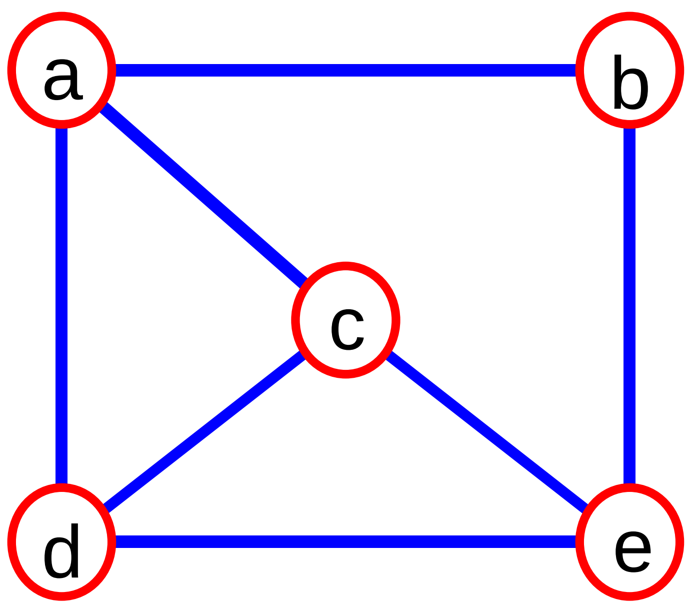
What is a Graph?

- A graph G = (V,E) is composed of:

  V: set of vertices

  E: set of edges connecting the vertices in V
- An edge e = (u,v) is a pair of vertices
- Example:

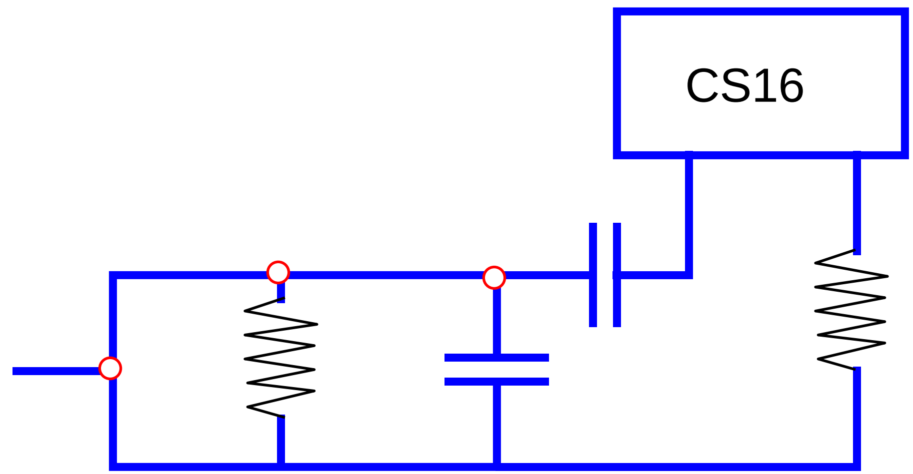


V= {a,b,c,d,e}

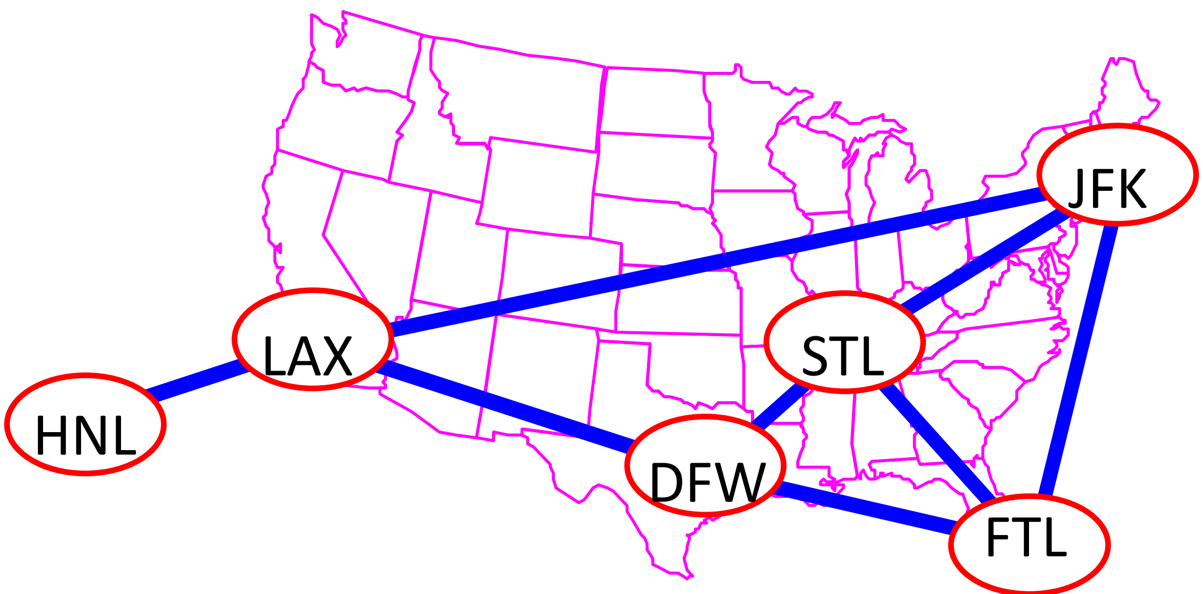
E= {(a,b),(a,c),(a,d),
(b,e),(c,d),(c,e),
(d,e)}

Applications

- electronic circuits

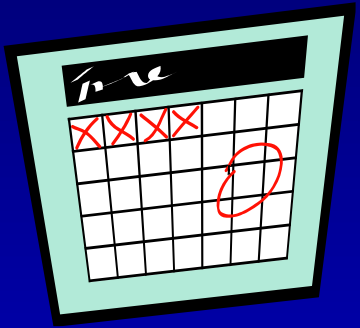


- networks (roads, flights, communications)

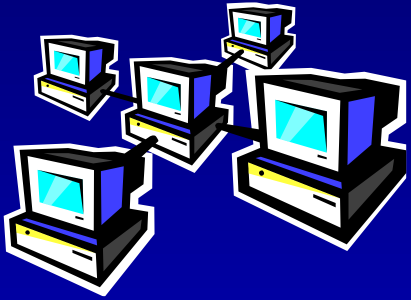


Applications

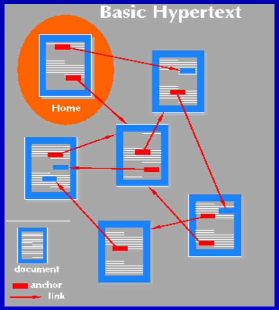
Maps



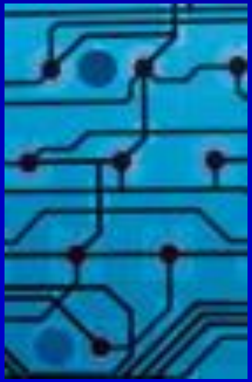
Schedules



Computer networks



Hypertext

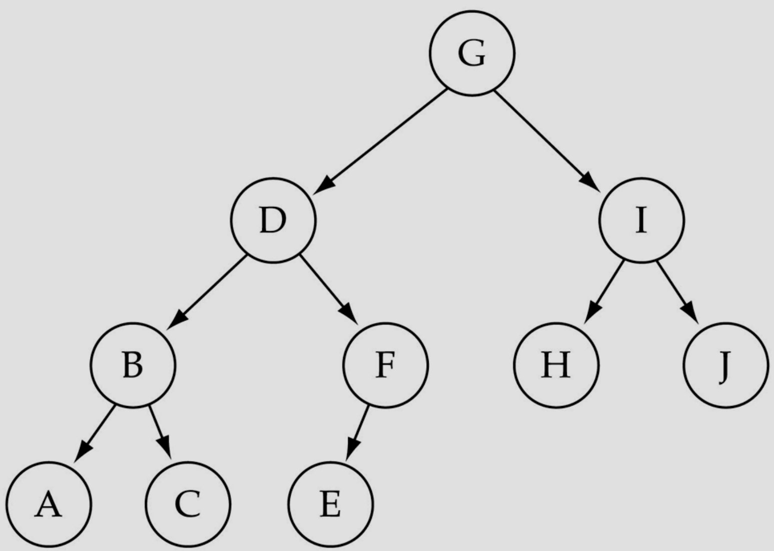


Circuits

Trees vs graphs

- Trees are special cases of graphs!!

