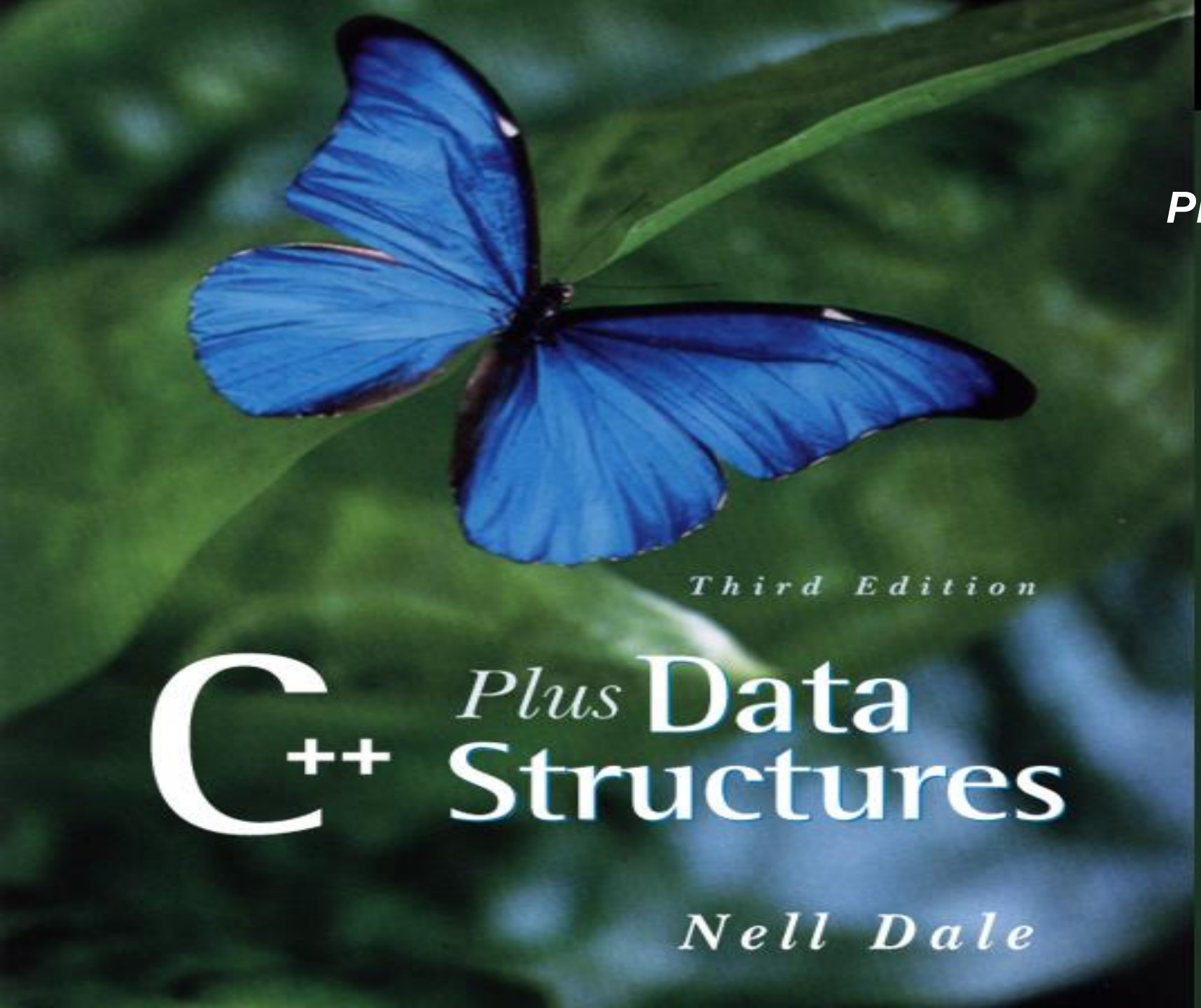


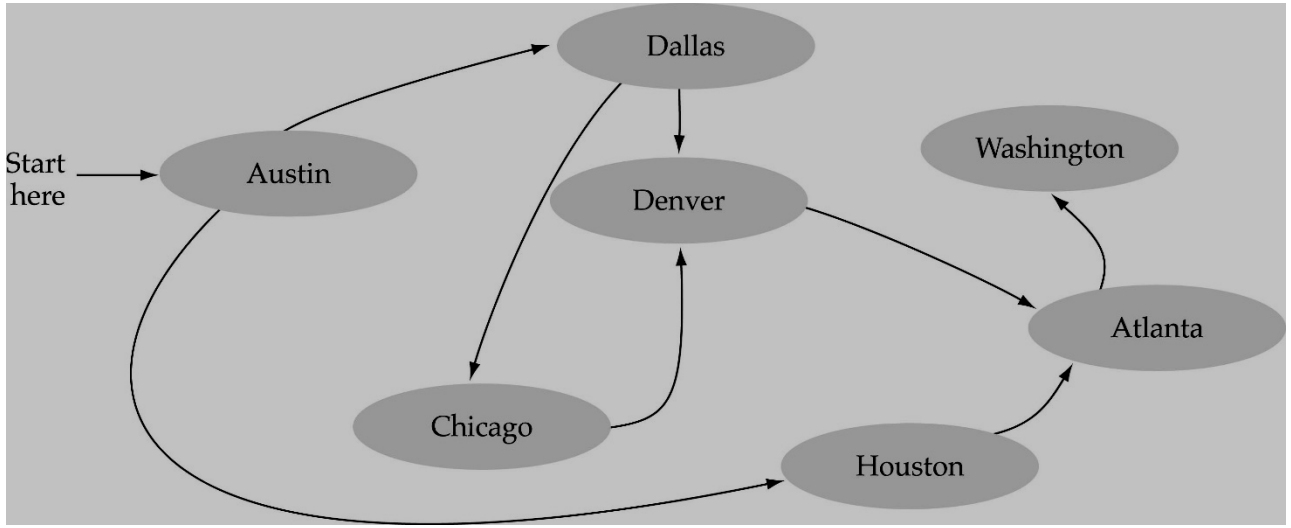
Chapter

9

Priority Queues, Heaps, and Graphs

What is a graph?

- A data structure that consists of a set of nodes (*vertices*) and a set of edges that relate the nodes to each other
- The set of edges describes relationships among the vertices



Formal definition of graphs

- A graph *G* is defined as follows:

  $G=(V,E)$

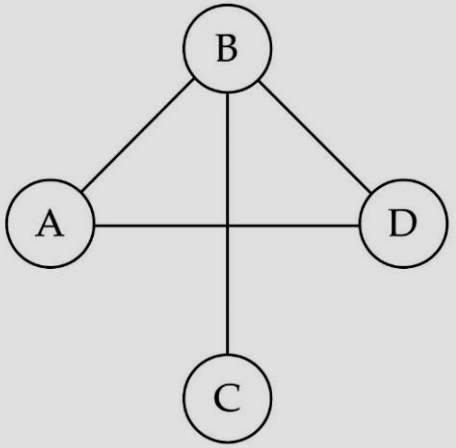
  V(G): a finite, nonempty set of vertices

  E(G): a set of edges (pairs of vertices)

Directed vs. undirected graphs

