Unit-III
Basic traversal & Search Techniques

K. RAGHAVA RAO

Professor in CSE

KL University

Krraocse@gmail.com

http://mcadaa.blog.com

Techniques for binary trees

- In a traversal of a binary tree, each element of the binary tree is visited exactly once.

- During the visit of an element, all action (make a clone, display, evaluate the operator, etc.) with respect to this element is taken.

Techniques for binary trees.

- L: moving left.
- D: printing the data.
- R: moving right.
- Six possible combination : LDR, LRD, DLR, DRL, RDL, RLD.
- Left before right : LDR(inorder), LRD(postorder), DLR(preordr)

Techniques for binary trees.

Binary Tree Traversal Methods

- Preorder(root,left,right)
- Inorder(left,root,right)
- Postorder(left,right,root)
- Level order

Techniques for binary trees.

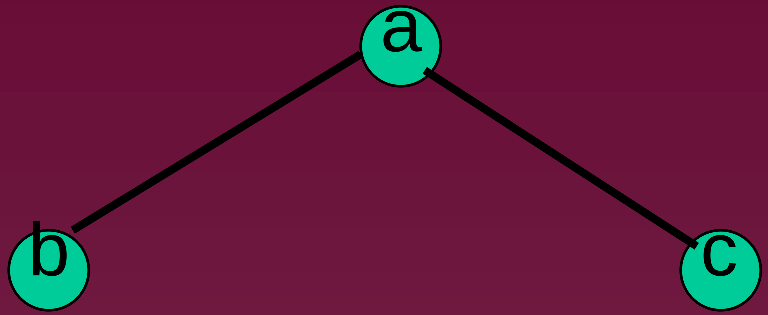
Preorder, Postorder and Inorder Algorithms

Algorithm *Preorder*(x)
Input: x is the root of a subtree.
if $x \neq$ NULL
  then output key(x);
    Preorder(left(x));
    Preorder(right(x));

Algorithm *Postorder*(x)
Input: x is the root of a subtree.
if $x \neq$ NULL
  then *Postorder*(left(x));
    Postorder(right(x));
    output key(x);

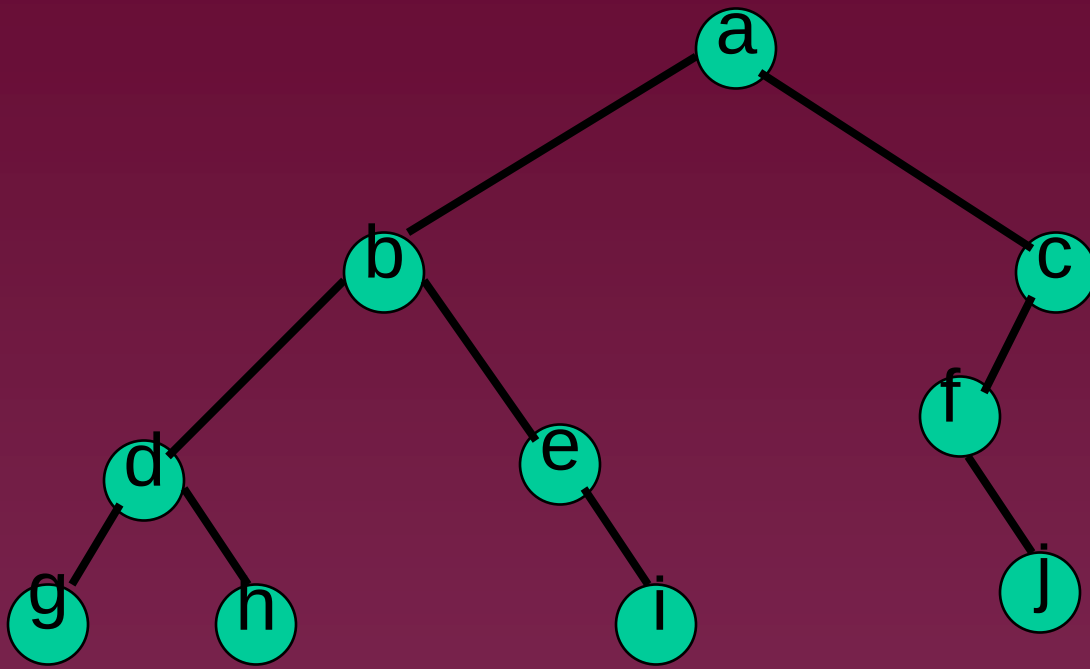
Algorithm *Inorder*(x)
Input: x is the root of a subtree.
if $x \neq$ NULL
  then *Inorder*(left(x));
    output key(x);
    Inorder(right(x));

Preorder Example (visit = print)



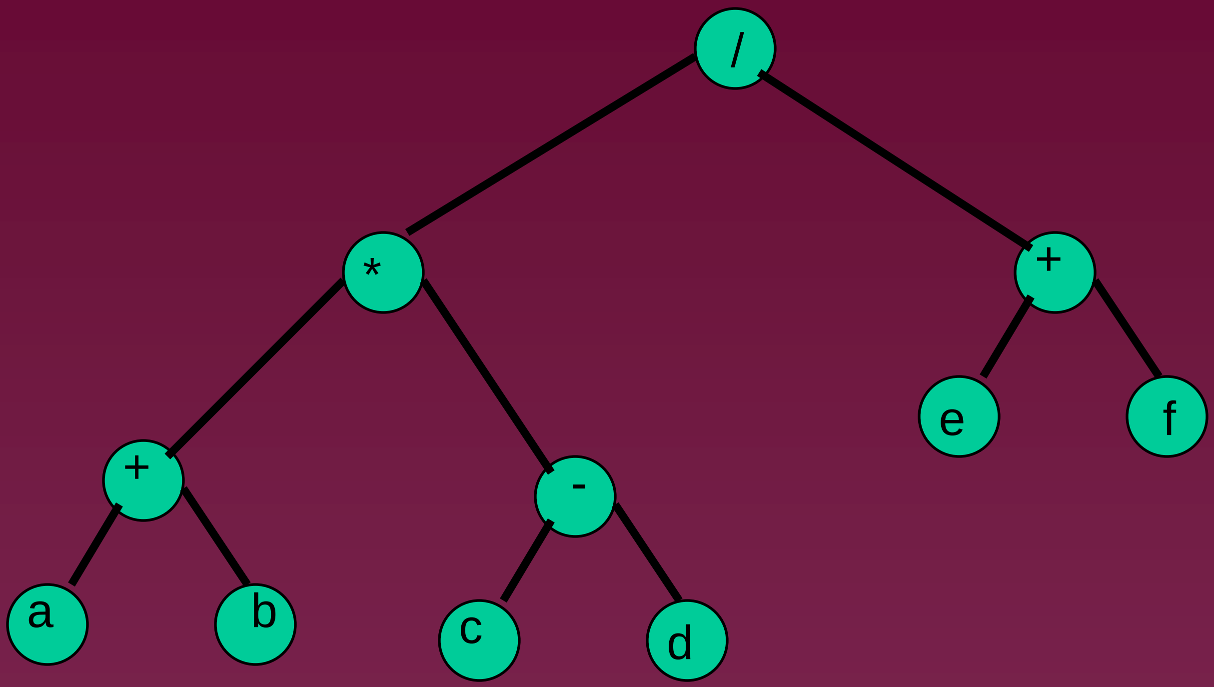
a b c

Preorder Example (visit = print)



a b d g h e i c f j

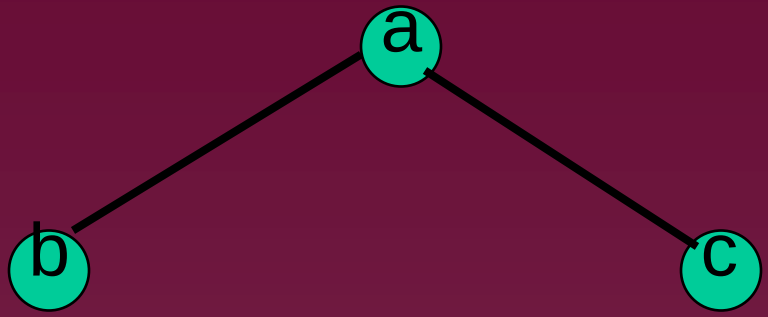
Preorder Of Expression Tree



/ * + a b - c d + e f

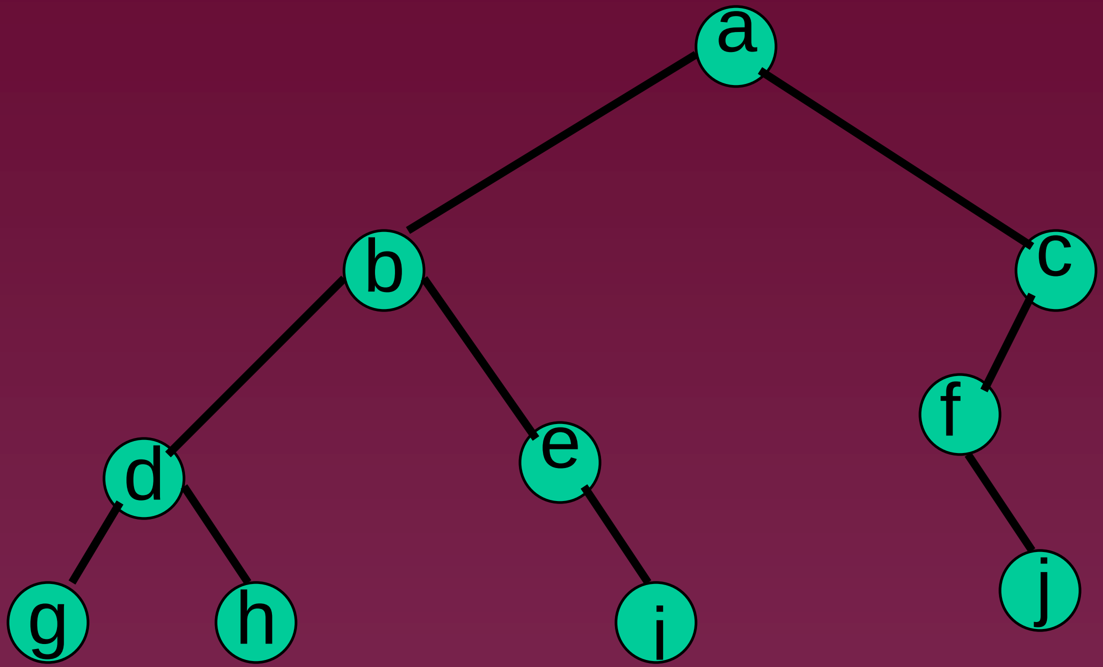
Gives prefix form of expression!

Inorder Example (visit = print)



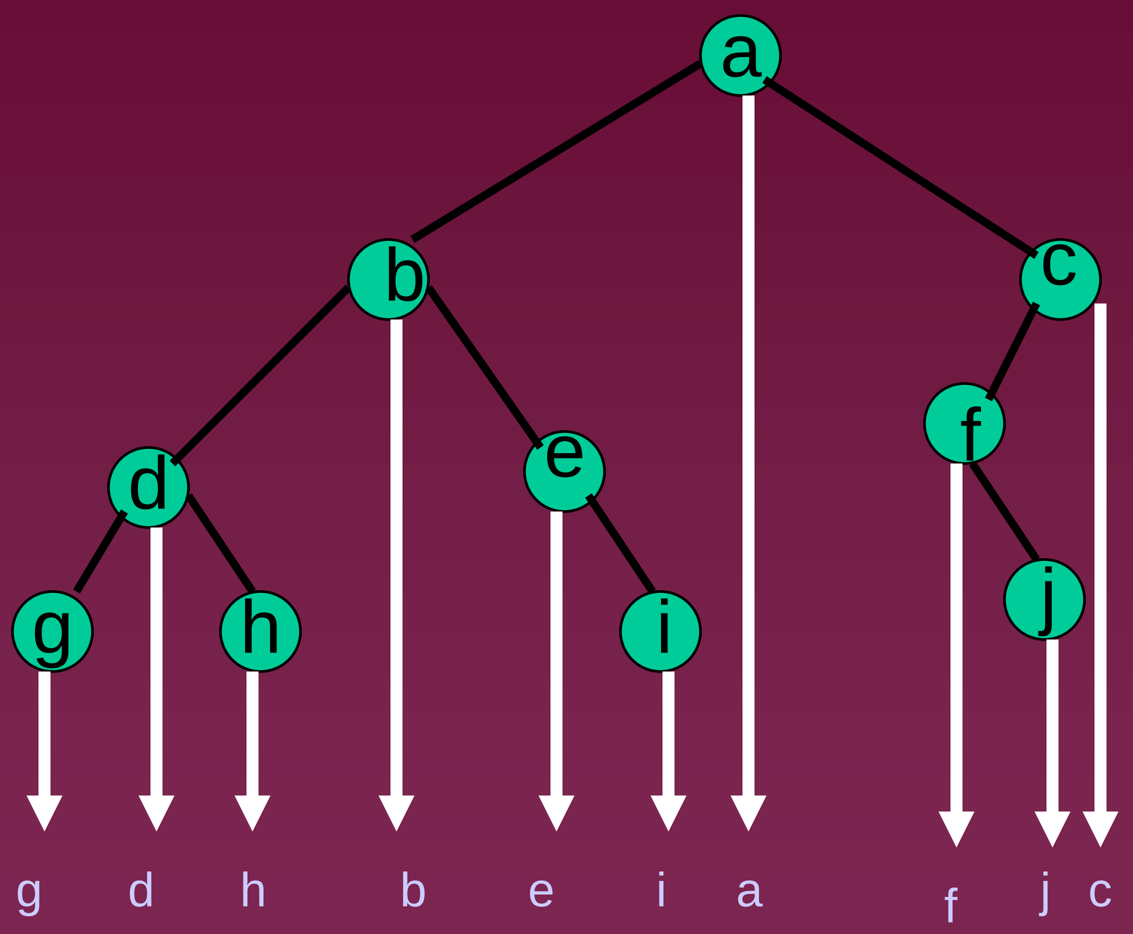
b a c

Inorder Example (visit = print)

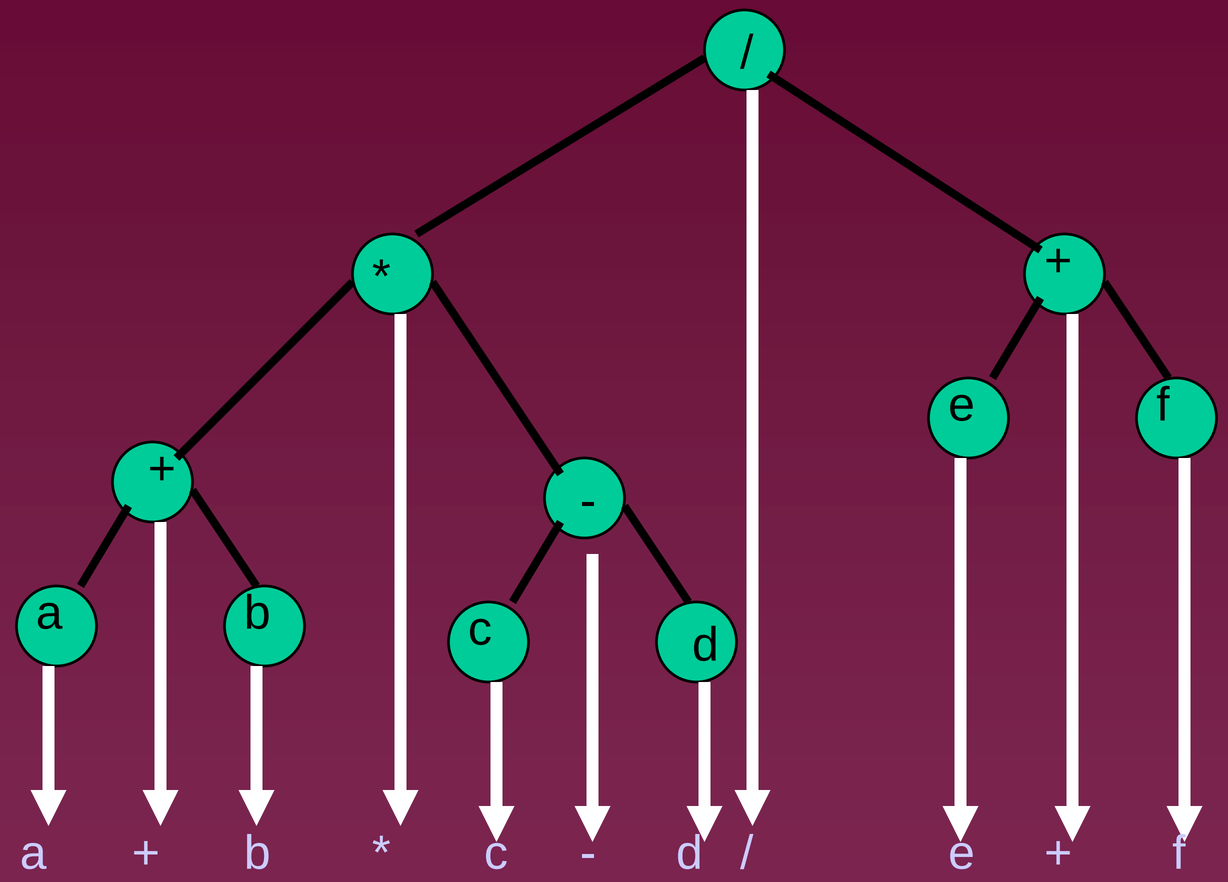


g d h b e i a f j c

Inorder By Projection (Squishing)

