# Chapter 9 Graph – part 1

**South China University of Technology**

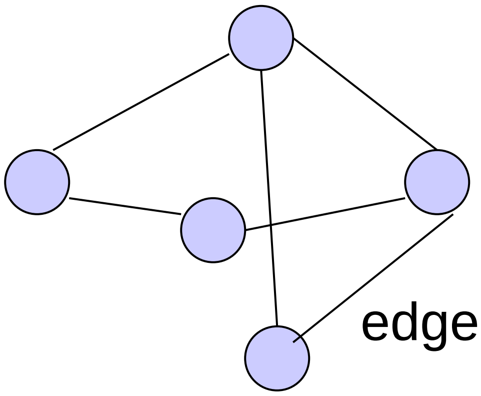
**College of Software Engineering**

**Huang Min**

# Definition of Graph

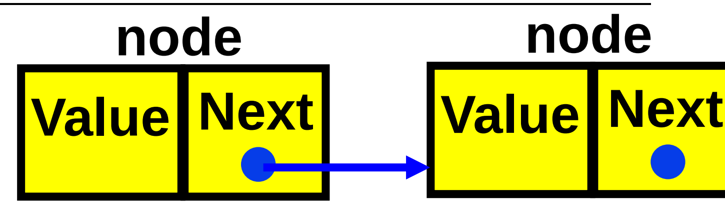
# Graphs

- Graphs are composed of
  - Nodes (vertices)
    - Labeled or unlabeled
  - Edges (arcs)
    - Directed or undirected
    - Labeled or unlabeled

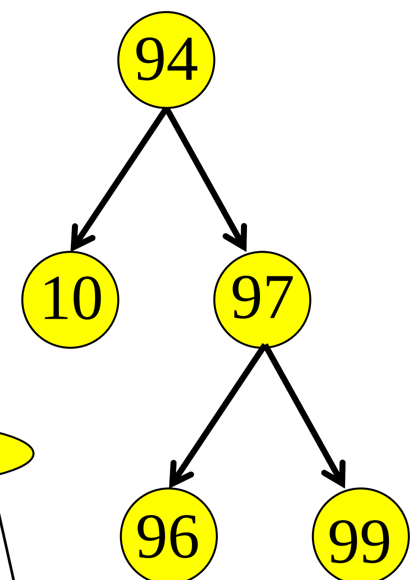


# Motivation for Graphs

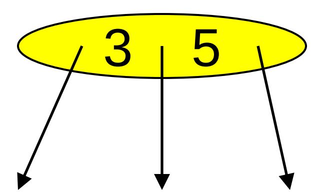
- Consider the data structures we have looked at so far…
- Linked list: nodes with 1 incoming edge + 1 outgoing edge




- Binary trees/heaps: nodes with 1 incoming edge + 2 outgoing edges




- B-trees: nodes with 1 incoming edge + multiple outgoing edges



# Motivation for Graphs

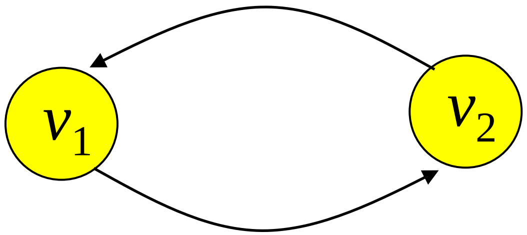
- Consider data structures for representing the following problems…
  - › Representing a Maze
    - (Nodes = rooms; Edge = door or passage)
  - › Representing Electrical Circuits
    - Nodes = battery, switch, resistor, etc; Edges = connections
  - › Program statements
    - Nodes = symbols/operators; Edges = relationships
  - › Information Transmission in a Computer Network
    - Nodes = computers; Edges = transmission rates
  - › Traffic Flow on Highways
    - Nodes = cities; Edges = # vehicles on connecting highway
  - › ……

# Graph Definition

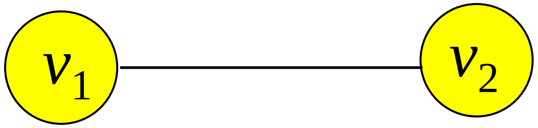
- A graph is simply a collection of nodes plus edges
  - Linked lists, trees, and heaps are all special cases of graphs
- The nodes are known as vertices (node = "vertex")
- Formal Definition: A graph $G$ is a pair $(V, E)$ where
  - $V$ is a set of vertices or nodes
  - $E$ is a set of edges that connect vertices

# Directed vs Undirected Graphs

- If the order of edge pairs ($v_1$, $v_2$) matters, the graph is directed (also called a digraph): $(v_1, v_2) \neq (v_2, v_1)$

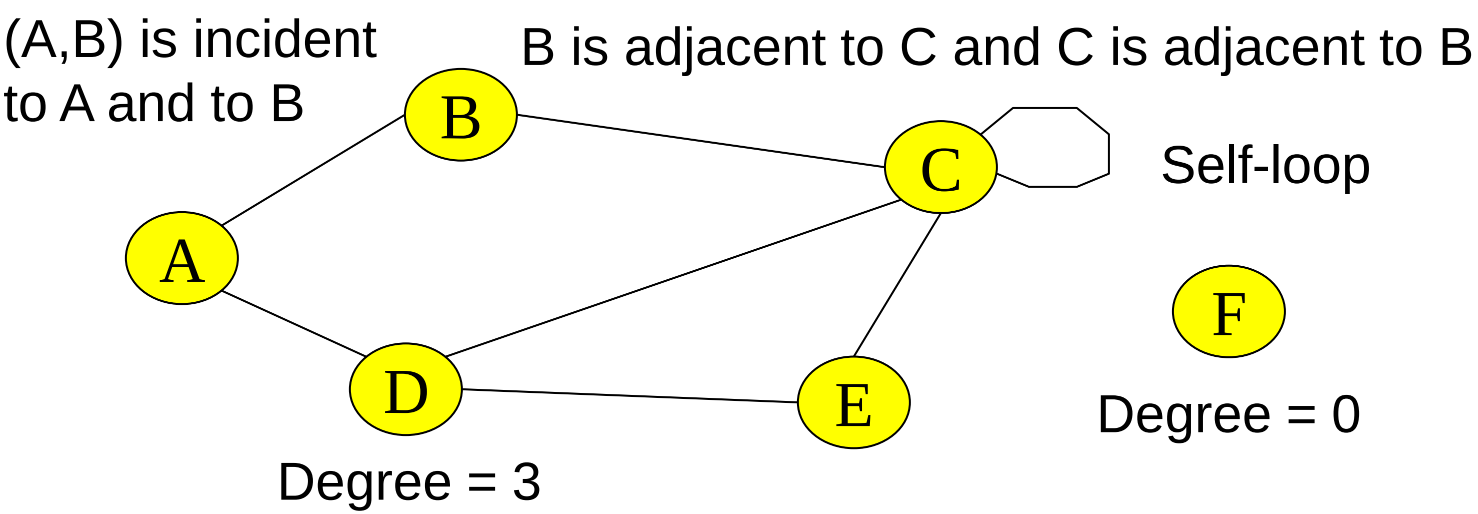



- If the order of edge pairs ($v_1$, $v_2$) does not matter, the graph is called an undirected graph: in this case, $(v_1, v_2) = (v_2, v_1)$



# Undirected Terminology

- Two vertices u and v are adjacent in an undirected graph G if {u,v} is an edge in G
  - › edge e = {u,v} is incident with vertex u and vertex v
- The degree of a vertex in an undirected graph is the number of edges incident with it, denoted with deg(v)
  - › a self-loop counts twice (both ends count)

