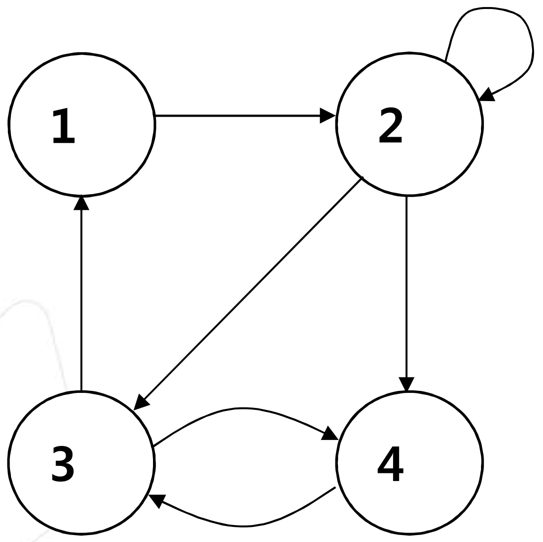
CSE 2017 Data Structures and Lab

Lecture #11: Graph

Eun Man Choi

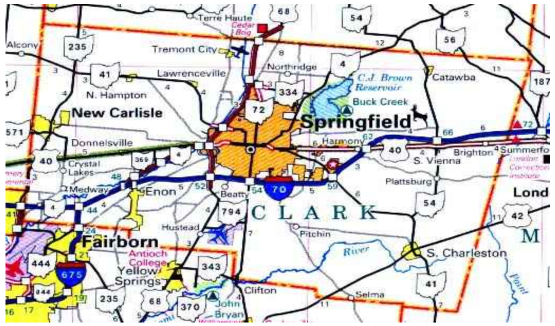
What is a graph?

- A data structure that consists of a set of nodes (*vertices*) and a set of edges between the vertices.
- The set of edges describes relationships among the vertices.



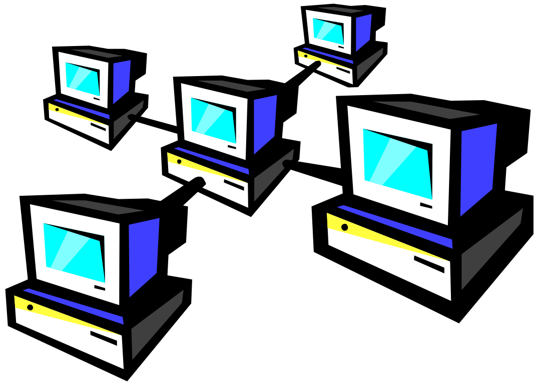
동국대학교 dongguk university

Applications

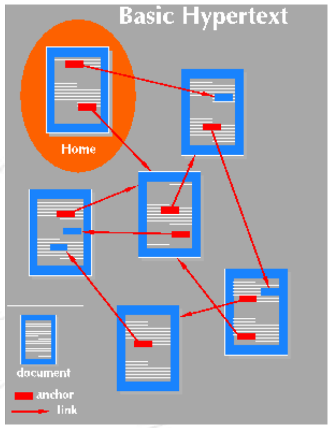


Maps

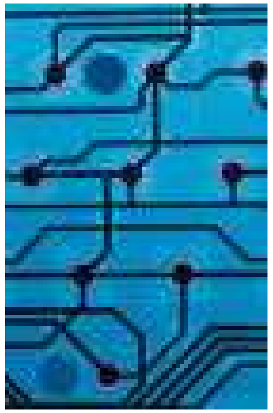
Schedules



Computer networks



Hypertext



Circuits

Formal definition of graphs

- A graph *G* is defined as follows:

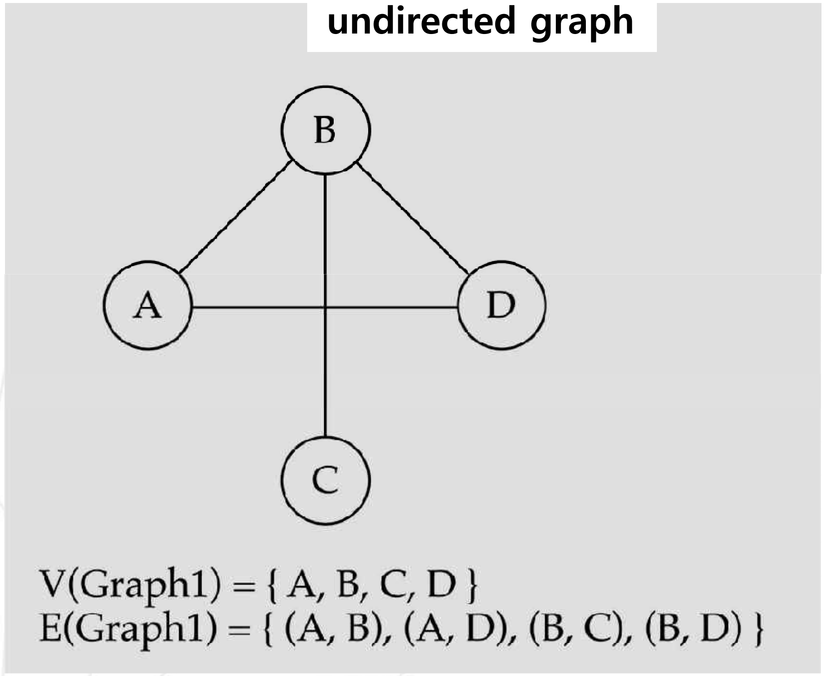
$$G=(V,E)$$

V: a finite, nonempty set of vertices

E: a set of edges (pairs of vertices)

Undirected graphs

- When the edges in a graph have no direction, the graph is called *undirected*



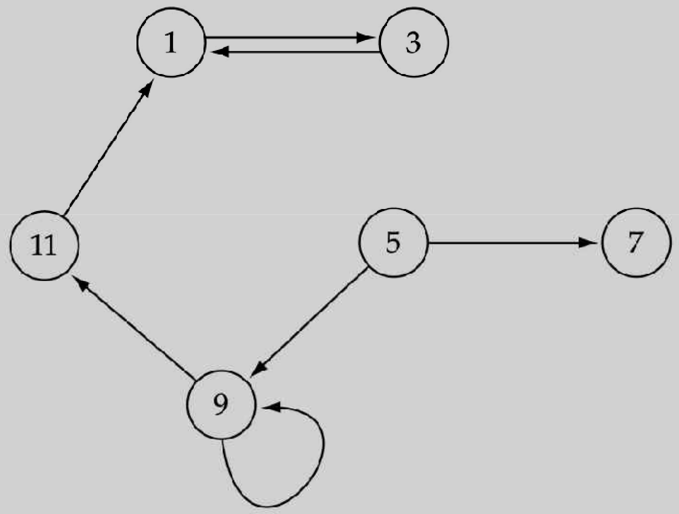
V(Graph1) = { A, B, C, D }
E(Graph1) = { (A, B), (A, D), (B, C), (B, D) }

The order of vertices in E is not important for undirected graphs!!

Directed graphs

- When the edges in a graph have a direction, the graph is called *directed*.

(b) Graph2 is a directed graph.



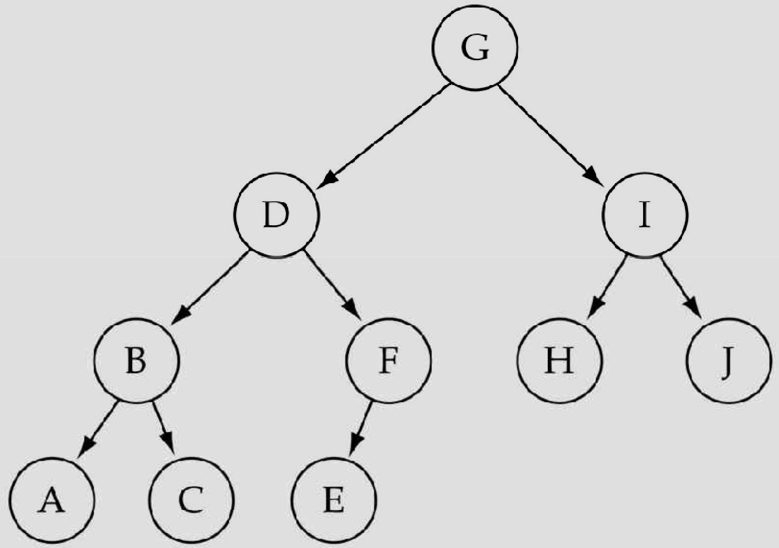
V(Graph2) = { 1, 3, 5, 7, 9, 11 }
E(Graph2) = {(1,3) (3,1) (5,9) (9,11) 1), (9, 9), (11, 1) }
(5,7)

The order of vertices in E is important for directed graphs!!

Trees vs graphs

- **Trees are special cases of graphs!!**

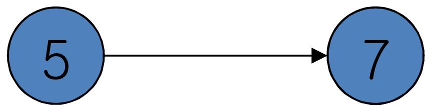
(c) Graph3 is a directed graph.



