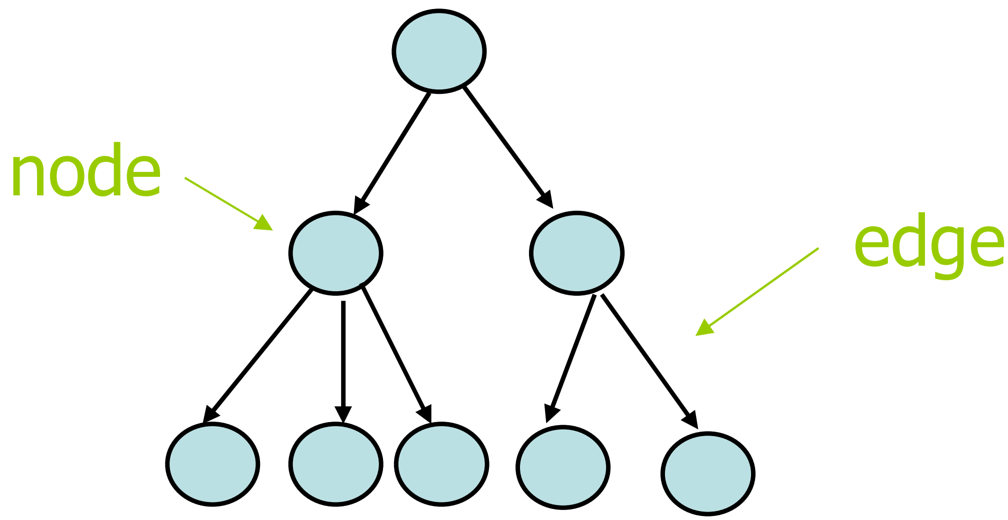
Data Structures

Introduction to trees and graphs

Trees

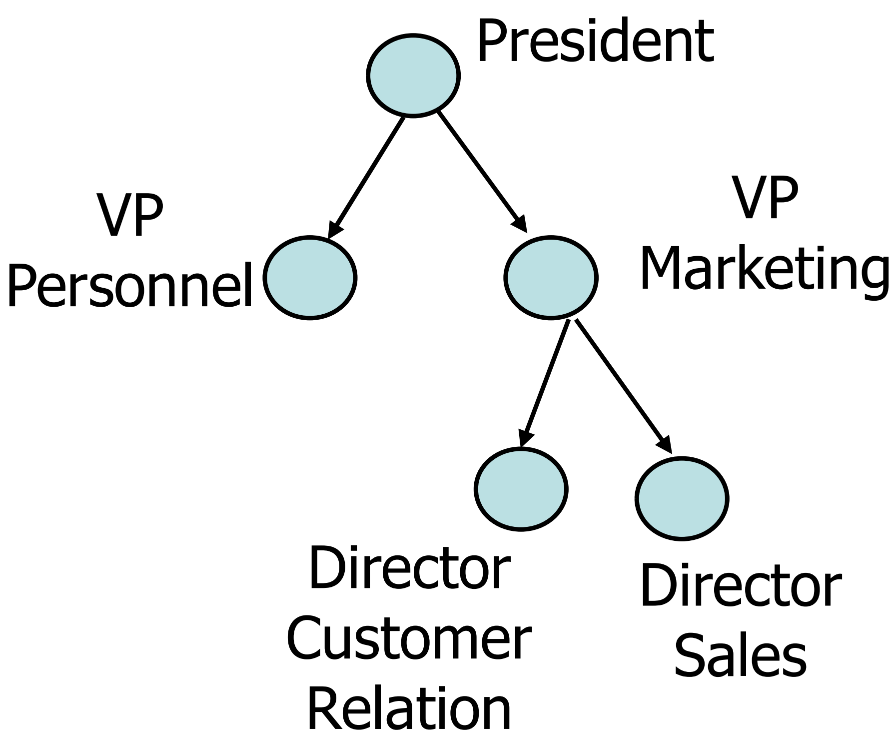
What is a tree?

- Trees are structures used to represent hierarchical relationship
- Each tree consists of nodes and edges
- Each node represents an object
- Each edge represents the relationship between two nodes.

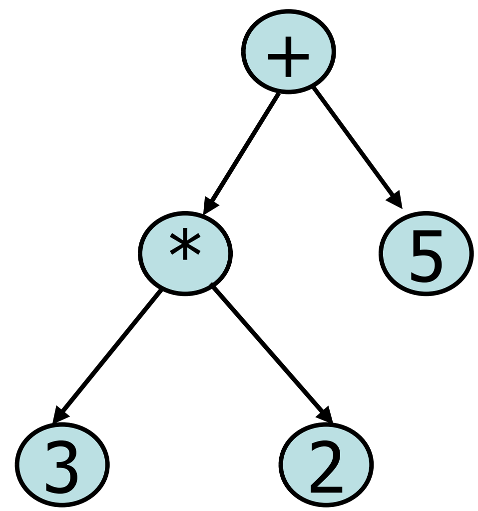


Some applications of Trees

Organization Chart

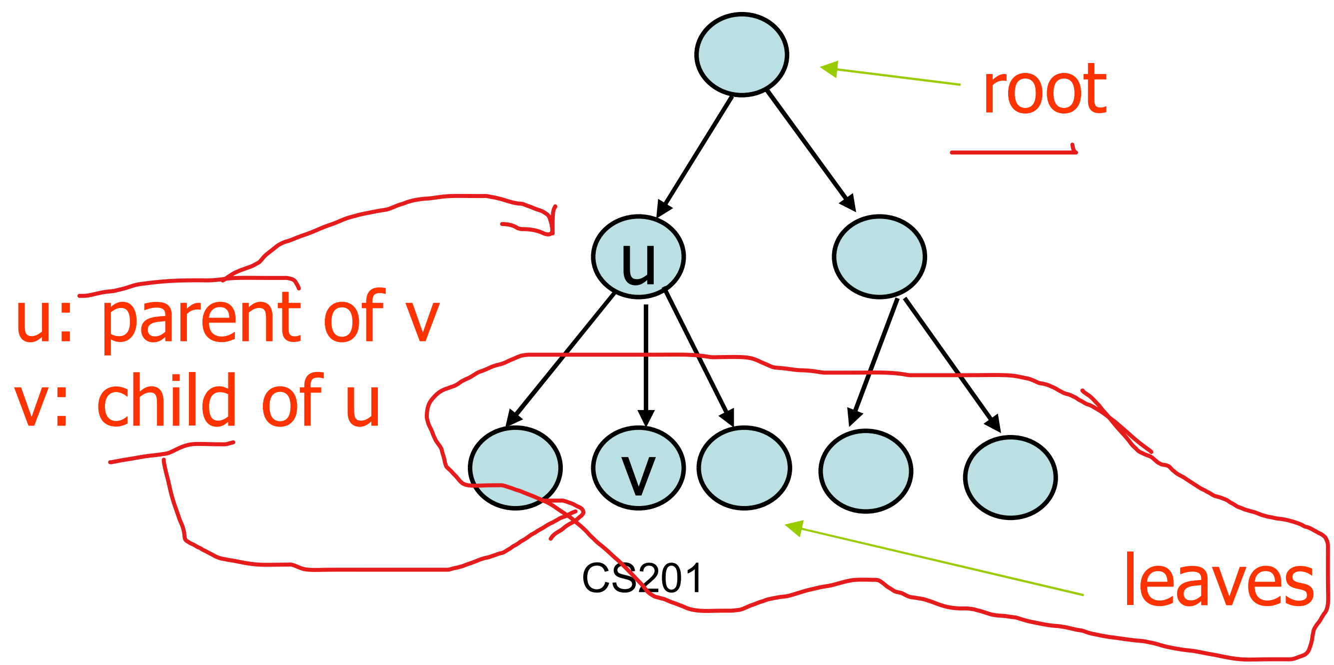


Expression Tree



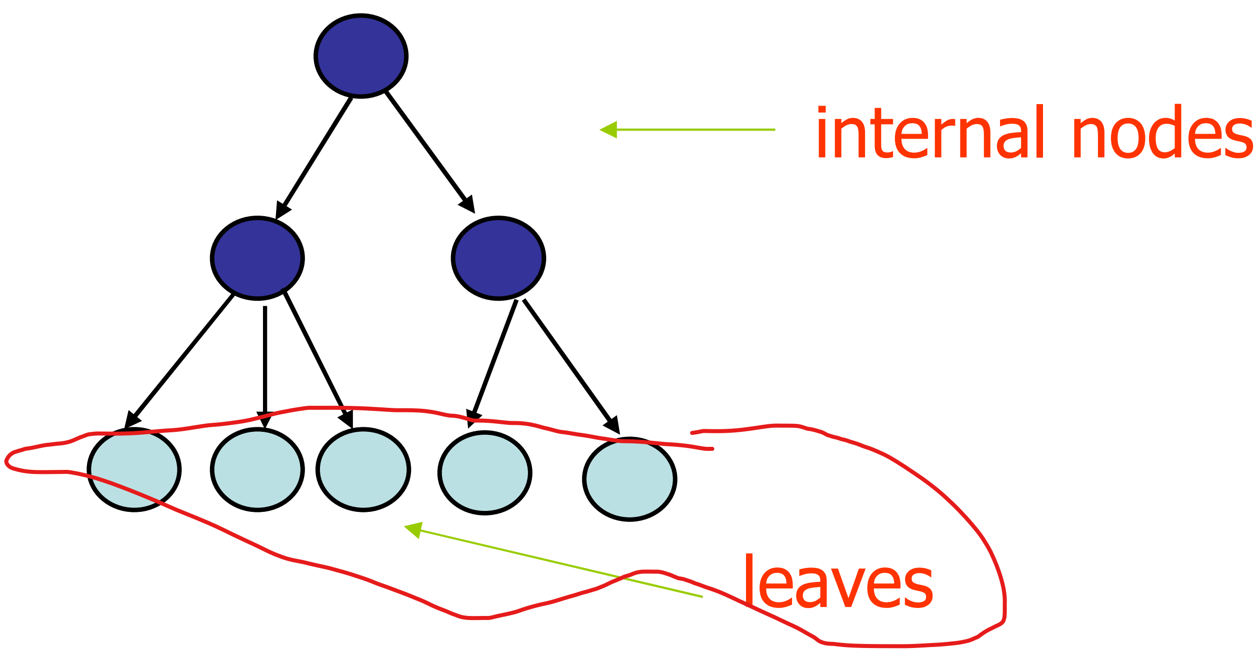
Terminology I

- For any two nodes u and v, if there is an edge pointing from u to v, u is called the parent of v while v is called the child of u. Such edge is denoted as (u, v).
- In a tree, there is exactly one node without parent, which is called the root. The nodes without children are called leaves.



Terminology II

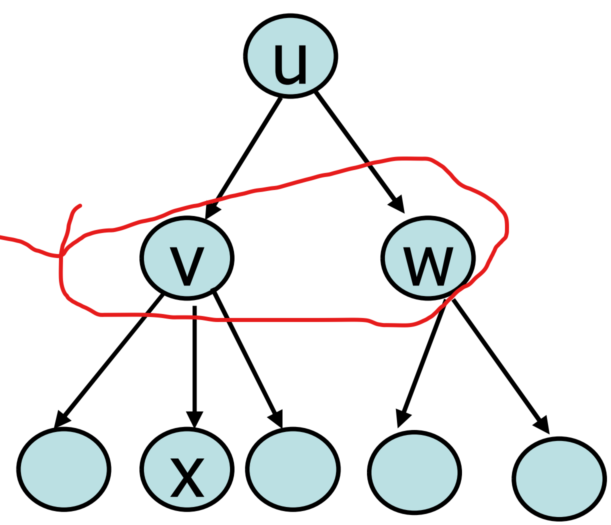
- In a tree, the nodes without children are called leaves. Otherwise, they are called internal nodes.



Terminology III

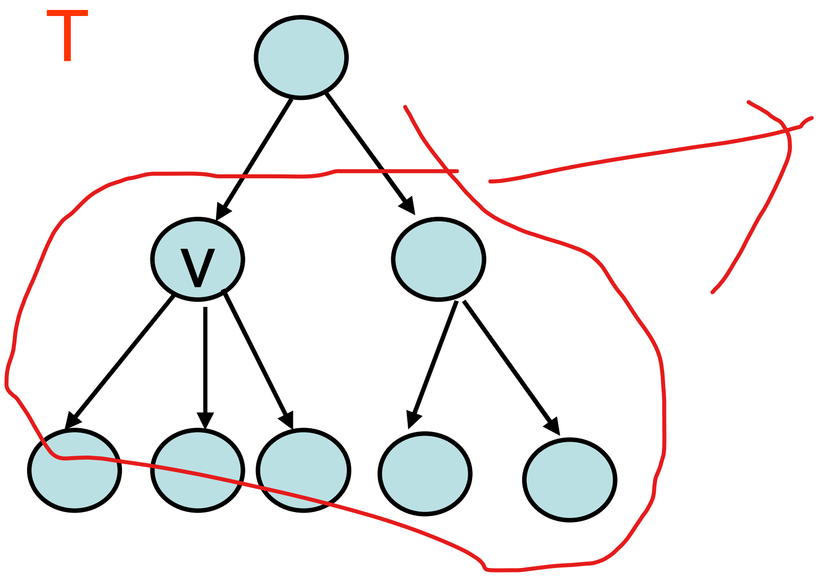
- If two nodes have the same parent, they are siblings.
- A node u is an ancestor of v if u is parent of v or parent of parent of v or …
- A node v is a descendent of u if v is child of v or child of child of v or …

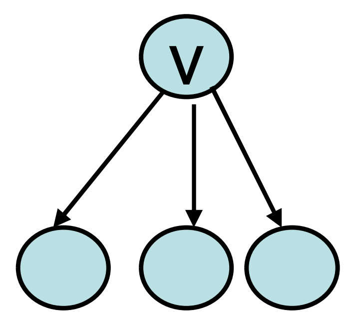
v and w are siblings
u and v are ancestors of x
v and x are descendents of u



