Data Structures and Algorithms Design

DSECLZG519

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Parthasarathy

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Contact Session #6

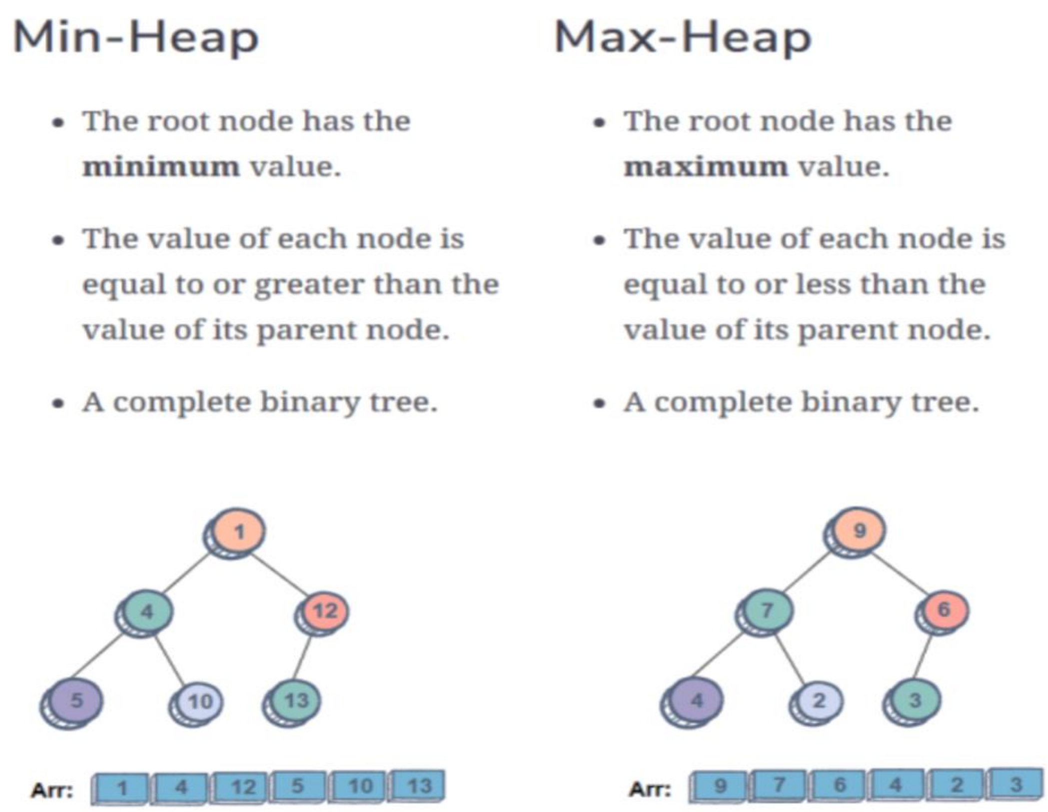
DSECLZG519 – Introduction to Graphs

Agenda for CS #6

1) Recap of CS#5
2) Priority Queues & Heaps
3) Introduction to Graphs
    - What is a Graph ?
    - Types of Graph
    - Terminologies
    - Applications
4) Graph Implementation
    - Adjacency matrix
    - Adjacency List
    - Edge List
5) Exercises

Recap of #5

- Heaps & Types
- Heapification
  - Max-Heapify()
  - Min- Heapify()
- Building Heap
  - Using *bottom up* using appropriate heapification
  - Using *top down* using repetitive insertion
- Insertion
  - Up Heap bubbling
- Deletion
  - Down Heap bubbling
- Application 1: Heap Sort



Priority Queues
(An Application of Heap)

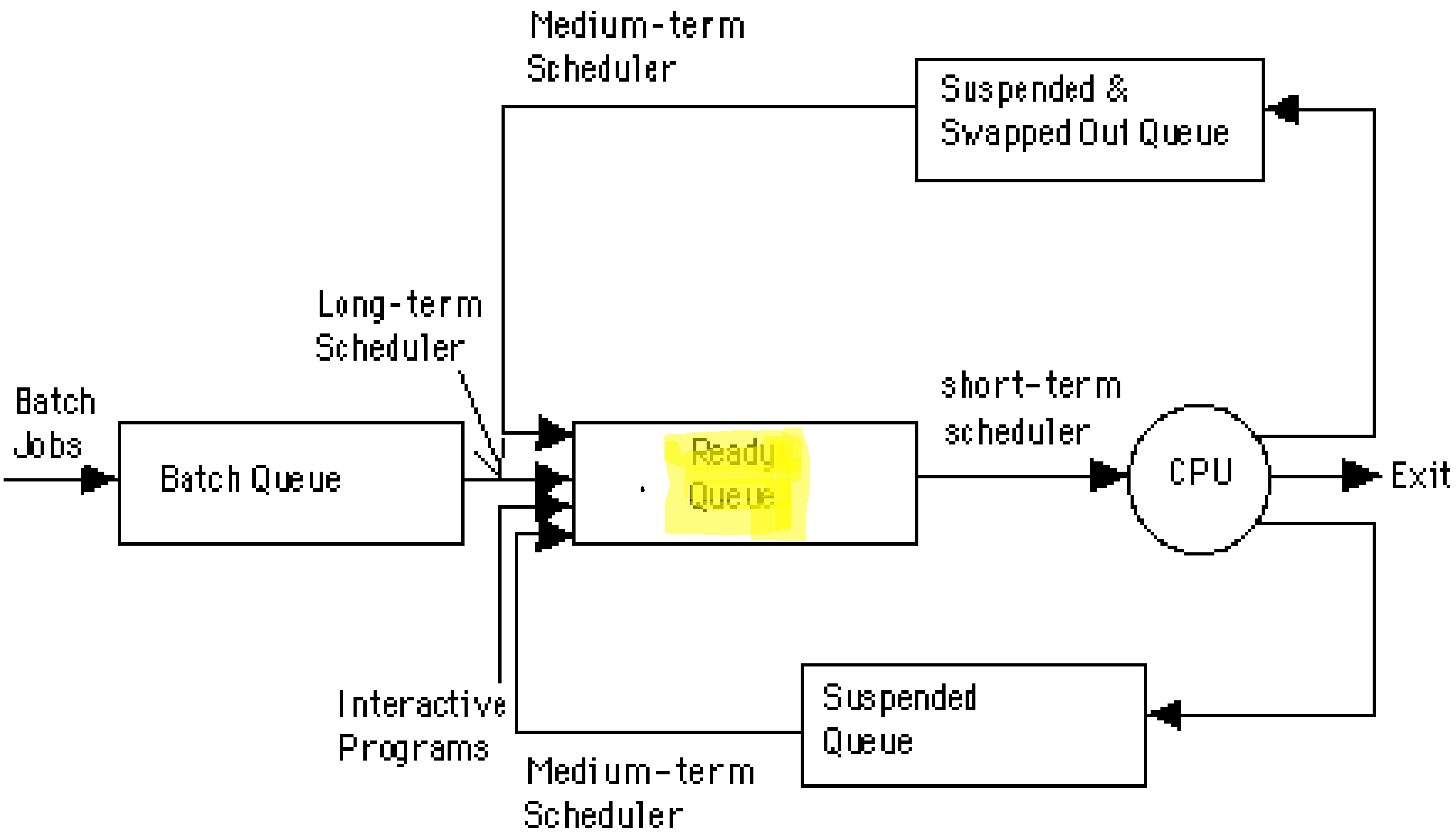
- A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key/priority.
- There's no "real" FIFO rule anymore.
- Two kinds : max-priority queues and min-priority queues, according to max-heaps and min-heaps.
- The key denotes the priority
- Max-heap is used to implement a max-Priority Queue
  - Always deletes & returns (extracts) an element with maximum priority
- Min-heap is used to implement a min-Priority Queue
  - Always deletes & returns (extracts) an element with minimum priority

Max – Priority Queue Applications

Example: **job scheduling** on shared computer

- jobs have priorities, are stored in a max-priority queue
- each time a new job is to be scheduled, it's got to be one of highest priority (**Extract-Max** operation)
- new jobs can be inserted using **Insert** operation
- in order to avoid "starvation", priorities can be increased (**Increase-Key** operation)

Max – Priority Queue Applications



Max- Priority Queue Operations

- **Insert**(S, x) inserts element x into set S
- **Maximum**(S) returns element of S with largest key
- **Extract-Max**(S) removes and returns element of S with largest key
- **Increase-Key**(S, x, k) increases x's key to new value k, assuming k is at least as large as x's old key

Min-priority queues offer Insert, Minimum, Extract-Min, and Decrease-Key.

