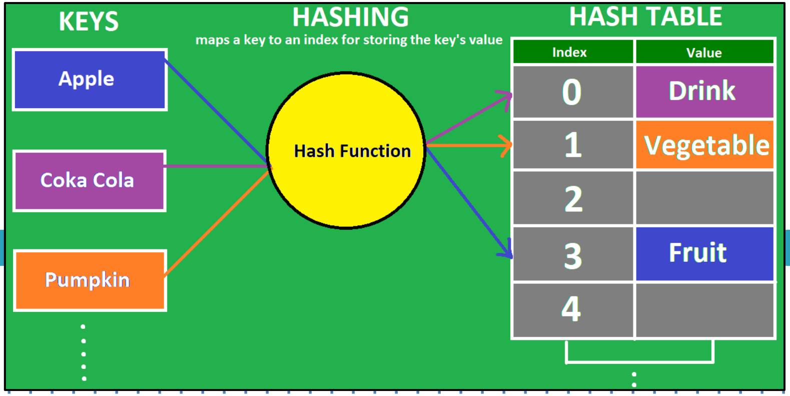
Data Structures and Algorithms

13. Graphs

Tentative Course Outline

1. Introduction and Overview
2. Arrays ✓
3. Simple Sorting
4. Stack and Quene ✓
5. Linkedlist ✓
6. Arraylist ✓
7. Recursion
8. Sorting
9. Binary Trees ✓
10. Red-Black Trees ✓
11. 234 Trees ✓
12. Hashtable ✓
13. Graph
14. Graph

Last Week…



- Hash tables are data structures that can provide very fast insertion, search, and deletion performance.
- Hash tables are array based.
- If there is too little data from the length of the array, there may be a loss of space; if there is too much data, there may be a loss of performance.
- If the array is too full, there may be serious performance loss. Elements can be transferred to a larger array, which is a costly operation.
- It is not possible to search the elements sequentially.

Last Week…

Collision Resolution Techniques

- When **two keys** map to the **same location (index)** in the hash table, collision occurs.
- We try to avoid it, but **number of keys** exceeds table size.
- Hash tables should support collision resolution.

There are two broad ways of collision resolution:

1. **Open Addressing**: Array-based implementation.

    (i) Linear probing (linear search)
    (ii) Quadratic probing (nonlinear search)
    (iii) Double hashing (uses two hash functions)

2. **Separate Chaining**: An array of linked list implementation.

Last Week…

Collision Resolution Techniques

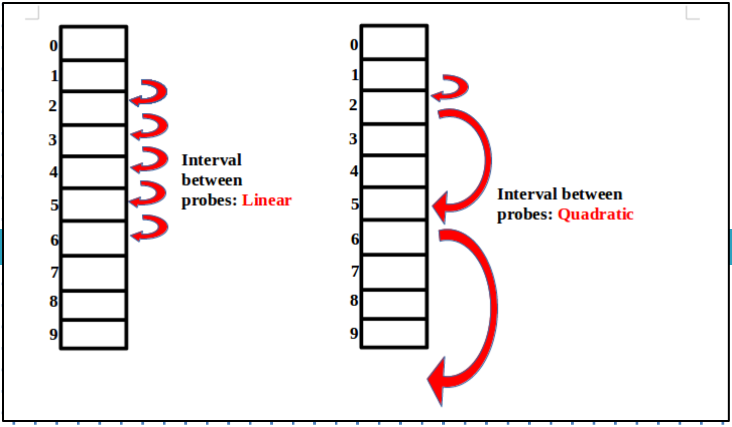
1. Open Adressing

- When a **second element** is assigned to the **same index**, **an empty position** is searched in the same array **for the new element**.
- The new element is placed in the first position found.

This process can be done in three different ways:

- ✓ Linear Probing
- ✓ Quadratic Probing
- ✓ Double Hashing

Last Week…



Intuition: **Quadratic probing** moves increasingly far away. Probes quickly "leave the neighborhood".

Linear Probing

The linear probe function
(h(key) + f(i)) % TableSize

A common technique is linear probing:
f(i) = i

So probe sequence is:
0th probe: **h(key) % TableSize**
1st probe: **(h(key) + 1) % TableSize**
2nd probe: **(h(key) + 2) % TableSize**
3rd probe: **(h(key) + 3) % TableSize**
…
ith probe: **(h(key) + i) % TableSize**

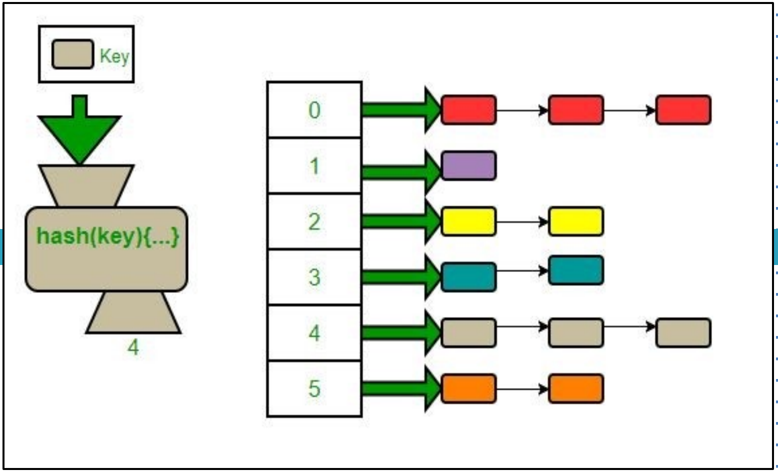
Quadratic Probing

We can avoid primary clustering by changing the probe function
(h(key) + f(i)) % TableSize

A common technique is quadratic probing:
f(i) = i^2

So probe sequence is:
0th probe: **h(key) % TableSize**
1st probe: **(h(key) + 1) % TableSize**
2nd probe: **(h(key) + 4) % TableSize**
3rd probe: **(h(key) + 9) % TableSize**
…
ith probe: **(h(key) + i^2) % TableSize**

Last Week…



Collision Resolution Techniques

2. Closed Adressing (Separate Chaining)

- Chaining allows for **multiple objects** to reside **within the same array location**.
- The array is changed to be an **array of lists** or some other data structure, allowing us to **store multiple items per index.**
- We often use an array of **linked lists**, hence the name "chaining."