Terminology IV

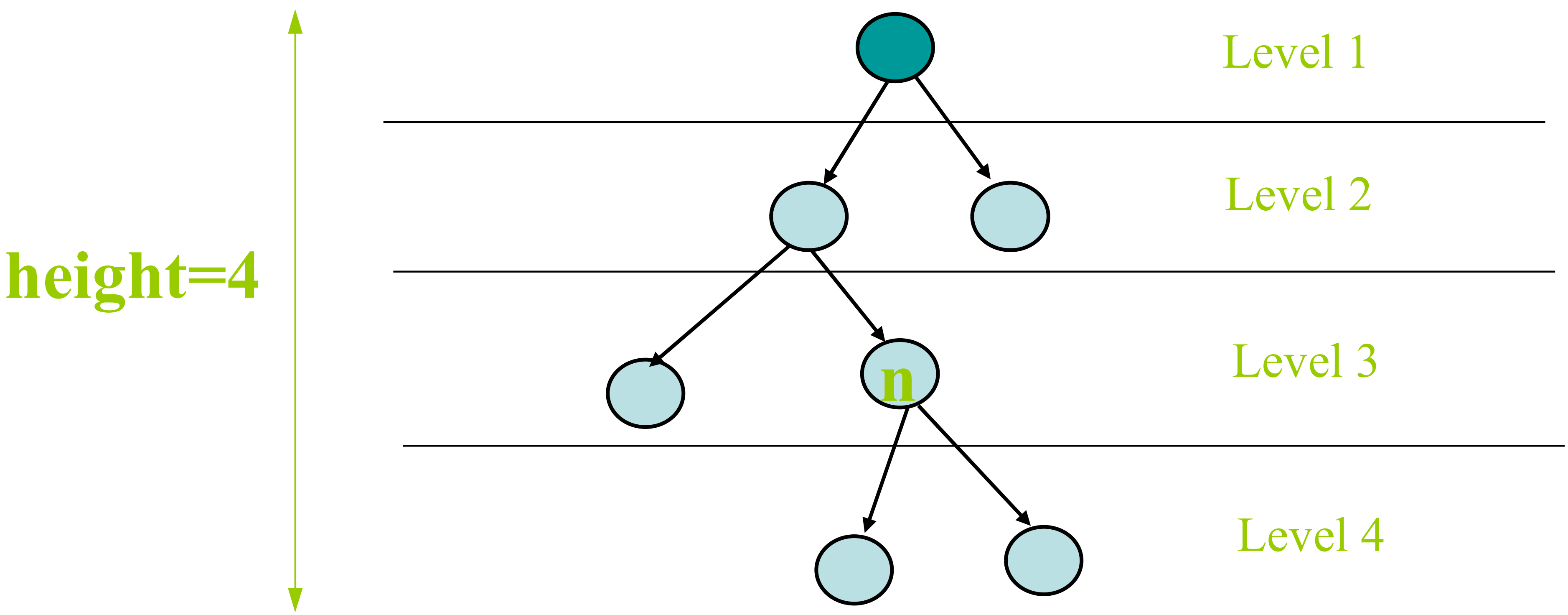
- A subtree is any node together with all its descendants.





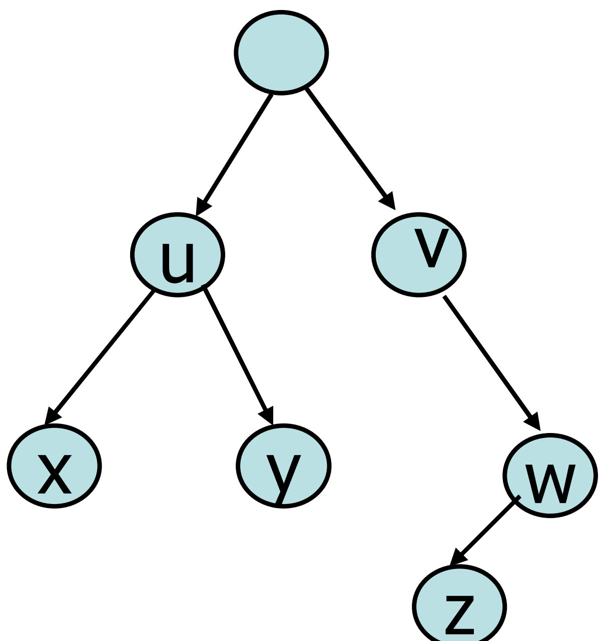
Terminology V

- Level of a node n: number of nodes on the path from root to node n
- Height of a tree: maximum level among all of its node



Binary Tree

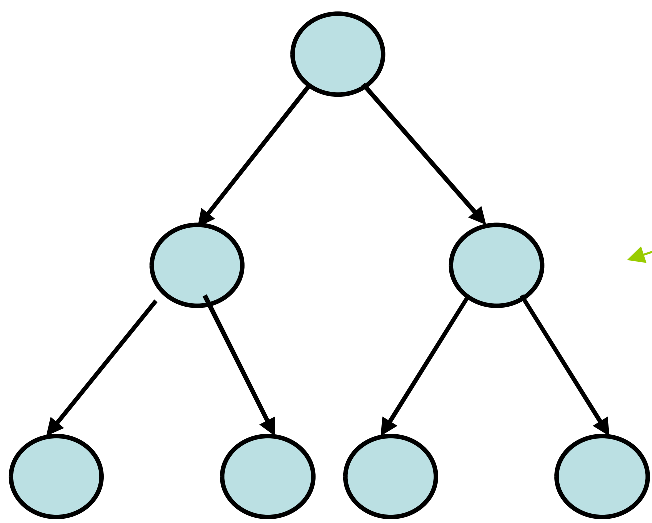
- Binary Tree: Tree in which every node has at most 2 children
- Left child of u: the child on the left of u
- Right child of u: the child on the right of u



x: left child of u
y: right child of u
w: right child of v
z: left child of w

Full binary tree

- If T is empty, T is a full binary tree of height 0.
- If T is not empty and of height h >0, T is a full binary tree if both subtrees of the root of T are full binary trees of height h-1.



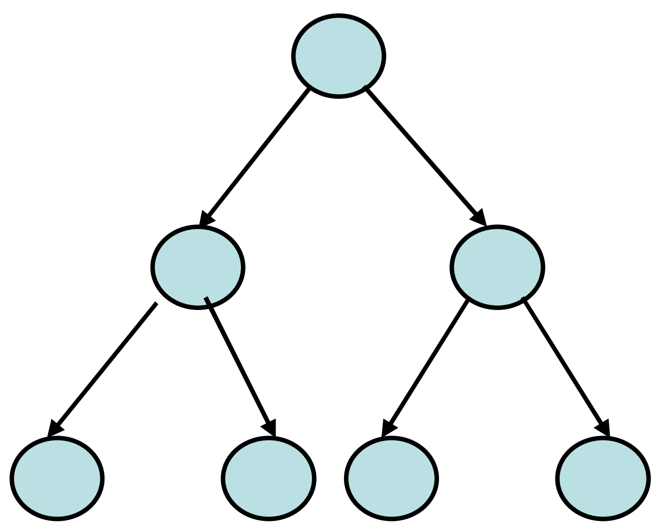
Full binary tree
of height 3

Property of binary tree (I)

- A full binary tree of height h has 2^h-1 nodes

No. of nodes $= 2^0 + 2^1 + ... + 2^{(h-1)}$

$= 2^h - 1$



Level 1: 2^0 nodes

Level 2: 2^1 nodes

Level 3: 2^2 nodes

Property of binary tree (II)

- Consider a binary tree T of height h. The number of nodes of T $\leq 2^h-1$

Reason: you cannot have more nodes than a full binary tree of height h.

Property of binary tree (III)

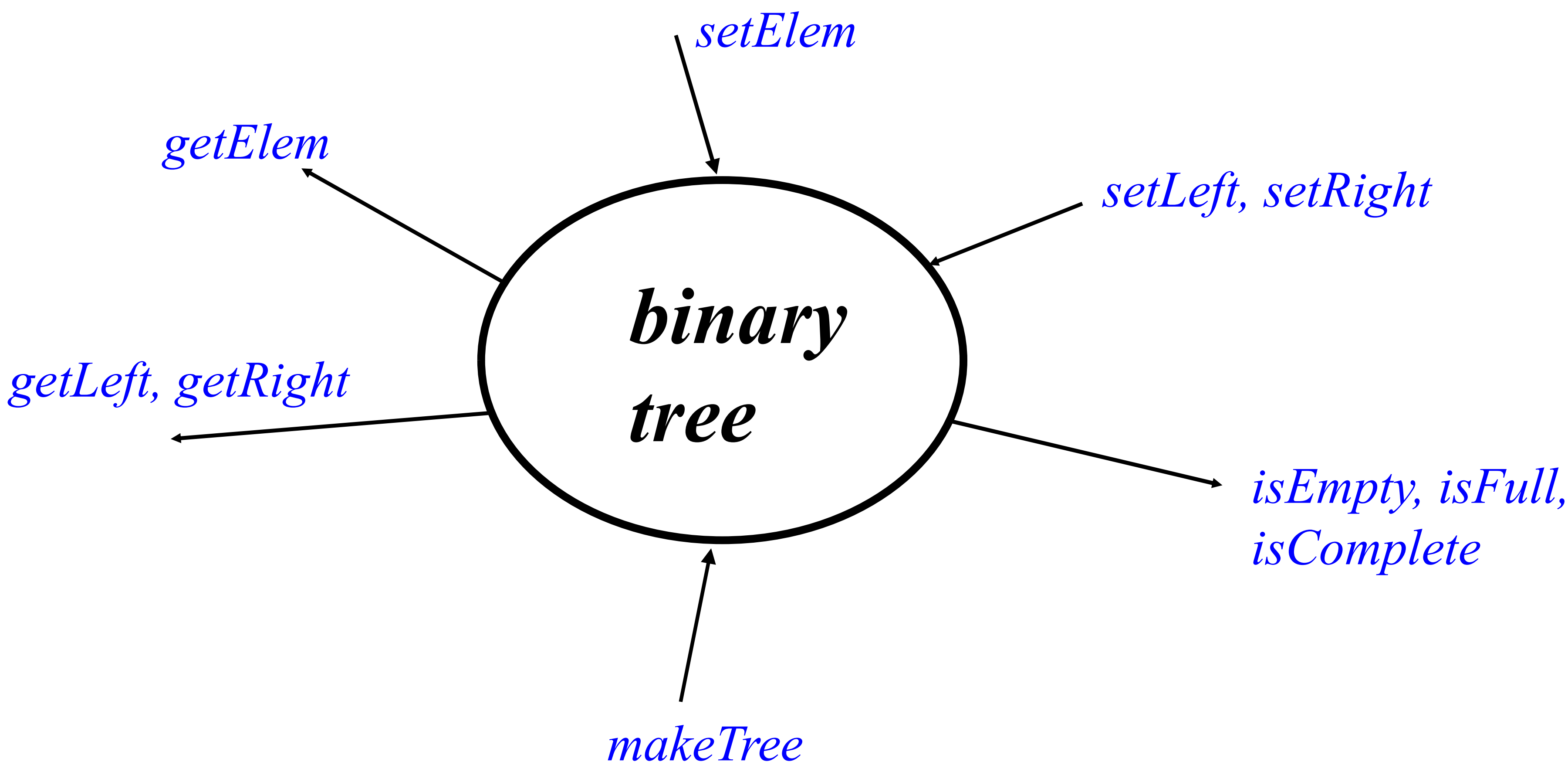
- The minimum height of a binary tree with n nodes is log(n+1)

By property (II), $n \leq 2^h-1$

Thus, $2^h \geq n+1$

That is, $h \geq \log_2 (n+1)$

Binary Tree ADT

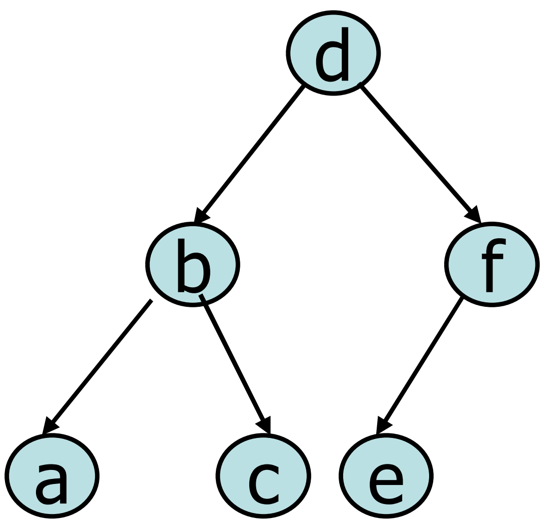


Representation of a Binary Tree

- An array-based representation
- A reference-based representation

An array-based representation

–1: empty tree



nodeNum	item	leftChild	rightChild
0	d	1	2
1	b	3	4
2	f	5	-1
3	a	-1	-1
4	c	-1	-1
5	e	-1	-1
6	?	?	?
7	?	?	?
8	?	?	?
9	?	?	?
...	.....	.....	....

root

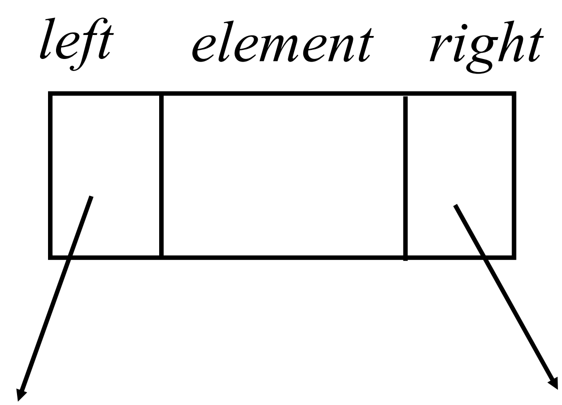
0

free

6

Reference Based Representation

NULL: empty tree

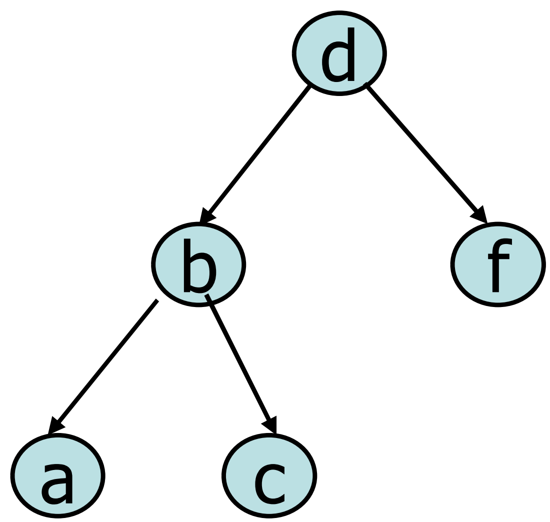


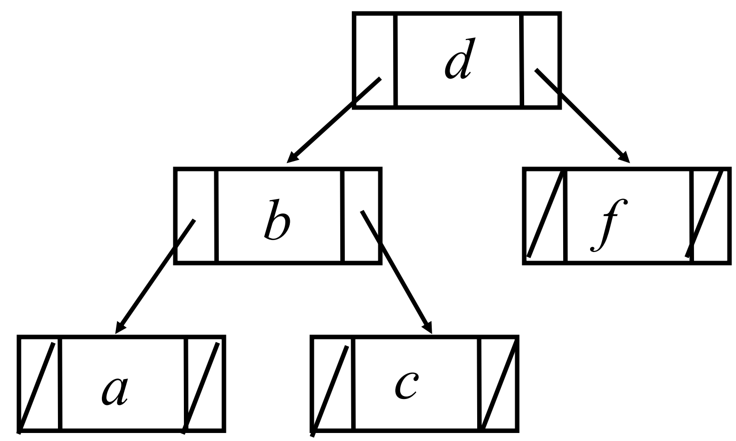
You can code this with a class of three fields:

Object element;

BinaryNode left;

BinaryNode right;





Tree Traversal

- Given a binary tree, we may like to do some operations on all nodes in a binary tree. For example, we may want to double the value in every node in a binary tree.
- To do this, we need a traversal algorithm which visits every node in the binary tree.

