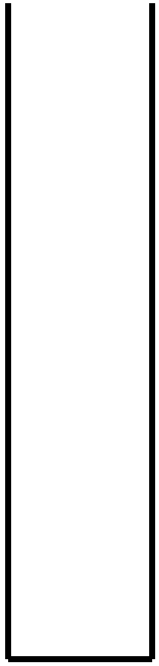
No unvisited nodes adjacent to D. Backtrack.

Walk-Through



Visited Array

A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

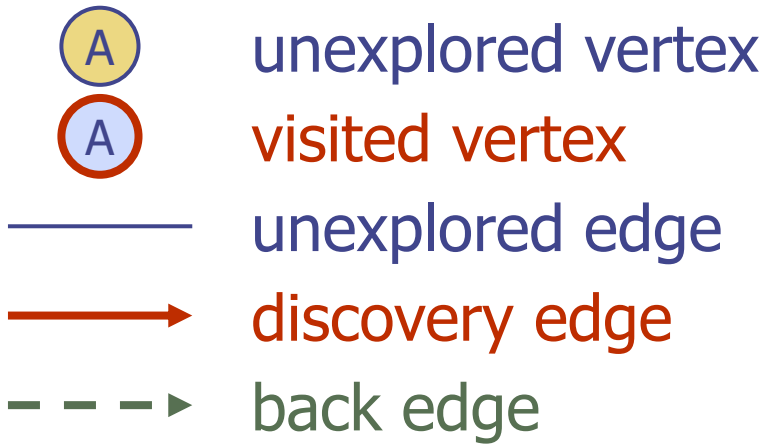


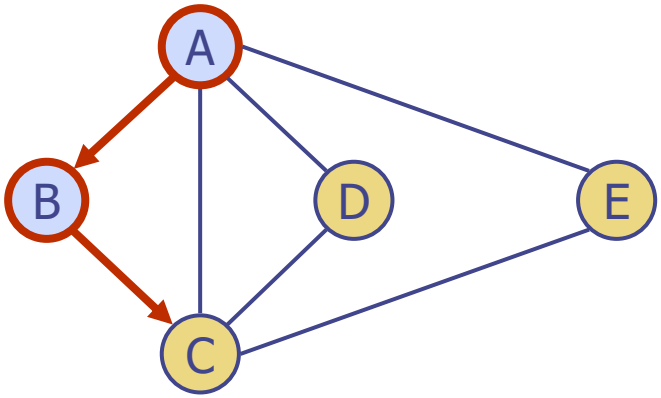
The order nodes are visited:

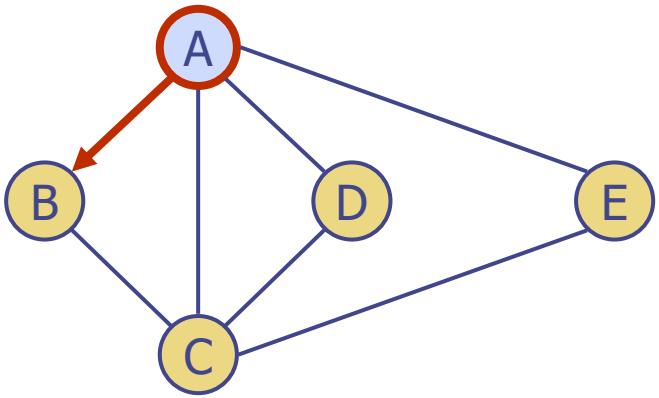
D, C, E, G, H, A, B, F

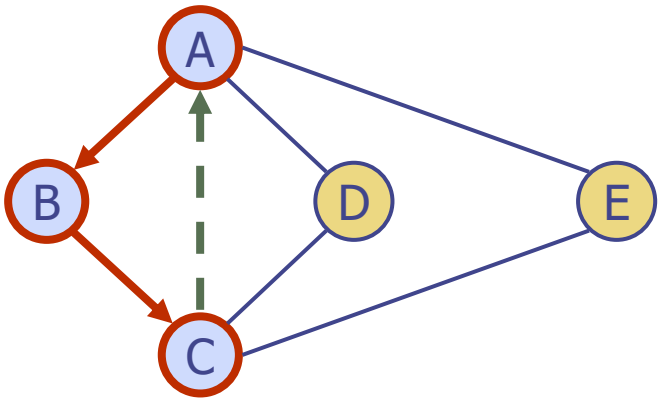
Stack is empty. Depth-first traversal is done.