Max – Priority Queue

Heaps are very convenient here:

- using max-heaps, we know that the largest element is in $A[1]$: we have $O(1)$ access to largest element
- removing/inserting elements and increasing keys means that we (basically) can call **Max-Heapify** at the right place (relatively efficient operation)

Max-Priority Queue Implementation

Heap-Maximum(A) <------ **O(1)**

1. **return** A[1]

Heap-Extract-Max(A) <------ **O(log n)**

1. if *heap-size*[A] < 1
2. **then error** “heap underflow”
3. *max* $\leftarrow$ A[1]
4. A[1] $\leftarrow$ A[*heap-size*[A]]
5. *heap-size*[A] $\leftarrow$ *heap-size*[A] - 1
6. Max-Heapify(A,1)
7. **return** *max*

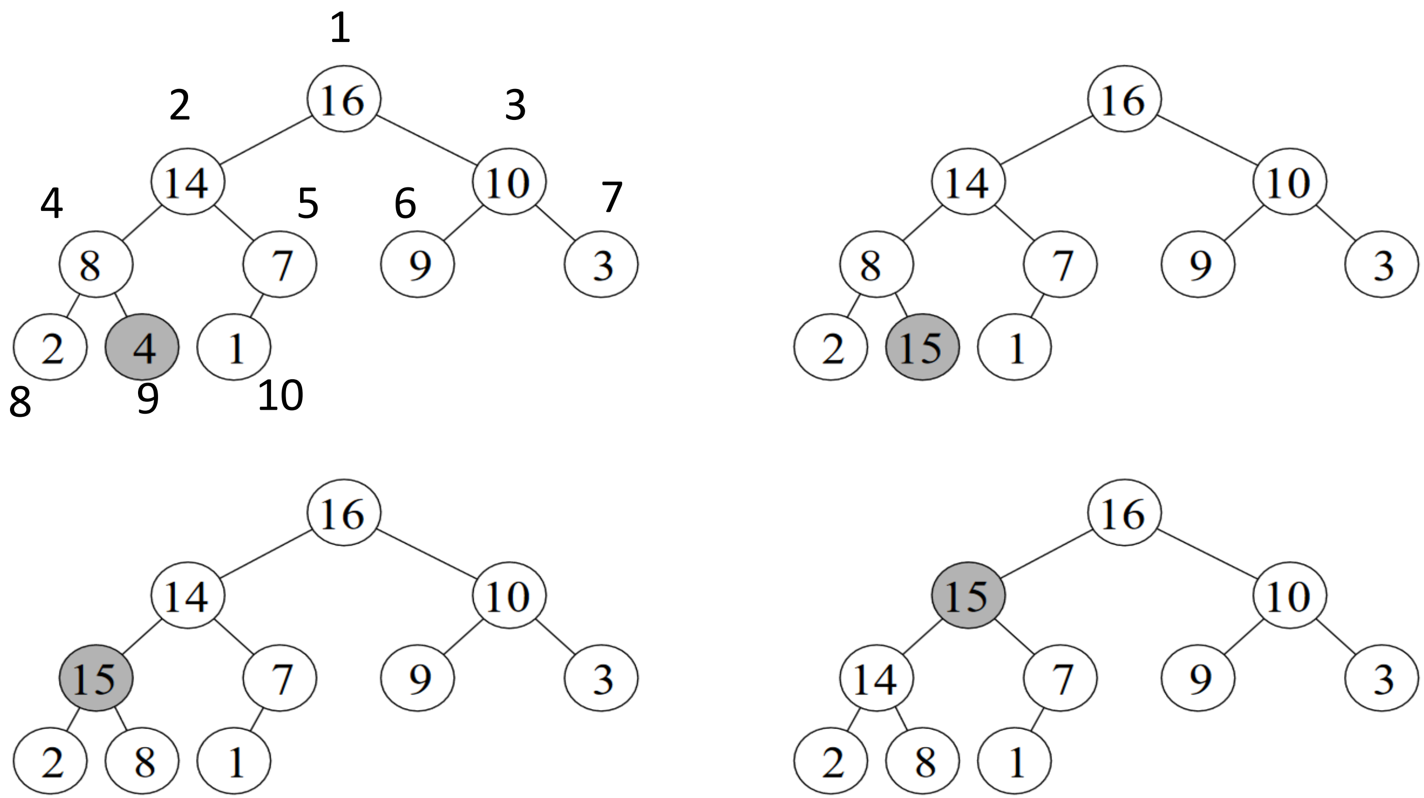
Technically, this part is removal from heap!

Example of Heap-Increase-Key

Example:

Heap-Increase-Key($A, 9, 15$)

Updates node 9 from 4 to 15



Max-Priority Queue Implementation

Heap-Increase-Key(A,i,key)

<-------- **O(log n)**
Height of the tree

1. **if** key < A[i]
2. **then error** "new key is smaller than current key"
3. A[i] ← key
4. **while** i > 1 and A[Parent(i)] < A[i]
5. **do** exchange A[i] ←→ A[Parent(i)]
6. i ← Parent(i)

Technically, this part is insertion into heap!

Max-Heap-Insert(A,key)

<-------- **O(log n)**

1. *heap-size*[A] ← *heap-size*[A]+1
2. A[*heap-size*[A]] ← $-\infty$
3. Heap-Increase-Key(A, *heap-size*[A], key)

Applications of Heap

- Heaps are used in heapsort!
- *Priority Queues:* Priority queues can be efficiently implemented using Heaps
- *Order statistics:* The Heap data structure can be used to efficiently find the kth smallest (or largest) element in an array.
- Heap Implemented priority queues are used in Graph algorithms like Prim’s Algorithm and Dijkstra’s algorithm.

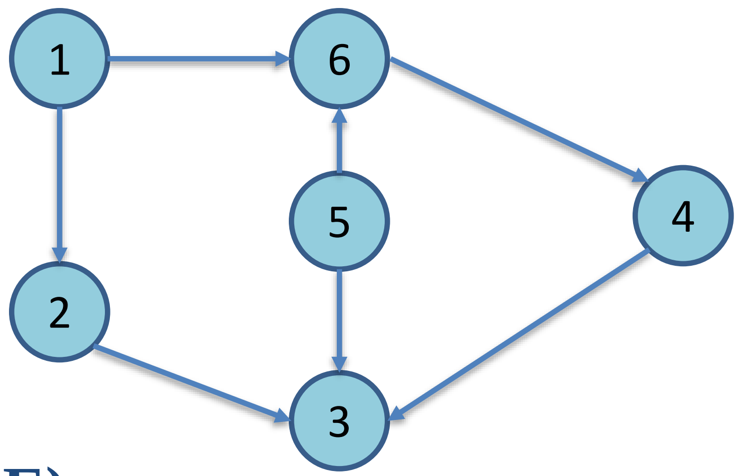
Exercise 1

- Write an algorithm to compute product of three minimum numbers [least 3 numbers] of a given array using min-heap.
- Example:
  - Original array elements:
    [**12**, 74, **9**, 50, 61, **41**]
  - Product of three minimum numbers of the given array:
    4428 [9 X 12 X 41]
- Write both steps and the algorithm
- Analyze the complexity of your solution

Graph

A graph G is defined as a pair of two sets V and E, where

- ***V*** is a set of nodes, called **vertices**
- ***E*** is a collection of pairs of vertices, called **edges**

