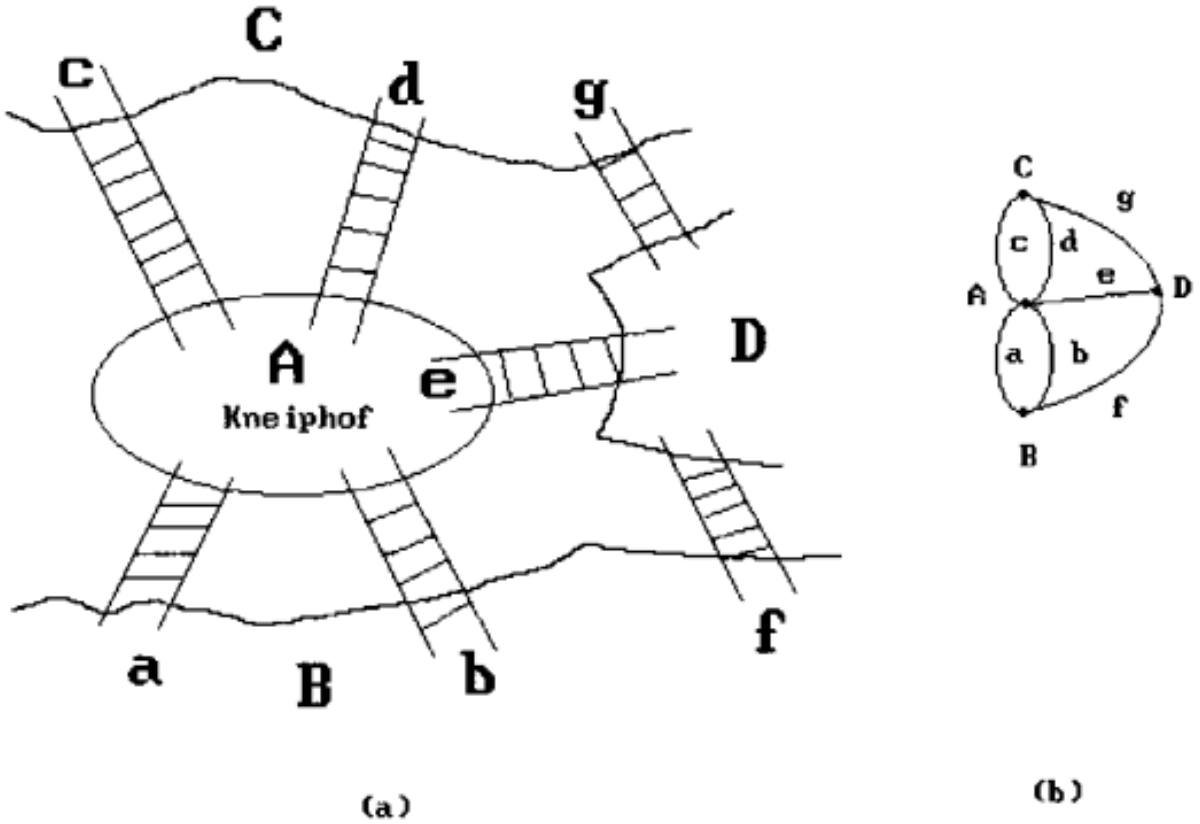
Module 5 Graphs

The Origin of Graph Theory – The Koenigsberg Bridge Problem

- The first recorded use of *graphs* dates back to 1736 by Leonhard Euler.
- The problem was based on the city of Koenigsberg, divided by the Pregel River.
- There were four land areas (A, B, C, D) connected by seven bridges (a–g).
- Problem Statement:
- Starting from any land area, is it possible to cross each bridge exactly once and return to the starting point?

The bridges of Koenigsber



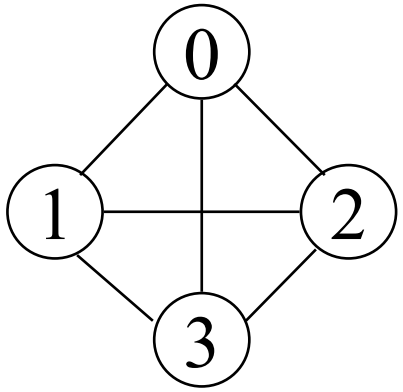
Euler's Solution and Significance

- Euler represented the problem using a graph (multigraph):
- Vertices → Land areas (A, B, C, D)
- Edges → Bridges (a–g)
- He proved that no such walk exists where every bridge is crossed once and returns to the start.
- Reason: More than two vertices have an odd degree (odd number of bridges).
- Significance:
- This discovery marked the birth of graph theory.
- Euler's reasoning applies to all graphs, not just this problem.

Graph Basics

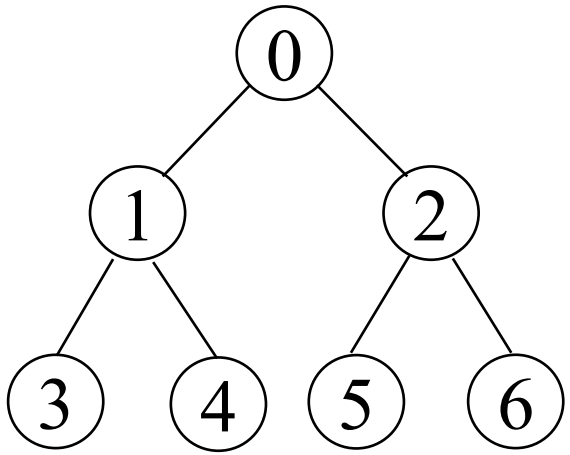
- A graph, G=(V, E), consists of two sets:
  - a finite set of vertices(V), and
  - a finite, possibly empty set of edges(E)
  - V(G) and E(G) represent the sets of vertices and edges of G, respectively
- Undirected graph
  - The pairs of vertices representing any edges is unordered
  - e.g., (v_0, v_1) and (v_1, v_0) represent the same edge
- Directed graph
  - Each edge as a directed pair of vertices
  - e.g. $<v_0, v_1>$ represents an edge, v_0 is the tail and v_1 is the head

