05 Graph (2)

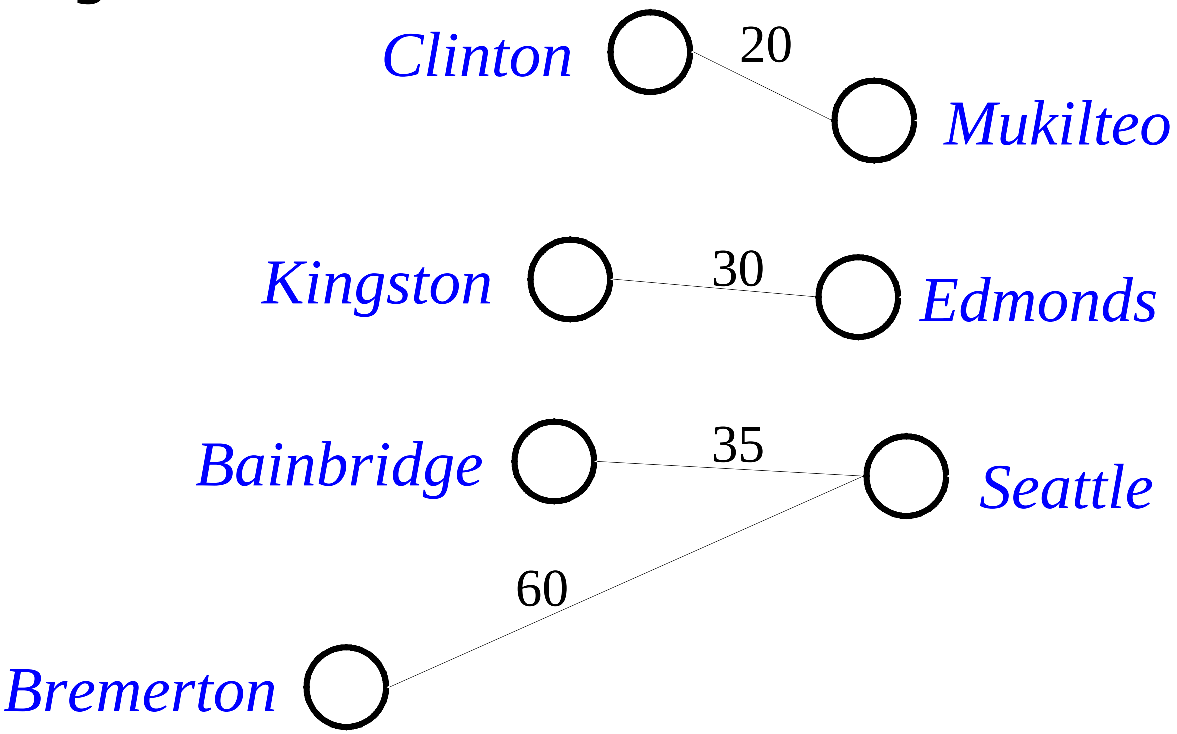
College of Computer Science, CQU

Outline

- Simple Path
- Connectivity
- Graph Traversals
- Topological Sorting

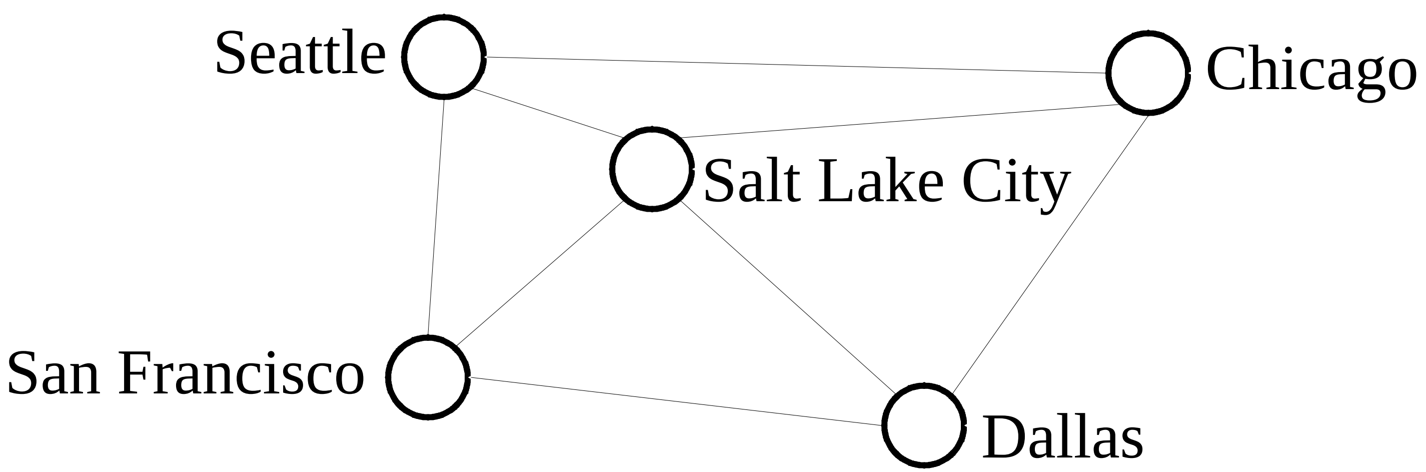
Weighted Graphs

- In a *weighted graph*, each edge has an associated weight or cost.



Paths

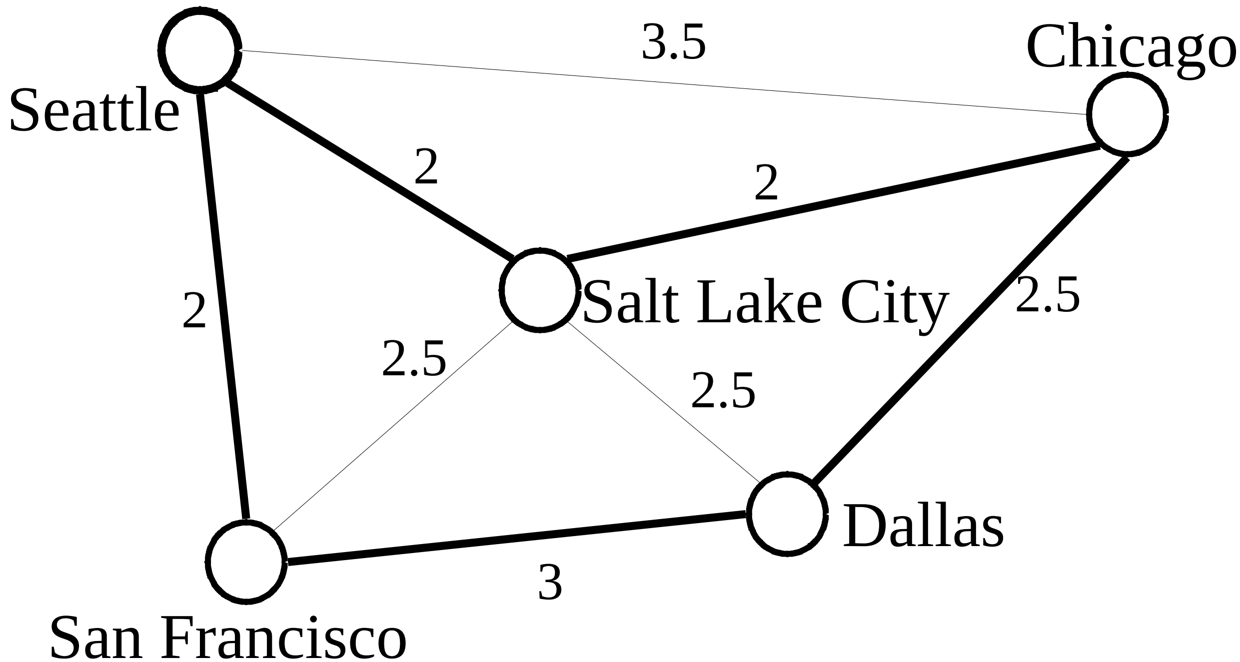
- A ***path*** is a list of vertices $\{v_1, v_2, \ldots, v_n\}$ such that (v_i, v_{i+1}) **E** for all 0 $i < n$.