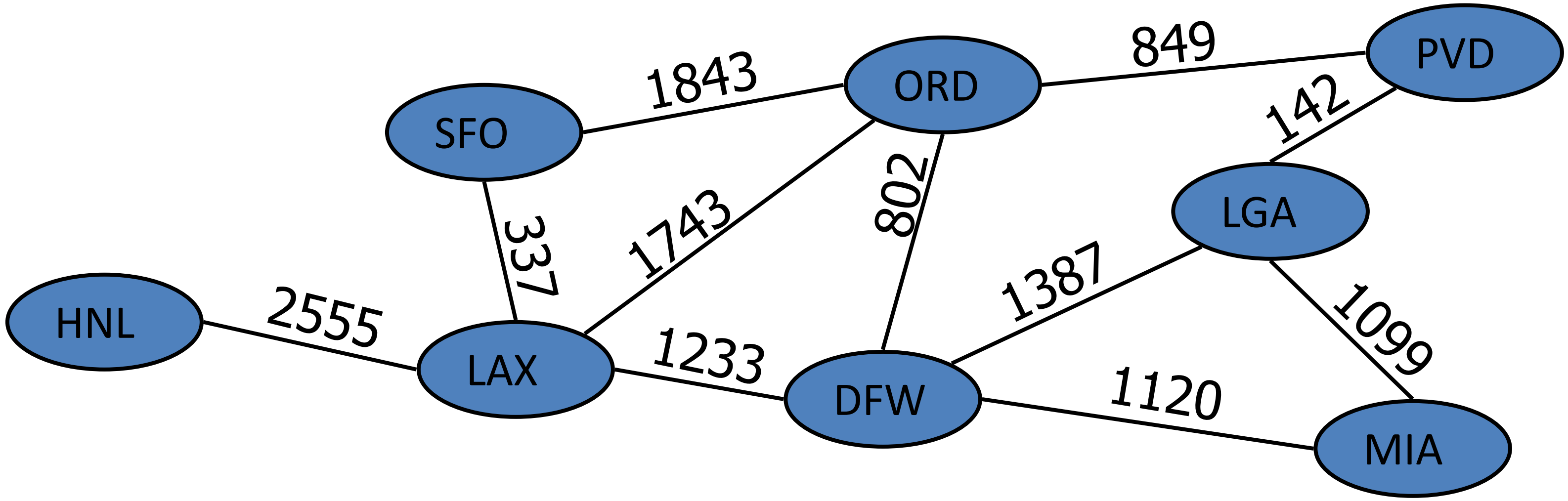


- **G = (V,E)**
- **V = { 1,2,3,4,5,6 } and |V| = 6**
- **E = { (1,6), (1,2), (2,3), (4,3), (5,3), (5,6), (6,4) } and |E| = 7**

Example

Example:

- A **vertex** represents an **airport** and stores the **three-letter airport code**
- An **edge** represents a flight route between two airports and stores the **mileage** of the route



Other Examples

Can you think of other examples of graphs modelling real – world situations ?

- Google maps !
- Facebook and friends ☺
- World Wide Web and Hyperlinks …
- Operating Systems for resource allocation
- …

Electronic circuits

Printed circuit board

Integrated circuit

Transportation networks

Highway network

Flight network

Computer networks

Local area network

Internet

Web

Databases

Entity-relationship diagram

Tree vs Graph

BASIS FOR COMPARISON	TREE	GRAPH
Path	Only one between two vertices.	More than one path is allowed.
Root node	It has exactly one root node.	Graph doesn't have a root node.
Loops	No loops are permitted.	Graph can have loops.
Complexity	Less complex	More complex comparatively
Traversal techniques	Pre-order, In-order and Post-order.	Breadth-first search and depth-first search.
Number of edges	n-1 (where n is the number of nodes)	Not defined
Model type	Hierarchical	Network

Edge Types

Directed edge (**with arrow**)

- ordered pair of vertices (***u***,***v***)
- first vertex ***u*** is the origin
- second vertex ***v*** is the destination
- e.g., a flight

Undirected edge (**no arrow**)

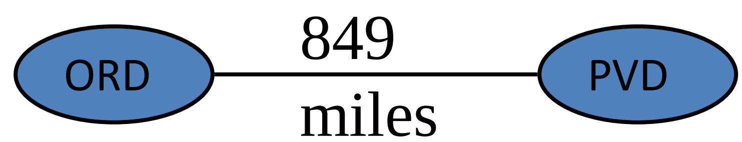
- unordered pair of vertices (***u***,***v***)
- e.g., a flight route

Directed graph

- all the edges are directed
- e.g., flight network

Undirected graph

- all the edges are undirected
- e.g., route network

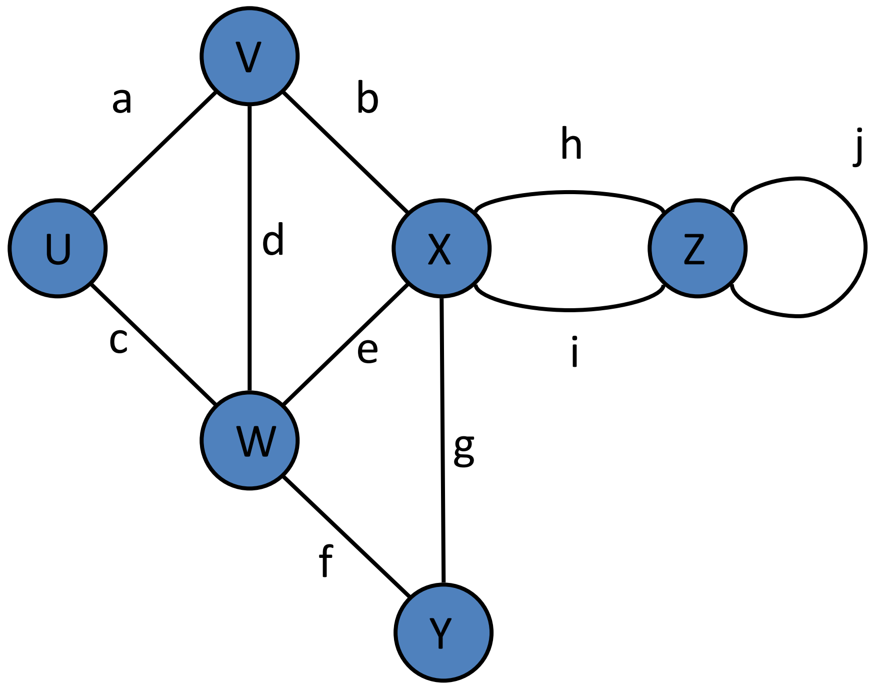


Analogy:

- A road between two points could be **one way** (directed) Or **two way** (undirected)
- Could have more than one edge (e.g. toll road and public road)

Graph Terminology

- End vertices (or endpoints) of an edge
  - *Ex: U and V are the endpoints of a*
- Edges incident on a vertex
  - *Ex: a, d, and b are incident on V*
- Adjacent vertices
  - *Ex: U and V are adjacent*
- Degree of a vertex
  - *Ex: X has degree 5*
- Parallel edges
  - *Ex: h and i are parallel edges*
- Self-loop
  - *Ex: j is a self-loop*



Graph Terminology

innovate achieve lead

Path

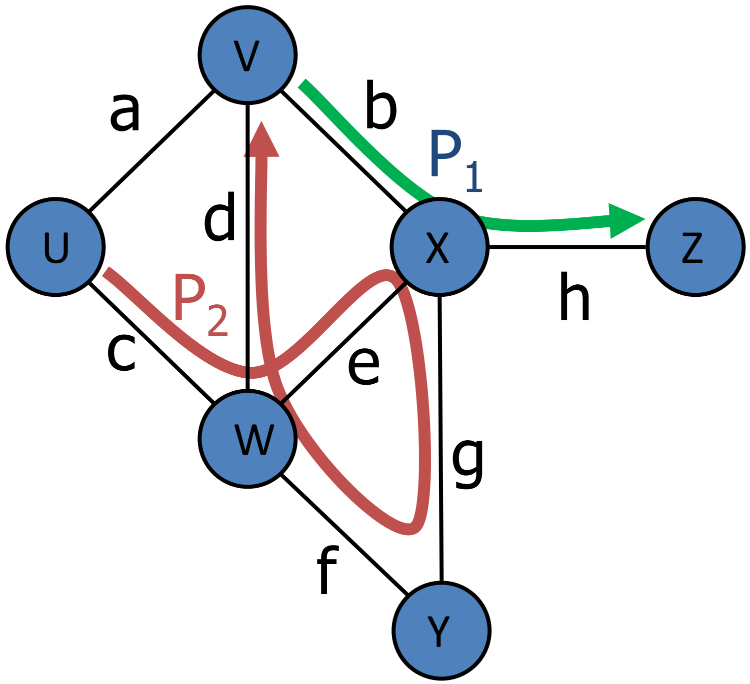
- Sequence of alternating vertices and edges
- Begins with a vertex
- Ends with a vertex
- Each edge is preceded and followed by its endpoints