Examples for Graph



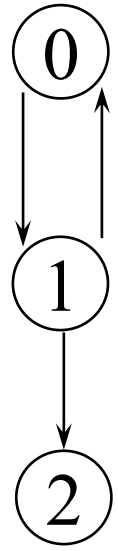
G_1

complete graph



G_2

incomplete graph



G_3

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

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

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

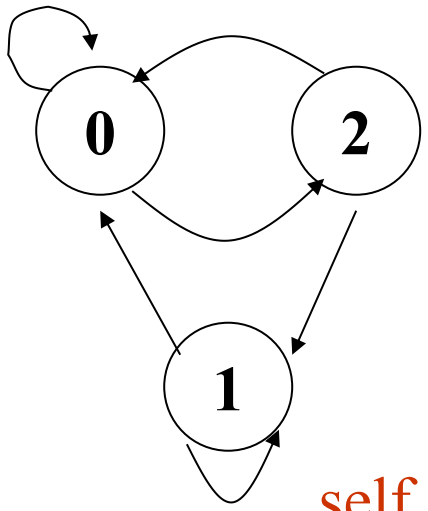
Complete Graph

- Definition: A complete graph is a graph that has the maximum number of edges.
- Undirected Graph: For an undirected graph with n vertices, the maximum number of edges is: $n(n-1)/2$
- Directed Graph: For a directed graph with n vertices, the maximum number of edges is: $n(n-1)$
- Example: Graph G1 (from the previous slide) is a complete graph.

Adjacent and Incident

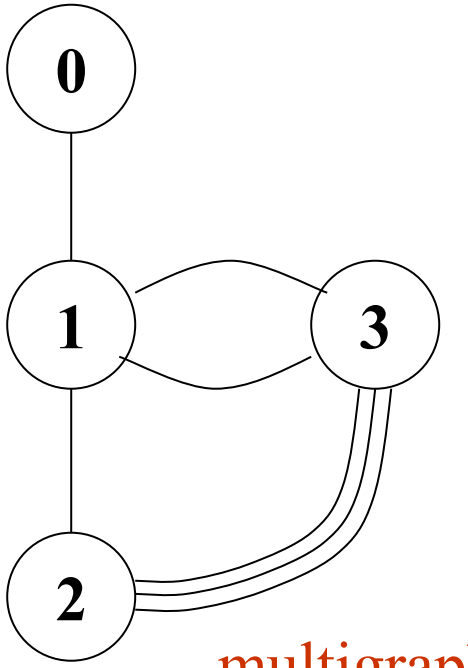
- If (v_0, v_1) is an edge in an undirected graph,
  - v_0 and v_1 are adjacent
  - The edge (v_0, v_1) is incident on vertices v_0 and v_1
- If $<v_0, v_1>$ is an edge in a directed graph
  - v_0 is adjacent to v_1, and v_1 is adjacent from v_0
  - The edge $<v_0, v_1>$ is incident on v_0 and v_1

Example of a graph with feedback loops and a multigraph



self edge

(a)



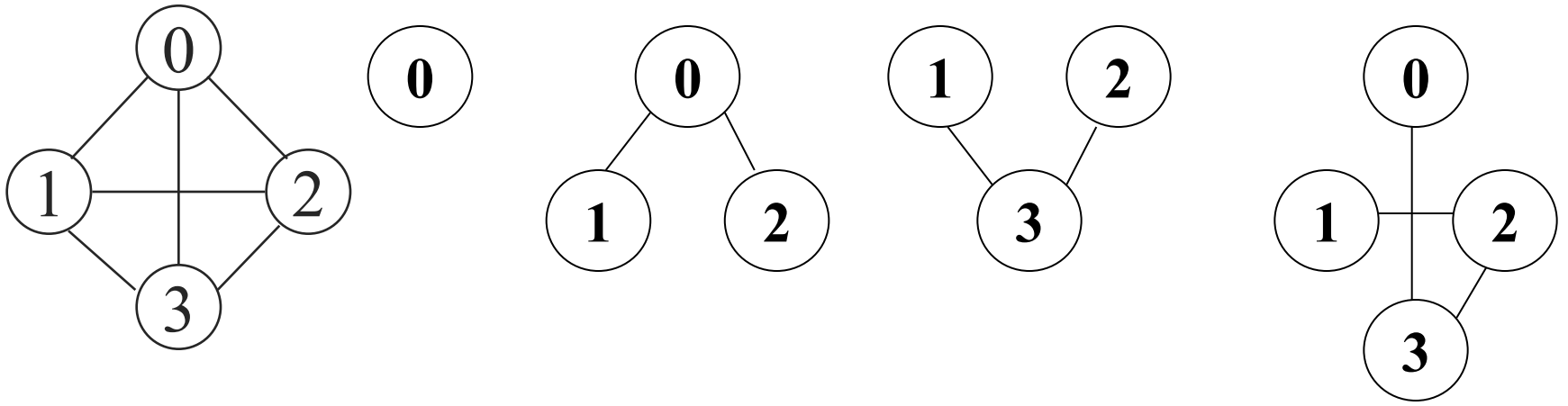
(b)