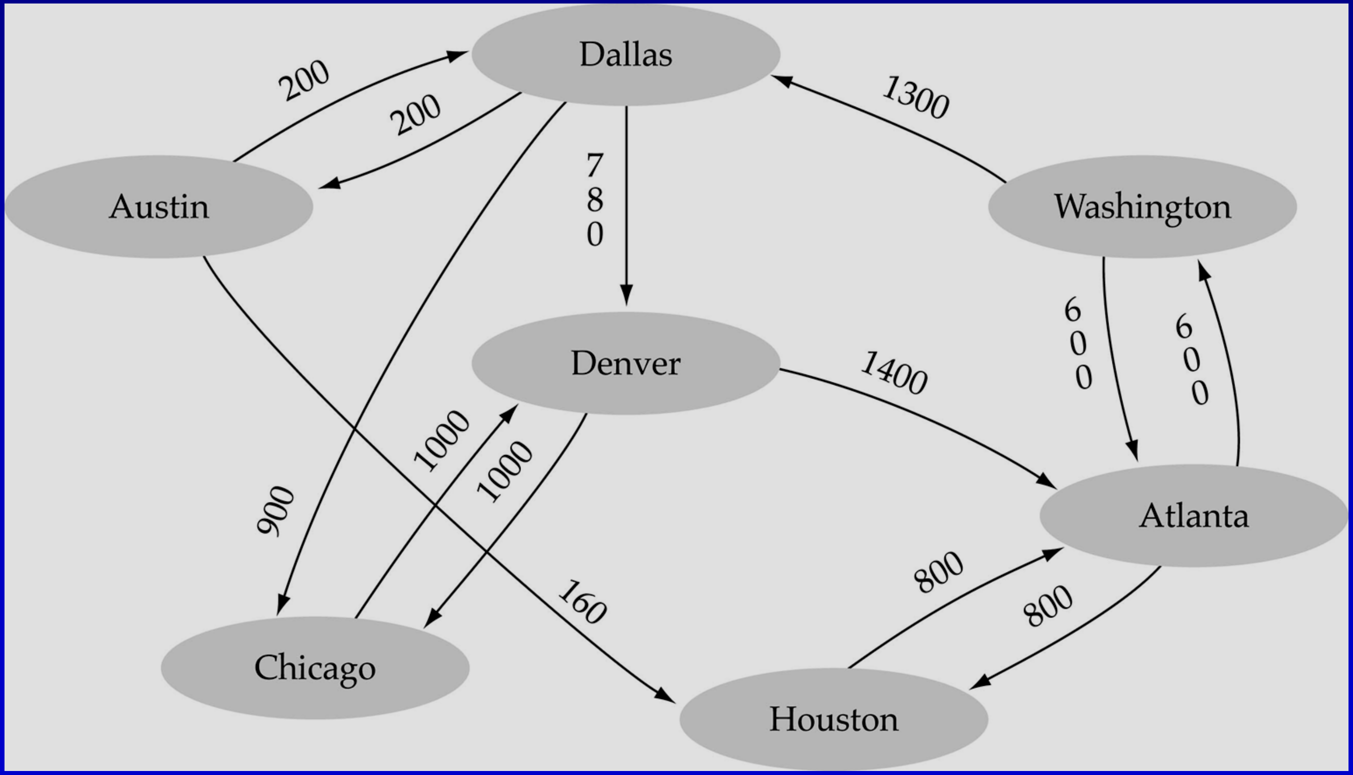
(c) Graph3 is a directed graph.



V(Graph3) = { A, B, C, D, E, F, G, H, I, J }
E(Graph3) = { (G, D), (G, I), (D, B), (D, F) (I, H), (I, J), (B, A), (B, C), (F, E) }

Graph terminology (cont.)

- Weighted graph: a graph in which each edge carries a value



Terminology: 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

Terminology: Degree of a Vertex

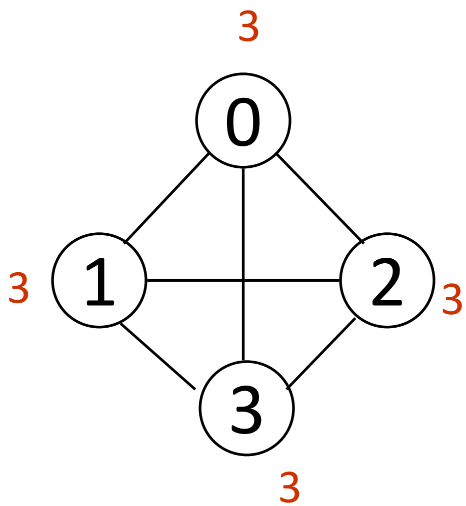
- The degree of a vertex is the number of edges incident to that vertex
- For directed graph,
  - the in-degree of a vertex v is the number of edges that have v as the head
  - the out-degree of a vertex v is the number of edges that have v as the tail
  - if d_i is the degree of a vertex i in a graph G with n vertices and e edges, the number of edges is

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

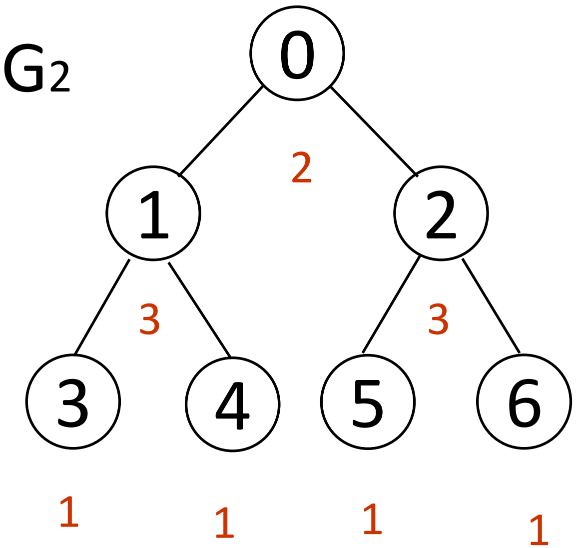
Why? Since adjacent vertices each count the adjoining edge, it will be counted twice

Examples

G_1

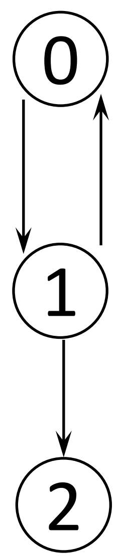


G_2



directed graph

in-degree
out-degree



in:1, out: 1

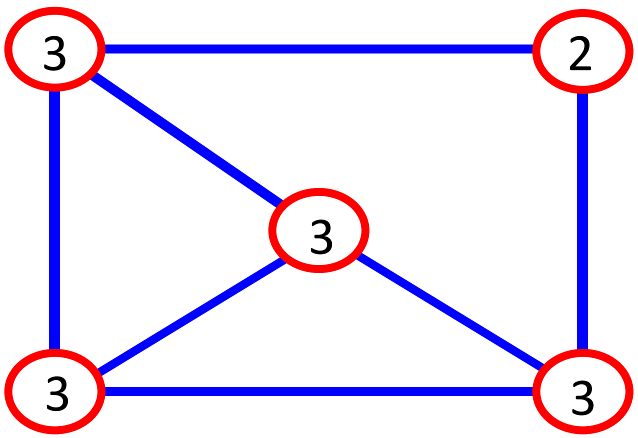
in: 1, out: 2

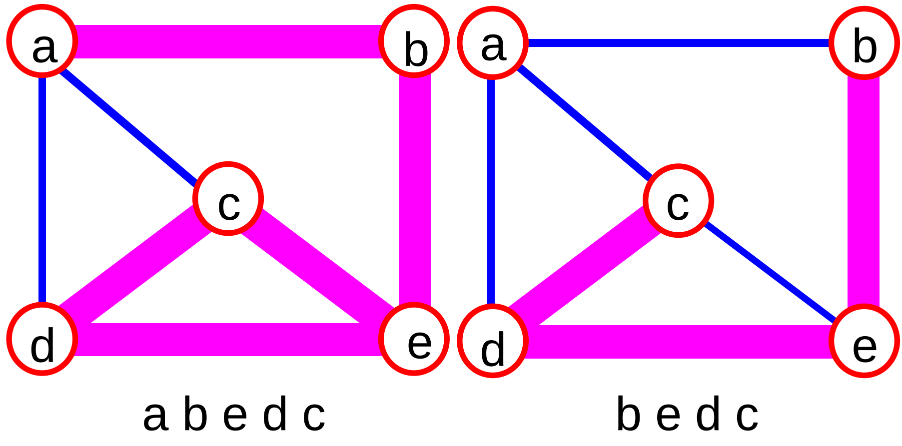
in: 1, out: 0

G_3

Terminology: Path

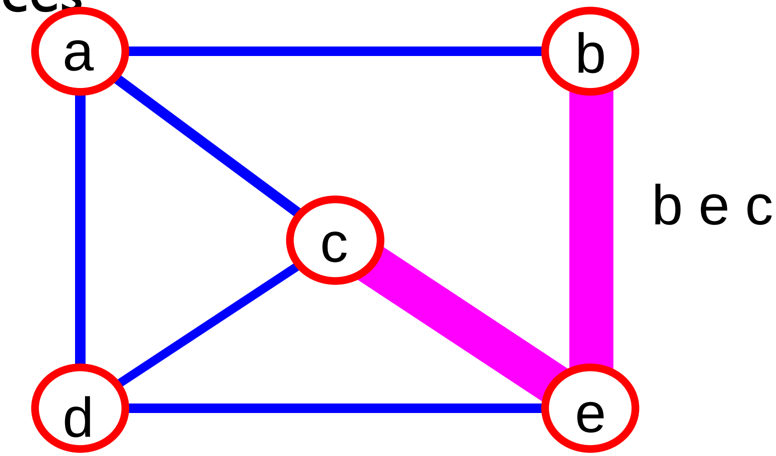
- path: sequence of vertices $v_1, v_2, \ldots v_k$ such that consecutive vertices v_i and v_{i+1} are adjacent.