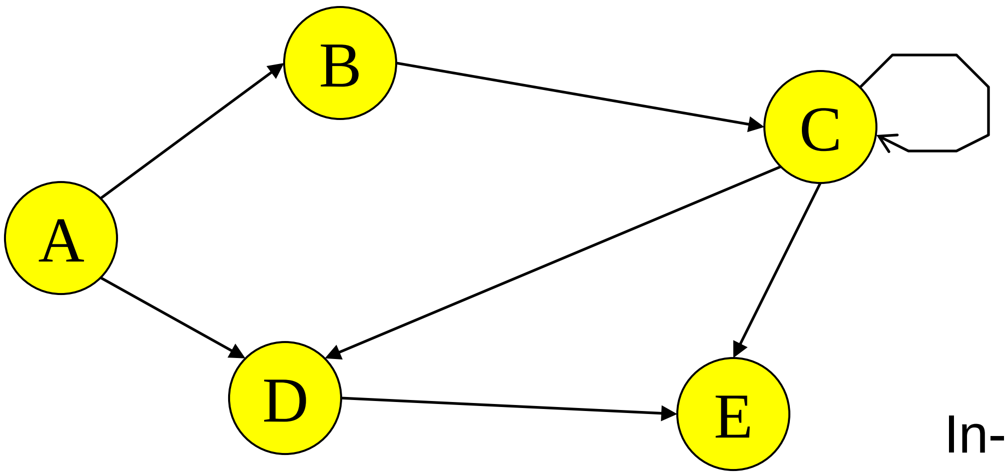


# Directed Terminology

- Vertex u is adjacent to vertex v in a directed graph G if (u,v) is an edge in G
  - vertex u is the initial vertex of (u,v)
- Vertex v is adjacent from vertex u
  - vertex v is the terminal (or end) vertex of (u,v)
- Degree
  - in-degree is the number of edges with the vertex as the terminal vertex
  - out-degree is the number of edges with the vertex as the initial vertex

# Directed Terminology

B adjacent to C and C adjacent from B




In-degree = 2
Out-degree = 1

In-degree = 0
Out-degree = 0

# Handshaking Theorem

- Let G=(V,E) be an undirected graph with |E|=e edges. Then

$$2e = \sum_{v \in V} \deg(v)$$

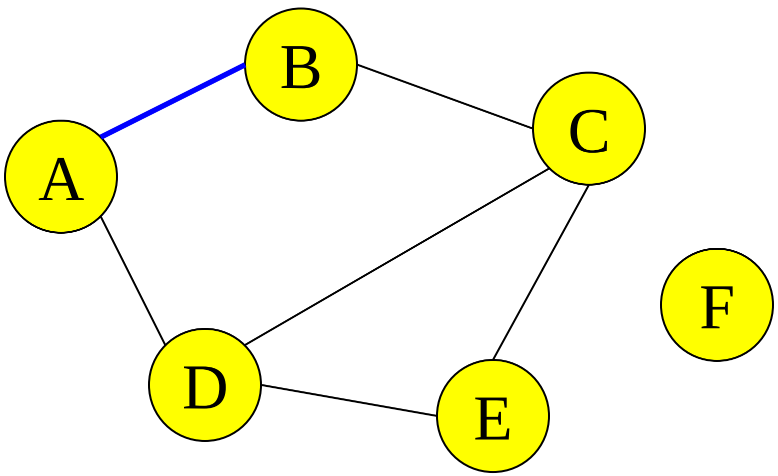
Add up the degrees of all vertices.

- Every edge contributes +1 to the degree of each of the two vertices it is incident with
  - number of edges is exactly half the sum of deg(v)
  - the sum of the deg(v) values must be even

# Graph Representations

- Space and time are analyzed in terms of:
  - Number of vertices = $|V|$ and
  - Number of edges = $|E|$
- There are at least two ways of representing graphs:
  - The *adjacency matrix* representation
  - The *adjacency list* representation

# Adjacency Matrix for a Undirected Digraph

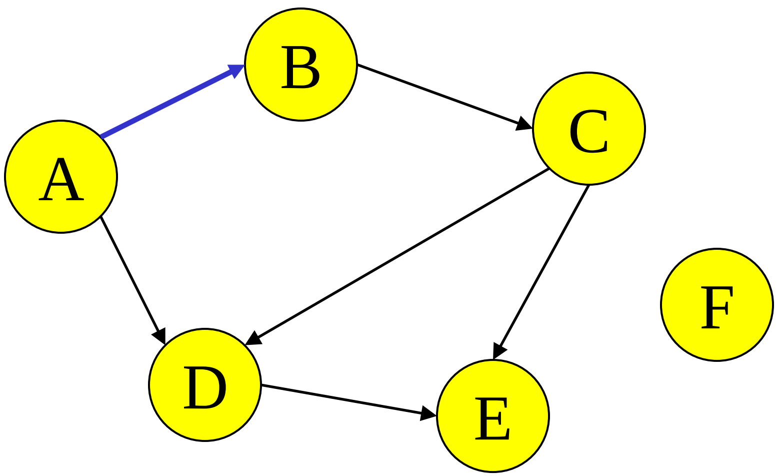



$$
\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ F \end{array}
\begin{array}{c}
\begin{array}{cccccc} A & B & C & D & E & F \end{array} \\
\begin{pmatrix}
0 & 1 & 0 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
\end{array}
$$

$$
M(v, w) = \begin{cases} 1 \text{ if } (v, w) \text{ is in E} \\ 0 \text{ otherwise} \end{cases}
$$

Space = $|V|^2$

# Adjacency Matrix for a Directed Digraph



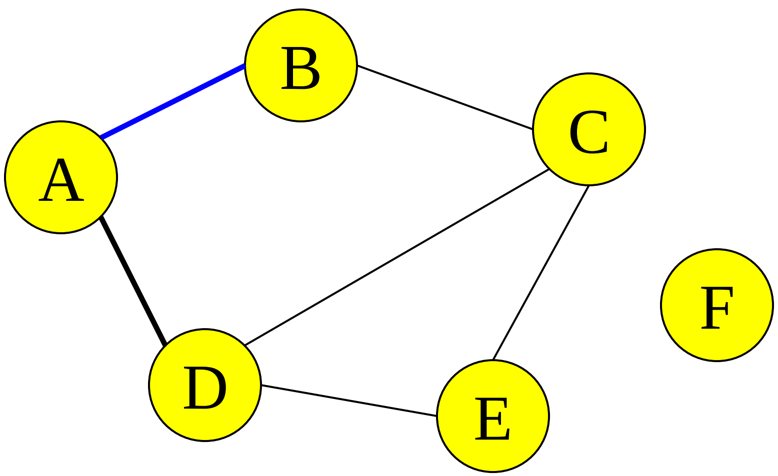

$$
M(v, w) = \begin{cases} 1 \text{ if } (v, w) \text{ is in E} \\ 0 \text{ otherwise} \end{cases}
$$

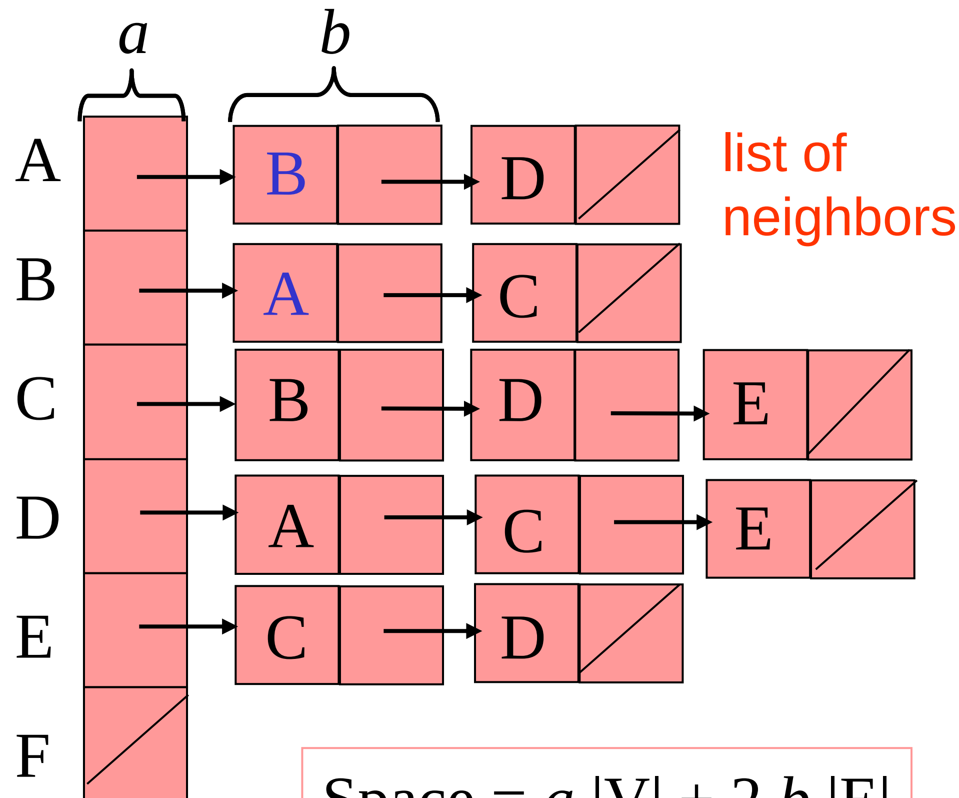
$$
\begin{array}{c c}
 & \begin{array}{cccccc} A & B & C & D & E & F \end{array} \\
\begin{array}{c} A \\ B \\ C \\ D \\ E \\ F \end{array} & \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}
\end{array}
$$