p = {SEA, SLC, CHI, DAL, SFO, SEA}

Path Length and Cost

- **Path length**: the number of edges in the path
- **Path cost**: the sum of the costs of each edge



length(p) = 5

cost(p) = 11.5

p = {SEA, SLC, CHI, DAL, SFO, SEA}

Simple Paths and Cycles

- A ***simple path*** repeats no vertices (except that the first can be the last):
  - p = {Seattle, Salt Lake City, San Francisco, Dallas}
  - p = {Seattle, Salt Lake City, Dallas, San Francisco, Seattle}

- A ***cycle*** is a path that starts and ends at the same node:
  - p = {Seattle, Salt Lake City, Dallas, San Francisco, Seattle}

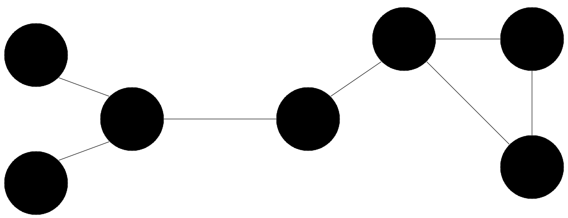
- A ***simple cycle*** is a cycle that repeats no vertices except that the first vertex is also the last (in undirected graphs, no edge can be repeated)

Connectivity

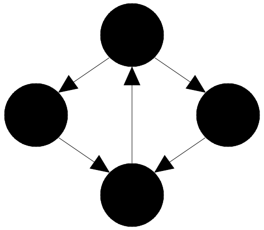
- If there is a path from vertex v_i to v_j, v_i and v_j are connected.
- An undirected graph are connected-----if, for every pair of distinct vertices v_i , v_j, there is a path from v_i i to v_j in G.
- Connected component(undirected graph)----a maximal connected subgrap.(a tree is graph that is connected and acycle(无环))
- Strongly connected(directed graph)--- if, for every pair of distinct vertices v_i , v_j, there is a path from v_i to v_j ,and also from v_j to v_i in G.

Connectivity

- ***Connected*** graph



- ***strongly connected*** graph



Connectivity

- Weakly connected(directed graph)---For a directed graph G, if the undirected graph obtained by suppressing the directions on the edges of G is connected.

- Strongly connected Component---a maximal subgraph that is strongly connected.

- Degree of v_i-----the number of edges incident to that vertex.

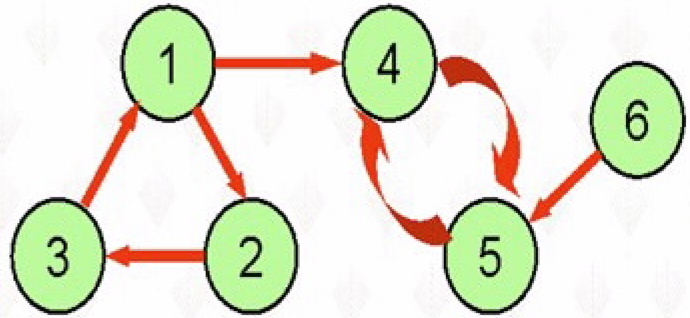
- In-degree---the number of edges that have v_i as the head.

- out-degree---the number of edges that have v_i as the tail.

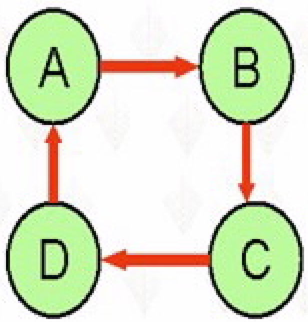
  the number of edges:

$$e = \frac{1}{2}\sum_{i=0}^{n-1} d_i$$

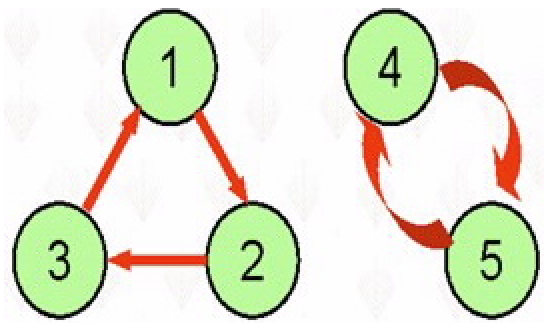
Where d_i is the degree of vertex v_i



weakly connected



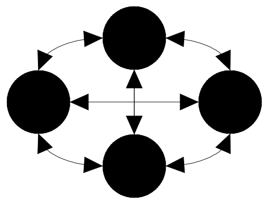
strongly connected



strongly connected component

Connectivity

- A ***complete*** graph has an edge between every pair of vertices



Connectivity

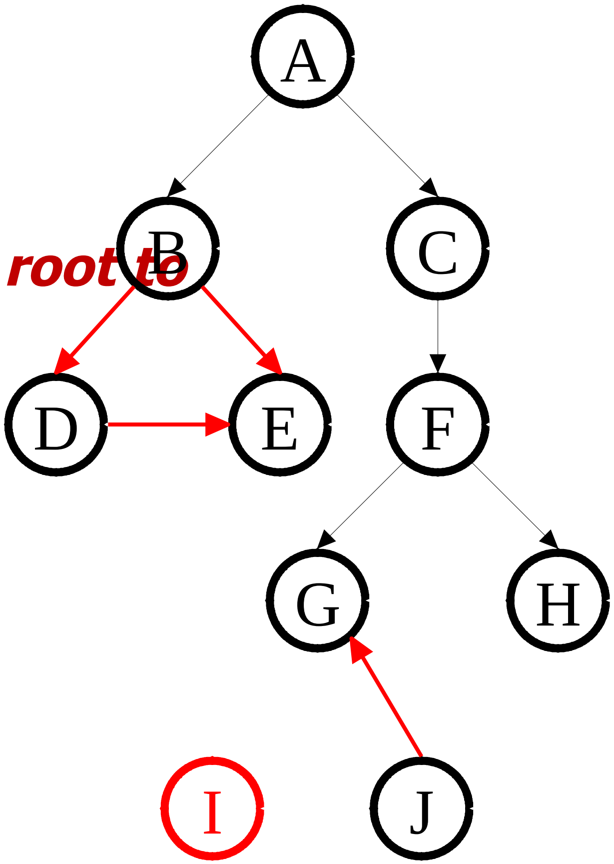
- Acyclic–a graph without cycles.
- directed acyclic graph(DAG)--a directed graph without cycles.
- free tree—a connected, undirected graph with cycles.
- spanning tree—a subgraph of undirected graph G, which is connected and without cycles.

Graph Density

- A ***sparse*** graph has O(|V|) edges
- A ***dense*** graph has ($|V|^2$) edges
- Anything in between is either ***sparsish*** or ***densy*** depending on the context.

