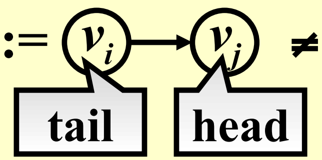
GRAPH ALGORITHMS

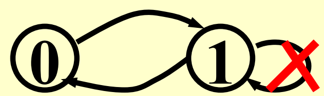
§1 Definitions

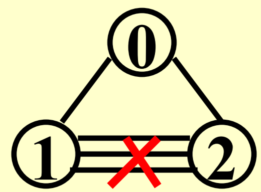
- **G(V, E)** where G ::= graph, V = V(G) ::= finite nonempty set of vertices, and E = E(G) ::= finite set of edges.

- **Undirected graph:** $(v_i , v_j) = (v_j , v_i)$ ::= the same edge.

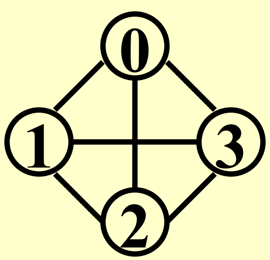
- **Directed graph (digraph):** $< v_i , v_j >$ ::= 

$\neq < v_j , v_i >$

- **Restrictions :**
  (1) **Self loop** is illegal.
  

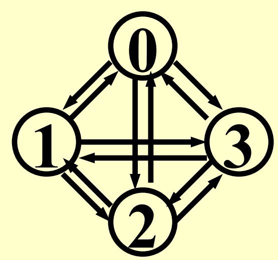
  (2) **Multigraph** is not considered
  

- **Complete graph:** a graph that has the maximum number of edges



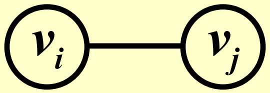
$\#$ of V $= n \Rightarrow$

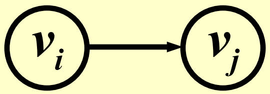
$\#$ of E $= C_n^2 = \frac{n(n-1)}{2}$



$\#$ of V $= n \Rightarrow$

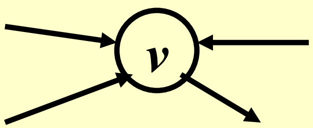
$\#$ of E $= P_n^2 = n(n-1)$

- 

  v_i and v_j are **adjacent** ;
  (v_i , v_j) is **incident on** v_i and v_j
- 

  v_i is **adjacent to** v_j ; v_j is **adjacent from** v_i ;
  < v_i , v_j > is **incident on** v_i and v_j
- **Subgraph G' ⊂ G** ::= V(G') ⊆ V(G) && E(G') ⊆ E(G)
- **Path (⊂ G) from v_p to v_q** ::= { v_p, v_{i1}, v_{i2}, ⋯, v_{in}, v_q } such that (v_p, v_{i1}), (v_{i1}, v_{i2}), ⋯, (v_{in}, v_q) or < v_p, v_{i1} >, ⋯, < v_{in}, v_q > belong to E(G)
- **Length of a path** ::= number of edges on the path
- **Simple path** ::= v_{i1}, v_{i2}, ⋯, v_{in} are distinct
- **Cycle** ::= simple path with $v_p = v_q$
- v_i and v_j in an undirected G are **connected** if there is a path from v_i to v_j (and hence there is also a path from v_j to v_i)
- An undirected graph G is **connected** if every pair of distinct v_i and v_j are connected

- **(Connected) Component of an undirected G** ::= the maximal connected subgraph
- **A tree** ::= a graph that is connected and *acyclic*
- **A DAG** ::= a directed acyclic graph
- **Strongly connected directed graph G** ::= for every pair of v_i and v_j in V(G), there exist directed paths from v_i to v_j and from v_j to v_i. If the graph is connected without direction to the edges, then it is said to be **weakly connected**
- **Strongly connected component** ::= the maximal subgraph that is strongly connected
- **Degree(v)** ::= number of edges incident to v. For a directed G, we have **in-degree** and **out-degree**. For example:



in-degree(v) = 3; out-degree(v) = 1; degree(v) = 4

- Given G with n vertices and e edges, then

$$e = \left(\sum_{i=0}^{n-1} d_i \right) \Big/ 2 \quad \text{where } d_i = \text{degree}(v_i)$$

❖ Representation of Graphs

Adjacency Matrix

adj_mat [n] [n] is defined for G(V, E) with n vertices, n 1 :

The trick is to store the matrix as a 1-D array:
adj_mat [$n(n+1)/2$] = { a_{11}, a_{21}, a_{22}, ..., a_{n1}, ..., a_{nn} }
The index for a_{ij} is ($i * (i - 1) / 2 + j$).