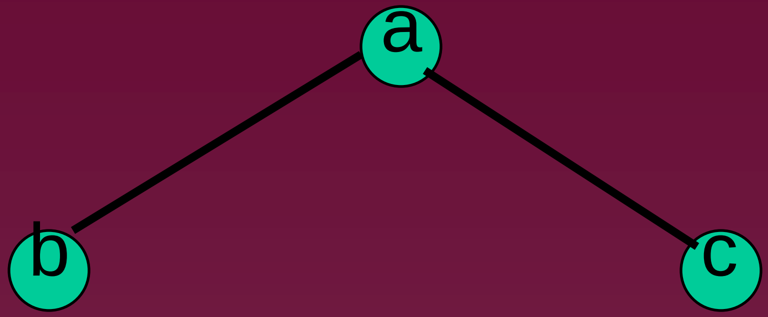


Inorder Of Expression Tree



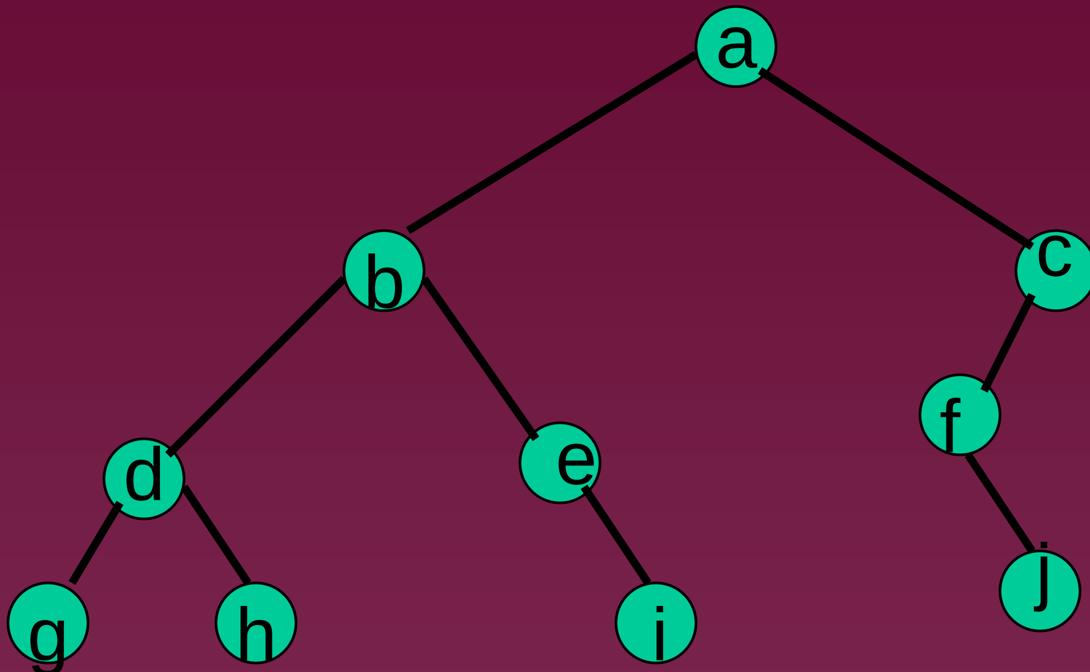
Gives infix form of expression (sans parentheses)!

Postorder Example (visit = print)



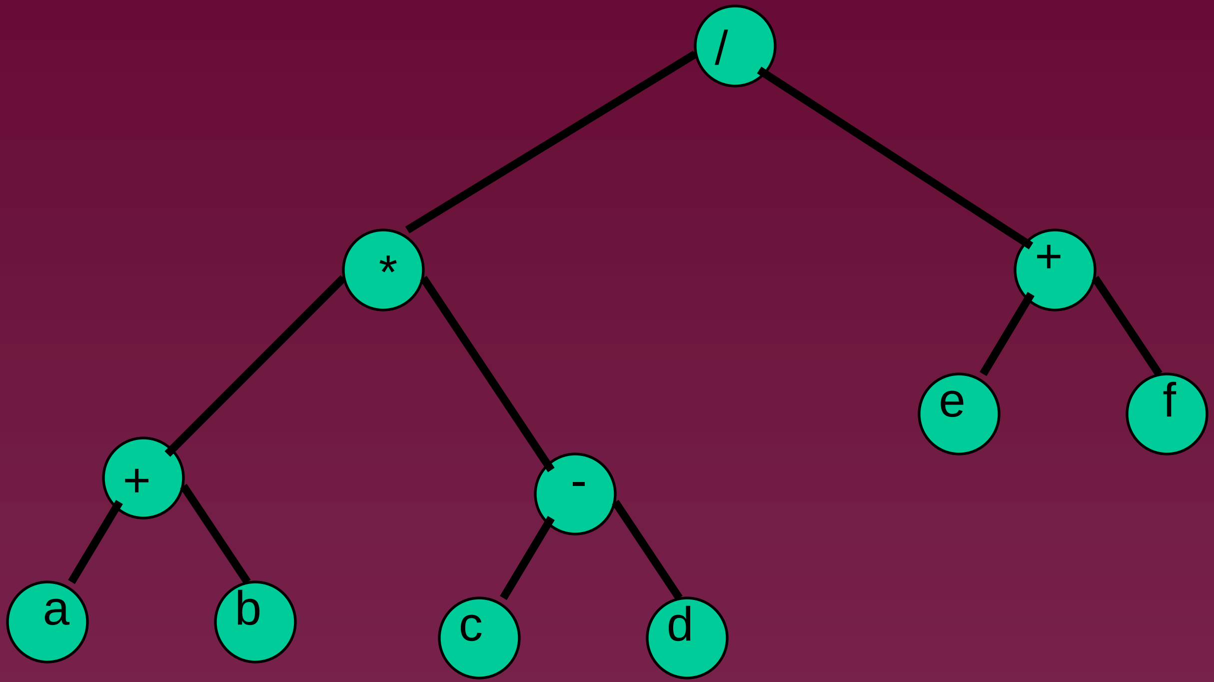
b c a

Postorder Example (visit = print)



g h d i e b j f c a

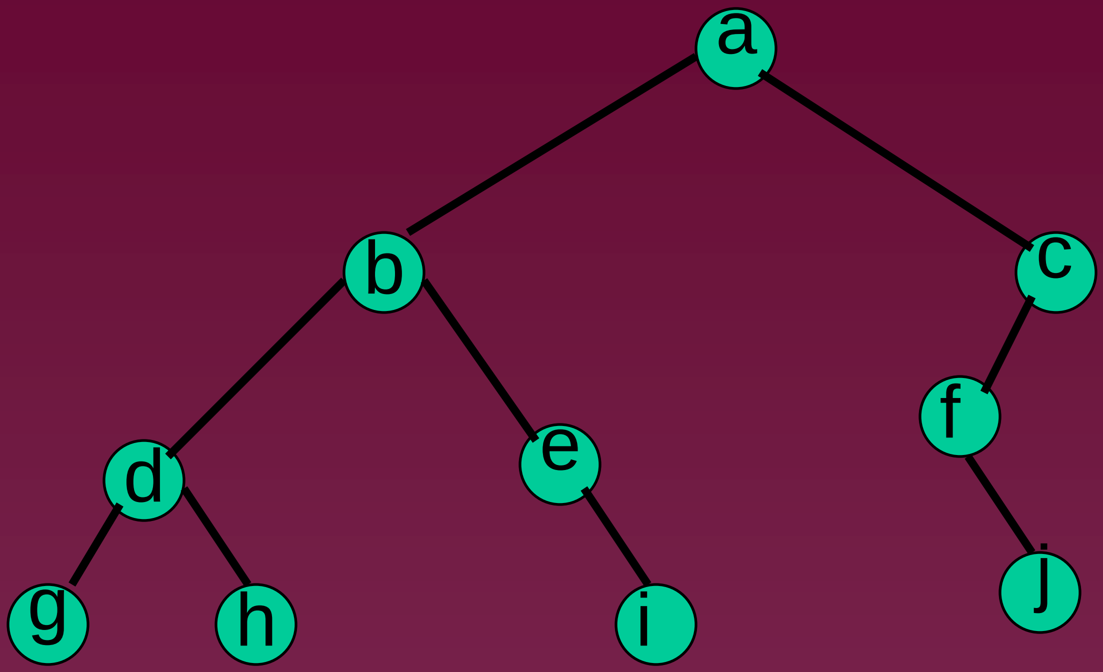
Postorder Of Expression Tree



a b + c d - * e f + /

Gives postfix form of expression!

Traversal Applications



- Make a clone.
- Determine height.
- Determine number of nodes.

Binary Tree Construction

- Suppose that the elements in a binary tree are distinct.
- Can you construct the binary tree from which a given traversal sequence came?
- When a traversal sequence has more than one element, the binary tree is not uniquely defined.
- Therefore, the tree from which the sequence was obtained cannot be reconstructed uniquely.

Some Examples

preorder = ab

inorder = ab

postorder = ab

level order = ab

Binary Tree Construction

- Can you construct the binary tree, given two traversal sequences?
- Depends on which two sequences are given.

Preorder And Postorder

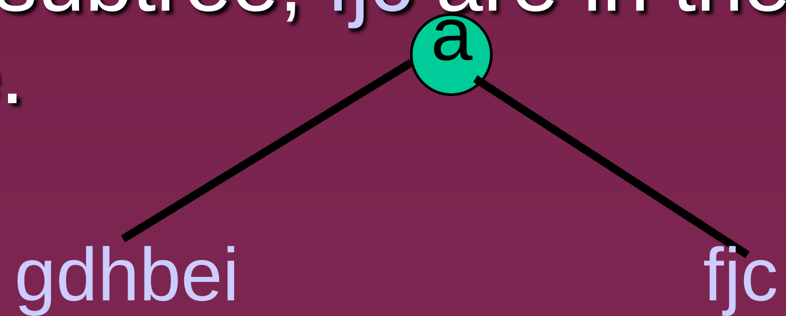
preorder = ab

postorder = ba

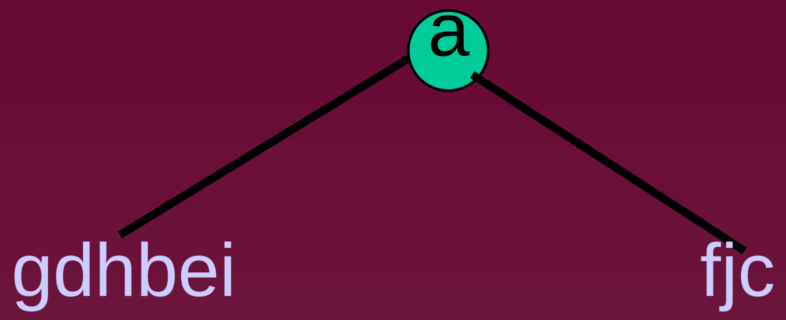
- Preorder and postorder do not uniquely define a binary tree.
- Nor do preorder and level order (same example).
- Nor do postorder and level order (same example).

Inorder And Preorder

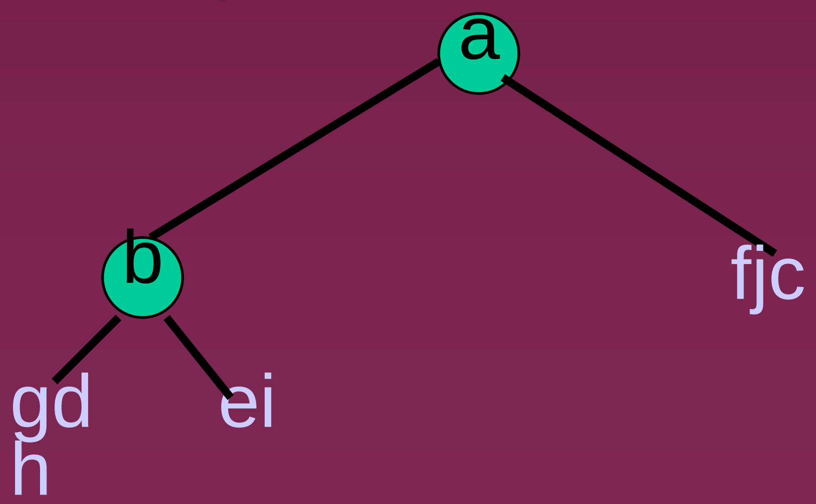
- inorder = g d h b e i a f j c
- preorder = a b d g h e i c f j
- Scan the preorder left to right using the inorder to separate left and right subtrees.
- a is the root of the tree; gdhbei are in the left subtree; fjc are in the right subtree.



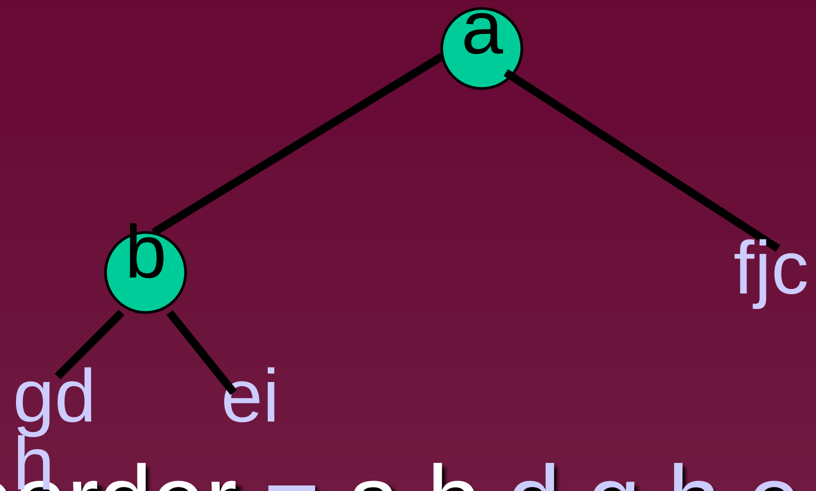
Inorder And Preorder



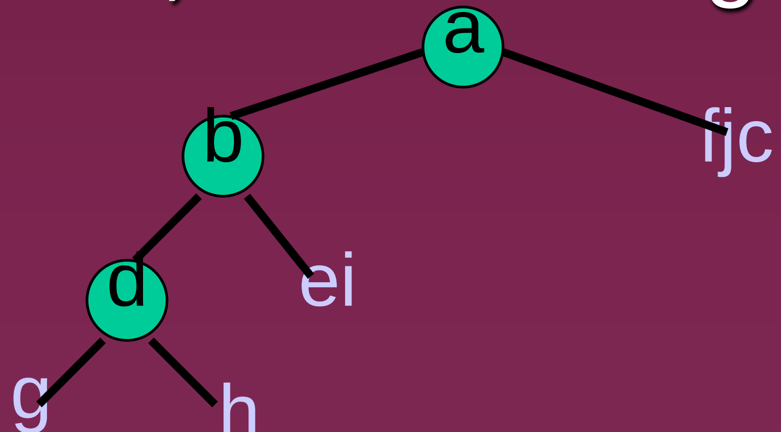
- preorder = a b d g h e i c f j
- b is the next root; gdh are in the left subtree; ei are in the right subtree.



