- Graphs

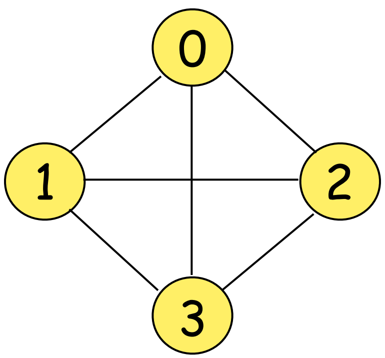
Application of Graphs

- Analysis of electrical circuits
- Finding shortest routes
- Project planning
- Identification of chemical compounds
- Linguistics
- Social Sciences, and so on …

Definition of A Graph

- A graph, G, consists of two sets, V and E.
  - V is a finite, nonempty set of vertices.
  - E is set of pairs of vertices called edges.
- The vertices of a graph G can be represented as V(G).
- Likewise, the edges of a graph, G, can be represented as E(G).
- Graphs can be either undirected graphs or directed graphs.
- For a undirected graph, a pair of vertices (u, v) or (v, u) represent the same edge.
- For a directed graph, a directed pair <u, v> has u as the tail and the v as the head. Therefore, <u, v> and <v, u> represent different edges.

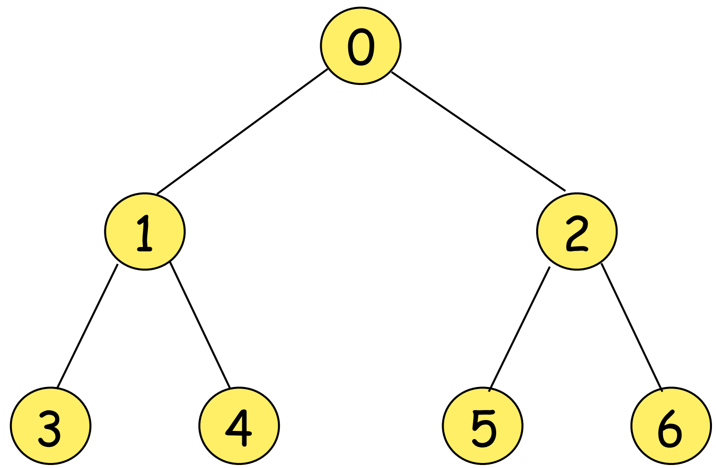
Three Sample Graphs



$V(G_1) = \{0, 1, 2, 3\}$

$E(G_1) = \{(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)\}$

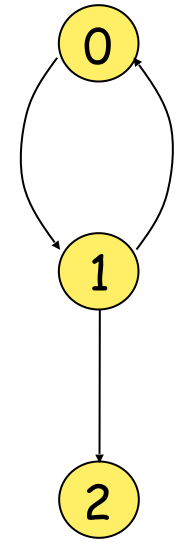
(a) G_1



$V(G_2) = \{0, 1, 2, 3, 4, 5, 6\}$

$E(G_2) = \{(0, 1), (0, 2), (1, 3), (1, 4), (2, 5), (2, 6)\}$

(b) G_2



$V(G_3) = \{0, 1, 2\}$

$E(G_3) = \{\langle 0, 1\rangle, \langle 1, 0\rangle, \langle 1, 2\rangle\}$

(c) G_3

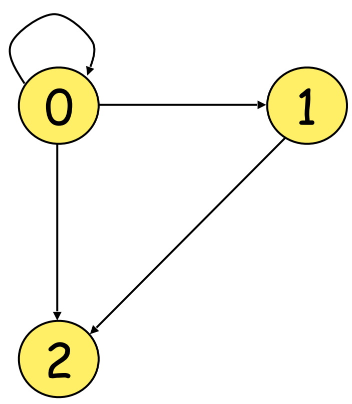
Graph Restrictions

- A graph may not have an edge from a vertex back to itself.
  - (v, v) or <v, v> are called self edge or self loop. If a graph with self edges, it is called **a graph with self edges**.
- A graph may not have multiple occurrences of the same edge.
  - If without this restriction, it is called a **multigraph**.

