Graph Algorithms

Fall 2020

School of Software Engineering

South China University of Technology

Graph Searching

Section 9.5 and 9.6

Graph Traversals

- Recall traversal of binary trees
  - Preorder, inorder, or postorder traversal

- Graph traversal: visit the vertices of a graph in some specific order based on the graph's topology.

- Traversal algorithms typically begins with a start vertex and attempt to visit the remaining vertices from there
  - The traversal can only follow the edges in the graph.
  - It may not be possible to reach all vertices if the graph is not connected
  - The algorithms shall make sure not to go into an infinite loop if the graph contains cycles.

Graph Traversals

- A mark bit is maintained for each vertex
  - If a marked vertex is encountered during traversal, it is not visited a second time
  - If not all vertices are marked when the algorithm completes, we continue the traversal from another unvisited vertex.

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

  for (v=0; v<G->n(); v++)
    if (G->getMark(v) == UNVISITED)
      doTraverse(G, v);
}
```

Graph Traversals

- Three typical graph traversal strategies
  - Depth-first search (DFS)
  - Breadth-first search (BFS)
  - Topological sort

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

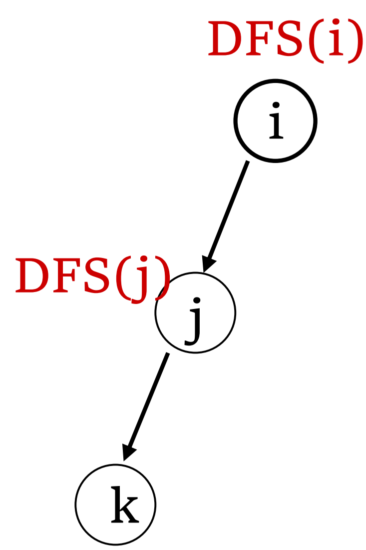
Graph Searching Methodology

- Breadth-First Search (BFS)
  - Use a queue to explore neighbors of source vertex, then neighbors of neighbors etc.
  - All vertices at a given distance (in number of edges) are explored before we go further
- Depth-First Search (DFS)
  - Searches down one path as deep as possible
  - When no vertices 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

Depth First Search Algorithm

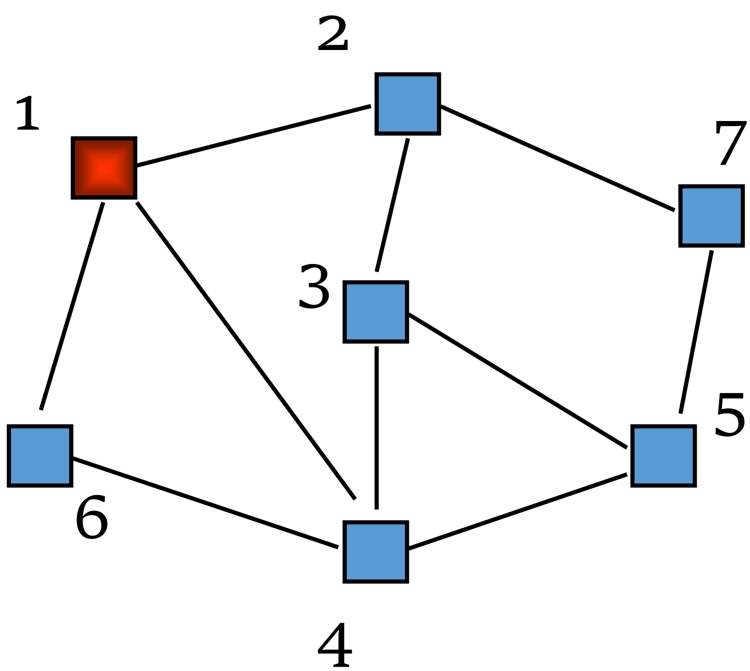
```
void DFS(Graph* G, int v) {
  PreVisit(G, v);  // Take some actions
  G->setMark(v, VISITED); //mark v
  for (int w=G->first(v); w<G->n(); w=G->next(v,w))
    if (G->getMark(w) == UNVISITED)
      DFS(G, w);
  PostVisit(G, v); // Take some actions
}
```

DFS Application: Spanning Tree

- Given a 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

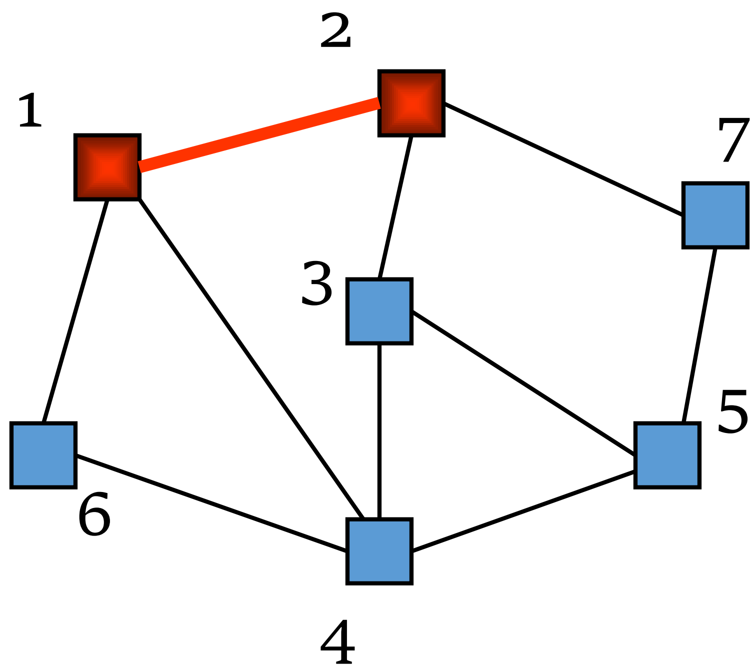
DFS Application: Spanning Tree

- Example of DFS: Graph connectivity and spanning tree



DFS(1)

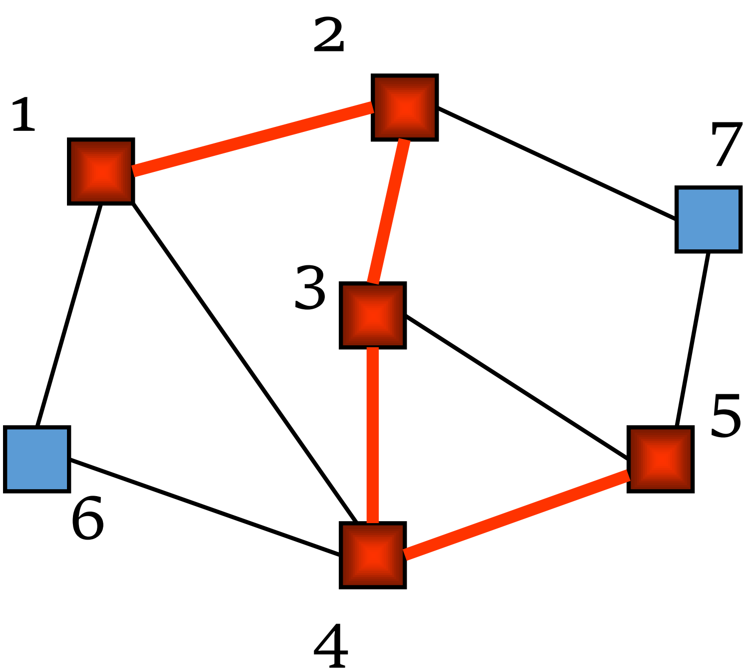
DFS Application: Spanning Tree



DFS(1)
 DFS(2)

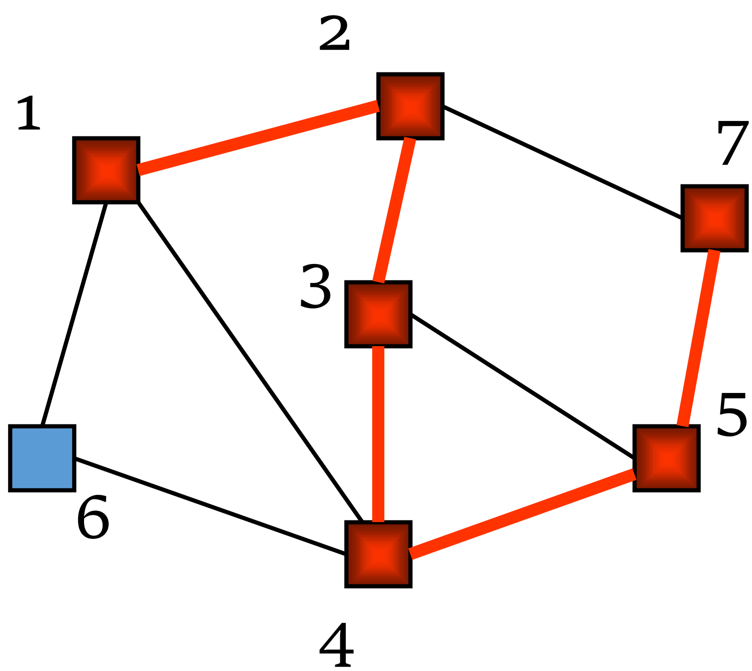
Red links will define the spanning tree if the graph is connected

DFS Application: Spanning Tree



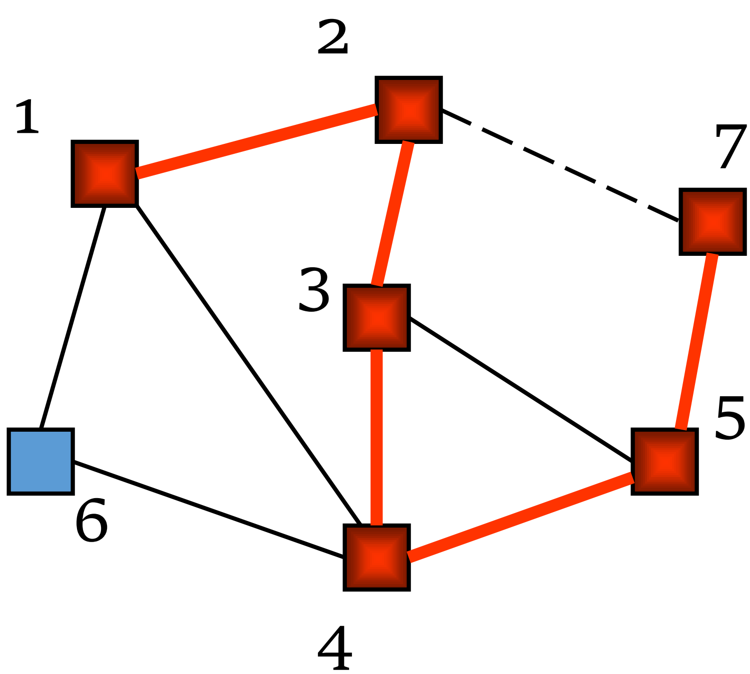