Inorder And Preorder



- preorder = a b d g h e i c f j
- d is the next root; g is in the left subtree; h is in the right subtree.

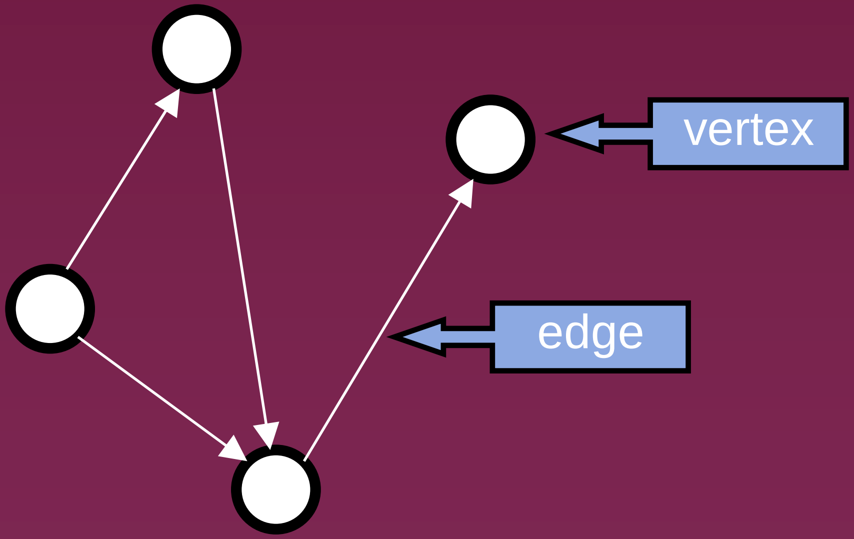


Inorder And Postorder

- Scan postorder from right to left using inorder to separate left and right subtrees.
- inorder = g d h b e i a f j c
- postorder = g h d i e b j f c a
- Tree root is a; gdhbei are in left subtree; fjc are in right subtree.

What is a graph?

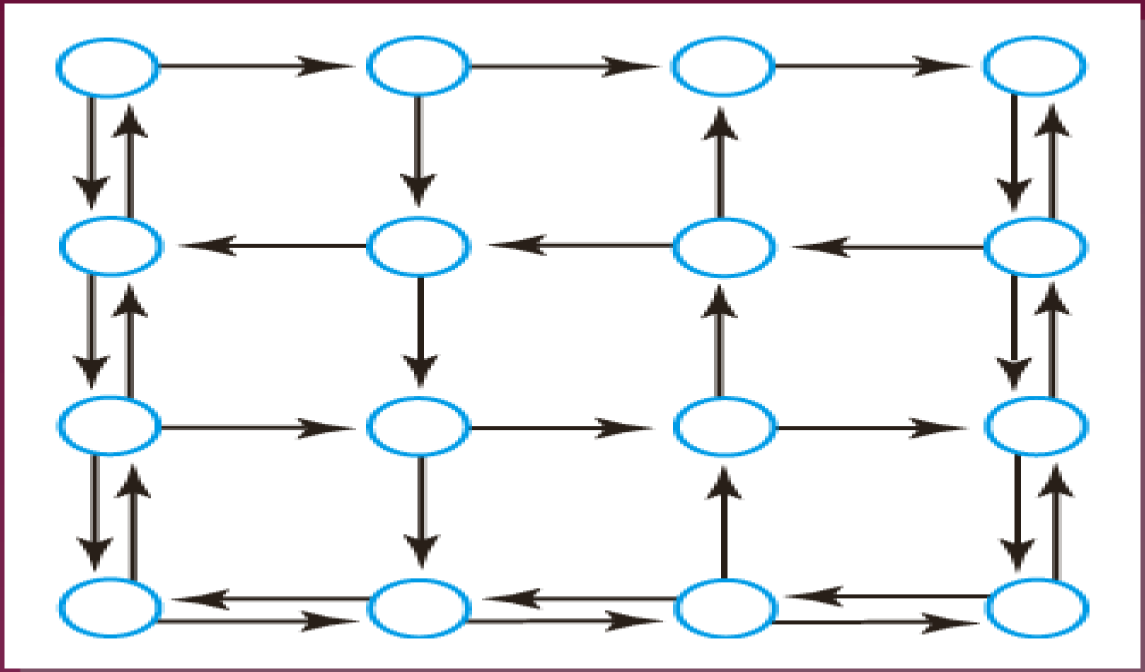
- A set of vertices and edges
  - Directed/Undirected
  - Weighted/Unweighted
  - Cyclic/Acyclic



Some Examples and Terminology

- A graph is a collection of distinct vertices and distinct edges
  - Edges can be directed or undirected
  - When it has directed edges it is called a digraph
- Vertices or nodes are connected by edges
- A subgraph is a portion of a graph that itself is a graph

Example : Street Maps



A directed graph representing a city's street map. Directed edges

Graph Paths

- A sequence of edges that connect two vertices in a graph
- In a directed graph the direction of the edges must be considered
  - Called a directed path
- A **cycle** is a path that begins and ends at same vertex
  - Simple path does not pass through any vertex more than once
- A graph with no cycles is acyclic

Weighted Graph

- A weighted graph has values on its edges
  - Weights or costs
- A path in a weighted graph also has weight or cost
  - The sum of the edge weights
- Examples of weights
  - Miles between nodes on a map
  - Driving time between nodes
  - Taxi cost between node locations

Representation of Graphs

- Adjacency Matrix
  - A V x V array, with matrix[i][j] storing whether there is an edge between the i^{th} vertex and the j^{th} vertex
- Adjacency Linked List
  - One linked list per vertex, each storing directly reachable vertices
- Edge List

Representation of Graphs

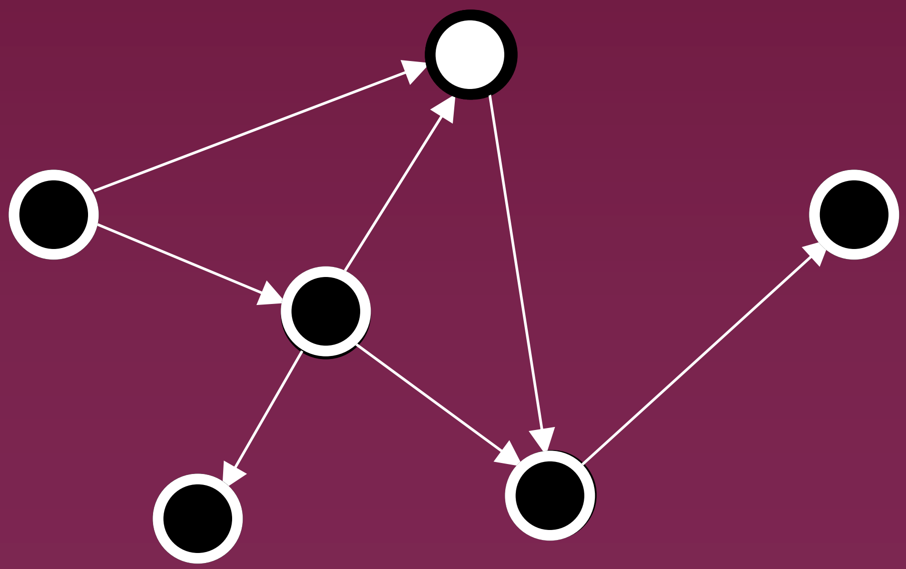
	Adjacency Matrix	**Adjacency Linked List**	**Edge List**
Memory Storage	O(V^2)	O(V+E)	O(V+E)
Check whether (*u*,*v*) is an edge	O(1)	O(deg(u))	O(deg(u))
Find all adjacent vertices of a vertex *u*	O(V)	O(deg(u))	O(deg(u))
deg(u): the number of edges connecting vertex *u*			

Graph Searching

- Why do we do graph searching? What do we search for?
- What information can we find from graph searching?
- How do we search the graph? Do we need to visit all vertices? In what order?

Depth-First Search (DFS)

- Strategy: Go as far as you can (if you have not visit there), otherwise, go back and try another way



DFS Implementation

```
DFS (vertex u) {
    mark u as visited
    for each vertex v directly reachable from u
     if v is unvisited
          DFS (v)
}
```

- Initially all vertices are marked as *unvisited*

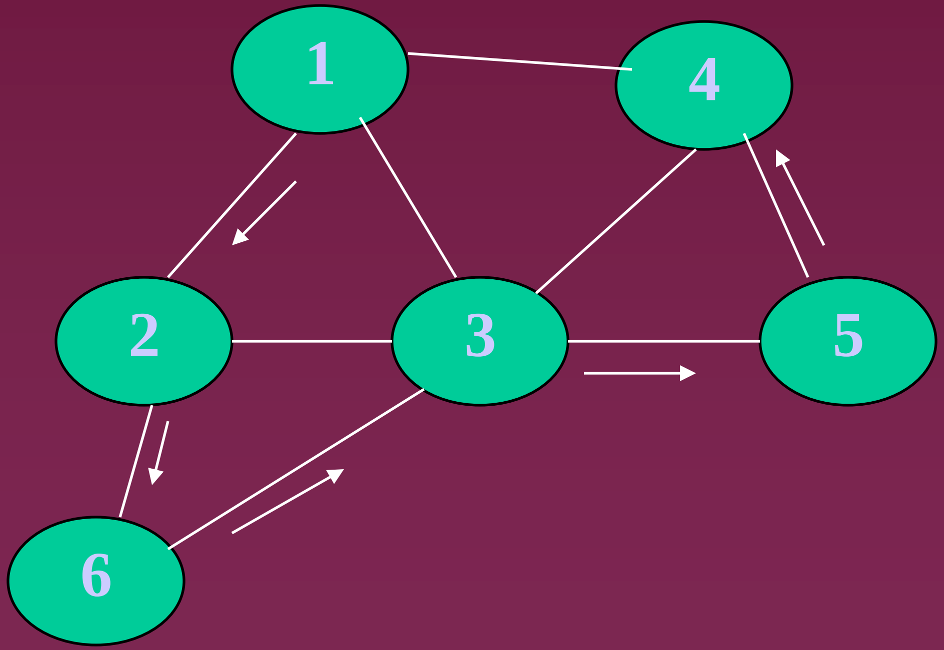
DFS Example-1

Depth first traversal: 1, 2, 6, 3, 5, 4

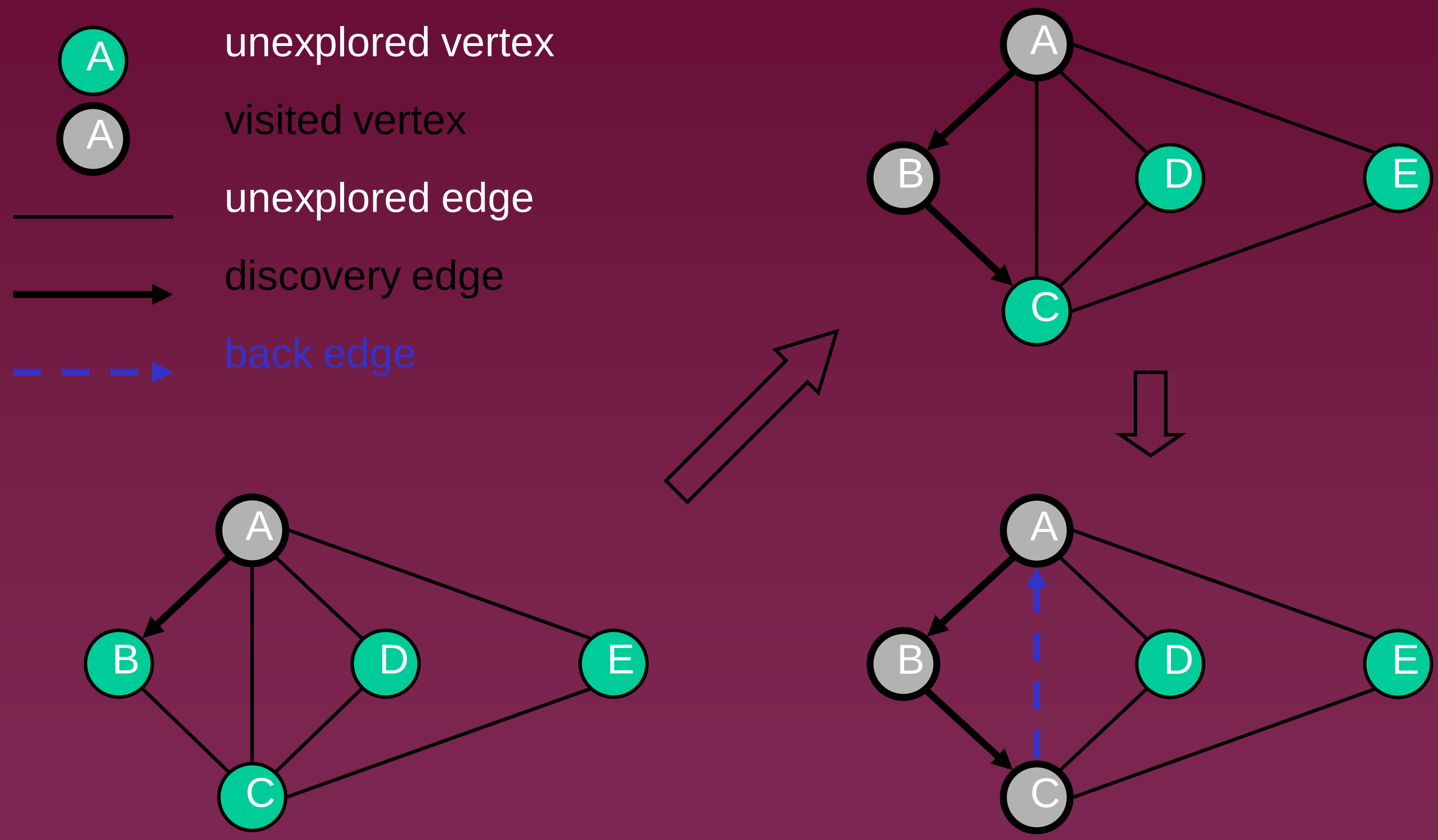
the particular order is dependent on the order of nodes in the adjacency lists