Space = $|V|^2$

# Adjacency List for a Undirected Digraph

For each $v$ in $V$, $L(v)$ = list of $w$ such that $(v, w)$ is in $E$

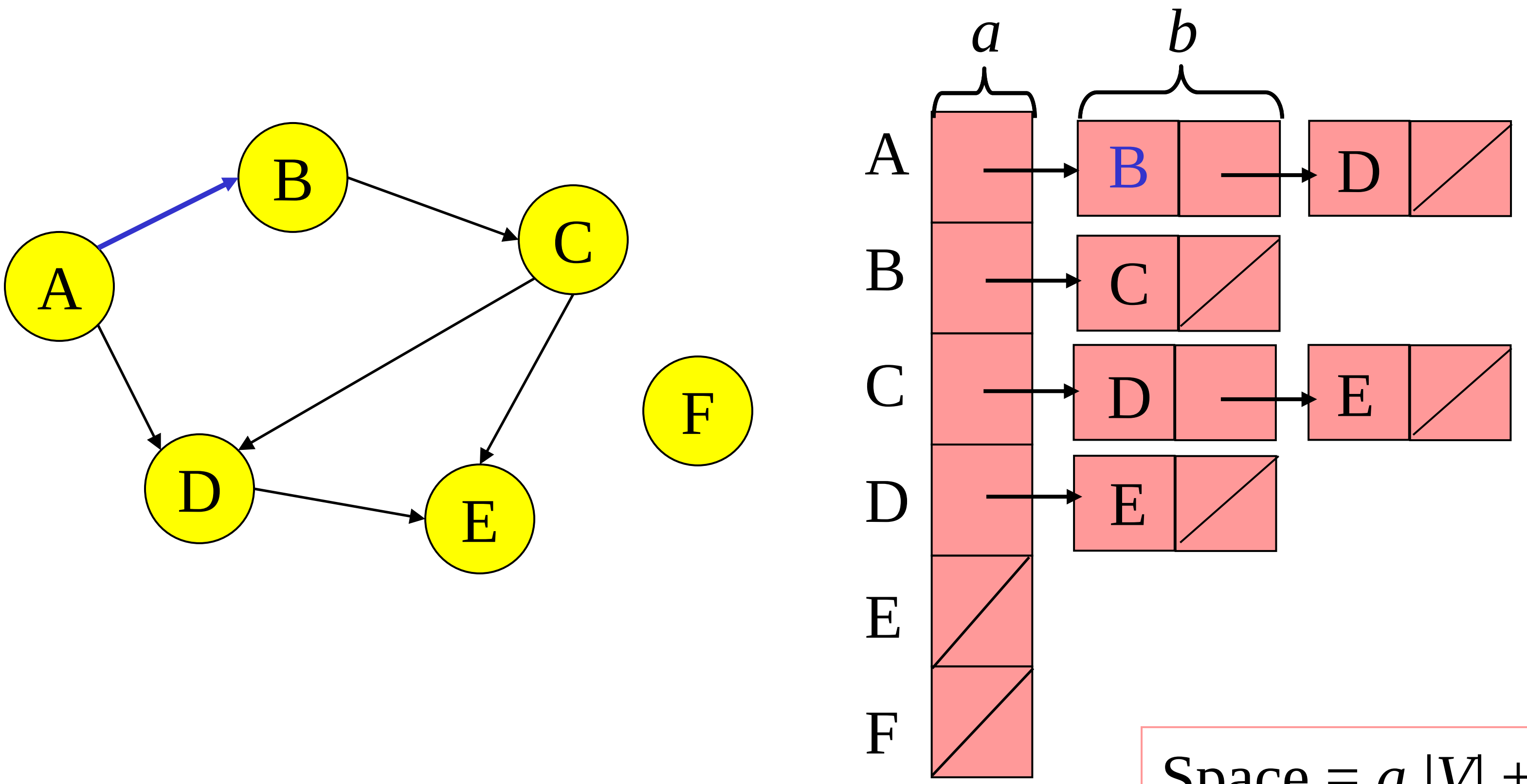



Space = $a\,|V| + 2\,b\,|E|$

# Adjacency List for a Directed Digraph

For each $v$ in $V$, $L(v)$ = list of $w$ such that $(v, w)$ is in $E$




Space = $a\,|V| + b\,|E|$

# Graph Searching

# Graph Searching

- Find Properties of Graphs
  - Spanning trees
  - Connected components
  - Bipartite structure
  - Biconnected components
- Applications
  - Finding the web graph – used by Google and others
  - Garbage collection – used in Java run time system
  - Alternating paths for matching

# Graph Searching Methodology Breadth-First Search (BFS)

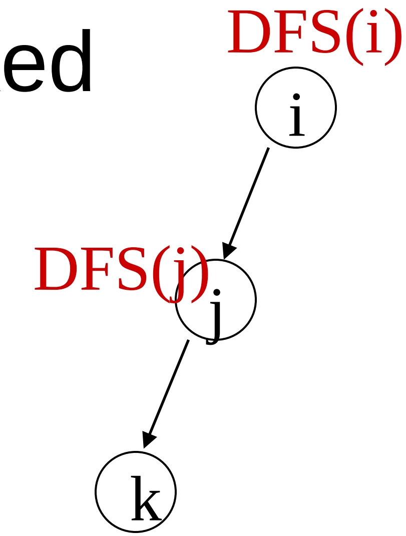
- Breadth-First Search (BFS)
  - Use a queue to explore neighbors of source vertex, then neighbors of neighbors etc.
  - All nodes at a given distance (in number of edges) are explored before we go further

# Graph Searching Methodology Depth-First Search (DFS)

- Depth-First Search (DFS)
  - Searches down one path as deep as possible
  - When no  nodes available, it backtracks
  - When backtracking, it explores side-paths that were not taken
  - Uses a stack (instead of a queue in BFS)
  - Allows an easy recursive implementation

# Depth First Search Algorithm

- Recursive marking algorithm
- Initially every vertex is unmarked



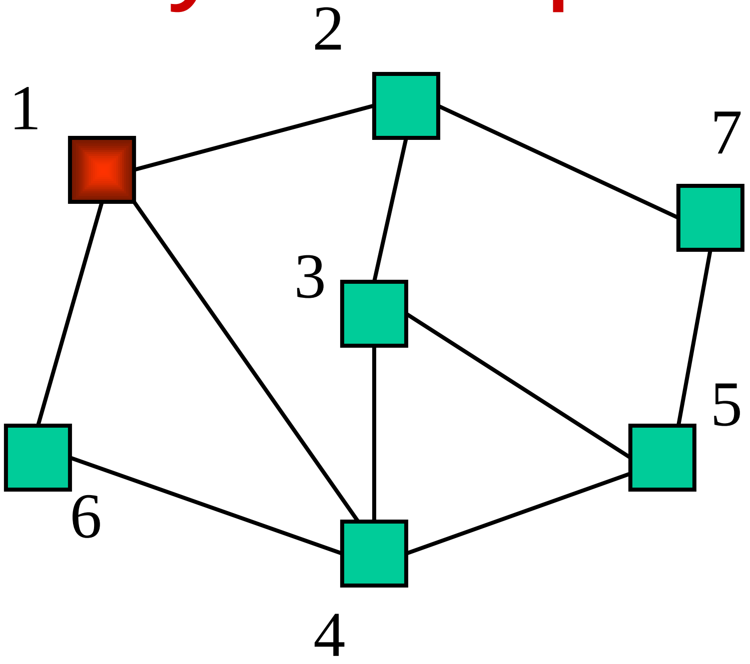

```
DFS(i: vertex)
   mark i;
   for each j adjacent to i do
      if j is unmarked then DFS(j)
end{DFS}
```