DFS Example 2

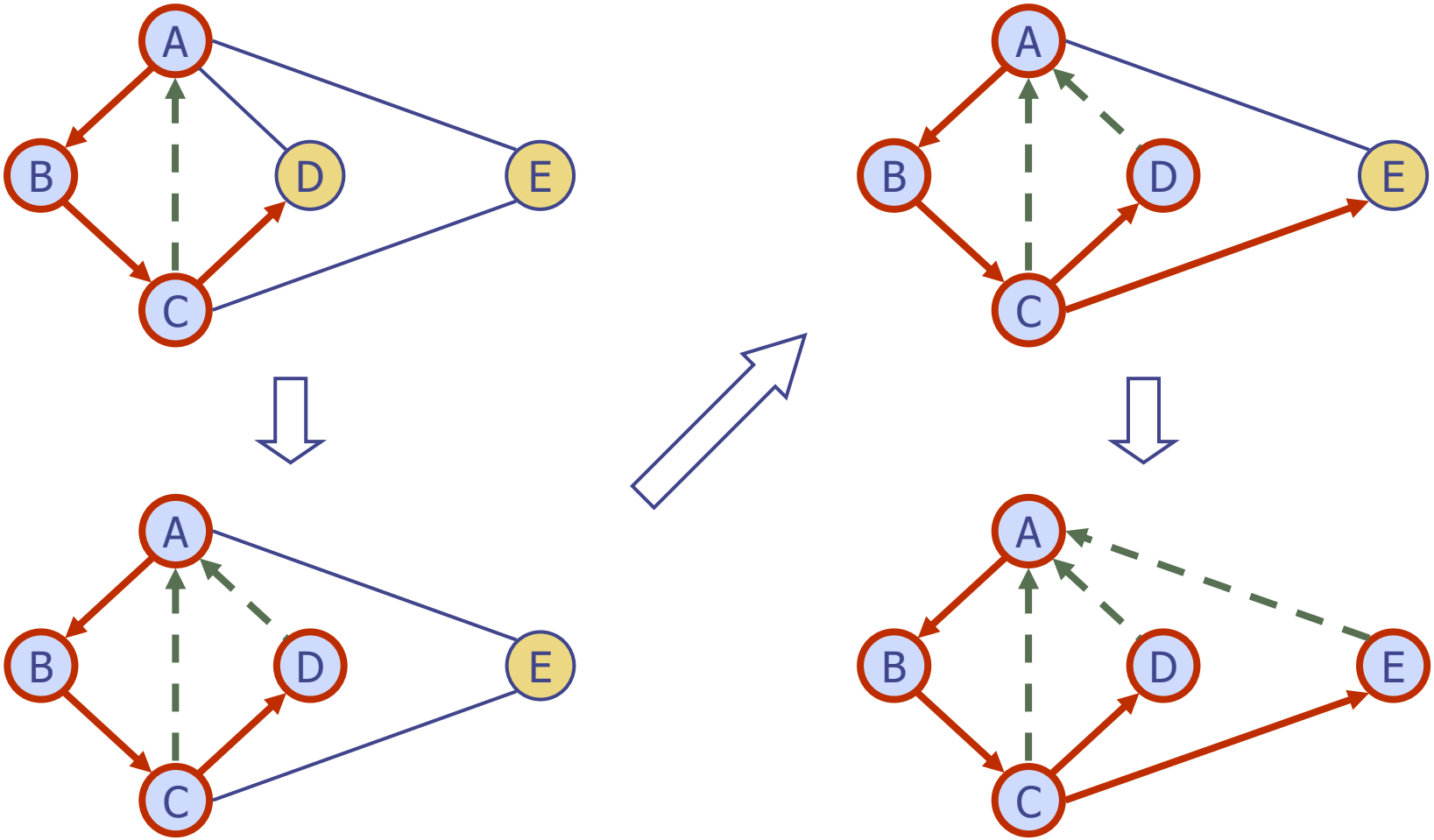








DFS Example 2



DFS Pseudocode

```
DFS-iterative (G, s):                                    //Where G is graph and s is source vertex
  let S be stack
  S.push( s )              //Inserting s in stack
  mark s as visited.
  while ( S is not empty):
      //Pop a vertex from stack to visit next
      v  =  S.top( )
     S.pop( )
     //Push all the neighbours of v in stack that are not visited
    for all neighbours w of v in Graph G:
        if w is not visited :
                 S.push( w )
                mark w as visited


DFS-recursive(G, s):
    mark s as visited
    for all neighbours w of s in Graph G:
        if w is not visited:
            DFS-recursive(G, w)
```

The time complexity of DFS is **O(V+E)** where V is the number of nodes and E is the number of edges.

DFS Applications

- Let G be a graph with n vertices and m edges represented with the adjacency list structure. A DFS traversal of G can be performed in O(n + m) time. Also, there exist O(n + m) -time algorithms based on DFS for the following problems:
  - Testing whether G is connected
  - Computing a spanning forest of G
  - Computing the connected components of G
  - Computing a path between two vertices of G , or reporting that no such path exists *[Ex: Good algorithm design exercise]*
  - Computing a cycle in G , or reporting that G has no cycles. *[Ex: Good algorithm design exercise.]*

Binary Search Tree (BST)

innovate achieve lead

Why Binary Search Tree?

- We have already discussed, different tree representations and in all of them we *did not* impose any restriction on the nodes data !!
- As a result, to search for an element we need to check both in left subtree and in right subtree.
- Due to this, the worst case complexity of search operation is O(n).

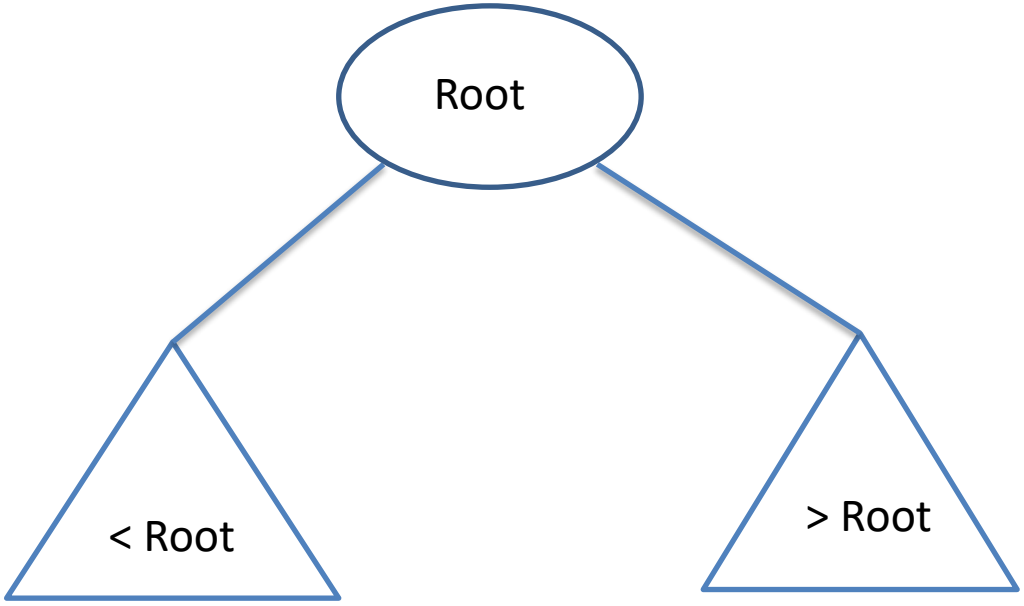
Binary Search Trees (BSTs)

- As the name suggests, the main use of this representation is for searching. In this representation we impose restriction on the kind of data a node can contain.
- As a result, it reduces the worst case average search operation to log n

Binary Search Tree - Property

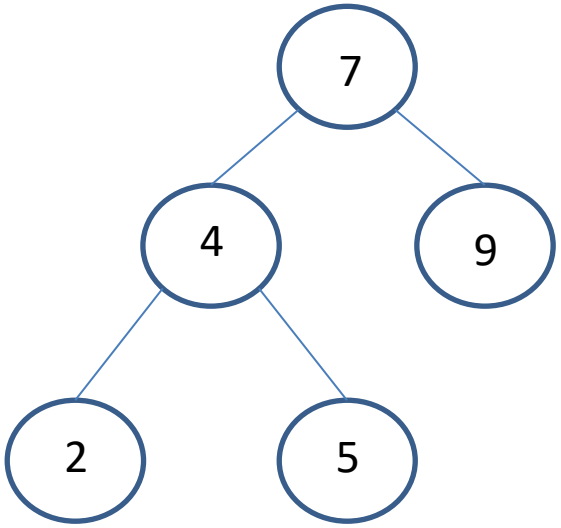
- In binary search tree, *all the left subtree elements should be less than root data and all the right subtree elements should be greater than root data*.
  - This is called binary search tree property, note that, this property should be satisfied at **every node** in the tree.
    - The left subtree of a node, contains only nodes with keys **less** than the nodes key.
    - The right subtree of a node, contains only nodes with keys **greater** than the nodes key.
    - Both the left and right subtrees must also be binary search trees.

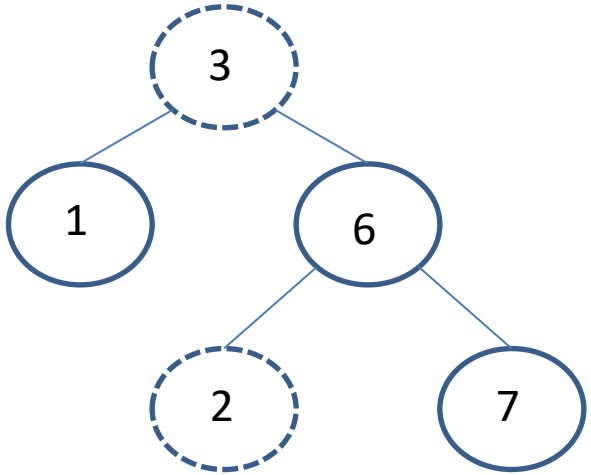
BST Property Diagrammatic Representation



Note: Many textbooks would depict that a BST cannot have duplicates and the = is not discussed. Nevertheless, refer to Introduction to Algorithm by CLRS – Chapter 12.