Not

degree(i) = $\sum_{j=0}$ adj_ma d)

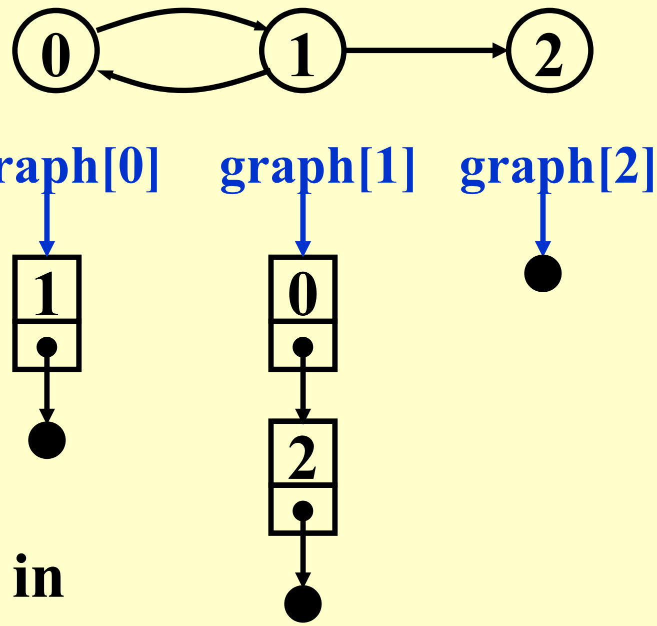
$+ \sum_{j=0}^{n-1}$ adj_mat[j][i] (if G is directed)

Adjacency Lists

Replace each row by a linked list

〖Example〗

$$\text{adj\_mat}[3][3] = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$



Note: The order of nodes in each list does not matter.

For undirected G:

$S = n$ heads $+ 2e$ nodes $= (n+2e)$ ptrs$+2e$ ints

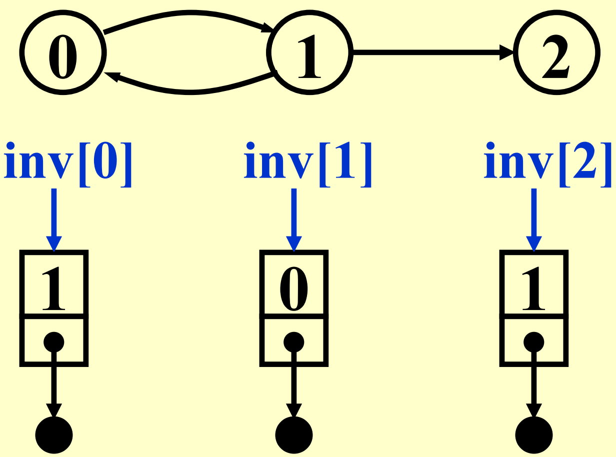
Degree(i) = number of nodes in graph[i] (if G is undirected).

T **of examine E(G) = O($n + e$)**

If G is directed, we need to find in-degree(v) as well.

Method 1 **Add inverse adjacency lists.**

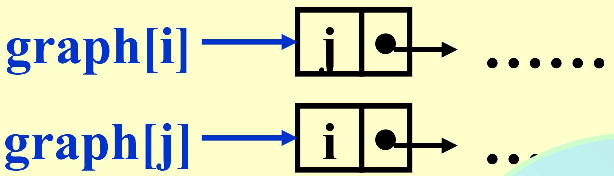
〖Example〗



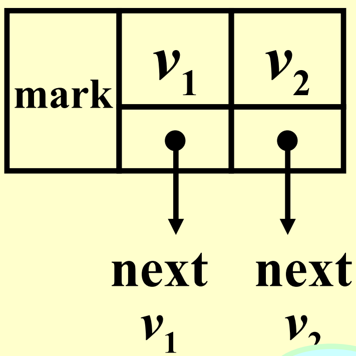
Method 2 **Multilist (Ch 3.2) representation for adj_mat[i] [j]**

Adjacency Multilists

In adjacency list, for each (*i*, *j*) we have two nodes:



Now let’s combine the two nodes int



Wait a minute ...
Look at the space taken:
(n+2e) ptrs + 2e ints
and “mark” is not counted.
What’s the advantage?

Sometimes we need to mark the edge after examine it, and then find the next edge. This representation makes it easy to do so.