Marks all vertices reachable from i

# DFS Application: Spanning Tree

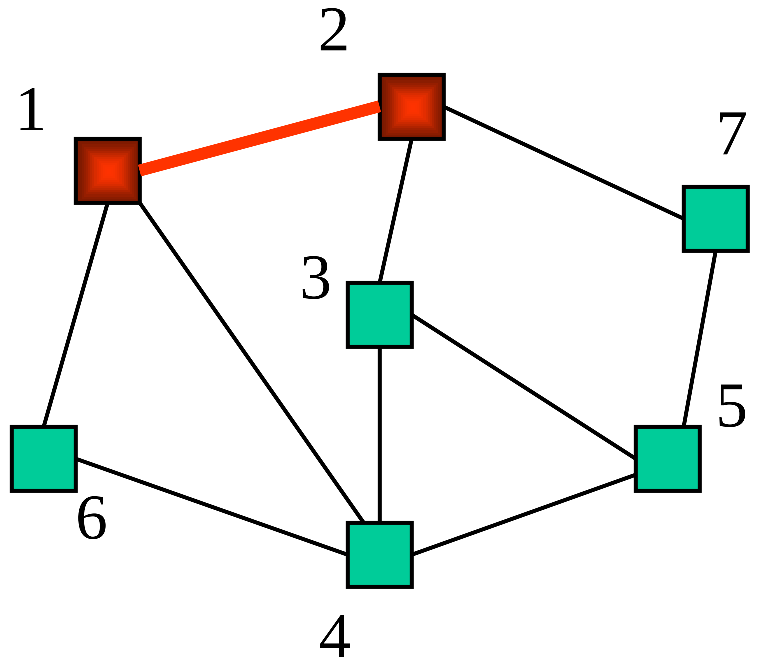
- Given a (undirected) graph G(V,E) a spanning tree of G is a graph G'(V',E')
  - V' = V, the tree touches all vertices (spans) the graph
  - E' is a subset of E such G' is connected and there is no cycle in G'
  - A graph is connected if given any two vertices u and v, there is a path from u to v

# Example of DFS: Graph connectivity and spanning tree




DFS(1)

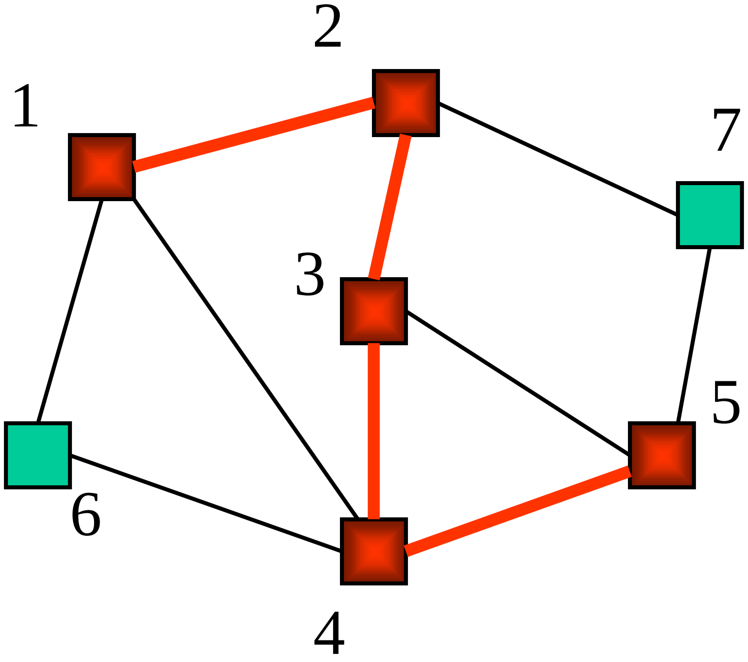
# Example Step 2




DFS(1)
DFS(2)

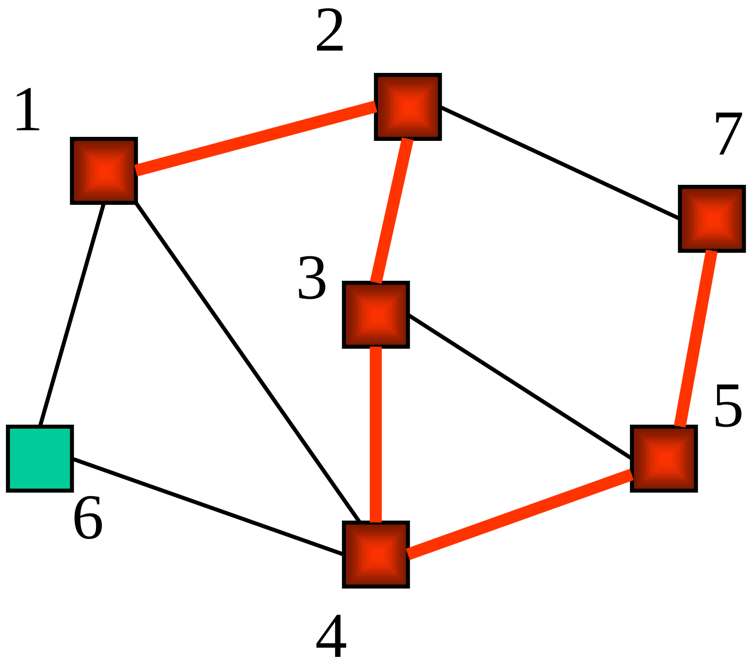
Red links will define the spanning tree if the graph is connected

# Example Step 5



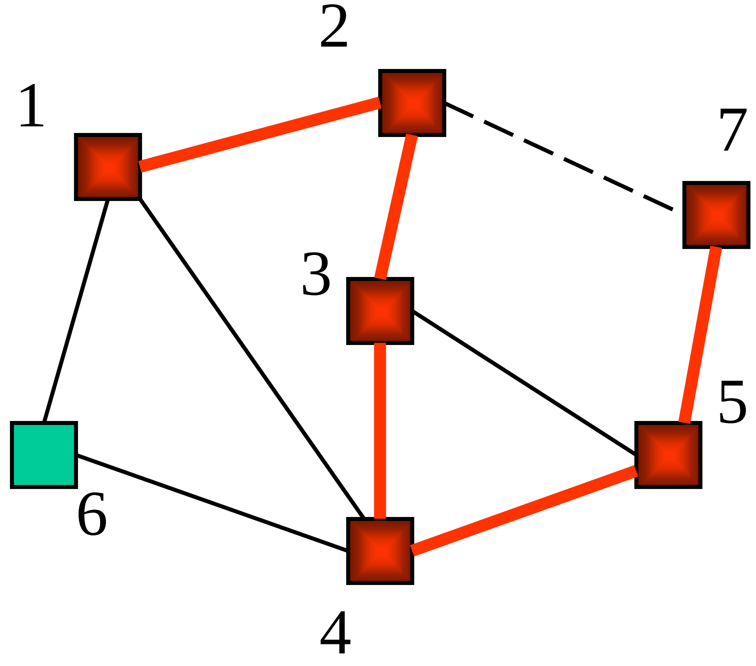

DFS(1)
DFS(2)
DFS(3)
DFS(4)
DFS(5)

# Example Steps 6 and 7




DFS(1)
DFS(2)
DFS(3)
DFS(4)
DFS(5)
~~DFS(3)~~
DFS(7)

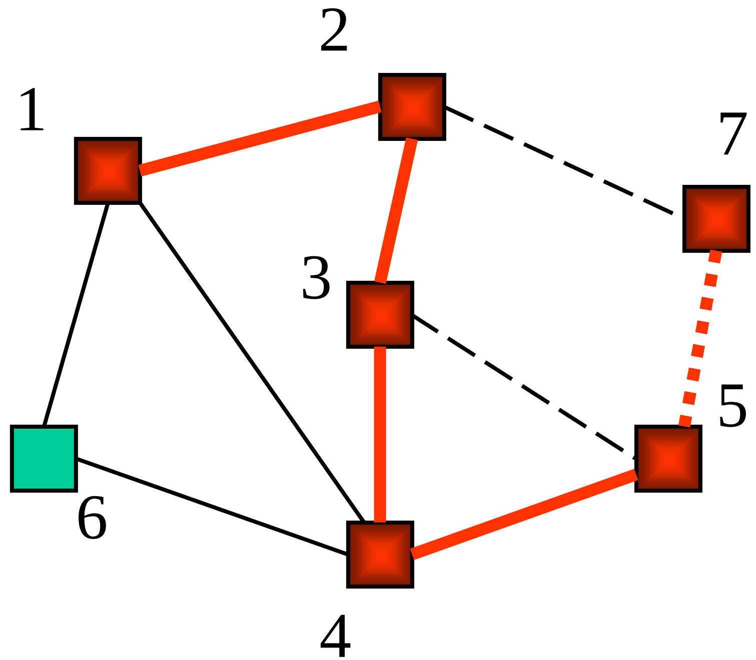
# Example Steps 8 and 9




DFS(1)
DFS(2)
DFS(3)
DFS(4)
DFS(5)
DFS(7)

Now back up.