Outline

1. Graph Terminology
2. Graph Algorithms

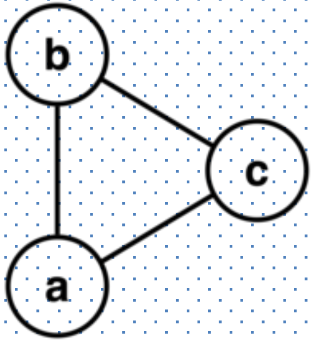
1. Graph Terminology

Graph: a data structure containing

a set of vertices V

a set of edges E, where an edge represents a connection between 2 vertices

edge is a pair (v, w) where v, w in V



Denote graph as G = (V, E)

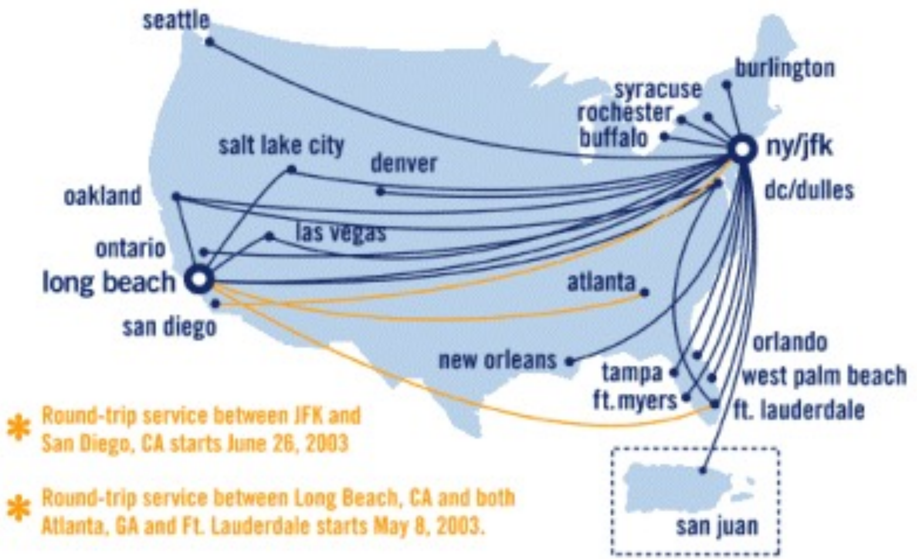
Example:

G = (V,E) where

V = {a, b, c} and E = {(a, b), (b, c), (c, a)}

1. Graph Terminology

Airline Routes

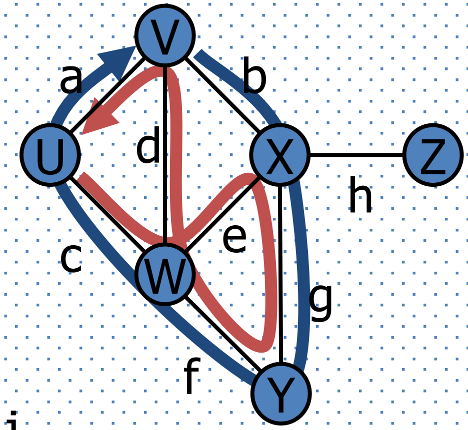


Nodes = cities
Edges = direct flights

1. Graph Terminology

Cycle: path from one node back to itself

Example: {V, X, Y, W, U, V}



Loop: edge directly from node to itself

Many graphs don't allow loops

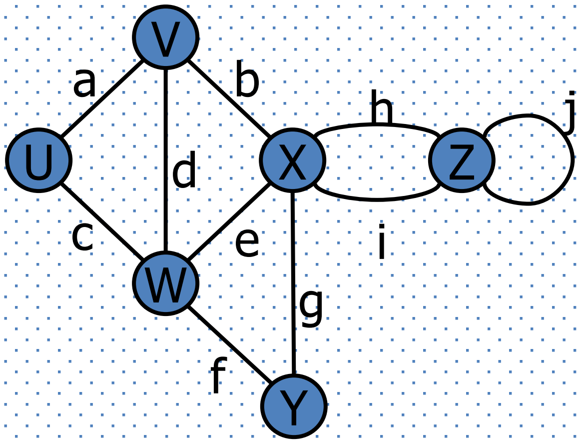
Degree: number of edges touching a vertex

Example: W has degree 4

What is the degree of X?

of Z?