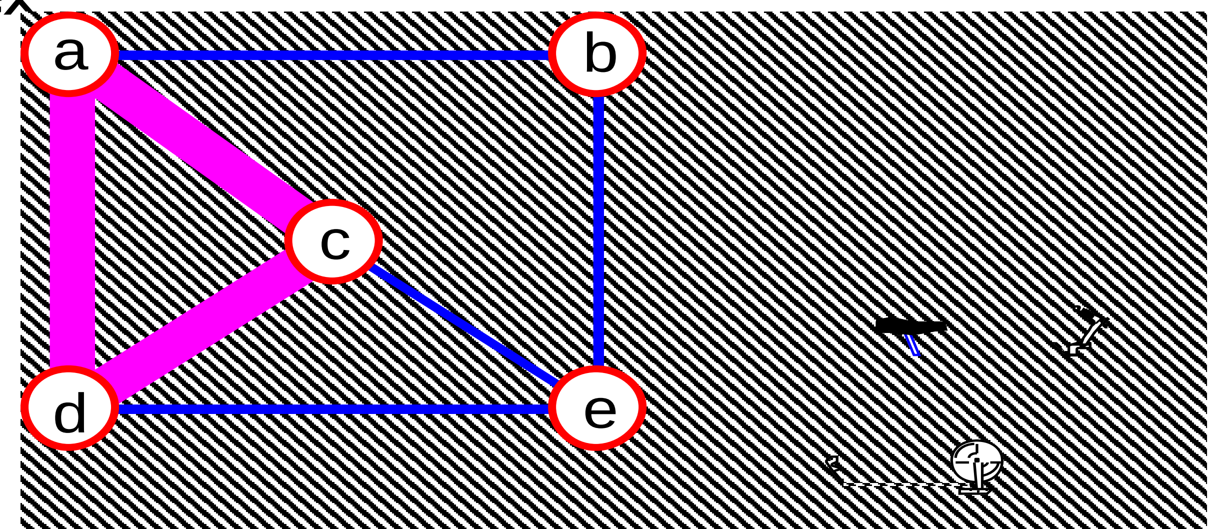


More Terminology

- simple path: no repeated vertices

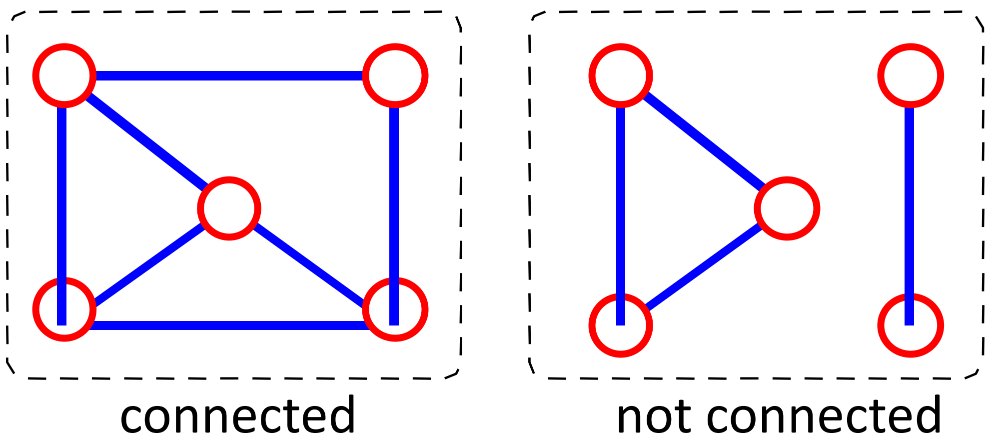


- cycle: simple path, except that the last vertex is the same as the first vertex