DFS(1)
DFS(2)
DFS(3)
DFS(4)
DFS(5)

DFS Application: Spanning Tree



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

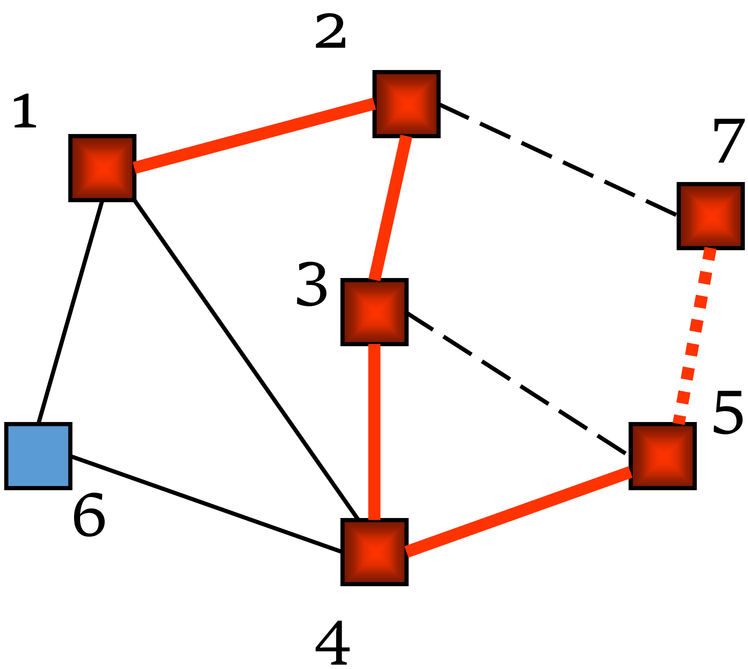
DFS Application: Spanning Tree



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

Now back up.

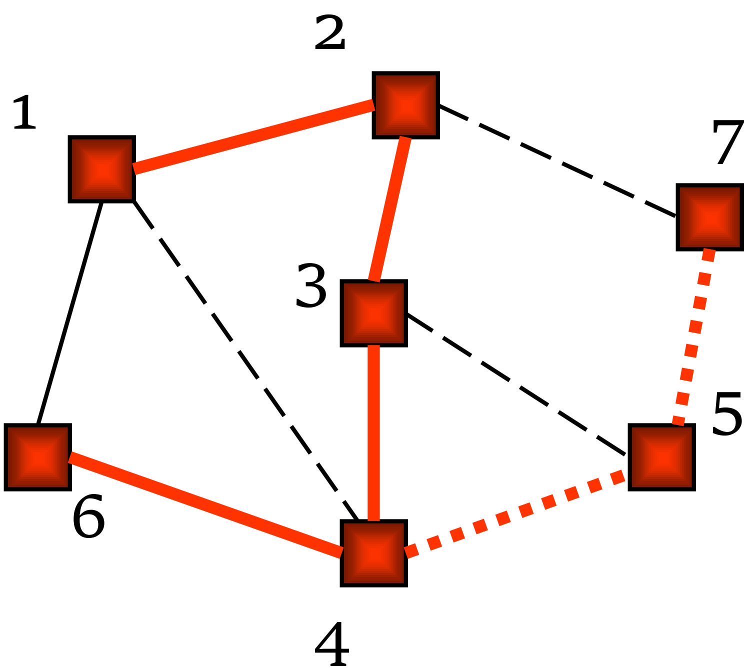
DFS Application: Spanning Tree



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

Back to 5,
but it has no
more neighbors.

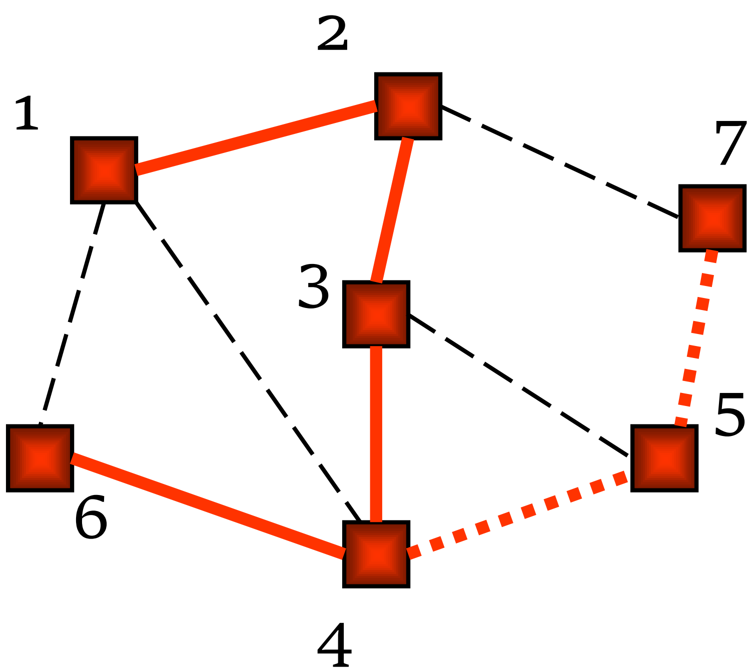
DFS Application: Spanning Tree



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

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

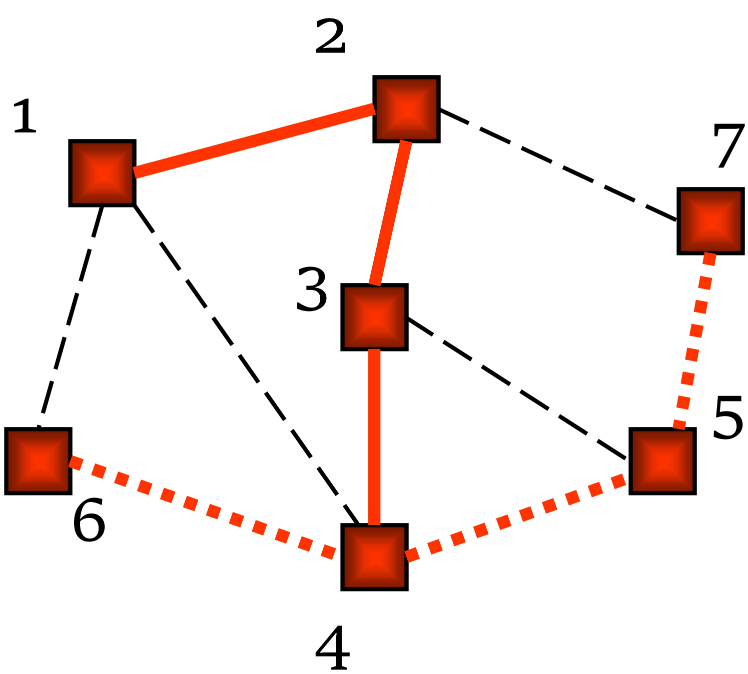
DFS Application: Spanning Tree



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

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

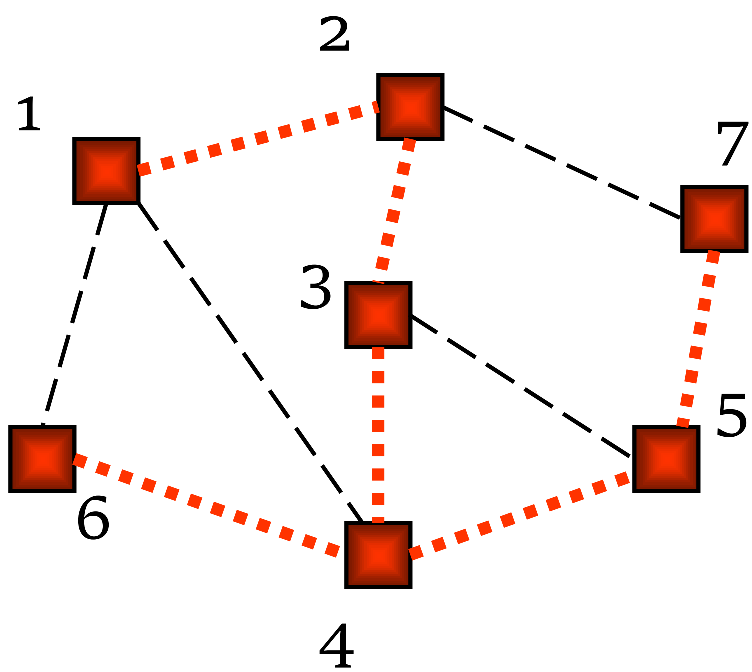
DFS Application: Spanning Tree



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

Back to 4.
Keep backing up.

DFS Application: Spanning Tree



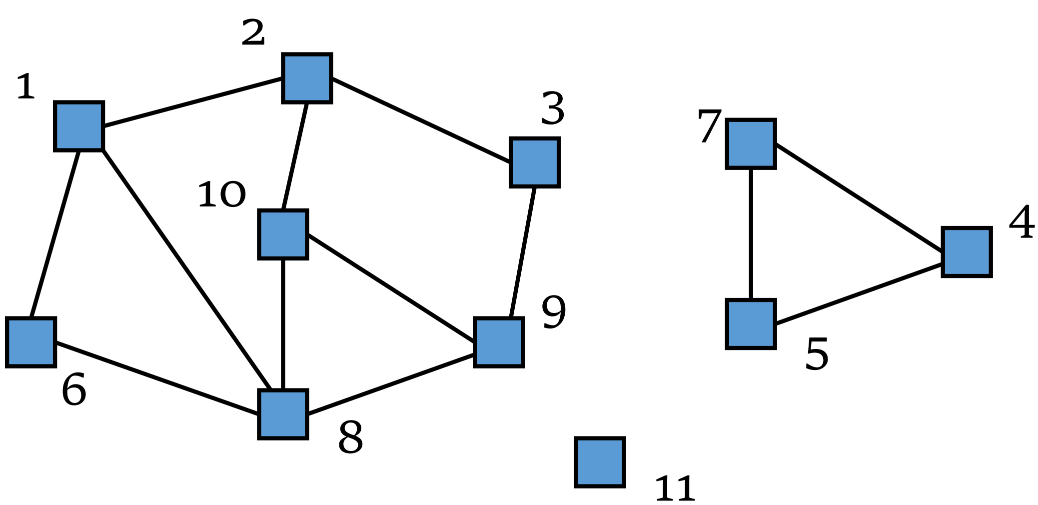
DFS(1)

All the way back to 1.

Done.

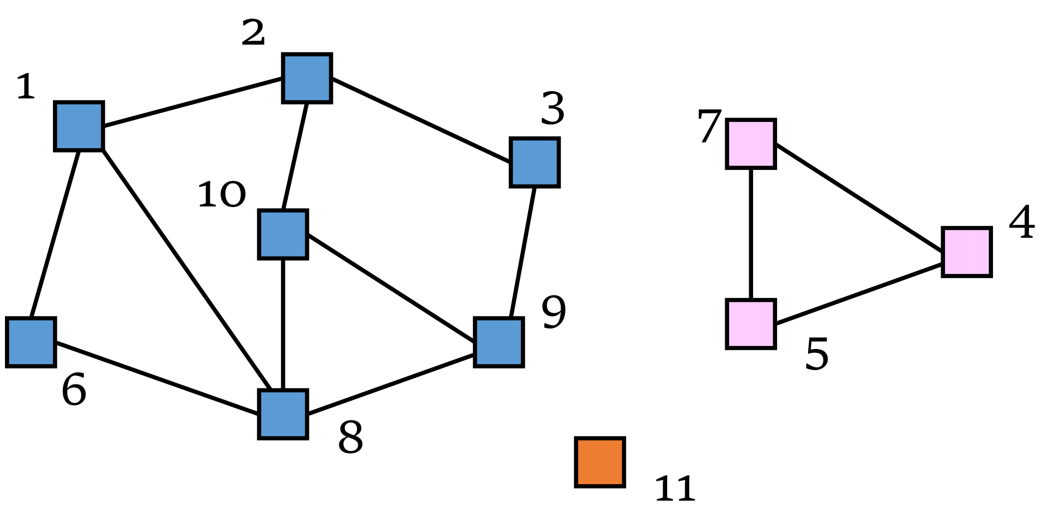
All nodes are marked so graph is connected; red links define a spanning tree

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 //
}
```