Adjacency lists

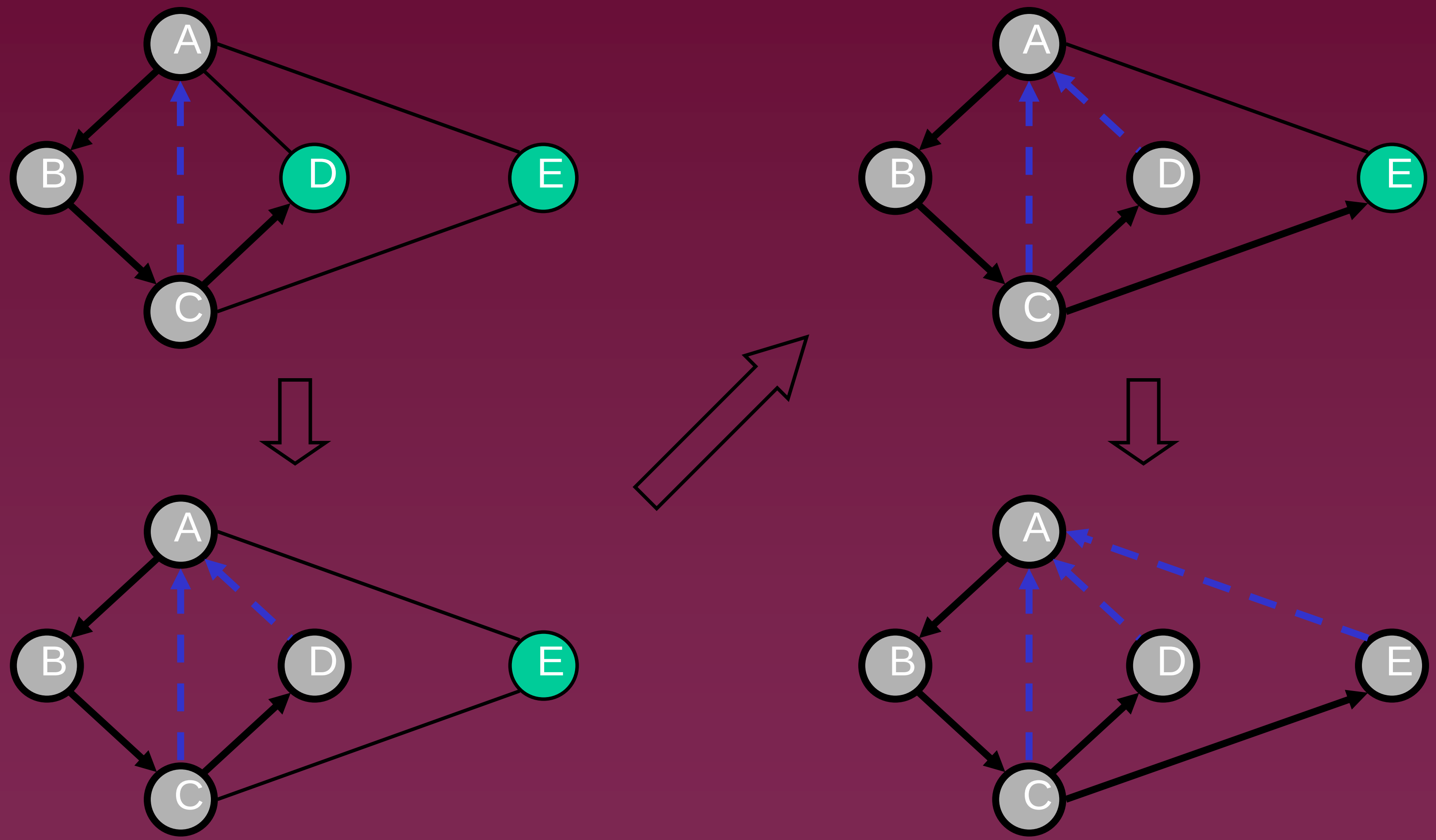
1: 2, 3, 4
2: 6, 3, 1
3: 1, 2, 6, 5, 4
4: 1, 3, 5
5: 3, 4
6: 2, 3



DFS Example-2



Example (cont.)



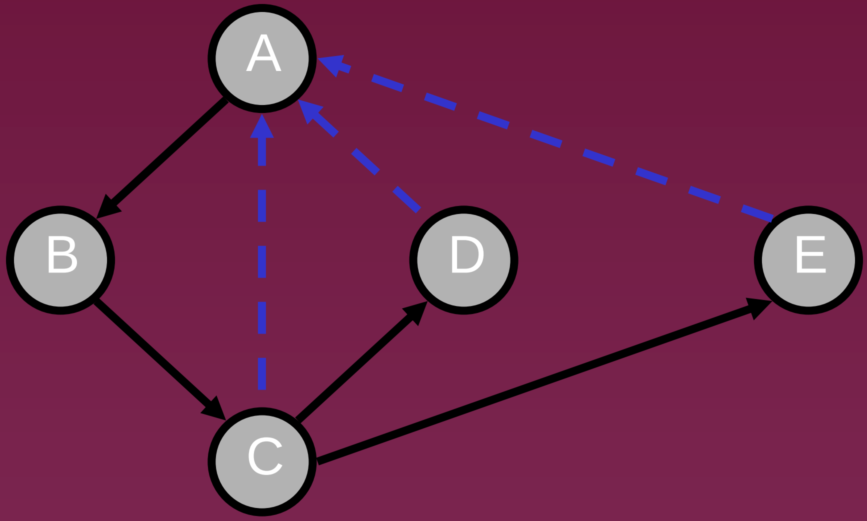
Properties of DFS

Property 1

DFS(*G*, *v*) visits all the vertices and edges in the connected component of *v*

Property 2

The discovery edges labeled by *DFS*(*G*, *v*) form a spanning tree of the connected component of *v*

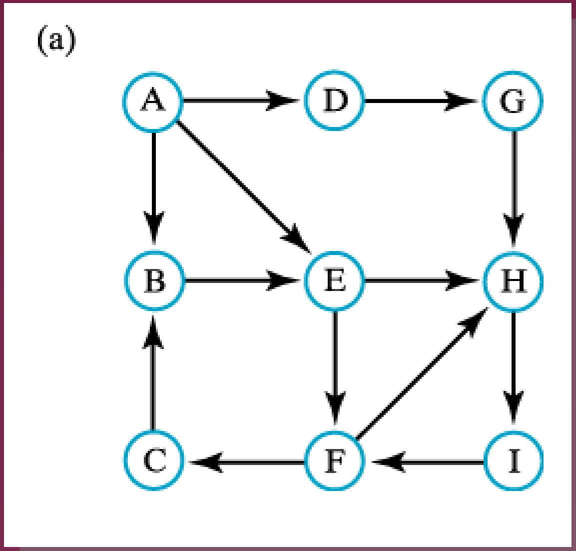


Analysis of DFS

- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or BACK
- Method incidentEdges is called once for each vertex
- DFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\Sigma_v \deg(v) = 2m$

Depth-First Traversal

A trace of a depth first traversal beginning at vertex A of the directed graph



topVertex	nextNeighbor	Visited vertex	vertexStack (top to bottom)	traversalOrder (front to back)
		A	A	A
A			A	
	B	B	BA	AB
B			BA	
	E	E	EBA	ABE
E			EBA	
	F	F	FEBA	ABEF
F			FEBA	
	C	C	CFEBA	ABEFC
C			FEBA	
F			FEBA	
	H	H	HFEBA	ABEFCH
H			HFEBA	
	I	I	IHFEBA	ABEFCHI
I			HFEBA	
H			FEBA	
F			EBA	
E			BA	
B			A	
A			A	
	D	D	DA	ABEFCHID
D			DA	
	G		GDA	ABEFCHIDG
G			DA	
D			A	
A			*empty*	ABEFCHIDG

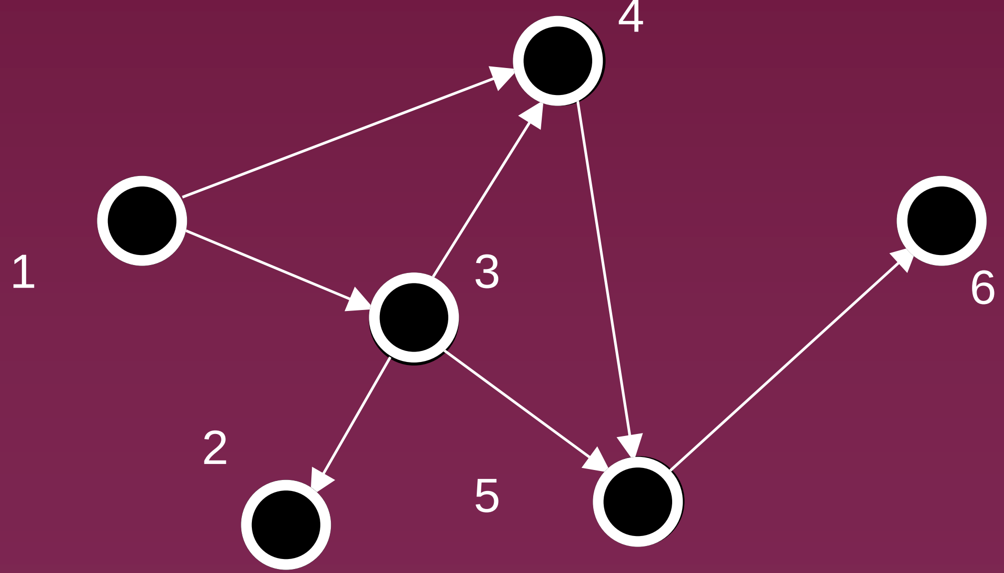
Breadth-First Search (BFS)

- Instead of going as far as possible, BFS tries to search all paths.
- BFS makes use of a queue to store visited (but not dead) vertices, expanding the path from the earliest visited vertices.

Simulation of BFS

- Queue:

1	4	3	5	2	6



Implementation

```
while queue Q not empty
    dequeue the first vertex u from Q
    for each vertex v directly reachable from u
      if v is unvisited
           enqueue v to Q
           mark v as visited
```

- Initially all vertices except the start vertex are marked as *unvisited* and the queue contains the start vertex only

Example-1

Breadth-first traversal: 1, 2, 3, 4, 6, 5

1: starting node

2, 3, 4 : adjacent to 1

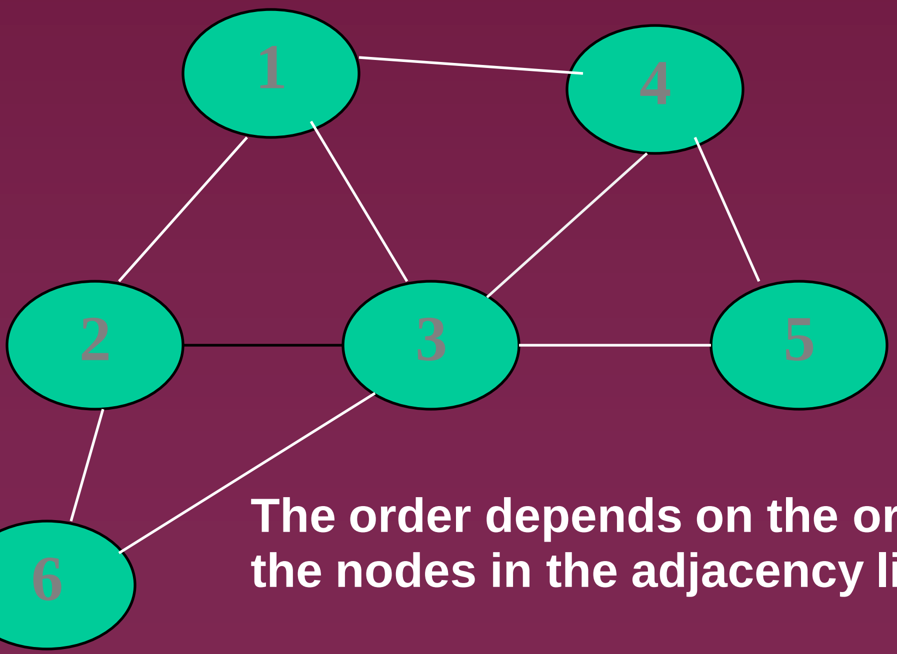
(at distance 1 from node 1)

6 : unvisited adjacent to node 2.