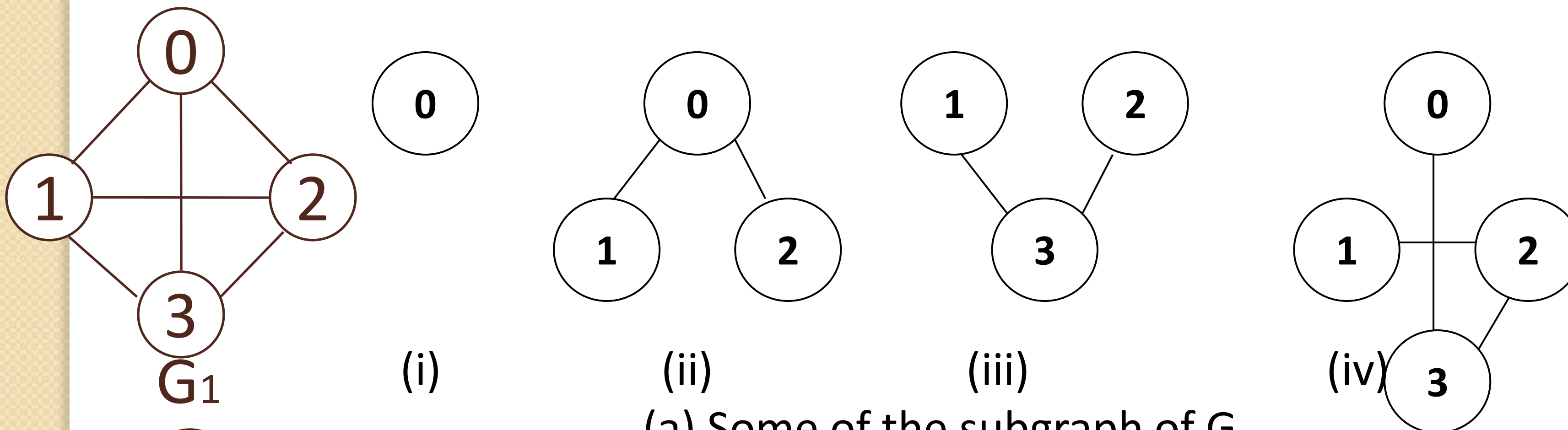
Even More Terminology

- connected graph: any two vertices are connected by some path

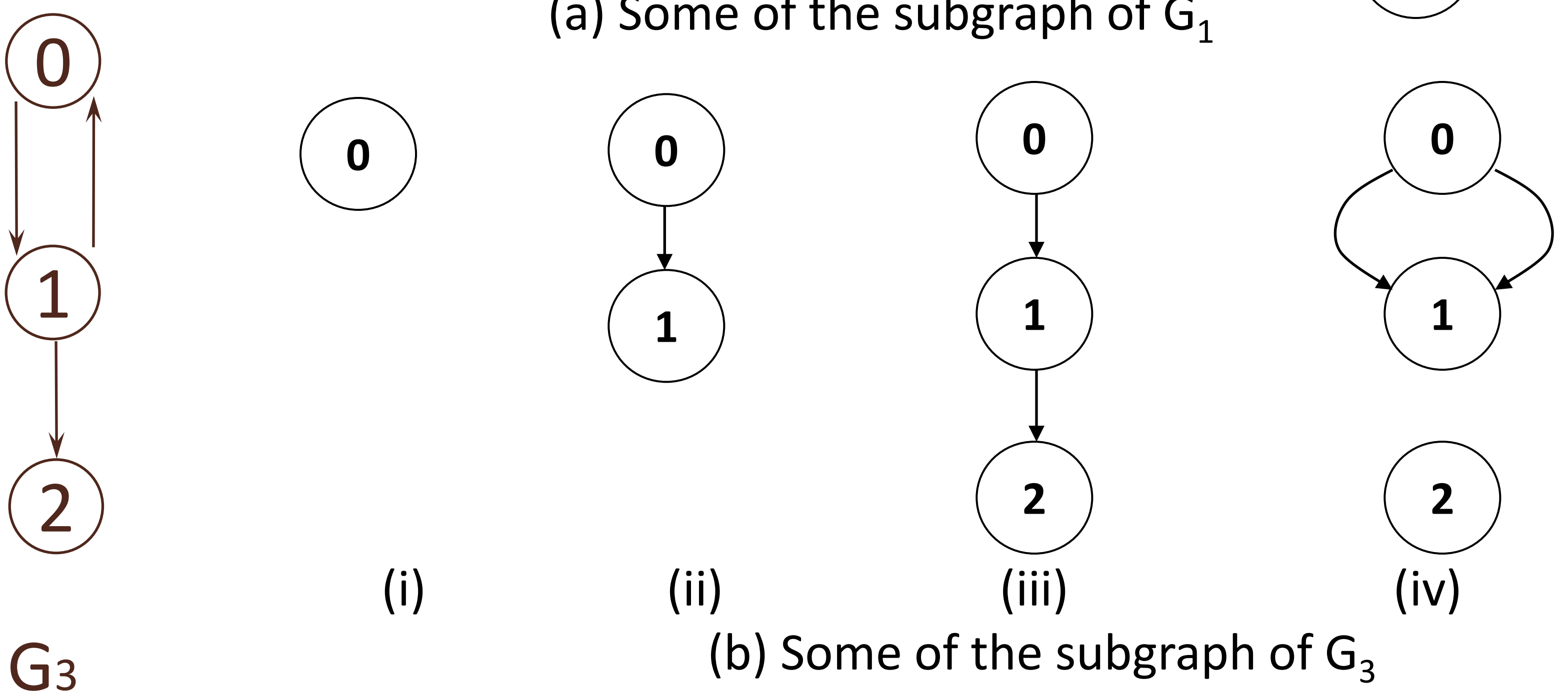


- subgraph: subset of vertices and edges forming a graph

Subgraphs Examples



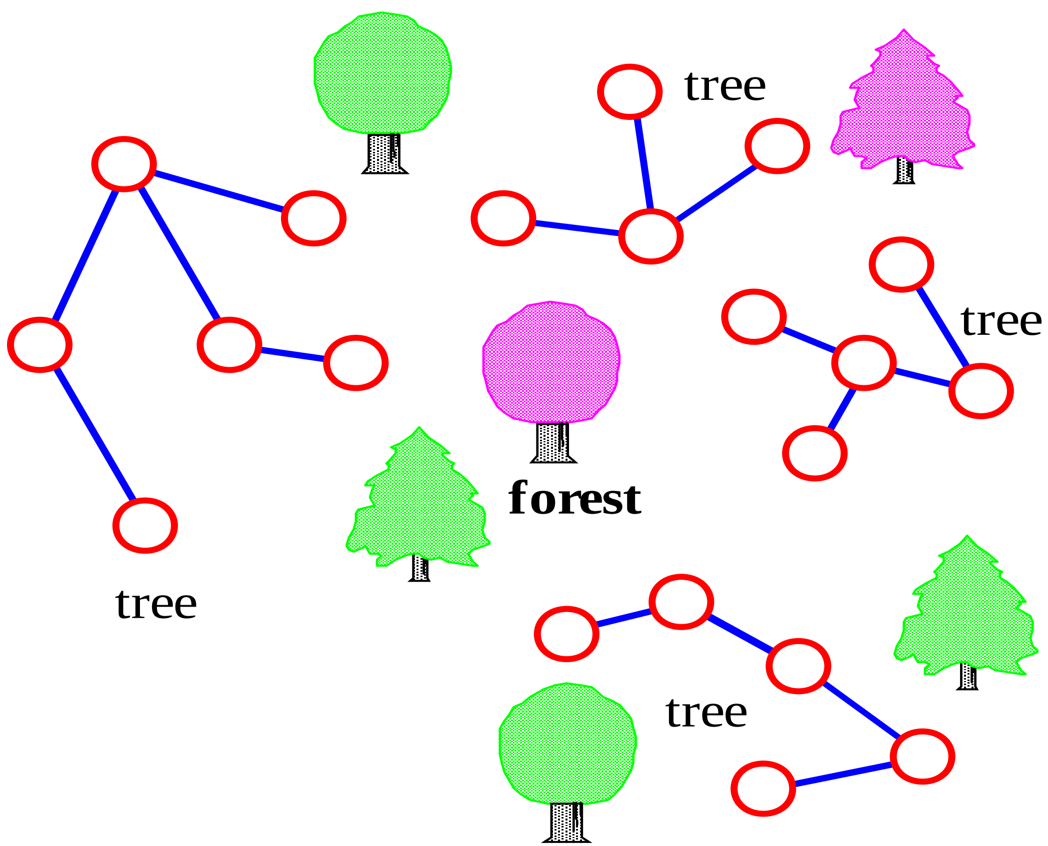
(a) Some of the subgraph of G_1



(b) Some of the subgraph of G_3

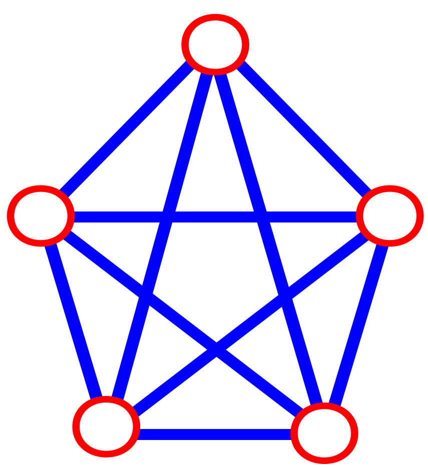
More…

- tree - connected graph without cycles
- forest - collection of trees



Connectivity

- Let **n** = #vertices, and **m** = #edges
- **A complete graph**: one in which all pairs of vertices are adjacent
- *How many total edges in a complete graph?*
  - Each of the n vertices is incident to **n-1** edges, however, we would have counted each edge twice! Therefore, intuitively, m = **n**(**n** -1)/2.
- Therefore, if a graph is not complete, m < **n**(**n** -1)/2



$\mathbf{n} = 5$

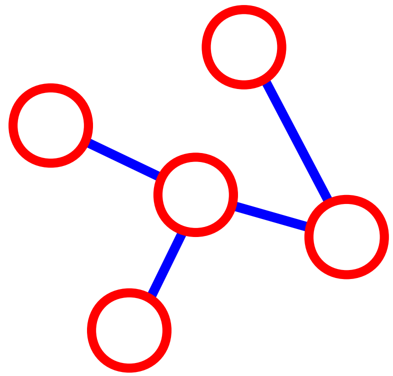
$\mathbf{m} = (5 * 4)/2 = 10$

More Connectivity

n = #vertices

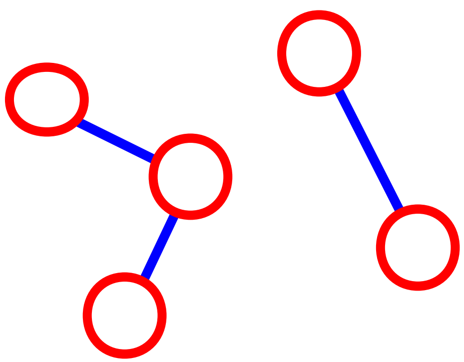
m = #edges

- For a tree **m** = **n** - 1



n = 5
m = 4

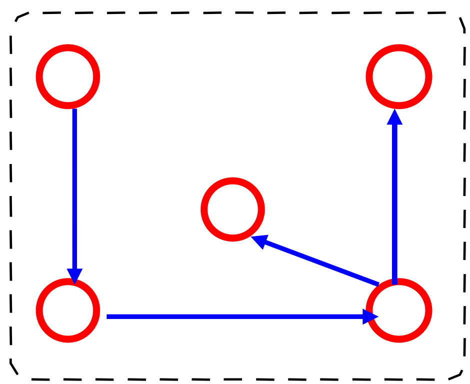
If **m** < **n** - 1, G is not connected



n = 5
m = 3

Oriented (Directed) Graph

- A graph where edges are directed



Directed vs. Undirected Graph

- An undirected graph is one in which the pair of vertices in a edge is unordered, $(v_0, v_1) = (v_1,v_0)$
- A directed graph is one in which each edge is a directed pair of vertices, $<v_0, v_1>$!= $<v_1,v_0>$

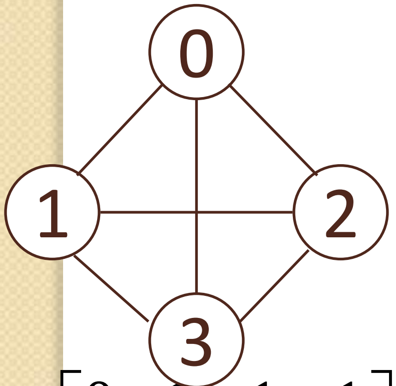
Graph Representations

- Adjacency Matrix
- Adjacency Lists

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 (v_i, v_j) 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

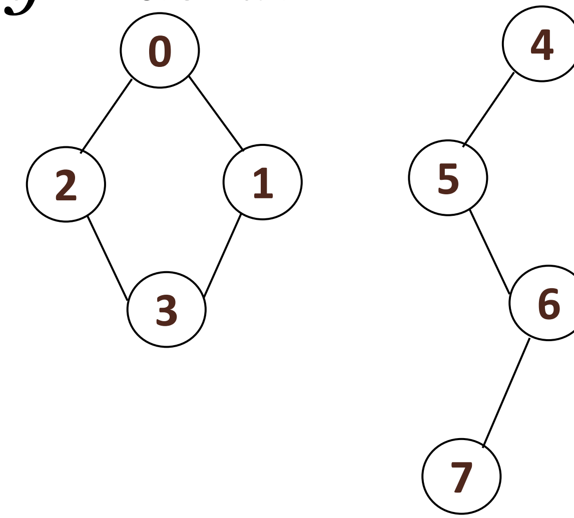


$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

G_1

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

G_2



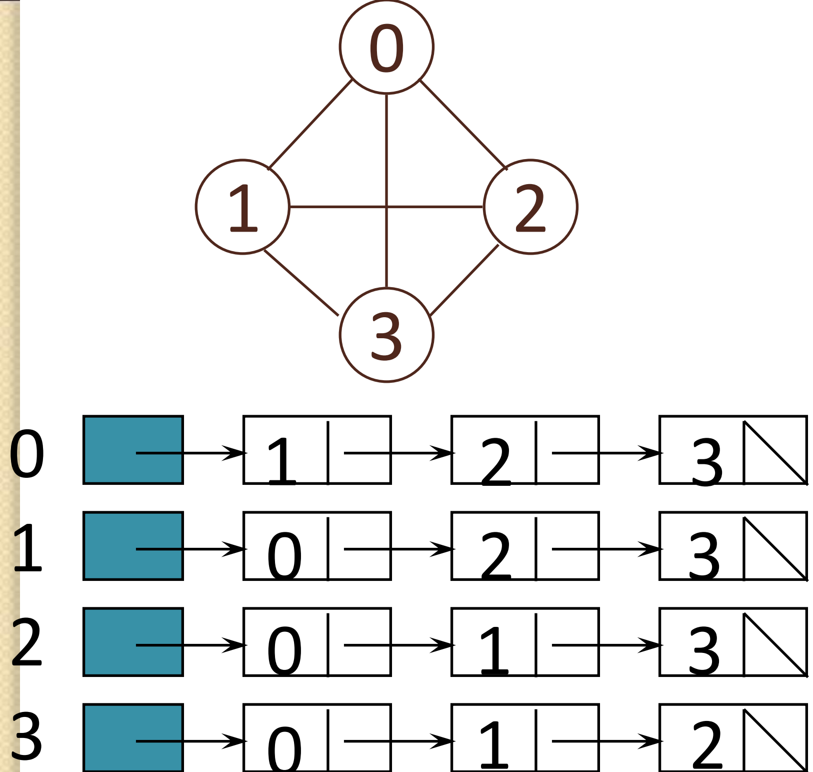
$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

G_4

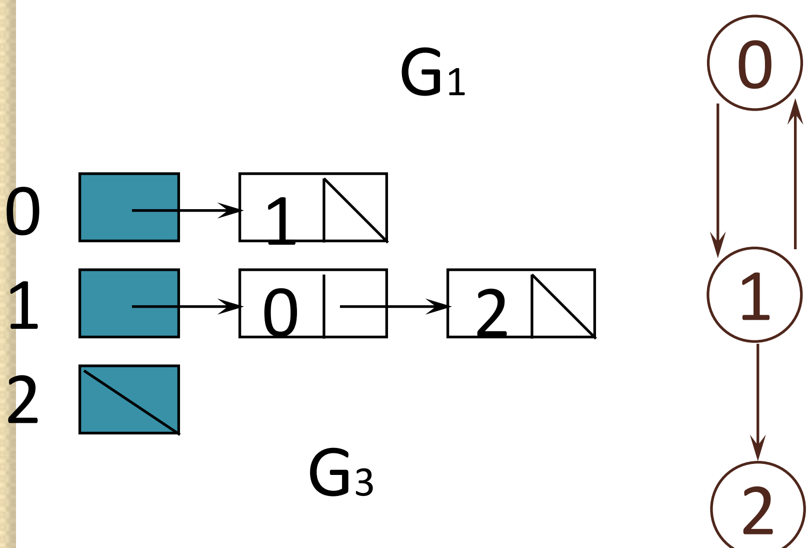
Adjacency Lists (data structures)

Each row in adjacency matrix is represented as an adjacency list.