multigraph:
multiple occurrences
of the same edge

Subgraph and Path

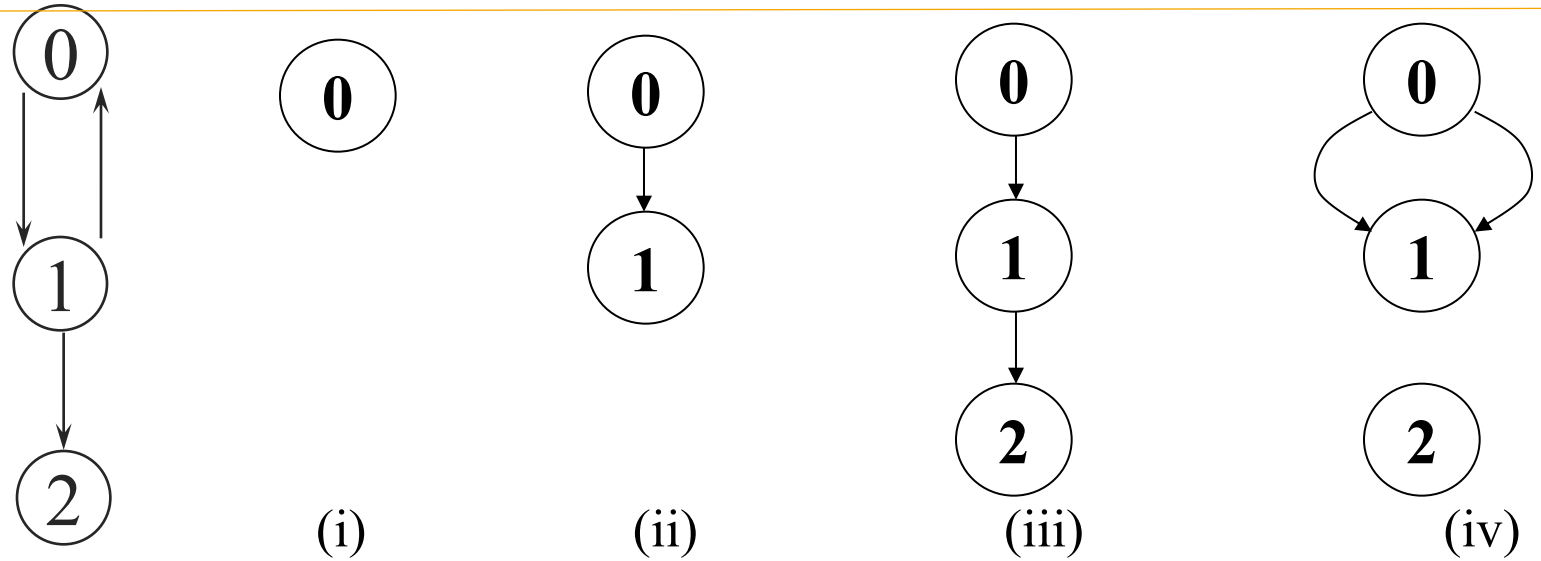
- A subgraph of G is a graph G′ such that V(G′) is a subset of V(G) and E(G′) is a subset of E(G)
- A path from vertex v_p to vertex v_q in a graph G, is a sequence of vertices, v_p, v_{i1}, v_{i2}, ..., v_{in}, v_q, such that (v_p, v_{i1}), (v_{i1}, v_{i2}), ..., (v_{in}, v_q) are edges in an undirected graph.
- The length of a path is the number of edges on it

Subgraphs of G_1 and G_3



G_1 (i) (ii) (iii) (iv)

(a) Some of the subgraph of G_1



(i) (ii) (iii) (iv)

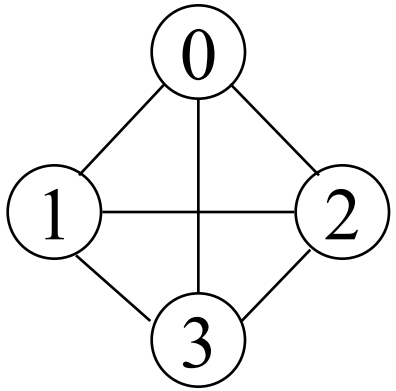
(b) Some of the subgraph of G_3

G_3

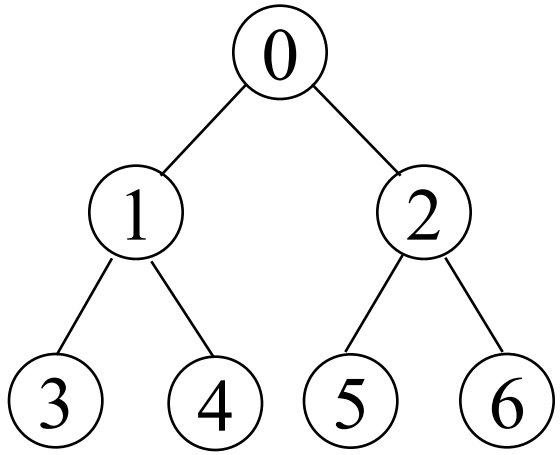
Simple Path and Style

- A simple path is a path in which all vertices, except possibly the first and the last, are distinct
- A cycle is a simple path in which the first and the last vertices are the same
- In an undirected graph G, ***two vertices***, v_0 and v_1, are connected if there is a path in G from v_0 to v_1
- An undirected ***graph*** is connected if, for every pair of distinct vertices v_i, vj, there is a path from vi to v_j

Connected



G_1



G_2

tree (acyclic graph)

Degree

- The degree of a vertex is the number of edges incident to that vertex

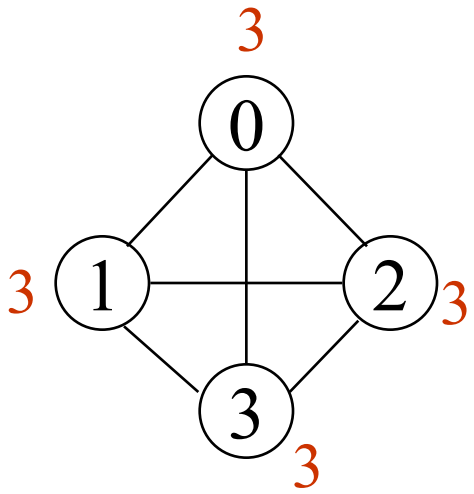
- For directed graph,
  - the in-degree of a vertex *v* is the number of edges that have *v* as the head
  - the out-degree of a vertex *v* is the number of edges that have *v* as the tail

  - if d_i is the degree of a vertex *i* in a graph *G* with *n* vertices and *e* edges, the number of edges is