5 : unvisited, adjacent to node 3

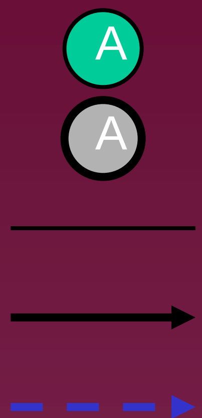
Adjacency lists

1: 2, 3, 4
2: 1, 3, 6
3: 1, 2, 4, 5, 6
4: 1, 3, 5
5: 3, 4
6: 2, 3



The order depends on the order of the nodes in the adjacency lists

Example-2 BFS



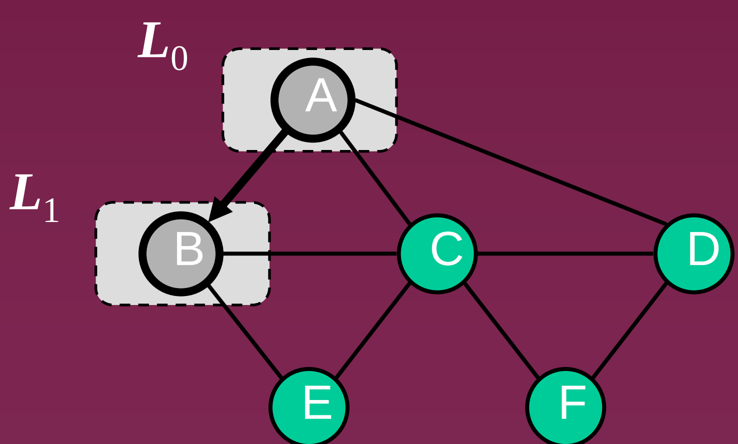
unexplored vertex

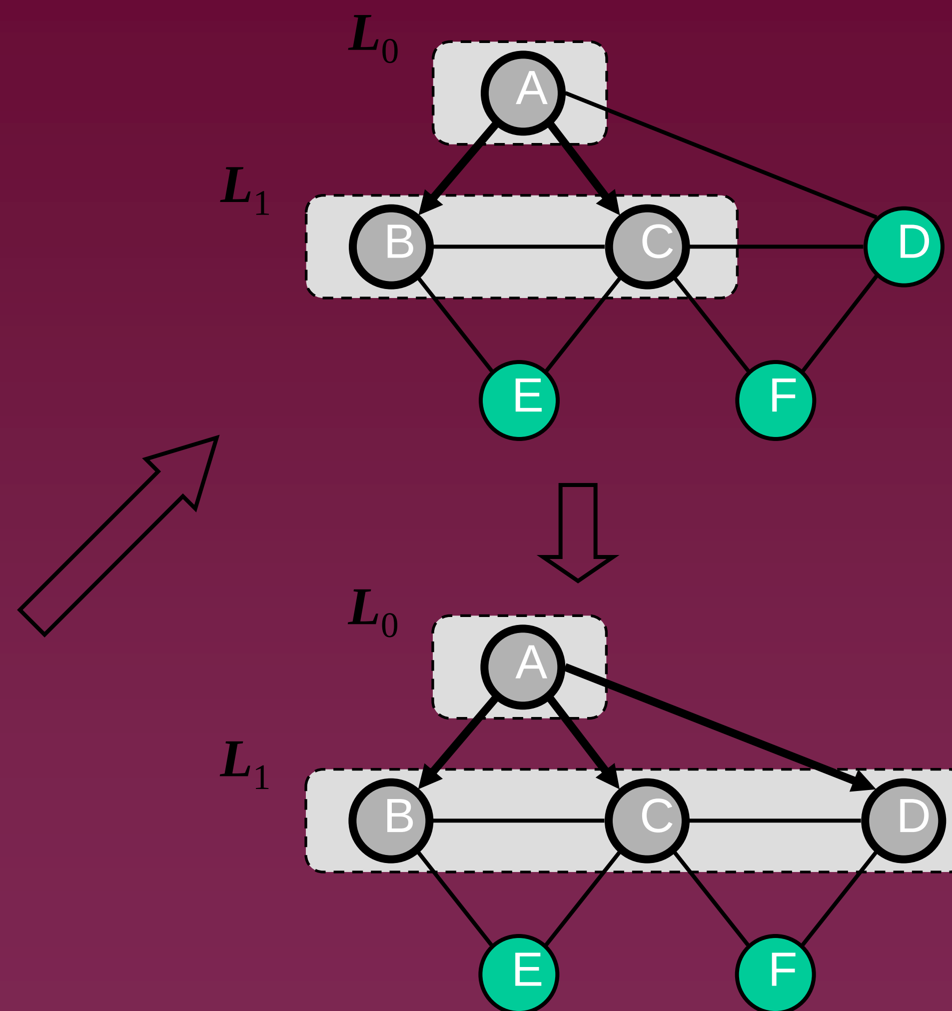
visited vertex

unexplored edge

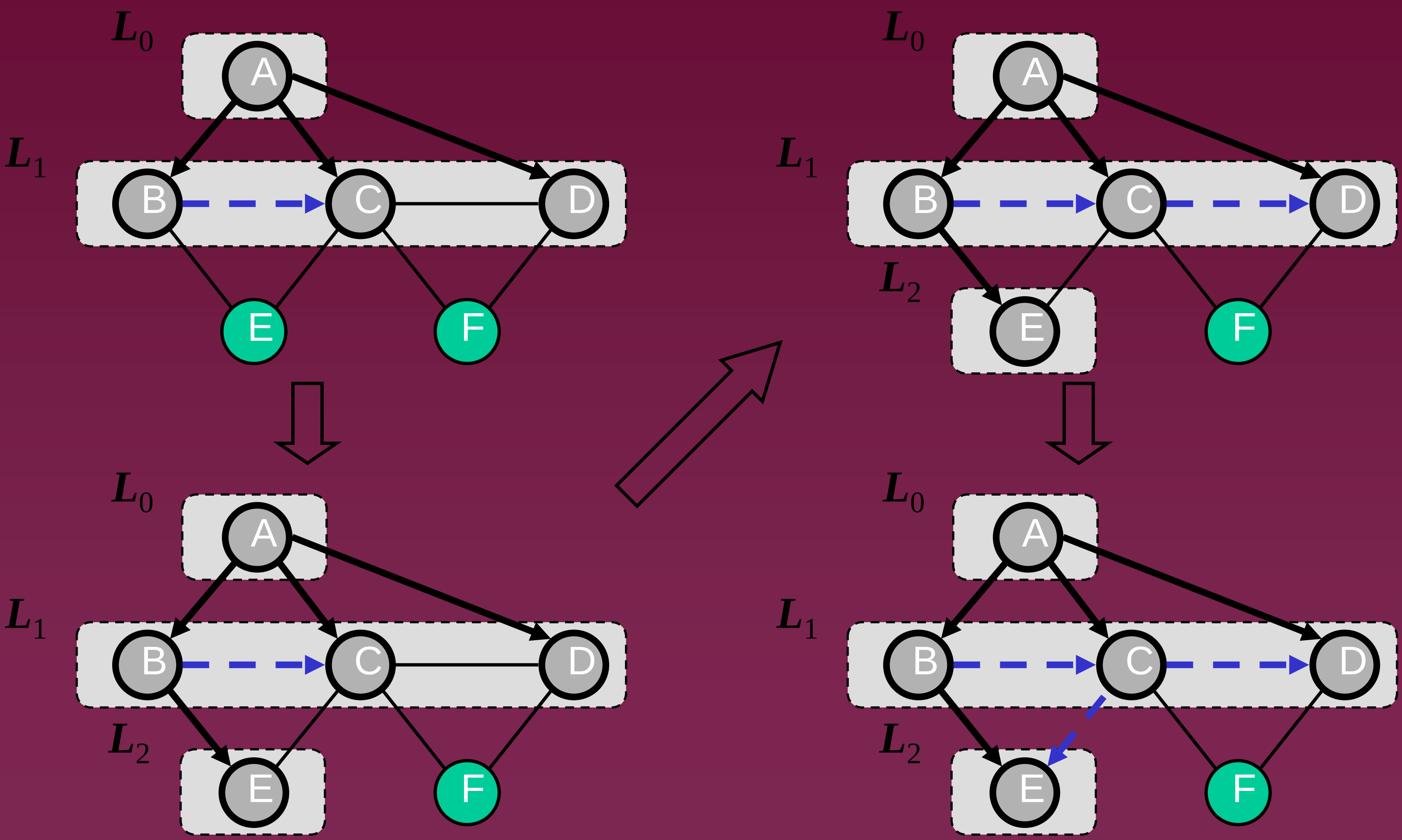
discovery edge

cross edge

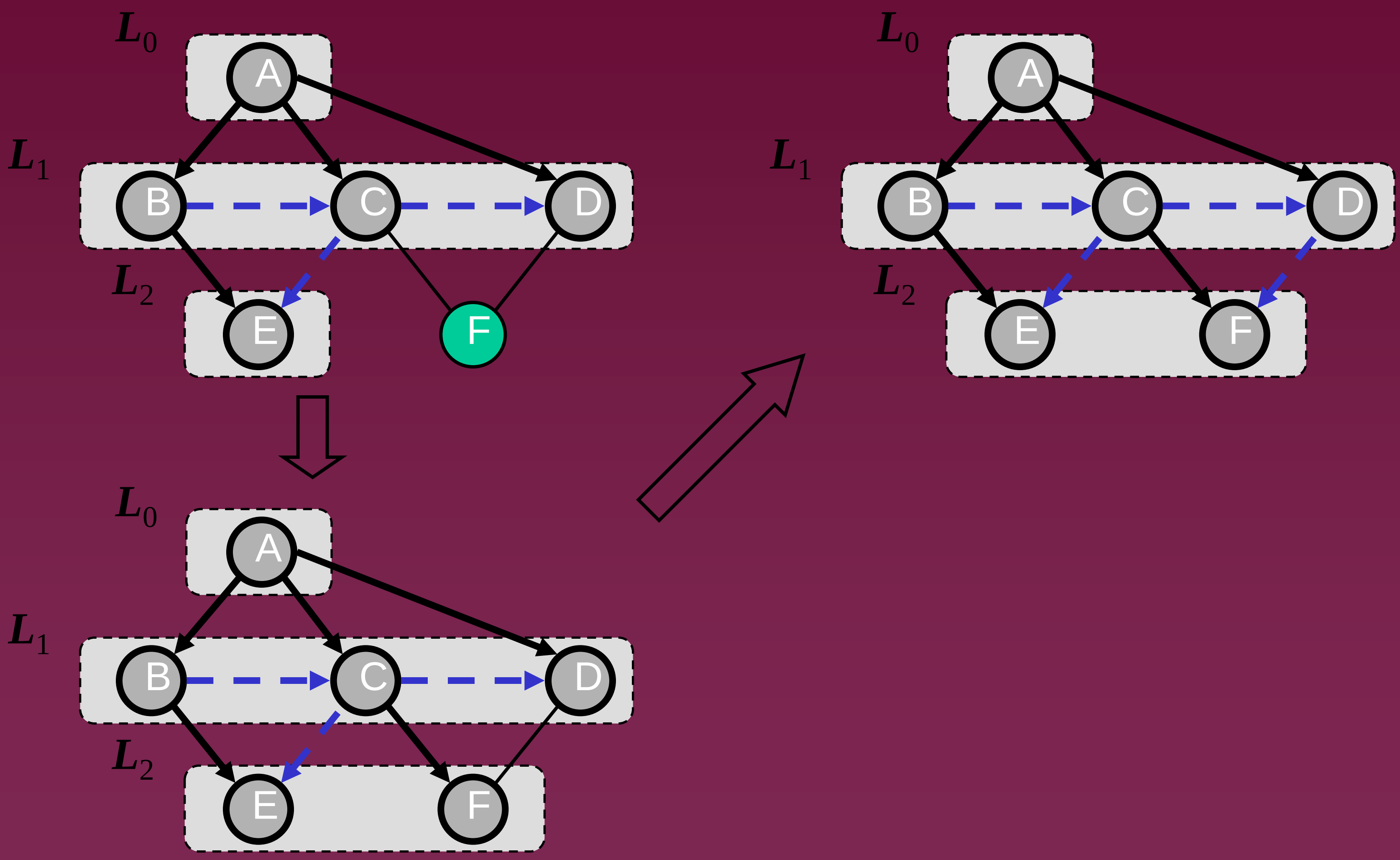




Example (cont.)



Example (cont.)



Properties

Notation

G_s: connected component of s

Property 1

$BFS(G, s)$ visits all the vertices and edges of G_s

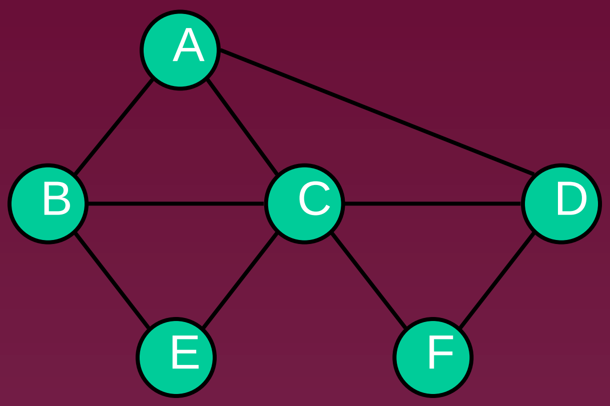
Property 2

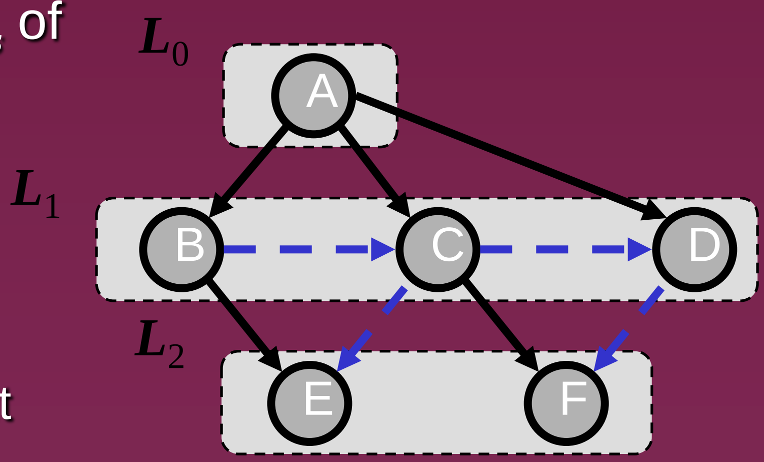
The discovery edges labeled by $BFS(G, s)$ form a spanning tree T_s of G_s

Property 3

For each vertex v in L_i

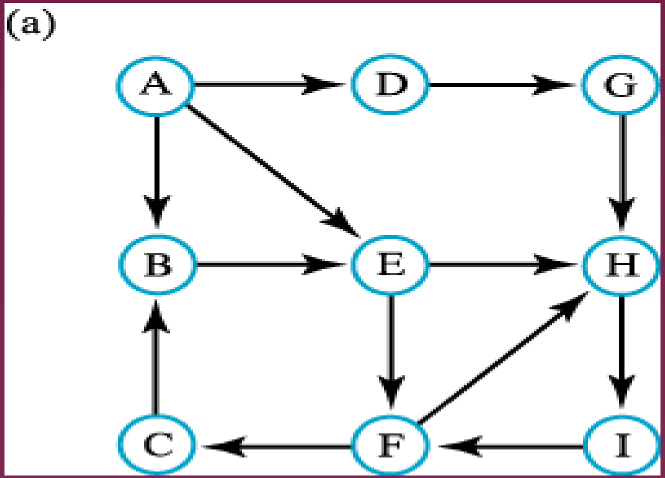
- The path of T_s from s to v has i edges
- Every path from s to v in G_s has at least i edges





Breadth-First Traversal

A trace of a breadth-first traversal for a directed graph, beginning at vertex A.



(b)

frontVertex	nextNeighbor	Visited vertex	vertexQueue	traversalOrder
		A	A	A
A			*empty*	
	B	B	B	A B
	D	D	B D	A B D
	E	E	B D E	A B D E
B			D E	
D			E	
	G	G	E G	A B D E G
E			G	
	F	F	G F	A B D E G F
	H	H	G F H	A B D E G F H
G			F H	
F			H	
	C	C	H C	A B D E G F H C
H			C	
	I	I	C I	A B D E G F H C I
C			I	
I			*empty*	

BFS – Complexity

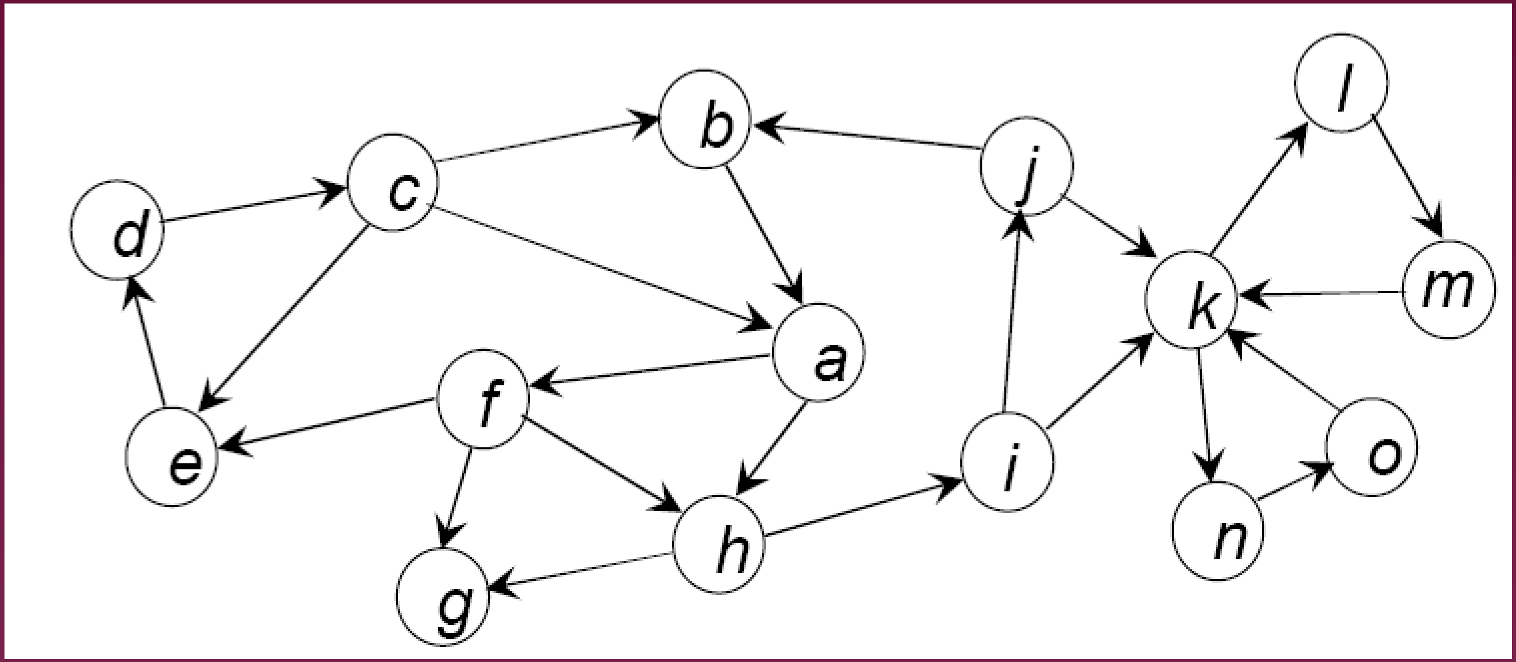
Step 1 : read a node from the queue **O(V)** times.

Step 2 : examine all neighbors, i.e. we examine all edges of the currently read node.