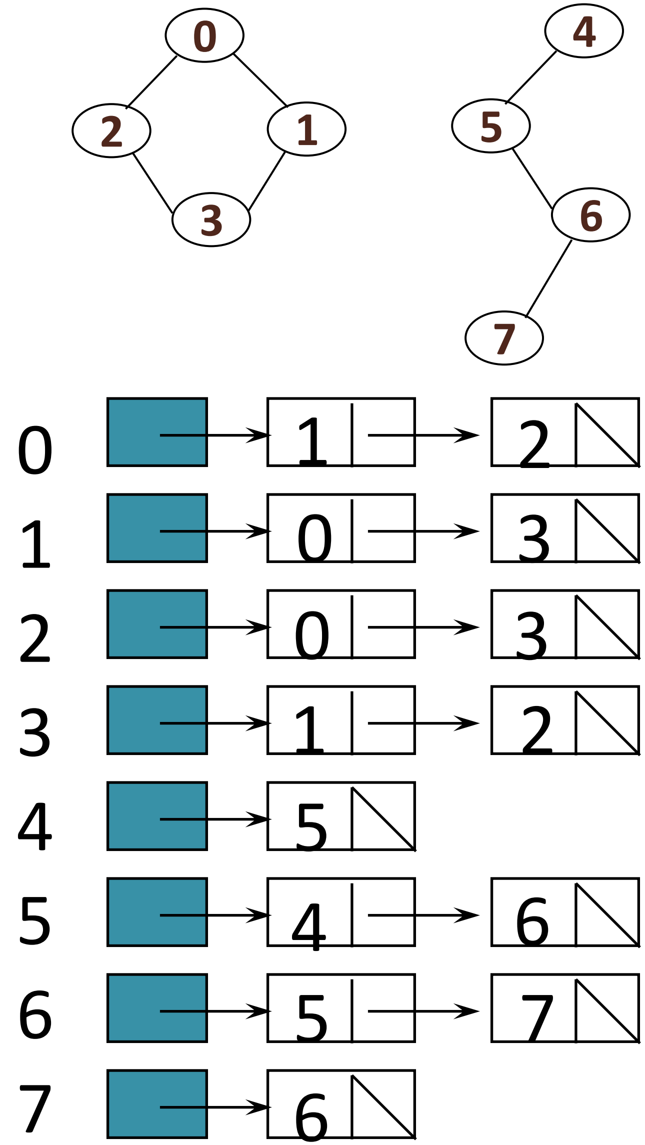
```
Const int MAX_VERTICES= 50
typedef struct node *node_pointer;
typedef struct node {
    int vertex;
    node *link;
}*node_pointer;
;
node_pointer graph[MAX_VERTICES];
int n=0; /* vertices currently in use */
```



G1



G3

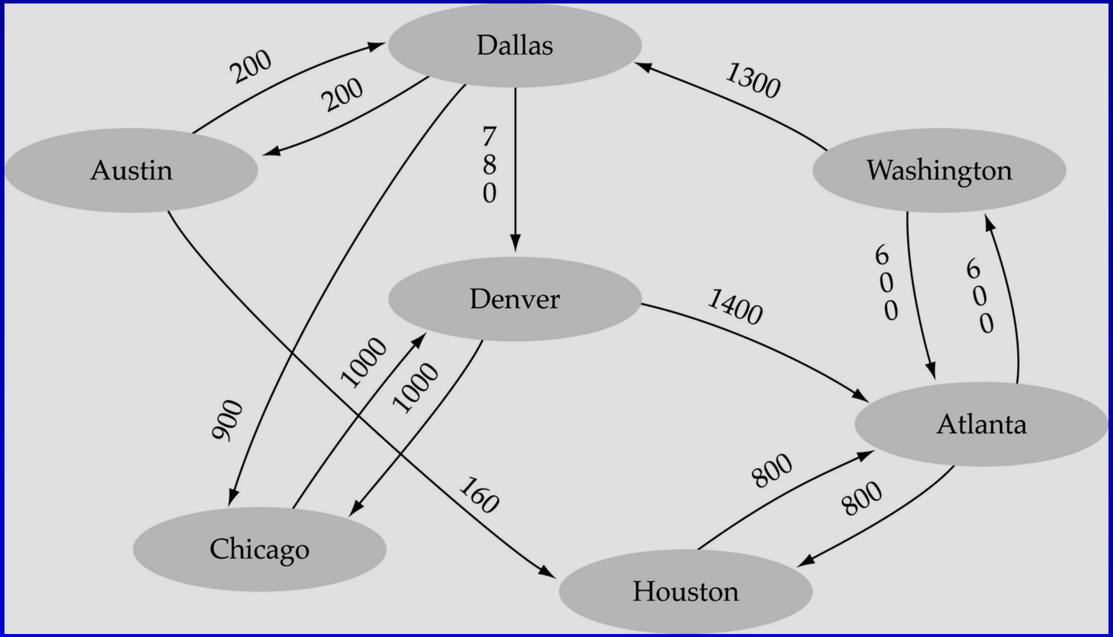


G4

An undirected graph with n vertices and e edges ==> n head nodes and 2e list nodes

Array-based implementation

- Use a 1D array to represent the vertices
- Use a 2D array (i.e., adjacency matrix) to represent the edges



Array-based implementation (cont'd)

graph

.numVertices 7

.vertices

[0]	"Atlanta "
[1]	"Austin "
[2]	"Chicago "
[3]	"Dallas "
[4]	"Denver "
[5]	"Houston "
[6]	"Washington"
[7]	
[8]	
[9]	

.edges

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
[0]	0	0	0	0	0	800	600	•	•	•
[1]	0	0	0	200	0	160	0	•	•	•
[2]	0	0	0	0	1000	0	0	•	•	•
[3]	0	200	900	0	780	0	0	•	•	•
[4]	1400	0	1000	0	0	0	0	•	•	•
[5]	800	0	0	0	0	0	0	•	•	•
[6]	600	0	0	1300	0	0	0	•	•	•
[7]	•	•	•	•	•	•	•	•	•	•
[8]	•	•	•	•	•	•	•	•	•	•
[9]	•	•	•	•	•	•	•	•	•	•

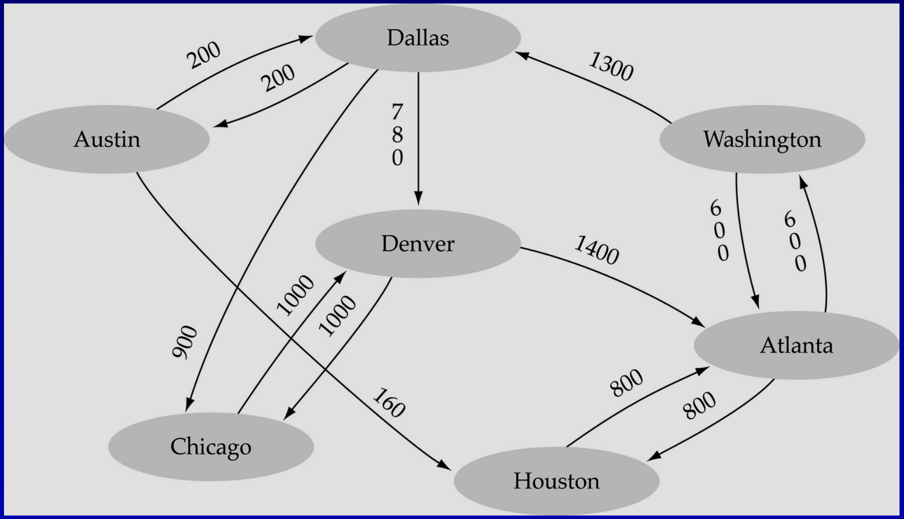
(Array positions marked '•' are undefined)

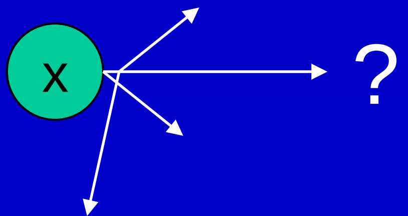
Array-Based Implementation (cont.)