Adjacent vertices: vertices connected directly by an edge



1. Graph Terminology

Path: a path from vertex A to B is a sequence of edges that can be followed starting from A to reach B

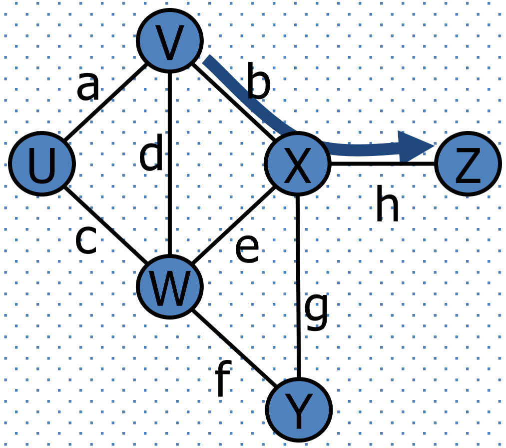
Can be represented as vertices visited or edges taken

Example: path from V to Z: {b, h} or {(v,x), (x,z)} or {V, X, Z}

Reachability: v_2 is reachable from v_1 if a path exists from v_1 to v_2

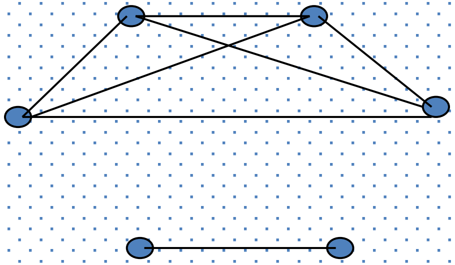
Connected graph: one in which it is possible to reach any node from any other

Is this graph connected?

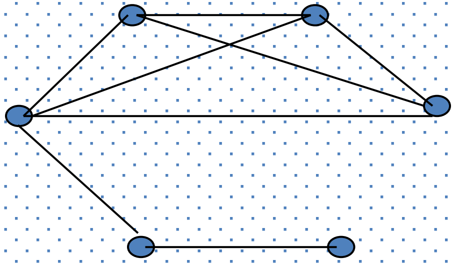


1. Graph Terminology

An graph is said to be *connected* if there is a path between every pair of nodes. Otherwise, the graph is *disconnected*.



Disconnected



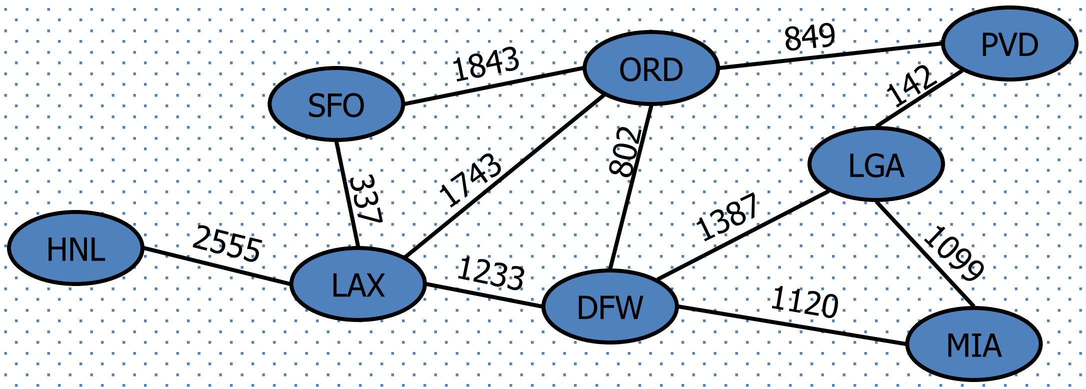
Connected

1. Graph Terminology

<u>Weighted graphs</u>

Weight: (optional) cost associated with a given edge.

