05 Graph (2)

College of Computer Science, CQU

Outline

- Graph Traversals
- Topological Sorting

Graph Traversals

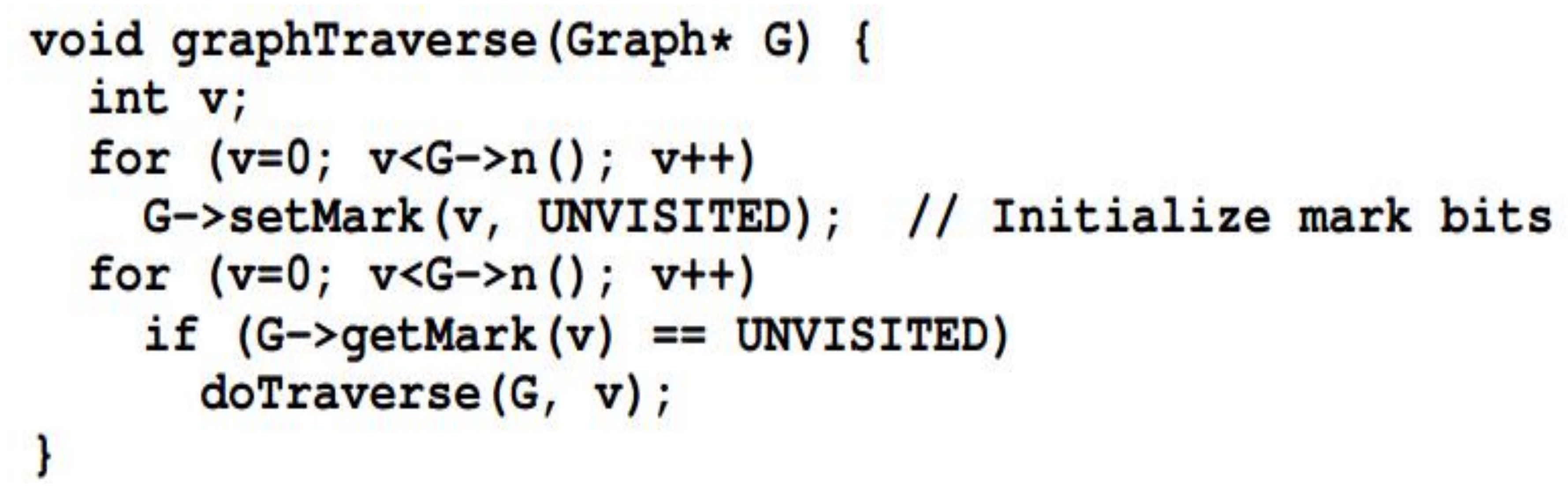
- A *Graph Traversals,* which is similar to a tree traversals in concept, is to visit every vertices of a graph exactly once in some specific order.

- 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

- Graph traversal algorithms can solve both of these problems by maintaining a **mark bit** for each vertex on the graph.
- At the beginning of the algorithm, the mark bit for all vertices is **cleared**.
- The mark bit for a vertex is set when the vertex is first visited during the traversal. If a marked vertex is encountered during traversal, it is not visited a second time.
- Once the traversal algorithm completes, we can check to see if all vertices have been processed by checking the mark bit array.
- If not all vertices are marked, we can continue the traversal from another unmarked vertex.