Simple path

- Path such that all its vertices and edges are *distinct*

Examples

- P_1=(V,b,X,h,Z) is a **simple** path
- P_2=(U,c,W,e,X,g,Y,f,W,d,V) is a path that is **not simple**



Graph Terminology

Cycle

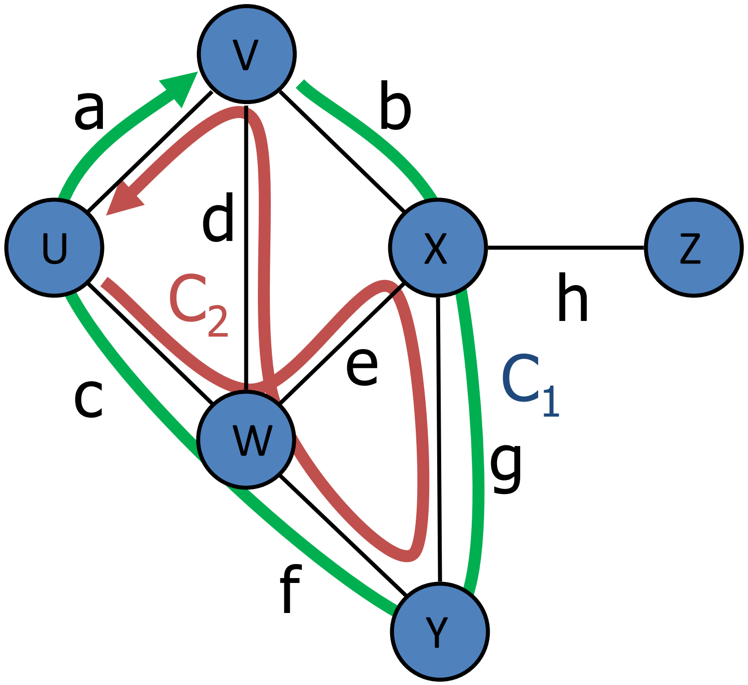
- **Circular** sequence of alternating vertices and edges
- each edge is preceded and followed by its endpoints

Simple Cycle

- Cycle such that all its vertices and edges are **distinct**

Examples

- C_1=(V,b,X,g,Y,f,W,c,U,a,↵) is a **simple cycle**
- C_2=(U,c,W,e,X,g,Y,f,W,d,V,a,↵) is a cycle that is **not simple**

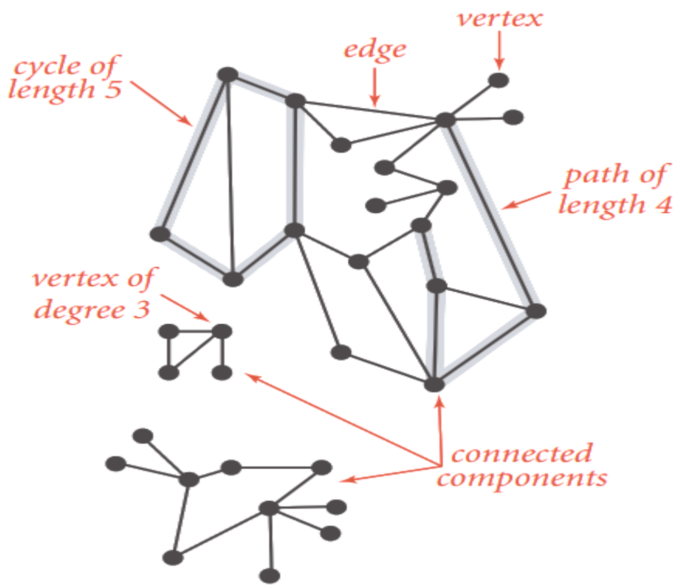


Graph Terminology

Graph terminology

Path. Sequence of vertices connected by edges.
Cycle. Path whose first and last vertices are the same.

Two vertices are connected if there is a path between them.

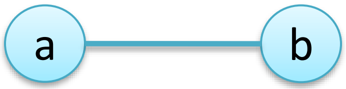


Graph Types

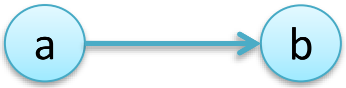
Null Graph : A graph having no edges.

Trivial Graph : A graph with only one vertex.

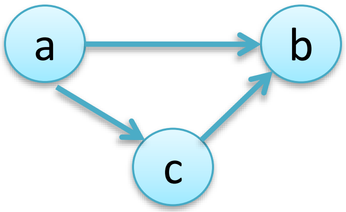
Undirected Graph : A graph that contains edges but the edges are not directed ones.



Directed Graph : A graph that contains edges which have direction.



Simple Graph : A graph that contains no loops and no parallel edges.

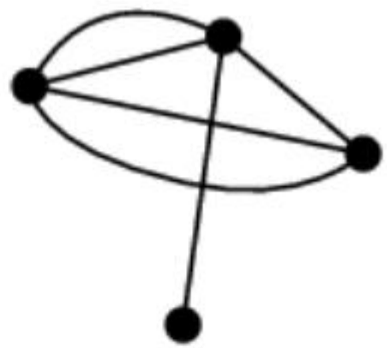


Many more ……

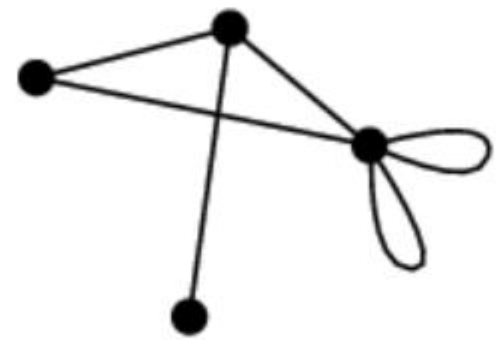
Quick Check

Which of the below is a Simple Graph ?

Simple Graph



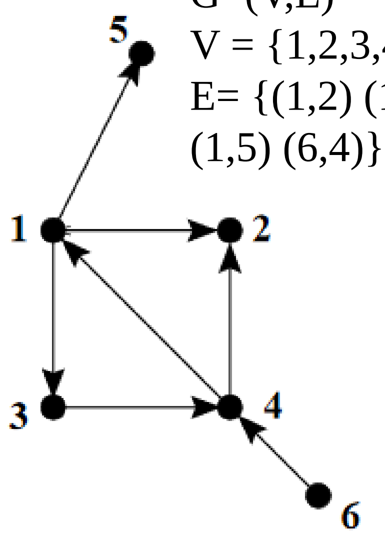
Non-Simple Graph with multiple (parallel) edges



Non-Simple Graph with loops

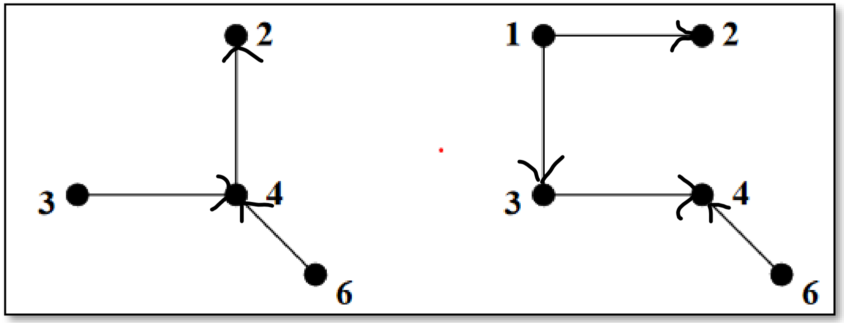
Subgraphs

- A **subgraph** S of a graph G is a graph such that :
  - The **vertices** of S are a **subset** of the vertices of G
  - The **edges** of S are a **subset** of the edges of G
- A **spanning subgraph** of G is a subgraph that contains all the **vertices** of G.