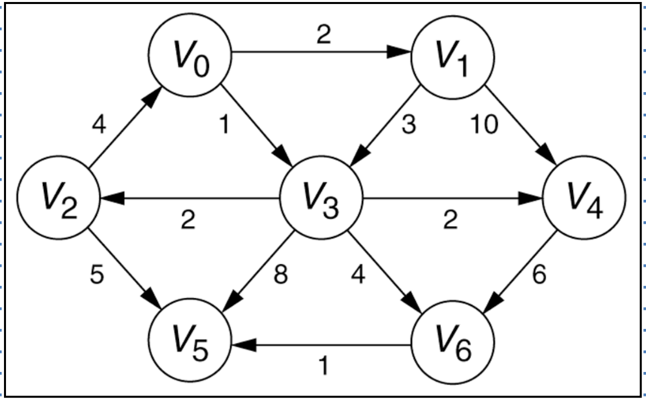
Example: graph of airline flights



1. Graph Terminology

Directed graph (digraph): edges are one-way connections between vertices

If graph is directed, a vertex has a separate in/out degree

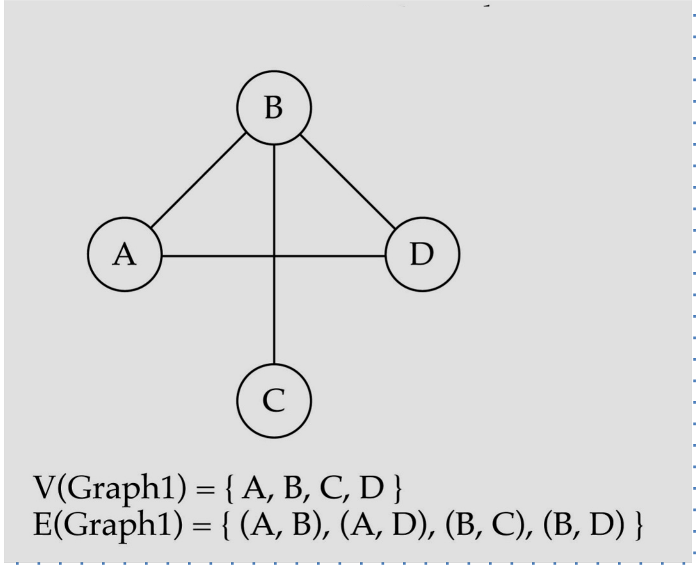


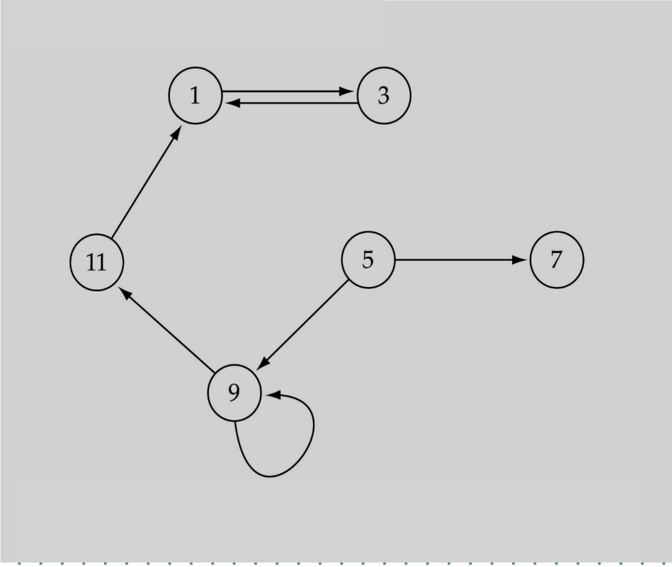
1. Graph Terminology

Directed vs. undirected graphs

When the edges in a graph have no direction, the graph is called undirected.

When the edges in a graph have a direction, the graph is called directed (or digraph)



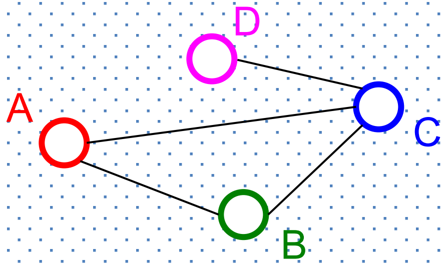


1. Graph Terminology

- In undirected graphs, edges have no specific direction. Edges are always "two-way”. Thus, (u, v) ∊ E implies (v, u) ∊ E.
- Only one of these edges needs to be in the set.

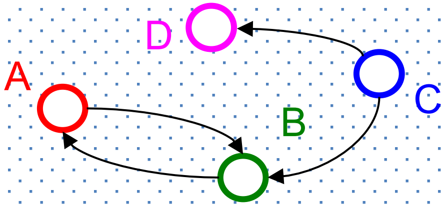
<u>Degree of a vertex</u>: number of edges containing that vertex (the number of adjacent vertices)

Deg(A) = 2
Deg(B) = 2
Deg(C) = 3
Deg(D) = 1

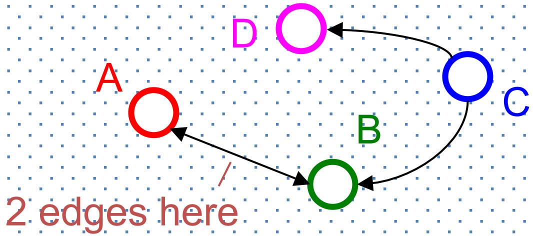


1. Graph Terminology

In directed graphs (or digraphs), edges have direction



or



Thus, (u, v) ∈ E does not imply (v, u) ∈ E.

Let (u, v) ∈ E mean u → v
Call u the source and v the destination.

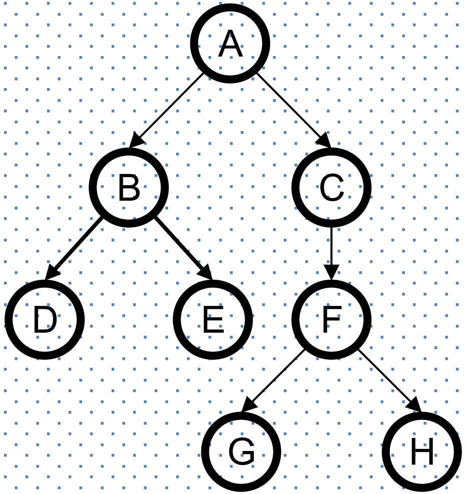
<u>In-Degree of a vertex</u>: number of in-bound edges (edges where the vertex is the destination)
<u>Out-Degree of a vertex</u>: number of out-bound edges (edges where the vertex is the source)

1. Graph Terminology

Trees as Graphs

Every tree is a graph with some restrictions:

- The tree is directed
- There is exactly one directed path from the root to every node



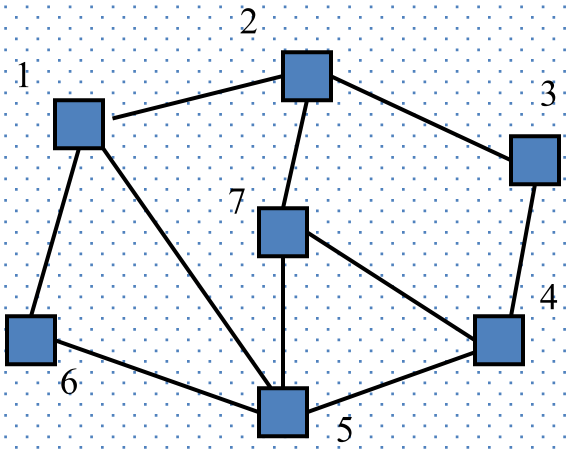
1. Graph Terminology

Implementing a Graph

To program a graph data structure, what information would we need to store?

For each vertex?

For each edge?

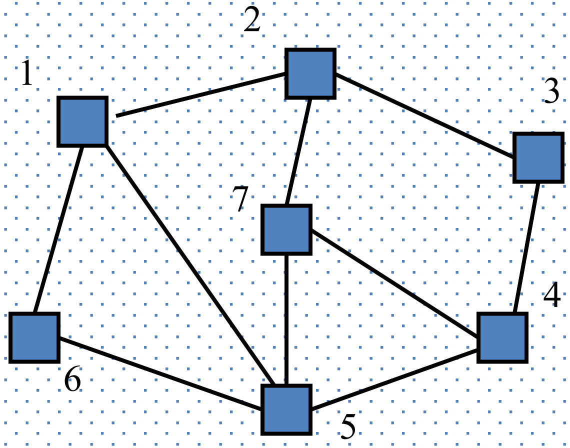


1. Graph Terminology

Implementing a Graph

What kinds of questions would we want to be able to answer about a graph *G*?