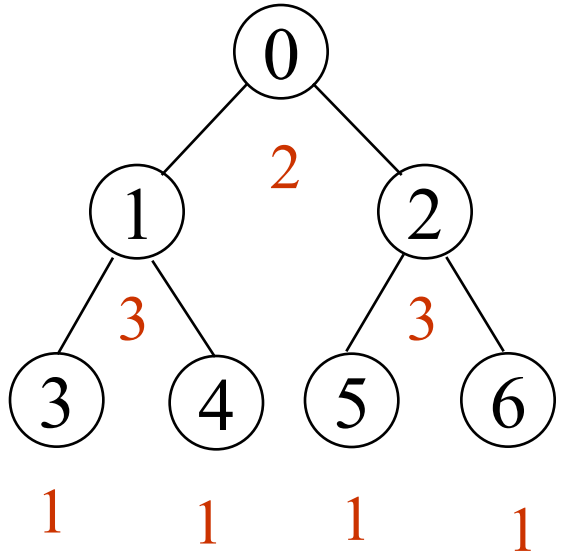
$$e = (\sum_{0}^{n-1} d_i) / 2$$

undirected graph

degree



G_1

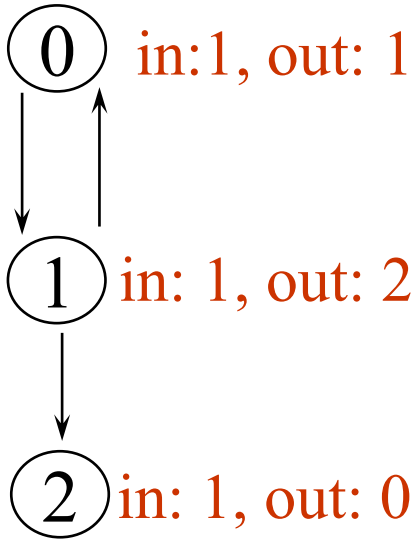


G_2

directed graph

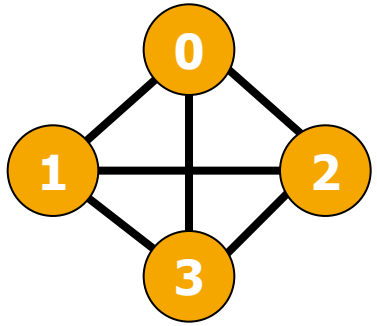
in-degree

out-degree

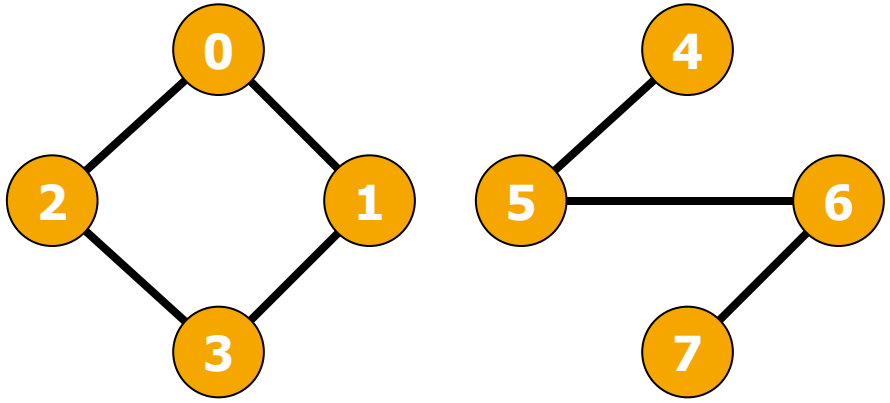


G_3

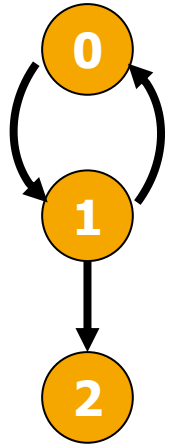
- Adjacency matrices
- Adjacency lists



G1

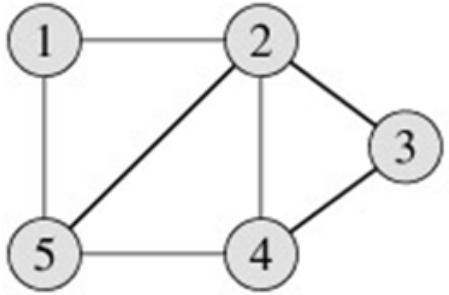


G4



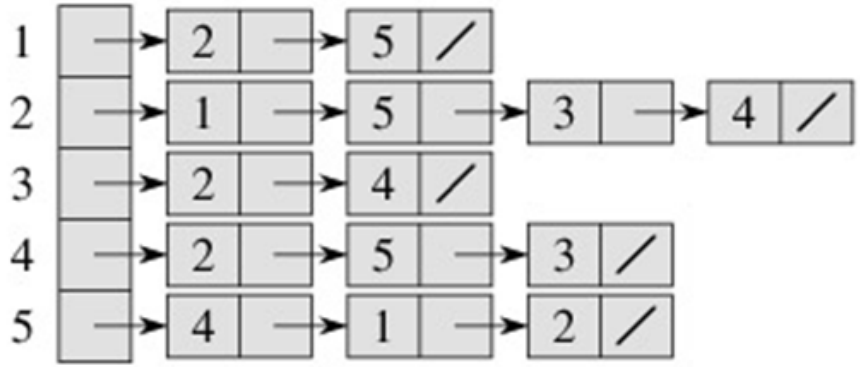
G3

Graph representation – undirected



(a)

Graph



(b)

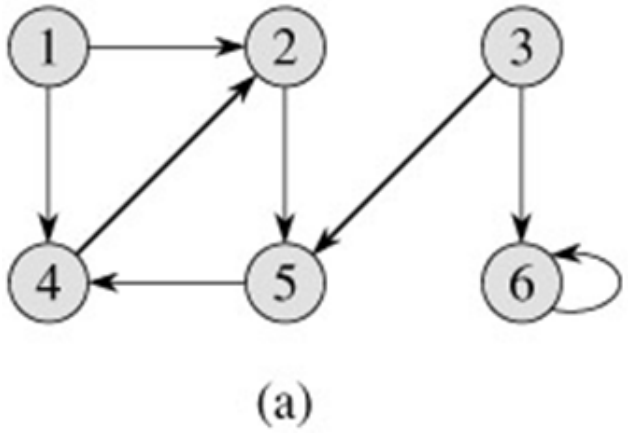
Adjacency list

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

(c)