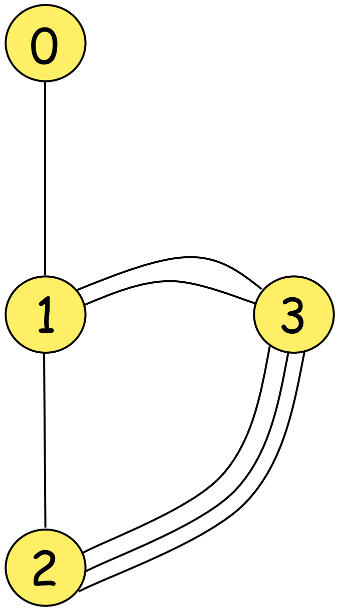
Complete Graph

- The number (max) of distinct unordered pairs (u, v) with $u \neq v$ in a graph with n vertices is $n(n-1)/2$.
- A *complete* unordered graph is an unordered graph with exactly $n(n-1)/2$ edges.
- A *complete* directed graph is a directed graph with exactly $n(n-1)$ edges.

Examples of Graphlike Structures



(a) Graph with a self edge



(b) Multigraph

Graph Edges

- If (u, v) is an edge in E(G), vertices u and v are adjacent and the edge (u, v) is the incident on vertices u and v.
- For a directed graph, <u, v> indicates u is adjacent to v and v is adjacent from u.

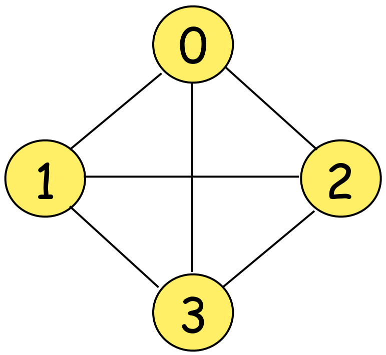
Degree of A Vertex

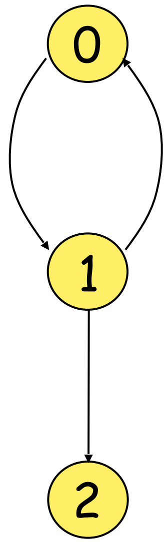
- *Degree of a vertex*: The *degree* of a vertex is the number of edges incident to that vertex.
- If G is a directed graph, then we define
  - *in-degree of a vertex*: is the number of edges for which vertex is the head.
  - *out-degree of a vertex*: is the number of edges for which the vertex is the tail.

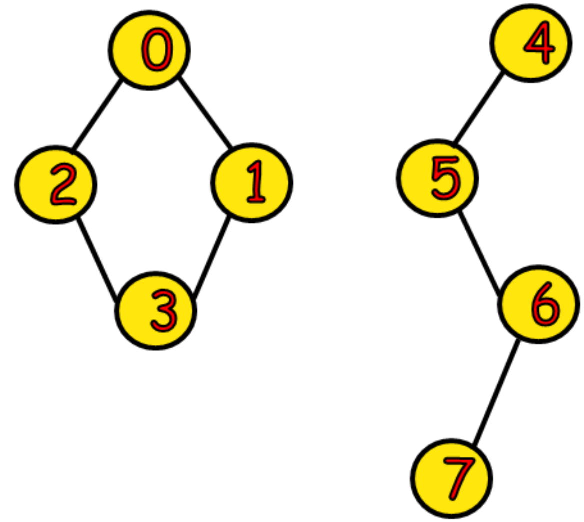
Adjacent Matrix

- Let G(V, E) be a graph with n vertices, $n \geq 1$. The adjacency matrix of G is a two-dimensional nxn array, A.
  - A[i][j] = 1 iff the edge (i, j) is in E(G).
  - The adjacency matrix for a undirected graph is symmetric, it may not be the case for a directed graph.
- For an undirected graph the degree of any vertex i is its row sum.
- For a directed graph, the row sum is the out-degree and the column sum is the in-degree.