Ways to traverse a tree

- There are three main ways to traverse a tree:
  - Pre-order:
    - (1) visit node, (2) recursively visit left subtree, (3) recursively visit right subtree
  - In-order:
    - (1) recursively visit left subtree, (2) visit node, (3) recursively right subtree
  - Post-order:
    - (1) recursively visit left subtree, (2) recursively visit right subtree, (3) visit node
  - Level-order:
    - Traverse the nodes level by level
- In different situations, we use different traversal algorithm.

Examples for expression tree

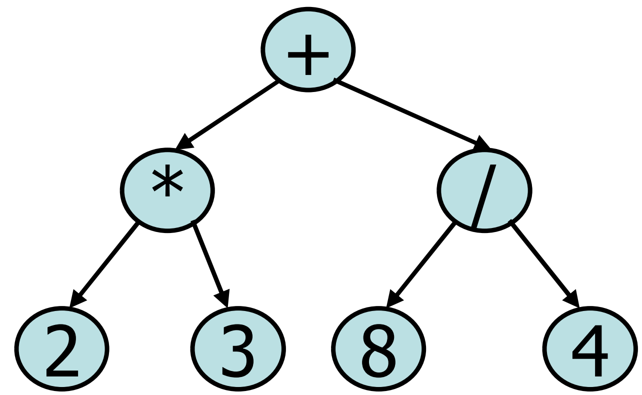
- By pre-order, (prefix)

  + * 2 3 / 8 4
- By in-order, (infix)

  2 * 3 + 8 / 4
- By post-order, (postfix)

  2 3 * 8 4 / +
- By level-order,

  + * / 2 3 8 4
- Note 1: Infix is what we read!
- Note 2: Postfix expression can be computed efficiently using stack



Pre-order

```
Algorithm pre-order(BTree x)
If (x is not empty) {
    print x.getItem();              // you can do other things!
    pre-order(x.getLeftChild());
    pre-order(x.getRightChild());
}
```

Pre-order example

Pre-order(a);

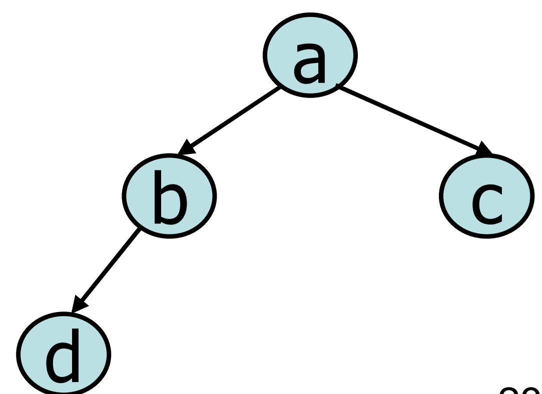
Print a;
Pre-order(b);
Pre-order(c);

Print b;
Pre-order(d);
Pre-order(null);

Print d;
Pre-order(null);
Pre-order(null);

Print c;
Pre-order(null);
Pre-order(null);

a b d c



Time complexity of Pre-order Traversal

- For every node x, we will call pre-order(x) one time, which performs O(1) operations.
- Thus, the total time = O(n).

In-order and post-order

```
Algorithm in-order(BTree x)
If (x is not empty) {
    in-order(x.getLeftChild());
    print x.getItem(); // you can do other things!
    in-order(x.getRightChild());
}

Algorithm post-order(BTree x)
If (x is not empty) {
    post-order(x.getLeftChild());
    post-order(x.getRightChild());
    print x.getItem(); // you can do other things!
}
```

In-order example

In-order(a);

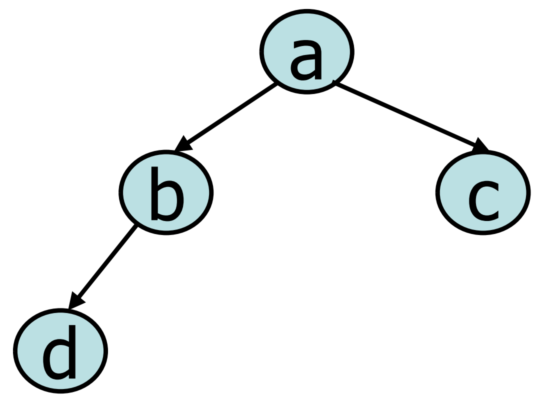
In-order(b);
Print a;
In-order(c);

In-order(d);
Print b;
In-order(null);

In-order(null);
Print d;
In-order(null);

In-order(null);
Print c;
In-order(null);

d b a c



Post-order example

Post-order(a);

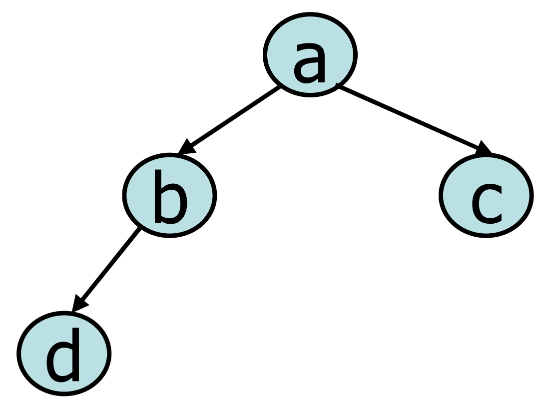
Post-order(b);
Post-order(c);
Print a;

Post-order(d);
Post-order(null);
Print b;

Post-order(null);
Post-order(null);
Print d;

Post-order(null);
Print c;
Post-order(null);

d b c a



Time complexity for in-order and post-order

- Similar to pre-order traversal, the time complexity is $O(n)$.

Level-order

- Level-order traversal requires a queue!

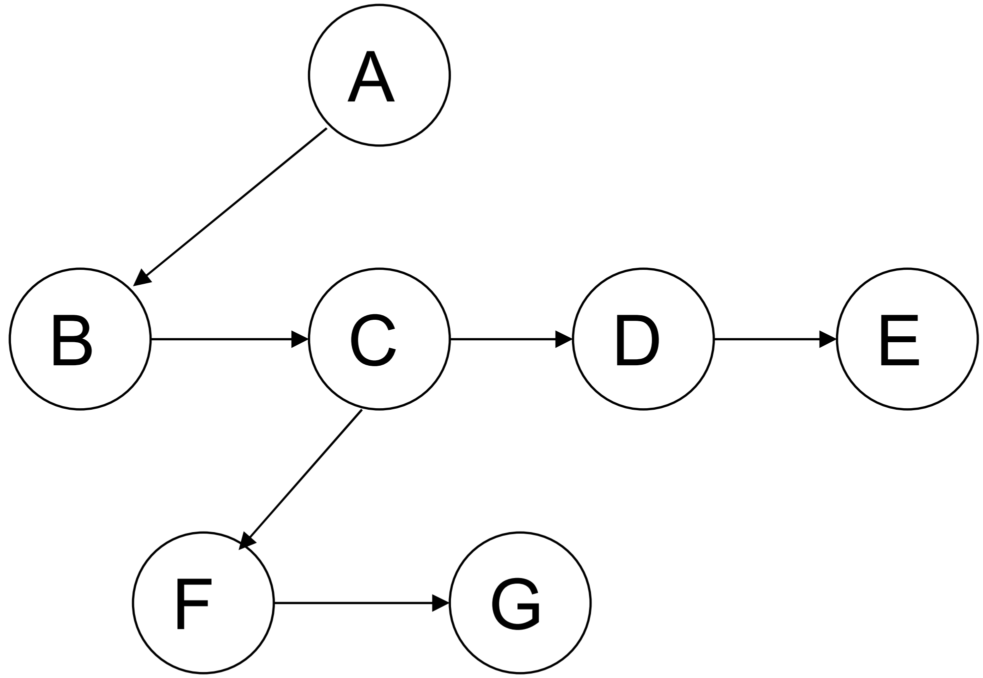
```
Algorithm level-order(BTree t)
 Queue Q = new Queue();
 BTree n;

 Q.enqueue(t);   // insert pointer t into Q

 while (! Q.empty()){
   n = Q.dequeue(); //remove next node from the front of Q

   if (!n.isEmpty()){
       print n.getItem();      // you can do other things
       Q.enqueue(n.getLeft());  // enqueue left subtree on rear of Q
       Q.enqueue(n.getRight()); // enqueue right subtree on rear of Q
   };
 };
```

Time complexity of Level-order traversal

- Each node will enqueue and dequeue one time.
- For each node dequeued, it only does one print operation!
- Thus, the time complexity is O(n).

General tree implementation

```
struct TreeNode
{
    Object      element
    TreeNode *firstChild
    TreeNode *nextsibling
}
```



because we do not know how many children a node has in advance.

- Traversing a general tree is similar to traversing a binary tree

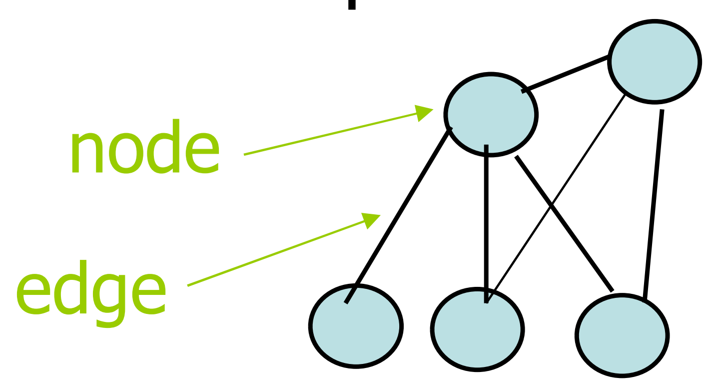
Summary

- We have discussed
  - the tree data-structure.
  - Binary tree vs general tree
  - Binary tree ADT
    - Can be implemented using arrays or references
  - Tree traversal
    - Pre-order, in-order, post-order, and level-order

Graphs

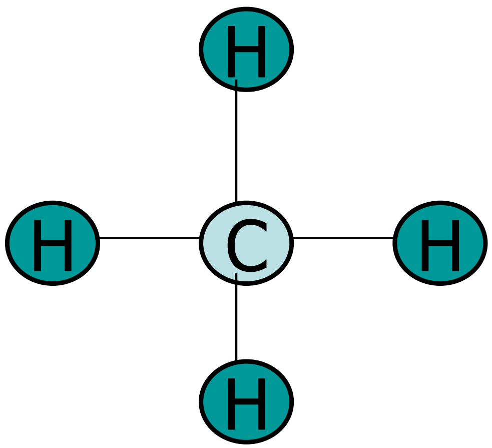
What is a graph?

- Graphs represent the relationships among data items
- A graph G consists of
  - a set V of nodes (vertices)
  - a set E of edges: each edge connects two nodes
- Each node represents an item
- Each edge represents the relationship between two items

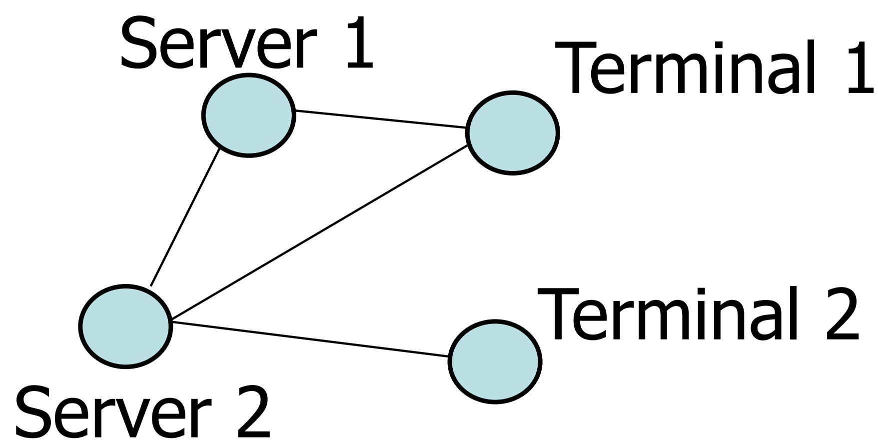


Examples of graphs

Molecular Structure



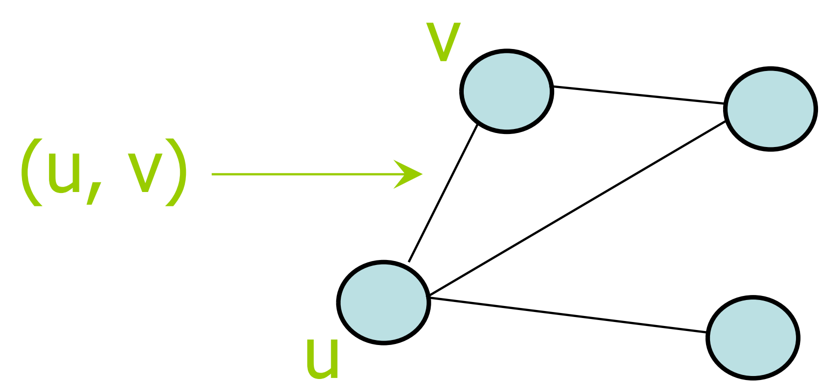
Computer Network



Other examples: electrical and communication networks, airline routes, flow chart, graphs for planning projects

Formal Definition of graph

- The set of nodes is denoted as V
- For any nodes u and v, if u and v are connected by an edge, such edge is denoted as (u, v)

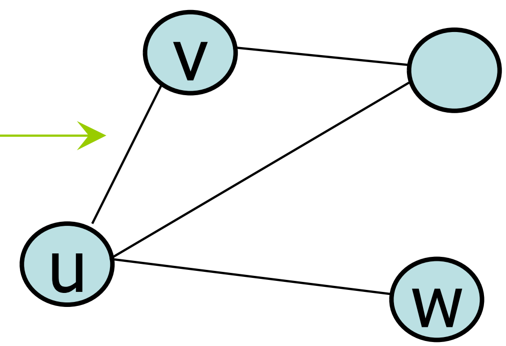


- The set of edges is denoted as E
- A graph G is defined as a pair (V, E)

Adjacent

- Two nodes u and v are said to be adjacent if $(u, v) \in E$

(u, v)

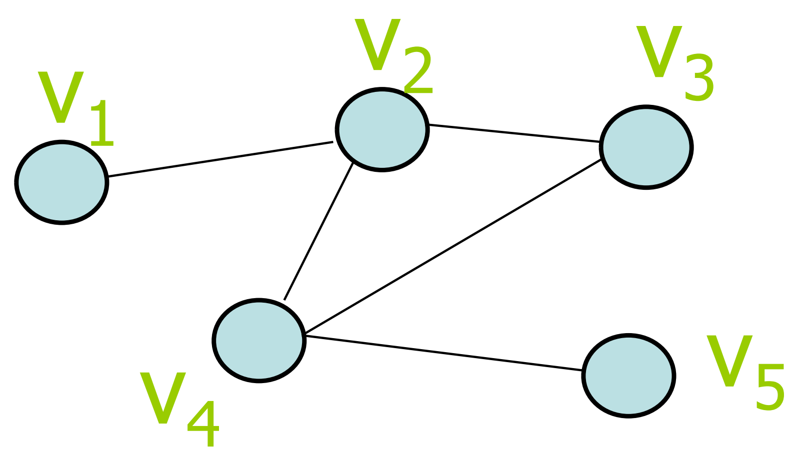


u and v are adjacent
v and w are not adjacent

Path and simple path

- A path from v_1 to v_k is a sequence of nodes $v_1, v_2, \ldots, v_k$ that are connected by edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$
- A path is called a simple path if every node appears at most once.