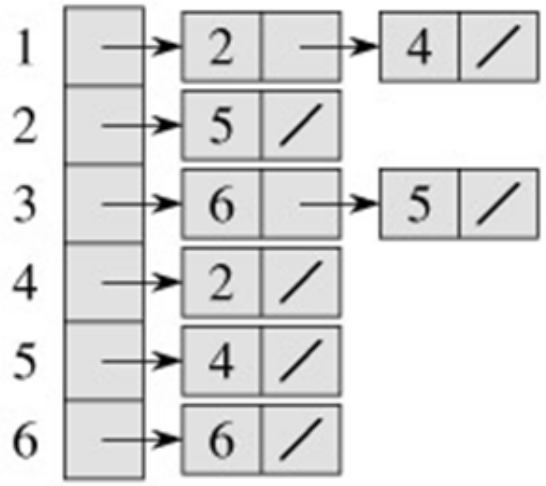
Adjacency matrix

Graph representation – directed



(a)

Graph



(b)

Adjacency list

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

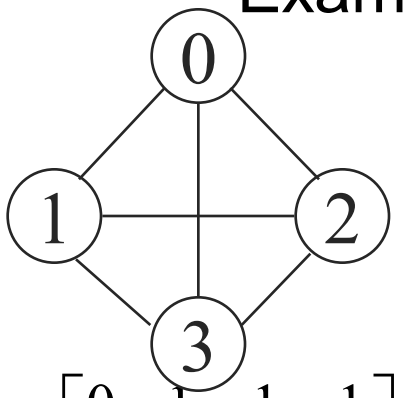
(c)

Adjacency matrix

Adjacency Matrix

- Let G=(V,E) be a graph with n vertices.
- The adjacency matrix of G is a two-dimensional n by n array, say adj_mat
- If the edge (vi, vj) is in E(G), adj_mat[i][j]=1
- If there is no such edge in E(G), adj_mat[i][j]=0
- The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric

Examples for Adjacency Matrix



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

G_1

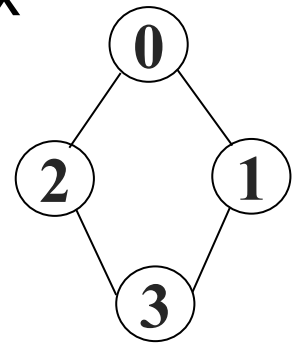
$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

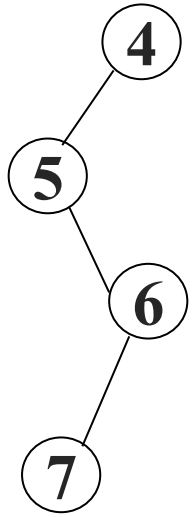
G_2

symmetric

undirected: $n^2/2$

directed: n^2

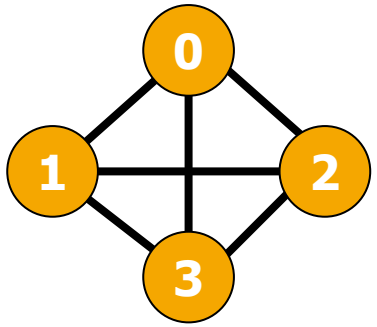




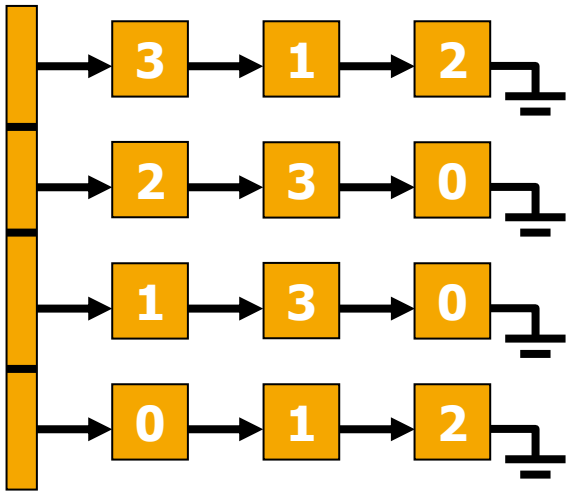
$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

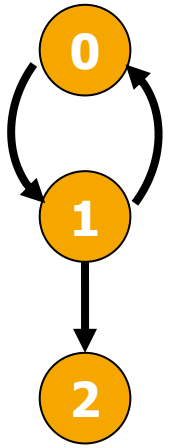
G_4

Adjacency lists

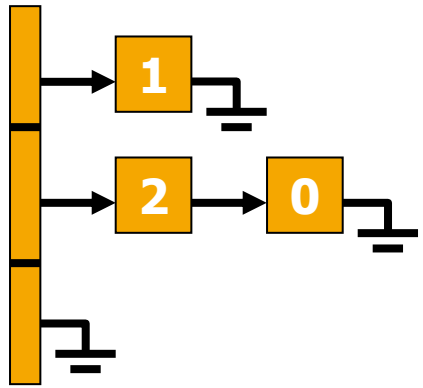


G1





G3



Graph representation using adjacency lists

```
// Node structure for adjacency list
typedef struct node {
    int vertex;
    struct node* next;
} Node;
// Array of adjacency lists (graph)
Node* graph[MAX_VERTICES];
// Number of vertices currently in use
int n = 0;
// Function to create a new node
Node* createNode(int vertex) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->vertex = vertex;
    newNode->next = NULL;
    return newNode;
}
```

Graph representation using adjacency lists

```
// Function to add an edge to the graph (undirected)
void addEdge(int src, int dest) {
    Node* newNode = createNode(dest);
    newNode->next = graph[src];
    graph[src] = newNode;
    // For undirected graph, add an edge from dest to src also
    newNode = createNode(src);
    newNode->next = graph[dest];
    graph[dest] = newNode;  }
```

```
// Function to display the adjacency list
void displayGraph() {
    for (int i = 0; i < n; i++) {
        Node* temp = graph[i];
        printf("Vertex %d: ", i);
        while (temp) {
            printf("%d -> ", temp->vertex);
            temp = temp->next;       }
        printf("NULL\n");    }
}
```