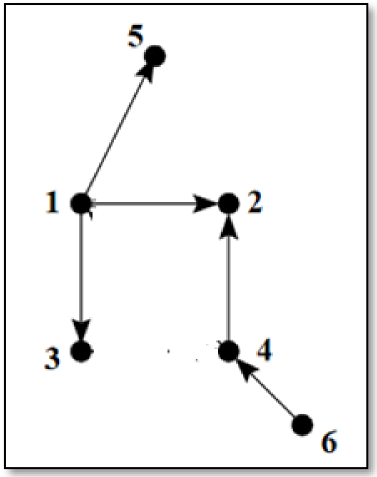


G=(V,E)
V = {1,2,3,4,5,6}
E= {(1,2) (1,3) (4,1) (4,2) (3,4) (1,5) (6,4)}

Subgraphs :

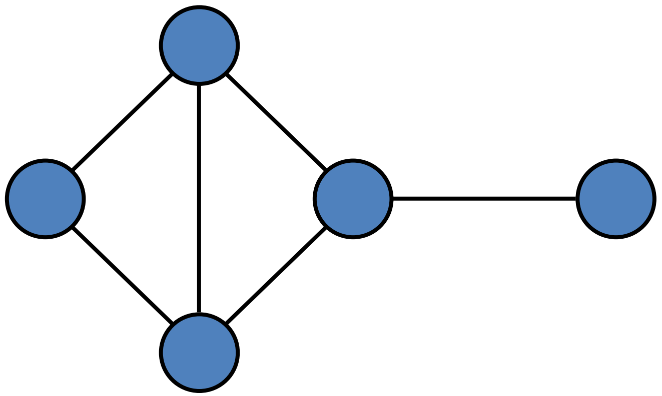


Spanning subgraph:

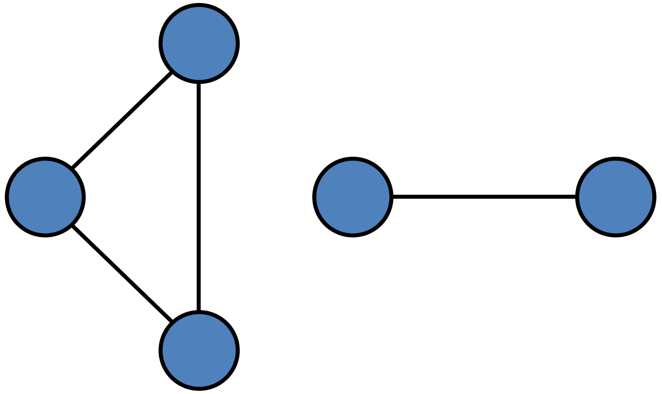


Connectivity

- A graph is **connected** if there is a path between every pair of vertices.
- From every vertex to any other vertex, there should be some path to traverse.
- That is called the connectivity of a graph. A graph with multiple disconnected vertices and edges is said to be disconnected.



Connected graph

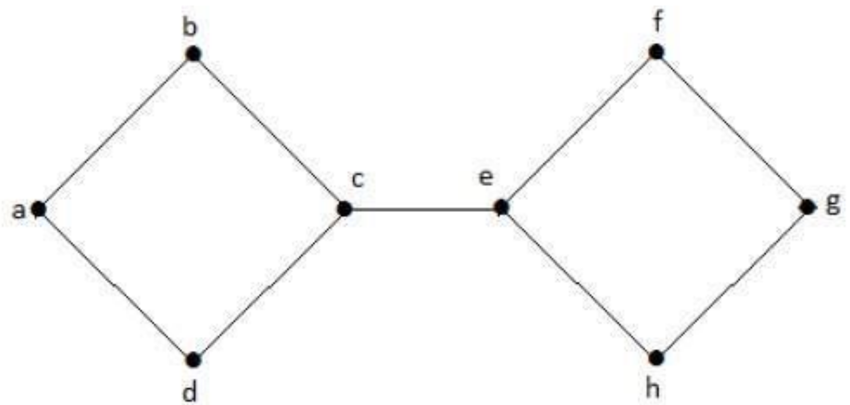


Non connected graph with two connected components

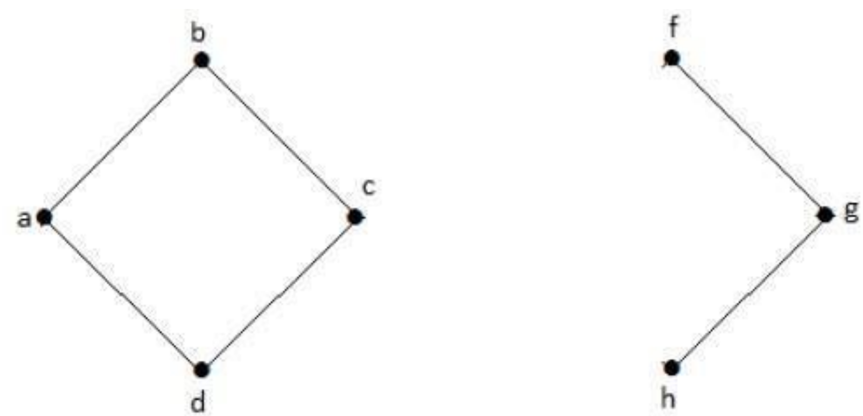
Connectivity – Cut Vertex

- Let ‘G’ be a connected graph. A vertex V ∈ G is called a cut vertex of ‘G’, if ‘G-V’ (Delete ‘V’ from ‘G’) results in a disconnected graph. Removing a cut vertex from a graph breaks it in to two or more graphs.

Note *− Removing a cut vertex will render a graph disconnected.*



By removing ‘e’ or ‘c’, the graph will become a disconnected graph.

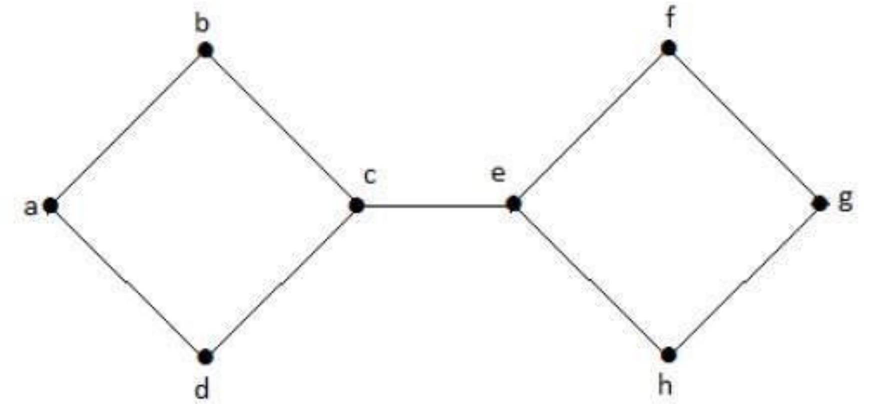


Hence it is a disconnected graph with cut vertex as ‘e’. Similarly, ‘c’ is also a cut vertex for the above graph.