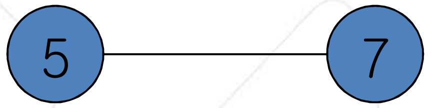
V(Graph3) = { A, B, C, D, E, F, G, H, I, J }
E(Graph3) = { (G, D), (G, I (D, B), (D, F) (I, H), (I, J), (B, A), (B, C), (F, E) }

Graph terminology

- **Adjacent nodes**: two nodes are adjacent if they are connected by an edge



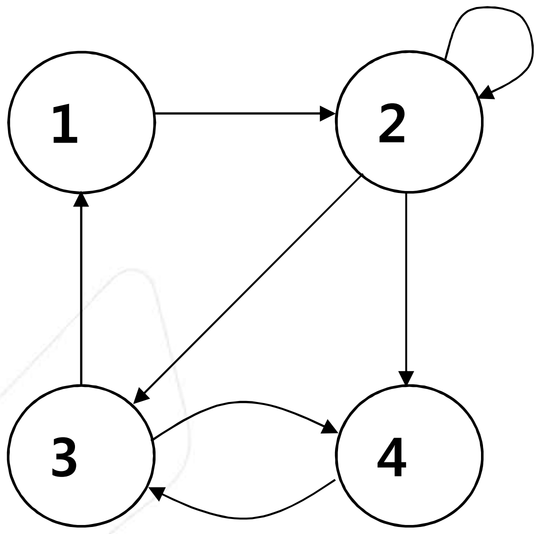
7 is adjacent from 5
or
5 is adjacent to 7



7 is adjacent from/to 5
or
5 is adjacent from/to 7

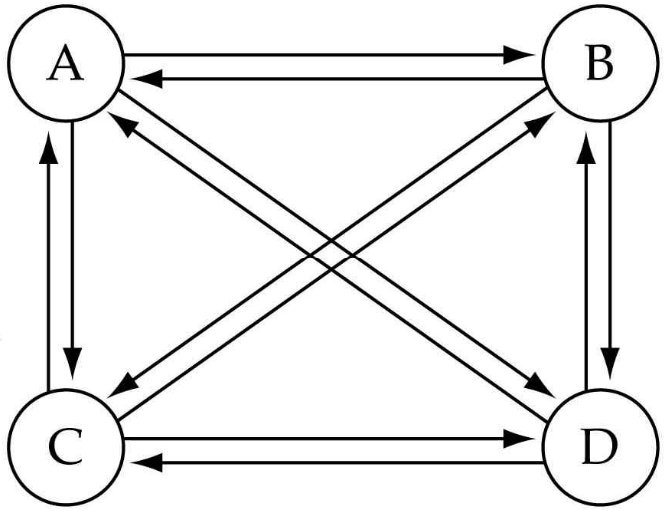
Graph terminology

- Path: a sequence of vertices that connect two nodes in a graph.
- The length of a path is the number of edges in the path.

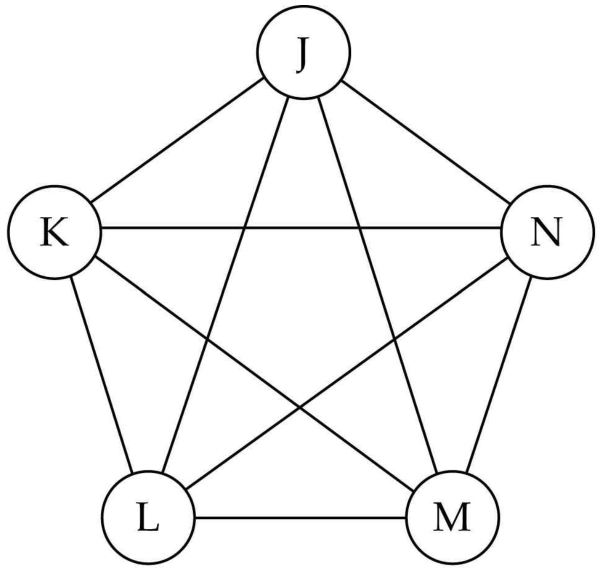


Graph terminology

- **Complete graph**: a graph in which every vertex is directly connected to every other vertex



(a) Complete directed graph.

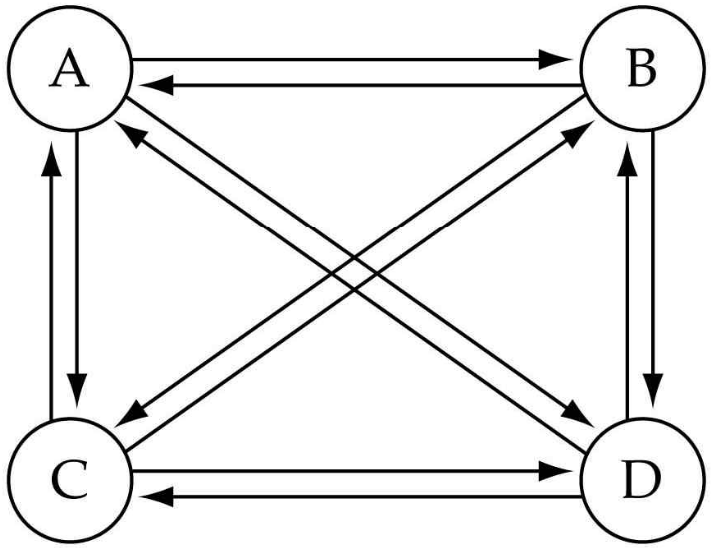


(b) Complete undirected graph.

Graph terminology (cont.)

- What is the number of edges E in a <u>complete directed graph</u> with V vertices?

$$E=V * (V-1)$$

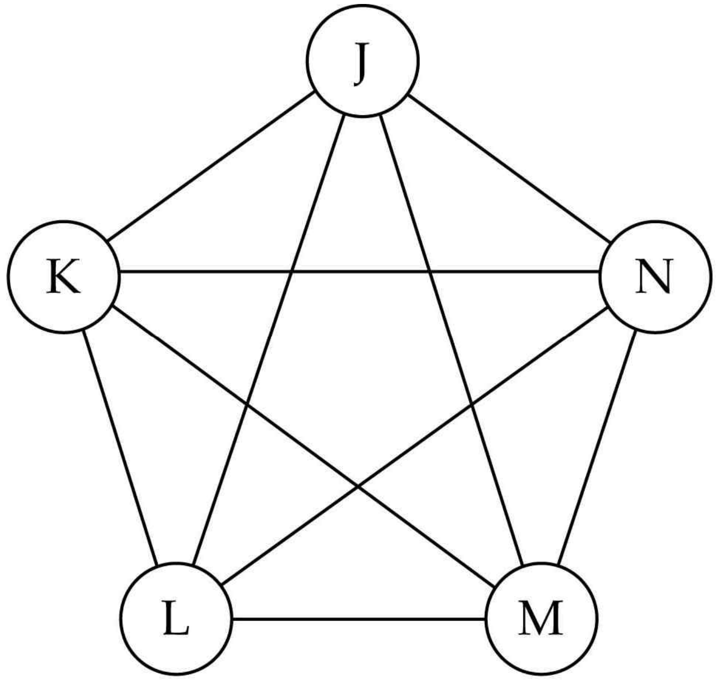


(a) Complete directed graph.

Graph terminology (cont.)

- **What is the number of edges E in a <u>complete undirected graph</u> with V vertices?**