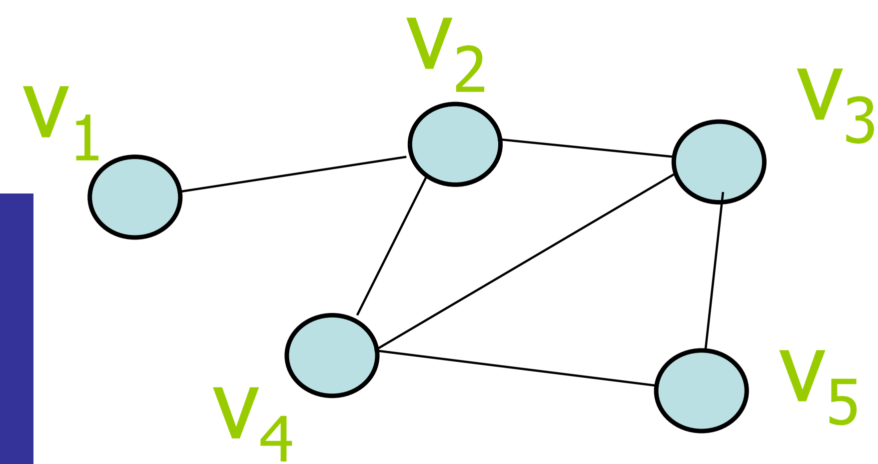
- $v_{2,} v_{3,} v_{4,} v_{2,} v_1$ is a path
- $v_{2,} v_{3,} v_{4,} v_5$ is a path, also it is a simple path



Cycle and simple cycle

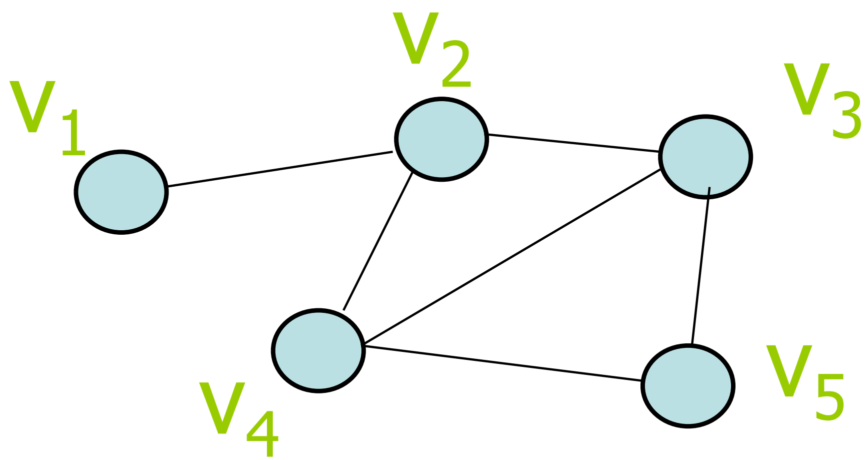
- A cycle is a path that begins and ends at the same node
- A simple cycle is a cycle if every node appears at most once, except for the first and the last nodes

- $v_2, v_3, v_4, v_5, v_3, v_2$ is a cycle
- v_2, v_3, v_4, v_2 is a cycle, it is also a simple cycle



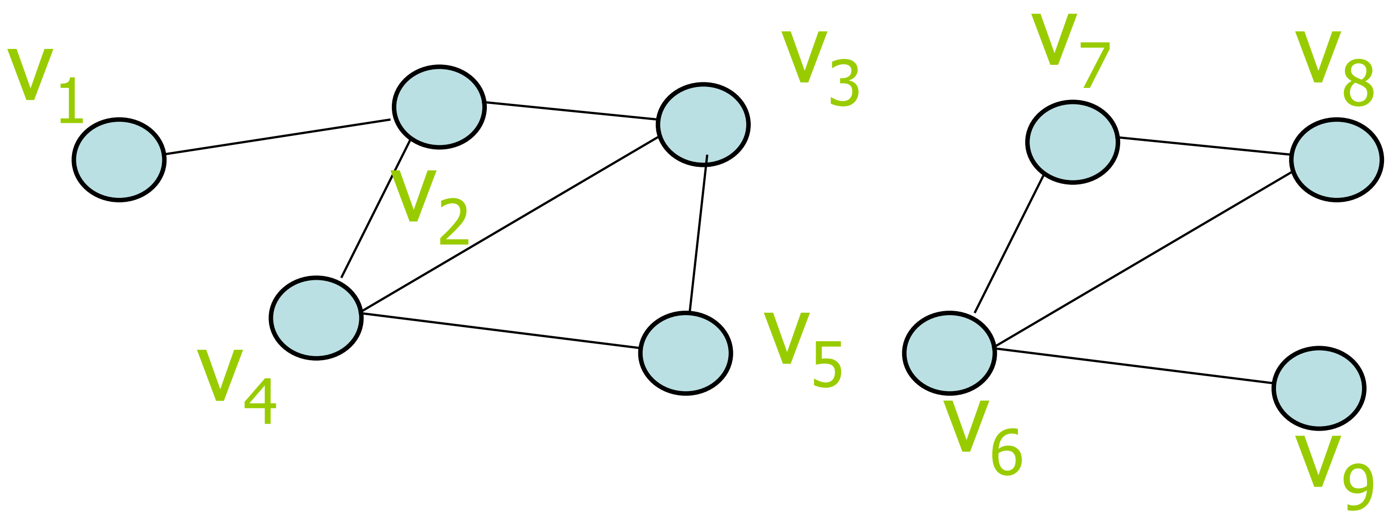
Connected graph

- A graph G is connected if there exists path between every pair of distinct nodes; otherwise, it is disconnected



This is a connected graph because there exists path between every pair of nodes

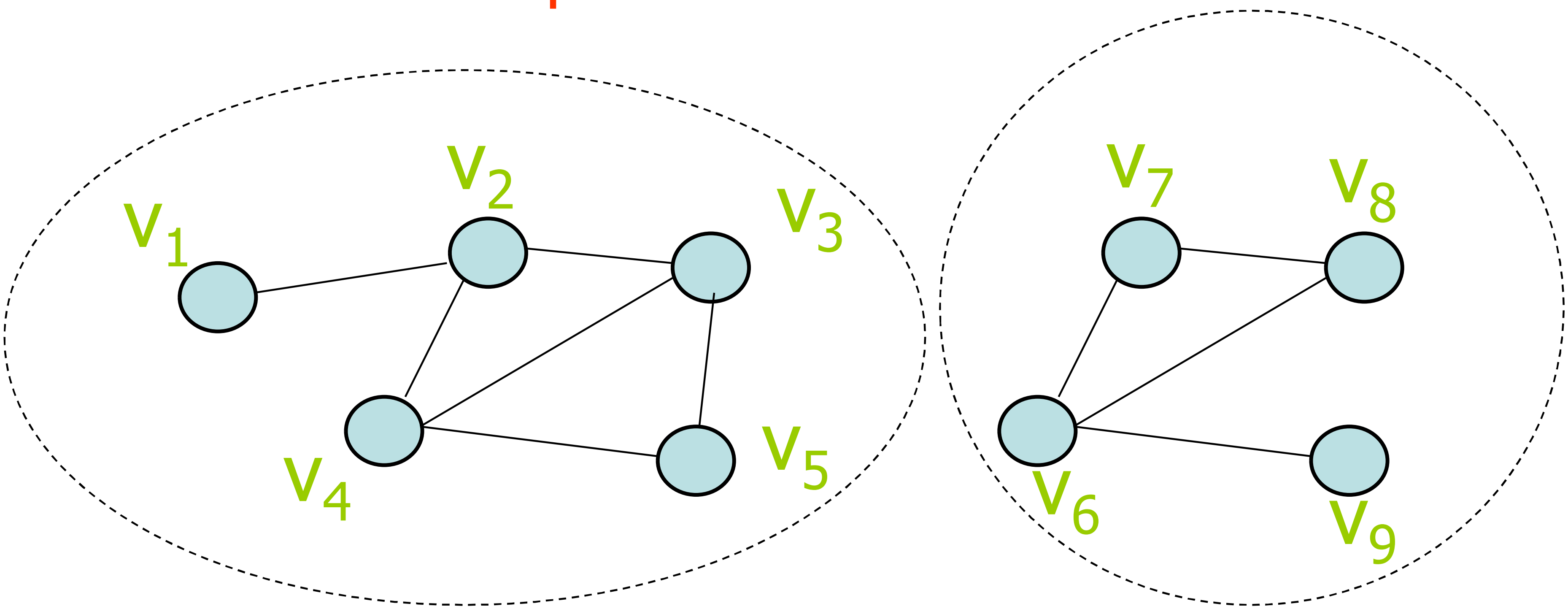
Example of disconnected graph



This is a disconnected graph because there does not exist path between some pair of nodes, says, v_1 and v_7

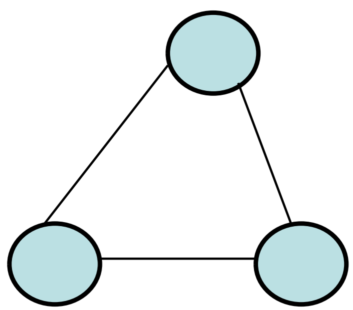
Connected component

- If a graph is disconnect, it can be partitioned into a number of graphs such that each of them is connected. Each such graph is called a connected component.