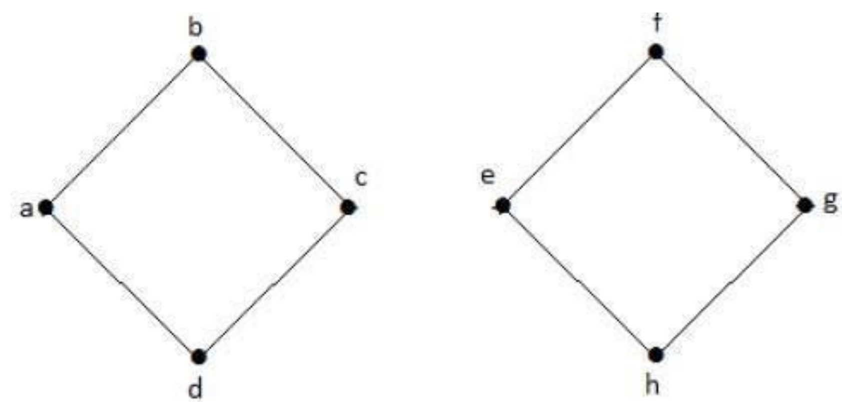
Connectivity – Cut Edge

- Let ‘G’ be a connected graph. An edge ‘e’ ∈ G is called a cut edge if ‘G-e’ results in a disconnected graph.
- If removing an edge in a graph results in to two or more graphs, then that edge is called a Cut Edge.

Note − *Removing a cut edge will render a graph disconnected.*



By removing the edge (c, e) from the graph, it becomes a disconnected graph.

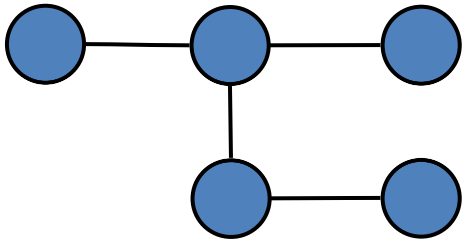


In the above graph, removing the edge (c, e) breaks the graph into two which is nothing but a disconnected graph. Hence, the edge (c, e) is a cut edge of the graph.

Trees and Forests

- A (free) **tree** is an undirected graph T such that
  - T is **connected**
  - T has **no cycles**

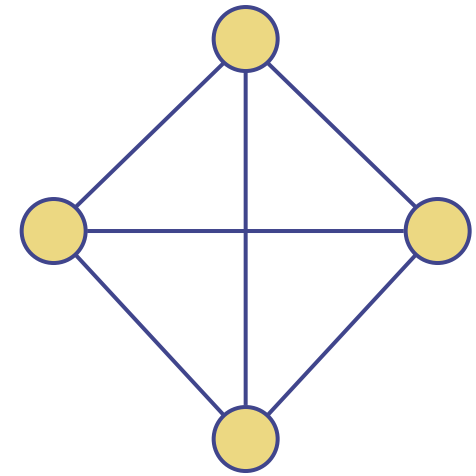
    [This definition of tree is different from the one of a rooted tree]

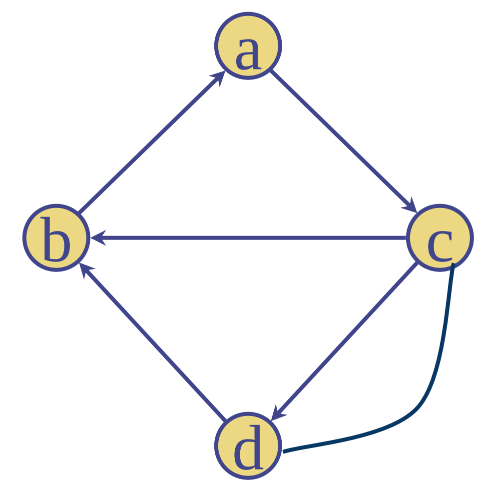


Tree

Operations on a Graph

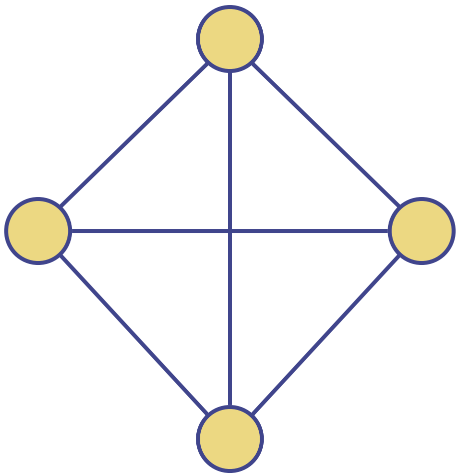
- Return the number, n, of vertices in G.
- Return the number, m, of edges in G.
- Return a set or list containing all n vertices in G.
- Return a set or list containing all m edges in G.
- Return some vertex, v, in G.
- Return the degree, deg(v), of a given vertex, v, in G.
- Return a set or list containing all the edges incident upon a given vertex, v, in G.
- Return a set or list containing all the vertices adjacent to a given vertex, v, in G.
- Return the two end vertices of an edge, e, in G; if e is directed, indicate which vertex is the origin of e and which is the destination of e.
- Return whether two given vertices, v and w, are adjacent in G.

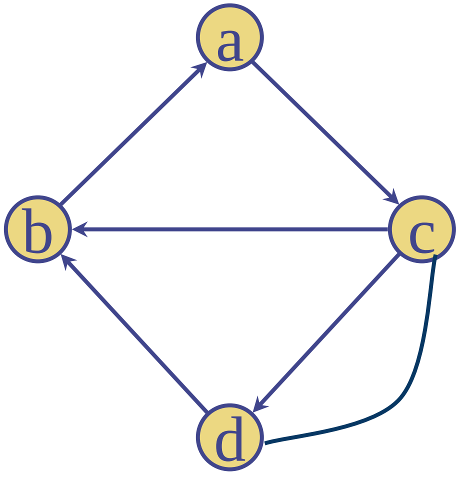




Operations on a Graph

- Indicate whether a given edge, e, is directed in G.
- Return the in-degree of v, inDegree(v).
- Return a set or list containing all the incoming (or outgoing) edges incident upon a given vertex, v, in G.
- Return a set or list containing all the vertices adjacent to a given vertex, v, along incoming (or outgoing) edges in G.
- Insert a new directed (or undirected) edge, e, between two given vertices, v and w, in G.
- Insert a new (isolated) vertex, v, in G.
- Remove a given edge, e, from G.
- Remove a given vertex, v, and all its incident edges from G.