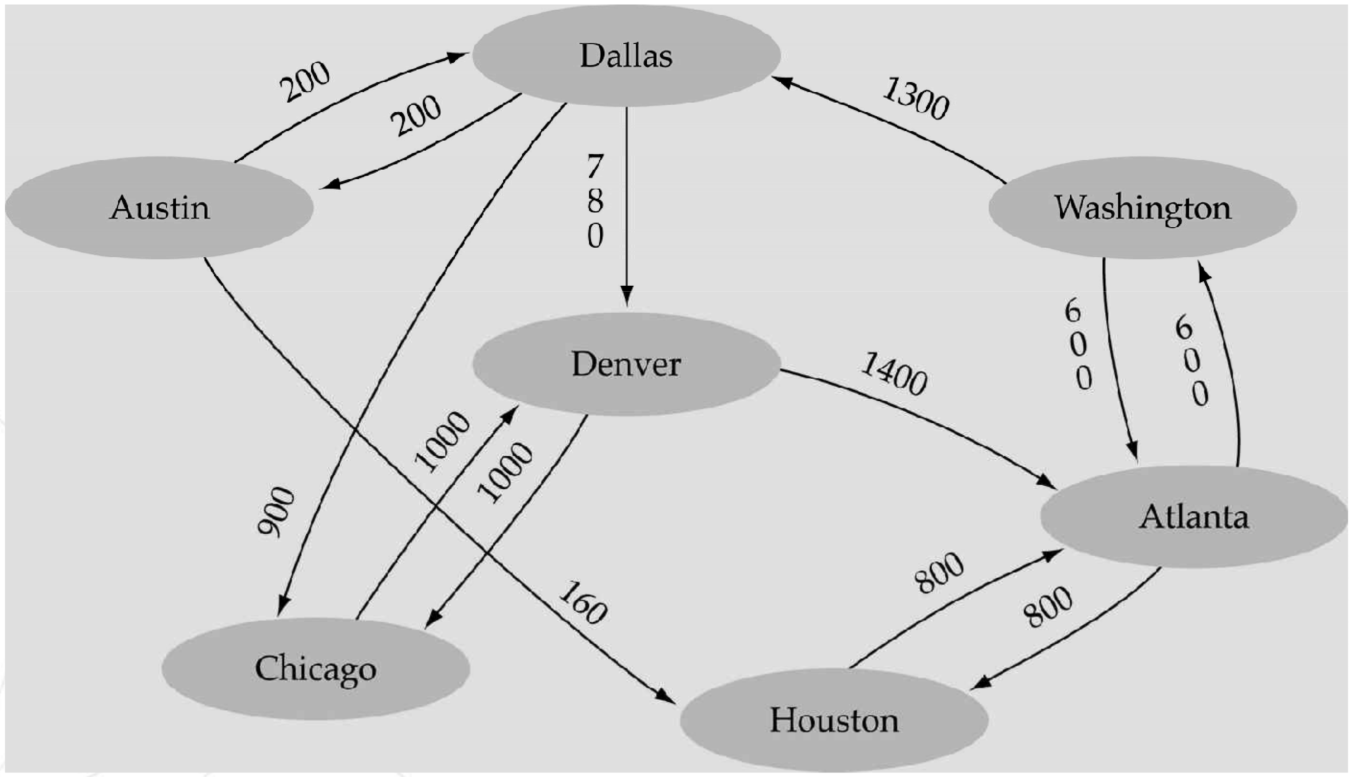
E=V* (V-1) / 2



(b) Complete undirected graph.

Graph terminology (cont.)

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



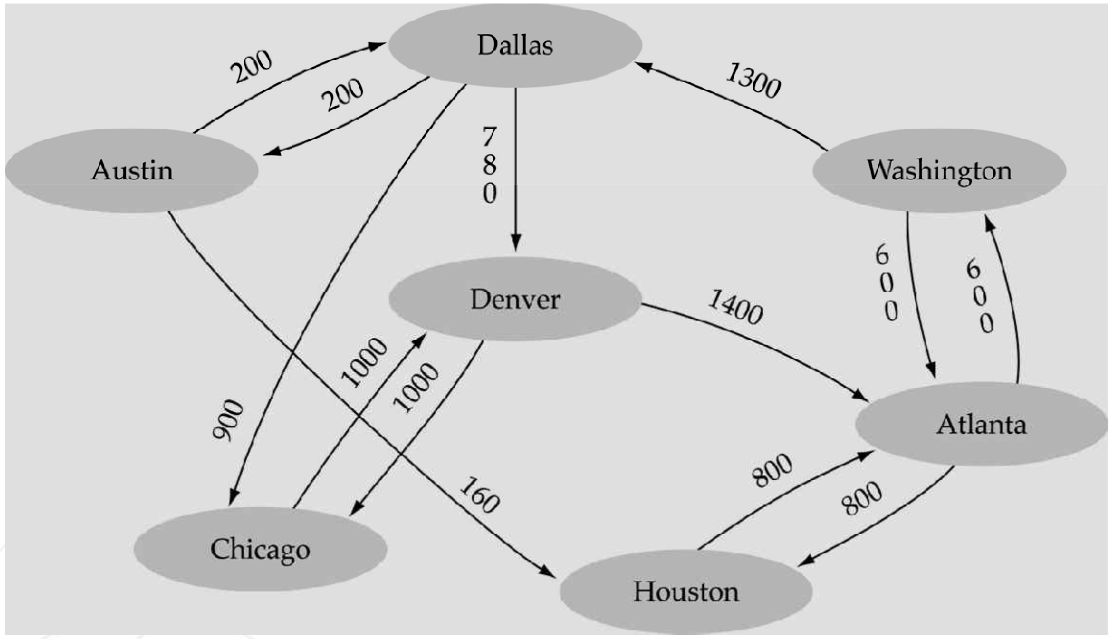
Graph Implementation

- Array-based

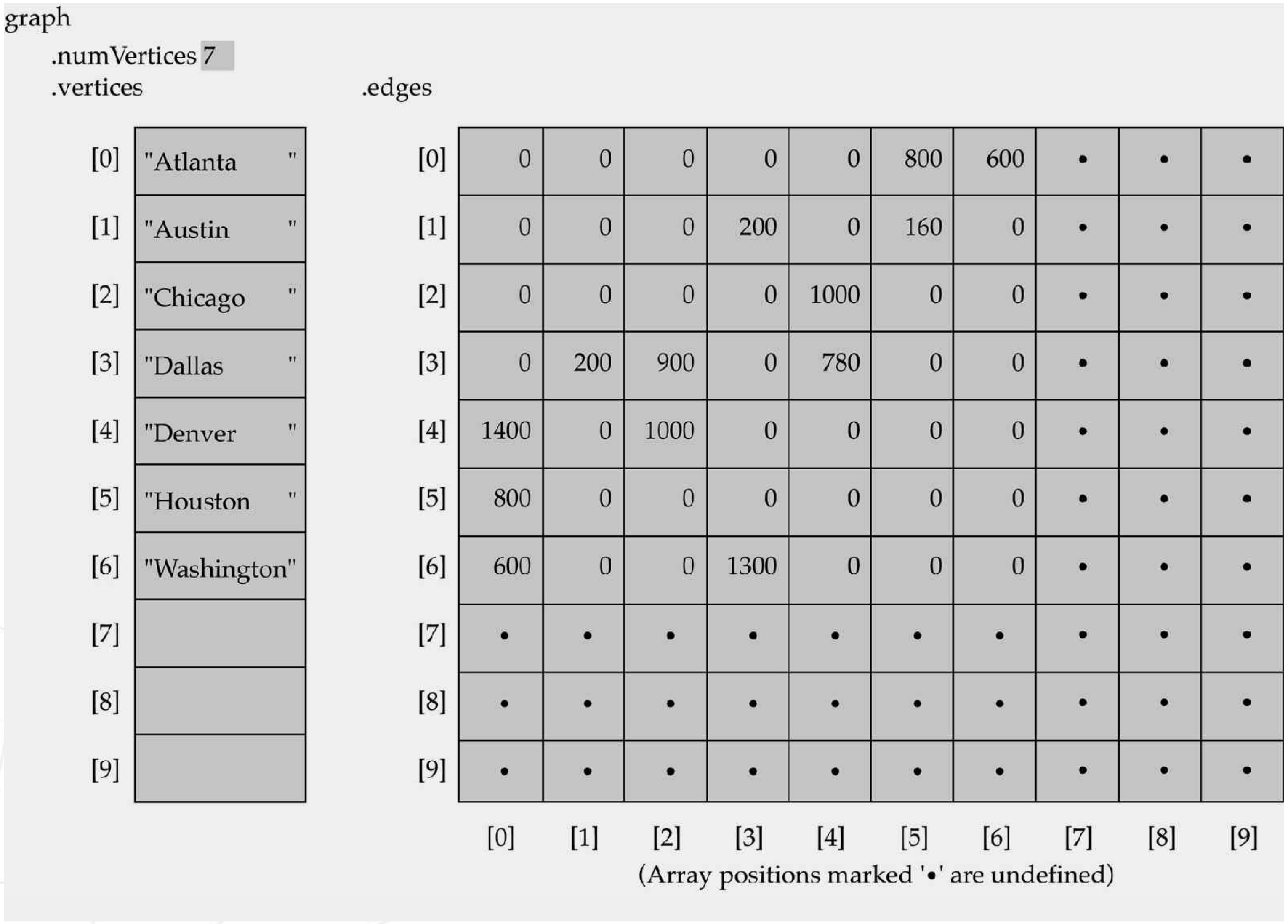
- Linked-list-based

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]	•	•	•	•	•	•	•	•	•	•

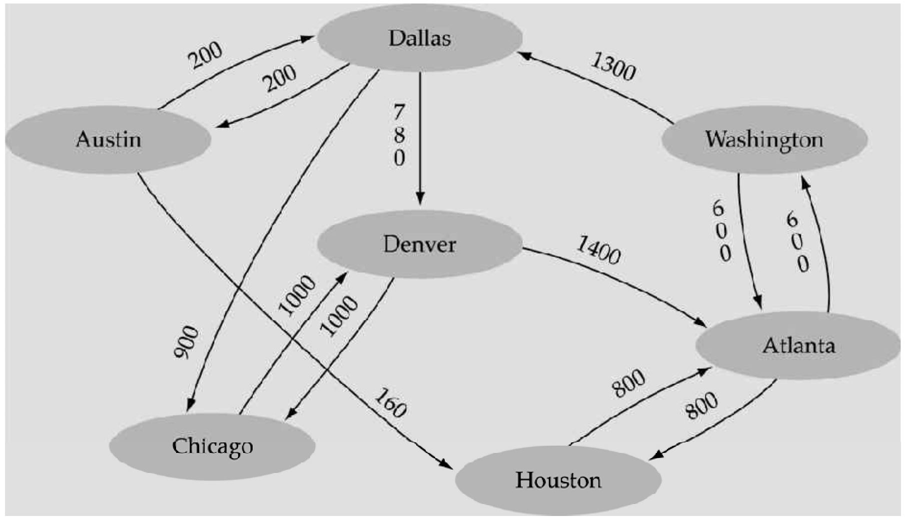
(Array positions marked '•' are undefined)

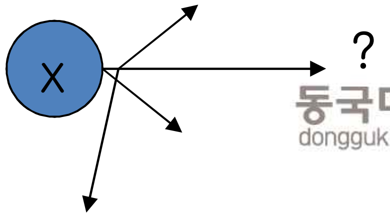
Array-Based Implementation (cont.)