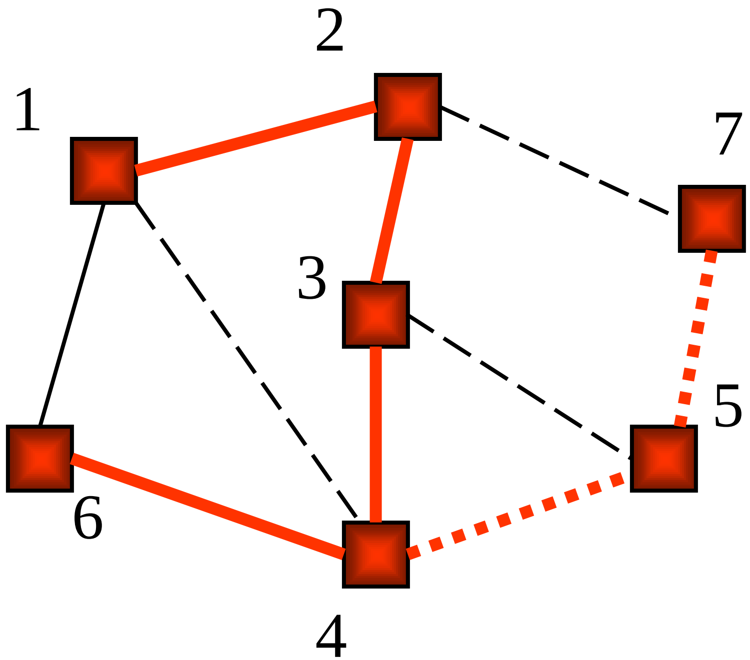
# Example Step 10 (backtrack)




DFS(1)
DFS(2)
DFS(3)
DFS(4)
DFS(5)

Back to 5,
but it has no
more neighbors.

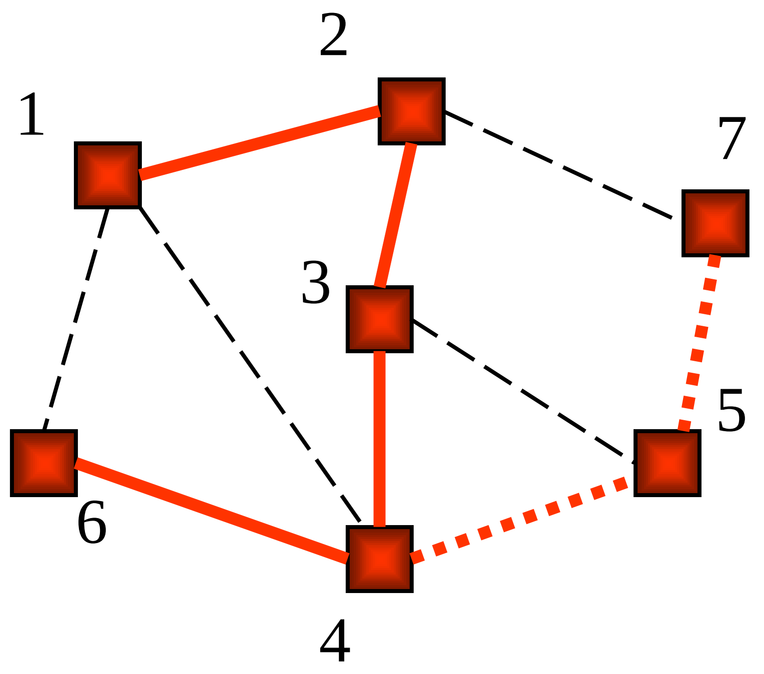
# Example Step 12




DFS(1)
 DFS(2)
  DFS(3)
   DFS(4)
    DFS(6)

Back up to 4.
From 4 we can
get to 6.

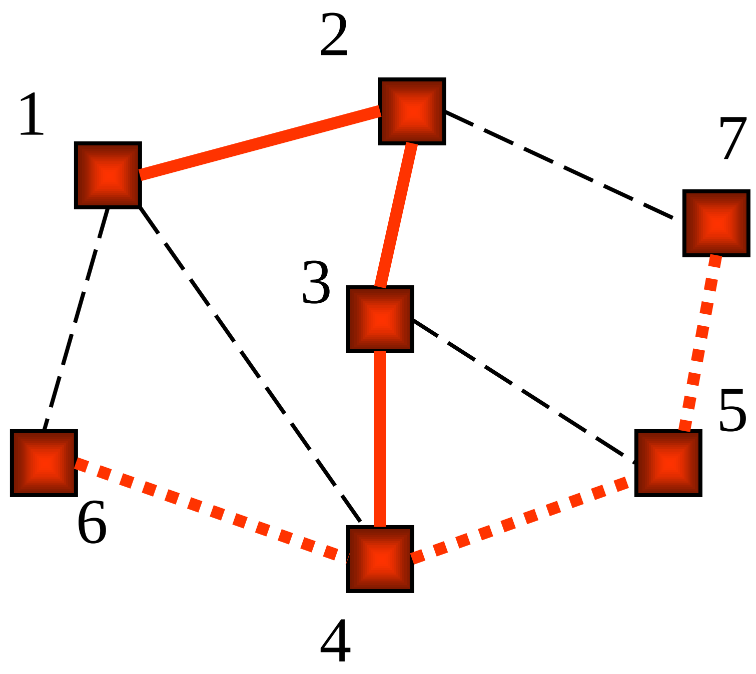
# Example Step 13




DFS(1)
DFS(2)
DFS(3)
DFS(4)
DFS(6)

From 6 there is nowhere new to go. Back up.

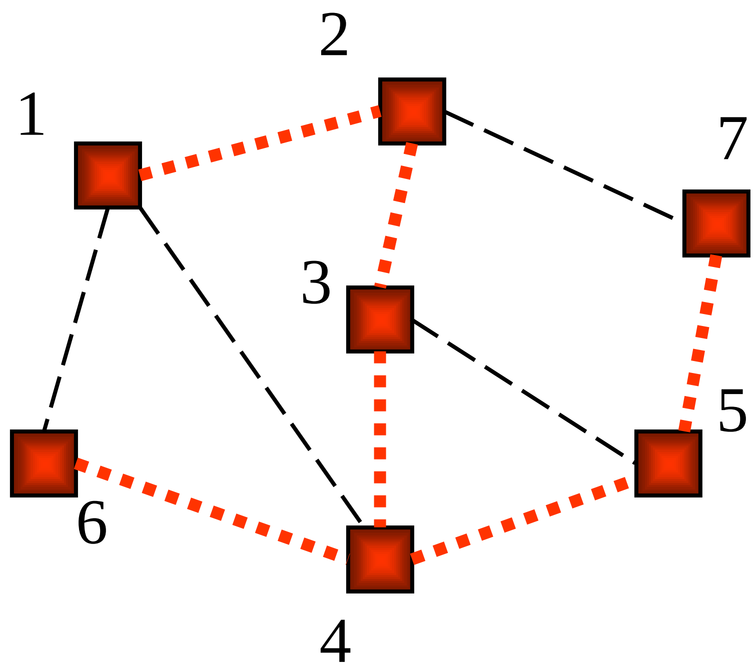
# Example Step 14




DFS(1)
 DFS(2)
  DFS(3)
   DFS(4)

Back to 4.
Keep backing up.