Not oriented graph: **2*E** edges to examine

Hence the complexity of BFS is O(V + 2*E)

Graph -Traversal Exercise-1



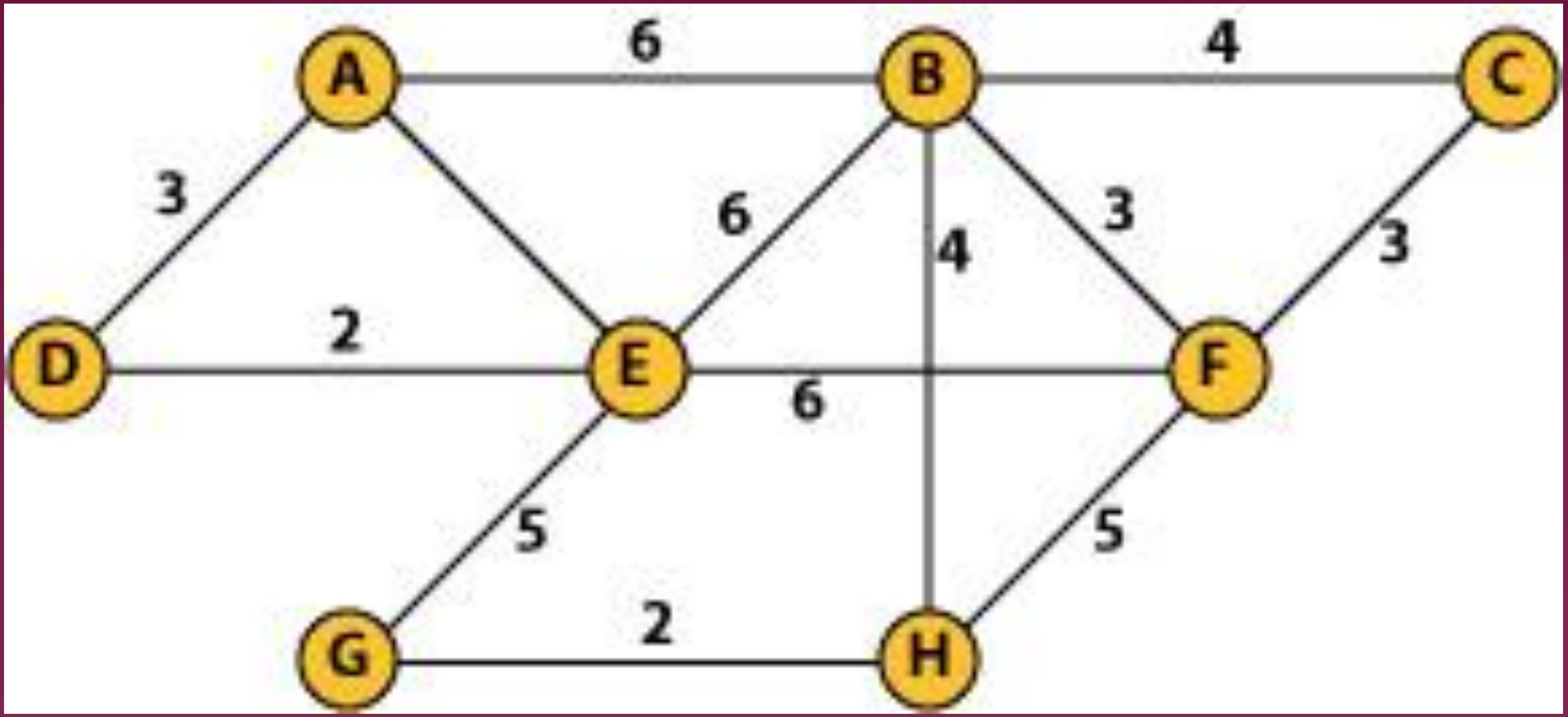
Breadth-First and Depth-First Traversal starting from a

Some of the possible Answers

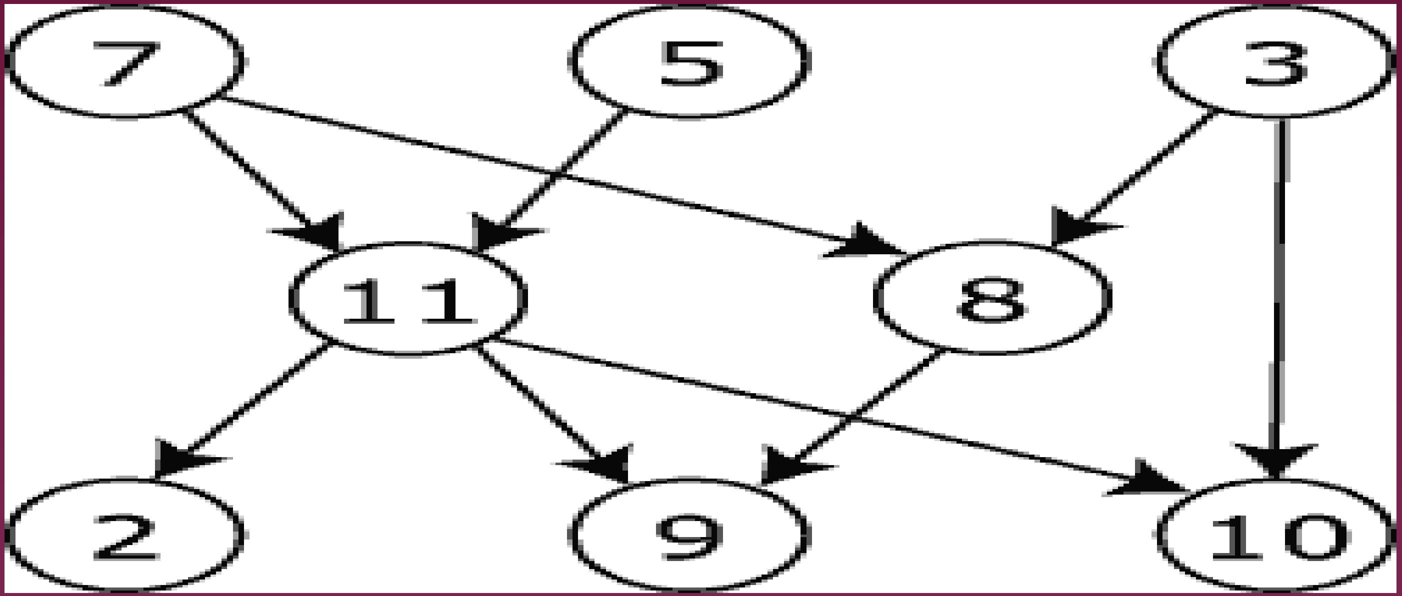
- Breadth-first
  – a f h e g i d j k c l n b m o
- Depth-first
  – a f e d c b g h i j k l m n o

Exercise-2

Write BFS,DFS paths

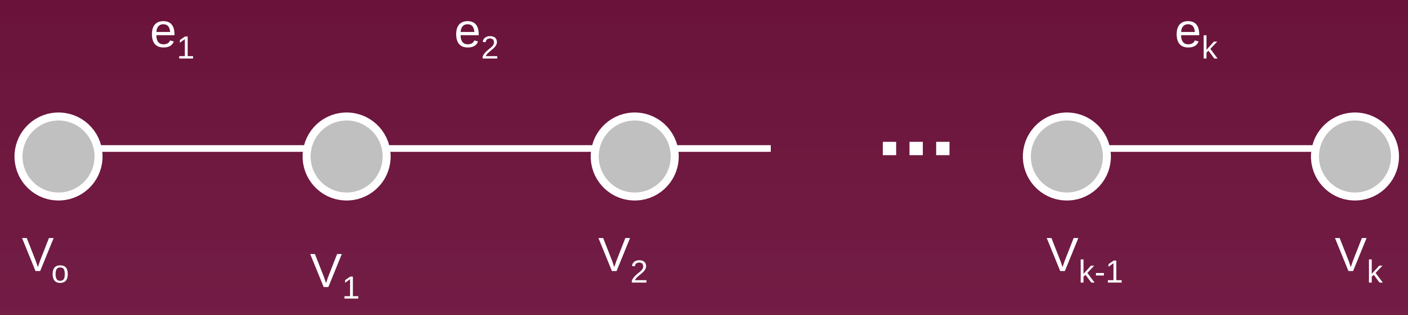


Exercise-3



Connected Components and Spanning Trees: Paths in Graphs

- *Path p*

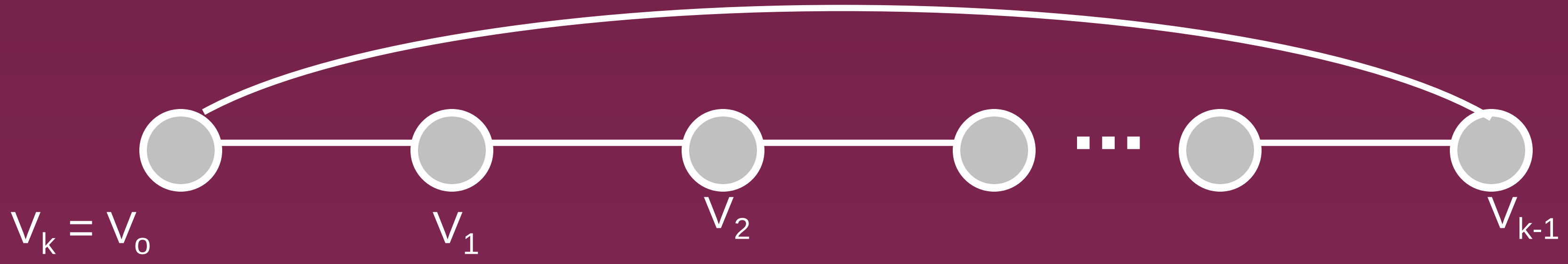


P is a sequence of vertices $v_0, v_1, \ldots, v_k$ where for i=1,…k, v_{i-1} is adjacent to v_i

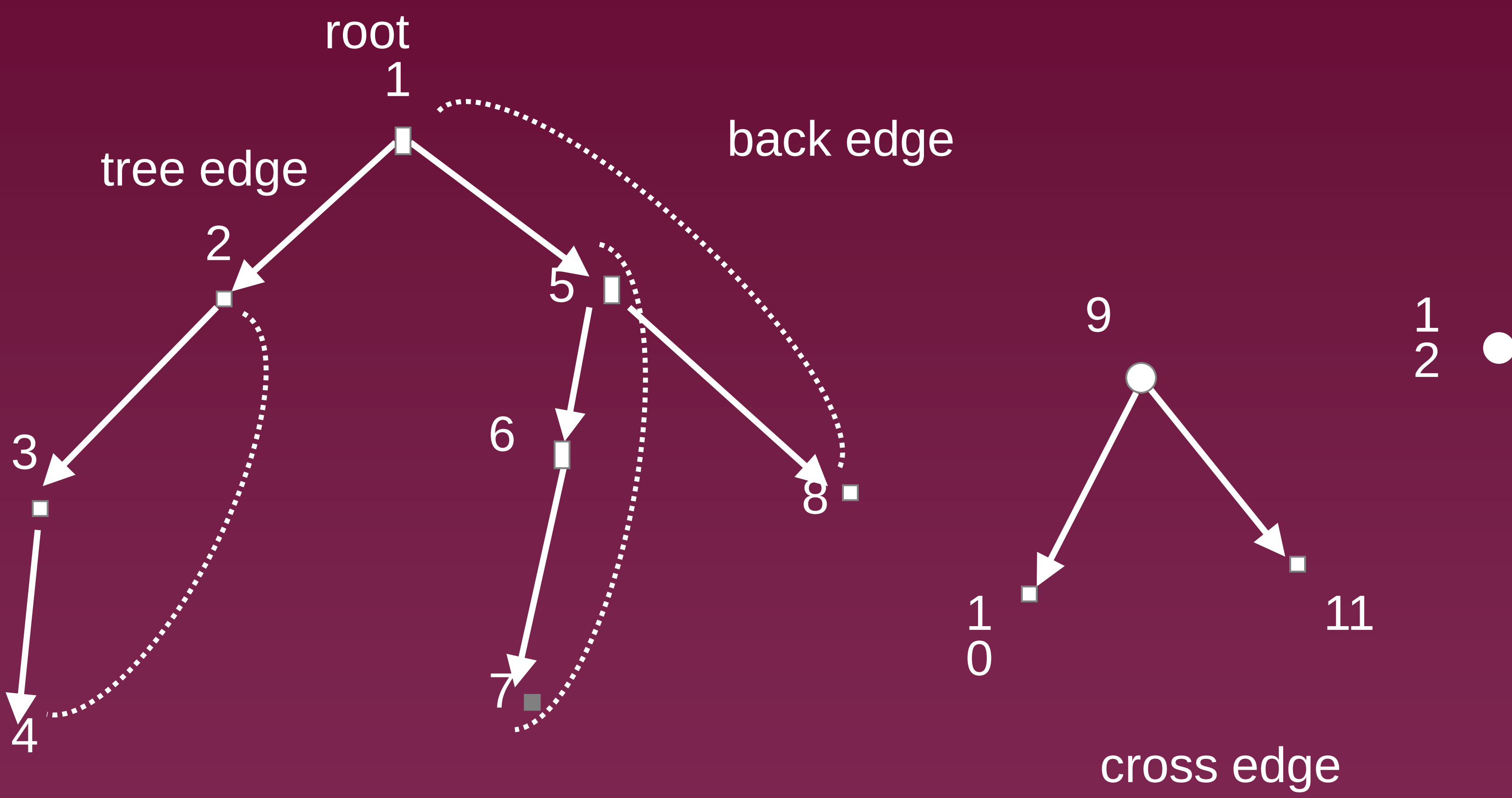
Equivalently, p is a sequence of edges $e_1, \ldots, e_k$ where for i = 2,…k edges e_{i-1}, e_i share a vertex

Simple Paths and Cycles

- *Simple path*
  no edge or vertex repeated,
  except possibly $v_o = v_k$
- *Cycle*
  a path p with $v_o = v_k$ where k>1



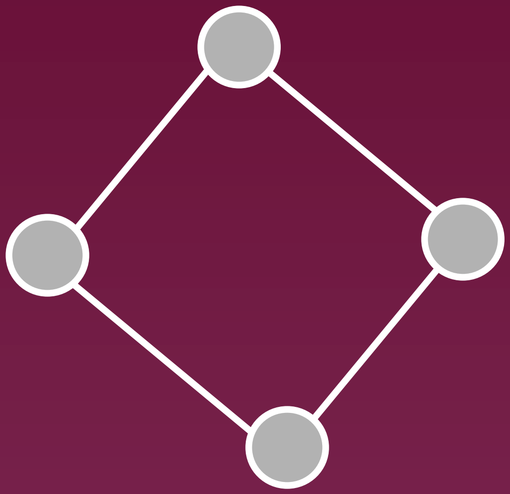
Example Spanning Tree of a Graph



Classification of Edges of G with Spanning Tree T

- An edge (u,v) of T is *tree edge*
- An edge (u,v) of G-T is *back edge* if u is a descendent or ancestor of v.
- Else (u,v) is a *cross edge*

Biconnected Undirected Graphs



(or G is single edge)

G is *biconnected* if $\exists$ two disjoint paths between each pair of vertices

Bi-connected components & DFS

- Biconnected component has 2 components:

  1)A biconnected component of a undirected graph is a maximal biconnected subgraph, that is, a bi-nconnected subgraph not contained in any larger bi-nconnected subgraph.

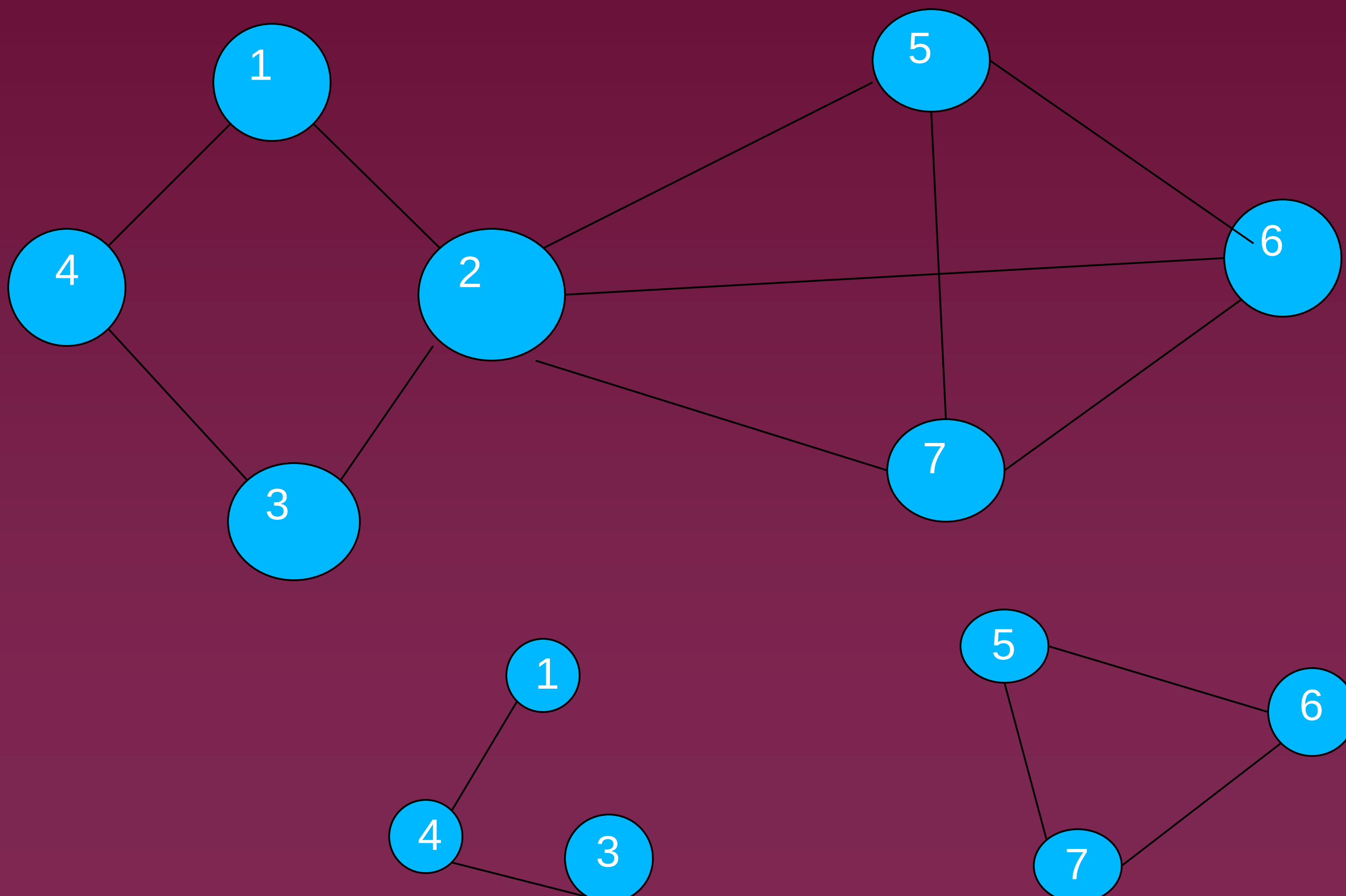
  2)Articulation point:

  Let G=(V,E) be a connected undirected graph then an articulation point of graph ‘G’ is a vertex whose removal disconnects the graph ‘G’.