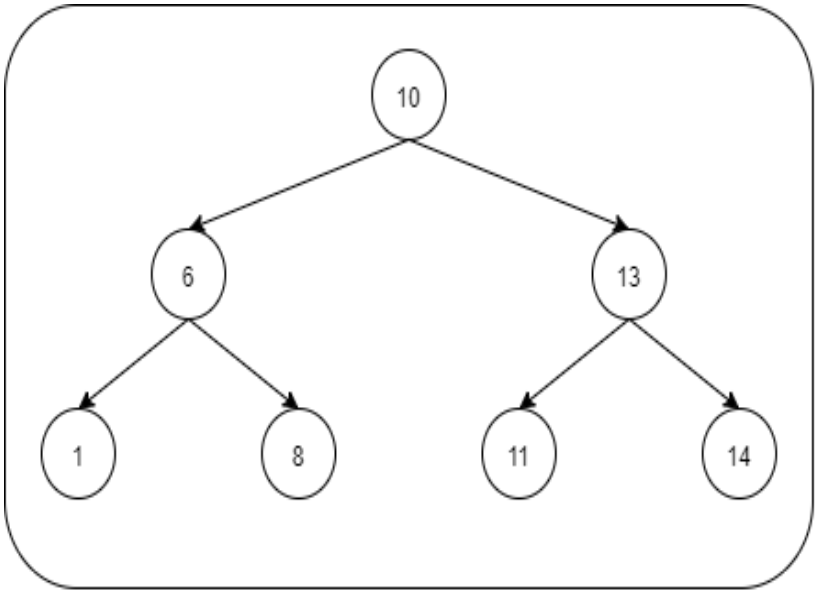
Example

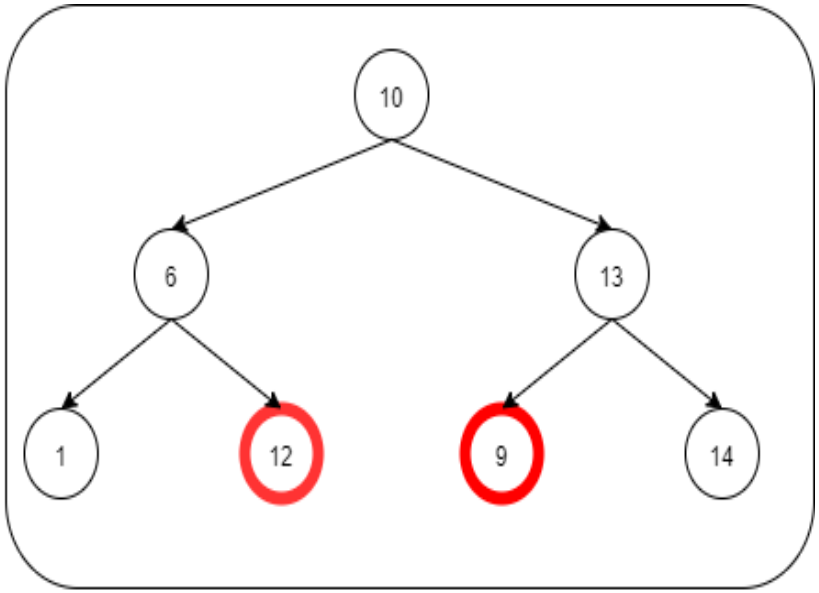




Example: The left tree is a binary search tree and right tree is not binary search tree [at node 6 it's not satisfying the binary search tree property]

Binary Search Trees

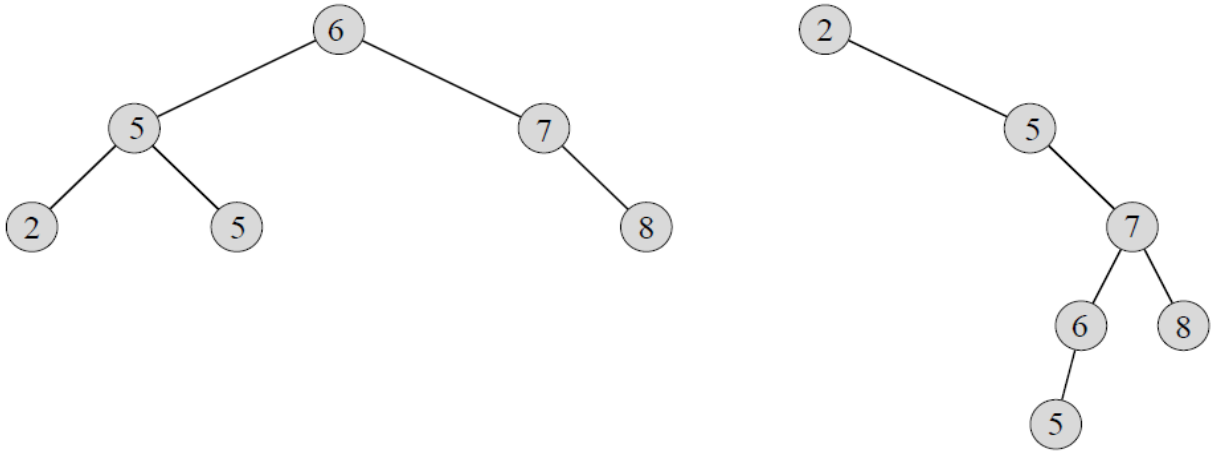




BST with Duplicate Keys

The keys in a binary search tree are always stored in such a way as to satisfy the binary search tree property :

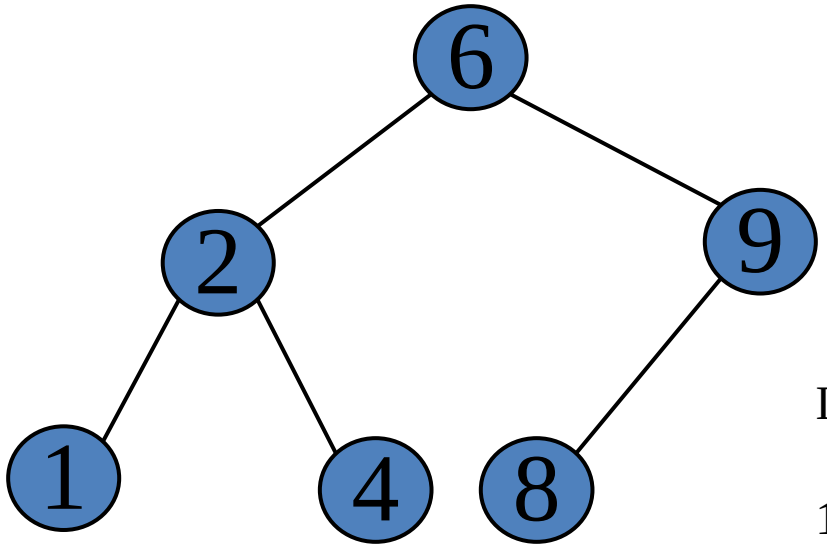
Let x be a node in a binary search tree. If y is a node in the left subtree of x, then y.key <= x.key. If y is a node in the right subtree of x, then y.key >=x.key.