Weighted Edges

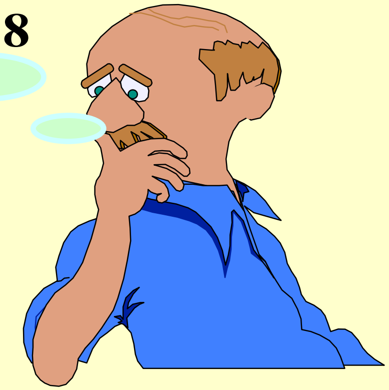
- **adj_mat [i] [j] = weight**
- **adjacency lists \ multilists : add a weight field to the node.**

§2 Topological Sort

〖 Example 〗 Courses needed for a computer science degree at a hypothetical university

Course number	Course name	Prerequisites
C1	Programming I	None
C2		None
C3		C2
C4		
C5		
C6		
C7		, C6
C8	Assembly	C3
C9	Operating Systems	, C8
C10	Programming Languages	C7
C11	Compiler Design	C10
C12	Artificial Intelligence	C7
C13	Computational Theory	C7
C14	Parallel Algorithms	C13
C15	Numerical Analysis	C6

How shall we convert this list into a graph?



- **AOV Network** ::= digraph G in which V(G) represents activities (e.g. the courses) and E(G) represents precedence relations (e.g. $C1 \longrightarrow C3$ means that C1 is a prerequisite course of C3).

- i is a **predecessor** of j ::= there is a path from i to j

  i is an **immediate predecessor** of j ::= $< i, j > \in E(G)$

  Then j is called a **successor** (**immediate successor**) of i

- **Partial order** ::= a precedence relation which is both **transitive** ($i \rightarrow k, k \rightarrow j \Rightarrow i \rightarrow j$) and **irreflexive** ($i \rightarrow i$ is impossible).

Note: If the precedence relation is reflexive, then there must be a
n i such that i is a predecessor of i. That is, i must be done b
efore i is started. Therefore if a project is **feasible**, it must
be **irreflexive**.

Feasible AOV network must be a dag (directed acyclic graph).

【 Definition 】 A topological order is a linear ordering of the vertices of a graph such that, for any two vertices, *i*, *j*, if *i* is a predecessor of *j* in the network then *i* precedes *j* in the linear ordering.

〖 Example 〗 One possible suggestion on course schedule for a computer science degree could be:

Course number	Course name	Prerequisites
C1	Programming I	None
C2	Discrete Mathematics	None
C4	Calculus I	None
C3	Data Structure	C1, C2
C5	Calculus II	C4
C6	Linear Algebra	C5
C7	Analysis of Algorithms	C3, C6
C15	Numerical Analysis	C6
C8	Assembly Language	C3
C10	Programming Languages	C7
C9	Operating Systems	C7, C8
C12	Artificial Intelligence	C7
C13	Computational Theory	C7
C11	Compiler Design	C10
C14	Parallel Algorithms	C13

Note: The topological orders may **not be unique** for a network. For example, there are several ways (topological orders) to meet the degree requirements in computer science.

Goal Test an AOV for feasibility, and generate a topological order if possible.

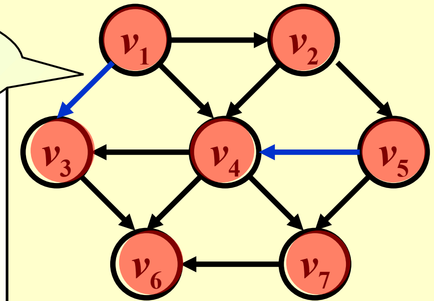
```
void Topsort( Graph G )
{   int  Counter;
    Vertex  V, W;
    for ( Counter = 0; Counter < NumVertex; Counter ++ ) {
            V = FindNewVertexOfDegreeZero( );    /* O( |V| ) */
            if ( V == NotAVertex ) {
                Error ( "Graph has a cycle" );   break;  }
            TopNum[ V ] = Counter; /* or output V */
            for ( each W adjacent to V )
                Indegree[ W ] – – ;
    }
}
```

$T = O(|V|^2)$

☞ **Improvement:** Keep all the unassigned vertices of degree 0 in a special box (queue or stack).

Mistakes in Fig 9.4 on p.289

```
void Topsort( Graph G )
{   Queue  Q;
    int  Counter = 0;
    Vertex  V, W;
    Q = CreateQueue( NumVertex );  MakeEmpty( Q );
    for ( each vertex V )
            if ( Indegree[ V ] == 0 )   Enqueue( V, Q );
    while ( !IsEmpty( Q ) ) {
            V = Dequeue( Q );
            TopNum[ V ] = ++ Counter; /* assign next */
            for ( each W adjacent to V )
                if ( – – Indegree[ W ] == 0 )  Enqueue( W, Q );
    }  /* end-while */
    if ( Counter != NumVertex )
            Error( “Graph has a cycle” );
    DisposeQueue( Q ); /* free memory */
}
```



Indegree

v_1	0
v_2	0
v_3	0
v_4	0
v_5	0
v_6	0
v_7	0