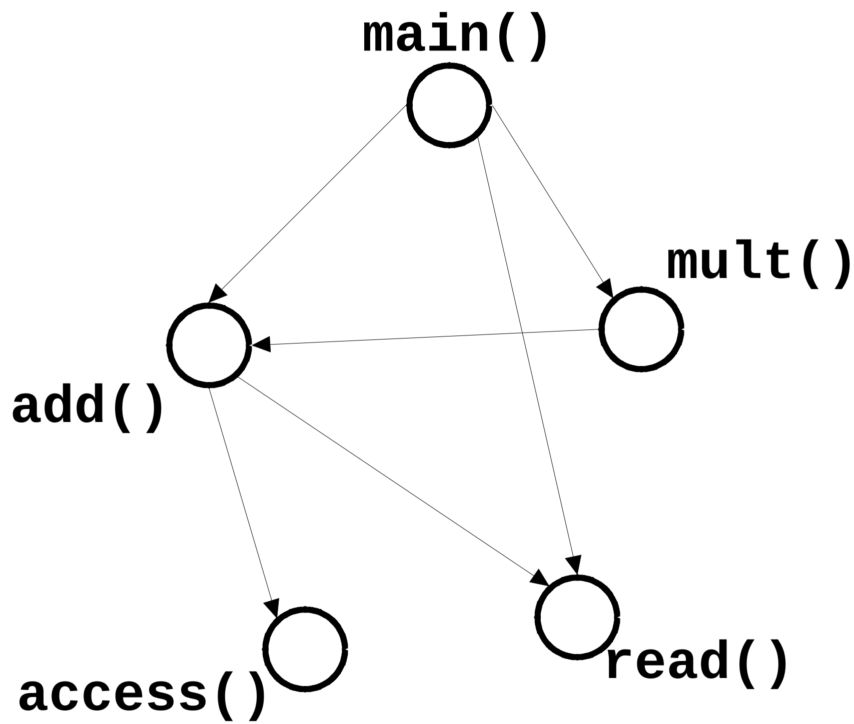
Trees as Graphs

- Every tree is a graph with some restrictions:
  - the tree is ***directed***
  - there are ***no cycles*** (directed or undirected)
  - there is a ***directed path from the root to every node***



Directed Acyclic Graphs (DAGs)

- DAGs are directed graphs with no cycles
- Trees   DAGs   Graphs



Graph Traversals

- A ***Graph Traversals,*** which is similar to a tree traversals in concept, is to visit every vertices of a graph exactly once in some specific order.

- Graph traversals begin with start vertex, and attempt to visit remaining vertices. There are two problems:
  - If the graph is not connected, it may not be possible to reach all vertices.
  - The graph may contains cycles, and a vertex may be reached more than one times.

Graph Traversals

```
void graphTraverse(Graph* G) {
  int v;
  for (v=0; v<G->n(); v++)
    G->setMark(v, UNVISITED);  // Initialize mark bits
  for (v=0; v<G->n(); v++)
    if (G->getMark(v) == UNVISITED)
      doTraverse(G, v);
}
```

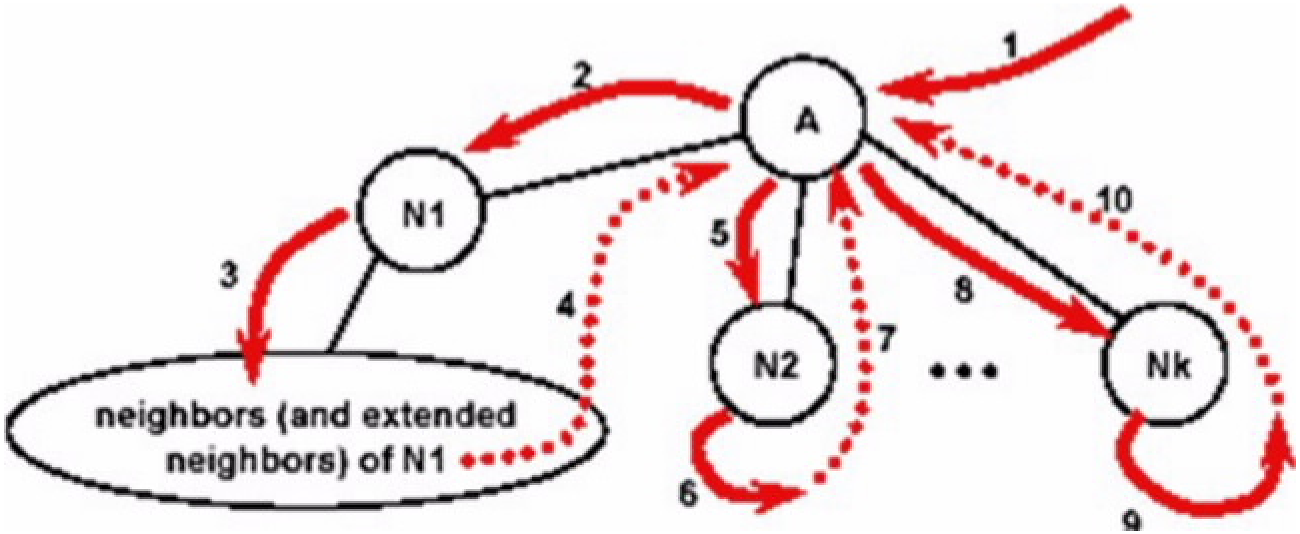
- The order in which the vertices are visited is important. There are two common traversals:
  - Depth-first
  - Breadth-first

Depth-First Search (DFS)

- Assume a particular node has been designated as the starting point. Let A be the node visited and suppose A has neighbors $N_1, N_2, \ldots, N_k$.

- A depth-first search will:
  - visit N_1,then
  - Proceed to traverse all the unvisited neighbors of N_1 , then
  - proceed to traverse the remaining neighbors of A in similar fashion

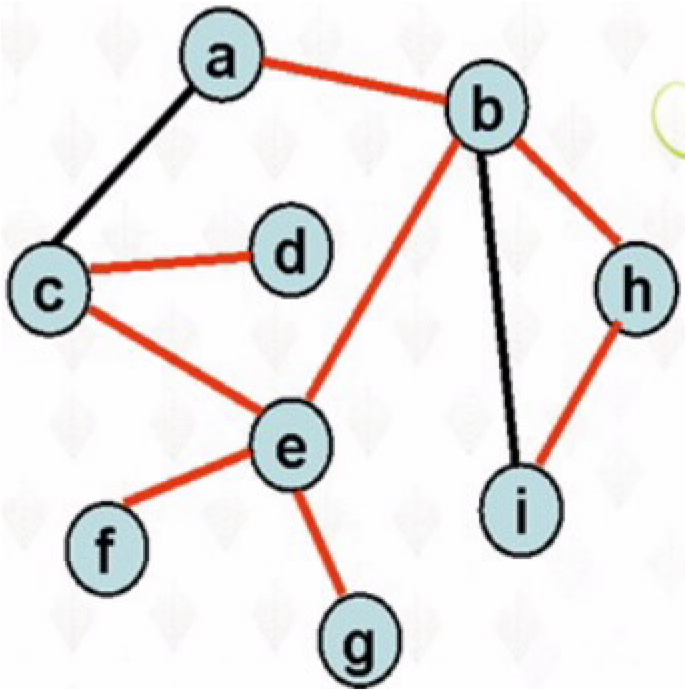
Depth-First Search (DFS)



Depth-First Search (DFS)

- Assume the node labeled **a** has been designated as the starting point, a depth-first traversal would visit the graph nodes in the order:

  a b e c d f g h i

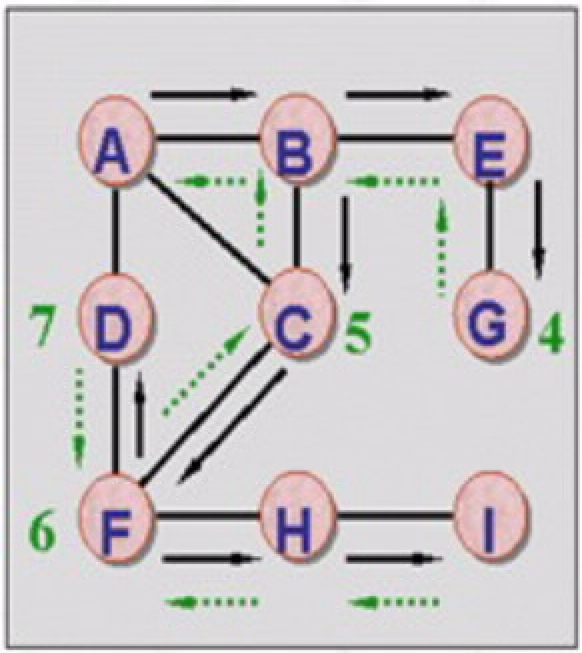


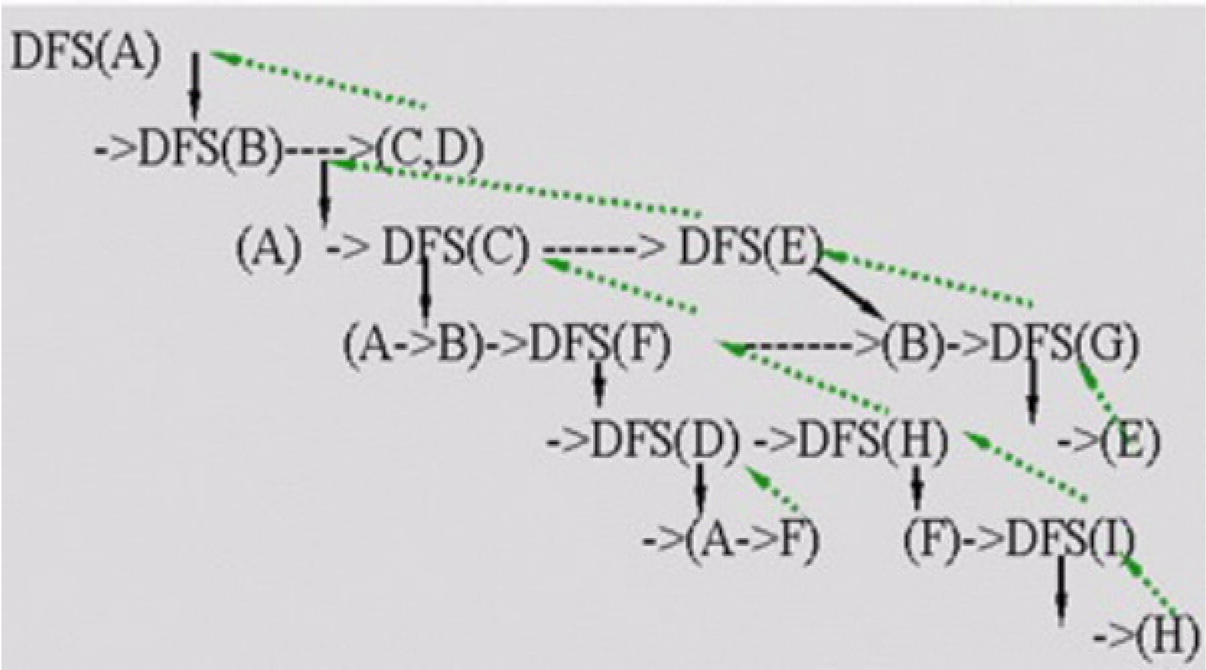
- Note that if the edges taken during the depth-first traversal are marked, they define a tree (not necessarily binary) which includes all the nodes of the graph. Such a tree is a spanning tree for the graph. We call the tree **DFS tree**.

Implementing a DFS

```
void DFS(Graph* G, int v) { // Depth first search
  PreVisit(G, v);           // Take appropriate action
  G->setMark(v, VISITED);
  for (int w=G->first(v); w<G->n(); w = G->next(v,w))
    if (G->getMark(w) == UNVISITED)
      DFS(G, w);
  PostVisit(G, v);          // Take appropriate action
}
```

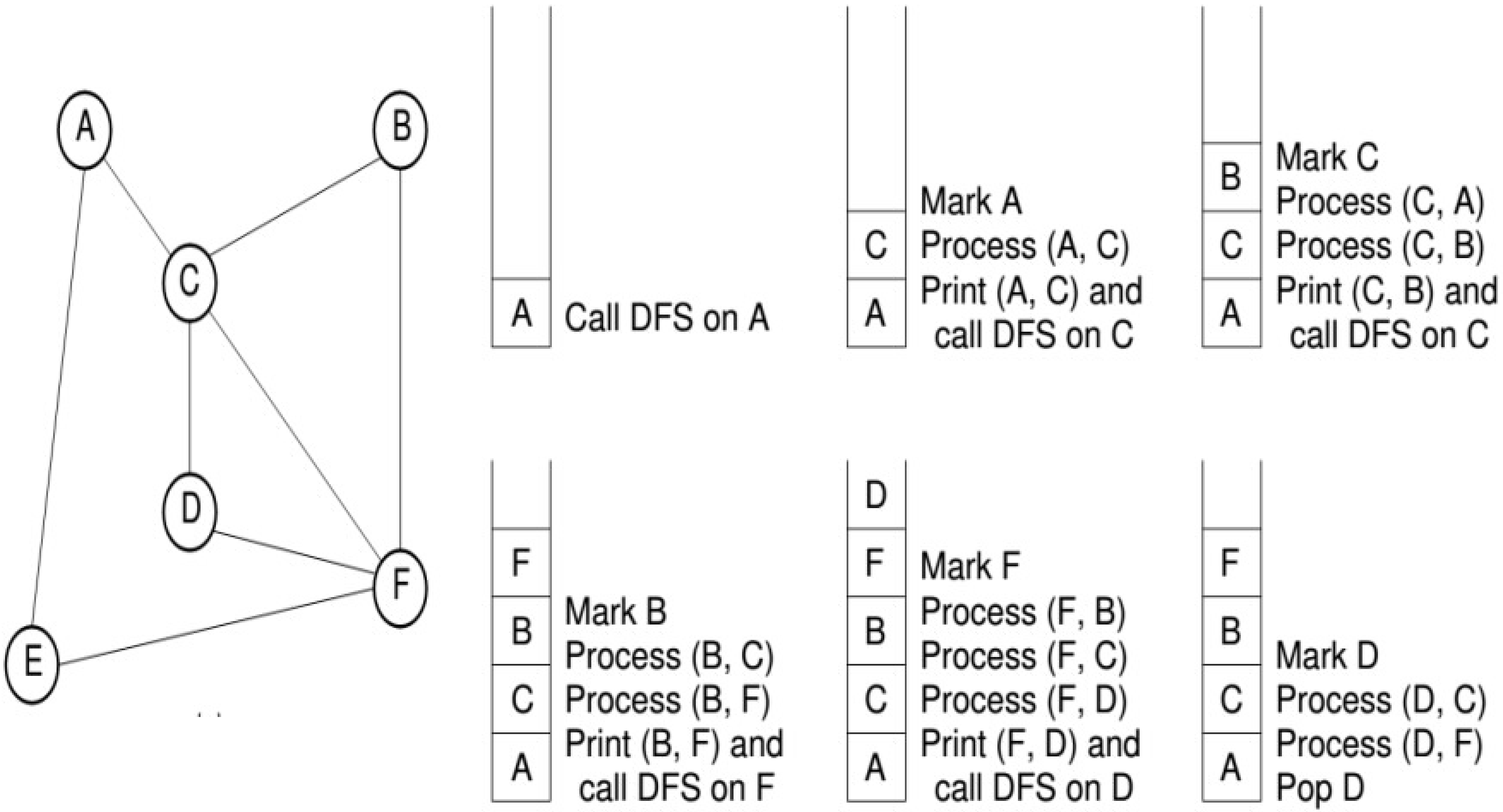
Depth-First Search (DFS)