Performance DFS

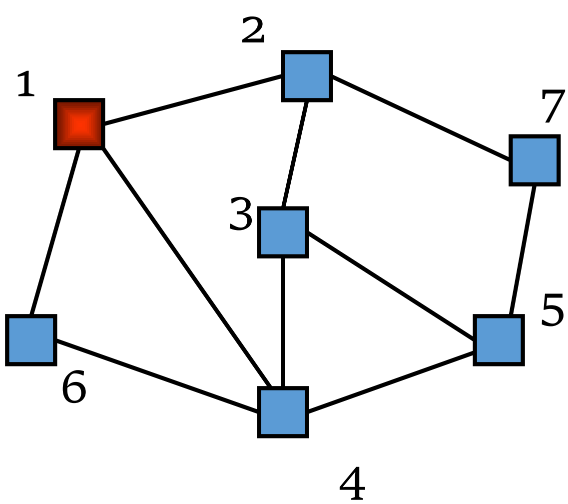
- n vertices and m edges
- Storage complexity $O(n + m)$
- Time complexity $O(n + m)$
- Linear Time!

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}
```

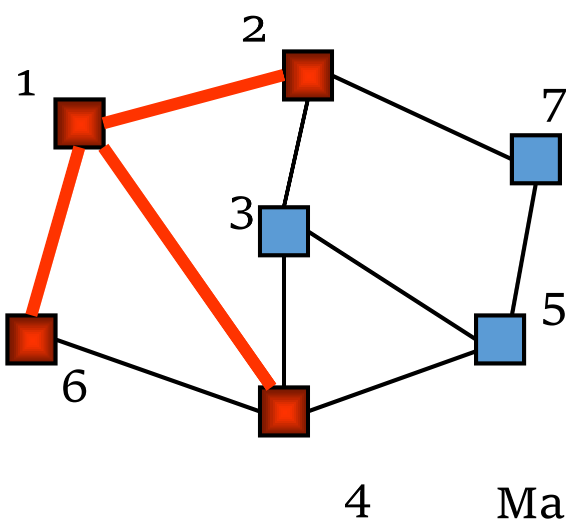
Can do Connectivity using BFS

- 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

```
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 Q
    v = Q->dequeue();
    PreVisit(G, v);   // Take some actions