Graph representation using adjacency lists

```
int main() {
    printf("Enter the number of vertices: ");
    scanf("%d", &n);
    // Initialize graph
    for (int i = 0; i < n; i++) {
        graph[i] = NULL;    }
    int edges;
    printf("Enter the number of edges: ");
    scanf("%d", &edges);
    printf("Enter each edge (source destination):\n");
    for (int i = 0; i < edges; i++) {
        int src, dest;
        scanf("%d %d", &src, &dest);
        addEdge(src, dest);    }
    printf("\nAdjacency List Representation:\n");
    displayGraph();
    return 0;}
```

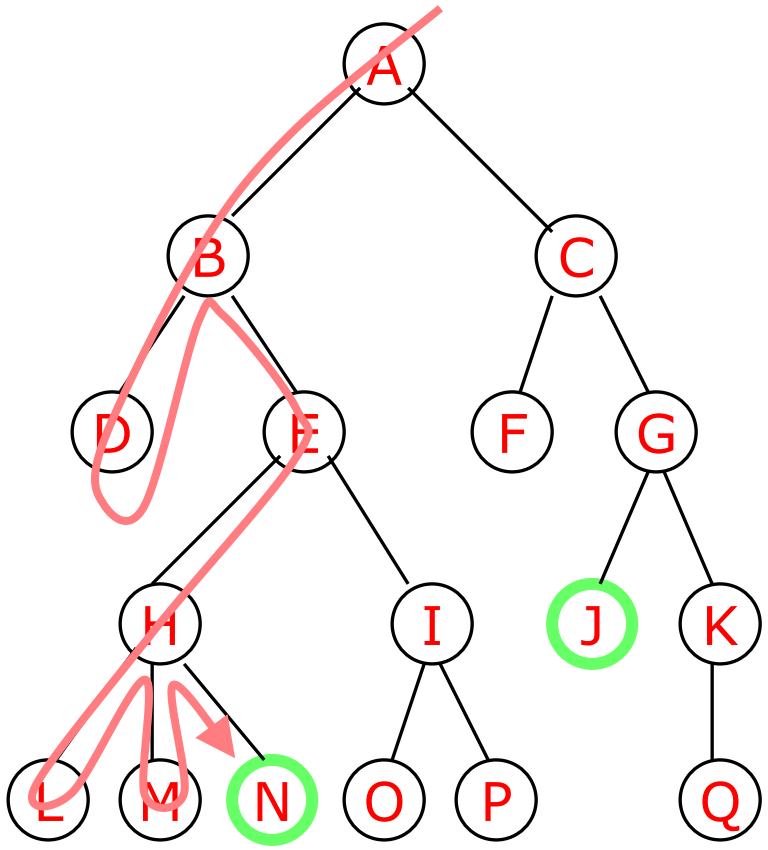
Some Graph Operations

- Traversal
  Given G=(V,E) and vertex v, find all w∈V, such that w connects v.
  - Depth First Search (DFS)
    preorder tree traversal
  - Breadth First Search (BFS)
    level order tree traversal

Depth First Search (DFS) – Concept Overview

- DFS begins by visiting a starting vertex (v).
- Visiting means performing an operation (e.g., printing the vertex).
- From vertex *v*, we move to an unvisited adjacent vertex (w).
- The current position in *v*'s adjacency list is stored on a stack.
- The search continues deeper until a vertex (*M*) has no unvisited adjacent vertices.
- At this point, we backtrack by removing a vertex from the stack and continue.

Depth-first search



- DFS explores a path all the way to a leaf before backtracking and exploring another path
- For example, after searching A, then B, then D, the search backtracks and tries another path from B
- Node are explored in the order A B D E H L M N I O P C F G J K Q

DFS Implementation

```
#define MAX_VERTICES 50
#define FALSE 0
#define TRUE 1
// Node structure for adjacency list
typedef struct node {
    int vertex;
    struct node* next;
} Node;
// Graph representation (array of adjacency lists)
Node* graph[MAX_VERTICES];
// Visited array to mark visited vertices
short int visited[MAX_VERTICES];
int n; // Number of vertices
// Function to create a new node
Node* createNode(int vertex) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->vertex = vertex;
    newNode->next = NULL;
    return newNode; }
```

DFS Implementation

```
// Function to add an edge to the graph (undirected)
void addEdge(int src, int dest) {
    Node* newNode = createNode(dest);
    newNode->next = graph[src];
    graph[src] = newNode;
    // For undirected graph, also add the reverse edge
    newNode = createNode(src);
    newNode->next = graph[dest];
    graph[dest] = newNode;}
```

```
// Depth First Search function
void dfs(int v) {
    Node* w;
    visited[v] = TRUE;
    printf("%5d", v);
    for (w = graph[v]; w != NULL; w = w->next) {
        if (!visited[w->vertex])
            dfs(w->vertex);
    }  }
```

DFS Implementation

```
// Function to initialize visited array
void initializeVisited() {
    for (int i = 0; i < n; i++)
        visited[i] = FALSE;}
int main() {
    int edges, src, dest, startVertex;
    printf("Enter number of vertices: ");
    scanf("%d", &n);
```

```
// Initialize graph and visited array
    for (int i = 0; i < n; i++) {
        graph[i] = NULL;
        visited[i] = FALSE;  }
printf("Enter number of edges: ");
    scanf("%d", &edges);
    printf("Enter each edge (source destination):\n");
    for (int i = 0; i < edges; i++) {
        scanf("%d %d", &src, &dest);
        addEdge(src, dest);    }
```