- 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

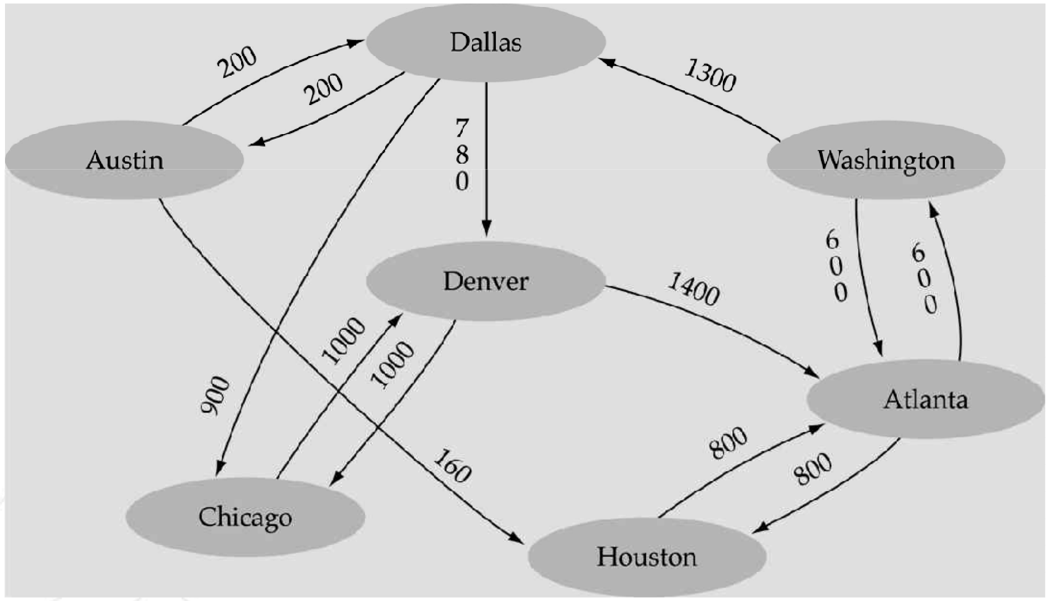
    from another vertex



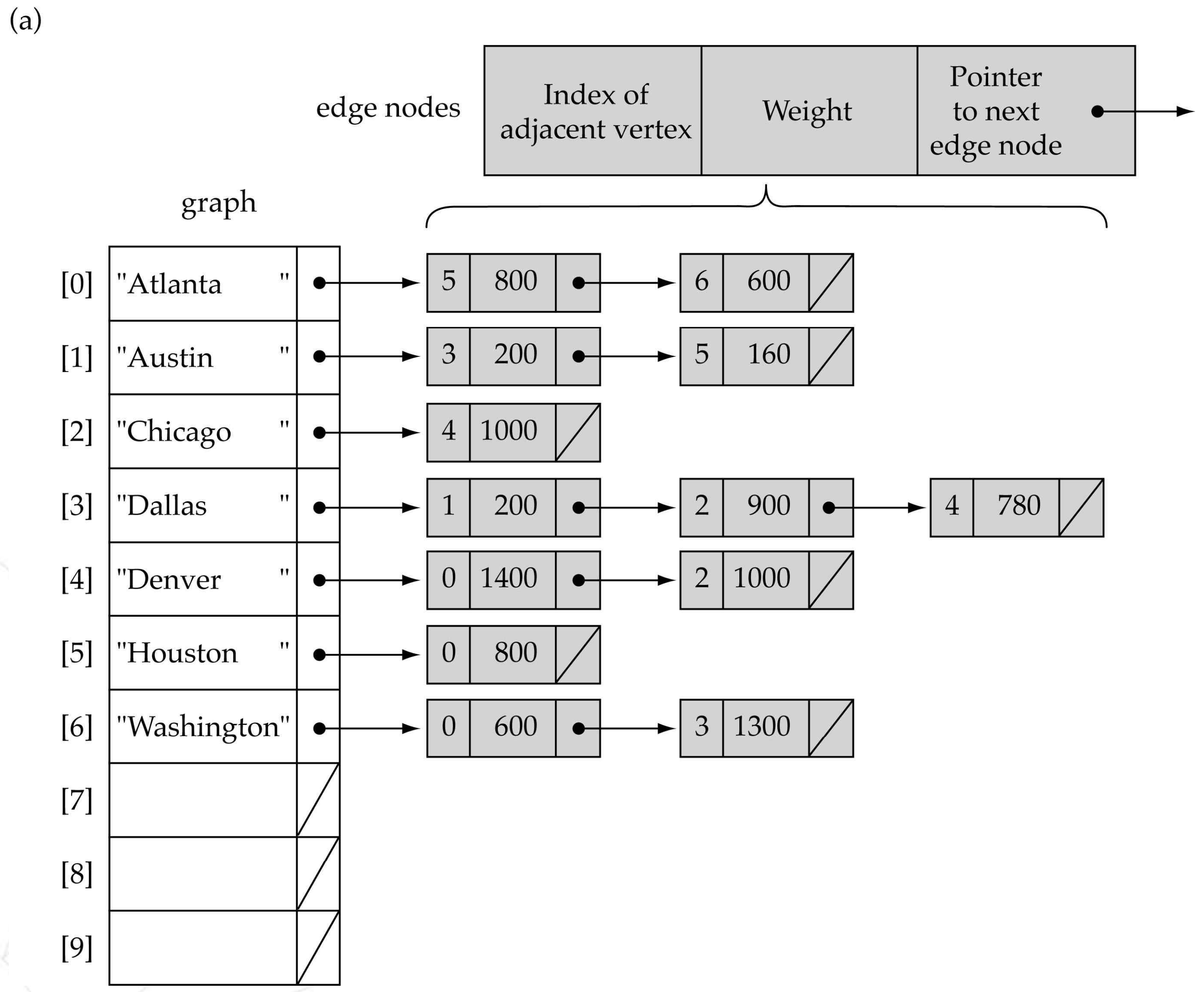


Linked-list-based implementation

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



Linked-list-based implementation (cont'd)



Link-List-based Implementation (cont.)

- Memory required
  - O(V + E)

O(V) for sparse graphs since E=O(V)

- Preferred when

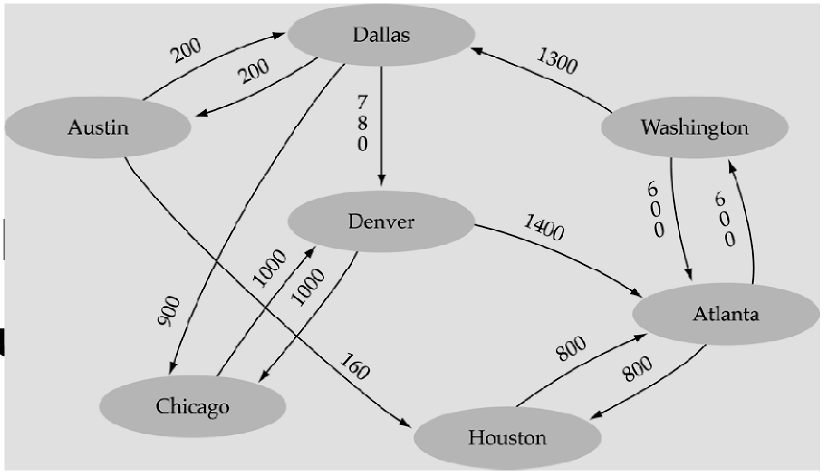
O(V^2) for dense graphs since E=O(V^2)

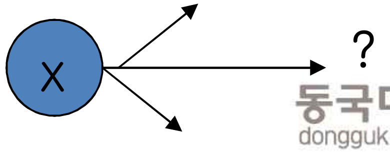
  - for sparse graphs: E = O(V)
- Disadvantage
  - No quick way to determine whet