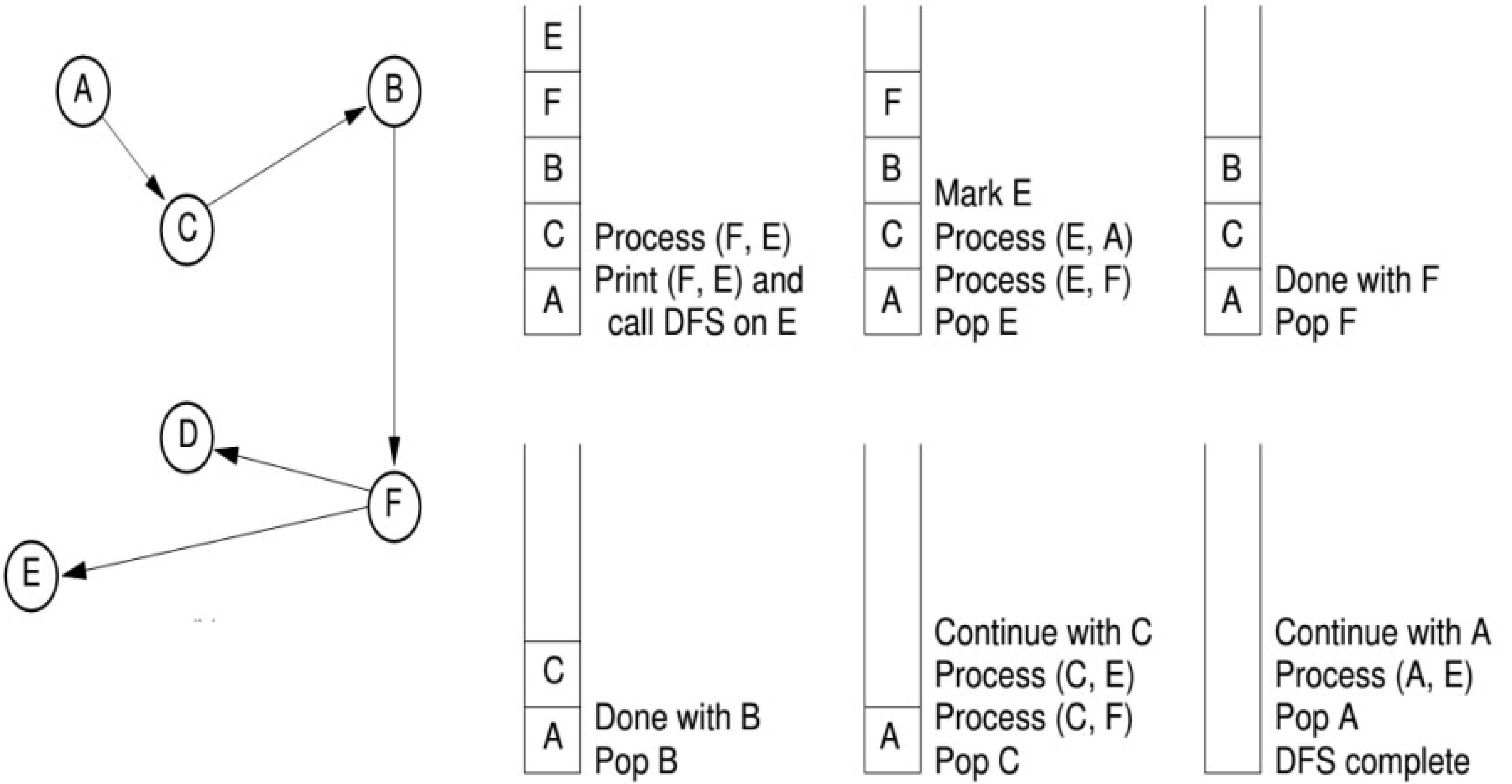


- **The cost of DFS traversal is O(|V|+|E|)**

Depth-First Search (DFS)



Depth-First Search (DFS)

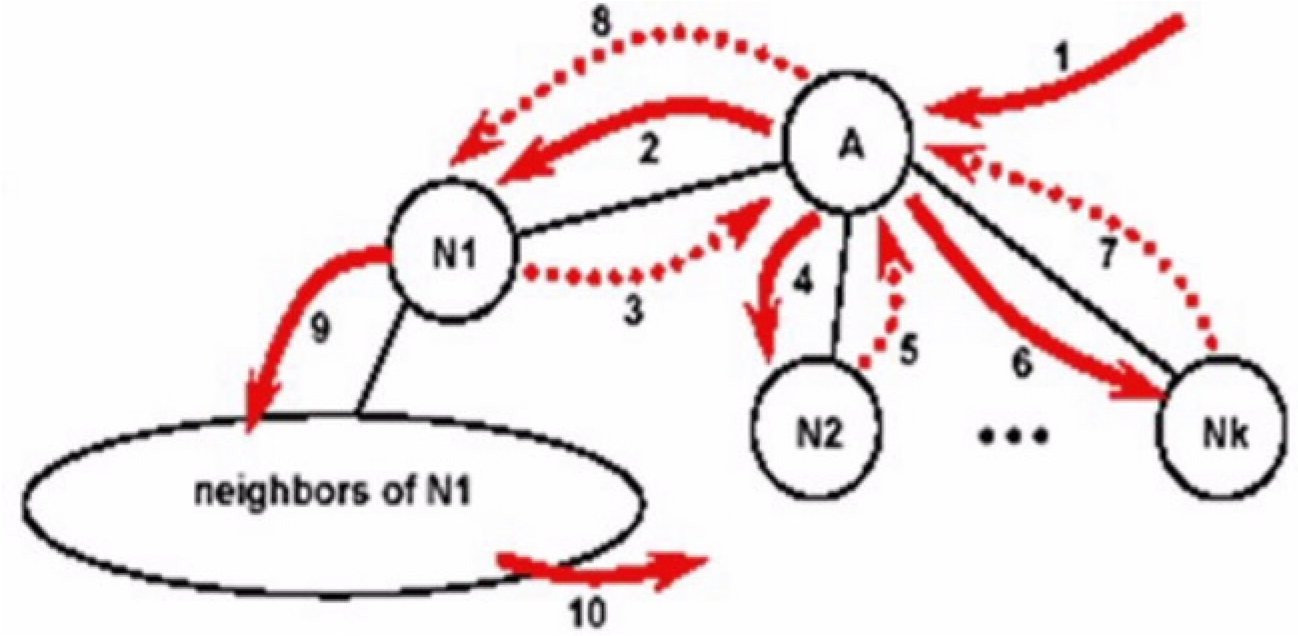


Breadth-First Search (BFS)

- Assume a particular node has been designated as the starting point. Let A be the node visited and suppose A has neighbors $N_1, N_2, \ldots, N_k$.

- A breadth-first search will:
  - visit N_1,then N_2,and so forth through N_k, then
  - Proceed to traverse all the unvisited immediate neighbors of N_1 , then
  - proceed to traverse the immediate neighbors of $N_2, \ldots, N_k$ in similar fashion

Breadth-First Search (BFS)

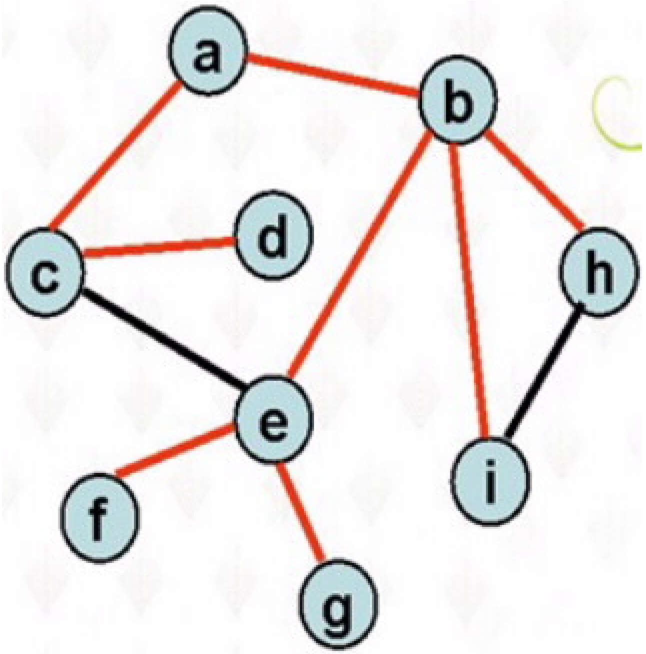


Breadth-First Search (DFS)

- Assume the node labeled **a** has been designated as the starting point, a breadth-first traversal would visit the graph nodes in the order:

  a b c h i d f g

- Note that if the edges taken during the breadth-first traversal are marked, they define a tree (not necessarily binary) which includes all the nodes of the graph. Such a tree is a spanning tree for the graph. We call the tree **BFS tree**. This is usually different from the depth-first spanning tree.