Bi-connected components & DFS

Articulation Point:

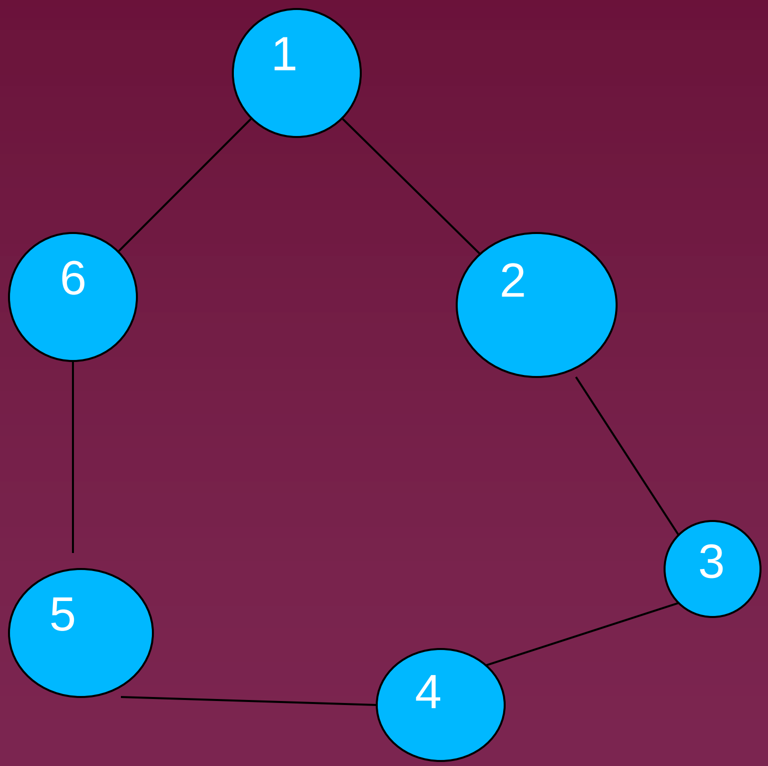
Here 2 is the articulation point after deleting vertex 2 then graph is divided into 2 components.



Bi-connected components & DFS

Bi-Connected Graph:

A graph ‘G’ is said to be Bi-connected if it contains no articulation point.

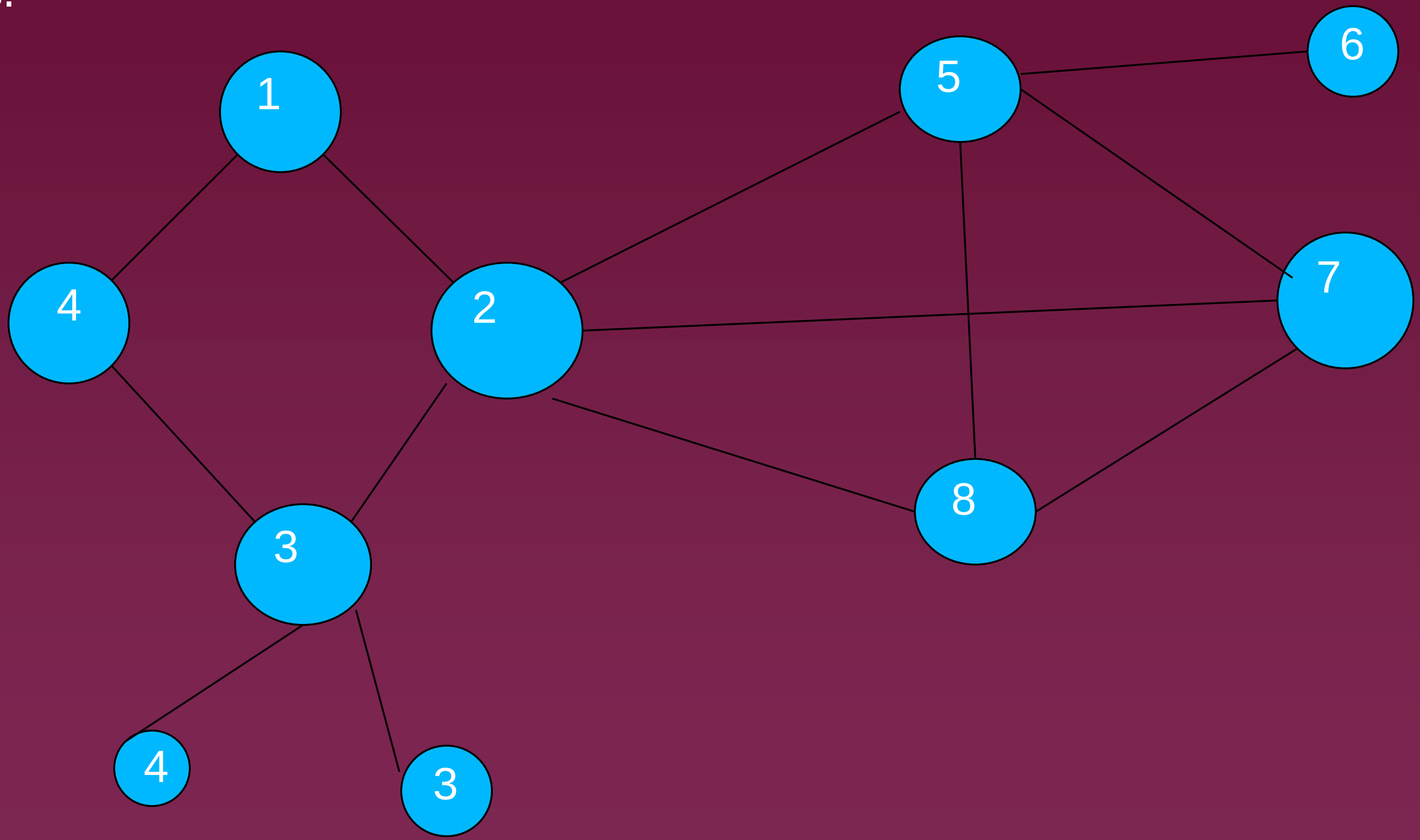


If we deleting the vertex ‘6’ then the graph won’t divide in to 2 components.
If there exists any articulation point , it is an undesirable feature in communication network where joint point between two networks failure in case of joint node fails.

Bi-connected components & DFS

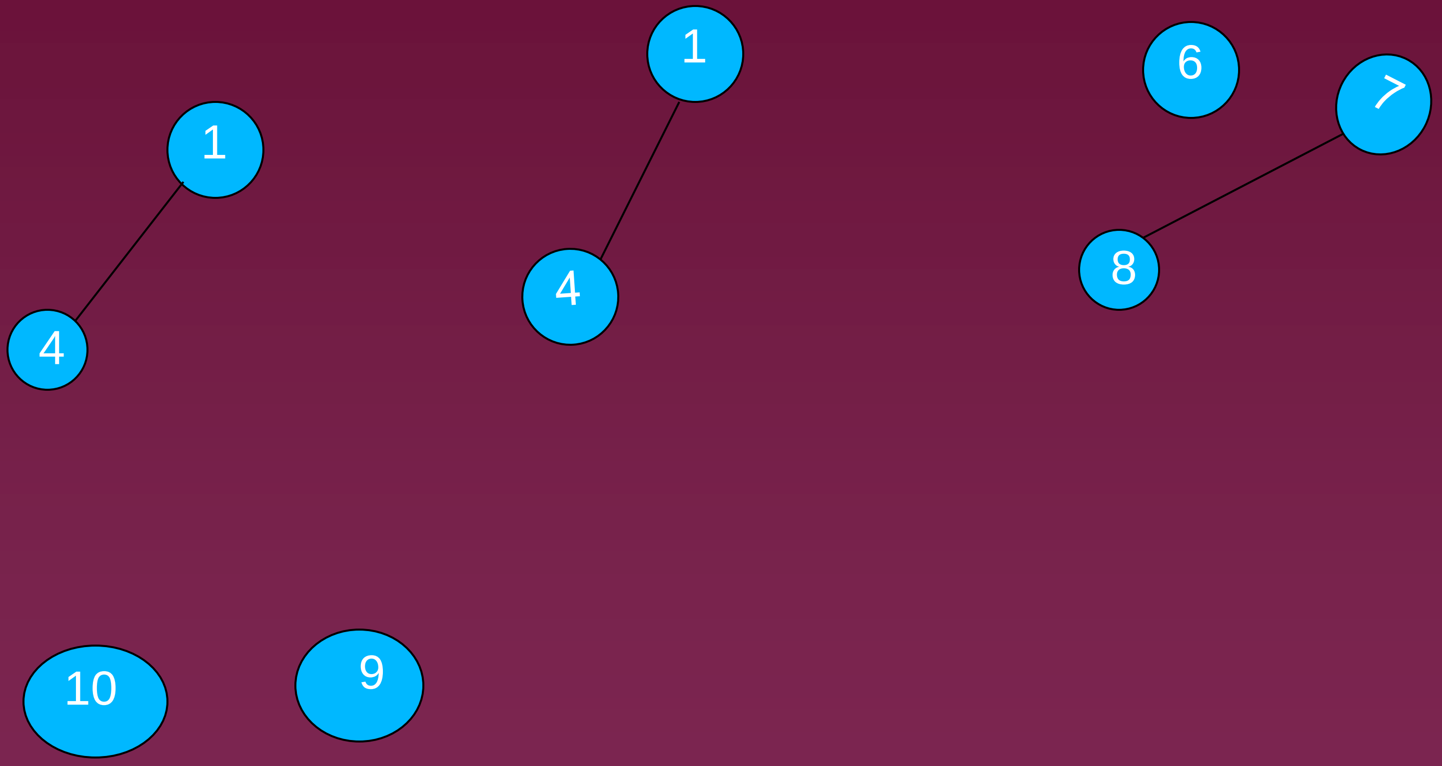
Articulation Point:

Here 2 is the articulation point after deleting vertex 2 then graph is divided into 2 components.



In the above the articulation points are: 2,3 and 5

Bi-connected components & DFS



Bi-connected components & DFS

Identification of Bi-Connected components :

Definition: A Bi-Connected graph G=(V,E) be a connected graph which has no articulation points. A Bi-Connected component of graph ‘G’ is maximal Bi-connected sub graphs.

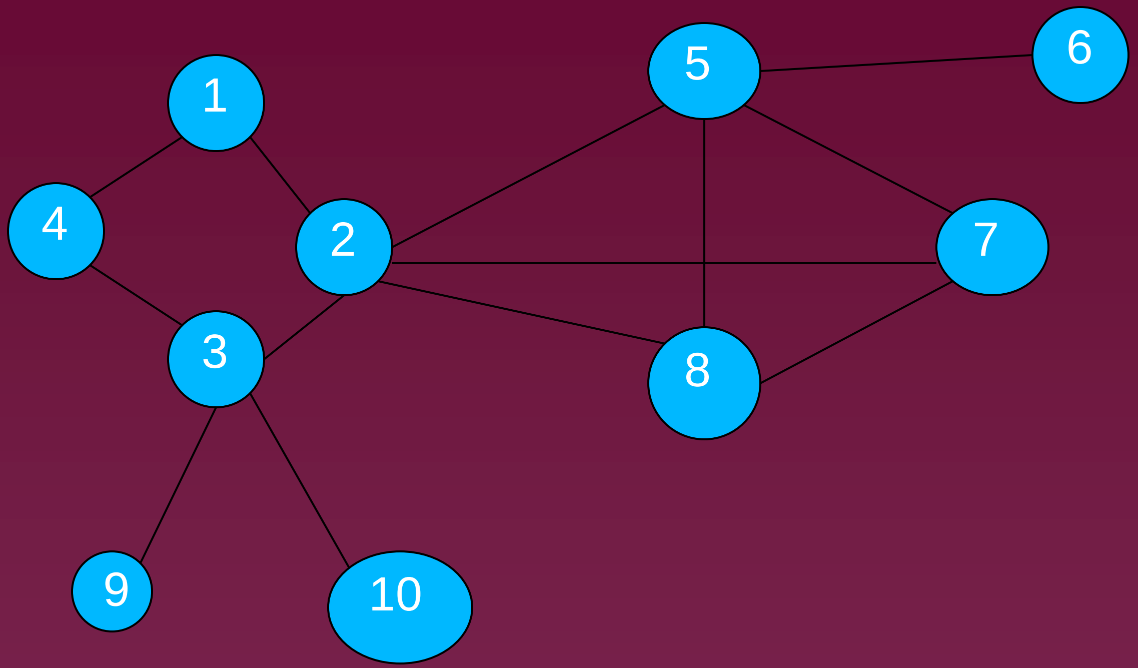
To construct Bi-connected components using 3 rules:

1) Two different Bi-components should not have any common edge.