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);
}
```

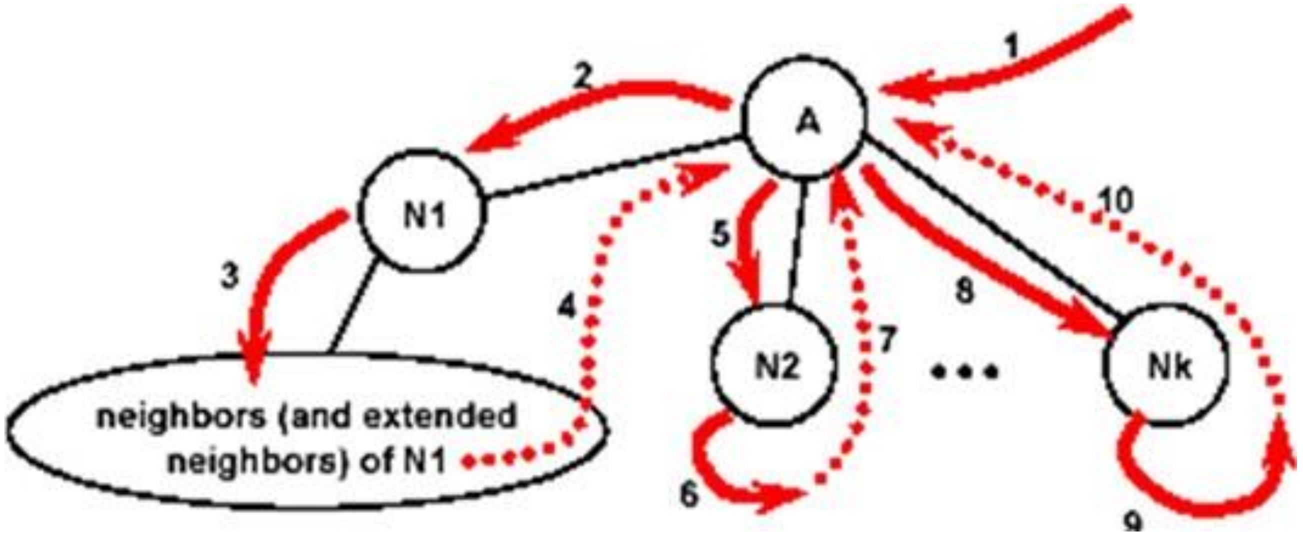
- The order in which the vertices are visited is important. There are two common traversals:
  - Depth-first
  - Breadth-first

Depth-First Search (DFS)

- Assume a particular node has been designated as the starting point. Let A be the last 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

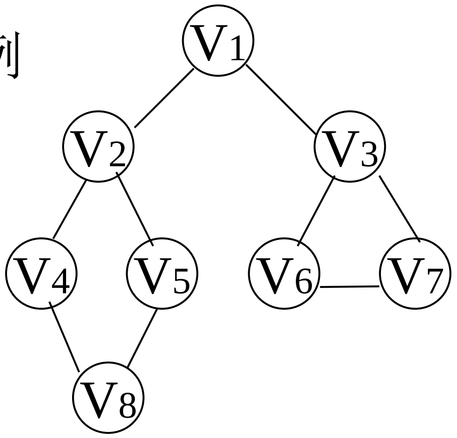
Depth-First Search (DFS)



Depth-First Search (DFS)

❖方法：从图的某一顶点V1出发，访问此顶点；然后依次从V1的未被访问的邻接点出发，深度优先遍历图，直至图中所有和V1相通的顶点都被访问到；若此时图中尚有顶点未被访问，则另选图中一个未被访问的顶点作起点，重复上述过程，直至图中所有顶点都被访问为止。

例



为了在遍历过程中便于区分顶点是否已被访问，附设访问标志数组mark[0..n-1]，，其初值为UNVISITED ，一旦某个顶点被访问，则其相应的分量置为VISITED。

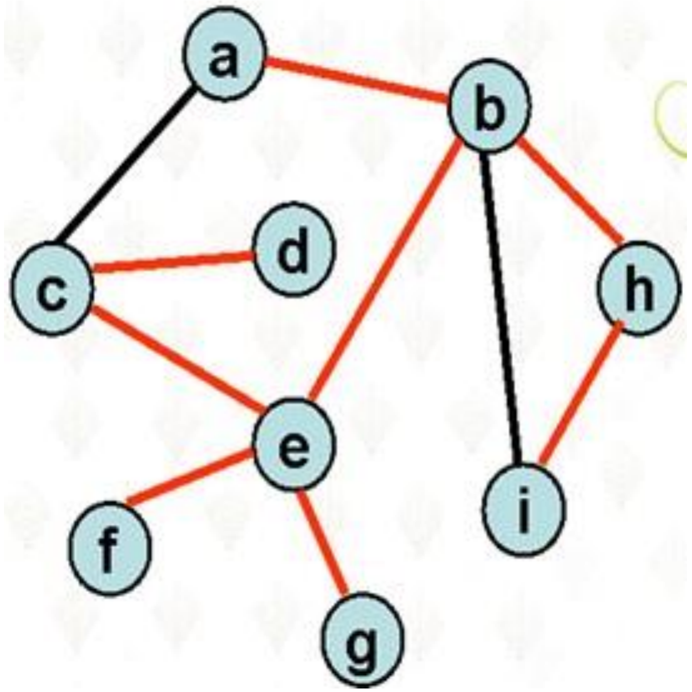
深度遍历：V1⇒ V2 ⇒ V4⇒ V8⇒ V5 ⇒ V3 ⇒ V6 ⇒ V7

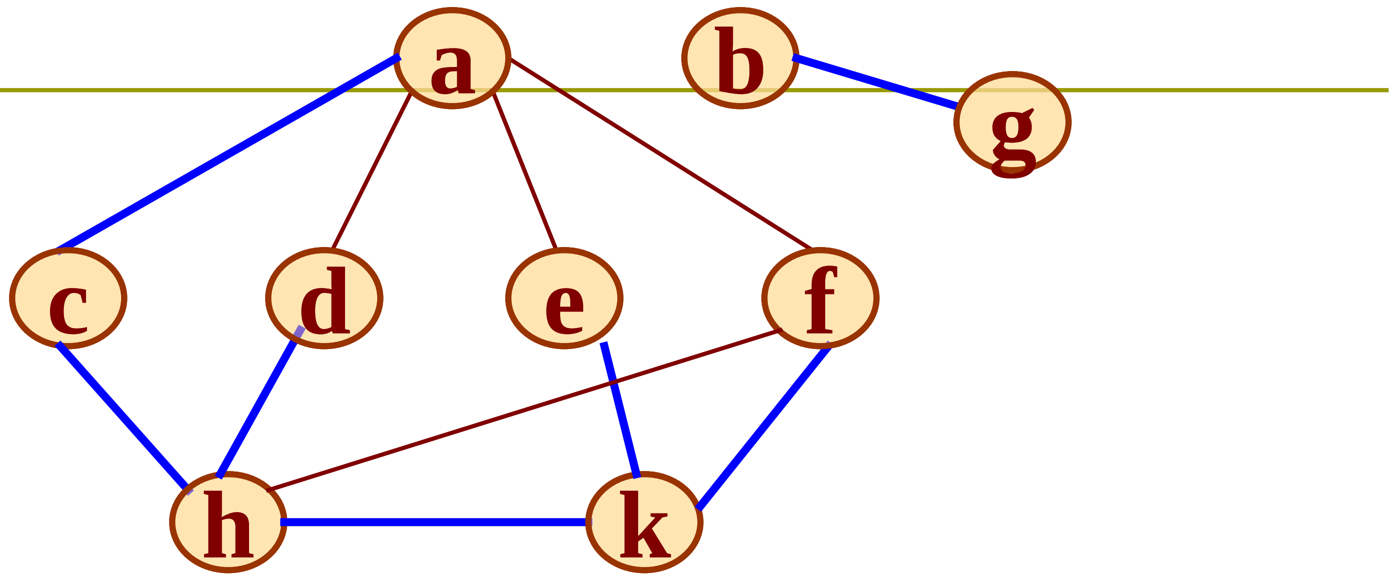
Depth-First Search (DFS)

- 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**.



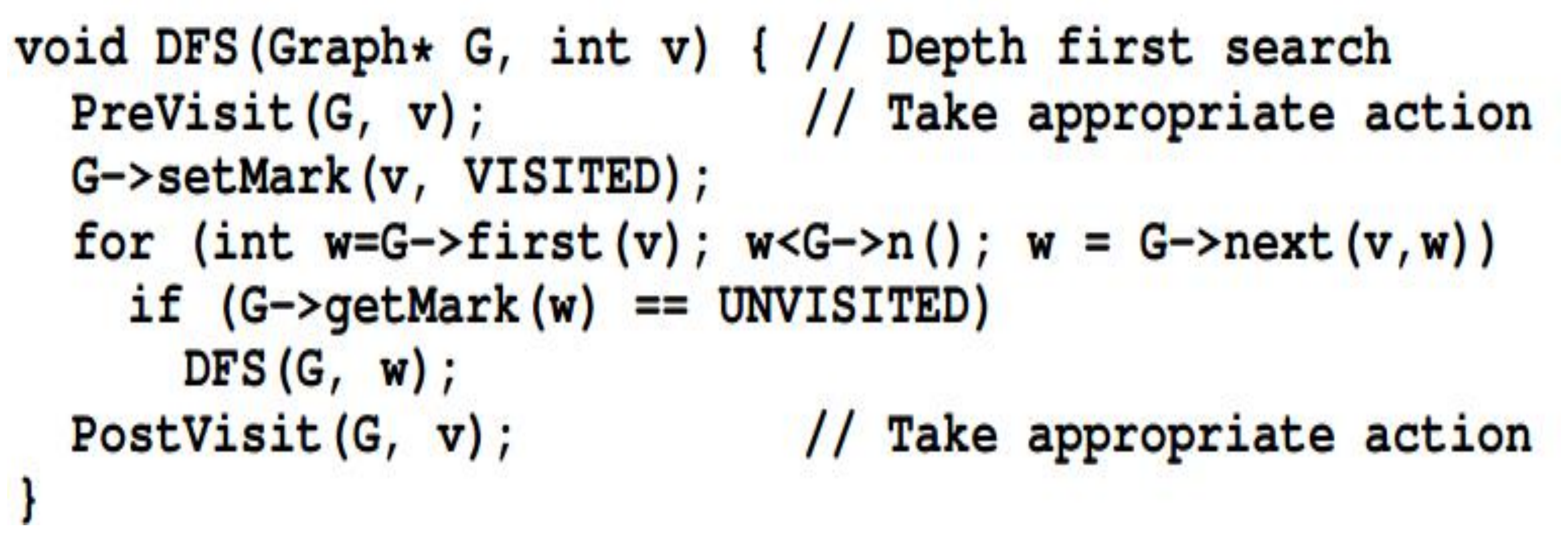


访问标志：

0	1	2	3	4	5	6	7	8
T	T	T	T	T	T	T	T	T

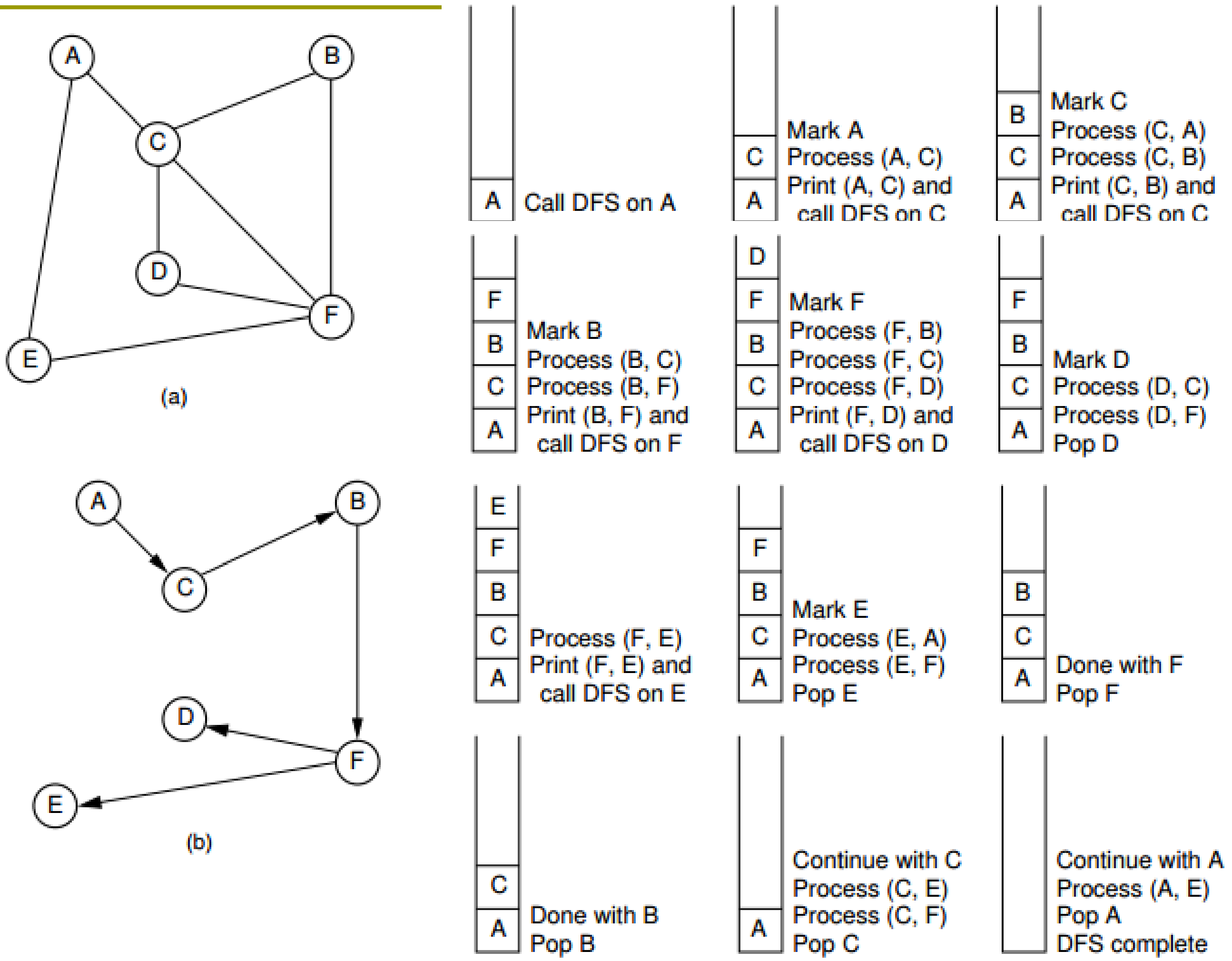
访问次序： a c h d k f e b g

Implementing a DFS

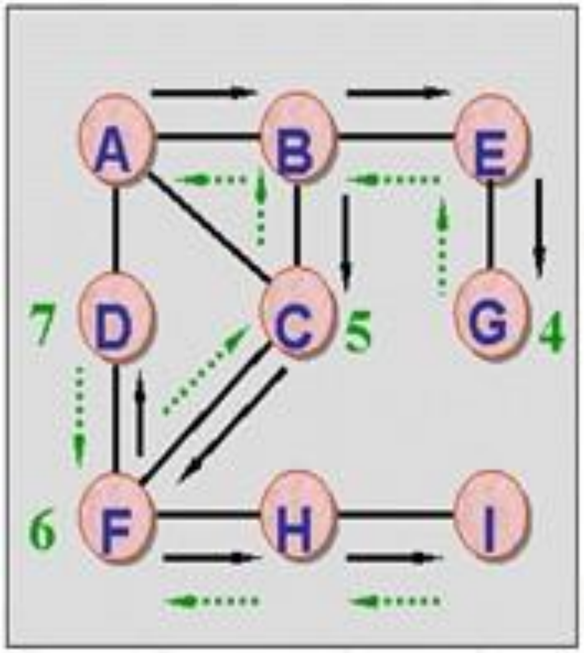


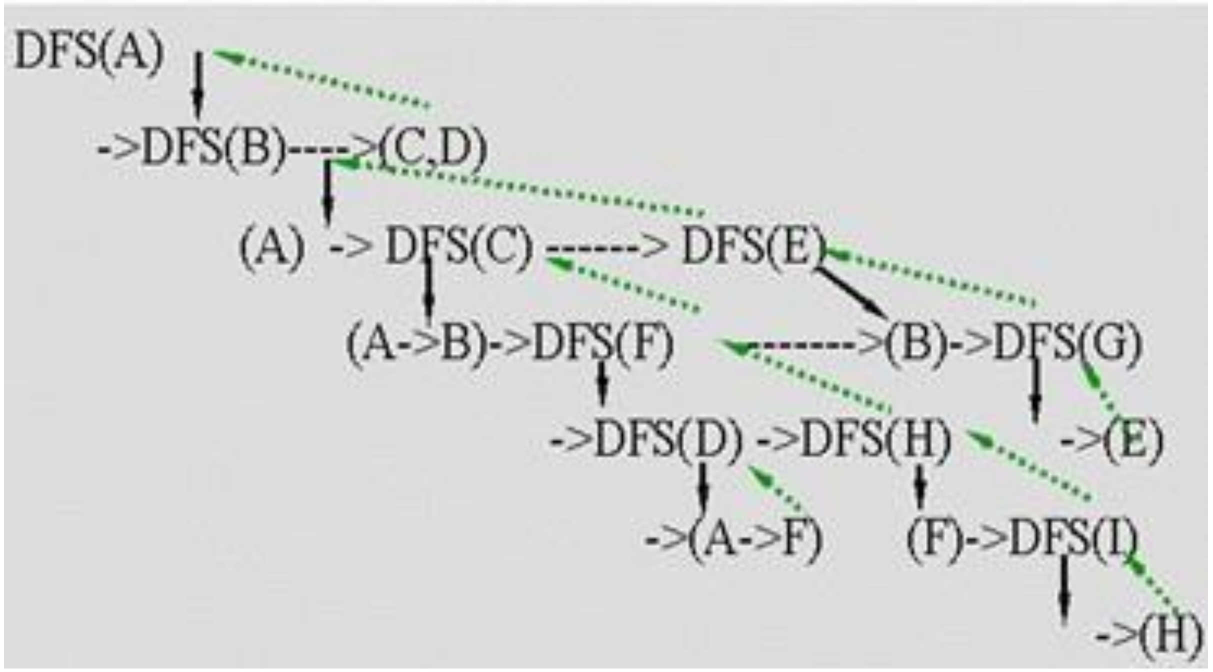
```
void DFS(Graph* G, int v) { // Depth first search
  PreVisit(G, v);            // Take appropriate action
  G->setMark(v, VISITED);
  for (int w=G->first(v); w<G->n(); w = G->next(v,w))
    if (G->getMark(w) == UNVISITED)
      DFS(G, w);
  PostVisit(G, v);           // Take appropriate action
}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