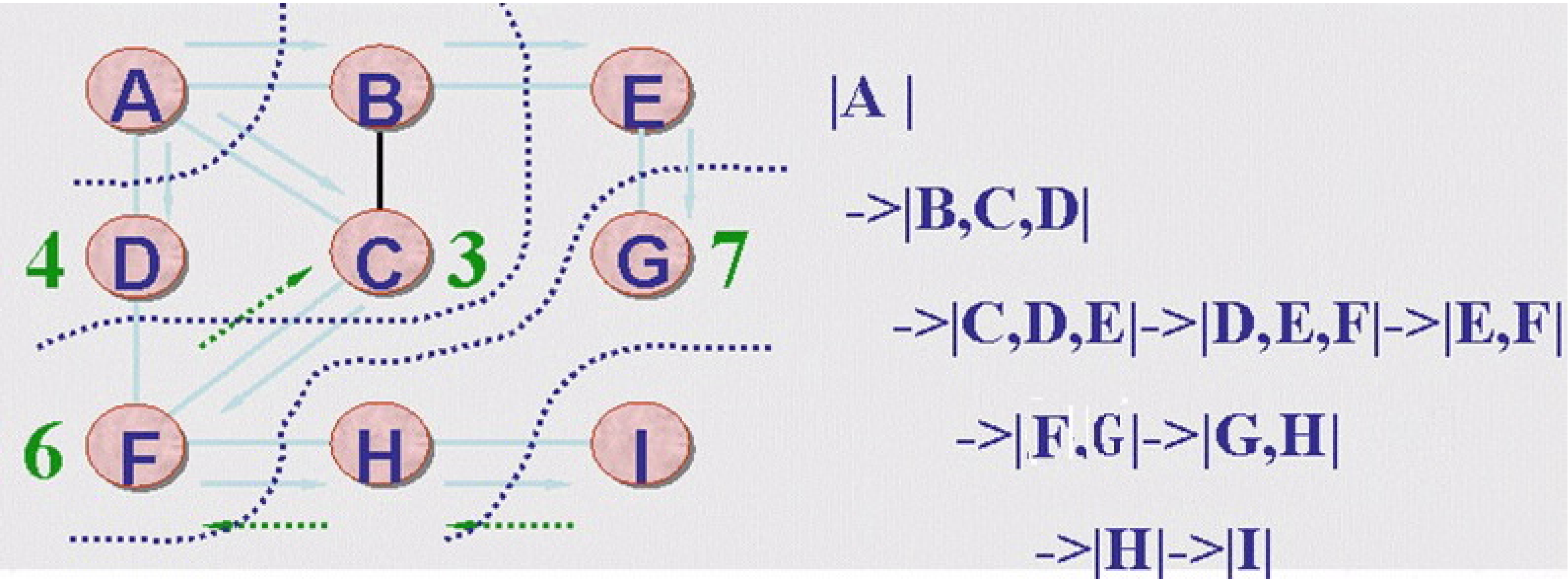


Implementing a BFS

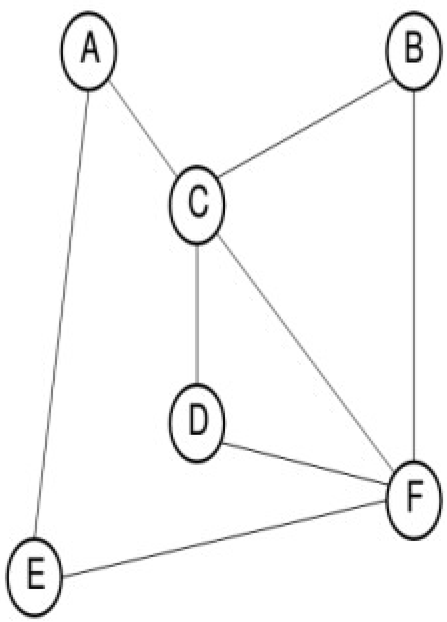
- The breadth-first traversal uses a local queue to organize the graph nodes into the proper order:

```
void BFS(Graph* G, int start, Queue<int>* Q) {
  int v, w;
  Q->enqueue(start);          // Initialize Q
  G->setMark(start, VISITED);
  while (Q->length() != 0) { // Process all vertices on Q
    v = Q->dequeue();
    PreVisit(G, v);           // Take appropriate action
    for (w=G->first(v); w<G->n(); w = G->next(v,w))
      if (G->getMark(w) == UNVISITED) {
        G->setMark(w, VISITED);
        Q->enqueue(w);
      }
  }
}
```

Breadth-First Search (BFS)



Breadth-First Search (BFS)



	A

Initial call to BFS on A.
Mark A and put on the queue.

	C	E

Dequeue A.
Process (A, C).
Mark and enqueue C. Print (A, C)
Process (A, E).
Mark and enqueue E. Print(A, E).

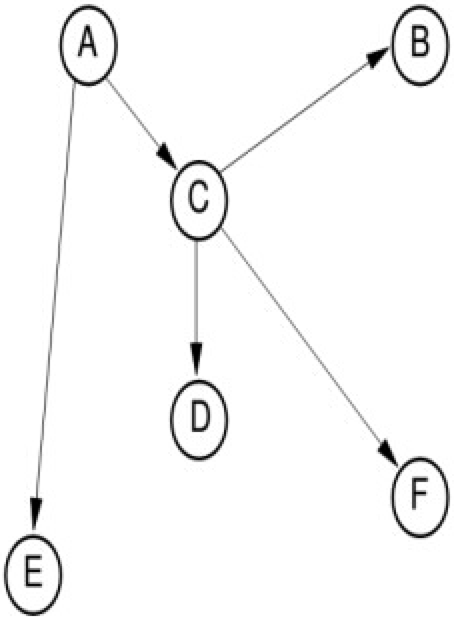
	E	B	D	F

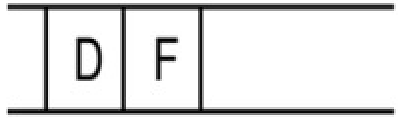
Dequeue C.
Process (C, A). Ignore.
Process (C, B).
Mark and enqueue B. Print (C, B).
Process (C, D).
Mark and enqueue D. Print (C, D).
Process (C, F).
Mark and enqueue F. Print (C, F).

	B	D	F

Dequeue E.
Process (E, A). Ignore.
Process (E, F). Ignore.

Breadth-First Search (BFS)



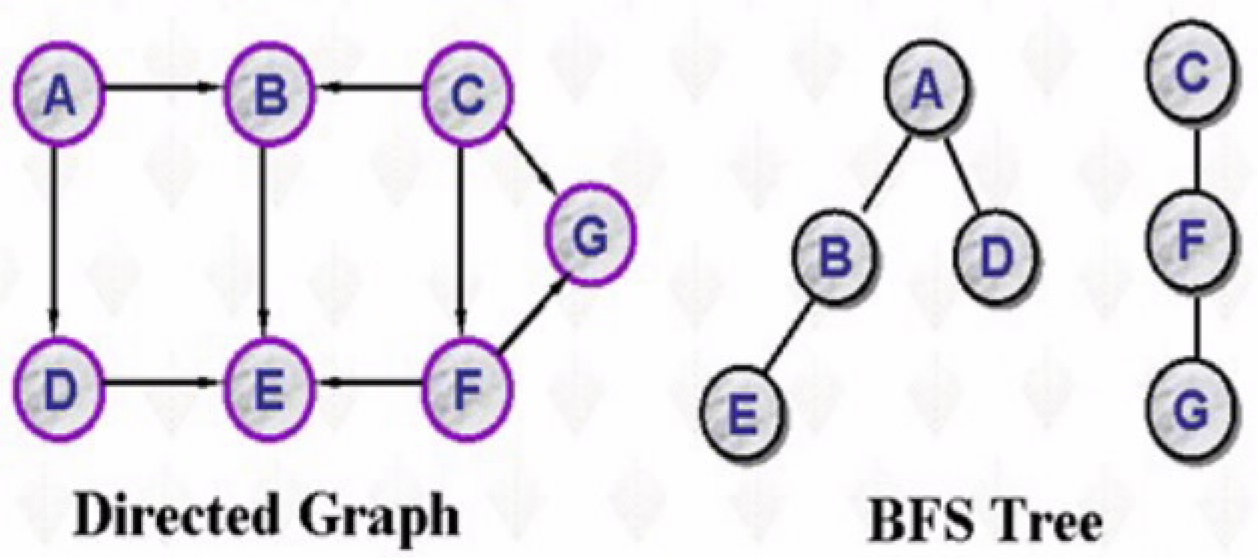


Dequeue B.
Process (B, C). Ignore.
Process (B, F). Ignore.