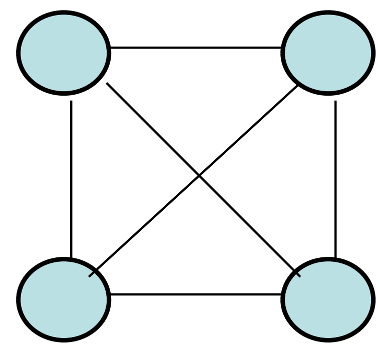


Complete graph

- A graph is complete if each pair of distinct nodes has an edge



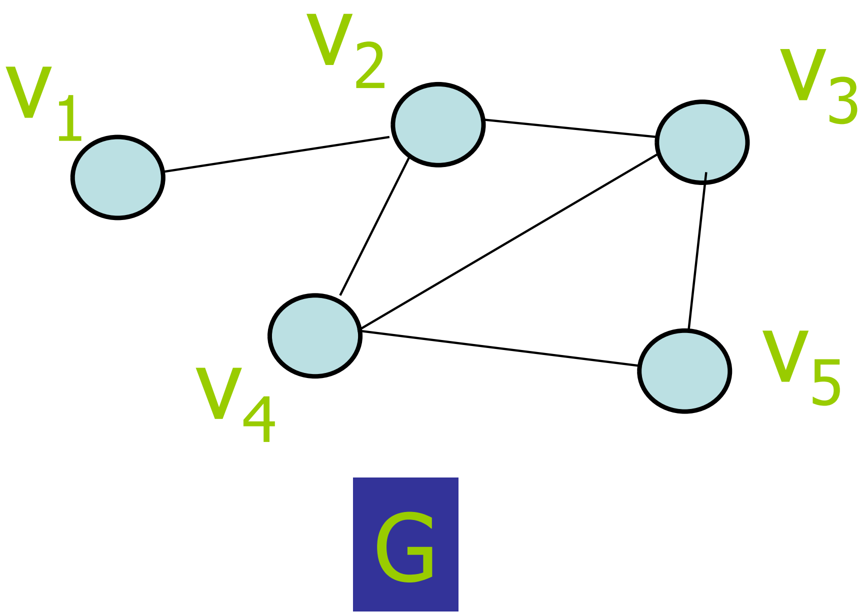
Complete graph with 3 nodes

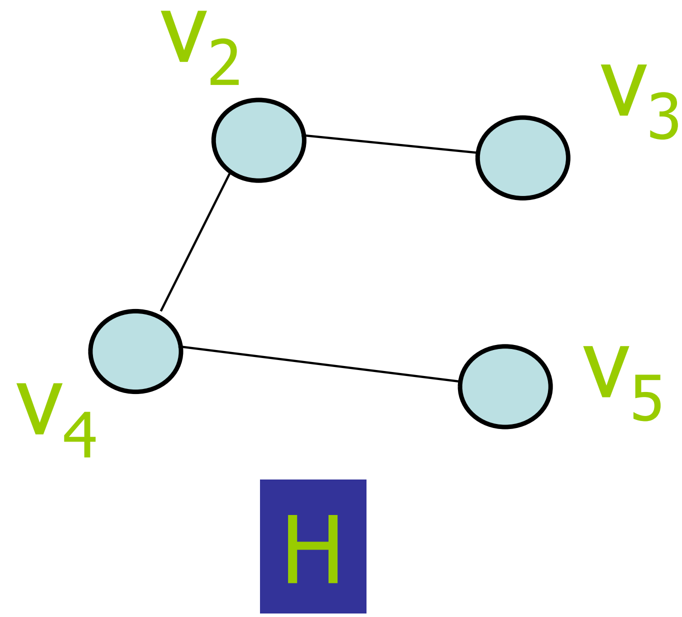


Complete graph with 4 nodes

Subgraph

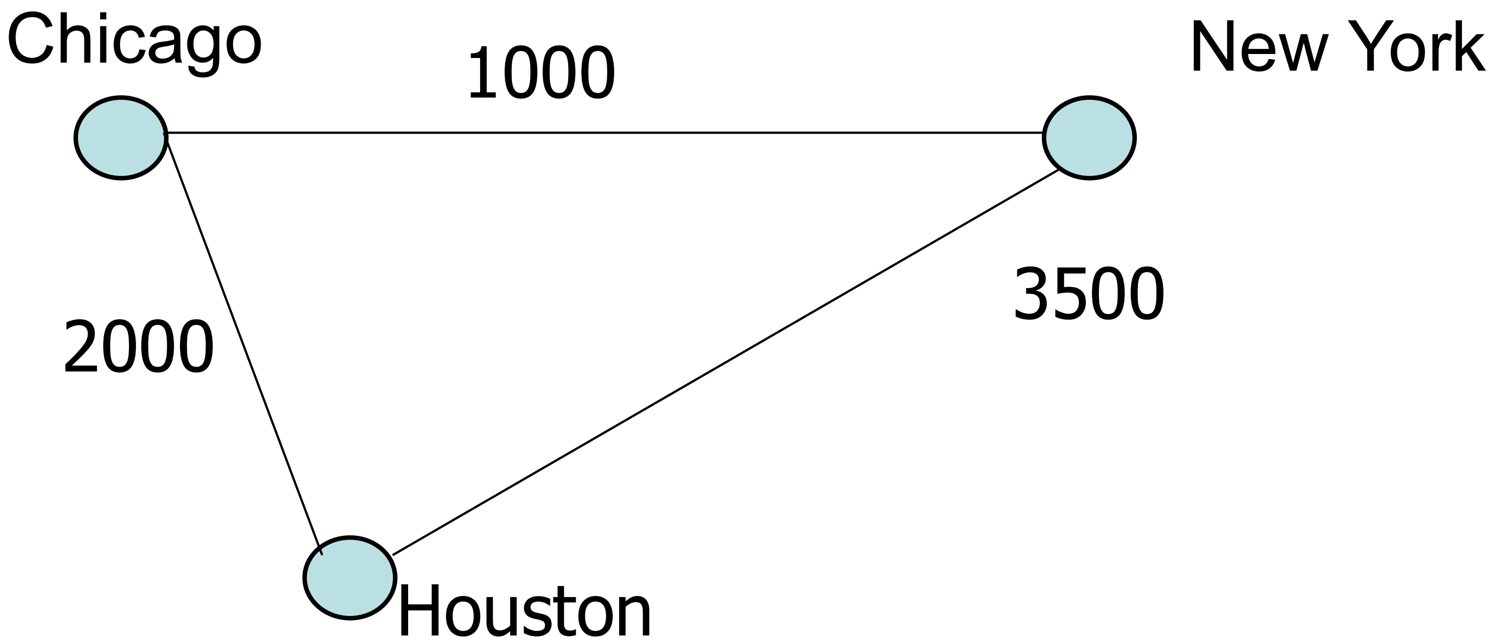
- A subgraph of a graph $G = (V, E)$ is a graph $H = (U, F)$ such that $U \subseteq V$ and $F \subseteq E$.





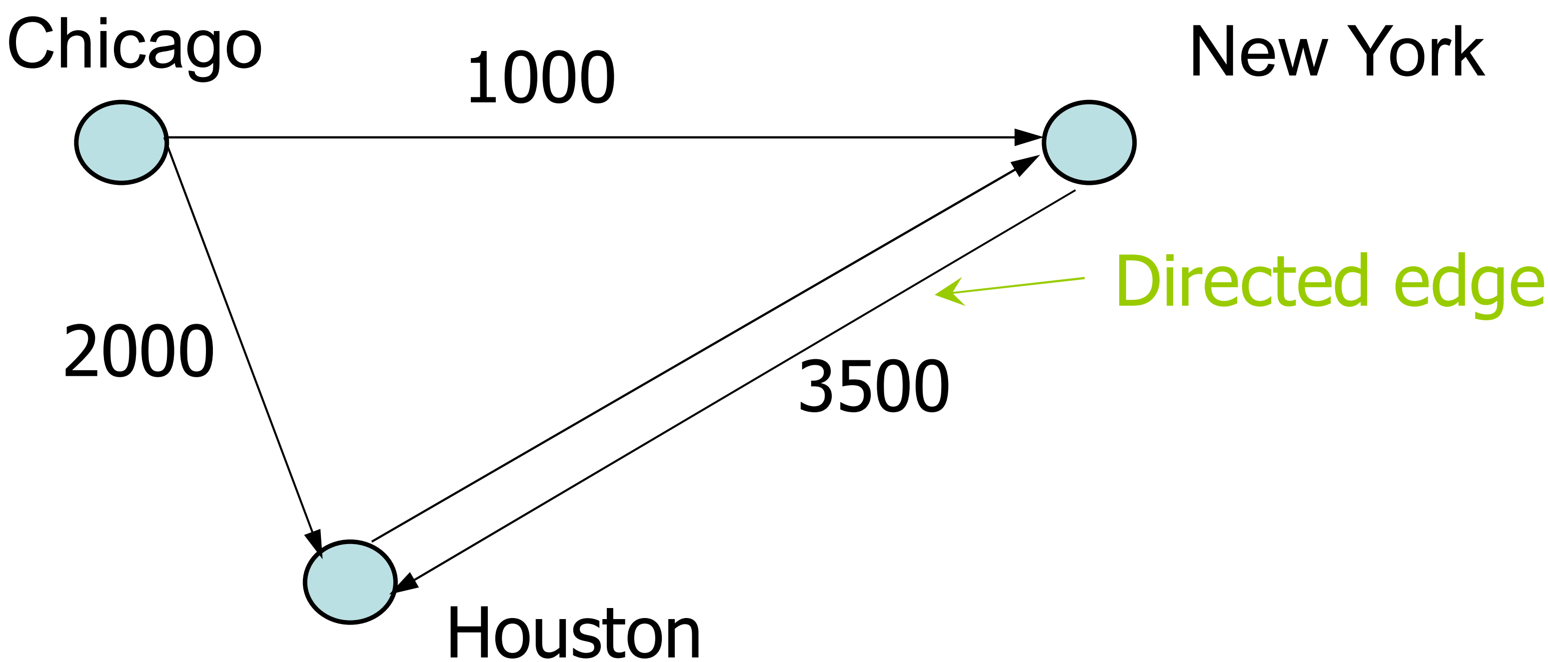
Weighted graph

- If each edge in G is assigned a weight, it is called a weighted graph

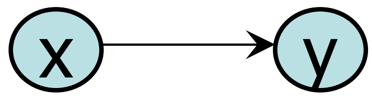


Directed graph (digraph)

- All previous graphs are undirected graph
- If each edge in E has a direction, it is called a directed edge
- A directed graph is a graph where every edges is a directed edge



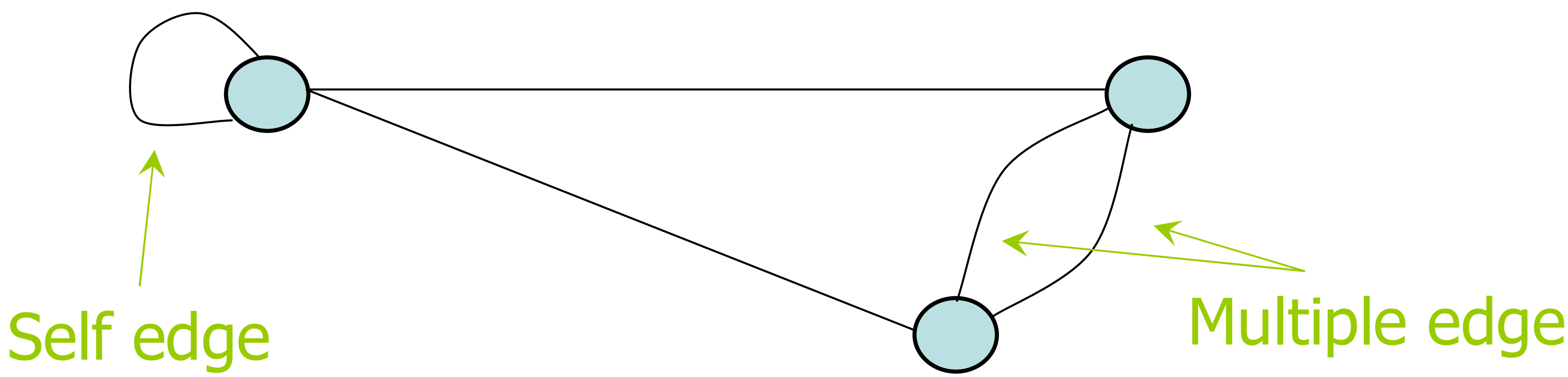
More on directed graph



- If (x, y) is a directed edge, we say
  - y is adjacent to x
  - y is successor of x
  - x is predecessor of y
- In a directed graph, directed path, directed cycle can be defined similarly

Multigraph

- A graph cannot have duplicate edges.
- Multigraph allows multiple edges and self edge (or loop).



Property of graph

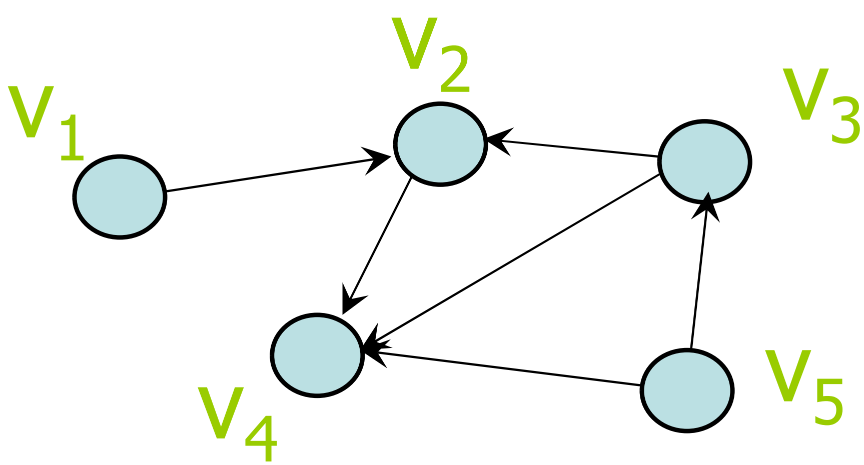
- A undirected graph that is connected and has no cycle is a tree.
- A tree with n nodes have exactly n-1 edges.
- A connected undirected graph with n nodes must have at least n-1 edges.

Implementing Graph

- Adjacency matrix
  - Represent a graph using a two-dimensional array
- Adjacency list
  - Represent a graph using n linked lists where n is the number of vertices

Adjacency matrix for directed graph

$$\text{Matrix}[i][j] = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{if } (v_i, v_j) \notin E \end{cases}$$

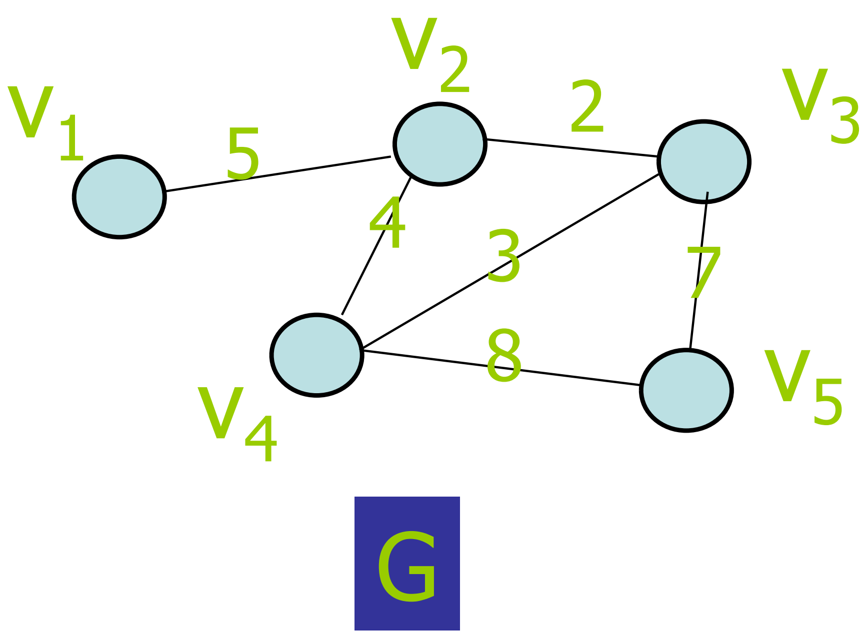


G

		1	2	3	4	5
		v_1	v_2	v_3	v_4	v_5
1	v_1	0	1	0	0	0
2	v_2	0	0	0	1	0
3	v_3	0	1	0	1	0
4	v_4	0	0	0	0	0
5	v_5	0	0	1	1	0