Implementing a DFS

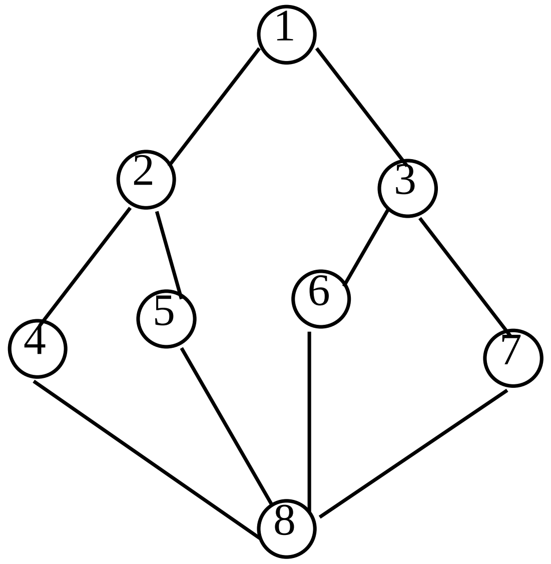


Depth-First Search (DFS)





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



无向图 G7

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

无向图 G7 的邻接矩阵

无向图G7，可以描述从顶点1出发的深度优先搜索遍历过程，示意图见下图，其中实线表示下一层递归调用，虚线表示递归调用的返回。

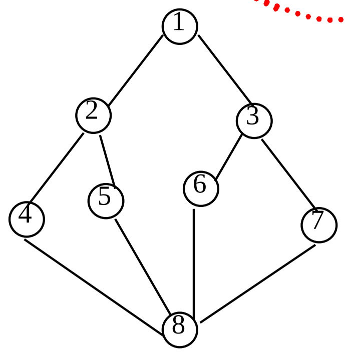
从图G7中，可以得到从顶点1的遍历结果为1, 2, 4, 8, 5, 6, 3, 7。同样可以分析出从其它顶点出发的遍历结果。

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

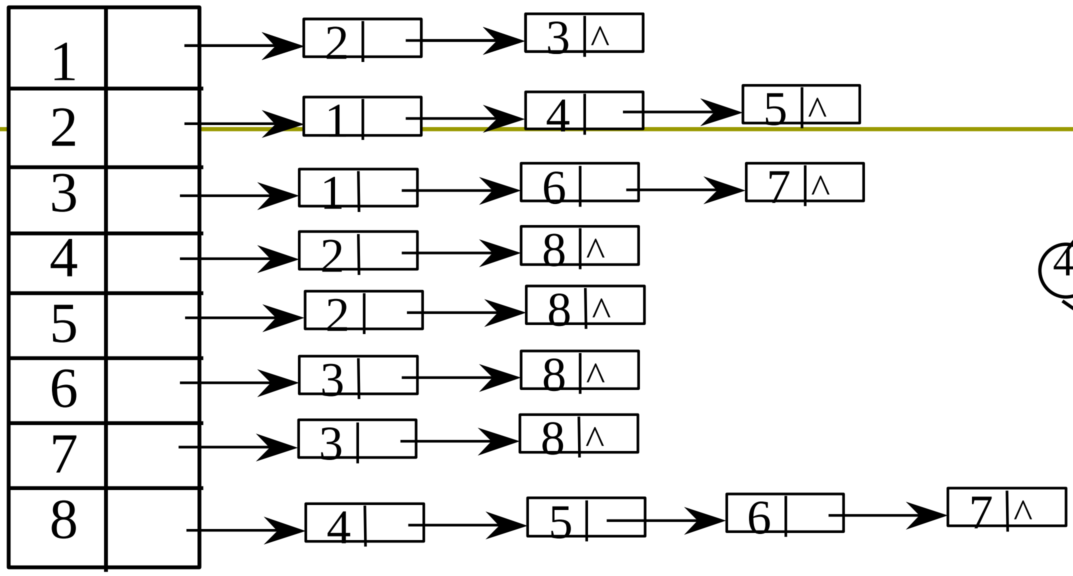
无向图 G7 的邻接矩阵

$$\begin{tikzcd}
\text{DFS(1)} \arrow[r] & \text{DFS(2)} \arrow[r] \arrow[l, dotted, bend left] & \text{DFS(4)} \arrow[r] \arrow[l, dotted, bend left] & \text{DFS(8)} \arrow[r] \arrow[l, dotted, bend left] \arrow[d] & \text{DFS(5)} \arrow[l, dotted, bend left] & & \\
 & & & \text{DFS(6)} \arrow[r] \arrow[u, dotted, bend left] & \text{DFS(3)} \arrow[r] \arrow[l, dotted, bend left] & \text{DFS(7)} \arrow[l, dotted, bend left] &
\end{tikzcd}$$