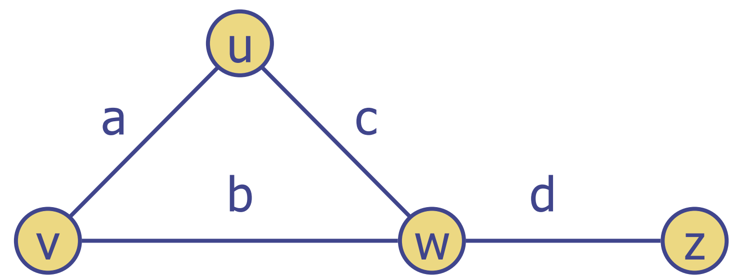


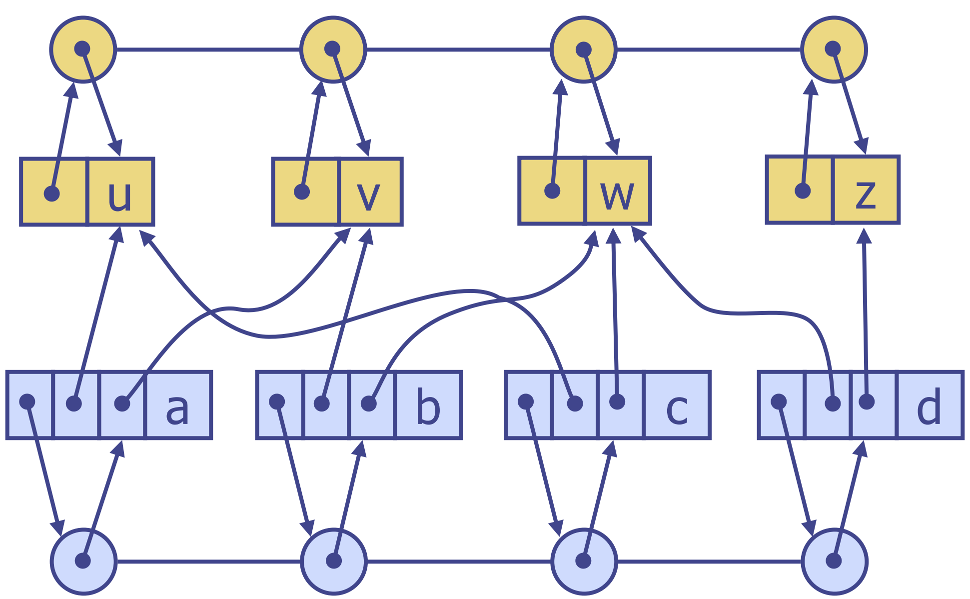
Graph Representation Strategies

- Edge List
- Adjacency Matrix
- Adjacency List

Edge List

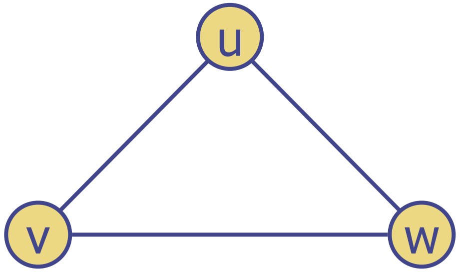
- Vertex object
  - element
  - reference to position in vertex sequence
- Edge object
  - element
  - origin vertex object
  - destination vertex object
  - reference to position in edge sequence
- Vertex sequence
  - sequence of vertex objects
- Edge sequence
  - sequence of edge objects





Edge List Simplified

```
[
   (u,v),
   (v,w),
   (u,w)
]
```



Example 2:

edge_list = [(0, 1), (1, 2), (2, 3), (0, 2), (3, 2), (4, 5), (5, 4)]

Adjacency Matrix

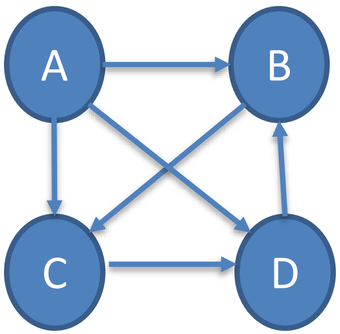
innovate achieve lead

The adjacency matrix A of a graph G is formally defined as :

$$A[i][j] = \begin{cases} 1 & \textit{if there is an edge from vertex } i \textit{ to vertex } j \\ 0 & \textit{if there is no edge from vertex } i \textit{ to vertex } j \end{cases}$$

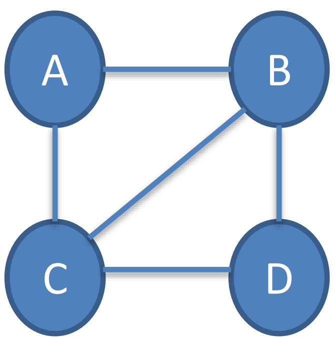
- It is clear from the definition that an adjacency matrix of a graph with n vertices is a boolean square matrix with n rows and n columns with entries 1's and 0's (bit-matrix)
- Note that the above definition holds true for both directed and undirected graph. *If it's a adjacency matrix of a undirected graph, then the matrix will look symmetric and diagonal being 0's*.
- Note: If there are multiple edges from vertex u to v, then the number of edges can also be used in the matrix.

Adjacency Matrix Examples



Adjacency Matrix

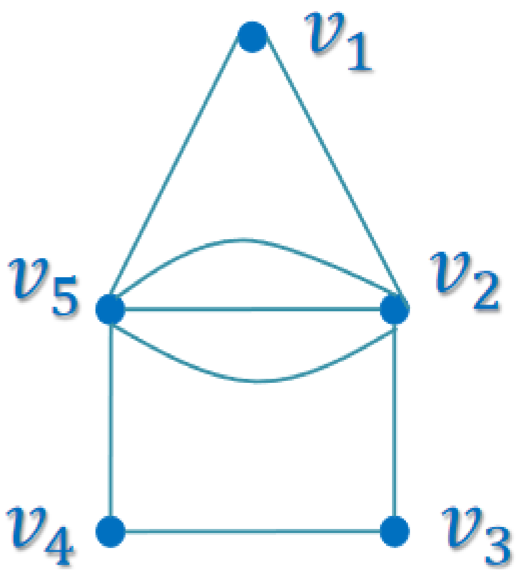
	A	B	C	D
A	0	1	1	1
B	0	0	1	0
C	0	0	0	1
D	0	1	0	0



Adjacency Matrix

	A	B	C	D
A	0	1	1	0
B	1	0	1	1
C	1	1	0	1
D	0	1	1	0

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

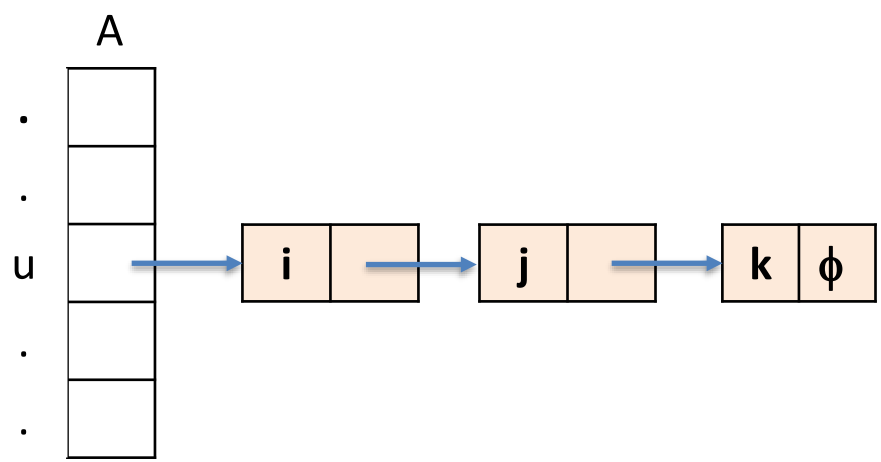
- Fast to tell whether edge exists between any two vertices *i* and *j* (and to get its weight)

Disadvantages

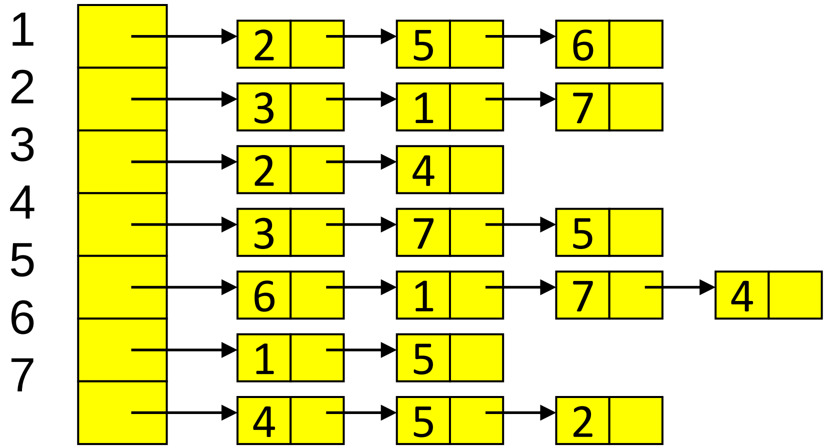
- Consumes a lot of memory on **sparse** graphs (ones with few edges)
- Redundant information for undirected graphs

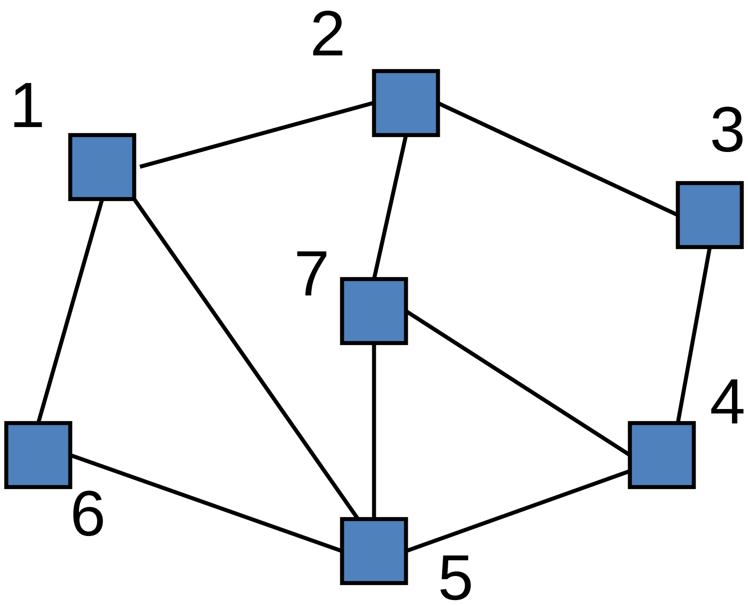
Adjacency Lists

- 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 $\in$ 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