Dequeue D.
Process (D, C). Ignore.
Process (D, F). Ignore.

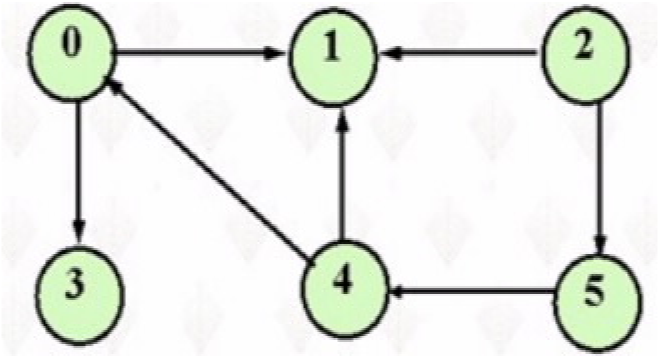
Dequeue F.
Process (F, B). Ignore.
Process (F, C). Ignore.
Process (F, D). Ignore.
BFS is complete.

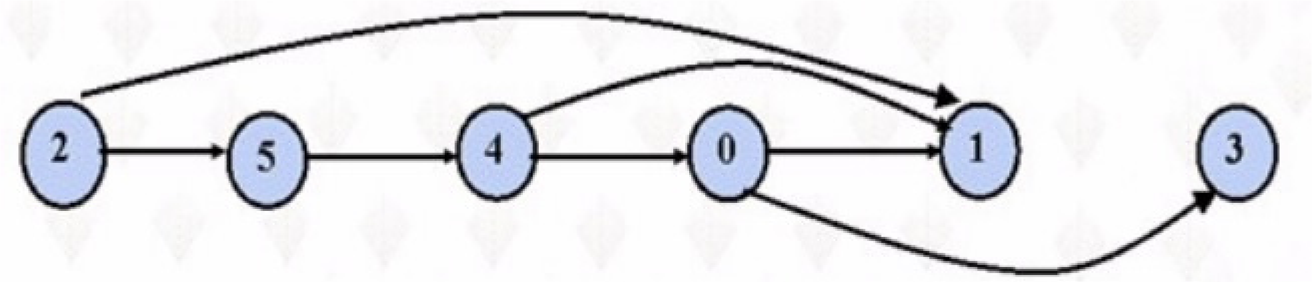
Breadth-First Search (BFS)



Topological Ordering

- Suppose that G is a directed graph which contains no directed cycles. Then a **topological ordering** of the vertices in G is a sequential listing of the vertices such that for any pair of vertices, v and w in G, if <v,w> is an edge in G then v precedes w in the sequential listing.