Adjacency Matrices







$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

(a) G_1

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \end{array} \begin{array}{c} \begin{array}{ccc} 0 & 1 & 2 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \end{array}$$

(b) G_3

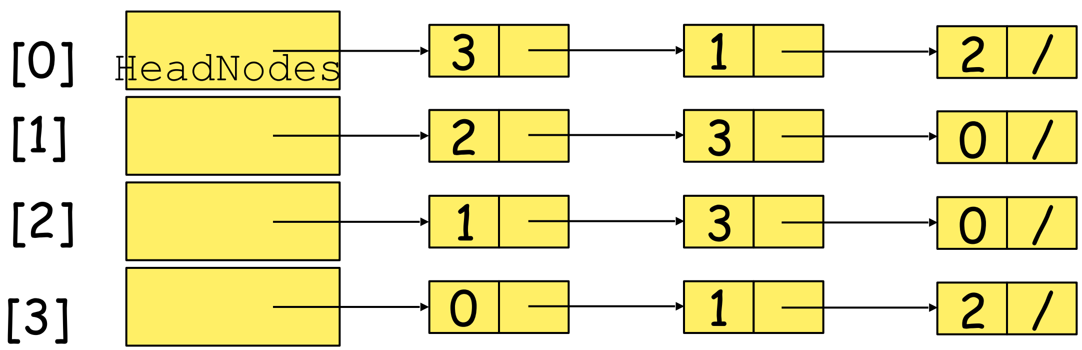
$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{array}{c} \begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

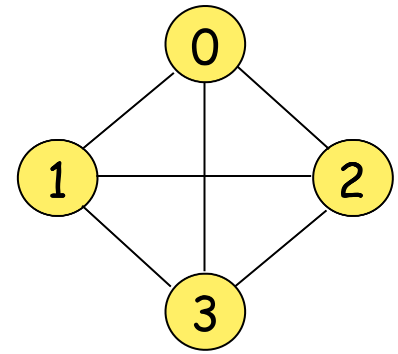
(c) G_4

Adjacency Lists

- Instead of using a matrix to represent the adjacency of a graph, we can use n linked lists to represent the n rows of the adjacency matrix.
- Each node in the linked list contains two fields: data and link.
  - data: contain the indices of vertices adjacent to a vertex i.
  - Each list has a head node.
- For an undirected graph with n vertices and e edges, we need n head nodes and 2e list nodes.
- The degree of any vertex may be determined by counting the number nodes in its adjacency list.
- The number of edges in G can be determined in $O(n + e)$.
- For a directed graph (also called digraph),
  - the out-degree of any vertex can be determined by counting the number of nodes in its adjacency list.
  - the in-degree of any vertex can be obtained by keeping another set of lists called inverse adjacency lists.

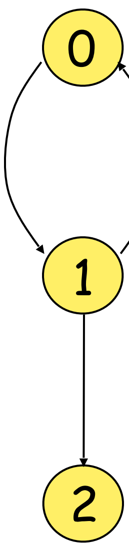
Adjacent Lists

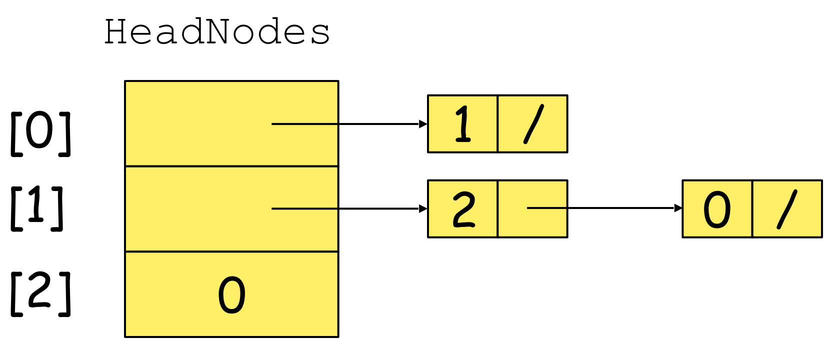




(a) G_1

$$
\begin{array}{c|cccc}
 & 0 & 1 & 2 & 3 \\
\hline
0 & 0 & 1 & 1 & 1 \\
1 & 1 & 0 & 1 & 1 \\
2 & 1 & 1 & 0 & 1 \\
3 & 1 & 1 & 1 & 0
\end{array}
$$