    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);
      }
  }
}
```

Depth-First vs Breadth-First

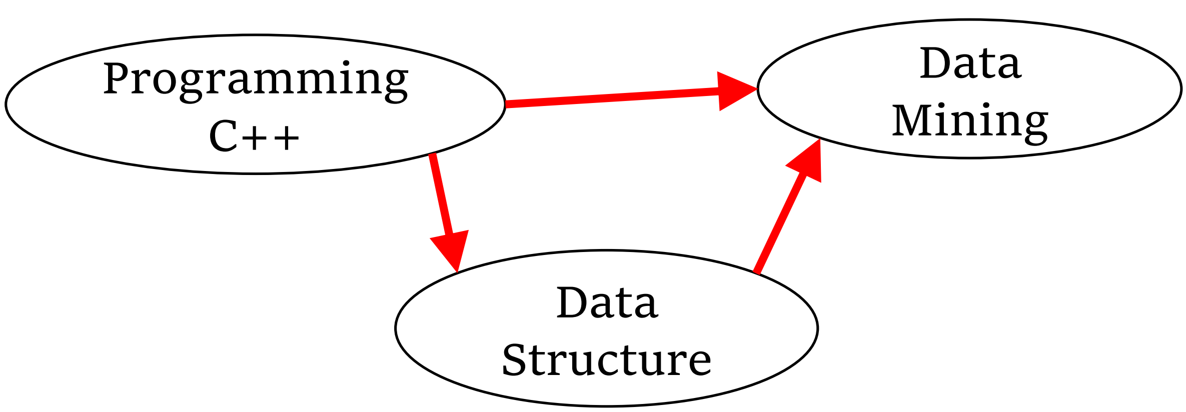
- Depth-First
  - Stack or recursion
  - Many applications
- Breadth-First
  - Queue (recursion no help)
  - Can be used to find shortest paths from the start vertex

Topological Sort

Section 9.2

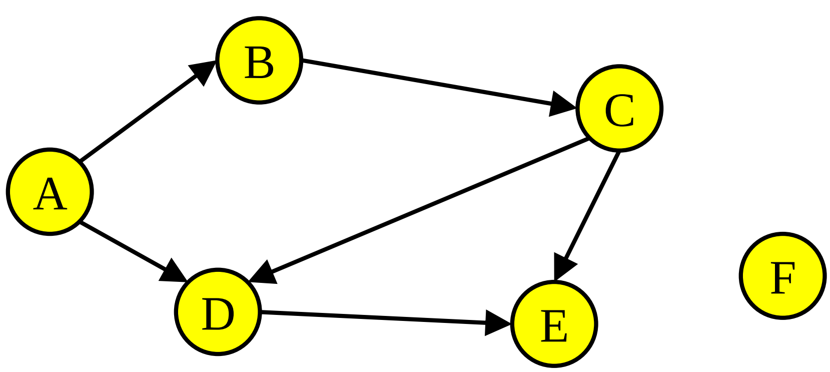
Topological Sort

- Given a series of tasks such as classes or jobs with prerequisite constraints, one task cannot be started until its prerequisites are completed.
- Organize the tasks into a linear order so that they are completed one at a time without violating any prerequisites.



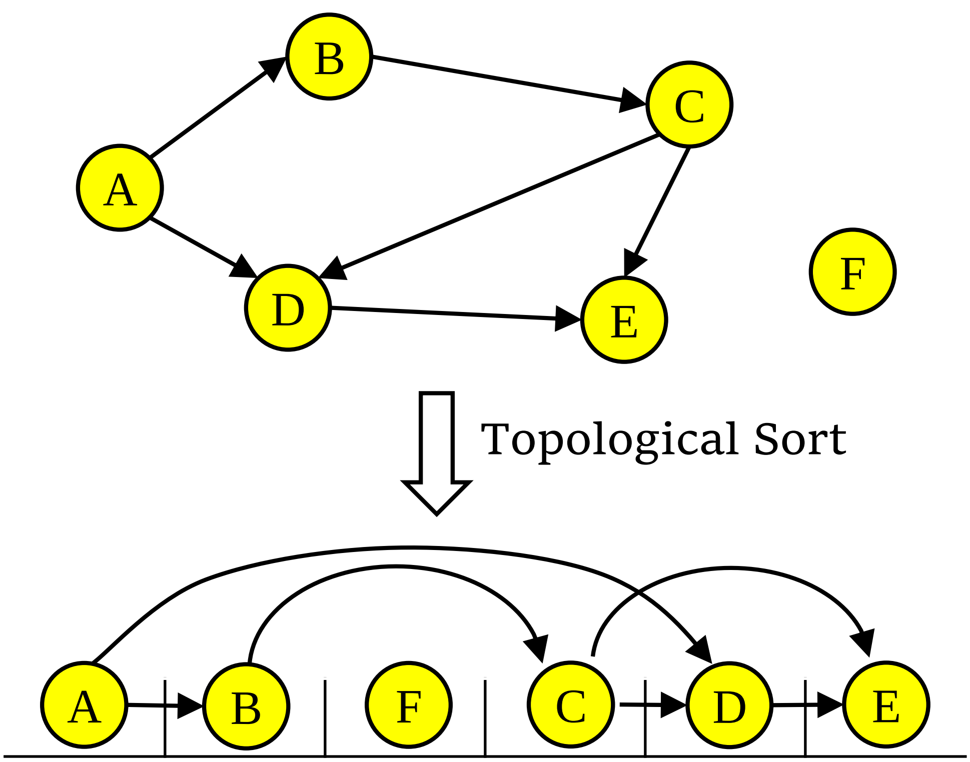
Topological Sort

- Given a digraph G = (V, E), find a linear ordering of its vertices such that:
  - for any edge (v, w) in E, v precedes w in the ordering



Topological Sort

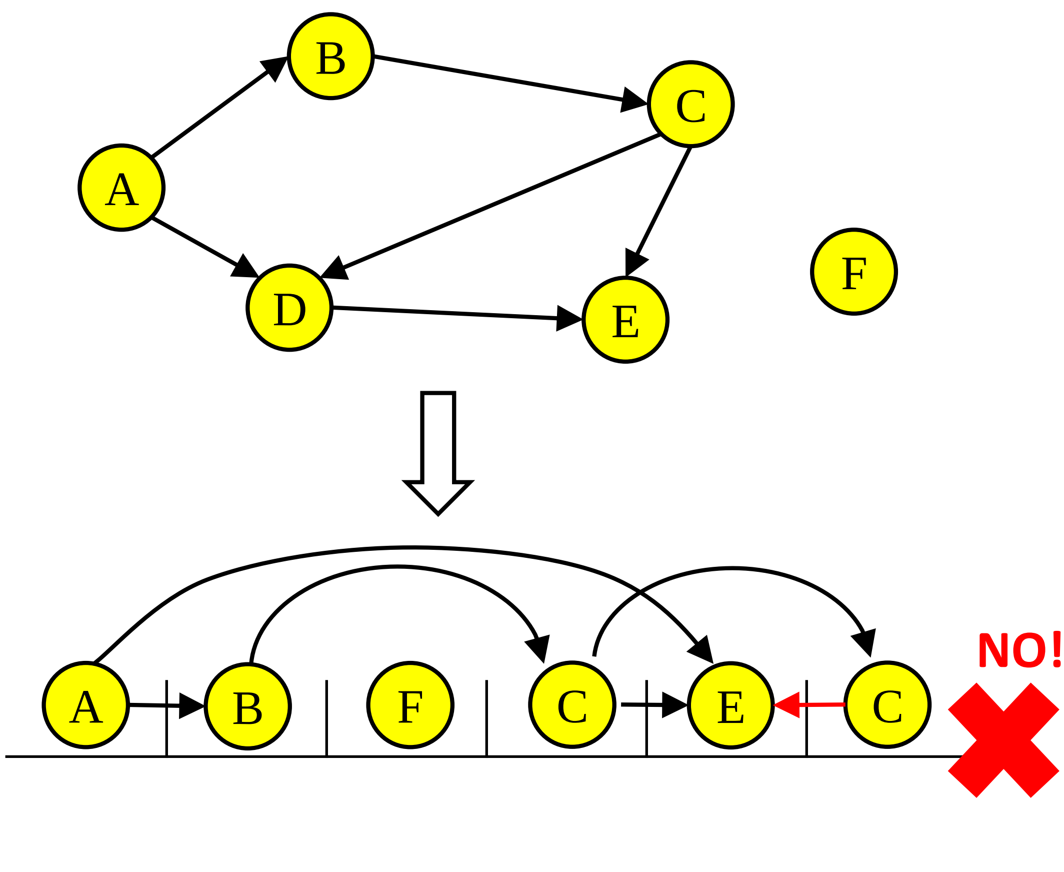
- **Any** linear ordering in which all the arrows go to the right is a valid solution



Note that F can go anywhere in this list because it is not connected. Also the solution is not unique.

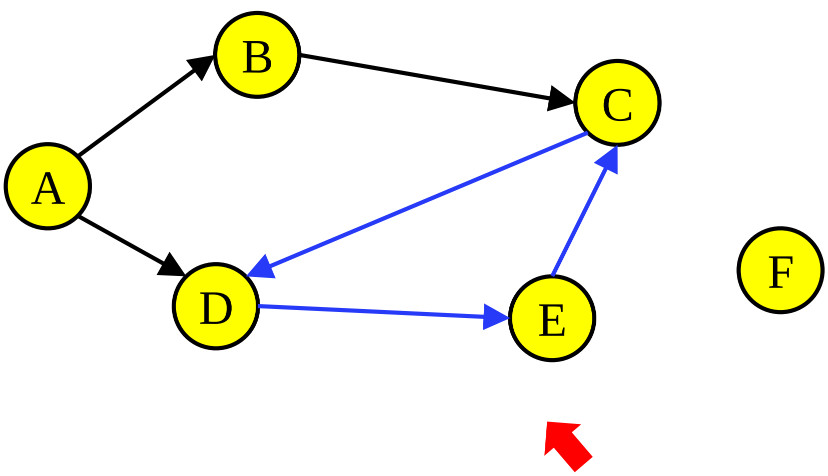
Topological Sort