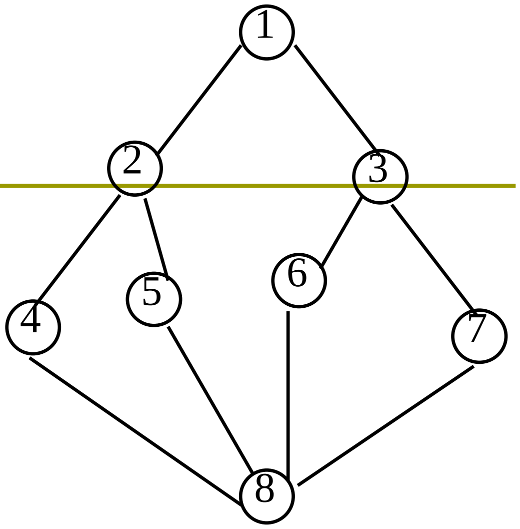
邻接矩阵深度优先搜索示意图



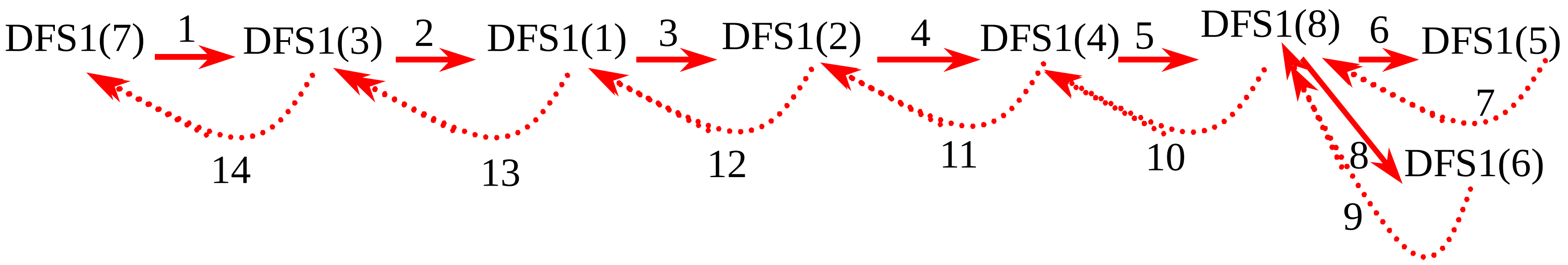
无向图 G7



G7 的邻接表



无向图 G7



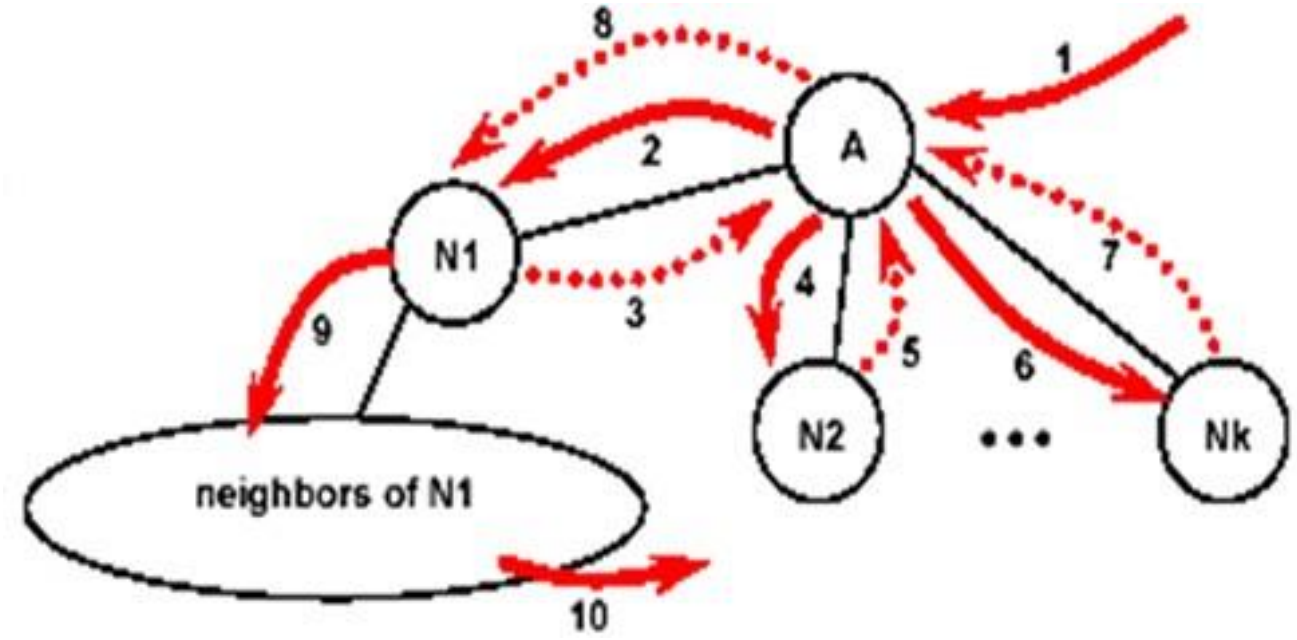
邻接表深度优先搜索示意图

Breadth-First Search (BFS)

- Assume a particular node has been designated as the starting point. Let A be the last 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

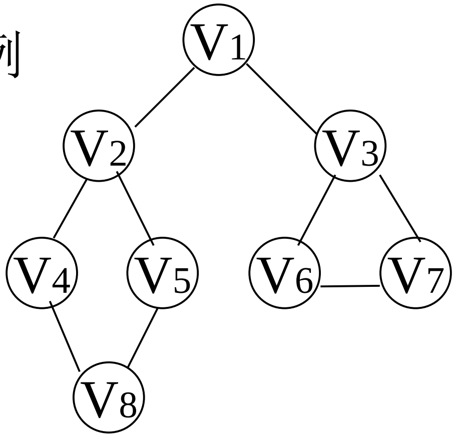
Breadth-First Search (BFS)



Breadth-First Search (BFS)

❖方法：从图的某一顶点V1出发，访问此顶点后，依次访问V1的各个未曾访问过的邻接点；然后分别从这些邻接点出发，广度优先遍历图，直至图中所有已被访问的顶点的邻接点都被访问到；若此时图中尚有顶点未被访问，则另选图中一个未被访问的顶点作起点，重复上述过程，直至图中所有顶点都被访问为止

例



和深度优先搜索类似，在遍历的过程中也需要一个访问标志数组。并且，为了顺次访问路径长度为2、3、…的顶点，需附设队列以存储已被访问的路径长度为1、2、… 的顶点。

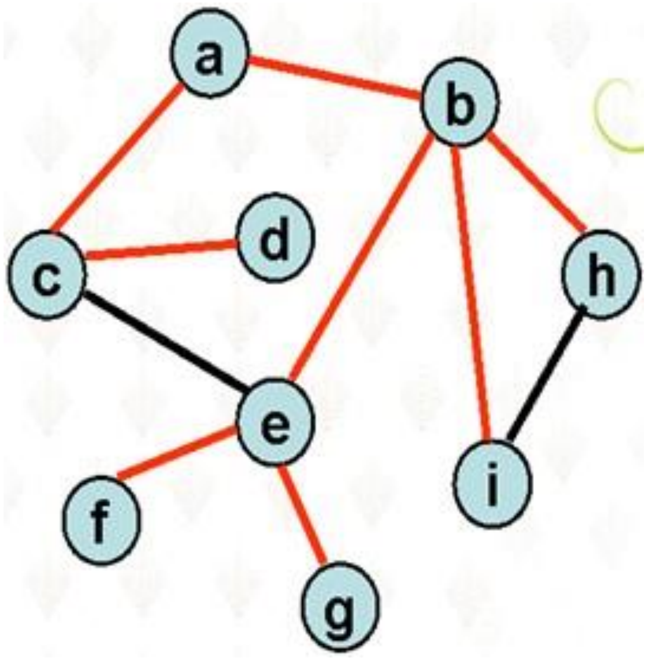
广度遍历：V1⇒ V2 ⇒ V3⇒ V4⇒ V5⇒ V6⇒V7 ⇒ V8

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:

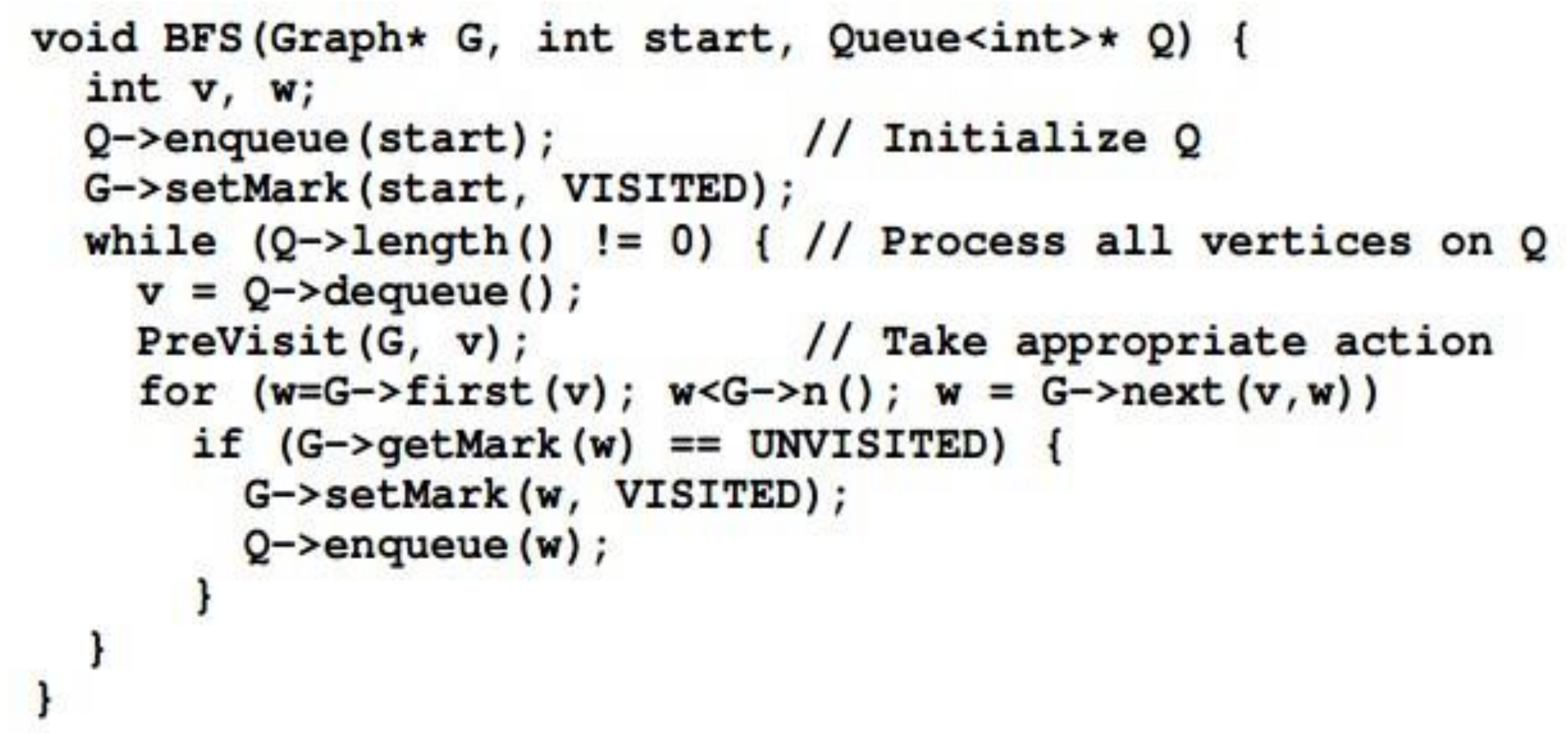
  a b c h i e d f g

- 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.