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

ected graph.

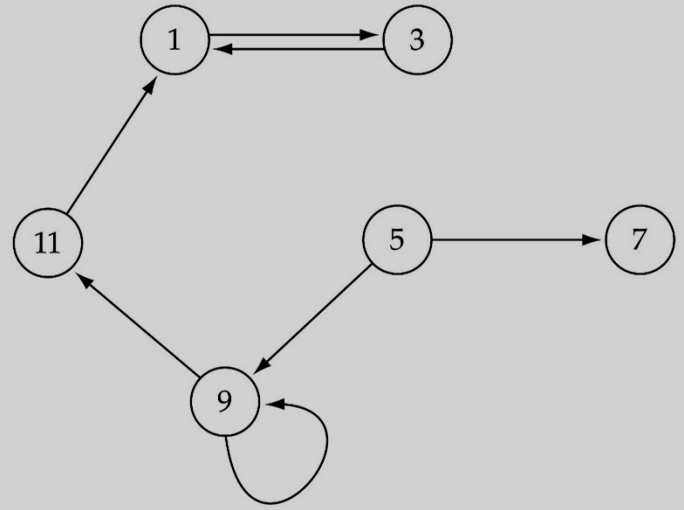


V(Graph1) = { A, B, C, D }
E(Graph1) = { (A, B), (A, D), (B, C), (B, D) }

Directed vs. undirected graphs (cont.)

- When the edges in a graph have a direction, the graph is called *directed* (or *digraph*)

(b) Graph2 is a directed graph.



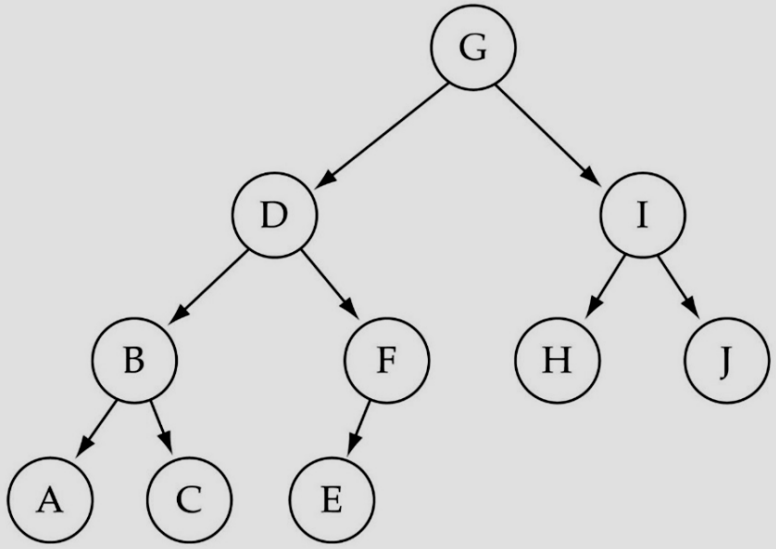
V(Graph2) = { 1, 3, 5, 7, 9, 11 }
E(Graph2) = {(1,3) (3,1) (5,9) (9,11) (5,7) (9, 9), (11, 1) }

Warning: if the graph is directed, the order of the vertices in each edge is important !!