- bad example



Topological Sort

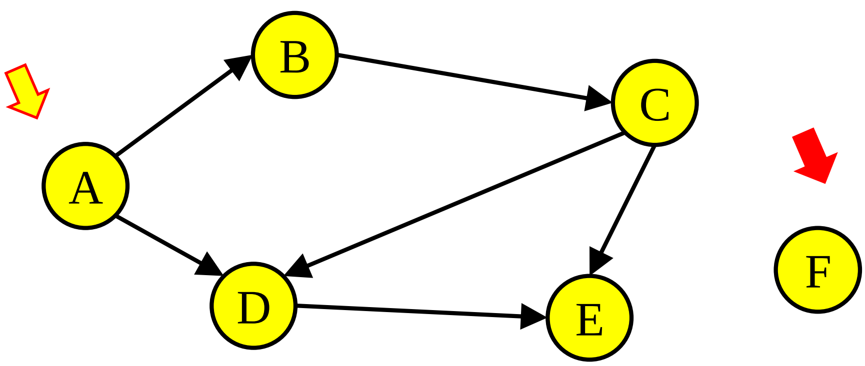
- **Only** direct acyclic graphs (DAG) can be topological sorted



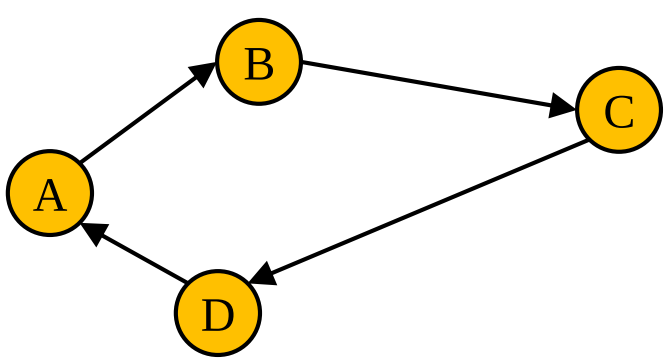
cyclic graph

Topological Sort Algorithm

- Step 1: Identify vertices that have no incoming edges
  - The "in-degree" of these vertices is zero
  - Select one of such vertices

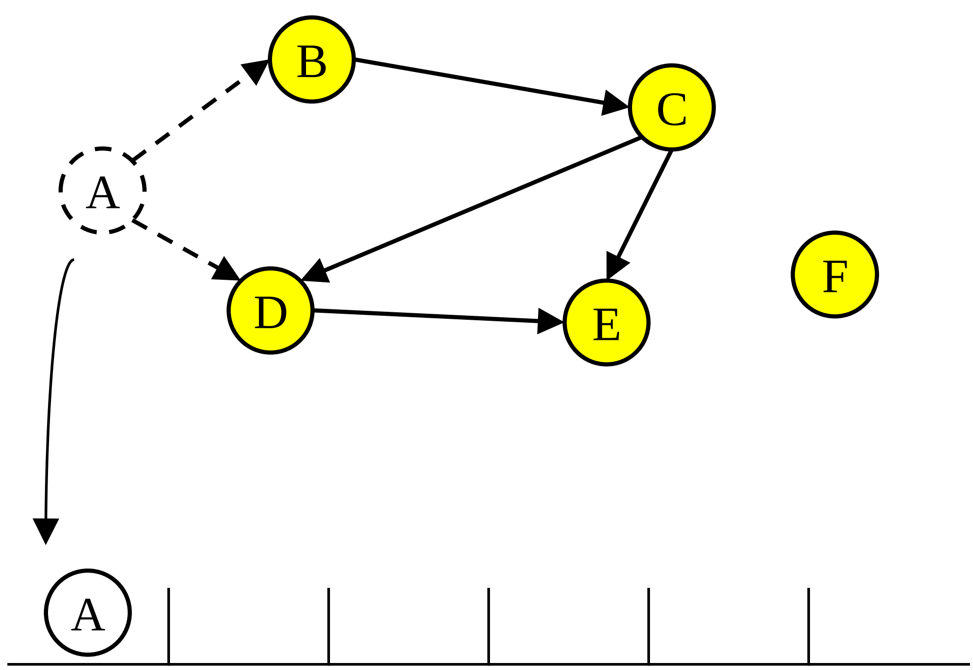


  - If no such vertices, graph has only cycle(s) (cyclic graph), Topological sort not possible – Halt



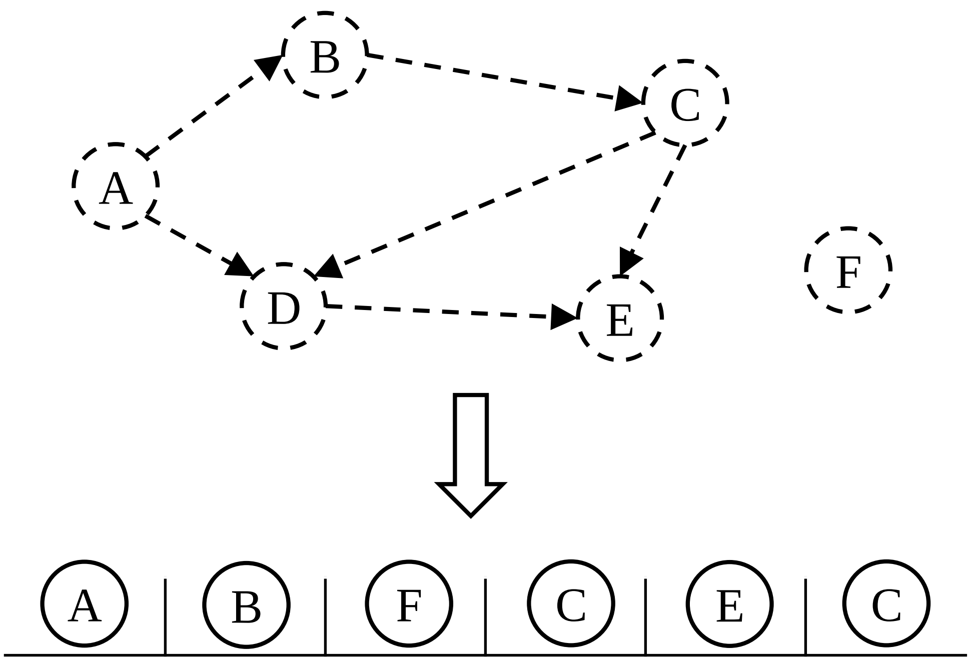
Topological Sort Algorithm

- Step 2: Delete this vertex of in-degree 0 and all its outgoing edges from the graph. Place it in the output



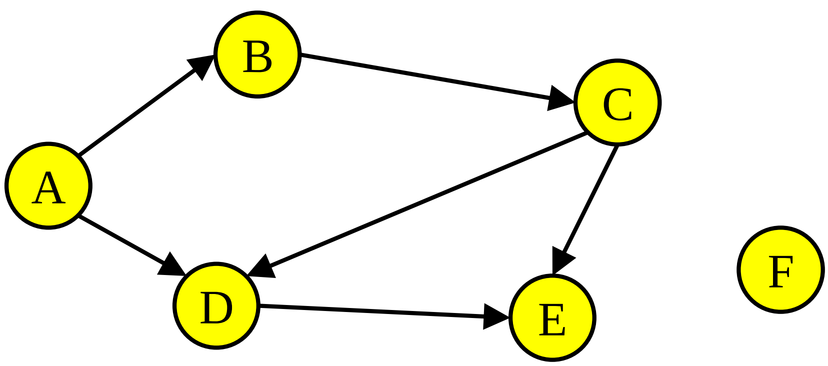
Topological Sort Algorithm

- Repeat Step 1 and Step 2 until graph is empty



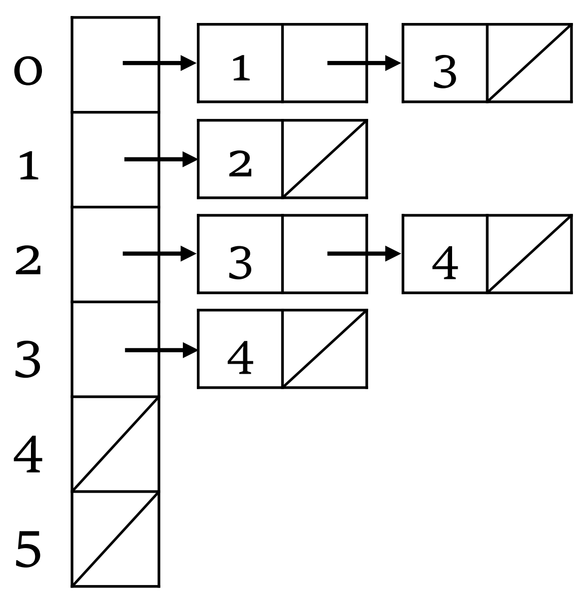
Topological Sort Algorithm Implementation

- Assume adjacency list representation



① Translation array

0	1	2	3	4	5
A	B	C	D	E	F



Topological Sort Algorithm Implementation

② Calculate In-degrees

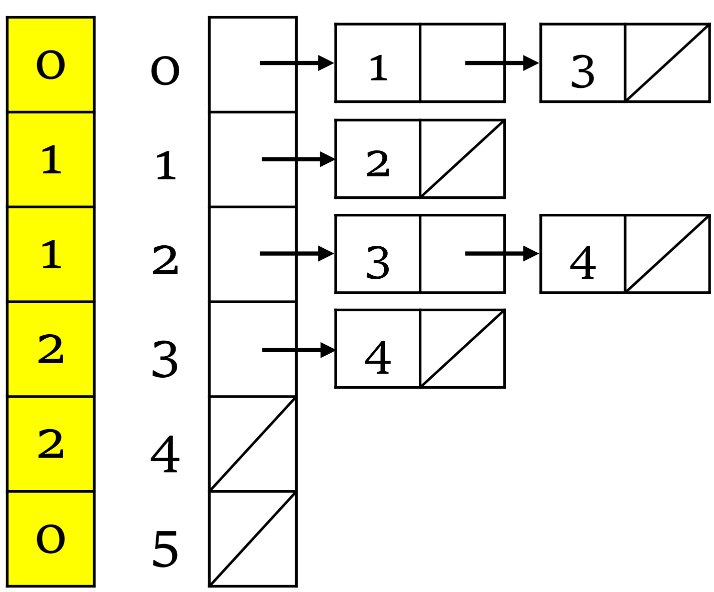
```
// in-degree array D[];
// vertex array A[]
for i = 0 to n-1 do D[i] := 0;
endfor