- Memory required
  - $O(V+V^2)=O(V^2)$
- Preferred when
  - The graph is **dense:** $E = O(V^2)$
- Advantage
  - Can quickly determine

  if there is an edge between two vertices
- Disadvantage
  - No quick way to determine the vertices adjacent

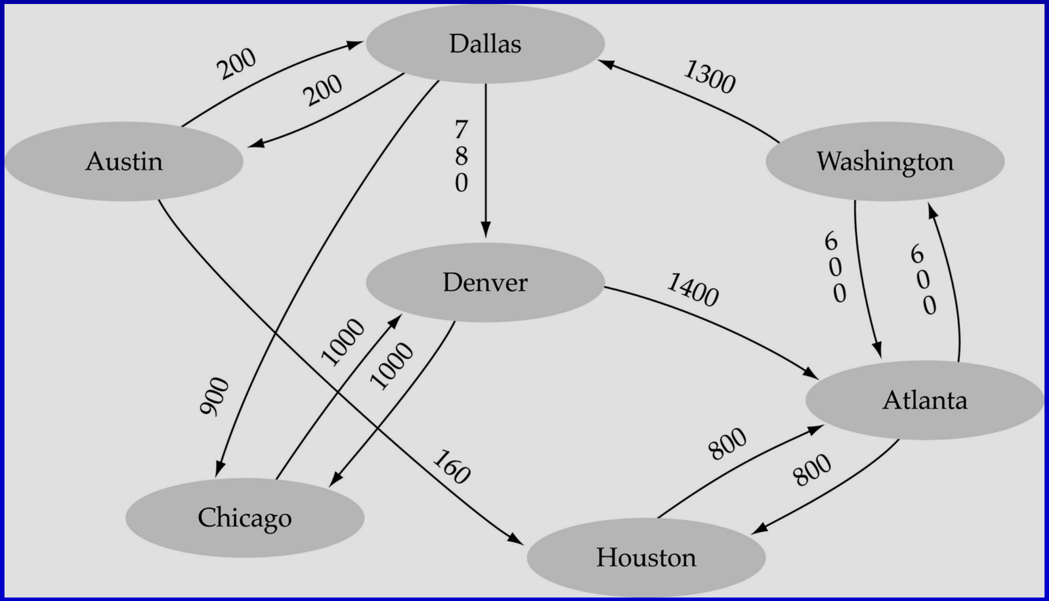
    from another vertex



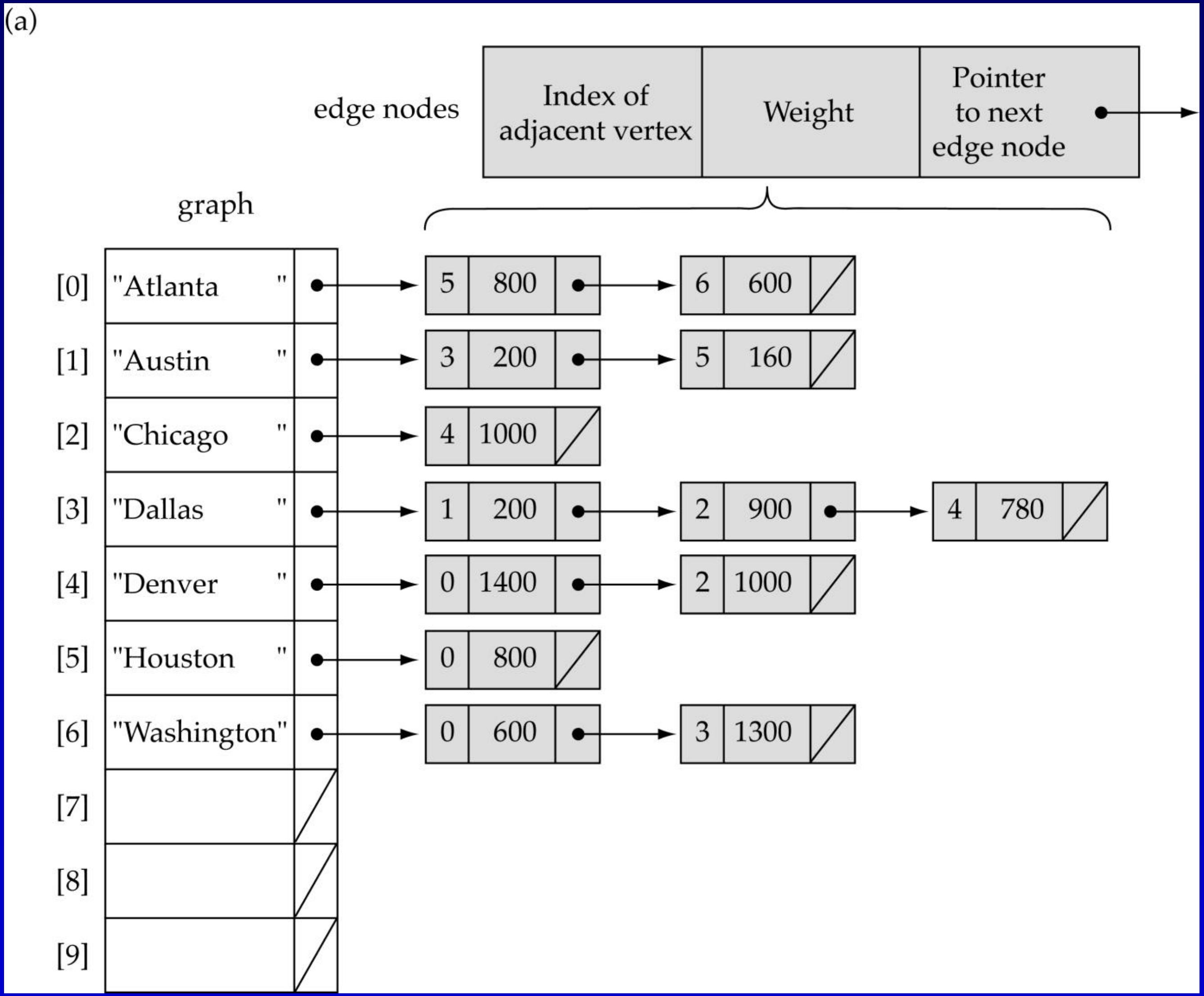


Linked-list-based implementation

- Use a 1D array to represent the vertices
- Use a list for each vertex *v* which contains the vertices which are adjacent **from** *v* (adjacency list)



Linked-list-based implementation (cont'd)



Graph Traversal

- Problem: Search for a certain node or traverse all nodes in the graph
- Depth First Search
- Breadth First Search

Visualizations

https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

Graph Traversals

- Depth-First Traversals.
- Breadth-First Traversal.

Depth-First-Search (DFS)

- Main idea:
  - Travel as far as you can down a path
  - Back up *as little as possible* when you reach a "dead end" (i.e., next vertex has been "marked" or there is no next vertex)
- DFS uses a *stack* !

Depth-First Search

- DFS follows the following rules:

1. Select an unvisited node x, visit it, and treat as the current node
2. Find an unvisited neighbor of the current node, visit it, and make it the new current node;
3. If the current node has no unvisited neighbors, backtrack to the its parent, and make that parent the new current node;
4. Repeat steps 3 and 4 until no more nodes can be visited.
5. If there are still unvisited nodes, repeat from step 1.

Depth-First Traversal Algorithm

- In this method, After visiting a vertex v, which is adjacent to w1, w2, w3, ...; Next we visit one of v's adjacent vertices, w1 say. Next, we visit all vertices adjacent to w1 before coming back to w2, etc.

- Must keep track of vertices already visited to avoid cycles.

- The method can be implemented using recursion or iteration.

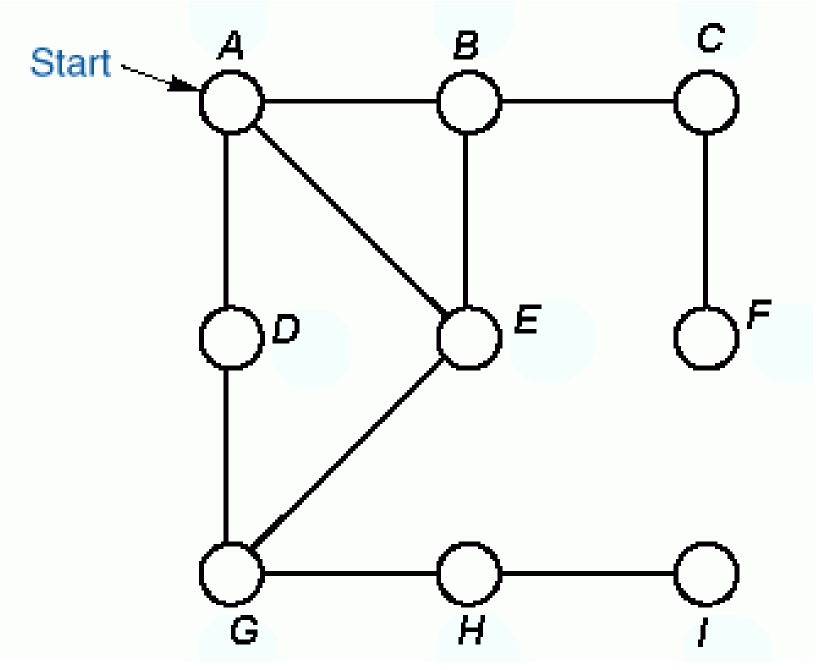
- The iterative preorder depth-first algorithm is:

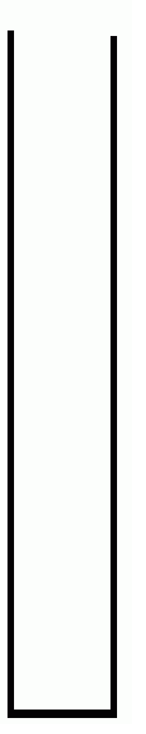
```
push the starting vertex onto the stack
while(stack is not empty){
    pop a vertex off the stack, call it v
    if v is not already visited, visit it
     push vertices adjacent to v, not visited, onto the stack
 }
```

- Note: Adjacent vertices can be pushed in any order; but to obtain a unique traversal, we will push them in reverse alphabetical order.

Example

- Demonstrates depth-first traversal using an explicit stack.