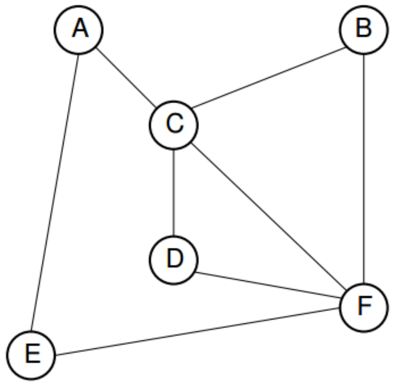
Implementing a BFS

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

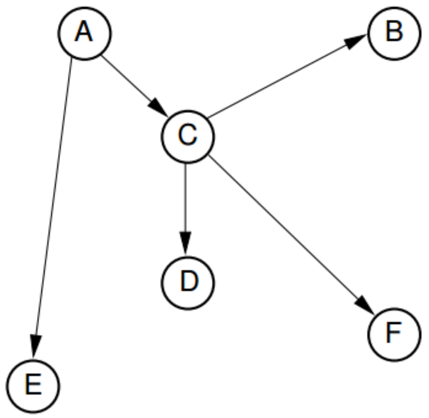


```
void BFS(Graph* G, int start, Queue<int>* Q) {
  int v, w;
  Q->enqueue(start);          // Initialize Q
  G->setMark(start, VISITED);
  while (Q->length() != 0) { // Process all vertices on Q
    v = Q->dequeue();
    PreVisit(G, v);           // Take appropriate action
    for (w=G->first(v); w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

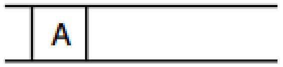
Breadth-First Search (DFS)



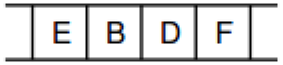
(a)



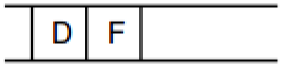
(b)



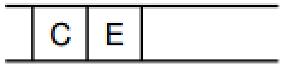
Initial call to BFS on A.
Mark A and put on the queue.



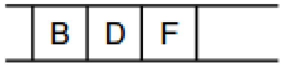
Dequeue C.
Process (C, A). Ignore.
Process (C, B).
Mark and enqueue B. Print (C, B).
Process (C, D).
Mark and enqueue D. Print (C, D).
Process (C, F).
Mark and enqueue F. Print (C, F).



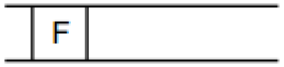
Dequeue B.
Process (B, C). Ignore.
Process (B, F). Ignore.



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



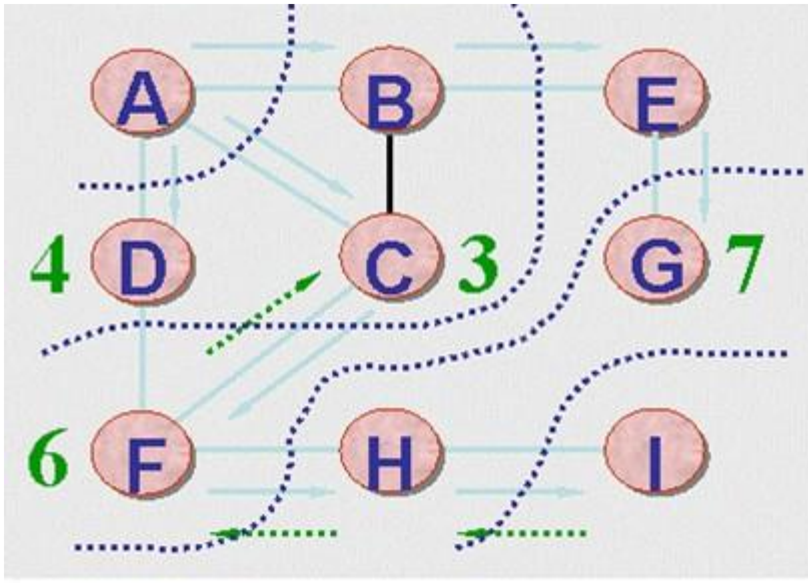
Dequeue E.
Process (E, A). Ignore.