for i = 0 to n-1 do
  x := A[i];
  while x ≠ null do
    D[x.value] := D[x.value] + 1;
    x := x.next;
  endwhile
endfor
```

Topological Sort Algorithm Implementation

② Calculate In-degrees

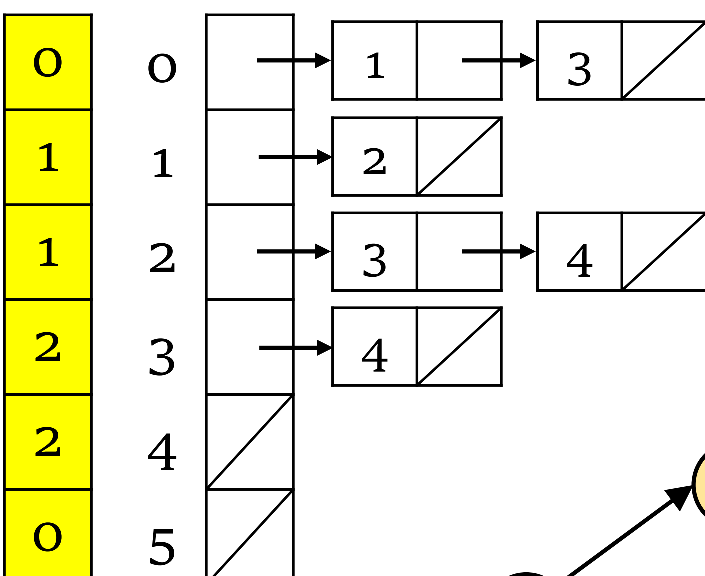
0	1	2	3	4	5
0	1	1	2	2	0

In-Degree array; or add a field to vertex array

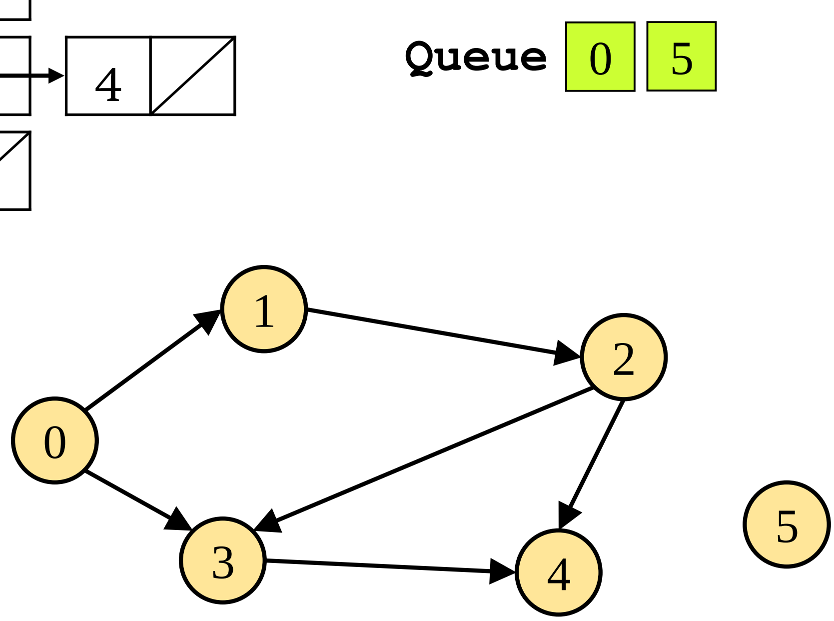


Topological Sort Algorithm Implementation

- Maintaining Degree 0 Vertices
  - Key idea: Initialize and maintain a queue (or stack) of vertices with In-Degree 0

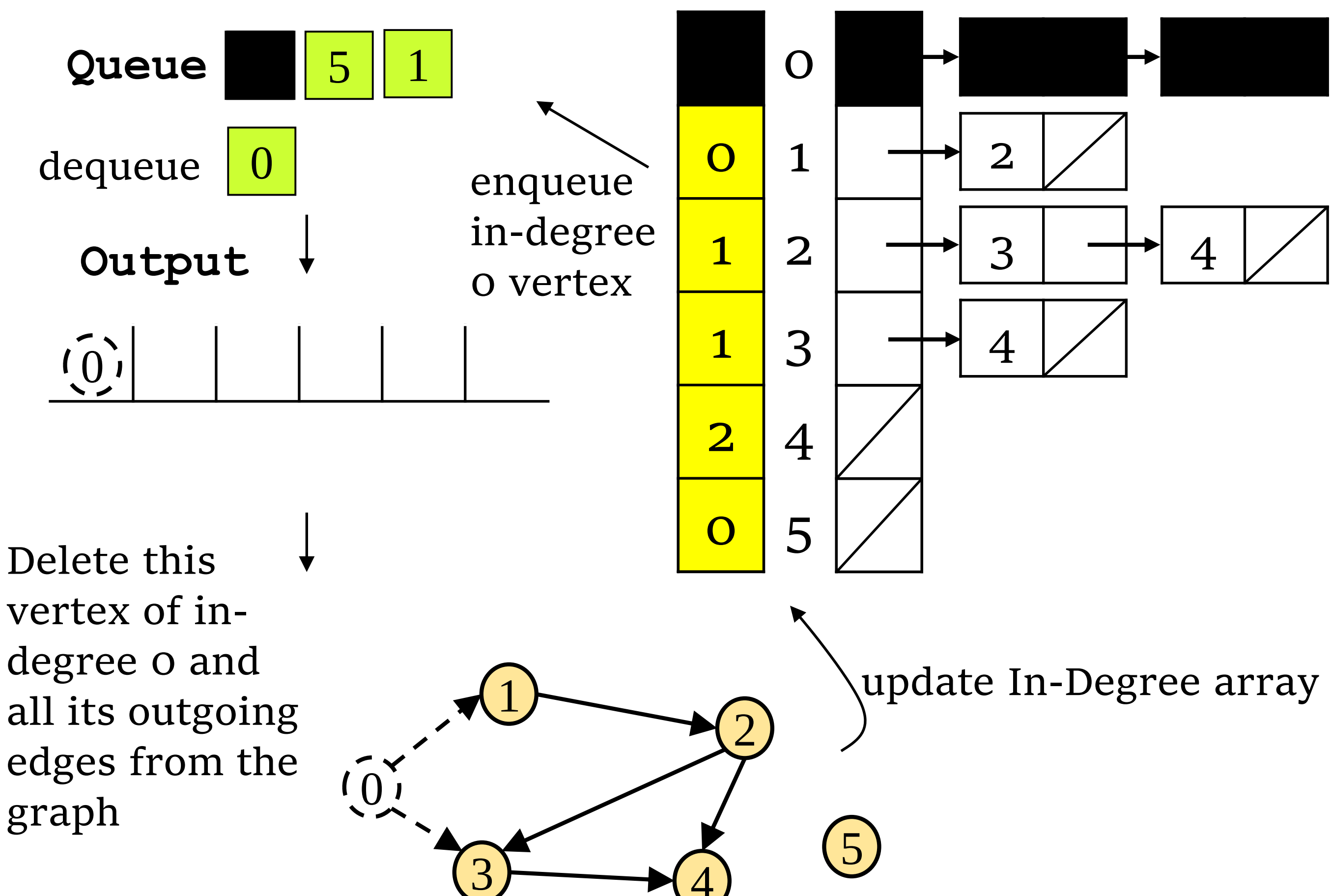


③ enqueue the vertices with In-Degree 0



Topological Sort Algorithm Implementation

④ dequeue vertex from queue to output;
update In-Degree array;
enqueue any vertex whose In-Degree becomes zero.



Topological Sort Algorithm

1. Store each vertex's In-Degree in an array D
2. Initialize queue with all "in-degree=0" vertices
3. While there are vertices remaining in the queue:
   (a) Dequeue and output a vertex
   (b) Reduce In-Degree of all vertices adjacent to it by 1
   (c) Enqueue any of these vertices whose In-Degree became zero
4. If all vertices are output then success, otherwise there is a cycle.

Topological Sort