(a) (b)

BST – Inorder gives Sorted!

- An Inorder traversal of a binary search trees visits the keys in increasing order.
- Outputs keys in sorted order.



Inorder Traversal (Left, root, right)

1,2,4,6,8,9 ➔ Sorted !

Operations on BST

Main Operations: Following are the main operations that are supported by binary search trees:

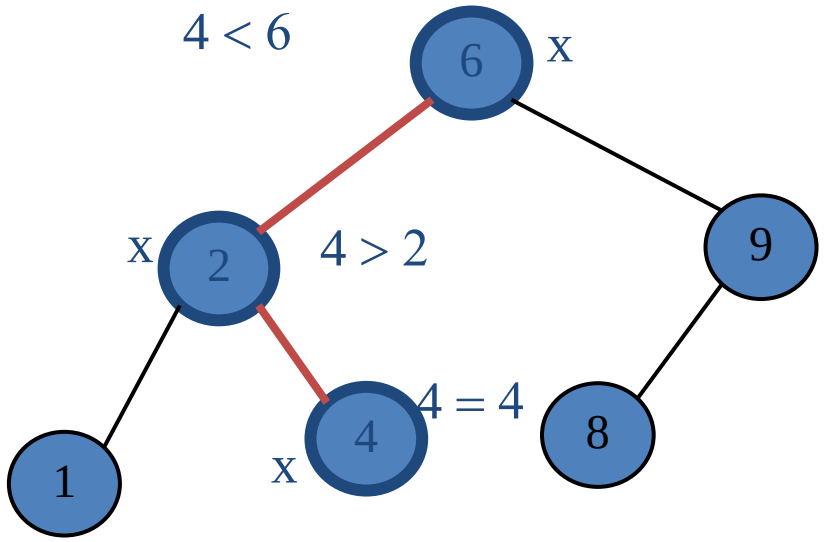
- **Find**/find minimum/find maximum element in binary search trees.
- **Inserting** an element in binary search trees.
- **Deleting** an element from binary search trees.

Important Notes on Binary Search Trees

- Since root data is always between left subtree data and right subtree data, performing Inorder traversal on binary search tree produces a sorted list.
- While solving problems on binary search trees, first we process left subtree, then root data and finally we process right subtree. This means, depending on the problem only the intermediate step (processing root data change and we do not touch the first and third steps)
- If we are searching for an element x, then at each internal node we compare the value of the current node to our search element x. If the answer is “smaller” then the search continues in the left subtree. If the answer is “greater” then the search continues in the right subtree. If the answer is “equal” then the search terminates successfully. Finally, if we reach an external empty node, then search terminates unsuccessfully.
- In other words, the binary search trees consider either left or right subtrees for searching an element but not both.

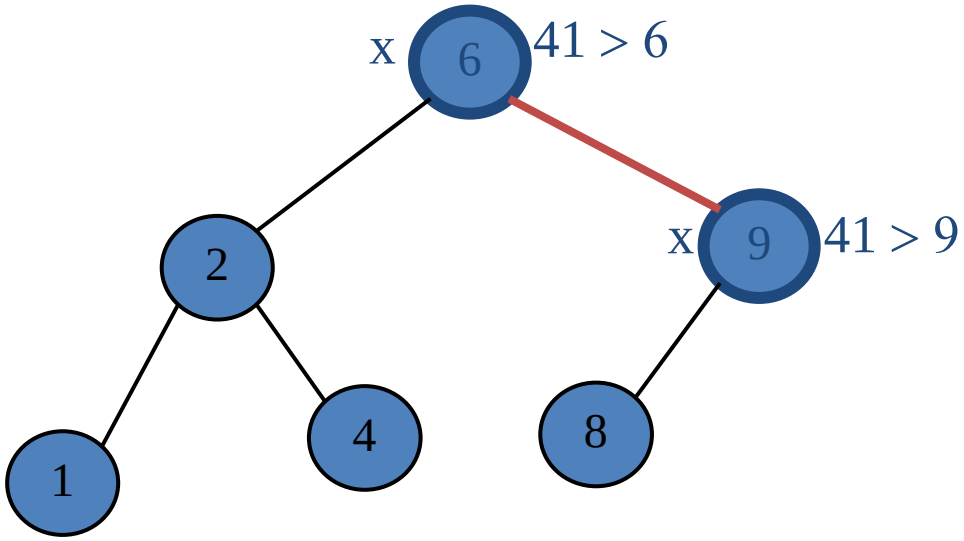
Finding an element in Binary Search Trees

- Find operation is straight forward in a BST. Start with the root and keep moving left or right using the BST property.
- If the data we are searching is same as nodes then we return current nodes. if the data is not present, we end up in a Null Link.
- Find/Search element 4

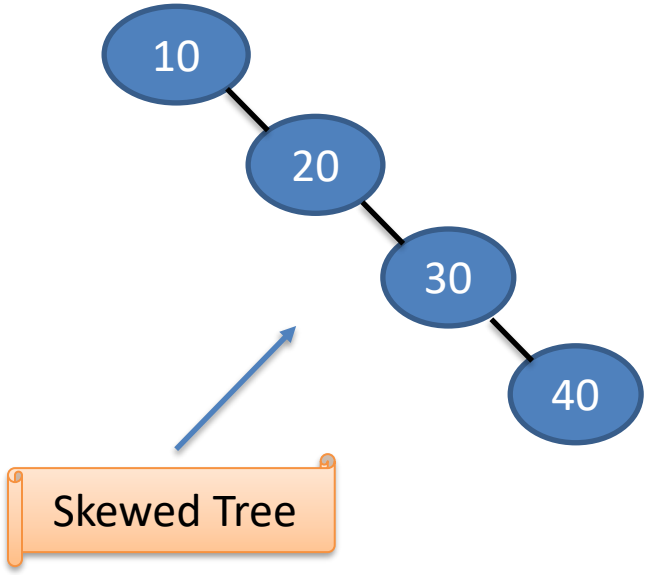


Find in BST

Search element 41



Search element 41



What is the time complexity of finding the element in binary search tree ?