there is an edge between vertices u

- Advantage
  - Can quickly determine the

vertices adjacent from a given vertex





Graph specification based on adjacency matrix representation

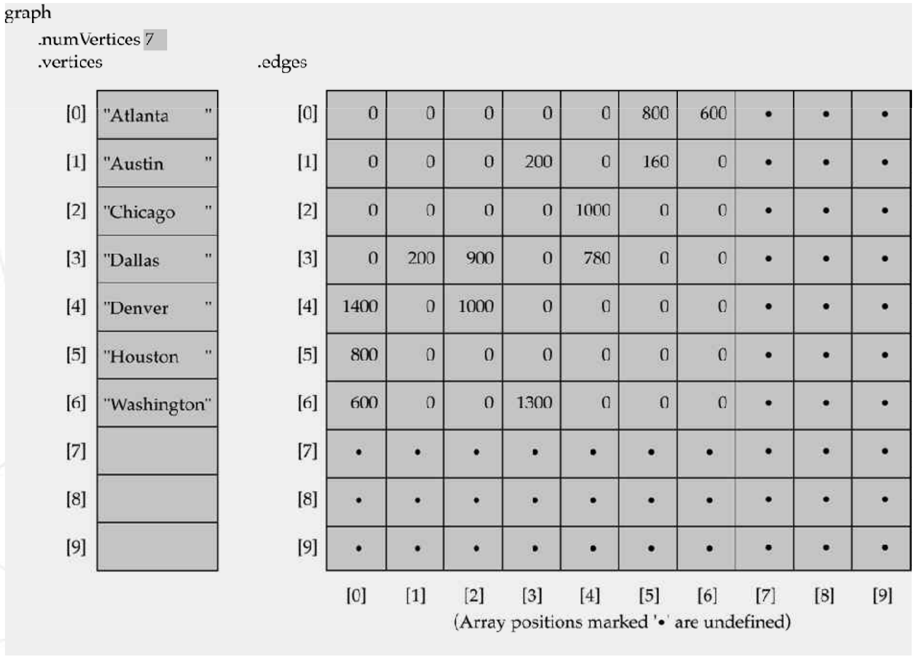
```
const int NULL_EDGE = 0;

template<class VertexType>
class GraphType {
  public:
    GraphType(int);
    ~GraphType();
    void MakeEmpty();
    bool IsEmpty() const;
    bool IsFull() const;
    void AddVertex(VertexType);
    void AddEdge(VertexType, VertexType, int);
    int WeightIs(VertexType, VertexType);
    void GetToVertices(VertexType, QueType<VertexType>&);
    void ClearMarks();
    void MarkVertex(VertexType);
    bool IsMarked(VertexType) const;
```

```
  private:
    int numVertices;
    int maxVertices;
    VertexType* vertices;
    int **edges;
    bool* marks;
};
```

```
template<class VertexType>
GraphType<VertexType>::GraphType(int maxV)
{
 numVertices = 0;
 maxVertices = maxV;

 vertices = new VertexType[maxV];

 edges = new int[maxV];
 for(int i = 0; i < maxV; i++)
   edges[i] = new int[maxV];

 marks = new bool[maxV];
}
```

```
template<class VertexType>
GraphType<VertexType>::~GraphType()
{
 delete [] vertices;

 for(int i = 0; i < maxVertices; i++)
   delete [] edges[i];
 delete [] edges;

 delete [] marks;
}
```

```
void GraphType<VertexType>::AddVertex(VertexType vertex)
{
 vertices[numVertices] = vertex;

 for(int index = 0; index < numVertices; index++) {
    edges[numVertices][index] = NULL_EDGE;
    edges[index][numVertices] = NULL_EDGE;
 }

 numVertices++;
}
```



graph
.numVertices 7
.vertices

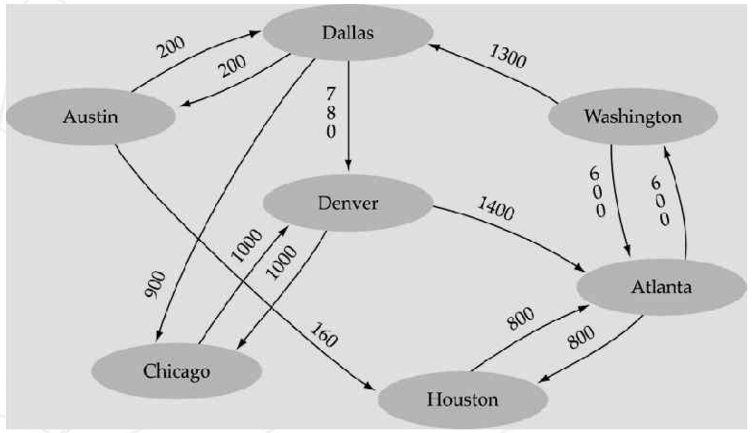
[0]	"Atlanta "
[1]	"Austin "
[2]	"Chicago "
[3]	"Dallas "
[4]	"Denver "
[5]	"Houston "
[6]	"Washington"
[7]	
[8]	
[9]	

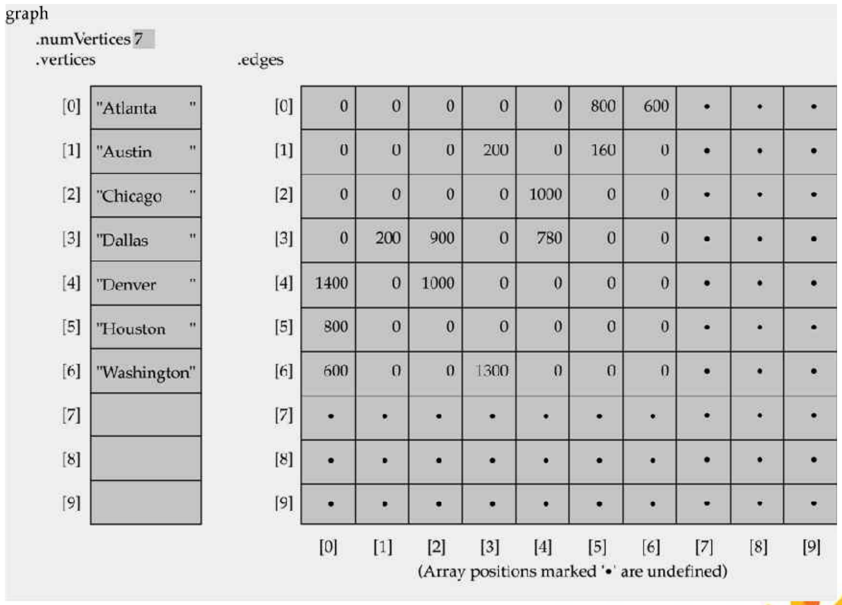
.edges

	[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)

```
template<class VertexType>
void GraphType<VertexType>::AddEdge(VertexType fromVertex,
  VertexType toVertex, int weight)
{
 int row;
 int column;

 row = IndexIs(vertices, fromVertex);
 col = IndexIs(vertices, toVertex);
 edges[row][col] = weight;
}
```




graph
.numVertices 7
.vertices

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

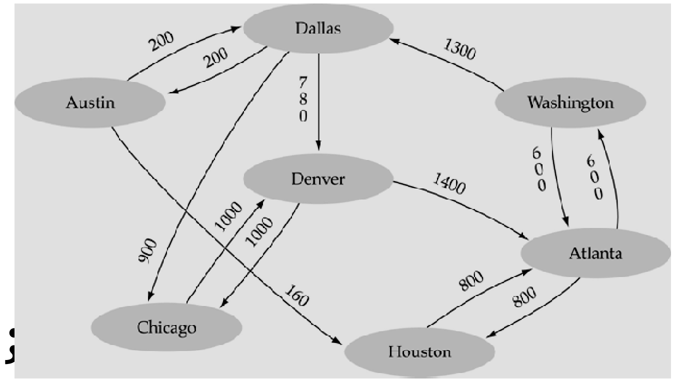
.edges

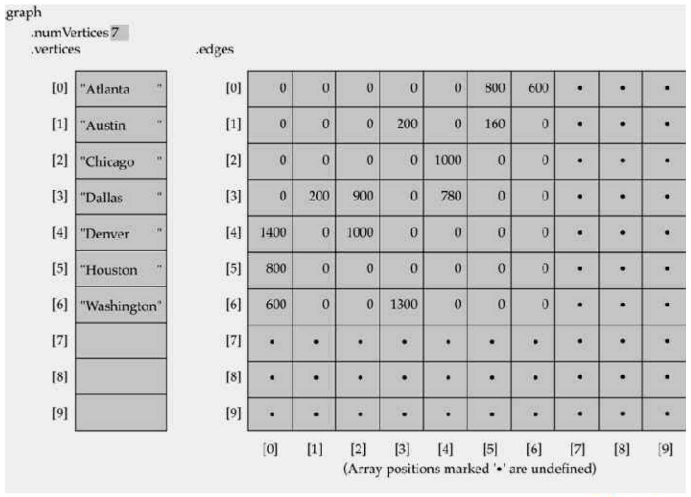
[0]	0	0	0	0	0	800	600	•	•	•
[1]	0	0	0	200	0	160	0	•	•	•
[2]	0	0	0	0	1000	0	0	•	•	•
[3]	0	200	900	0	780	0	0	•	•	•
[4]	1400	0	1000	0	0	0	0	•	•	•
[5]	800	0	0	0	0	0	0	•	•	•
[6]	600	0	0	1300	0	0	0	•	•	•
[7]	•	•	•	•	•	•	•	•	•	•
[8]	•	•	•	•	•	•	•	•	•	•
[9]	•	•	•	•	•	•	•	•	•	•
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]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

(Array positions marked '•' are undefined)