```
//Queue-based topological sort
void topsort(Graph* G, Queue<int>* Q) {
   int InDgree[G->n()];  //the size is the number of vertices
   int v, w;
   for (v=0; v<G->n(); v++) InDgree[v] = 0; //initialize

   //compute indgree for every vertex
   for (v=0; v<G->n(); v++) // Process every edge
       for (w=G->first(v); w<G->n(); w=G->next(v,w))
          InDgree [w]++;   // Add to w's InDgree

   for (v=0; v<G->n(); v++) // Initialize Queue Q
       if (InDgree[v] == 0) // enqueue v with no prerequisites
           Q->enqueue(v);

   while (Q->length() != 0) { //process vertices in Q
       v = Q->dequeue();
       printout(v);     // output v

      for (w=G->first(v); w<G->n(); w=G->next(v,w)) {
          InDgree[w]--;    // One less prerequisite
          if (InDgree[w] == 0) //vertex v is now free
             Q->enqueue(w);
      }
   }
}
```

breadth-first

Topological Sort Analysis

- Initialize In-Degree array: $O(|V| + |E|)$
- Initialize Queue with In-Degree 0 vertices: $O(|V|)$
- Dequeue and output vertex:
  - $|V|$ vertices, each takes only $O(1)$ to dequeue and output: $O(|V|)$
- Reduce In-Degree of all vertices adjacent to a vertex and Enqueue any In-Degree 0 vertices:
  - $O(|E|)$
- For input graph $G=(V,E)$ run time $= O(|V| + |E|)$
  - Linear time!

Another Topological Sort Algorithm

- Topological Sort using a depth-first strategy
  - When a vertex is visited, do nothing
  - When the recursion pops back to that vertex, print the vertex
  - A topological sort is printed in reversed order.

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

void tophelp(Graph* G, int v) { // Process 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);    // output v
}
```

DFS, BFS and Topological Sort

- Depth-first search (DFS)
  - For both directed and undirected graphs
  - It is implemented using a stack or recursion
- Breadth-first search (BFS)
  - For both directed and undirected graphs
  - It is implemented using a queue
- Topological sort
  - For directed acyclic graphs (DAG)
  - It is implemented using DFS or a queue-based method

Homework 7-1

- Textbook Exercises 9.1