O(n) *Why?!*

Finding element in BST

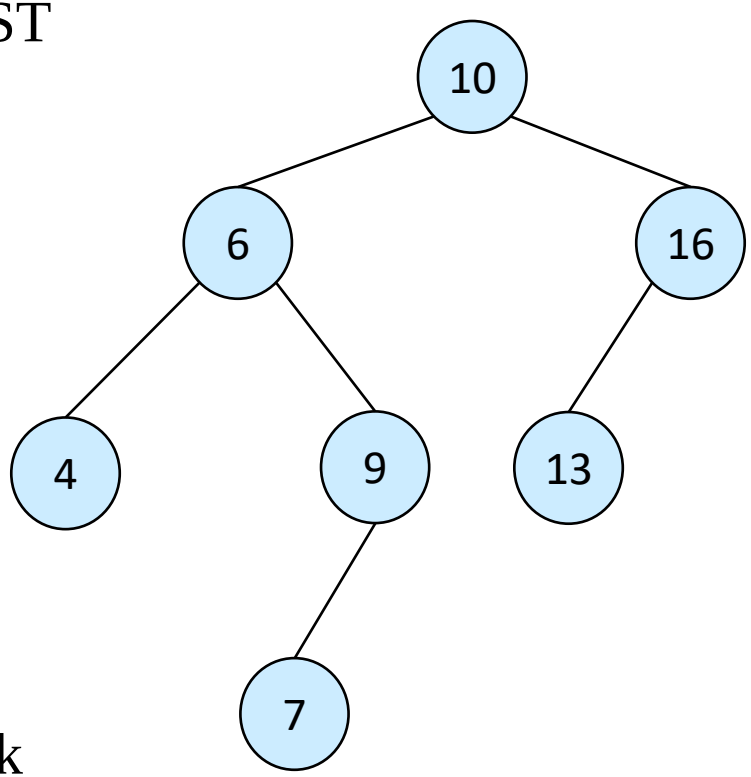
innovate achieve lead

```
Struct Binary Search TreeNode*find(struct
Binary search tree node* root,int data){
If(root == Null)
     return Null;
If(data < root → data)
     return find (root → left,data);
Else if (data > root → data)
     return find (root → right, data);
     return root ;
}
```

Time Complexity : O(n), in worst case (when BST is a skew tree)
Space complexity : O(n), for recursive stack.

Finding minimum element in Binary search trees

In BSTs, the minimum element, is the left-most node, which does not has left child. In the BST below, the minimum element is 4.



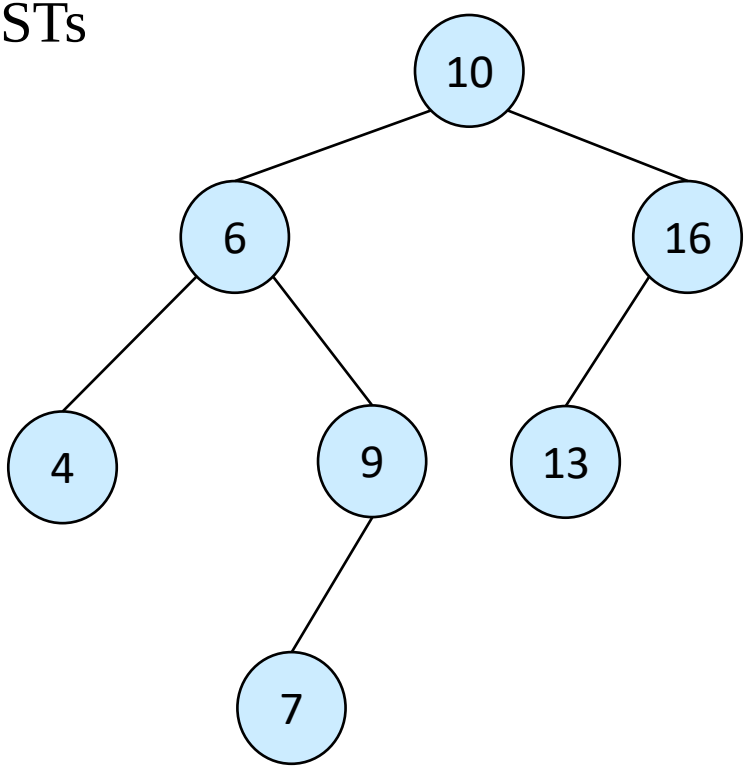
Tree-Minimum(x)

1. **while** $left[x] \neq NIL$
2. **do** $x \leftarrow left[x]$
3. **return** x

- Time complexity O(n) in the worst case
- Space complexity O(n) for recursive Stack

Finding max element in Binary search trees

In BSTs the Maximum element is the right –most node which does not have right child. In the BSTs above the maximum element is 16.



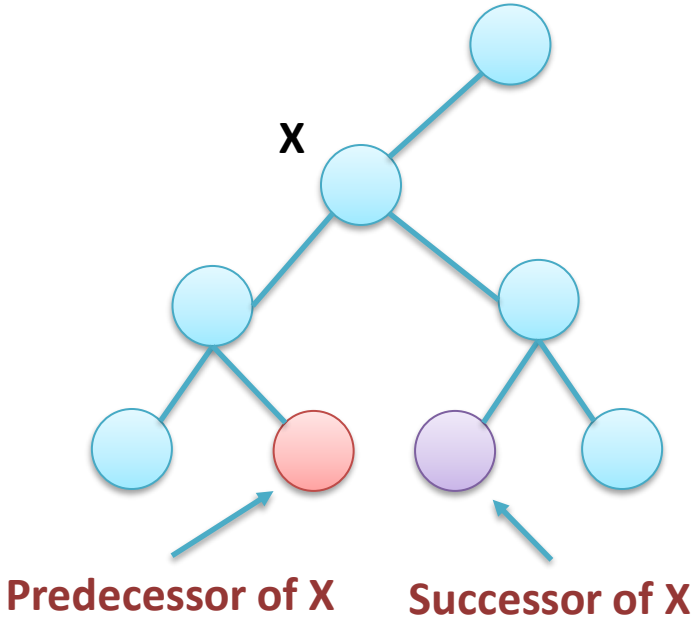
Tree-Maximum(*x*)

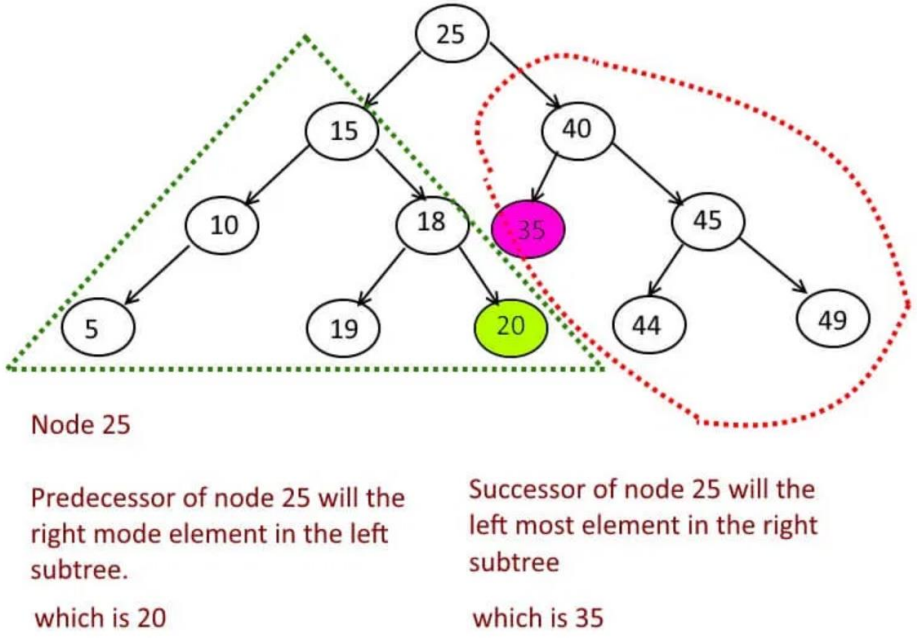
1. **while** *right*[*x*] ≠ *NIL*
2. **do** *x* ← *right* [*x*]
3. **return** *x*

- Time complexity O(n) in the worst case
- Space complexity O(n) for recursive Stack

Where is In-order Predecessor & Successor ?