# Example Step 17




DFS(1)

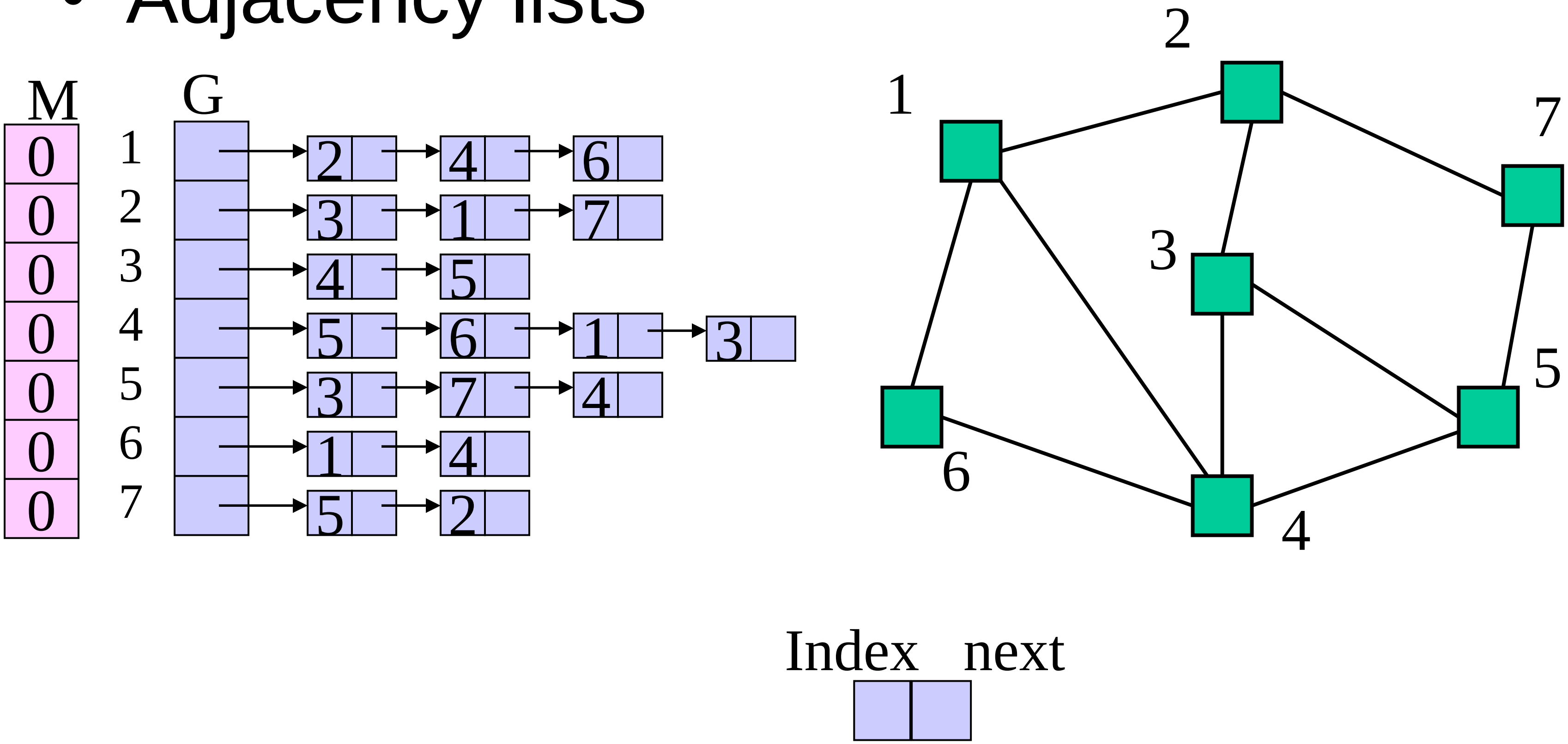
All the way back to 1.

Done.

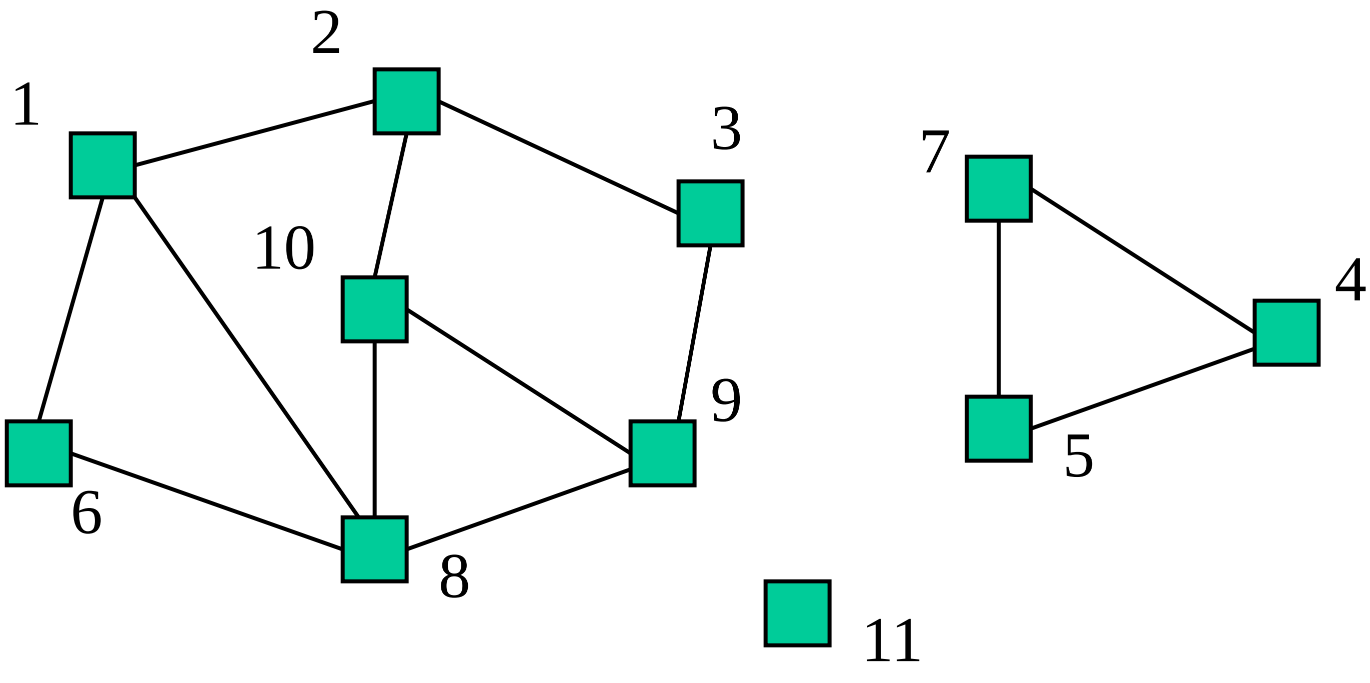
All nodes are marked so graph is connected; red links define a spanning tree