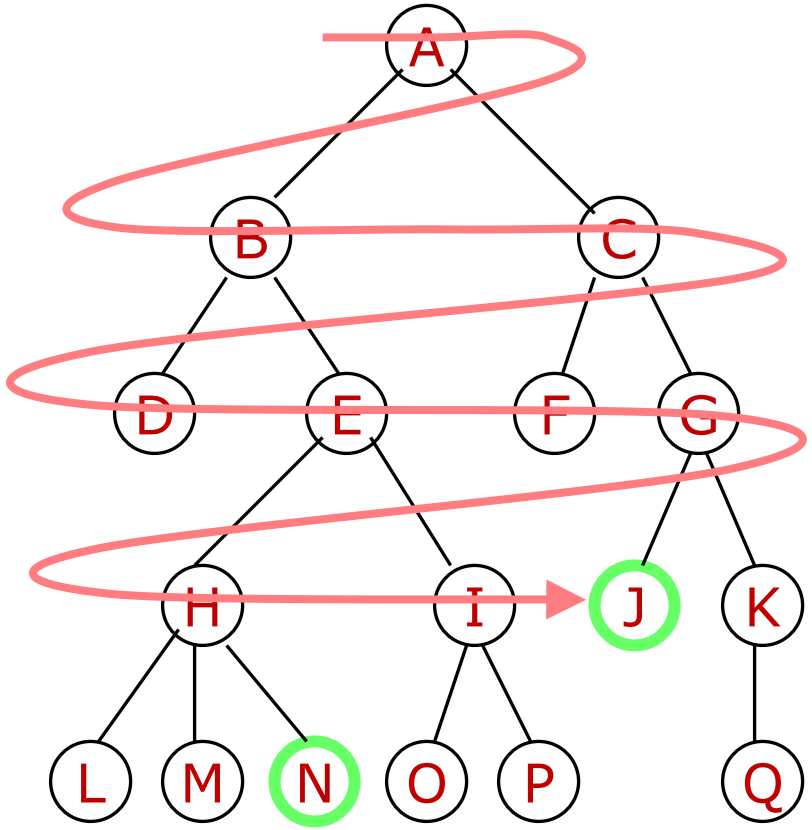
DFS Implementation

- printf("Enter starting vertex for DFS: ");
- scanf("%d", &startVertex);
- printf("\nDepth First Search Traversal:\n");
- dfs(startVertex);
- printf("\n");
- return 0;
- }

Breadth First Search (BFS) – Concept Overview

- BFS starts from a given vertex v and marks it as *visited*.
- It visits all vertices adjacent to *v* (neighbors of v).
- After visiting all neighbors, BFS proceeds to the next vertex in the queue.
- This process continues level by level, visiting all vertices reachable from the start vertex.
- BFS ensures that the shortest path (in terms of edges) from the start vertex is found first.

Breadth-first search



- A breadth-first search (BFS) explores nodes nearest the root before exploring nodes further away
- For example, after searching A, then B, then C, the search proceeds with D, E, F, G
- Node are explored in the order A B C D E F G H I J K L M N O P Q

BFS Implementation

- #define MAX_VERTICES 50
- #define FALSE 0
- #define TRUE 1
- // Node structure for adjacency list
- typedef struct node {
- int vertex;
- struct node* next;
- } Node;
- // Global graph and visited array
- Node* graph[MAX_VERTICES];
- short int visited[MAX_VERTICES];
- int n; // Number of vertices
- // Queue node structure
- typedef struct queue {
- int vertex;
- struct queue* link;
- } QueueNode;
- typedef struct queue* queue_pointer;

BFS Implementation

- void addq(queue_pointer* front, queue_pointer* rear, int vertex);
- int deleteq(queue_pointer* front);
- void bfs(int v);
- Node* createNode(int vertex);
- void addEdge(int src, int dest);

---------- QUEUE FUNCTIONS ----------

- // Enqueue operation
- void addq(queue_pointer* front, queue_pointer* rear, int vertex) {
- queue_pointer temp =(queue_pointer)malloc(sizeof(QueueNode));
- temp->vertex = vertex;
- temp->link = NULL;
- if (*rear) {
- (*rear)->link = temp;
- } else {
- *front = temp; }
- *rear = temp; }

BFS Implementation

```
// Dequeue operation
int deleteq(queue_pointer* front) {
    queue_pointer temp = *front;
    int item;
    if (!temp)
        return -1;  // queue empty
    item = temp->vertex;
    *front = temp->link;
    free(temp);
    return item; }
```

```
Node* createNode(int vertex) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->vertex = vertex;
    newNode->link = NULL;
    return newNode;}
void addEdge(int src, int dest) {
    Node* newNode = createNode(dest);
    newNode->link = graph[src];
    graph[src] = newNode;
    newNode = createNode(src);
    newNode->link = graph[dest];
    graph[dest] = newNode;}
```

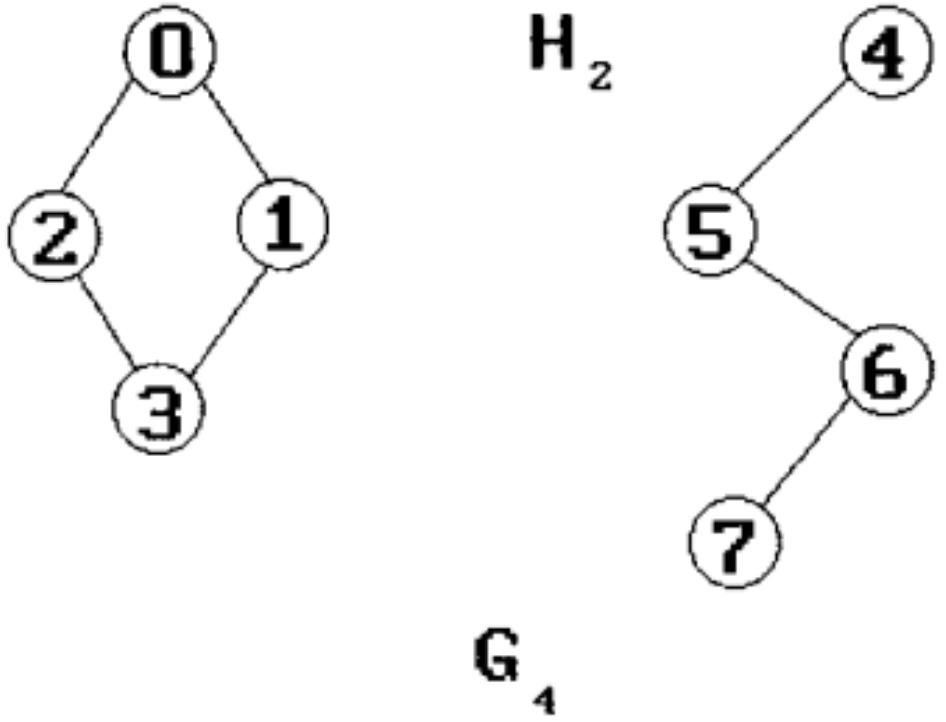
BFS Implementation

```
void bfs(int v) {
    Node* w;
    queue_pointer front = NULL, rear = NULL;
    printf("%5d", v);
    visited[v] = TRUE;
    addq(&front, &rear, v);
    while (front) {
        v = deleteq(&front);
        for (w = graph[v]; w; w = w->link) {

    if (!visited[w->vertex]) {
                printf("%5d", w->vertex);
                visited[w->vertex] = TRUE;
                addq(&front, &rear, w->vertex);
            }
        }
    }
int main()
……..
```