If X has two children then its in-order predecessor is the maximum value in its left subtree and its in-order successor is the minimum value in its right subtree.





Note : When the node X does not have children, how to find the predecessor ? Refer CLRS

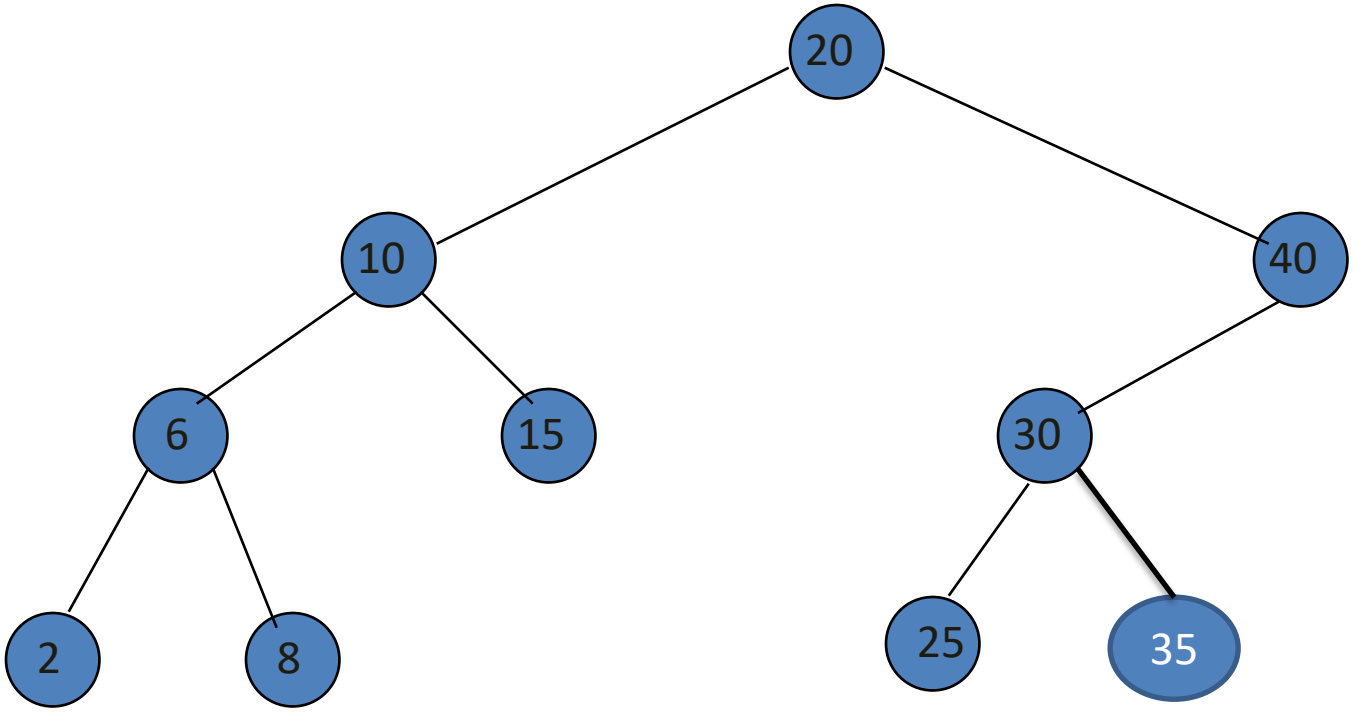
Insertion in BST

Insert k in a BST:

- Begin at the root
- Compare the element k with the element at current node
- If value we want to insert is less than key of current node, go to the left subtree
- Otherwise go to the right subtree
- If we reach a NULL, create a node with the value we are inserting and place it here