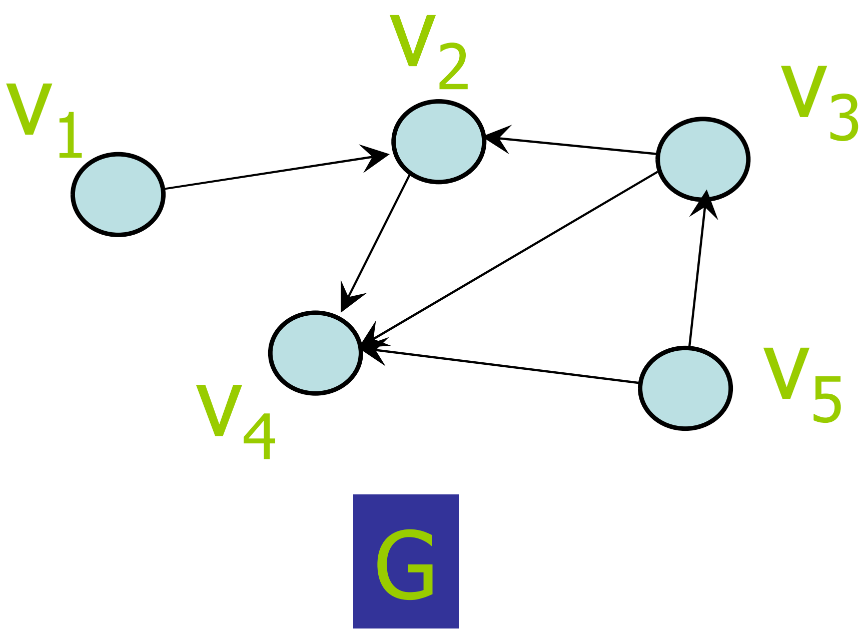
Adjacency matrix for weighted undirected graph

$$\text{Matrix}[i][j] = \begin{cases} w(v_i, v_j) & \text{if } (v_i, v_j) \in E \text{ or } (v_j, v_i) \in E \\ \infty & \text{otherwise} \end{cases}$$



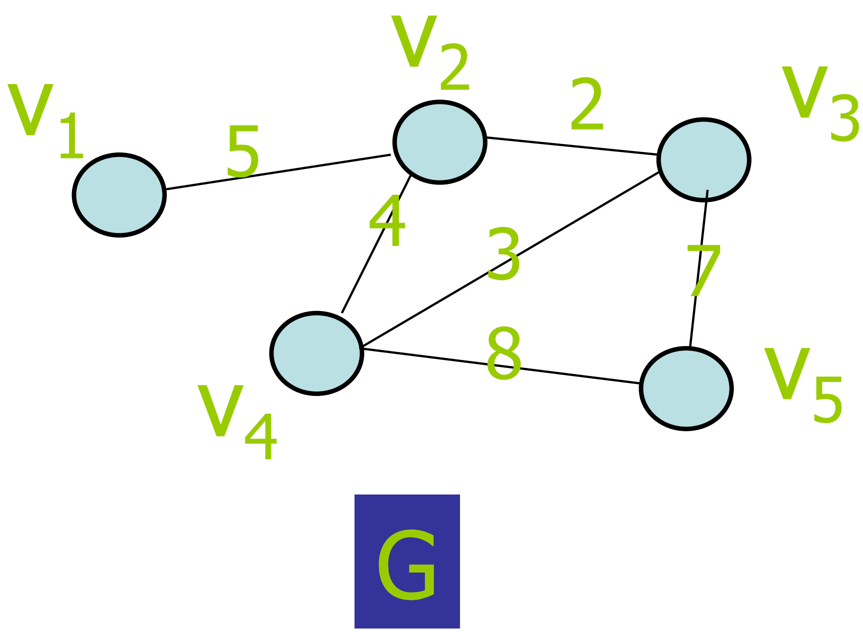
		1	2	3	4	5
		v_1	v_2	v_3	v_4	v_5
1	v_1	∞	5	∞	∞	∞
2	v_2	5	∞	2	4	∞
3	v_3	0	2	∞	3	7
4	v_4	∞	4	3	∞	8
5	v_5	∞	∞	7	8	∞

Adjacency list for directed graph



1	v_1	$\to v_2$
2	v_2	$\to v_4$
3	v_3	$\to v_2 \to v_4$
4	v_4	
5	v_5	$\to v_3 \to v_4$

Adjacency list for weighted undirected graph



1	v_1	$\rightarrow$	$v_2(5)$					
2	v_2	$\rightarrow$	$v_1(5)$	$\rightarrow$	$v_3(2)$	$\rightarrow$	$v_4(4)$
3	v_3	$\rightarrow$	$v_2(2)$	$\rightarrow$	$v_4(3)$	$\rightarrow$	$v_5(7)$
4	v_4	$\rightarrow$	$v_2(4)$	$\rightarrow$	$v_3(3)$	$\rightarrow$	$v_5(8)$
5	v_5	$\rightarrow$	$v_3(7)$	$\rightarrow$	$v_4(8)$		

Pros and Cons

- Adjacency matrix
  - Allows us to determine whether there is an edge from node i to node j in O(1) time
- Adjacency list
  - Allows us to find all nodes adjacent to a given node j efficiently
  - If the graph is sparse, adjacency list requires less space

Problems related to Graph

- Graph Traversal
- Topological Sort
- Spanning Tree
- Minimum Spanning Tree
- Shortest Path

Graph Traversal Algorithm

- To traverse a tree, we use tree traversal algorithms like pre-order, in-order, and post-order to visit all the nodes in a tree
- Similarly, graph traversal algorithm tries to visit all the nodes it can reach.
- If a graph is disconnected, a graph traversal that begins at a node v will visit only a subset of nodes, that is, the connected component containing v.

Two basic traversal algorithms

- Two basic graph traversal algorithms:
  - Depth-first-search (DFS)
    - After visit node v, DFS strategy proceeds along a path from v as deeply into the graph as possible before backing up
  - Breadth-first-search (BFS)
    - After visit node v, BFS strategy visits every node adjacent to v before visiting any other nodes

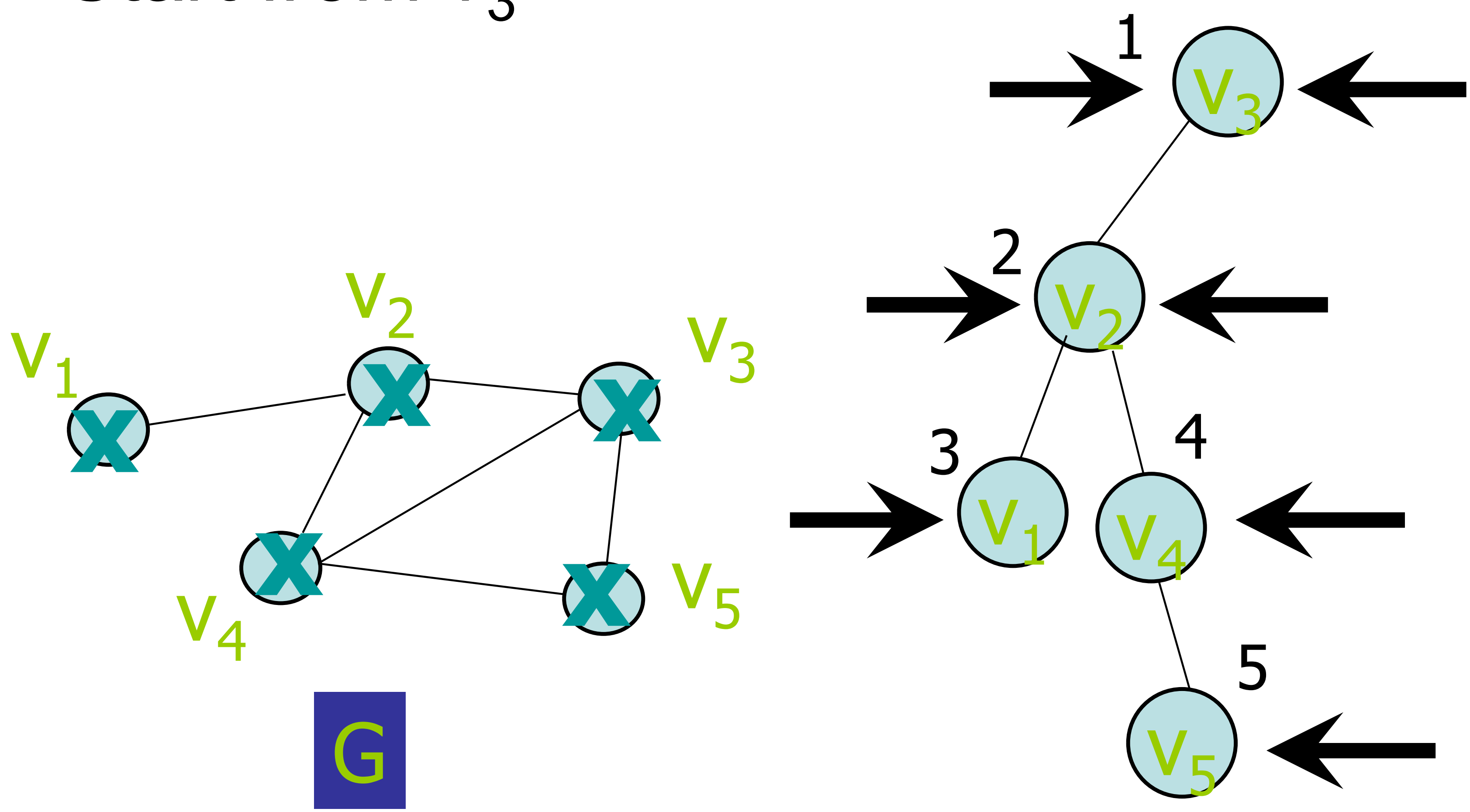
Depth-first search (DFS)

- DFS strategy looks similar to pre-order. From a given node v, it first visits itself. Then, recursively visit its unvisited neighbours one by one.
- DFS can be defined recursively as follows.

```
Algorithm dfs(v)
print v; // you can do other things!
mark v as visited;
for (each unvisited node u adjacent to v)
    dfs(u);
```

DFS example

- Start from v_3

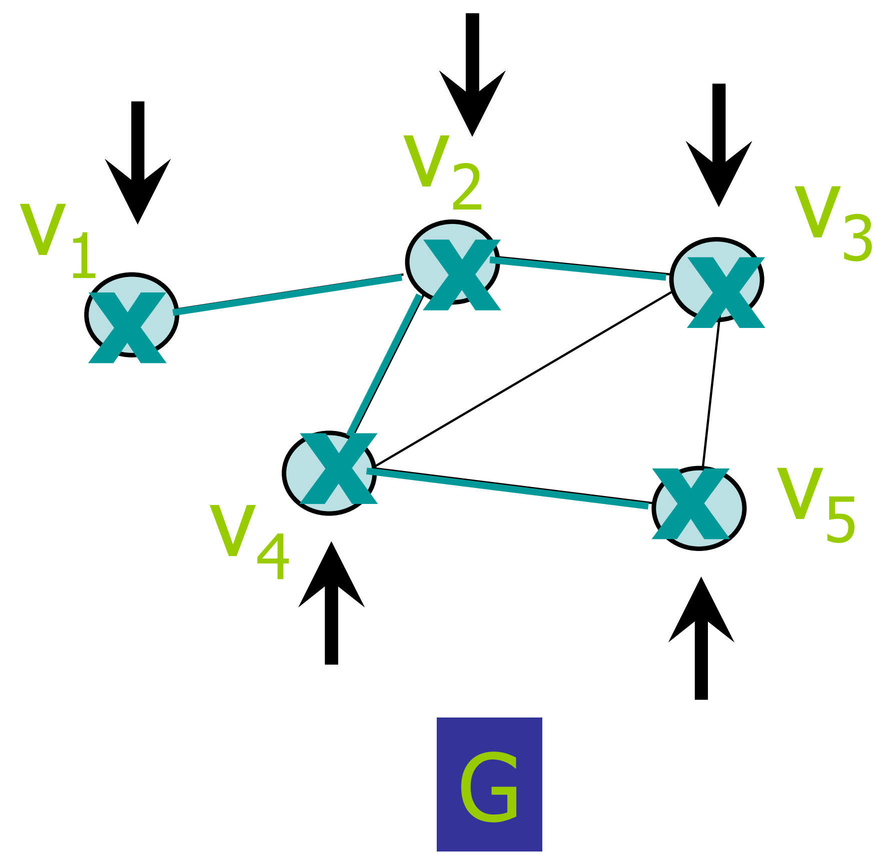


Non-recursive version of DFS algorithm

```
Algorithm dfs(v)
s.createStack();
s.push(v);
mark v as visited;
while (!s.isEmpty()) {
    let x be the node on the top of the stack s;
    if (no unvisited nodes are adjacent to x)
            s.pop(); // blacktrack
    else {
            select an unvisited node u adjacent to x;
            s.push(u);
            mark u as visited;
    }
}
```

Non-recursive DFS example

visit	stack
v_3	v_3
v_2	v_3, v_2
v_1	v_3, v_2, v_1
backtrack	v_3, v_2
v_4	v_3, v_2, v_4
v_5	v_3, v_2, v_4 , v_5
backtrack	v_3, v_2, v_4
backtrack	v_3, v_2
backtrack	v_3
backtrack	empty



Breadth-first search (BFS)

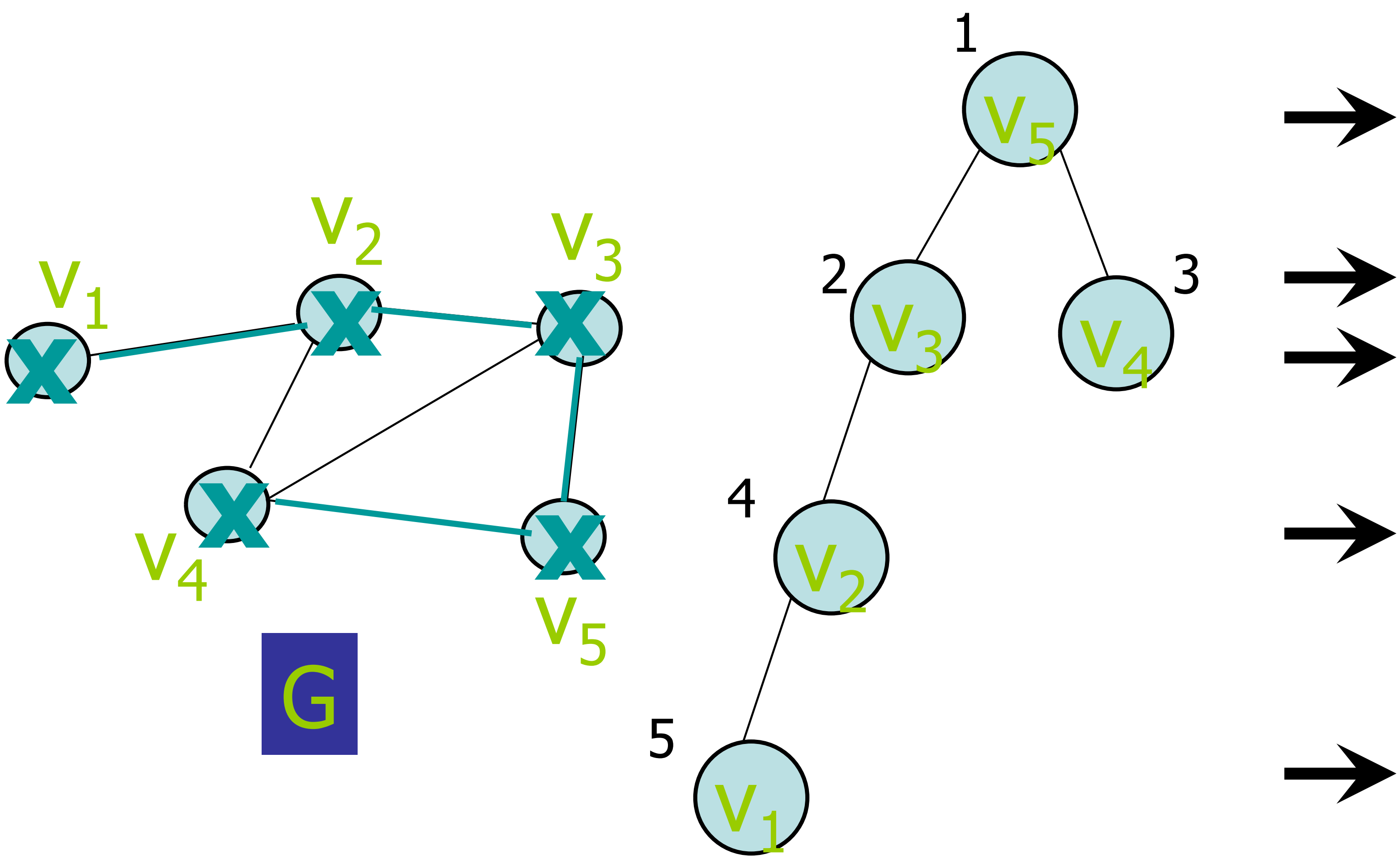
- BFS strategy looks similar to level-order. From a given node v, it first visits itself. Then, it visits every node adjacent to v before visiting any other nodes.
  - 1. Visit v
  - 2. Visit all v’s neigbours
  - 3. Visit all v’s neighbours’ neighbours
  - …
- Similar to level-order, BFS is based on a queue.