Graph Connectivity Using DFS/BFS

- **Goal:** Determine whether an undirected graph is *connected* – i.e., every vertex is reachable from any other vertex.
- **Method:**
- Perform a DFS(0) or BFS(0) starting from vertex 0.
- After the traversal, check if all vertices have been visited.
- **Example:**
- Applying dfs(0) on graph G4 visits only vertices 0–3, but not 4–7.
- ⇒ Hence, graph G4 is not connected.



Connected Components

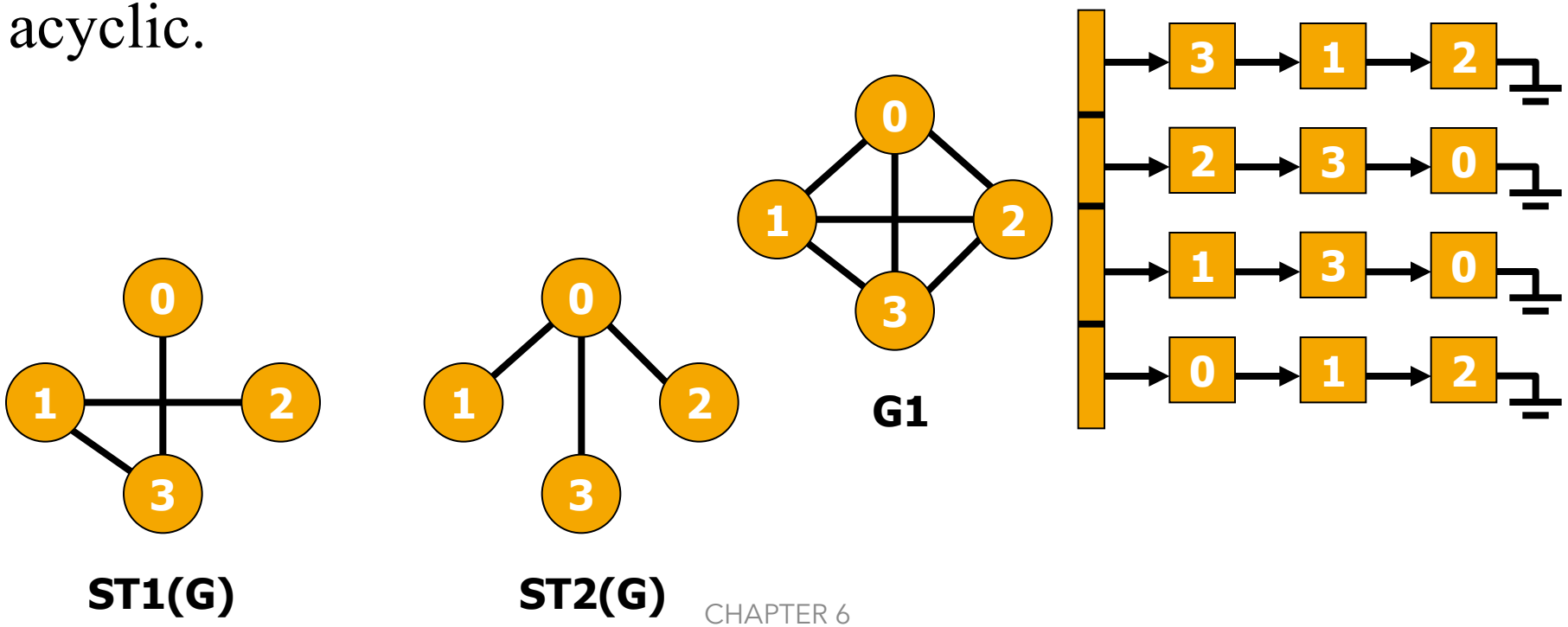
- A **connected component** is a maximal set of vertices where each vertex is reachable from any other within the same set.
- **Procedure:**
- Initialize all vertices as *unvisited*.
- For each vertex *v*:
  - If *v* is unvisited → call **DFS(v)** or **BFS(v)**.
  - All vertices visited in that call form one **connected component**.
- Repeat until all vertices are visited.

Function Example (connected):

```
for (v = 0; v < n; v++)
    if (!visited[v]) {
        printf("Component: ");
        dfs(v);
        printf("\n");
    }
```

Spanning Tree (ST)

- A spanning tree is a minimal subgraph G', such that V(G')=V(G) and G' is connected. Spanning Tree is always acyclic.



Spanning Trees in Connected Graphs

- When a graph G is connected, a Depth First Search (DFS) or Breadth First Search (BFS) starting at any vertex will visit all vertices in *G*.
- The traversal implicitly divides the edges into two sets:
- T (Tree edges): edges used/traversed during the search.
- N (Nontree edges): remaining edges not used in traversal.
- The edges in T form a tree that includes all vertices of *G*.
- Such a tree is called a Spanning Tree –
- A subgraph that includes all vertices of G and a subset of edges that form a tree (no cycles).
- Each connected graph can have multiple spanning trees.

Tree and Nontree Edges – Implementation View

During DFS/BFS traversal:

- When a new vertex `w` is visited from vertex `v`, the edge `(v, w)` is added to the Tree Edge Set (T).
- These edges collectively form a Spanning Tree.

Pseudocode Integration:

- if (!visited[w->vertex]) {
- addEdgeToTree(v, w->vertex); // store (v, w) as tree edge
- dfs(w->vertex); }

Spanning Tree Characteristics:

- Connects all vertices with minimum number of edges (n – 1 edges).
- Contains no cycles.
- May not be unique – different traversals can produce different spanning trees.

A complete graph and three of its spanning tree