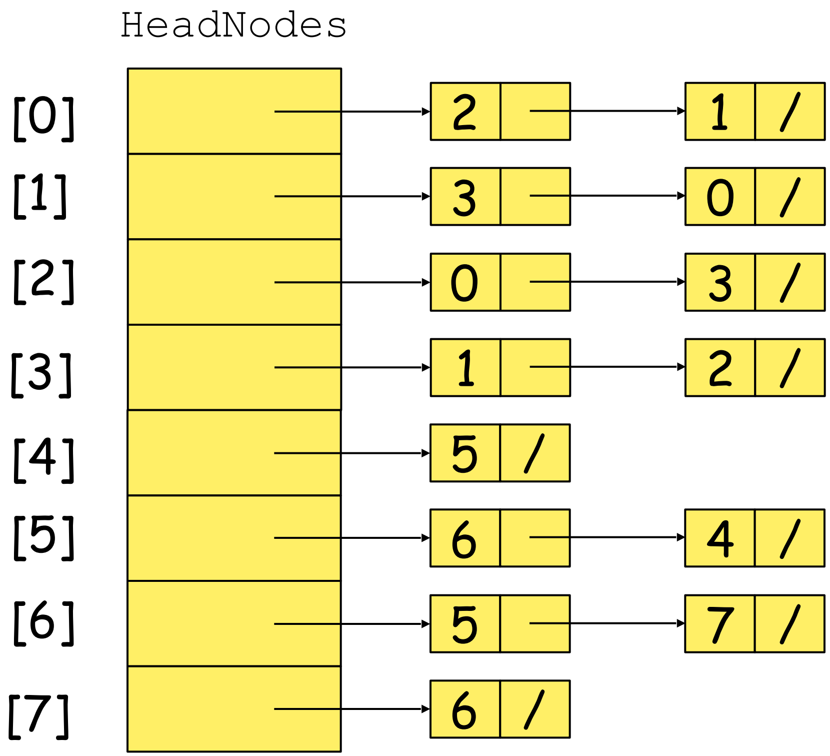


(b) G_3

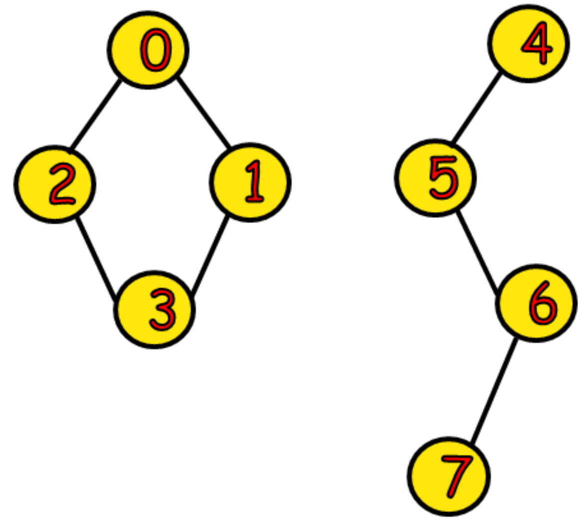
$$
\begin{array}{c|ccc}
 & 0 & 1 & 2 \\
\hline
0 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 \\
2 & 0 & 0 & 0
\end{array}
$$

(b) G_3

Adjacent Lists (Cont.)