- Which vertices are adjacent to vertex *v*?
- What edges touch vertex *v*?
- What are the edges of *G*?
- What are the vertices of *G*?
- What is the degree of vertex *v*?



Graph Implementation Strategies

1. Edge List
2. Adjacency Matrix
3. Adjacency List

1. Graph Terminology

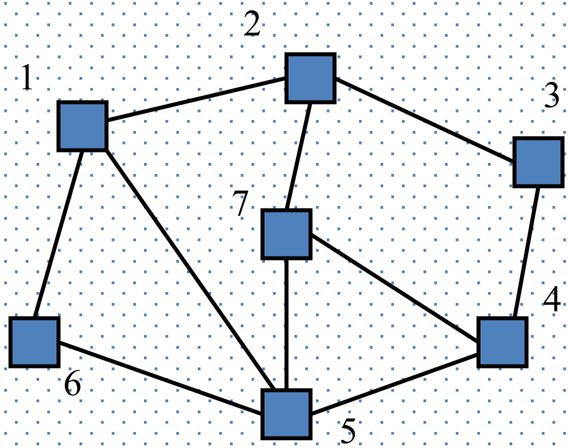
Implementing a Graph

1. Edge List

Edge List: an unordered list of all edges in the graph.

* This is NOT an array

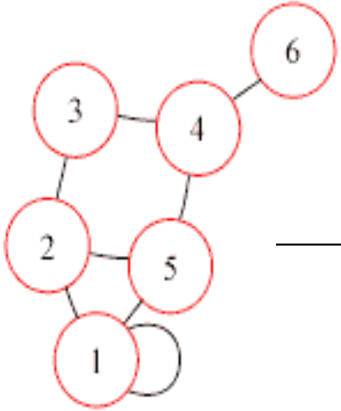
1	1	1	2	2	3	5	5	5	7
2	5	6	7	3	4	6	7	4	4



1. Graph Terminology

Implementing a Graph

2. Adjacency Matrix



$$\begin{pmatrix} 2 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}.$$

Adjacency Matrix: an n × n matrix where:

- **the nondiagonal entry a_{ij} is the number of edges joining vertex i and vertex j** (or the weight of the edge joining vertex i and vertex j)
- **the diagonal entry a_{ii} corresponds to the number of loops** (self-connecting edges) at vertex i

1. Graph Terminology

Implementing a Graph

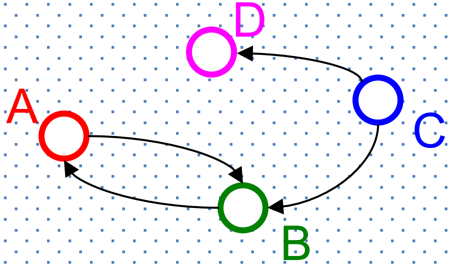
2. Adjacency Matrix

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



	A	B	C	D
A	F	T	F	F
B	T	F	F	F
C	F	T	F	T
D	F	F	F	F

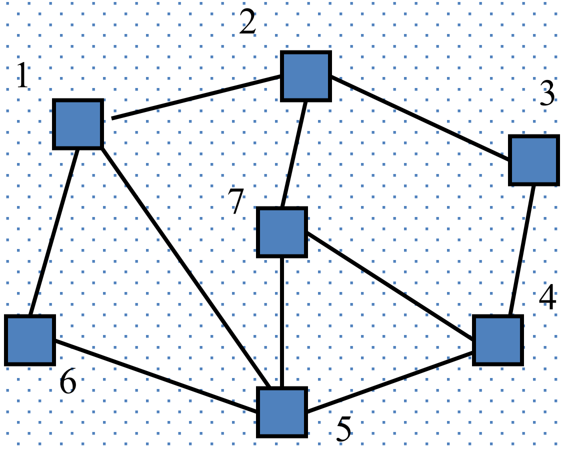
1. Graph Terminology

Implementing a Graph

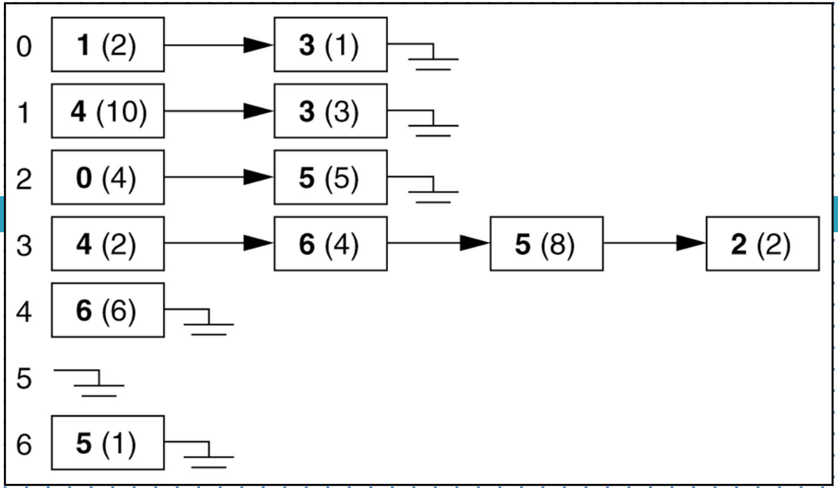
2. Adjacency Matrix

How do we figure out the degree of a given vertex?
How do we find out whether an edge exists from A to B?
How could we look for loops in the graph?