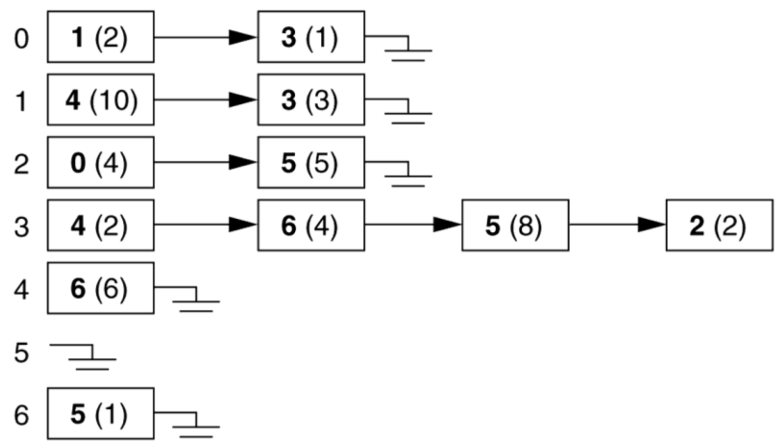
Adjacency List: Pros and Cons

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)



Asymptotic Performance

• n vertices, m edges • no parallel edges • no self-loops • Bounds are “big-Oh”	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$	$n + m$	n^2
incidentEdges(v)	m	deg(v)	n
areAdjacent (v, w)	m	min(deg(v), deg(w))	1
insertVertex(o)	1	1	n^2
insertEdge(v, w, o)	1	1	1
removeVertex(v)	m	deg(v)	n^2
removeEdge(e)	1	1	1

Exercise 2

Consider the following two adjacency matrices:

$$
(1)\quad \begin{array}{c|cccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline
1 & 0 & 1 & 0 & 0 & 1 & 0 \\
2 & 1 & 0 & 1 & 0 & 0 & 1 \\
3 & 0 & 1 & 0 & 0 & 0 & 1 \\
4 & 0 & 0 & 0 & 0 & 1 & 0 \\
5 & 1 & 0 & 0 & 1 & 0 & 1 \\
6 & 0 & 1 & 1 & 0 & 1 & 0 \\ \hline
\end{array}
\qquad
(2)\quad \begin{array}{c|cccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline
1 & 0 & 1 & 0 & 0 & 0 & 0 \\
2 & 1 & 0 & 0 & 0 & 0 & 0 \\
3 & 0 & 1 & 0 & 0 & 0 & 1 \\
4 & 1 & 0 & 0 & 1 & 1 & 0 \\
5 & 1 & 0 & 0 & 1 & 0 & 1 \\
6 & 0 & 1 & 1 & 0 & 0 & 0 \\ \hline
\end{array}
$$

For each, answer the following questions:

(a) Can this matrix be the adjacency matrix of an undirected graph? Could it represent a directed graph? Why or why not?

(b) For both adjacency matrices draw the corresponding undirected graph if possible, otherwise draw the corresponding directed graph.

(c) Give the adjacency list representation for both graphs.

Exercise 3

Create graph from given adjacency matrix:

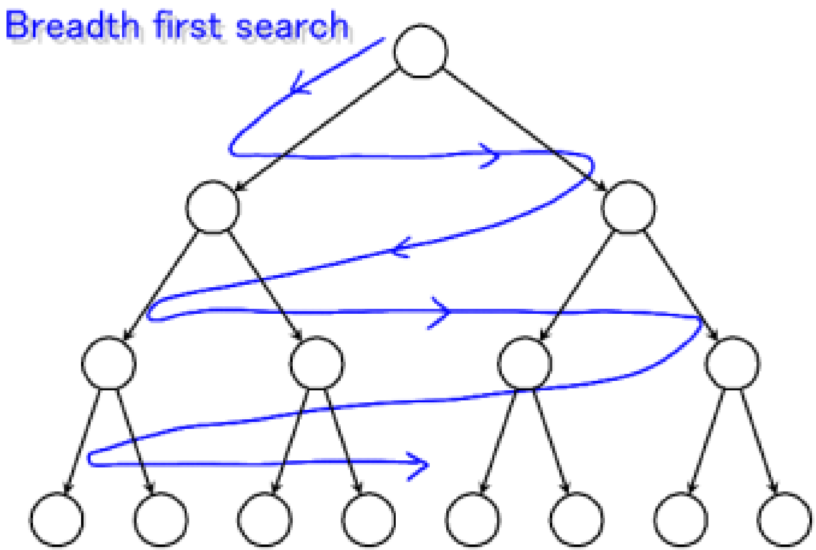
a)

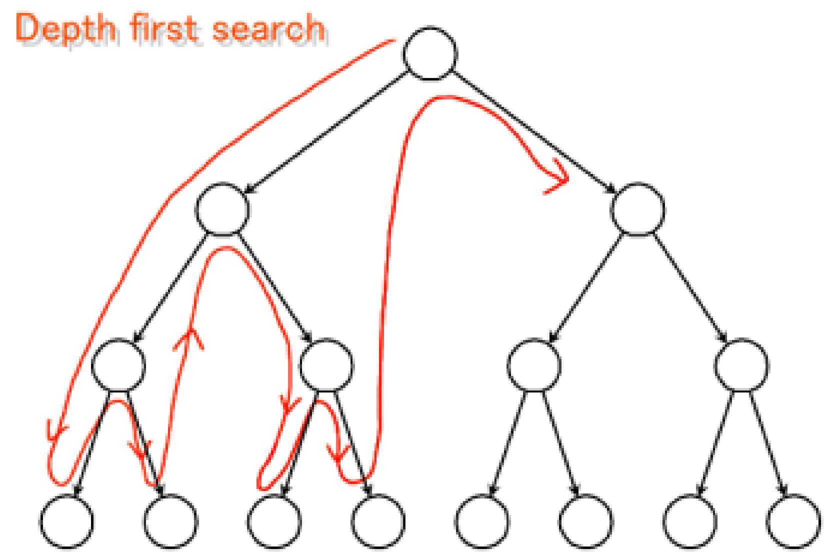
	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

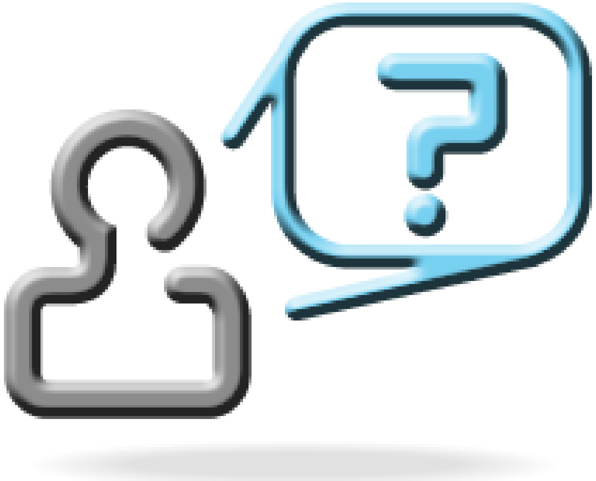
b)

$$A = \begin{bmatrix} 2 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 3 \\ 0 & 0 & 0 & 0 & 3 & 0 \end{bmatrix}$$

Graph Traversals







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

innovate achieve lead

Thank You for your time & attention !

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