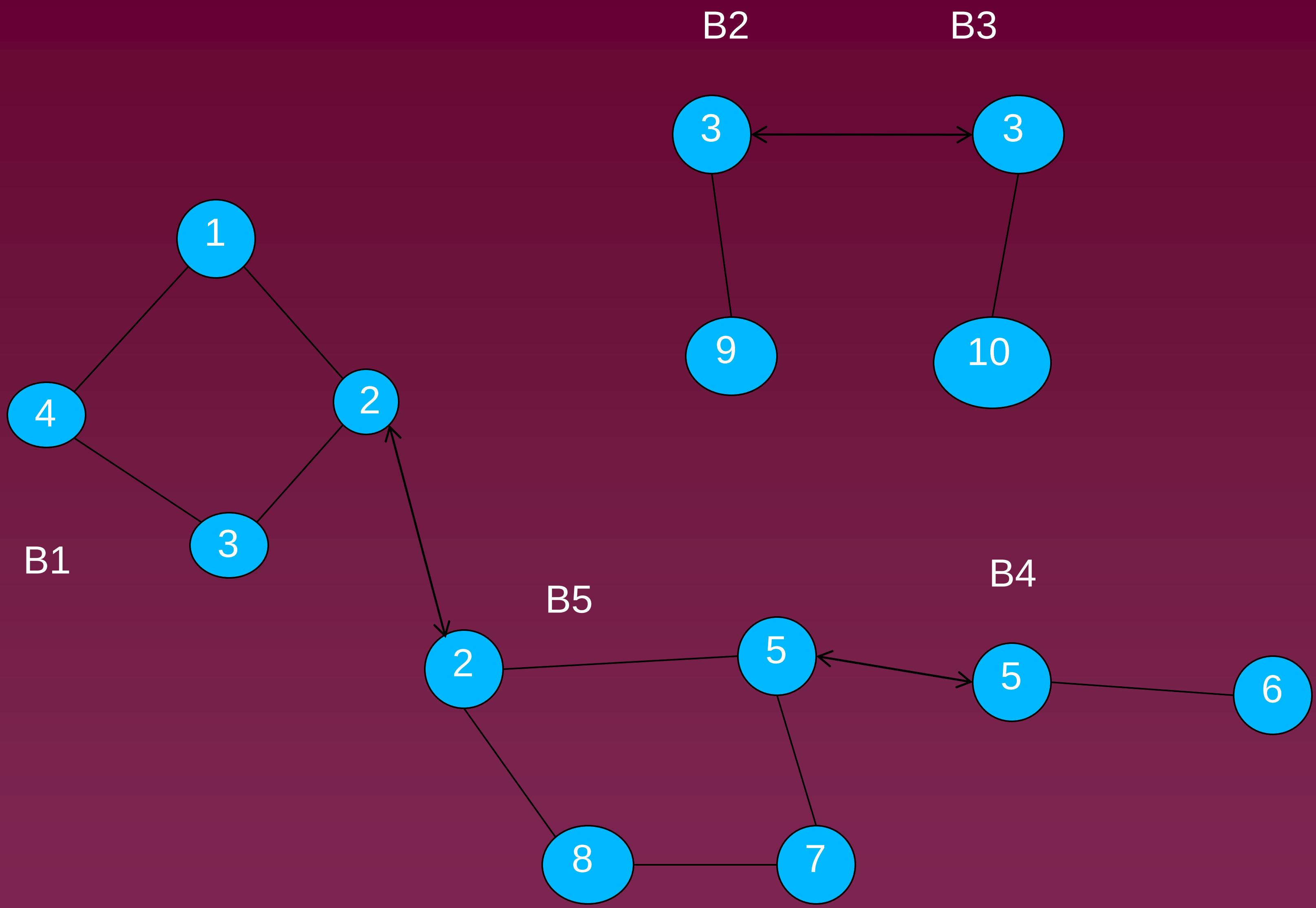
2) Two different Bi-connected components can have a common vertex.

3)The common vertex which is attaching 2 Bi-connected components must be an articulation point of ‘G’.

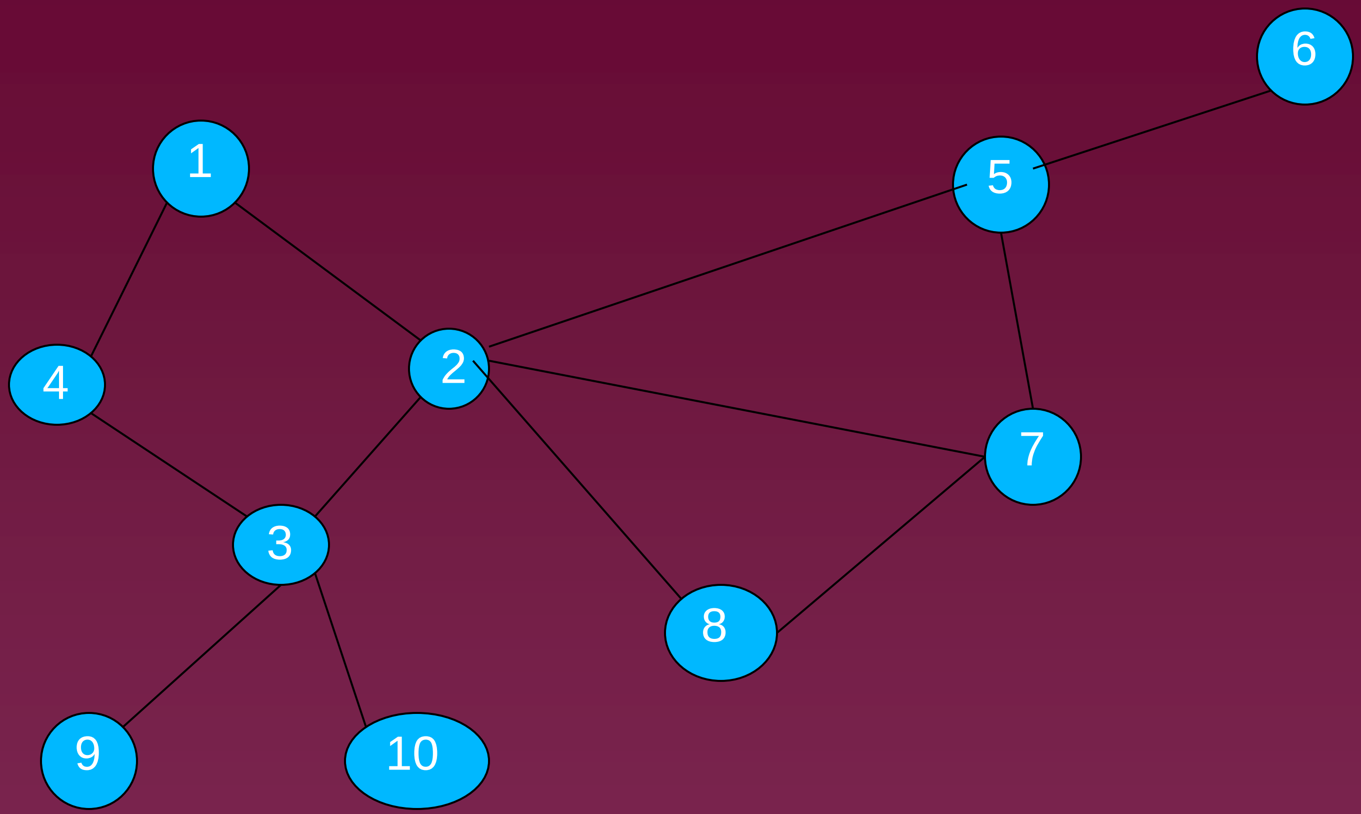
Bi-connected components & DFS



Bi-connected components & DFS

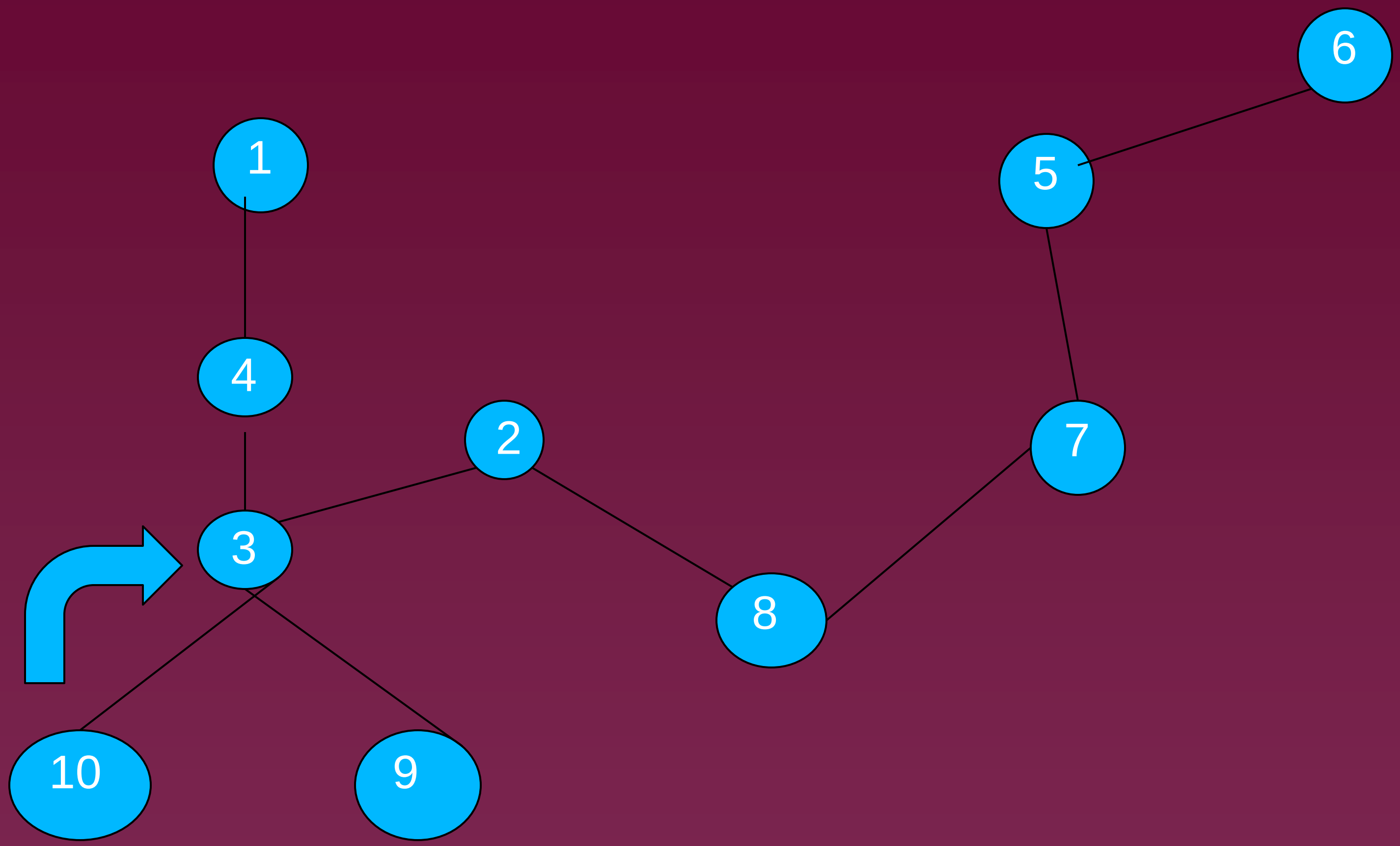


Draw Bi-connected Graph for this graph

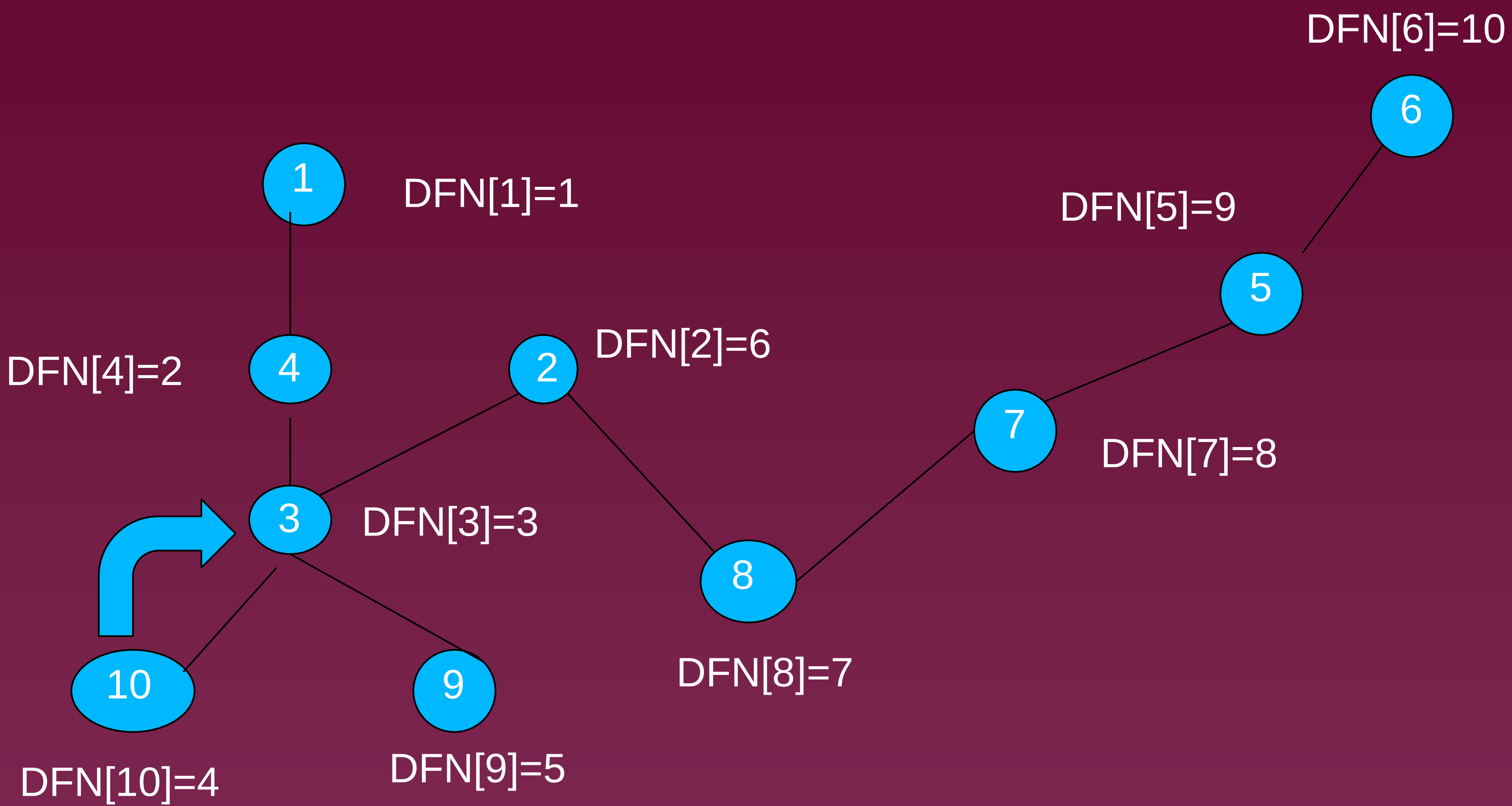


DFS spanning tree for the above directed graph in the next slide

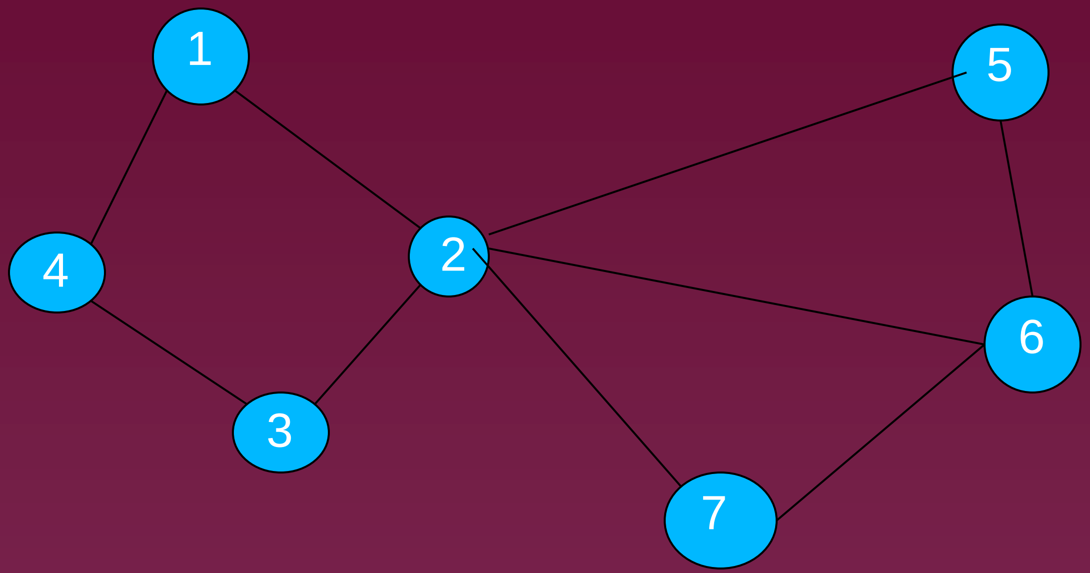
Depth First Search –Spanning Tree -example



DFS–Spanning Tree –traversing Number



Exercise-find DFS spanning tree and traversing number



Algorithm for constructing Bi-connected Graph

1. For each articulation point ‘a’ do

2. Let B1,B2,B3,……………….Bk are the Bi-connected components

3. Containing the articulation point ‘a’

4.Let V_i E B_i, V_i # a i<=i<=k

5. Add(V_i,V_{i+1}) to Graph G.

Vi-vertex belong Bi

Bi-Bi-connected component

i-vertex number 1 to k

a- articulation point

Bi-connected components

- Some vertices are in more than one component (which vertices are these?)