Process (E, F). Ignore.



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)



|A |

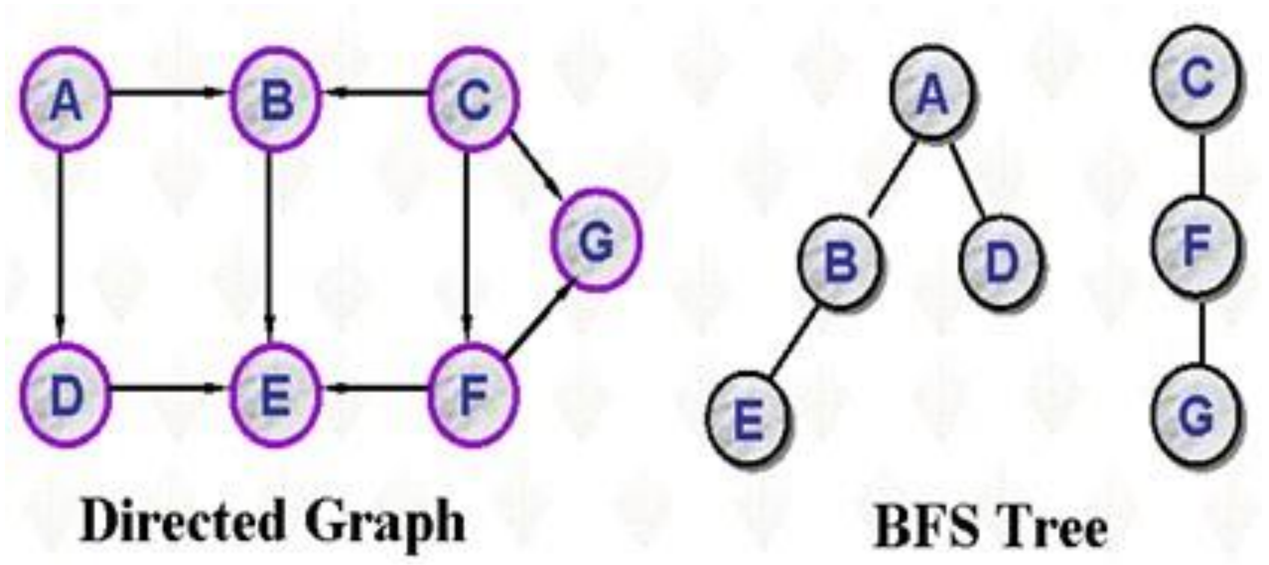
->|B,C,D|

->|C,D,E|->|D,E,F|->|E,F|

->|F,G|->|G,H|

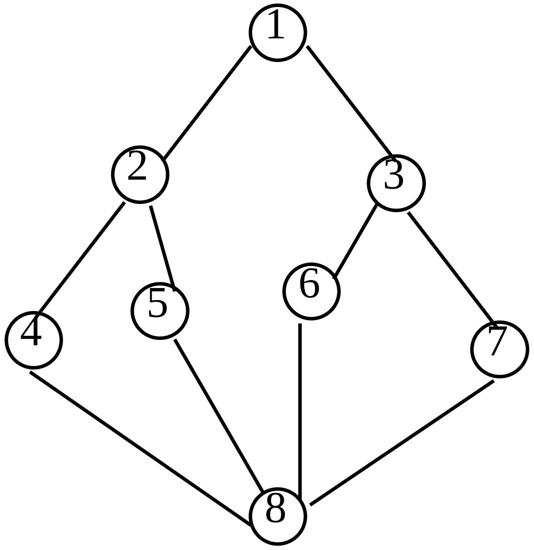
->|H|->|I|

Breadth-First Search (BFS)



(1) 用邻接矩阵实现图的广度优先搜索遍历

仍以无向图**G7**及其的邻接矩阵来说明对无向图**G7**的遍历过程



无向图 G7

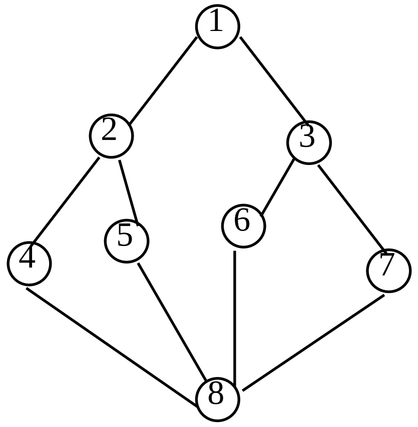
$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

无向图 G7 的邻接矩阵

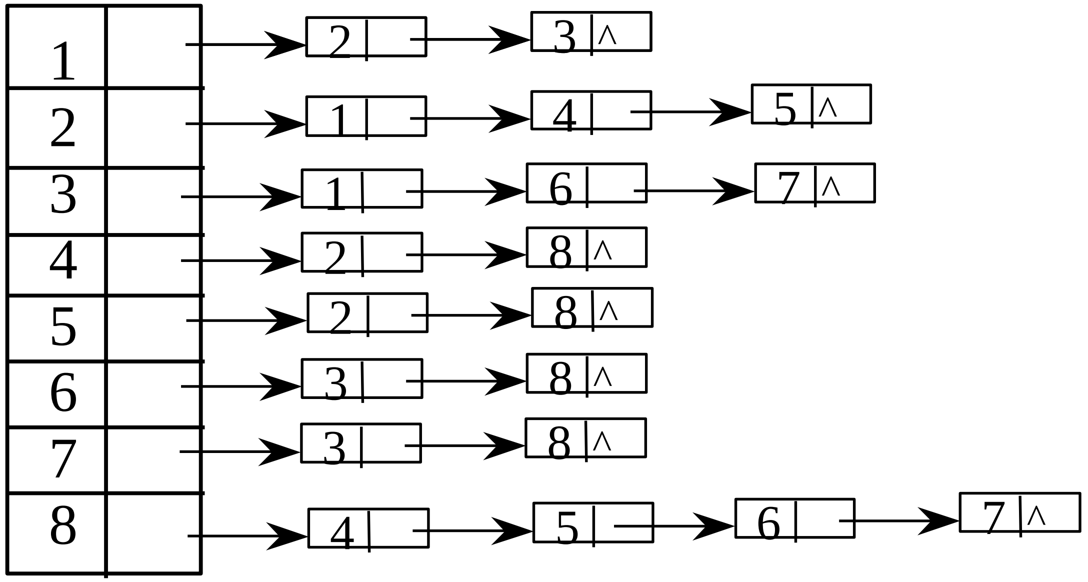
根据该算法用及图G7中的邻接矩阵，可以得到图G7的无向图G7 的广度优先搜索序列，若从顶点1 出发，广度优先搜索序列为：1，2，3，4，5， 6，7，8。若从顶点3出发，广度优先搜索序列为：3， 1， 6， 7，2， 8， 4， 5,从其它点出发的广度优先搜索序列可根据同样类似方法分析。

（2）用邻接表实现图的广序优先搜索遍历

仍以无向图G7及其邻接表来说明邻接表上实现广度优先搜索遍历的过程



无向图 G7



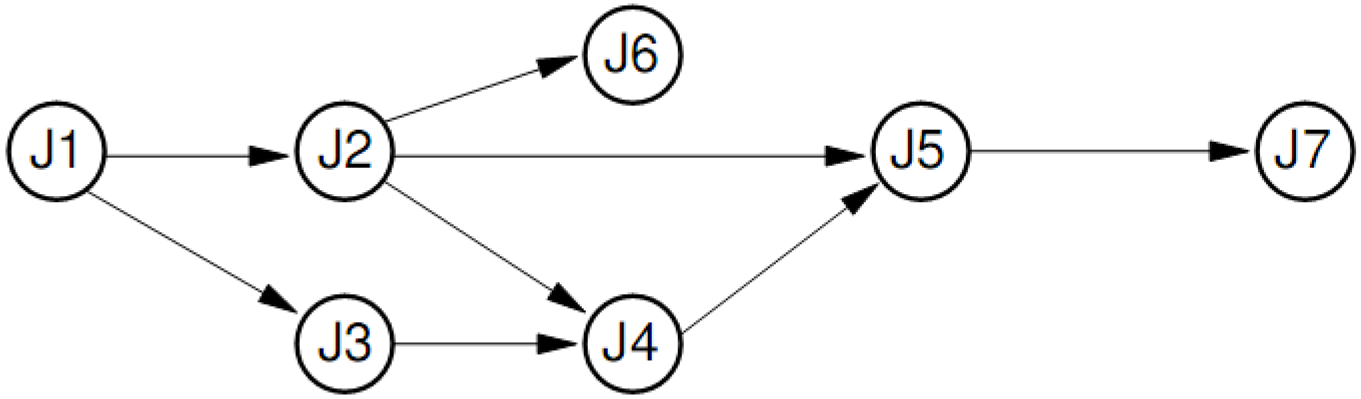
G7 的邻接表

根据该算法，可以得到图G7的广度优先搜索序列，若从顶点1出发，广度优先搜索序列为：1，2，3，4，5，6，7，8，若从顶点7出发，广度优先搜索序列为：7，3，8，1，6，4，5，2，从其它顶点出发的广度优先搜索序列可根据同样类似方法分析。

Topological Sort

- Assume that we need to schedule a series of tasks, such as classes or construction jobs, where we cannot start one task until after its prerequisites are completed.
- We wish to organize the tasks into a linear order that allows us to complete them one at a time without violating any rerequisites.
- modeling the problem using a **DAG**.