	1	2	3	4	5	6	7
1	0	1	0	0	1	1	0
2	1	0	1	0	0	0	1
3	0	1	0	1	0	0	0
4	0	0	1	0	1	0	1
5	1	0	0	1	0	1	1
6	1	0	0	0	1	0	0
7	0	1	0	1	1	0	0



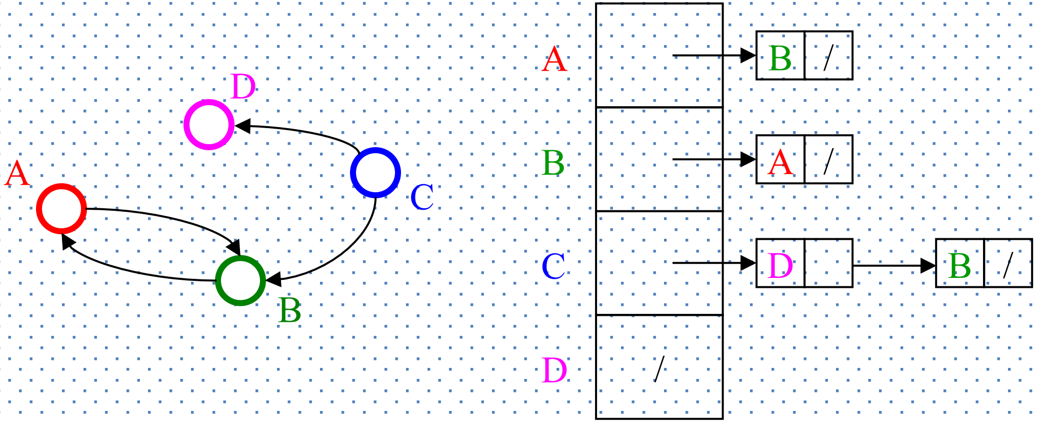
1. Graph Terminology



Implementing a Graph

3. Adjacency List

Adjacency List: stores edges as individual linked lists of references to each vertex's neighbors.



1. Graph Terminology

Implementing a Graph

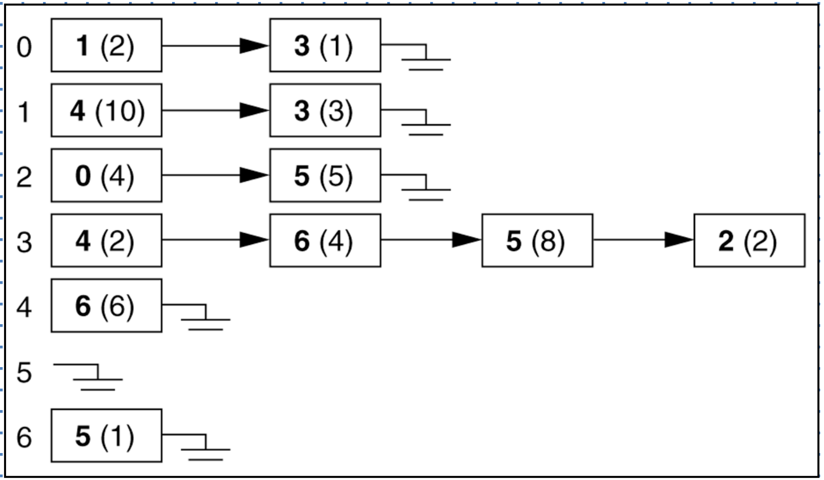
3. Adjacency List

Advantages:

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

Disadvantages:

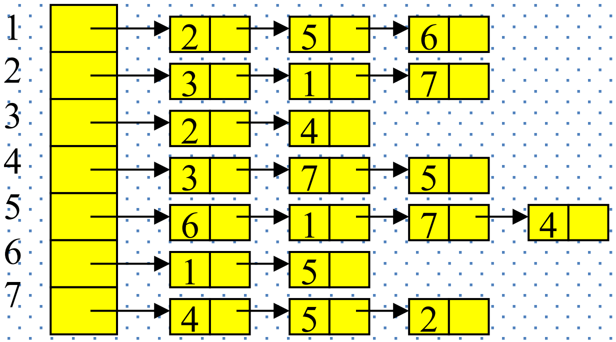
- determining whether an edge exists between two nodes: O(average degree)

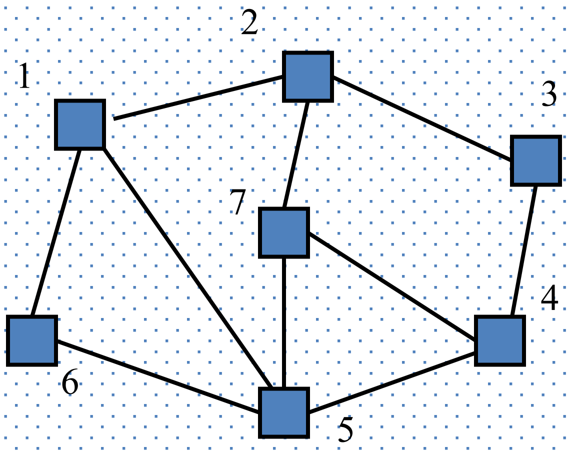


Implementing a Graph

3. Adjacency List

How do we figure out the degree of a given vertex?
How do we find out whether an edge exists from A to B?
How could we look for loops in the graph?





2. Graph Algorithms

Depth-First Search

- DFS is an edge-based technique.
- It uses the Stack data structure and performs two stages:
  - first : visited vertices are pushed into the stack
  - second : if there are no vertices then visited vertices are popped.