(c) G_4



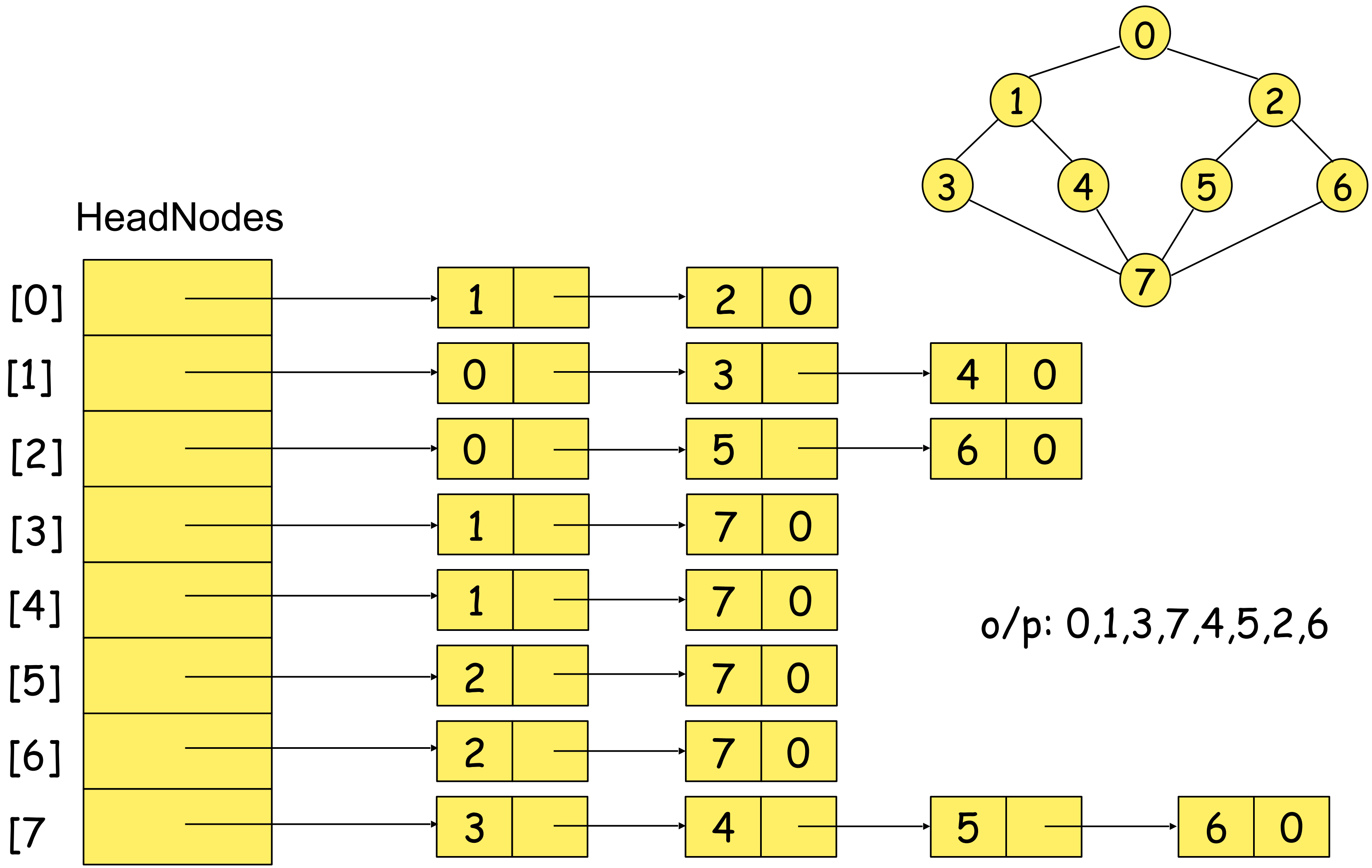
Graph Operations

- A general operation on a graph *G* is to visit all vertices in *G* that are reachable from a vertex v.
  - Depth-first search
  - Breadth-first search

Depth-First Search

- Starting from vertex v, an unvisited vertex w adjacent to v is selected and a depth-first search from w is initiated.
- When the search operation has reached a vertex u such that all its adjacent vertices have been visited, we back up to the last vertex visited that has an unvisited vertex w adjacent to it and initiate a depth-first search from w again.
- The above process repeats until no unvisited vertex can be reached from any of the visited vertices.

Graph G and Its Adjacency Lists



DFS Algorithm

/*Given an undirected graph G = (V,E) with n vertices and an array VISITED(n) initially set to False, this algorithm visits all vertices reachable from v. G and VISITED are global.*/

```
int Visited[]
Algorithm DFS(v)
{
        cout<<v<<endl;
   VISITED[v] = True
   for each vertex w adjacent to v do
        if VISITED[w] =False then
                call DFS(w)
}
```

Analysis of DFS

- If G is represented by its adjacency lists, the DFS time complexity is O(e).
- If G is represented by its adjacency matrix, then the time complexity to complete DFS is $O(n^2)$.

Breadth-First Search

- Starting from a vertex v, visit all unvisited vertices adjacent to vertex v.
- Unvisited vertices adjacent to these newly visited vertices are then visited, and so on.
- If an adjacency matrix is used, the BFS complexity is $O(n^2)$.
- If adjacency lists are used, the time complexity of BFS is $O(e)$.

o/p: 0,1,2,3,4,5,6,7

```
void bfs(int v)
{
  node_pointer w;
  cout<<v;
  visited[v] = TRUE;
  q.Insert(v);
   while (!q.IsEmpty()) {
    v= q.Delete();
    for (w=graph[v]; w; w=w->link)
      if (!visited[w->vertex]) {
            cout<<w->vertex;
            visited[w->vertex] = TRUE;
            q.Insert( w->vertex);
       }
   }
}
```