Algorithm for BFS

```
Algorithm bfs(v)
q.createQueue();
q.enqueue(v);
mark v as visited;
while(!q.isEmpty()) {
    w = q.dequeue();
    for (each unvisited node u adjacent to w) {
        q.enqueue(u);
        mark u as visited;
    }
}
```

BFS example

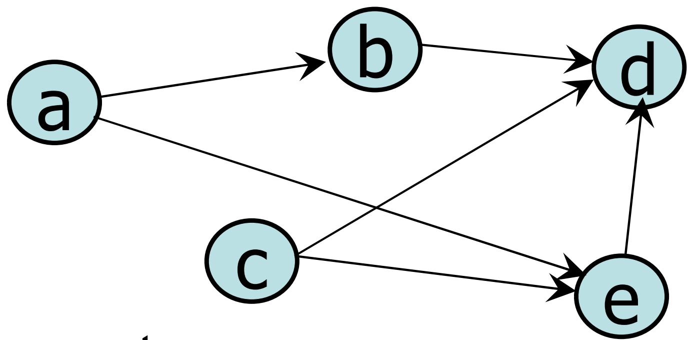
- Start from v_5



Visit	Queue (front to back)
v_5	v_5
	empty
v_3	v_3
v_4	v_3, v_4
	v_4
v_2	v_4, v_2
	v_2
	empty
v_1	v_1
	empty

Topological order

- Consider the prerequisite structure for courses:



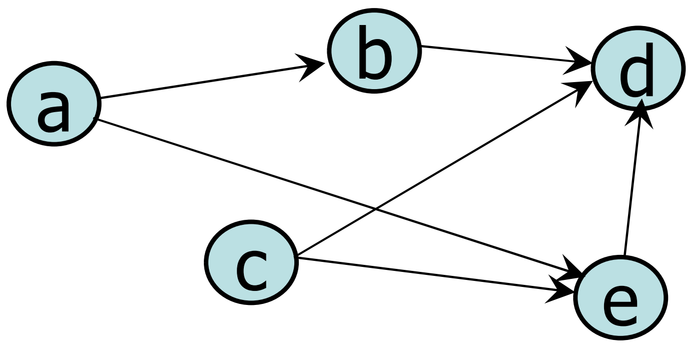
- Each node x represents a course x
- (x, y) represents that course x is a prerequisite to course y
- Note that this graph should be a directed graph without cycles (called a directed acyclic graph).
- A linear order to take all 5 courses while satisfying all prerequisites is called a topological order.
- E.g.
  - a, c, b, e, d
  - c, a, b, e, d

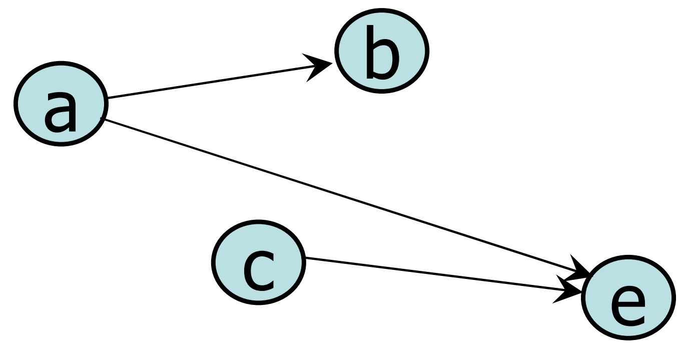
Topological sort

- Arranging all nodes in the graph in a topological order

```
Algorithm topSort
n = |V|;
for i = 1 to n {
   select a node v that has no successor;
   aList.add(1, v);
   delete node v and its edges from the graph;
}
return aList;
```

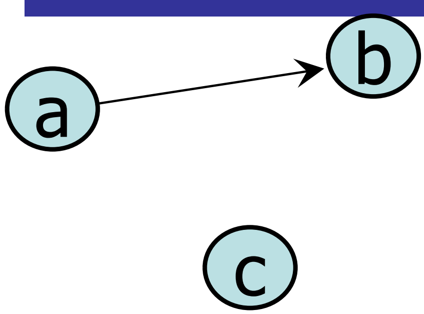
Example

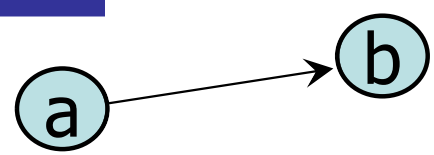


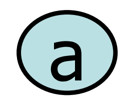


1. d has no successor! Choose d!

2. Both b and e have no successor! Choose e!







3. Both b and c have no successor! Choose c!

4. Only b has no successor! Choose b!

5. Choose a!
The topological order is **a,b,c,e,d**

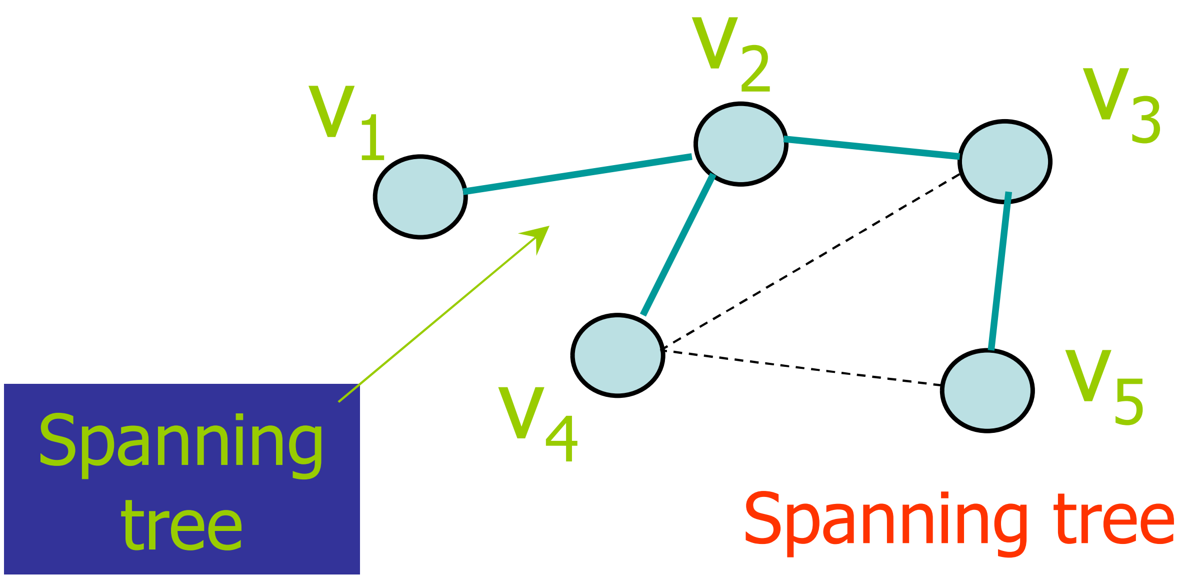
Topological sort algorithm 2

- This algorithm is based on DFS

```
Algorithm topSort2
s.createStack();
for (all nodes v in the graph) {
    if (v has no predecessors) {
        s.push(v);
        mark v as visited;
    }
}
while (!s.isEmpty()) {
    let x be the node on the top of the stack s;
    if (no unvisited nodes are adjacent to x) { // i.e. x has no unvisited successor
        aList.add(1, x);
        s.pop(); // blacktrack
    } else {
        select an unvisited node u adjacent to x;
        s.push(u);
        mark u as visited;
    }
}
return aList;
```

Spanning Tree

- Given a connected undirected graph G, a spanning tree of G is a subgraph of G that contains all of G's nodes and enough of its edges to form a tree.



Spanning tree is not unique!

DFS spanning tree

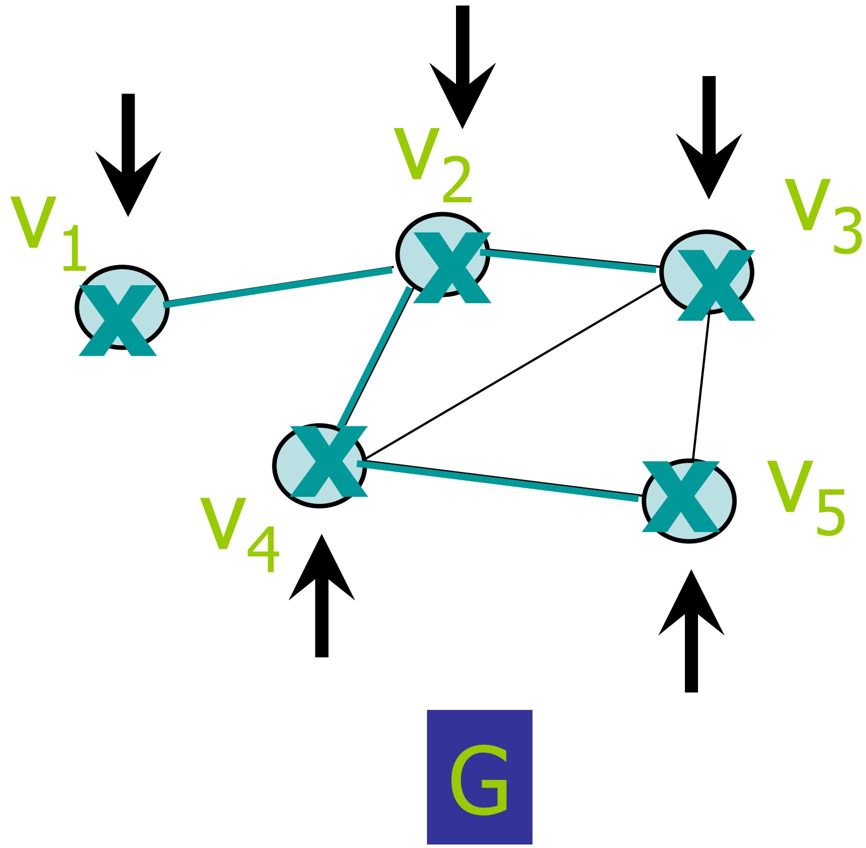
- Generate the spanning tree edge during the DFS traversal.

```
Algorithm dfsSpanningTree(v)
mark v as visited;
for (each unvisited node u adjacent to v) {
   mark the edge from u to v;
   dfsSpanningTree(u);
}
```

- Similar to DFS, the spanning tree edges can be generated based on BFS traversal.

Example of generating spanning tree based on DFS

	stack
v_3	v_3
v_2	v_3, v_2
v_1	v_3, v_2, v_1
backtrack	v_3, v_2
v_4	v_3, v_2, v_4
v_5	v_3, v_2, v_4, v_5
backtrack	v_3, v_2, v_4
backtrack	v_3, v_2
backtrack	v_3
backtrack	empty

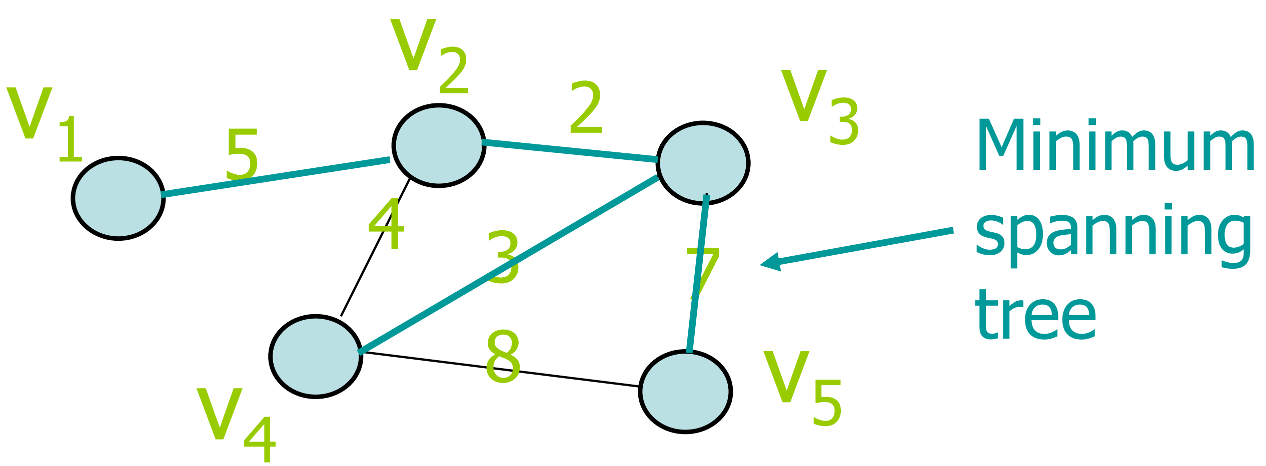


Minimum Spanning Tree

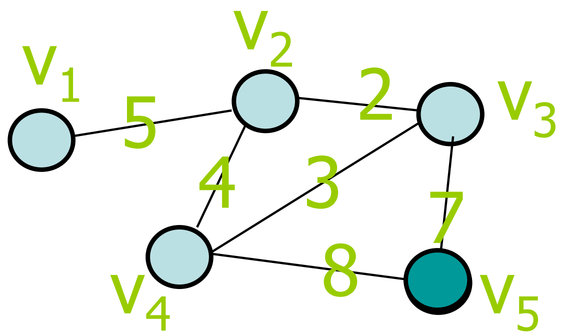
- Consider a connected undirected graph where
  - Each node x represents a country x
  - Each edge (x, y) has a number which measures the cost of placing telephone line between country x and country y
- Problem: connecting all countries while minimizing the total cost
- Solution: find a spanning tree with minimum total weight, that is, minimum spanning tree