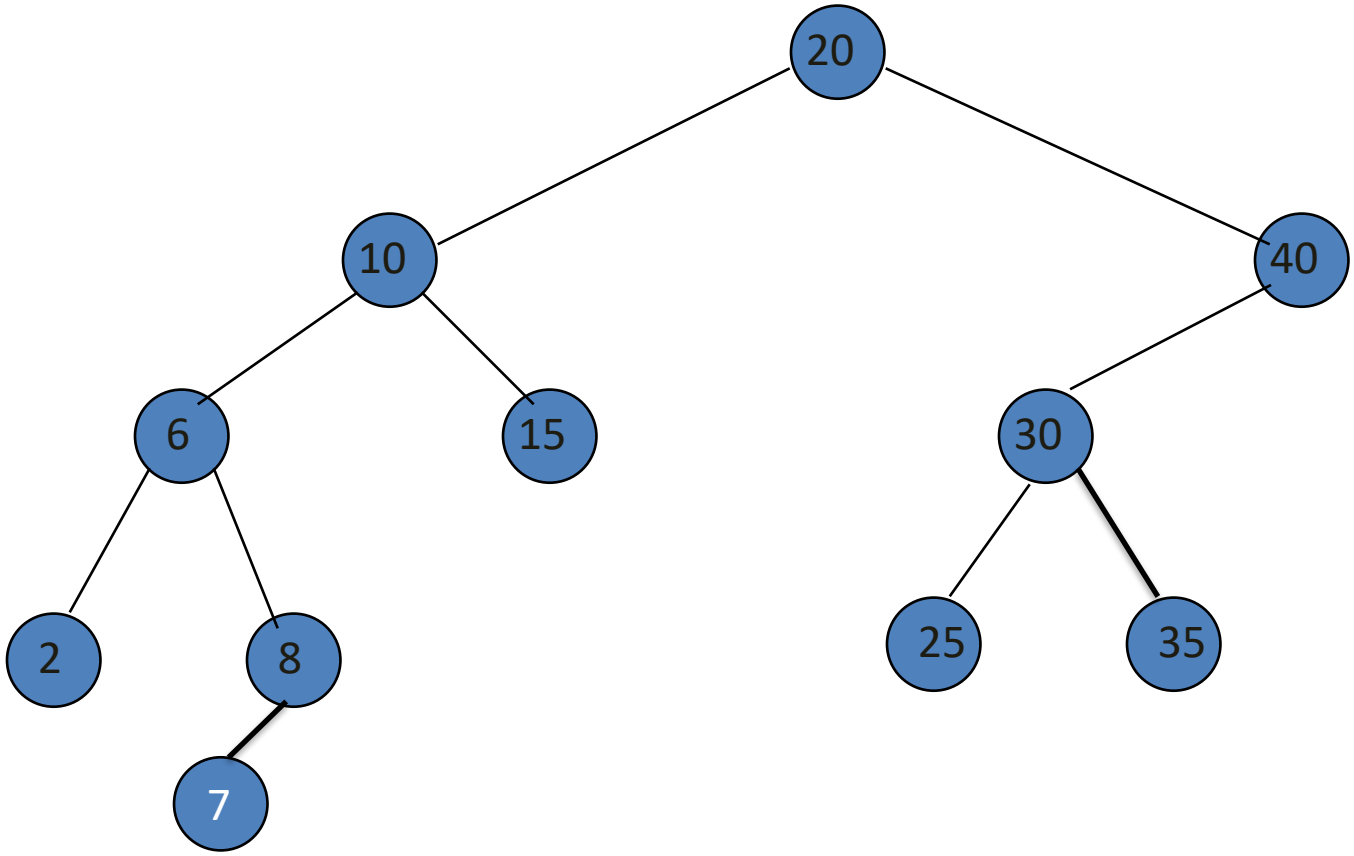
Insertion in BST Example

Insert element 35 in the given BST and draw the resulting BST



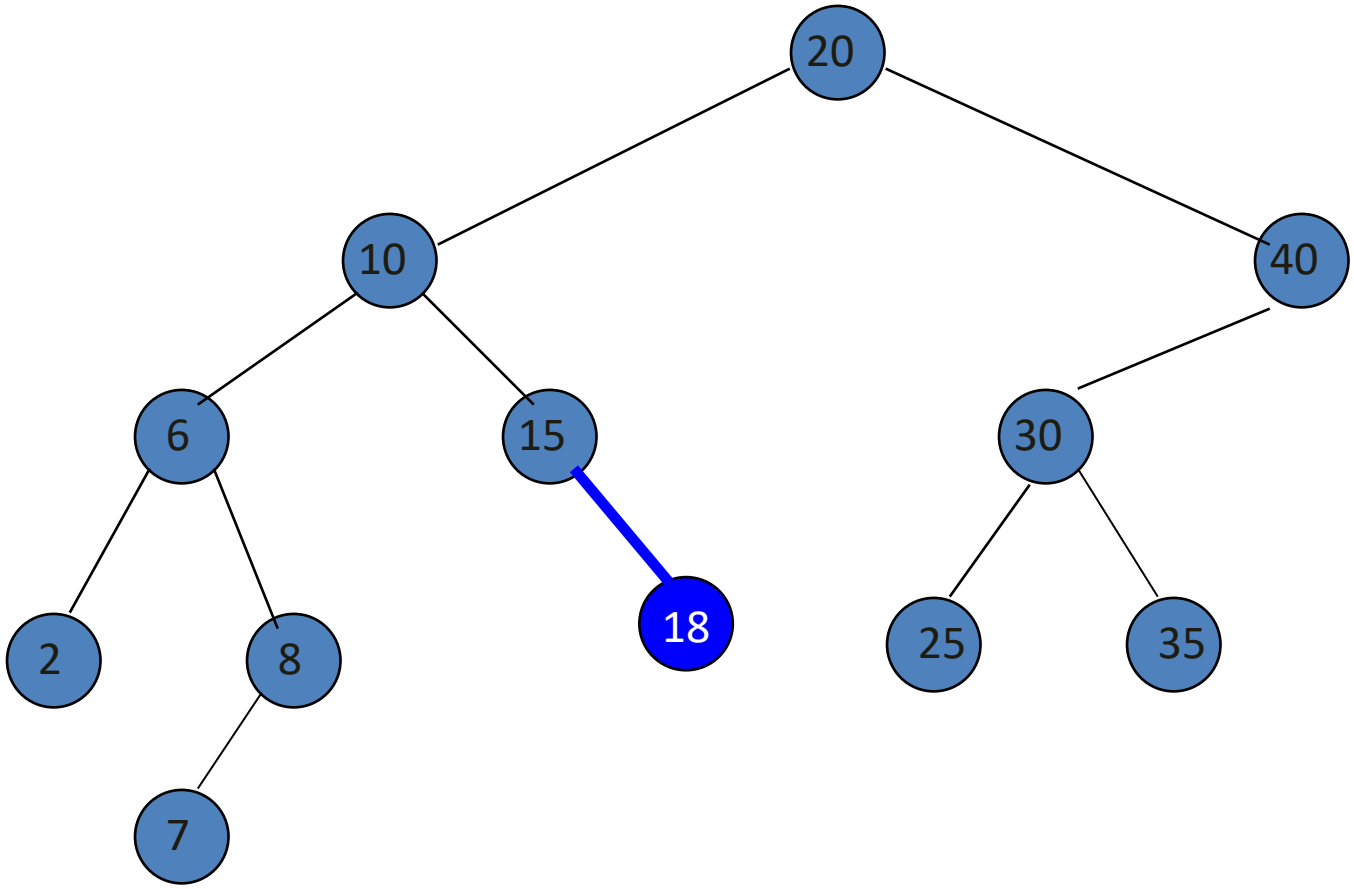
Insertion in BST Example

Insert element 7 in the given BST and draw the resulting BST



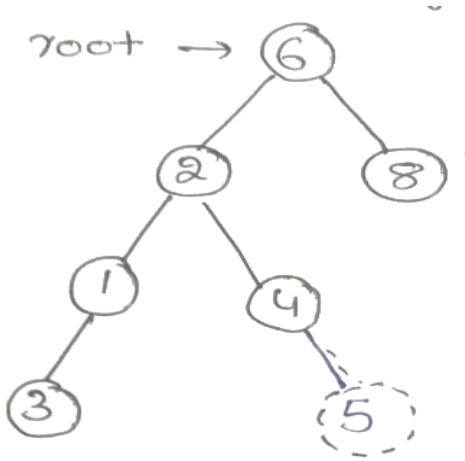
Insertion in BST Example

Insert element 18 in the given BST and draw the resulting BST



Analysis of Insertion in BST

- To insert data into binary search tree, first we need to find the location for that element. We can find the location of insertion by following the same mechanism of that of find operation.
- While finding the location if the data is already there then we can simply neglect and come out.
- Otherwise, insert data at the last location on the path traversed.
- As an example, let us consider the tree:
- The dotted node indicates the element (5) to be inserted, to insert 5, traverse the tree as using the find function. At node with key 4, we need to go right but there is no subtree, so 5 is not in the tree, and this is the correct location for insertion.
- Time complexity : O(n), in worst case



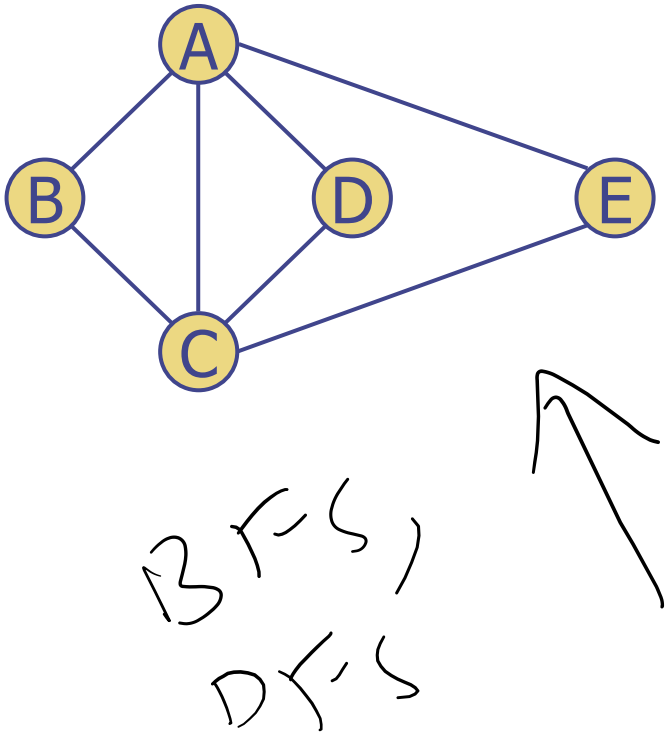
Deletion in BST

Delete k

- Search k
- If k is present
  - Deletion involves three cases
    - Node has no children (leaf)
    - Node has one child
    - Node has two children
- If k not present
  - Return a message as element not present