# Adjacency List Implementation

- Adjacency lists

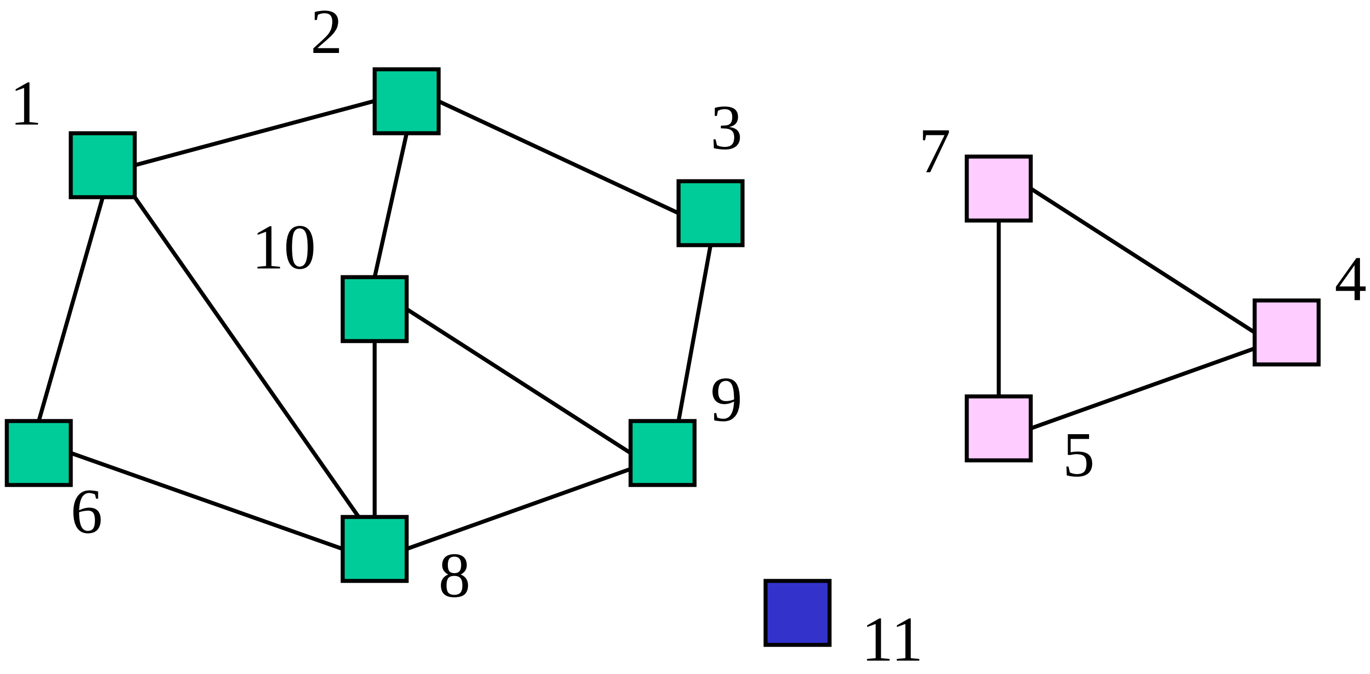


# Another Use for Depth First Search: Connected Components




3 connected components

# Connected Components



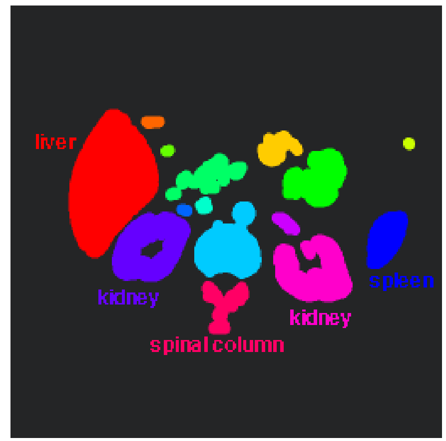

3 connected components are labeled

# Depth-first Search for Labeling Connected components

```
Main {
i : integer
for i = 1 to n do M[i] := 0;  //initial label is zero
label := 1;
for i = 1 to n do
  if M[i] = 0 then DFS(G,M,i,label); //if i is not labeled
  label := label + 1;                  then call DFS
}
DFS(G[]: node ptr array, M[]: int array, i,label: int) {
  v : node pointer;
  M[i] := label;
  v := G[i];  // first neighbor //
  while v ≠ null do  // recursive call (below)
    if M[v.index] = 0 then DFS(G,M,v.index,label);
    v := v.next;  // next neighbor //
}
```

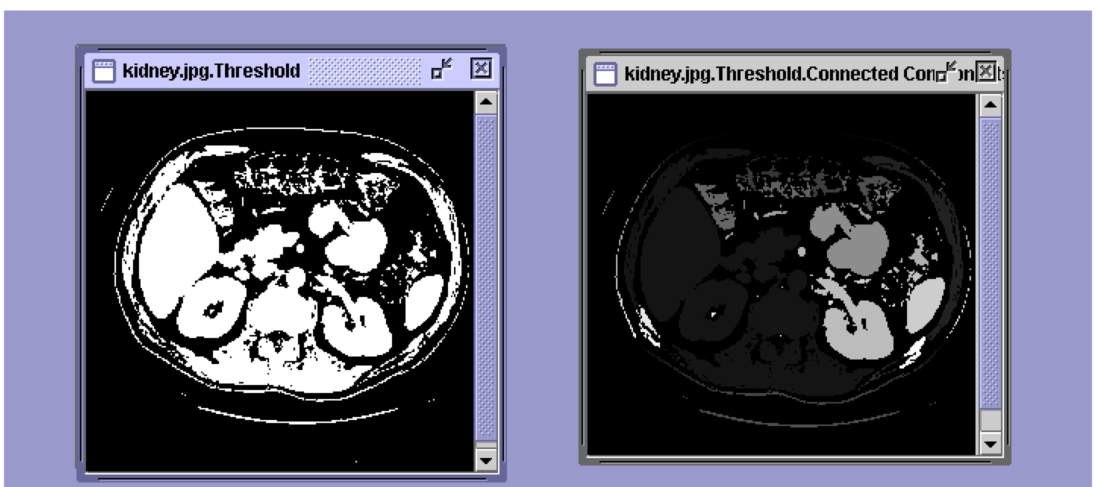
# Connected Components for Image Analysis

1




4

2




3

# Performance DFS

- n vertices and m edges
- Storage complexity $O(n + m)$
- Time complexity $O(n + m)$
- Linear Time!