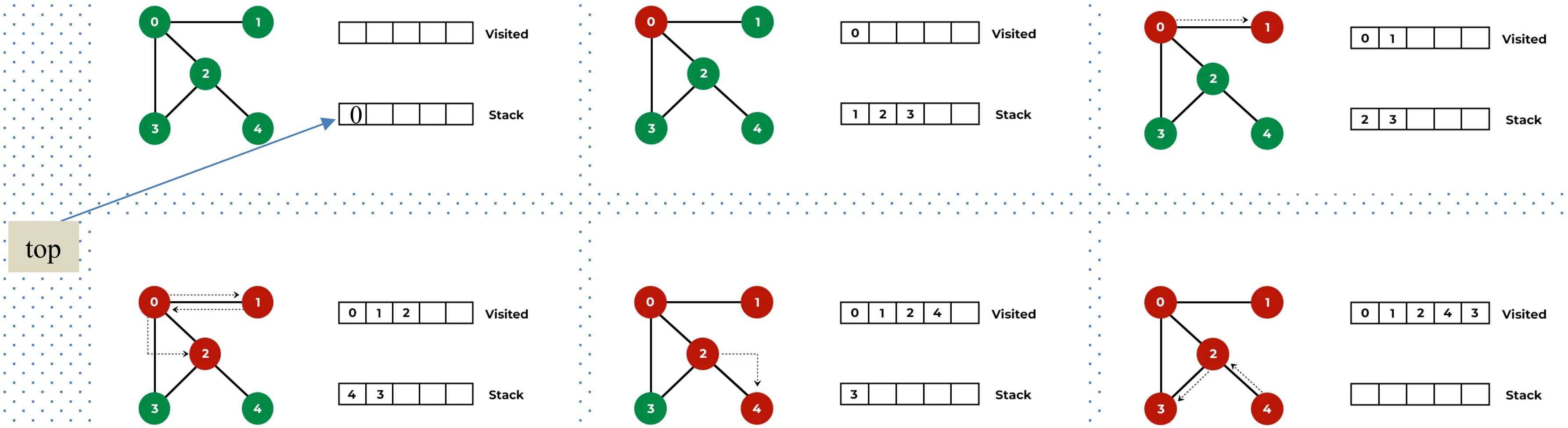
```
    A
   / \
  B   D
 /   / \
C   E   F
```

```
A, B, C, D, E, F
```

2. Graph Algorithms

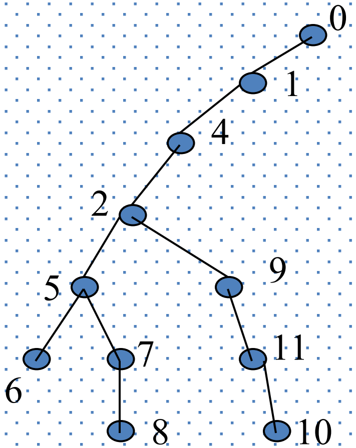
Depth-First Search

```
DFS(G,v)   ( v is the vertex where the search starts )
   Stack S := {};   ( start with an empty stack )
   for each vertex u, set visited[u] := false;
   push S, v;
   while (S is not empty) do
      u := pop S;
      if (not visited[u]) then
         visited[u] := true;
         for each unvisited neighbour w of u
            push S, w;
      end if
   end while
END DFS()
```

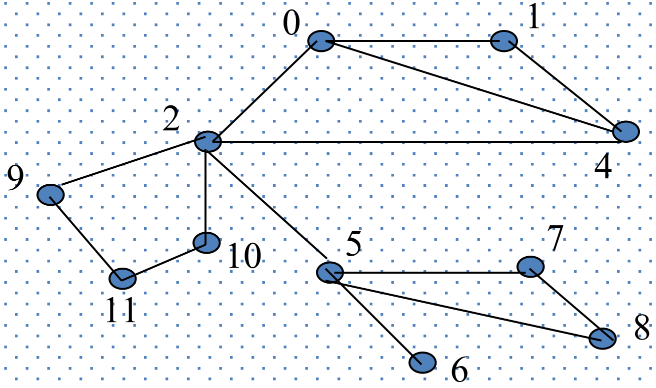


2. Graph Algorithms

Illustration of DFS



DFS Tree



Graph G

2. Graph Algorithms

Breadth-First Search

- **BFS** is a vertex-based technique.
- It uses a Queue data structure that follows first in first out.
- In BFS, one vertex is selected at a time when it is visited and marked then its adjacent are visited and stored in the queue.
- It is slower than DFS.

```
     A
    / \
   B   C
  /   / \
 D   E   F
```