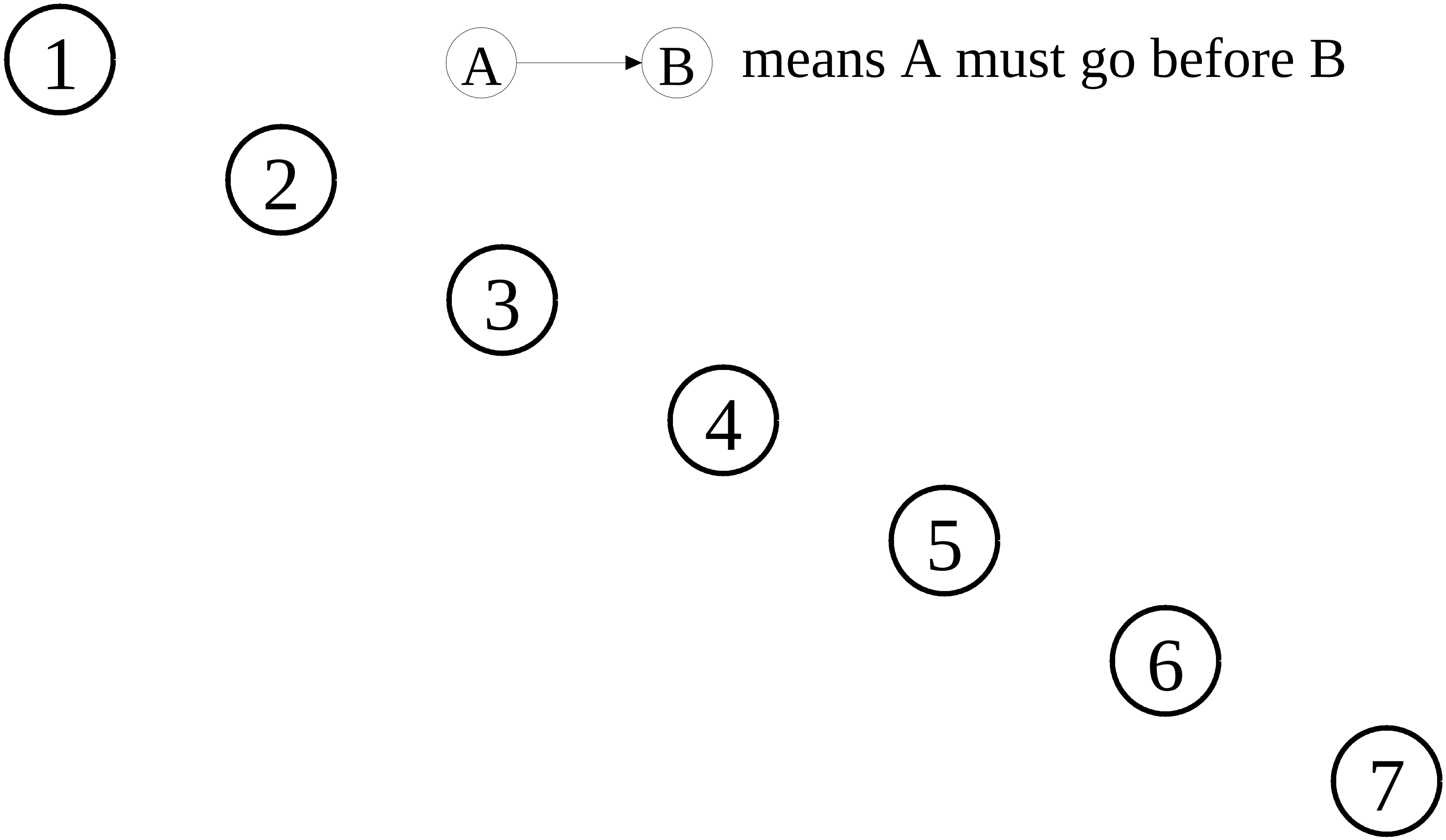




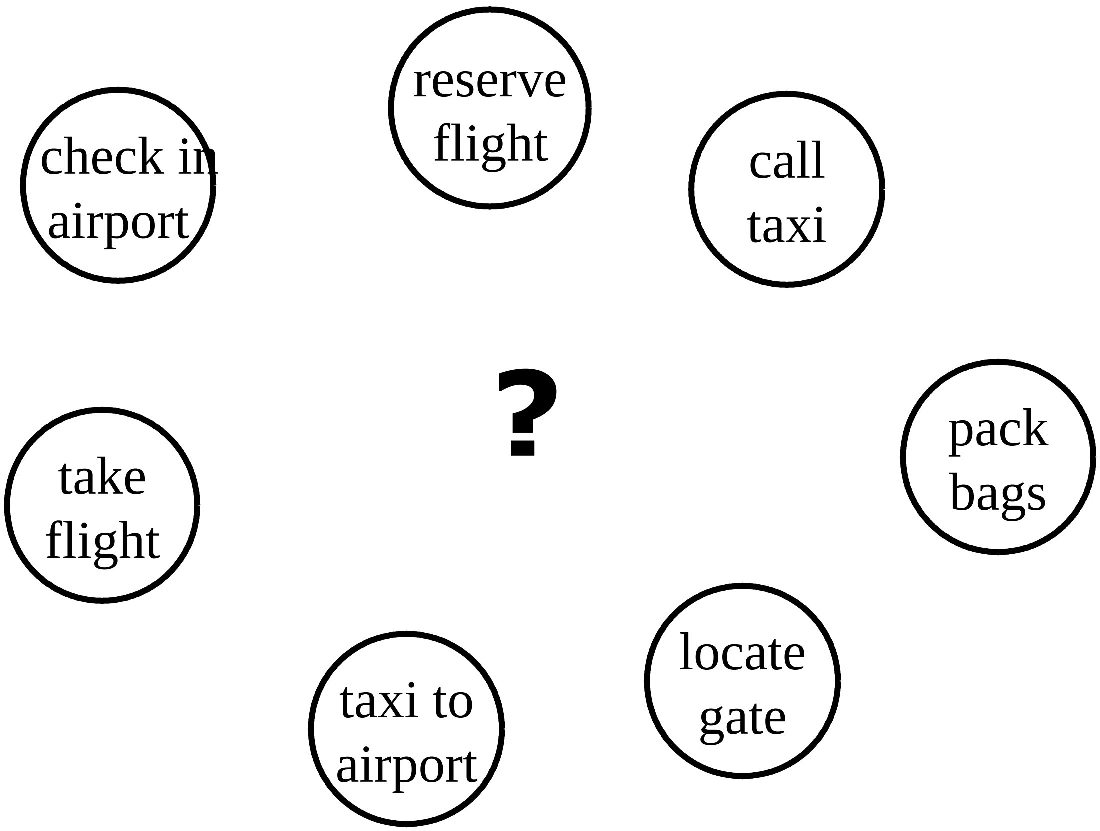
Applications of Topological Ordering

- The process of laying out the vertices of a DAG in a linear order to meet the prerequisite rules is called a topological sort.
- Applications of topological ordering are relatively common…
  - prerequisite relationships among courses
  - glossary of technical terms whose definitions involve dependencies
  - Organization of topics in a book or a course.

Total Order

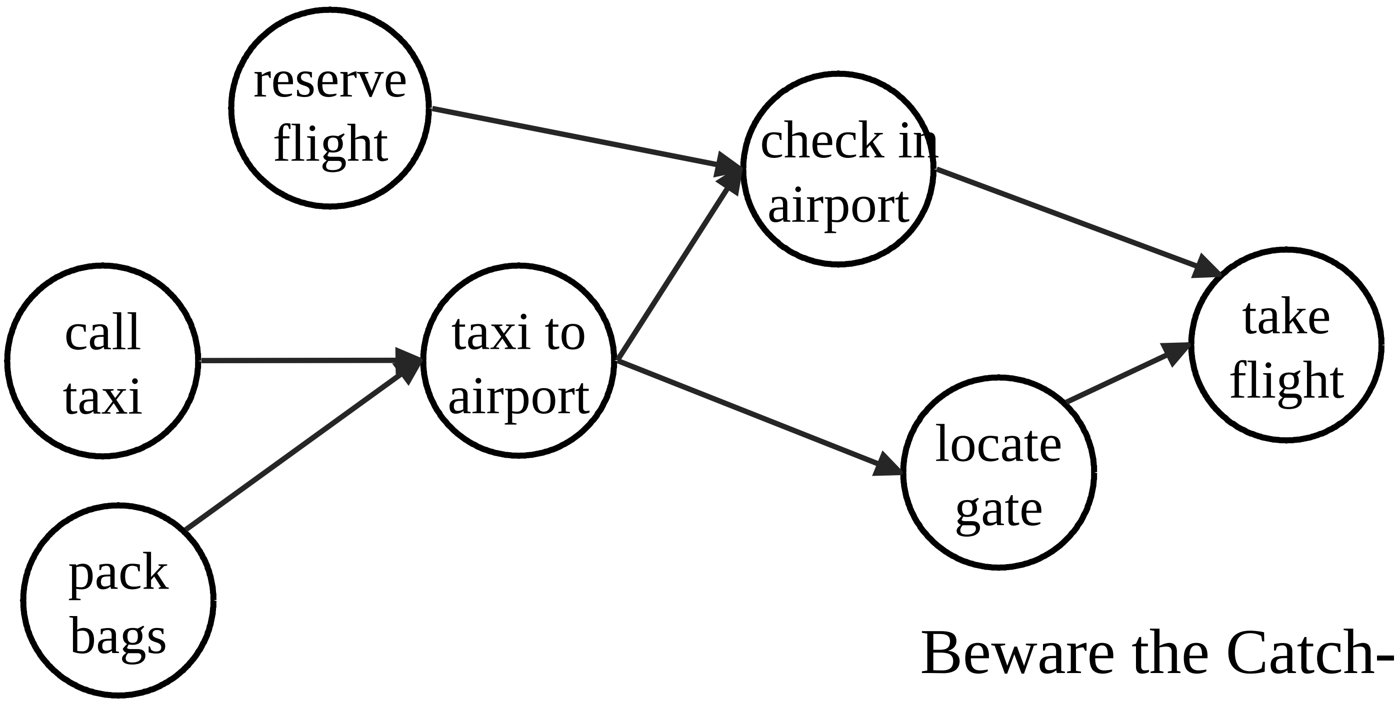


Partial Order: Planning a Trip



Topological Sort

- Given a graph, $\mathbf{G = (V, E)}$, output all the vertices in $\mathbf{V}$ such that no vertex is output before any other vertex with an edge to it.



Beware the Catch-22!

Topo-Sort Take One

- Label each vertex's ***in-degree*** (# of inbound edges)

- While there are vertices remaining
  - Pick a vertex with in-degree of zero and output it
  - Reduce the in-degree of all vertices adjacent to it
  - Remove it from the list of vertices

Topo-Sort Take One

- A topological sort may be found by performing a DFS on the graph. When a vertex is visited, no action is taken (i.e., function PreVisit does nothing). When the recursion pops back to that vertex, function PostVisit prints the vertex. This yields a topological sort in reverse order.

```
void topsort(Graph* G) {    // Topological sort: recursive
  int i;
  for (i=0; i<G->n(); i++) // Initialize Mark array
    G->setMark(i, UNVISITED);
  for (i=0; i<G->n(); i++) // Process all vertices
    if (G->getMark(i) == UNVISITED)
      tophelp(G, i);        // Call recursive helper function
}

void tophelp(Graph* G, int v) { // Process vertex v
  G->setMark(v, VISITED);
  for (int w=G->first(v); w<G->n(); w = G->next(v,w))
    if (G->getMark(w) == UNVISITED)
      tophelp(G, w);
  printout(v);                  // PostVisit for Vertex v
}
```

Topo-Sort Take Two

- Label each vertex's in-degree
- Put all in-degree-zero vertices in a queue

- While there are vertices remaining in the queue
  - Pick a vertex ***v*** with in-degree of zero and output it
  - Reduce the in-degree of all vertices adjacent to ***v***
  - Put any of these with new in-degree zero on the queue
  - Remove ***v*** from the queue

```
// Topological sort: Queue
void topsort(Graph* G, Queue<int>* Q) {
  int Count[G->n()];
  int v, w;
  for (v=0; v<G->n(); v++) Count[v] = 0; // Initialize
  for (v=0; v<G->n(); v++)    // Process every edge
    for (w=G->first(v); w<G->n(); w = G->next(v,w))
       Count[w]++;            // Add to v2's prereq count
  for (v=0; v<G->n(); v++)    // Initialize queue
    if (Count[v] == 0)        // Vertex has no prerequisites
      Q->enqueue(v);
  while (Q->length() != 0) { // Process the vertices
    v = Q->dequeue();
    printout(v);              // PreVisit for "v"
    for (w=G->first(v); w<G->n(); w = G->next(v,w)) {
      Count[w]--;             // One less prerequisite
      if (Count[w] == 0)      // This vertex is now free
        Q->enqueue(w);
    }
  }
}
```

Knowledge Points

- Chapter 11, pp.390-399

Homework

- P410, 11.4-11.8

-End-