Stack

Order of Traversal

1	2	3	4	5	6	7	8	9
A	B	C	F	E	G	D	H	I

Recursive preorder Depth-First Traversal Implementation

```
dfsPreorder(v){
   visit v;
   for(each neighbour w of v)
      if(w has not been visited)
         dfsPreorder(w);
}
```

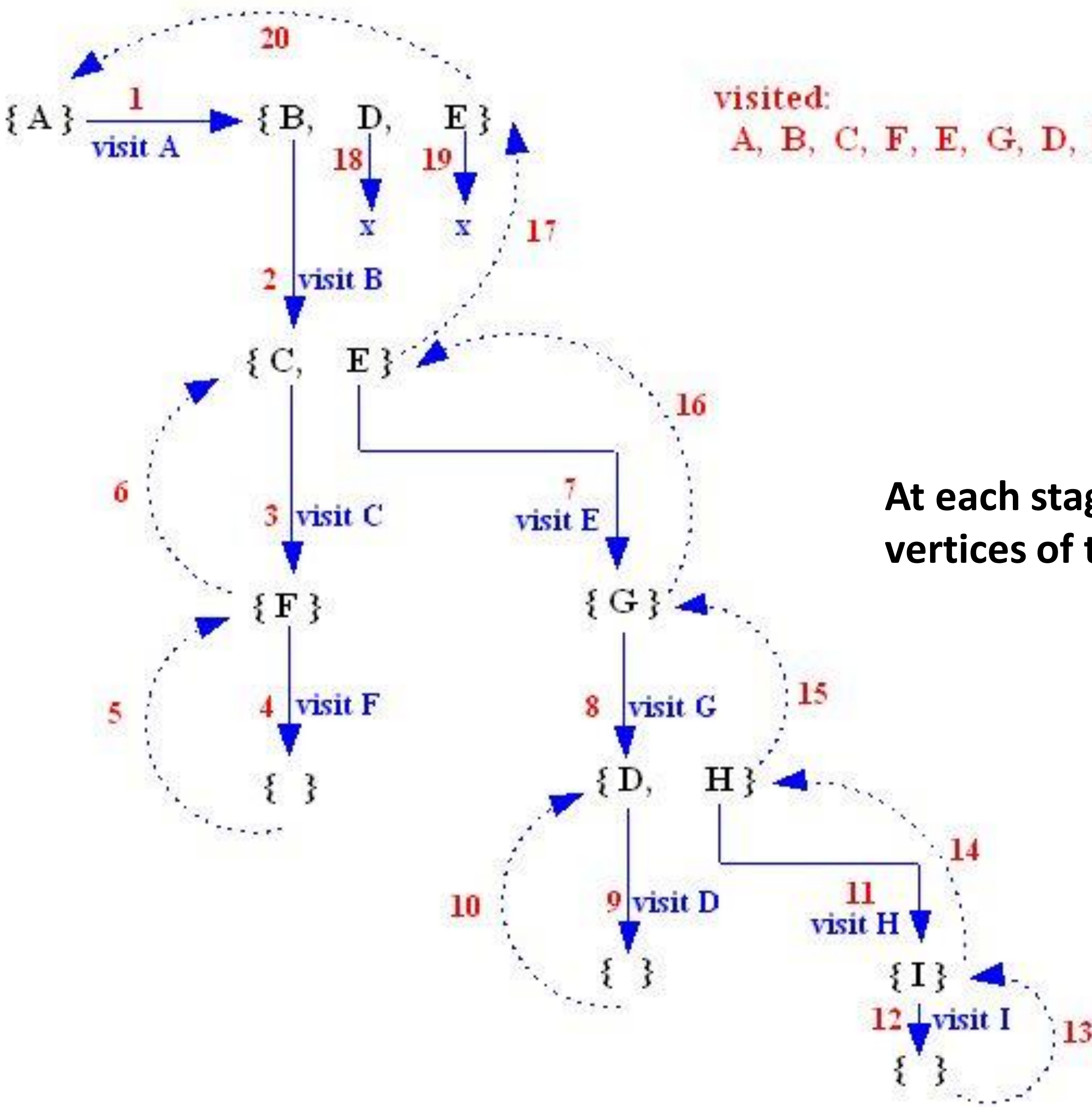
- The following is the code for the recursive preorderDepthFirstTraversal method of the AbstractGraph class:

```
public void preorderDepthFirstTraversal(Visitor visitor, Vertex start)
{
   boolean visited[] = new boolean[numberOfVertices];
   for(int v = 0; v < numberOfVertices; v++)
      visited[v] = false;
      preorderDepthFirstTraversal(visitor, start, visited);
}
```

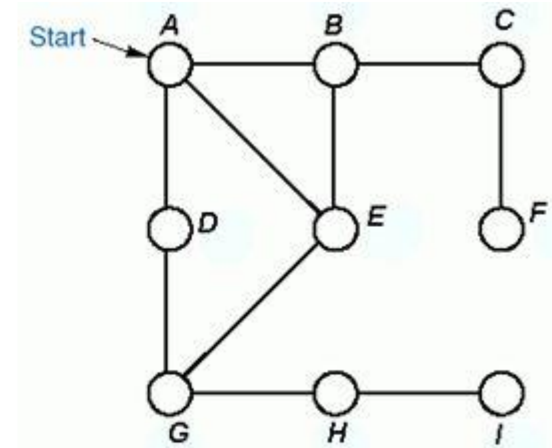
Recursive preorder Depth-First Traversal Implementation (cont'd)

```
private void preorderDepthFirstTraversal(Visitor visitor,
                              Vertex v, boolean[] visited)
{
   if(visitor.isDone())
      return;
   visitor.visit(v);
   visited[getIndex(v)] = true;

   Iterator p = v.getSuccessors();
   while(p.hasNext())    {
      Vertex to = (Vertex) p.next();
      if(! visited[getIndex(to)])
         preorderDepthFirstTraversal(visitor, to, visited);
   }
}
```

Recursive preorder Depth-First Traversal Implementation (cont'd)



visited:
A, B, C, F, E, G, D, H, I



At each stage, a set of unvisited adjacent vertices of the current vertex is generated.

Recursive postorder Depth-First Traversal Implementation

```
dfsPostorder(v){
   mark v;
   for(each neighbour w of v)
      if(w is not marked)
         dfsPostorder(w);

   visit v;
}
```

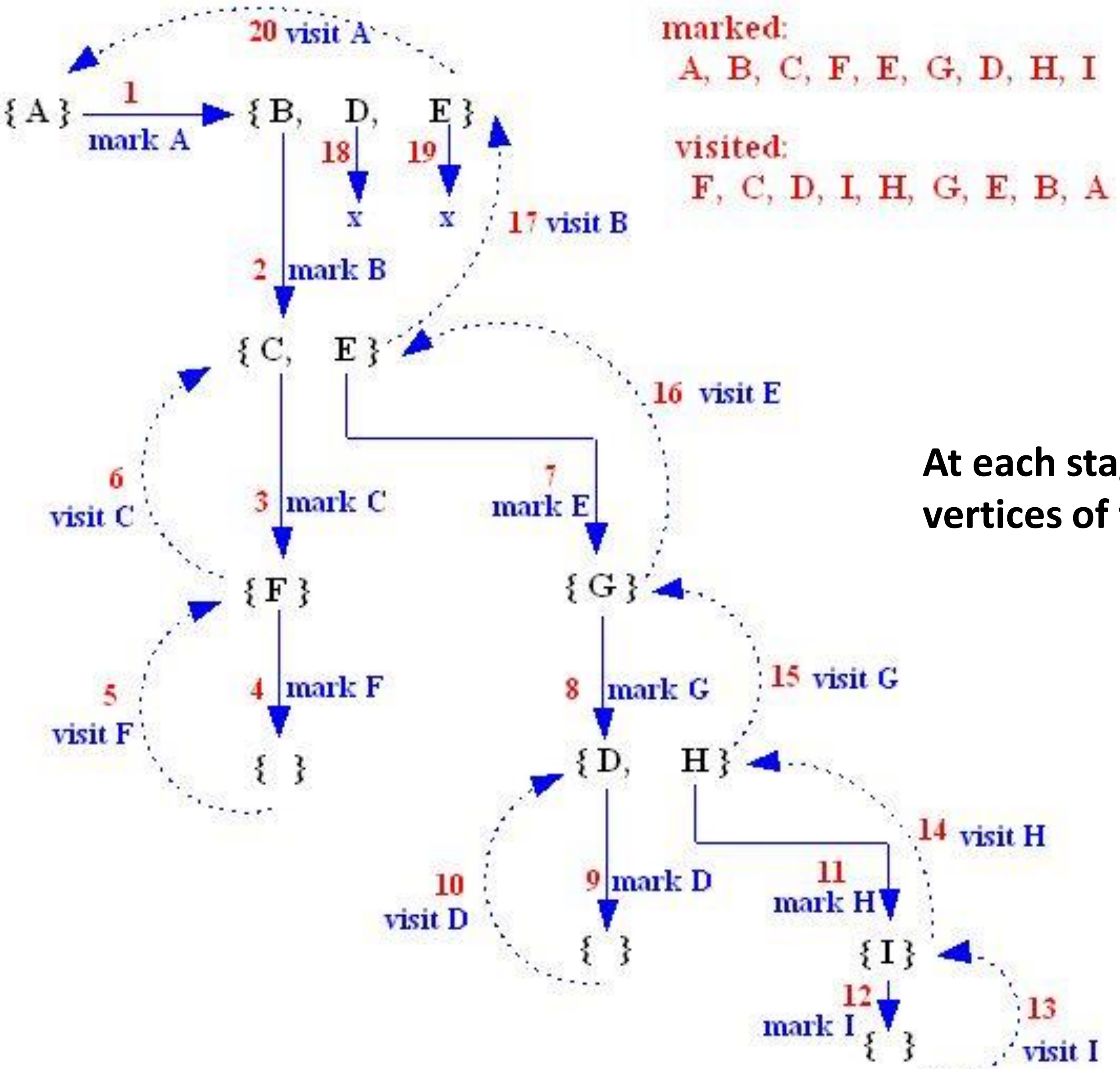
•The following is the code for the recursive postorderDepthFirstTraversal method of the AbstractGraph class:

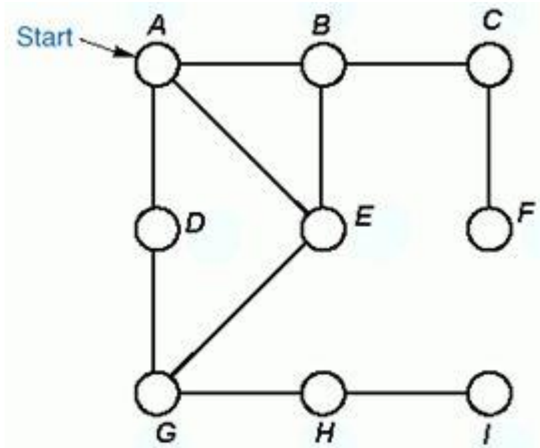
```java
public void postorderDepthFirstTraversal(Visitor visitor,
                                          Vertex start)
{
   boolean visited[] = new boolean[numberOfVertices];
   for(int v = 0; v < numberOfVertices; v++)
      visited[v] = false;

   postorderDepthFirstTraversal(visitor, start, visited);
}
```

Recursive postorder Depth-First Traversal Implementation (cont'd)

```
private void postorderDepthFirstTraversal(
            Visitor visitor, Vertex v, boolean[] visited)
{
   if(visitor.isDone())
      return;

   // mark v
   visited[getIndex(v)] = true;

   Iterator p = v.getSuccessors();
   while(p.hasNext()){
      Vertex to = (Vertex) p.next();
      if(! visited[getIndex(to)])
         postorderDepthFirstTraversal(visitor, to, visited);
   }

   // visit v
   visitor.visit(v);
}
```

Recursive postorder Depth-First Traversal Implementation (cont'd)





At each stage, a set of unmarked adjacent vertices of the current vertex is generated.

Breadth-First-Searching (BFS)

- Main idea:
  - Look at all possible paths at the same depth before you go at a deeper level
  - Back up *as far as possible* when you reach a "dead end" (i.e., next vertex has been "marked" or there is no next vertex)
- BFS uses a *queue* !

Breadth-First Search

- BFS follows the following rules:

1. Select an unvisited node x, visit it, have it be the root in a BFS tree being formed. Its level is called the current level.
2. From each node z in the current level, in the order in which the level nodes were visited, visit all the unvisited neighbors of z. The newly visited nodes from this level form a new level that becomes the next current level.
3. Repeat step 2 until no more nodes can be visited.
4. If there are still unvisited nodes, repeat from Step 1.

Breadth-First Traversal Algorithm

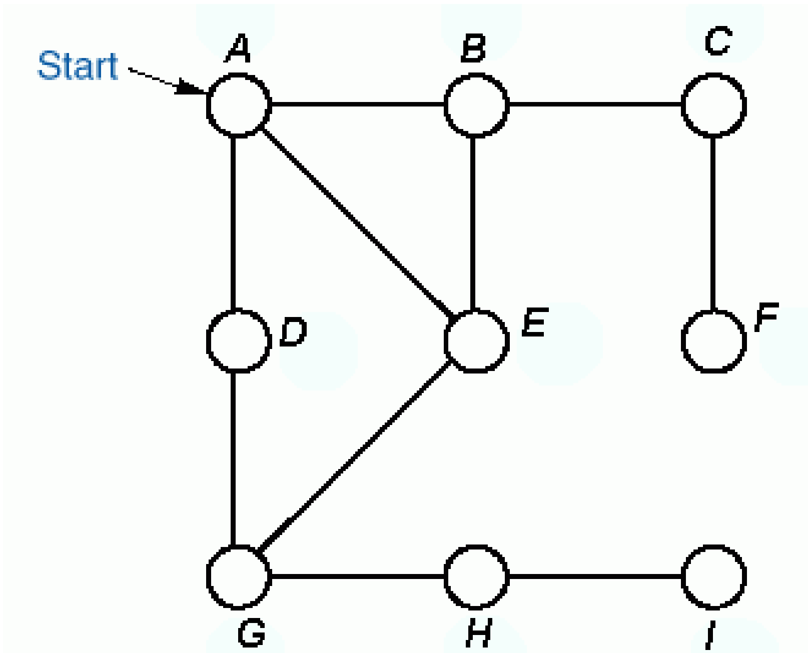
- In this method, After visiting a vertex v, we must visit all its adjacent vertices w1, w2, w3, ..., before going down next level to visit vertices adjacent to w1 etc.
- The method can be implemented using a queue.
- A boolean array is used to ensure that a vertex is enqueued only once.

```
enqueue the starting vertex
while(queue is not empty){
    dequeue a vertex v from the queue;
     visit v.
     enqueue vertices adjacent to v that were never enqueued;
  }
```

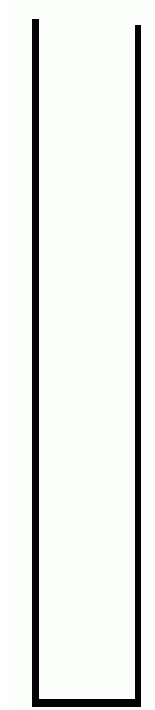
- Note: Adjacent vertices can be enqueued in any order; but to obtain a unique traversal, we will enqueue them in alphabetical order.

Example

- Demonstrating breadth-first traversal using a queue.



Queue front



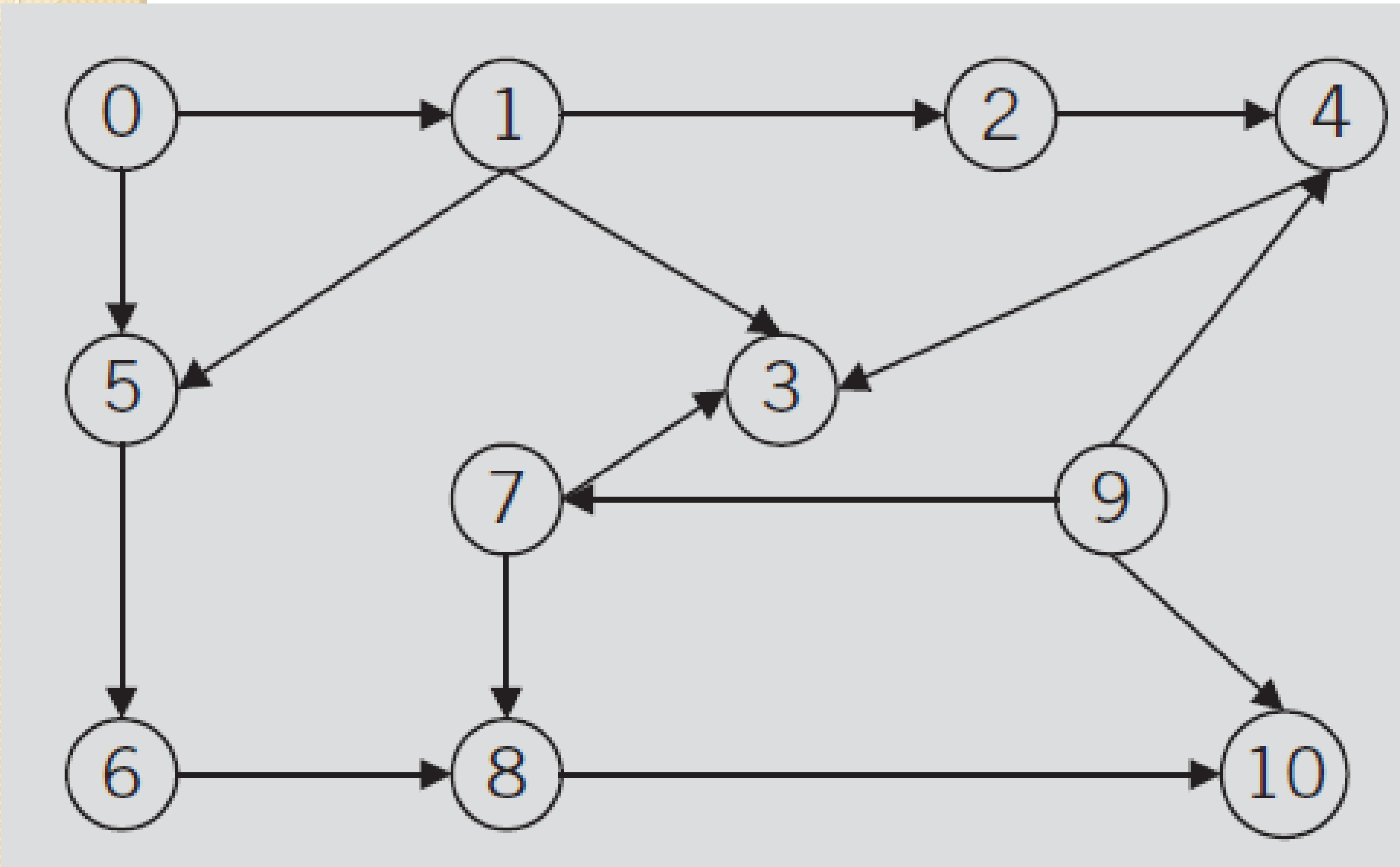
Queue rear

Order of Traversal

1	2	3	4	5	6	7	8	9
A	B	D	E	C	G	F	H	I

Breadth-First Traversal Implementation

```
public void breadthFirstTraversal(Visitor visitor, Vertex start){
   boolean enqueued[] = new boolean[numberOfVertices];
   for(int i = 0; i < numberOfVertices; i++) enqueued[i] = false;

   Queue queue = new QueueAsLinkedList();
   enqueued[getIndex(start)] = true;
   queue.enqueue(start);

   while(!queue.isEmpty() && !visitor.isDone())  {
      Vertex v = (Vertex) queue.dequeue();
      visitor.visit(v);
      Iterator it = v.getSuccessors();
      while(it.hasNext()) {
         Vertex to = (Vertex) it.next();
         int index = getIndex(to);
         if(!enqueued[index]) {
            enqueued[index] = true;
            queue.enqueue(to);
         }
      }
   }
}
```


0
1
2
4
5
3
7
9
6
8
10