동국대학교 dongguk university

```
template<class VertexType>
int GraphType<VertexType>::WeightIs(VertexType fromVertex,
  VertexType toVertex)
{
  int row;
  int column;

  row = IndexIs(vertices, fromVertex);
  col = IndexIs(vertices, toVertex);
  return edges[row][col];
}
```




graph
.numVertices 7
.vertices

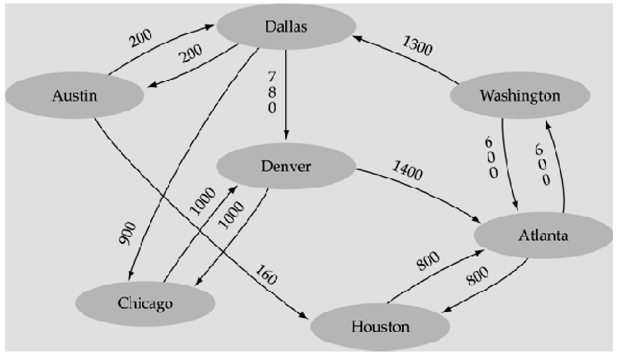
[0]	"Atlanta "
[1]	"Austin "
[2]	"Chicago "
[3]	"Dallas "
[4]	"Denver "
[5]	"Houston "
[6]	"Washington"
[7]	
[8]	
[9]	

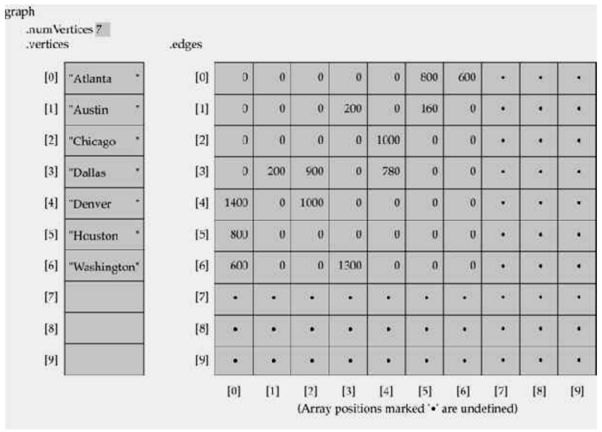
.edges

	[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)

```
template<class VertexType>
void GraphType<VertexType>::GetToVertices(VertexType vertex,
                                 QueTye<VertexType>& adjvertexQ)
{
 int fromIndex;
 int toIndex;

 fromIndex = IndexIs(vertices, vertex);
 for(toIndex = 0; toIndex < numVertices; toIndex++)
   if(edges[fromIndex][toIndex] != NULL_EDGE)
     adjvertexQ.Enqueue(vertices[toIndex]);
}
```





graph
.numVertices 7
.vertices

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

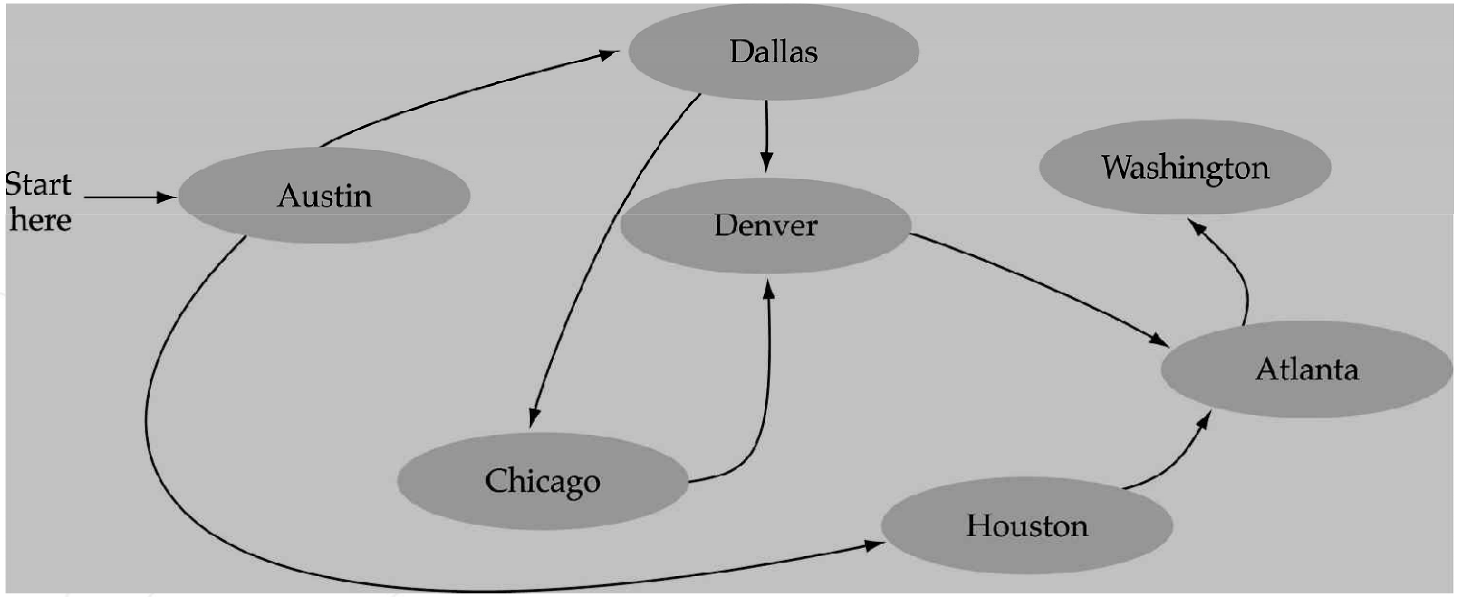
.edges

[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]	•	•	•	•	•	•	•	•	•	•
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

(Array positions marked '•' are undefined)

Graph searching

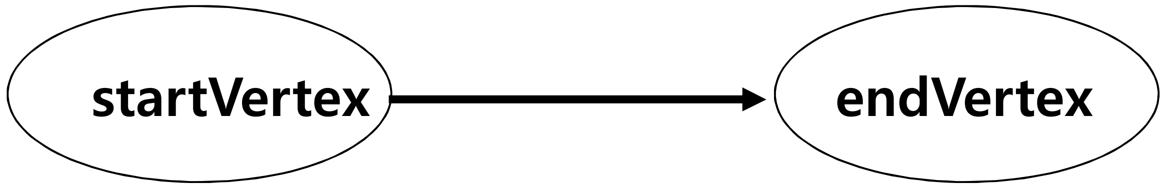
- ***Problem:*** find if there is a path between two vertices of the graph (e.g., Austin and Washington)
- ***Methods:*** Depth-First-Search (DFS) or Breadth-First-Search (BFS)



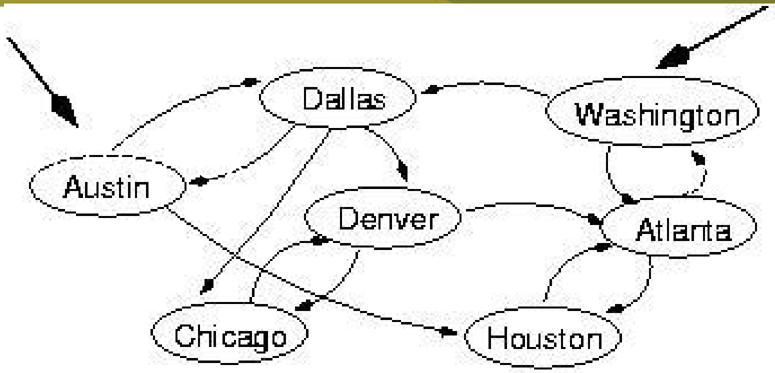
Depth-First-Search (DFS)

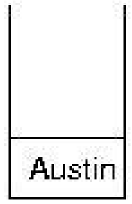
- Main idea:
  - Travel as far as you can down a path
  - Back up *as little as possible* when you reach a "dead end" (i.e., next vertex has been "marked" or there is no next vertex)
- DFS uses a *stack* !

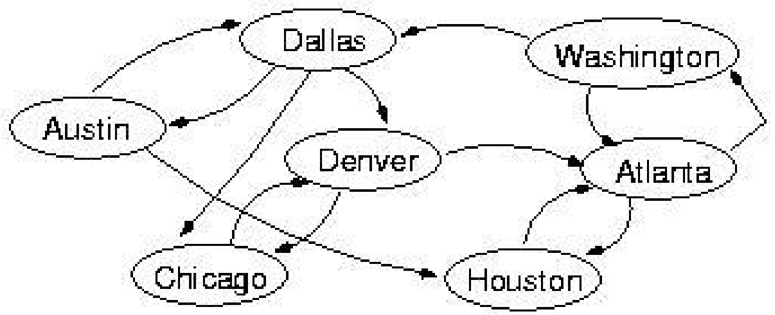
Depth-First-Search (DFS) *(cont.)*



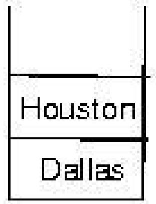
```
found = false
stack.Push(startVertex)
DO
  stack.Pop(vertex)
  IF vertex == endVertex
    found = true
  ELSE
   “mark” vertex
    Push all adjacent, not “marked”, vertices onto stack
WHILE !stack.IsEmpty() AND !found

IF(!found)
  Write "Path does not exist"
```