Exercise 1

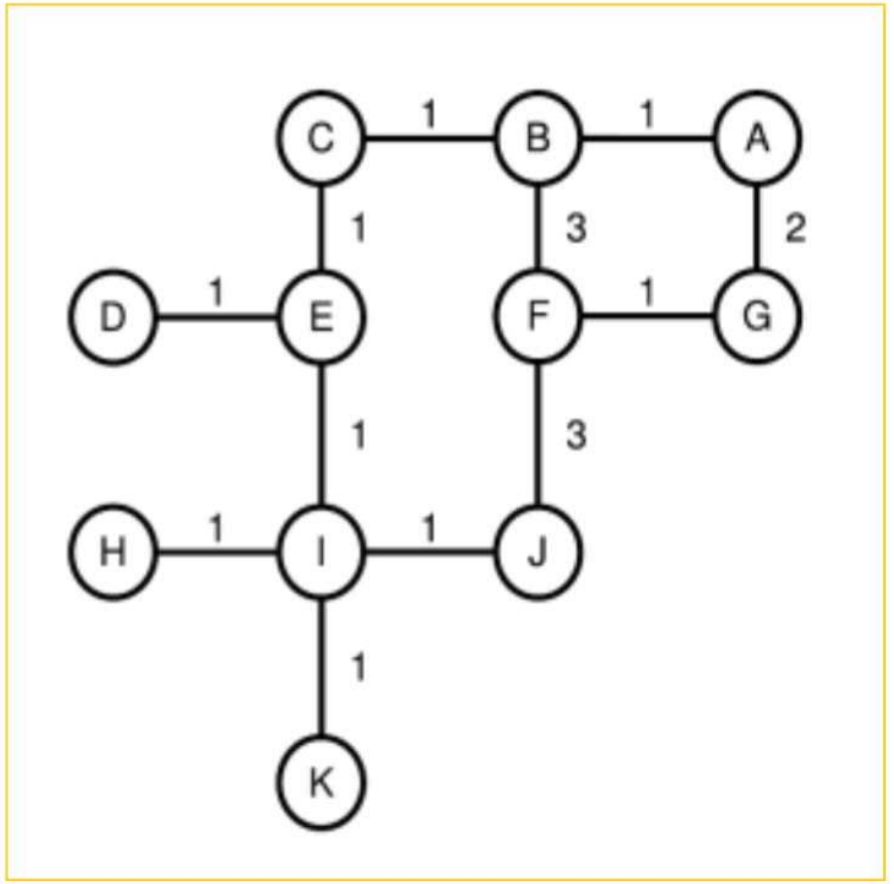
- Perform BFS and DFS on the below and assume A as the start node. Also show the final result and the cross-edges.



Exercise 2

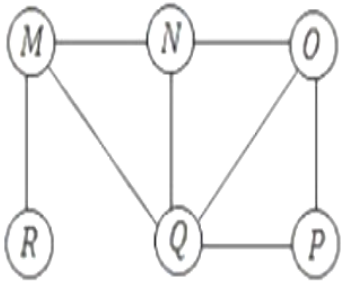
List the labels in the order they would be visited when performing:

- Breadth First Search
- Depth First Search
- Source vertex: A
- Assume that neighbors of the same node are visited in alphabetical order



Exercise 3

The Breadth First Search (BFS) algorithm has been implemented using the queue data structure. Which one of the following is a possible order of visiting the nodes in the graph below?



A MNOPQR

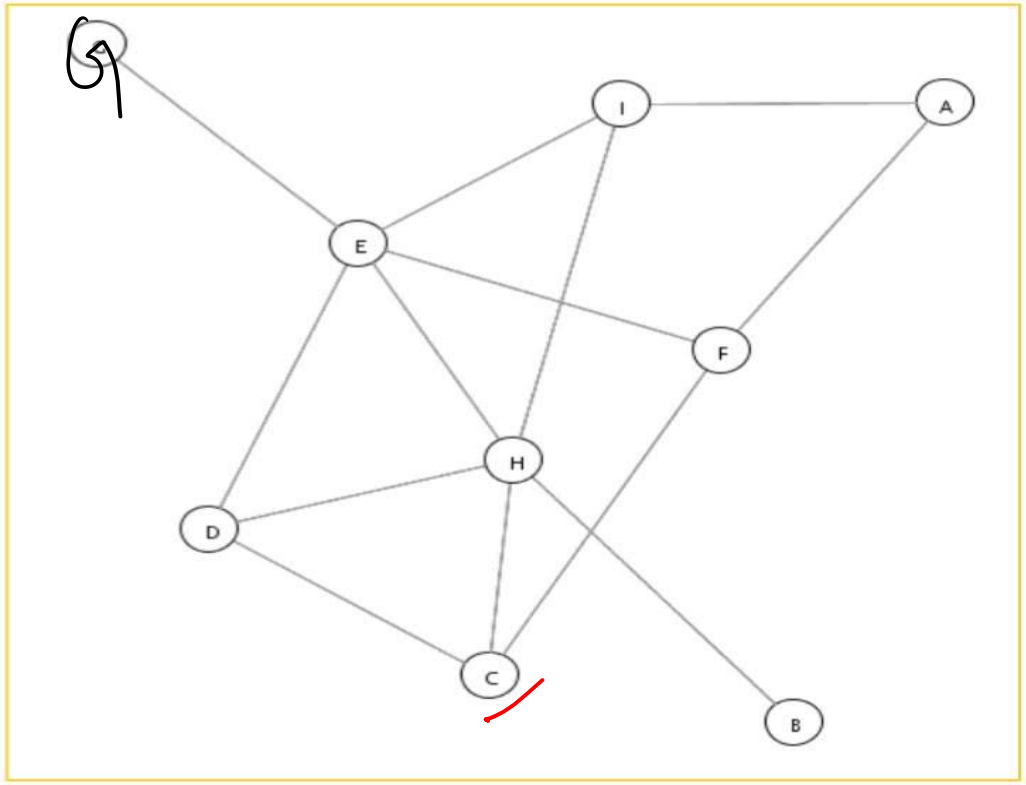
B NQMPOR

C QMNROP

D POQNMR

Exercise 4

- Implement both DFS and BFS traversal. Assume C to be the source vertex.



See you in the next class to explore more on Hashing !

Thank You for your time & attention !

Contact : parthasarathypd@wilp.bits-pilani.ac.in