- **topological sort :** the process of laying out the vertices of a DAG in a linear order to meet the prerequisite rules.

Topological Sort

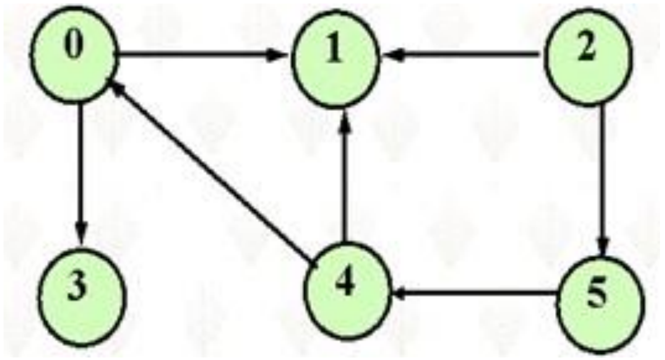


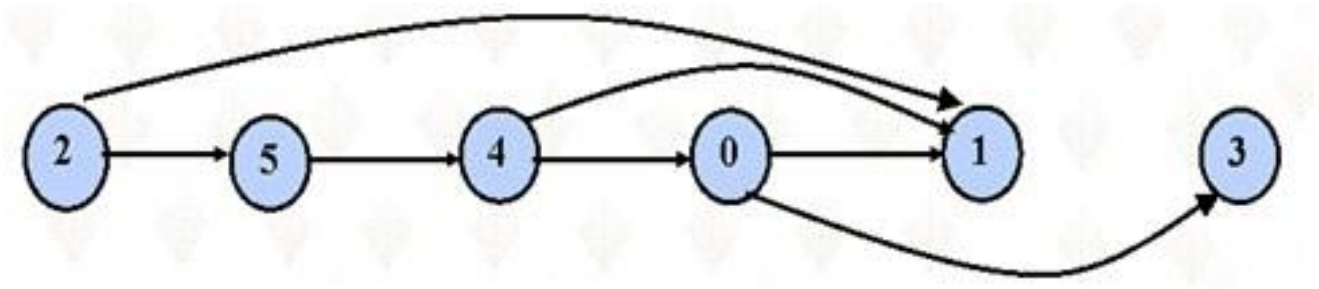
- An acceptable topological sort for this example is

J1, J2,J3, J4, J5, J6, J7.

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 by topological sort





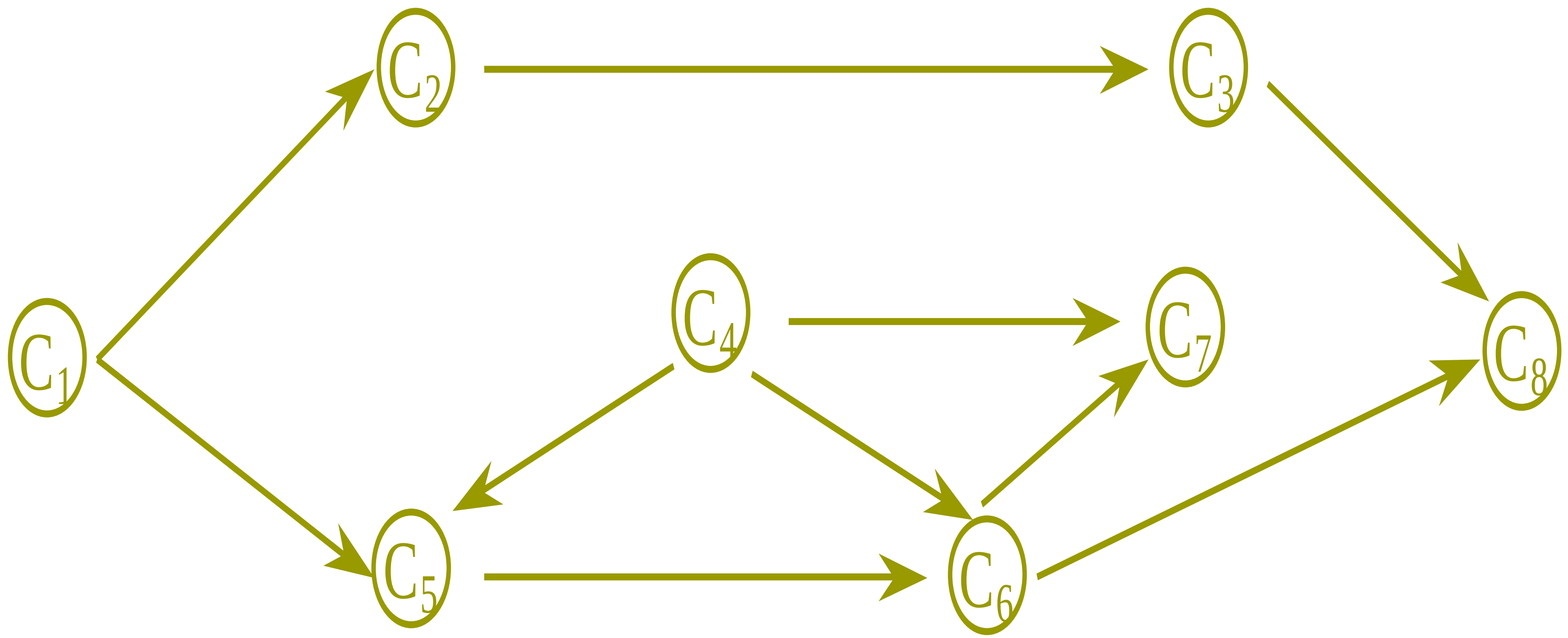
Applications of Topological Ordering

- 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.

某专业课程设置

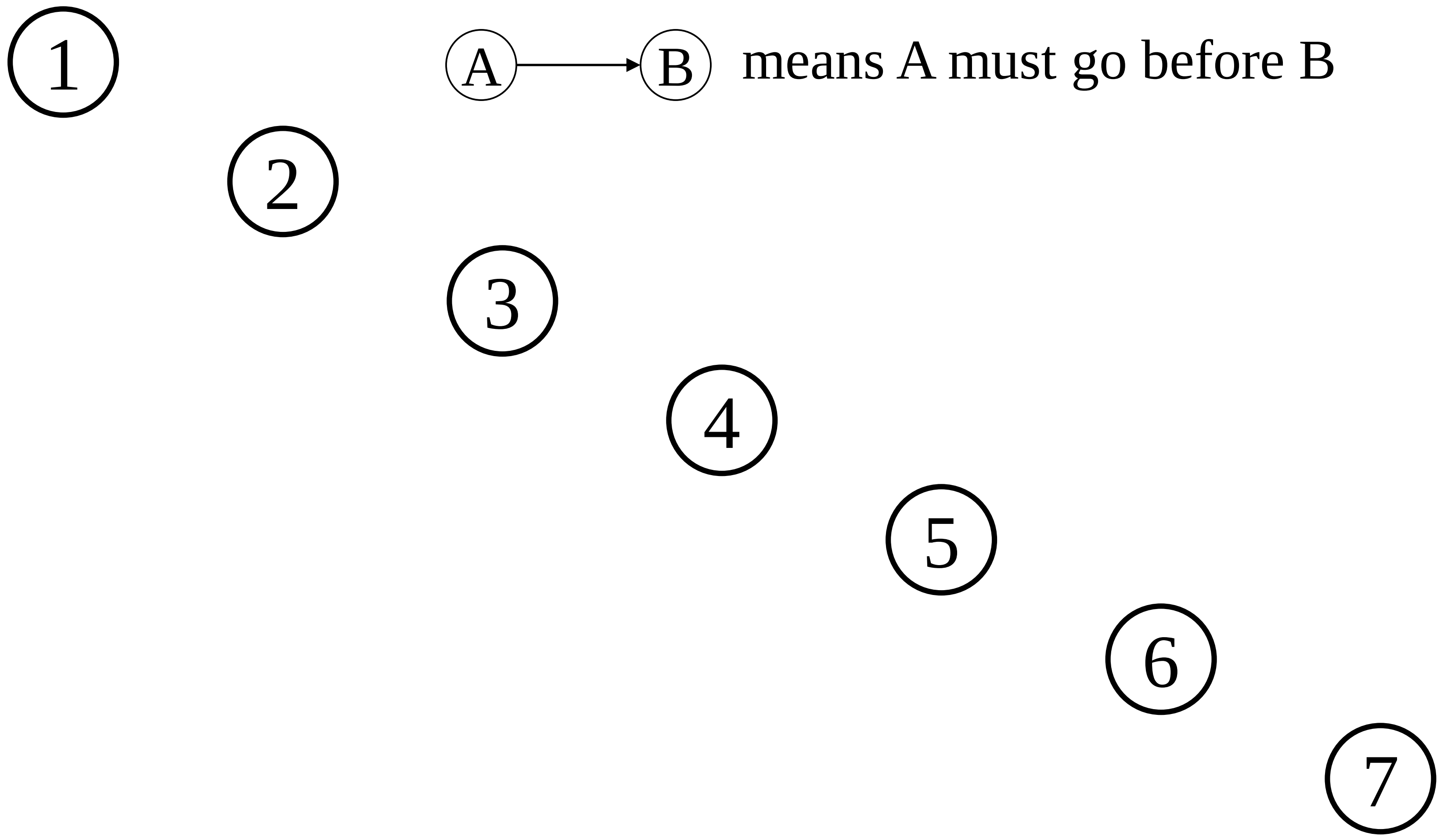
课程代号	课程代号	先行课程
C_1	高等数学	
C_2	普通物理	C_1
C_3	计算机原理	C_2
C_4	程序设计	
C_5	离散数学	C_1, C_4
C_6	数据结构	C_4, C_5
C_7	编译技术	C_4, C_6
C_8	操作系统	C_3, C_6

AOV NETWORK



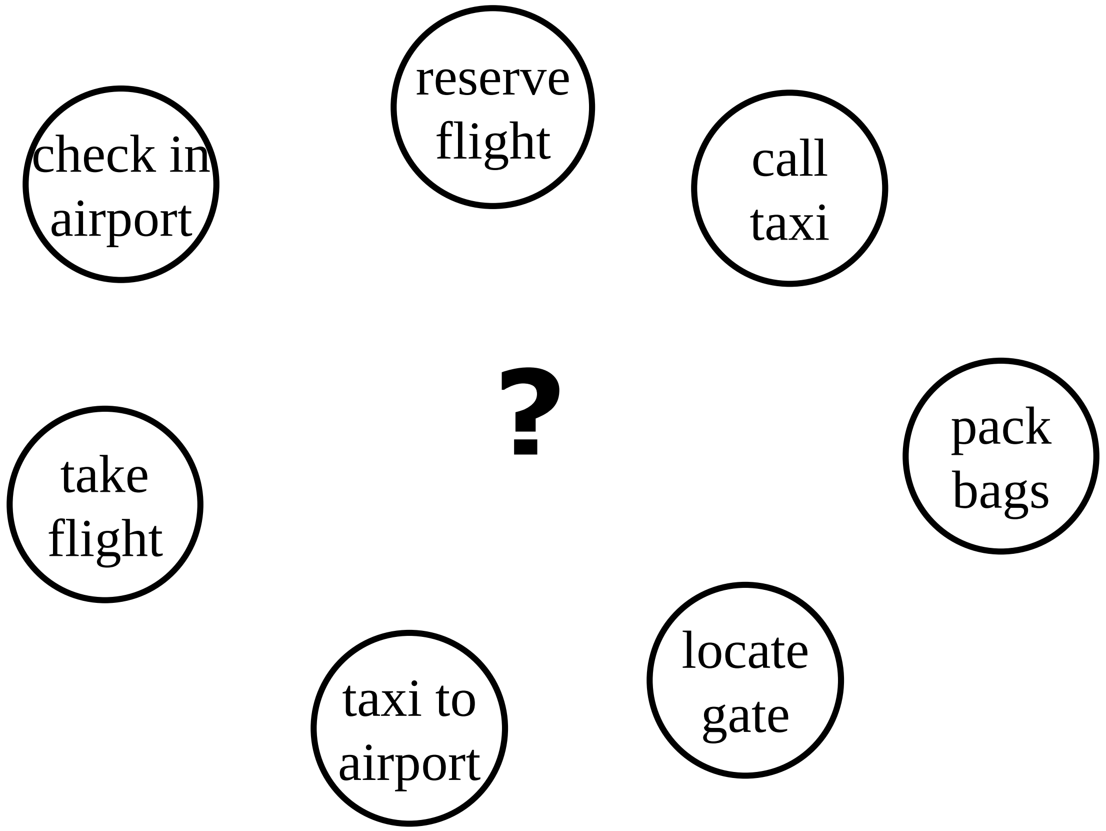
$$\begin{tikzcd}
 & C_2 \arrow[rr] & & C_3 \arrow[ddr] & \\
 & & C_4 \arrow[r] \arrow[dl] \arrow[dr] & C_7 & \\
C_1 \arrow[uur] \arrow[r] & C_5 \arrow[r] & C_6 \arrow[ur] \arrow[rr] & & C_8
\end{tikzcd}$$

Total Order



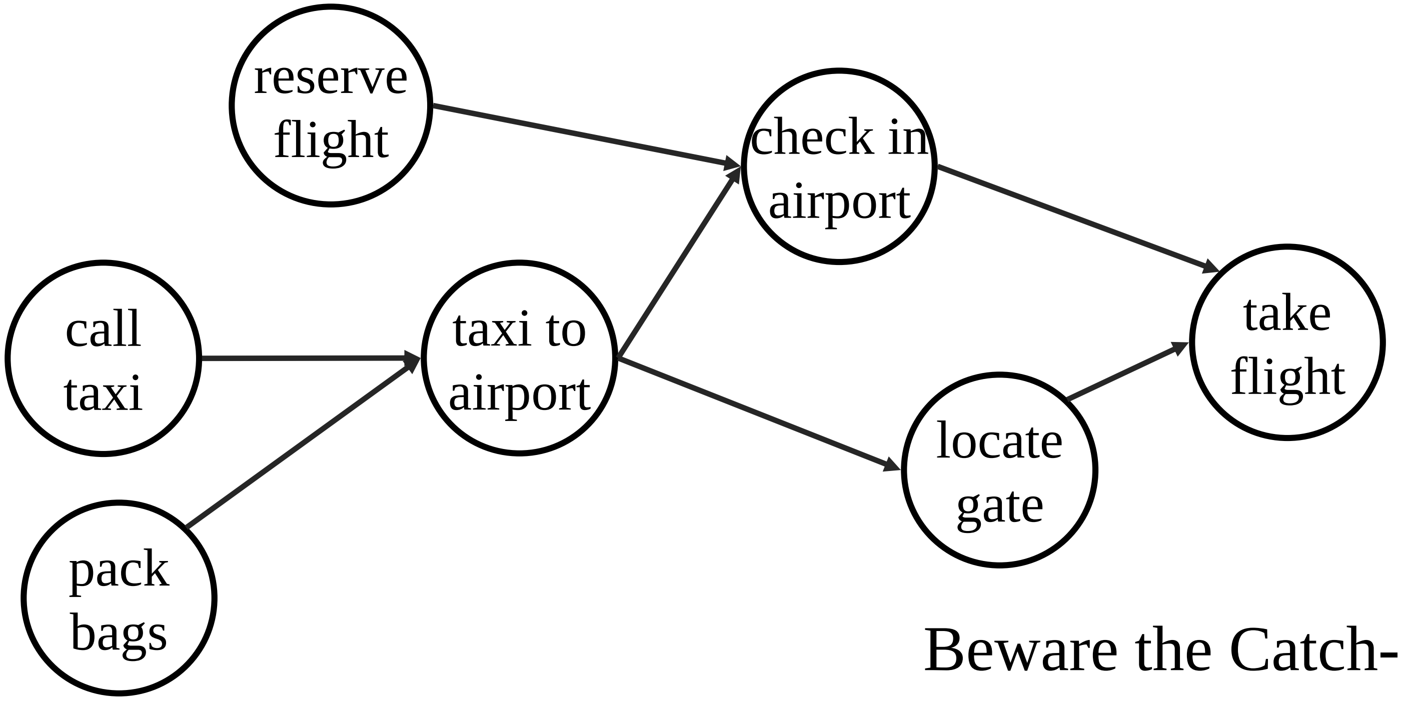
means A must go before B

Partial Order: Planning a Trip



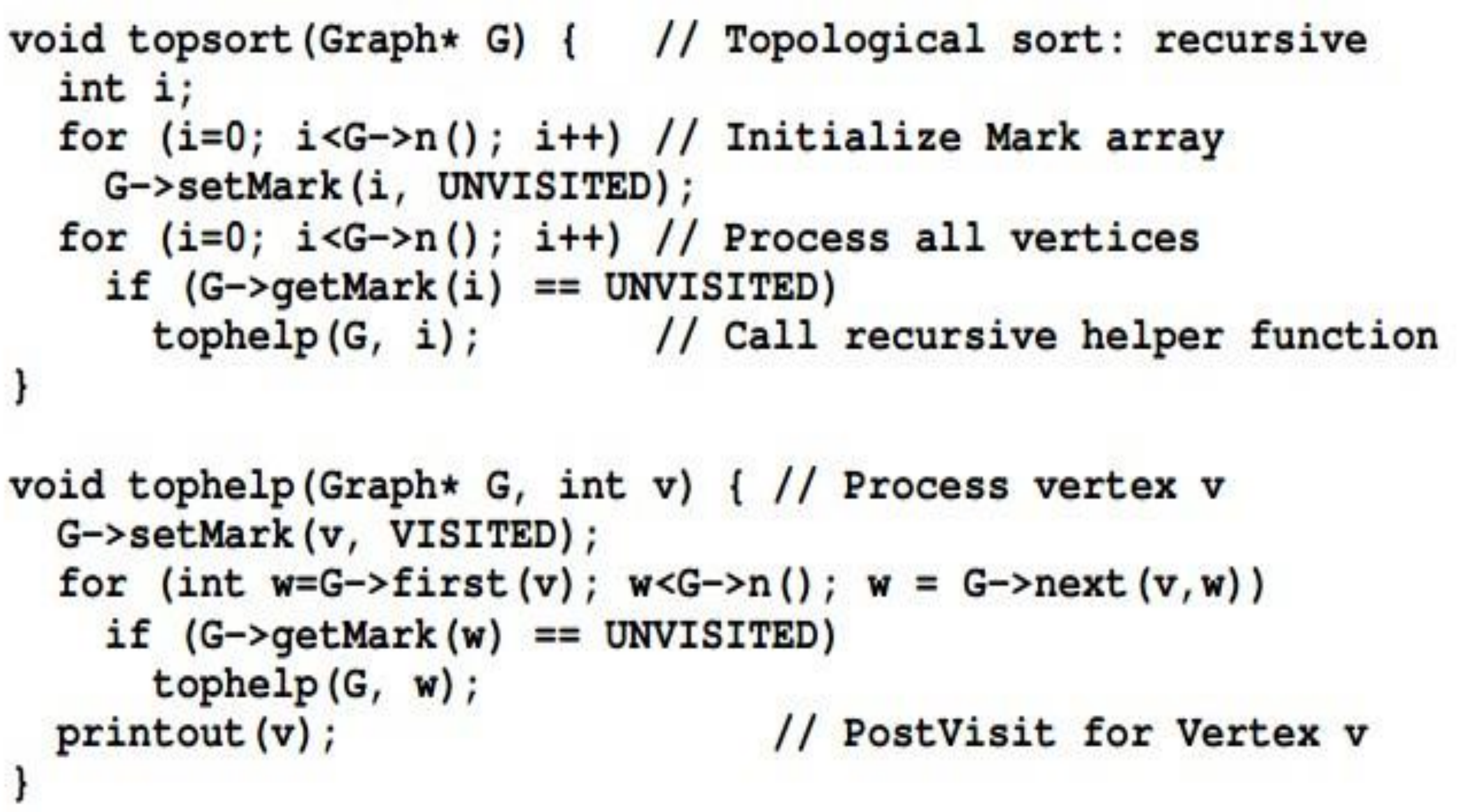
Topological Sort

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



Beware the Catch-22!

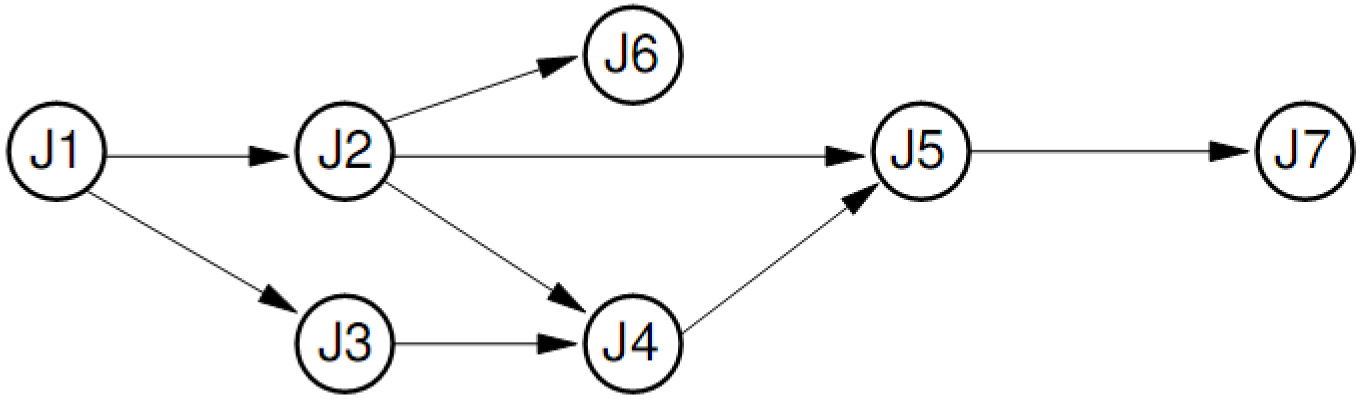
Topological Sort