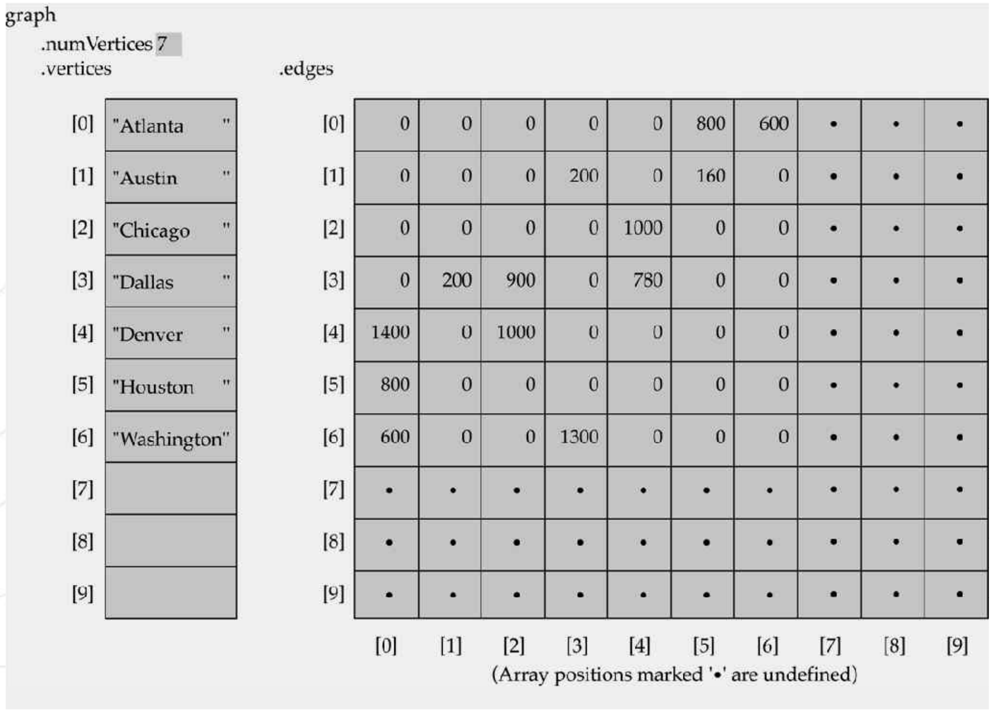






pop Austin





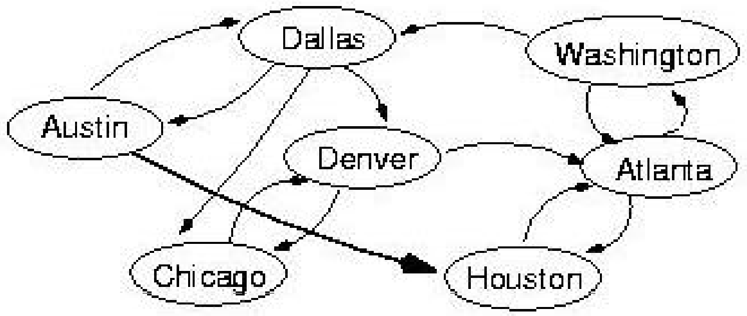
graph
.numVertices 7
.vertices

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

.edges

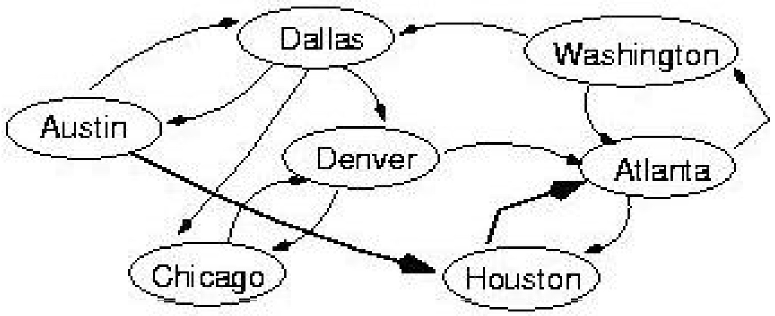
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
[0]	0	0	0	0	0	800	600	•	•	•
[1]	0	0	0	200	0	160	0	•	•	•
[2]	0	0	0	0	1000	0	0	•	•	•
[3]	0	200	900	0	780	0	0	•	•	•
[4]	1400	0	1000	0	0	0	0	•	•	•
[5]	800	0	0	0	0	0	0	•	•	•
[6]	600	0	0	1300	0	0	0	•	•	•
[7]	•	•	•	•	•	•	•	•	•	•
[8]	•	•	•	•	•	•	•	•	•	•
[9]	•	•	•	•	•	•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

(Array positions marked '•' are undefined)



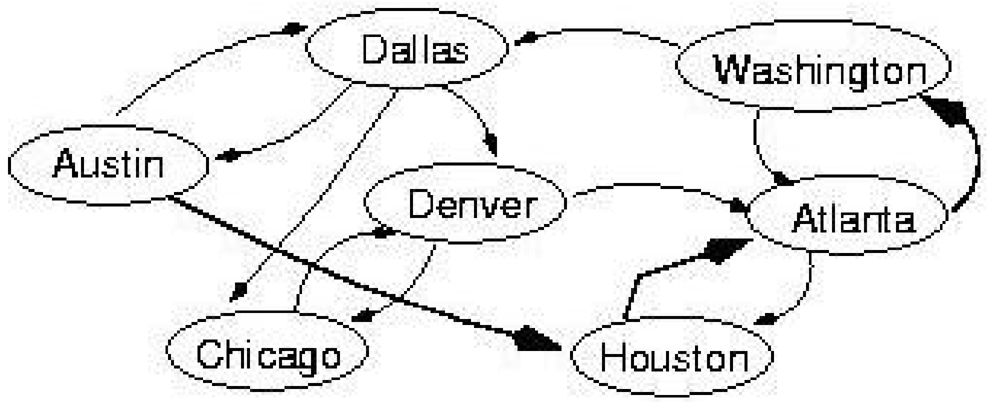
pop Houston

Atlanta
Dallas

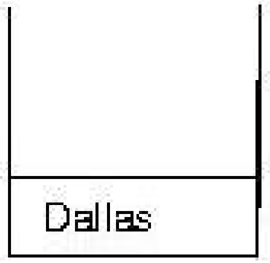


pop Atlanta

Washington
Dallas



pop Washington



```
template <class VertexType>
void DepthFirstSearch(GraphType<VertexType> graph,
  VertexType startVertex, VertexType endVertex)
{
 StackType<VertexType> stack;
 QueType<VertexType> vertexQ;

 bool found = false;
 VertexType vertex;
 VertexType item;

 graph.ClearMarks();
 stack.Push(startVertex);
 do {
   stack.Pop(vertex);
   if(vertex == endVertex)
     found = true;
```

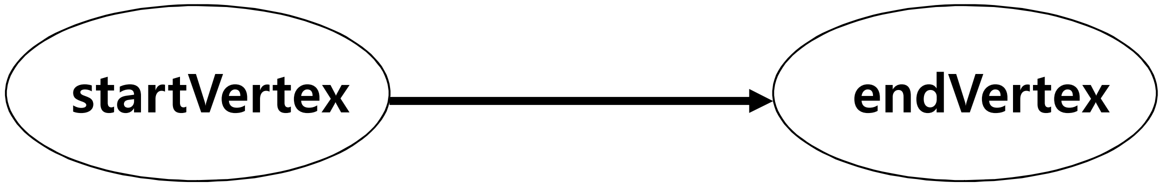
(continues)

```
  else
      if(!graph.IsMarked(vertex)) {
        graph.MarkVertex(vertex);
        graph.GetToVertices(vertex, vertexQ);

        while(!vertexQ.IsEmpty()) {
          vertexQ.Dequeue(item);
          if(!graph.IsMarked(item))
            stack.Push(item);
        }
      }

    } while(!stack.IsEmpty() && !found);

    if(!found)
      cout << "Path not found" << endl;
}
```

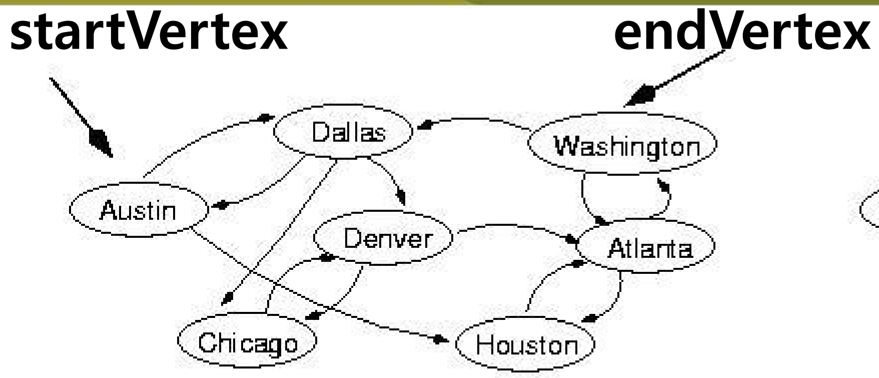
Breadth-First-Searching (BFS)

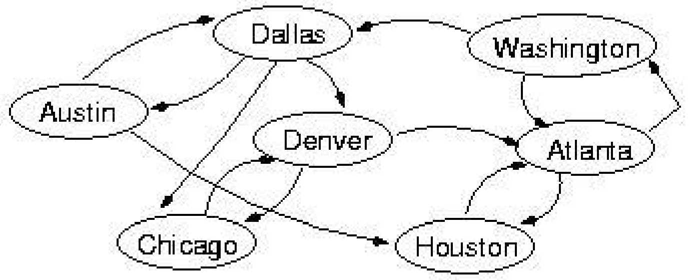
- Main idea:
  - Look at all possible paths at the same depth before you go at a deeper level
  - Back up *as far as possible* when you reach a "dead end" (i.e., next vertex has been "marked" or there is no next vertex)
- BFS uses a *queue* !