# Depth First Search

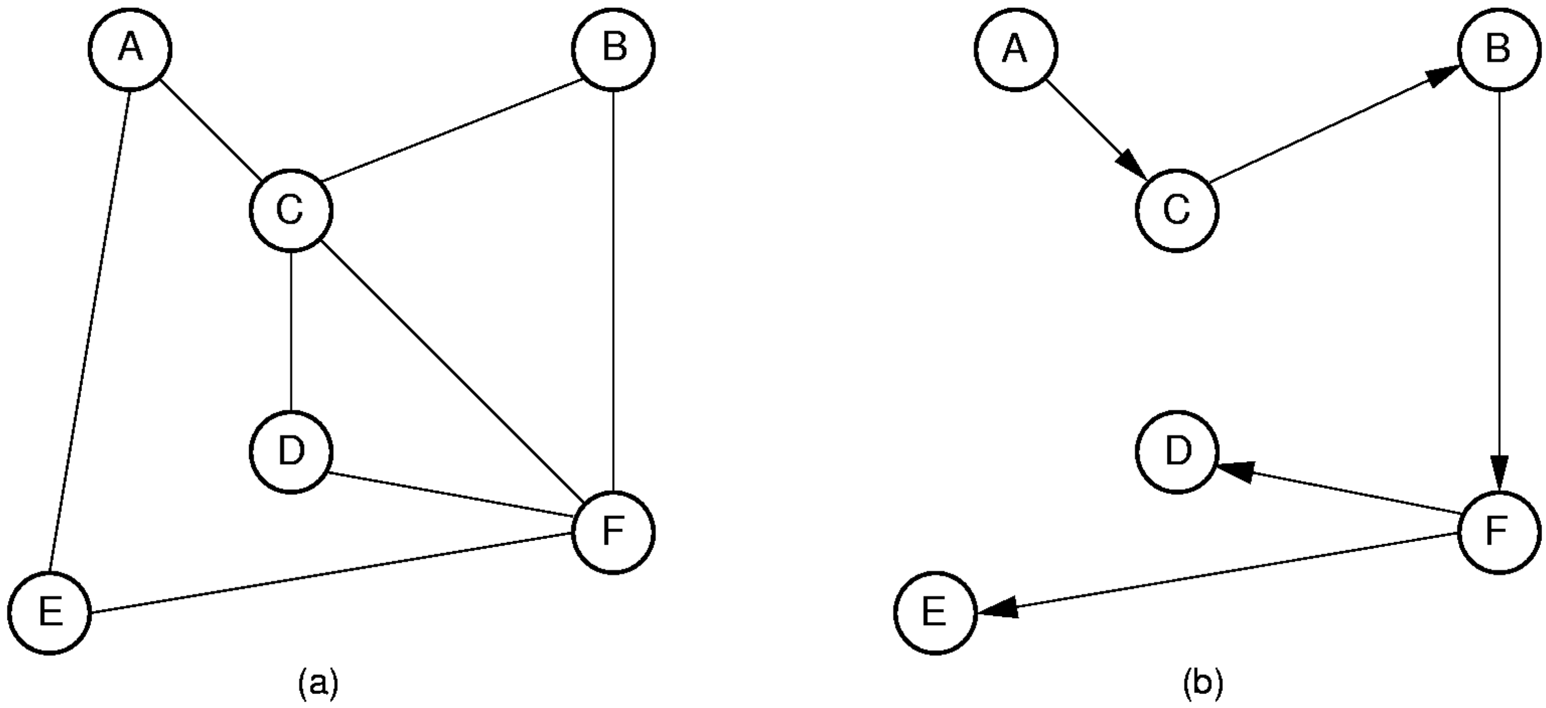



**Figure 11.8** (a) A graph. (b) The depth-first search tree for the graph when starting at Vertex *A*.

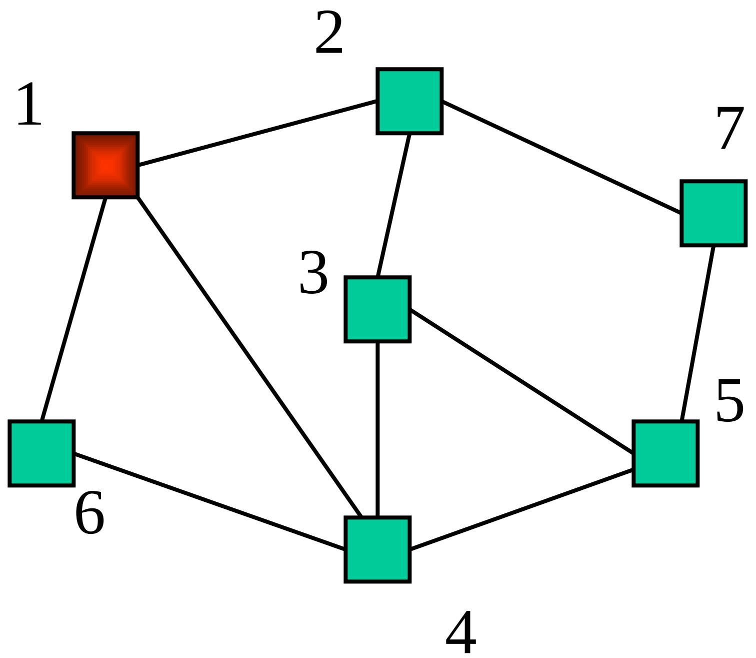
Order that nodes are processed: ACBFDE

# Breadth-First Search

```
BFS
Initialize Q to be empty;
Enqueue(Q,1) and mark 1;
while Q is not empty do
    i := Dequeue(Q);
    for each j adjacent to i do
        if j is not marked then
            Enqueue(Q,j) and mark j;
end{BFS}
```

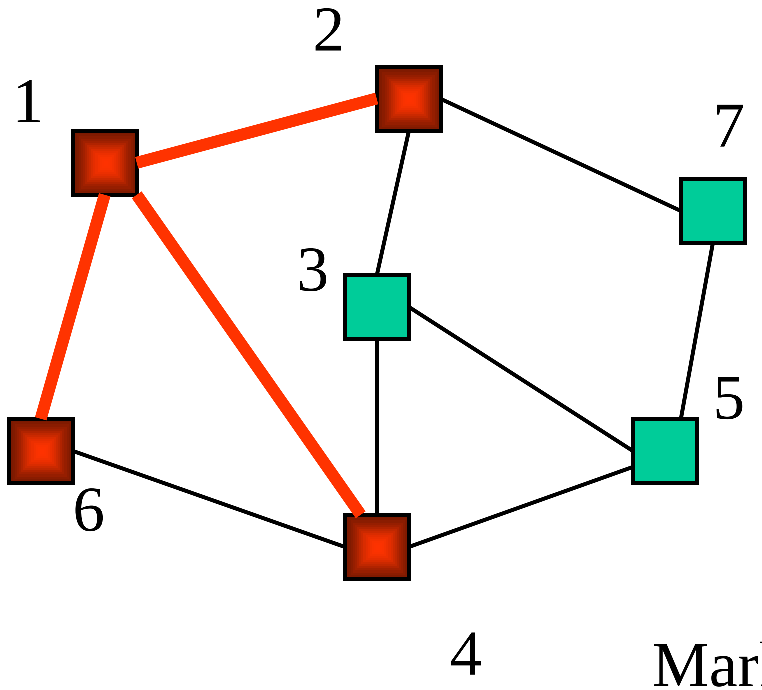
# Can do Connectivity using BFS

- Uses a queue to order search




Queue = 1

# Beginning of example




Queue = 2,4,6

Mark while on queue to avoid putting in queue more than once

# Breadth First Search

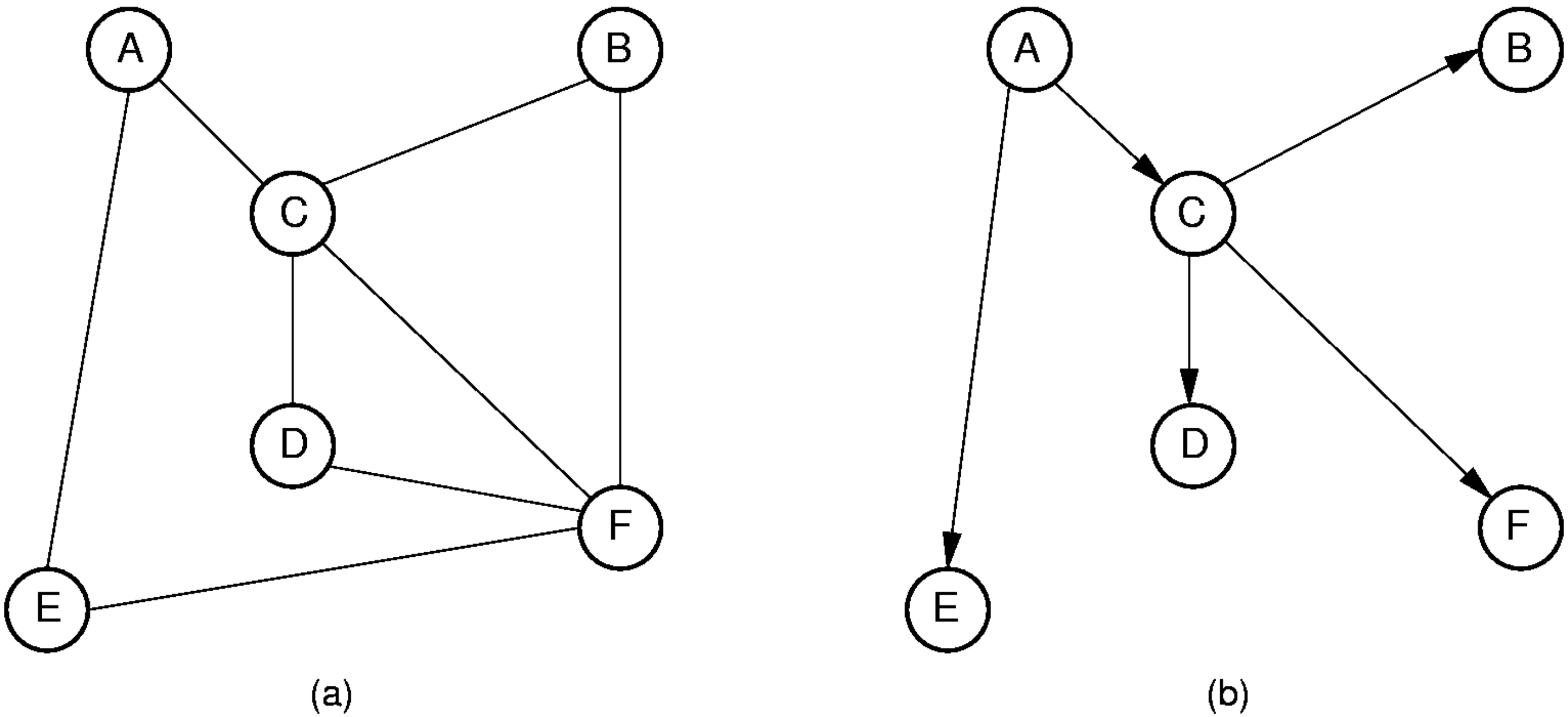



**Figure 11.11** (a) A graph. (b) The breadth-first search tree for the graph when starting at Vertex $A$.

Order that nodes are processed: ACEBDF

# Depth-First vs Breadth-First

- Depth-First
  - Stack or recursion
  - Many applications
- Breadth-First
  - Queue
  - Can be used to find shortest paths from the start vertex
  - Can be used to find short alternating paths for matching