```
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
}
```

Topological Sort



- An acceptable topological sort for this example is

  J1, J2,J3, J4, J5, J6, J7.

a b h c d g f e

在算法中需要用定量的描述替代定性的概念

没有前驱的顶点 == 入度为零的顶点

删除顶点及以它为尾的弧 == 弧头顶点的入度减1

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

Runtime?

Topo-Sort Take Two

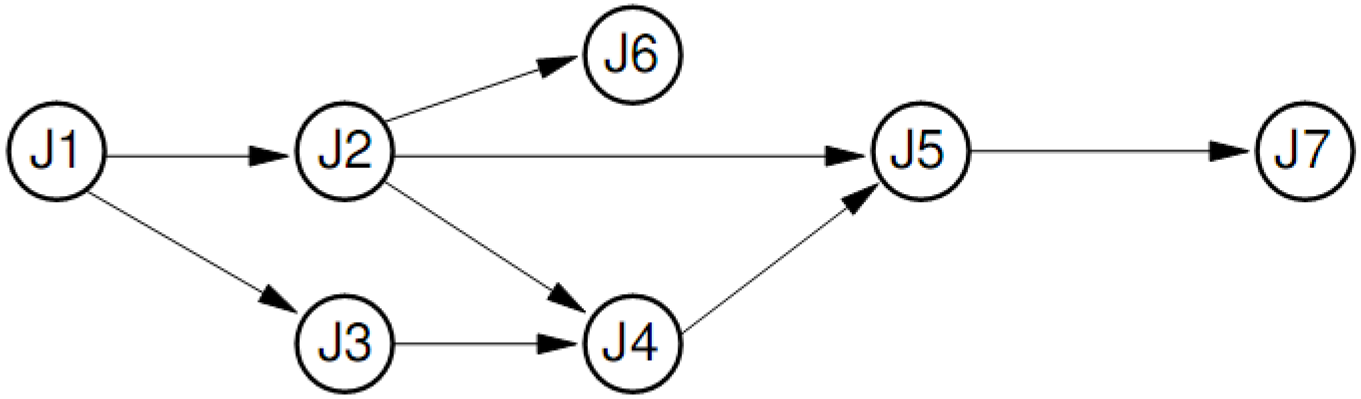
- Label each vertex's in-degree
- Put all in-degree-zero vertices in a queue

- 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

Runtime?

```
// 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);
    }
  }
}
```

Topological Sort



- An acceptable topological sort for this example is

  J1, J2,J3, J4, J5, J6, J7.

Knowledge Points

- Chapter 11, pp.390-399

Homework

- P410, 11.4-11.8

-End-