Formal definition of minimum spanning tree

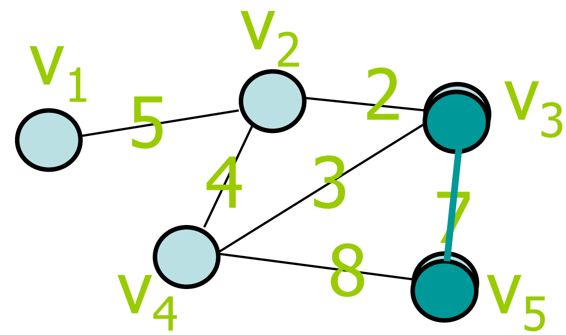
- Given a connected undirected graph G.
- Let T be a spanning tree of G.
- cost(T) = $\sum_{e \in T}$weight(e)
- The minimum spanning tree is a spanning tree T which minimizes cost(T)



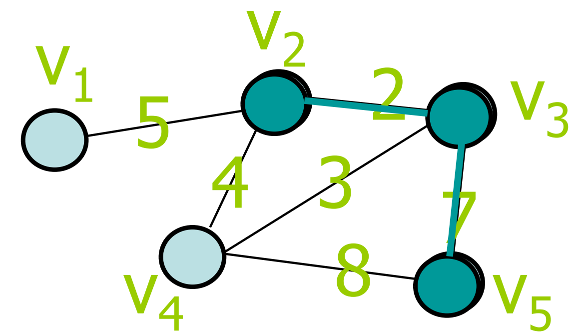
Prim’s algorithm (I)



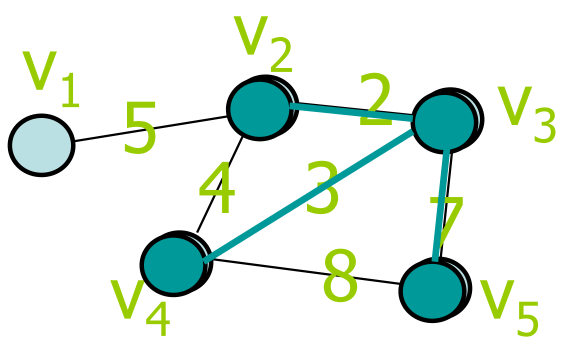
Start from v_5, find the minimum edge attach to v_5

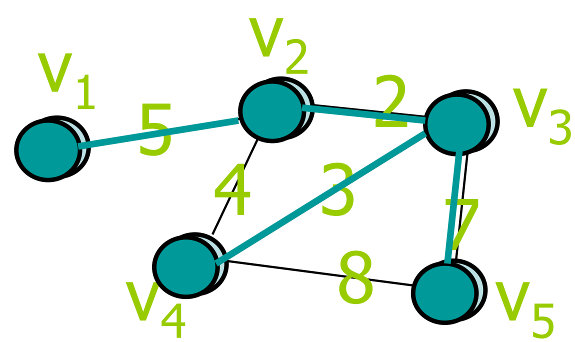


Find the minimum edge attach to v_3 and v_5



Find the minimum edge attach to v_2, v_3 and v_5





Find the minimum edge attach to v_2, v_3 , v_4 and v_5

Prim's algorithm (II)

Algorithm PrimAlgorithm(v)

- Mark node v as visited and include it in the minimum spanning tree;
- while (there are unvisited nodes) {
  - find the minimum edge (v, u) between a visited node v and an unvisited node u;
  - mark u as visited;
  - add both v and (v, u) to the minimum spanning tree;

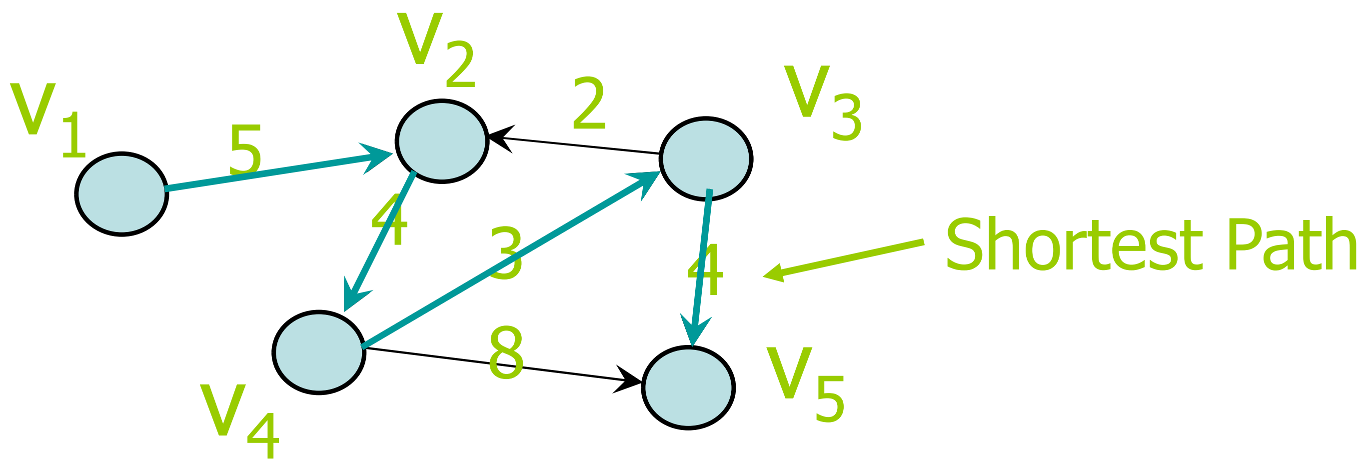
  }

Shortest path

- Consider a weighted directed graph
  - Each node x represents a city x
  - Each edge (x, y) has a number which represent the cost of traveling from city x to city y
- Problem: find the minimum cost to travel from city x to city y
- Solution: find the shortest path from x to y

Formal definition of shortest path

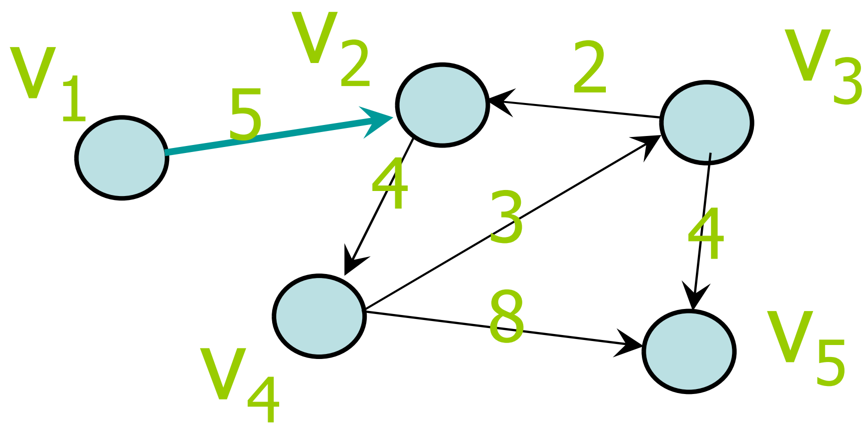
- Given a weighted directed graph G.
- Let P be a path of G from x to y.
- $\text{cost}(P) = \sum_{e \in P} \text{weight}(e)$
- The shortest path is a path P which minimizes cost(P)



Dijkstra's algorithm

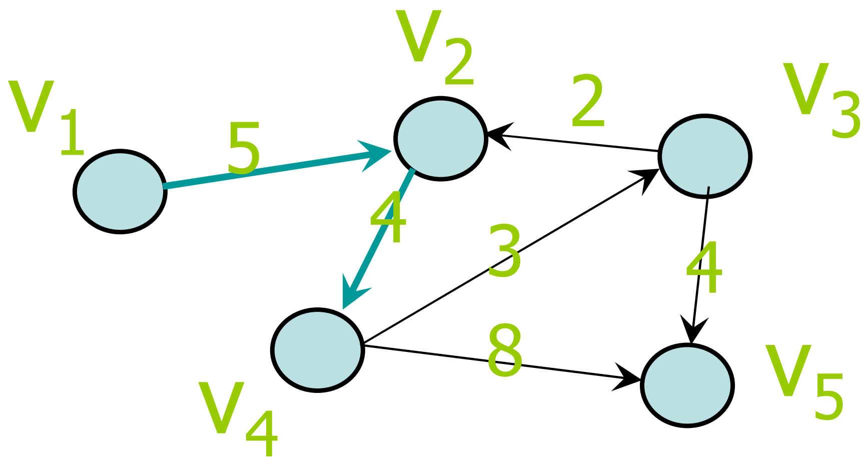
- Consider a graph G, each edge (u, v) has a weight $w(u, v) > 0$.
- Suppose we want to find the shortest path starting from v_1 to any node v_i
- Let VS be a subset of nodes in G
- Let $cost[v_i]$ be the weight of the shortest path from v_1 to v_i that passes through nodes in VS only.

Example for Dijkstra's algorithm



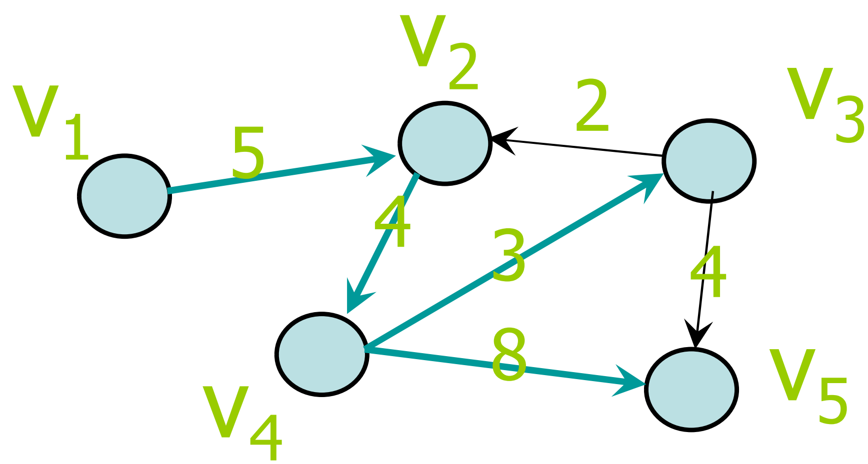
	v	VS	cost[v_1]	cost[v_2]	cost[v_3]	cost[v_4]	cost[v_5]
1		[v_1]	0	5	∞	∞	∞

Example for Dijkstra's algorithm



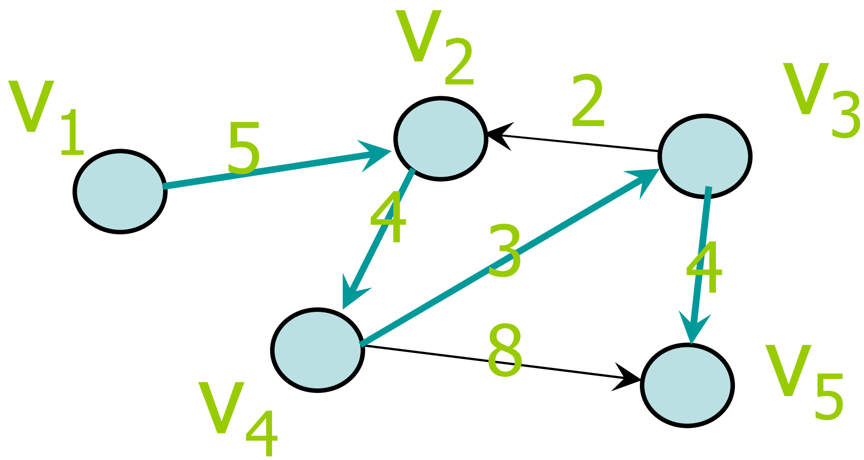
	v	VS	cost[v_1]	cost[v_2]	cost[v_3]	cost[v_4]	cost[v_5]
1		[v_1]	0	5	∞	∞	∞
2	v_2	[v_1, v_2]	0	5	∞	9	∞

Example for Dijkstra's algorithm



	v	VS	cost[v_1]	cost[v_2]	cost[v_3]	cost[v_4]	cost[v_5]
1		[v_1]	0	5	∞	∞	∞
2	v_2	[v_1, v_2]	0	5	∞	9	∞
3	v_4	[v_1, v_2, v_4]	0	5	12	9	17

Example for Dijkstra's algorithm



	v	VS	cost[v_1]	cost[v_2]	cost[v_3]	cost[v_4]	cost[v_5]
1		[v_1]	0	5	∞	∞	∞
2	v_2	[v_1, v_2]	0	5	∞	9	∞
3	v_4	[v_1, v_2, v_4]	0	5	12	9	17
4	v_3	[v_1, v_2, v_4, v_3]	0	5	12	9	16
5	v_5	[v_1, v_2, v_4, v_3, v_5]	0	5	12	9	16

Dijkstra's algorithm

Algorithm shortestPath()

n = number of nodes in the graph;
for i = 1 to n
    cost[v_i] = w(v_1, v_i);
VS = { v_1 };
for step = 2 to n {
    find the smallest cost[v_i] s.t. v_i is not in VS;
    include v_i to VS;
    for (all nodes v_j not in VS) {
        if (cost[v_j] > cost[v_i] + w(v_i, v_j))
            cost[v_j] = cost[v_i] + w(v_i, v_j);
    }
}

Summary

- Graphs can be used to represent many real-life problems.
- There are numerous important graph algorithms.
- We have studied some basic concepts and algorithms.
  - Graph Traversal
  - Topological Sort
  - Spanning Tree
  - Minimum Spanning Tree
  - Shortest Path