Breadth-First-Searching (BFS) (cont.)



```
found = false
queue.Enqueue(startVertex)
DO
  queue.Dequeue(vertex)
  IF vertex == endVertex
    found = true
  ELSE
   “mark” vertex
    Enqueue all adjacent, not “marked”,
 vertices onto queue
WHILE !queue.IsEmpty() AND !found

IF(!found)
  Write "Path does not exist"
```



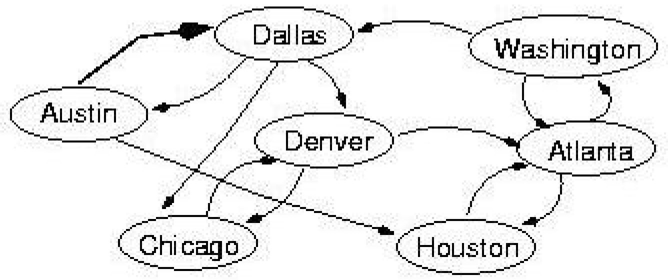


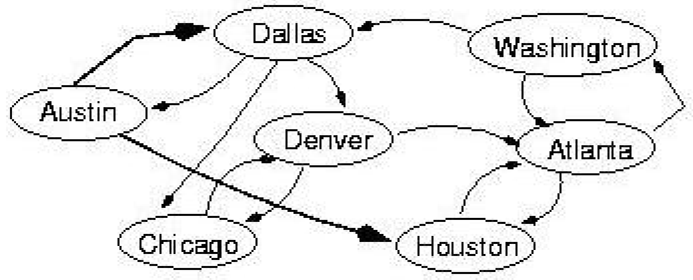
(initialization)

			Austin	

dequeue Austin

		Dallas	Houston	



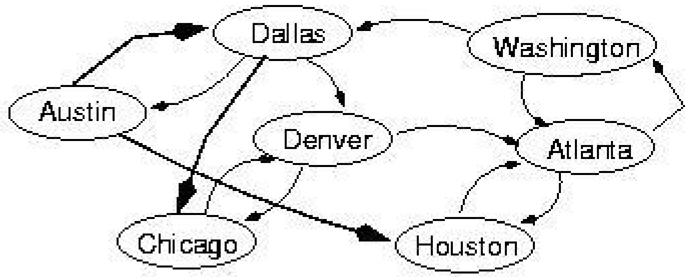


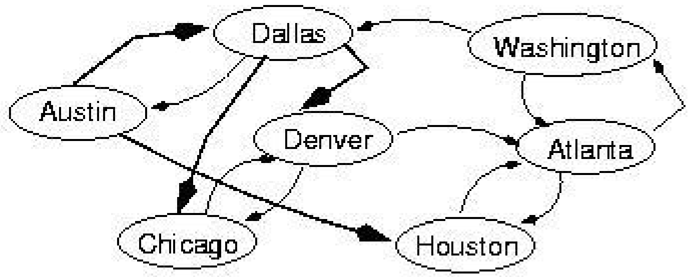
dequeue Dallas

	Houston	Chicago	Denver	

dequeue Houston

	Chicago	Denver	Atlanta	

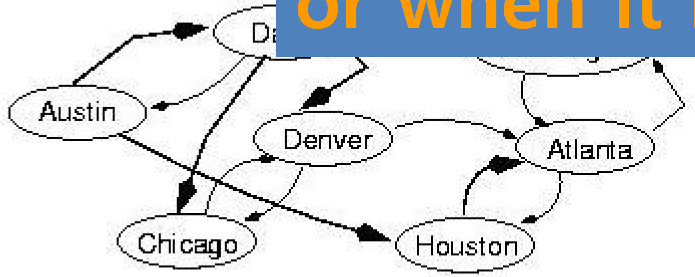


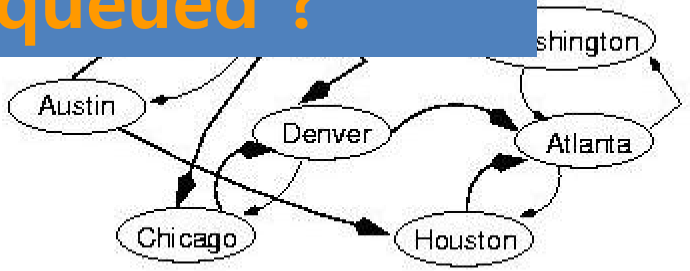


dequeue Chicago

	er	Atlanta

Duplicates: should we <u>mark</u> a vertex when it is Enqueued or when it is Dequeued ?





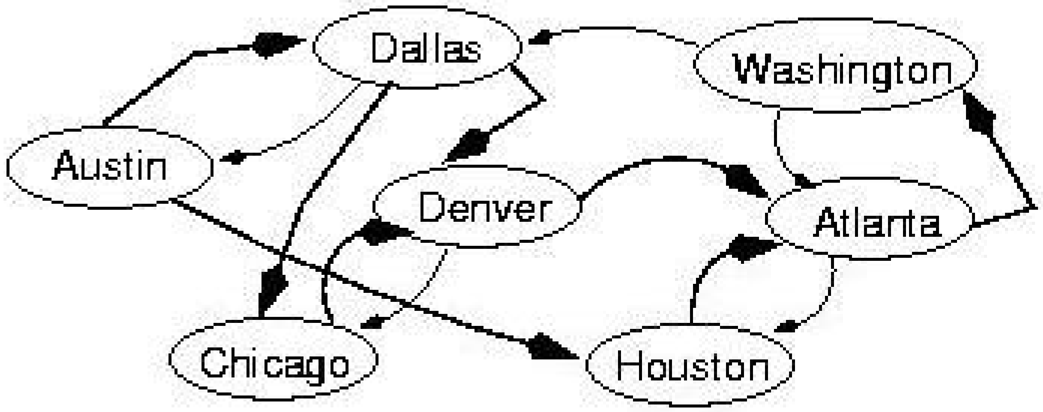
dequeue Atlanta

	Denver	Atlanta	Washington

. . . .

dequeue Denver, Atlanta

	Washington	Washington



dequeue Washington

		Washington	

```
template<class VertexType>
void BreadthFirtsSearch(GraphType<VertexType> graph,
  VertexType startVertex, VertexType endVertex);
{
 QueType<VertexType> queue;
 QueType<VertexType> vertexQ;

 bool found = false;
 VertexType vertex;
 VertexType item;

 graph.ClearMarks();
 queue.Enqueue(startVertex);
 do {
   queue.Dequeue(vertex);
   if(vertex == endVertex)
     found = true;
```

(continues)

“mark” when dequeue a vertex
→ allow duplicates!

```
else
    if(!graph.IsMarked(vertex)) {
      graph.MarkVertex(vertex);
      graph.GetToVertices(vertex, vertexQ);

      while(!vertxQ.IsEmpty()) {
        vertexQ.Dequeue(item);
        if(!graph.IsMarked(item))
          queue.Enqueue(item);
      }
    }

} while (!queue.IsEmpty() && !found);

if(!found)
  cout << "Path not found" << endl;
}
```

Time Analysis

```
template<class VertexType>
void BreadthFirtsSearch(GraphType<VertexType> graph,
  VertexType startVertex, VertexType endVertex);
{
 QueType<VertexType> queue;
 QueType<VertexType> vertexQ;

 bool found = false;
 VertexType vertex;
 VertexType item;

 graph.ClearMarks();
 queue.Enqueue(startVertex);
 do {
   queue.Dequeue(vertex);
   if(vertex == endVertex)
     found = true;
```

O(V)

$\leftarrow$ O(V) times

(continues)

```
else {
    if(!graph.IsMarked(vertex)) {
      graph.MarkVertex(vertex);
      graph.GetToVertices(vertex, vertexQ);

      while(!vertxQ.IsEmpty()) {
        vertexQ.Dequeue(item);
        if(!graph.IsMarked(item))
          queue.Enqueue(item);
      }
    }
  }
} while (!queue.IsEmpty() && !found);

if(!found)
  cout << "Path not found" << endl;
}
```

$O(V)$ – arrays
$O(E_{vi})$ – linked lists

$O(E_{Vi})$ times

Arrays: $O(V+V^2+E_{v1}+E_{v2}+\ldots)=O(V^2+E)=O(V^2)$

```
else {
    if(!graph.IsMarked(vertex)) {
      graph.MarkVertex(vertex);
      graph.GetToVertices(vertex, vertexQ);

      while(!vertxQ.IsEmpty()) {
        vertexQ.Dequeue(item);
        if(!graph.IsMarked(item))
          queue.Enqueue(item);
      }
    }
  }
} while (!queue.IsEmpty() && !found);

if(!found)
  cout << "Path not found" << endl;
}
```

O(V) - arrays

$O(E_{vi})$ – linked lists

$O(E_{Vi})$ times

Linked Lists: $O(V+2E_{v1}+2E_{v2}+\ldots)=O(V+E)$

$O(V^2)$ dense

$O(V)$ sparse