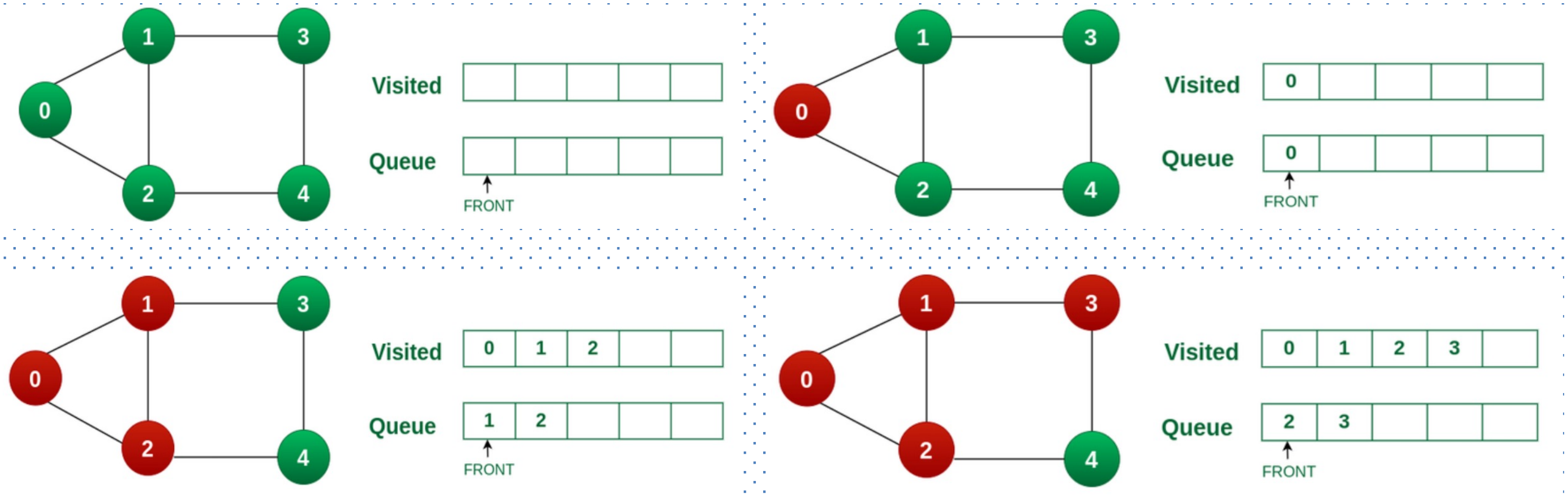
```
A, B, C, D, E, F
```

2. Graph Algorithms

```
BFS(startV) {
  Push startV to Queue
  Add  startV to visited

  while ( Queue is not empty )
    currentV = Pop Queue
    Print or Output currentV
    for each vertex adjV adjacent to currentV
      if ( adjV is not visited)
        Push adjV to Queue
        Add  adjV to visited
}
```

Breadth-First Search

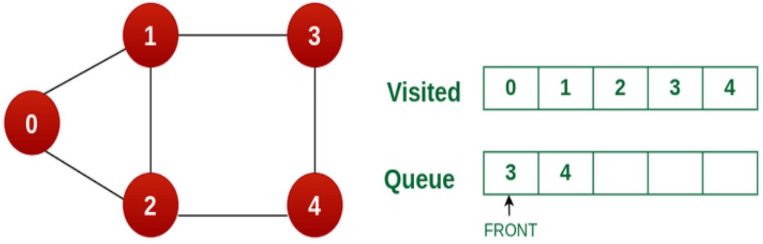


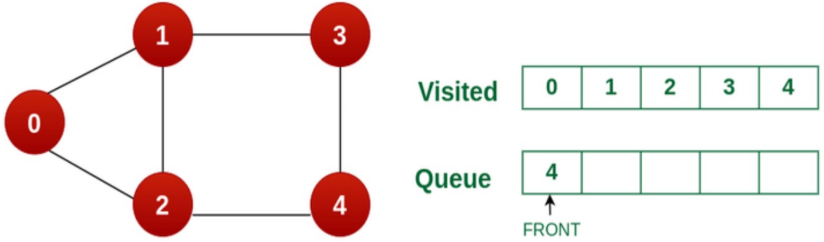
2. Graph Algorithms

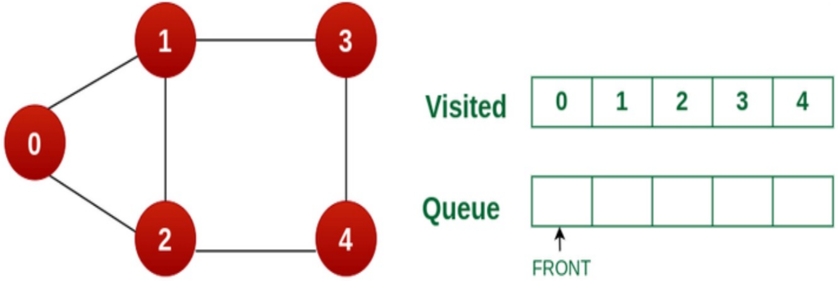
```
BFS(startV) {
  Push startV to Queue
  Add  startV to visited

  while ( Queue is not empty )
    currentV = Pop Queue
    Print or Output currentV
    for each vertex adjV adjacent to currentV
      if ( adjV is not visited)
        Push adjV to Queue
        Add  adjV to visited
}
```

<u>Breadth-First Search</u>



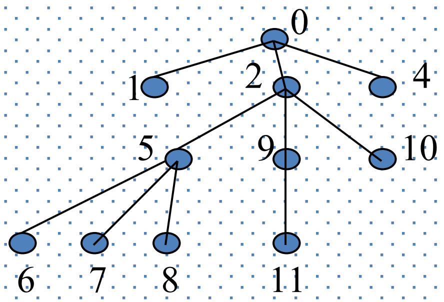




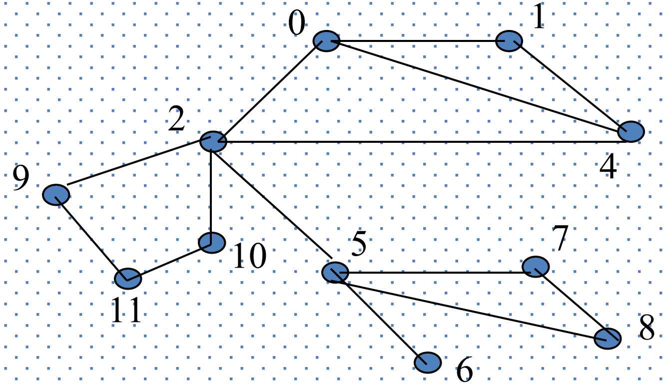
Queue becomes empty, So, terminate these process of iteration.

2. Graph Algorithms

Illustration of BFS



BFS Tree



Graph G

Data Structures
and
Algorithms