Trees vs. graphs

- Trees are special cases of graphs!!

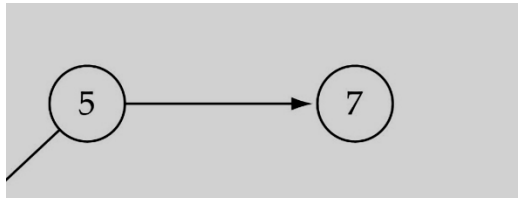
(c) Graph3 is a directed graph.



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



5 is adjacent to 7
7 is adjacent from 5

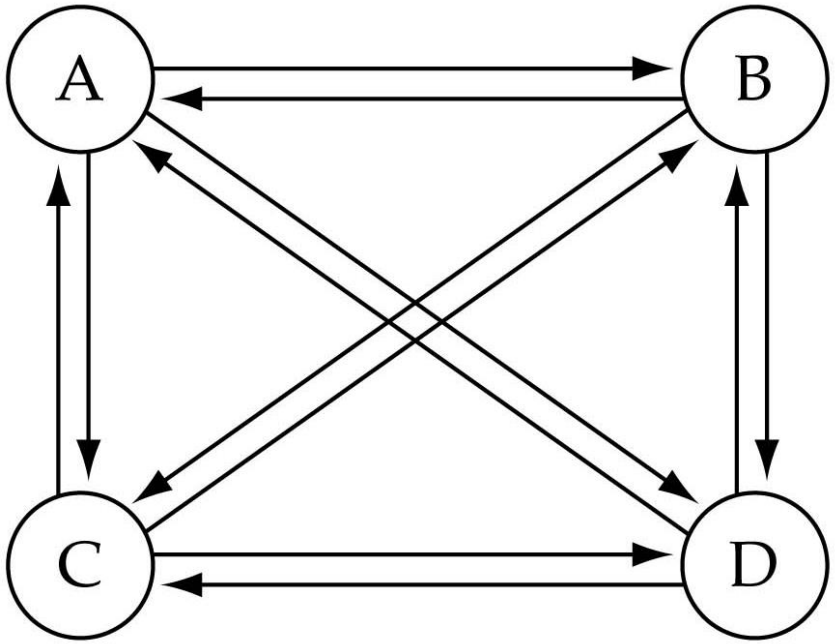
- Path: a sequence of vertices that connect two nodes in a graph
- Complete graph: a graph in which every vertex is directly connected to every other vertex

Graph terminology (cont.)

- What is the number of edges in a complete directed graph with N vertices?

*N * (N-1)*

$O(N^2)$



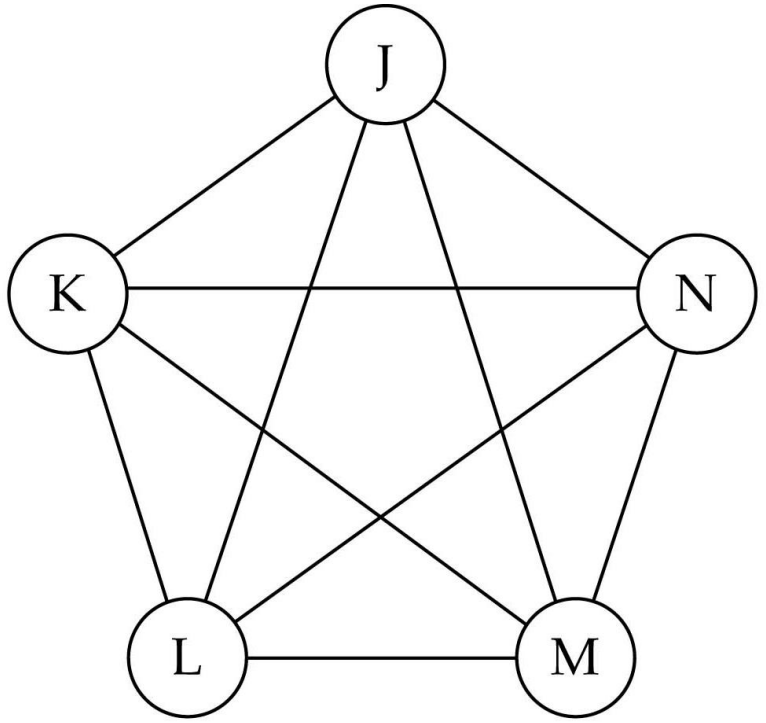
(a) Complete directed graph.

Graph terminology (cont.)

- What is the number of edges in a complete undirected graph with N vertices?

*N * (N-1) / 2*

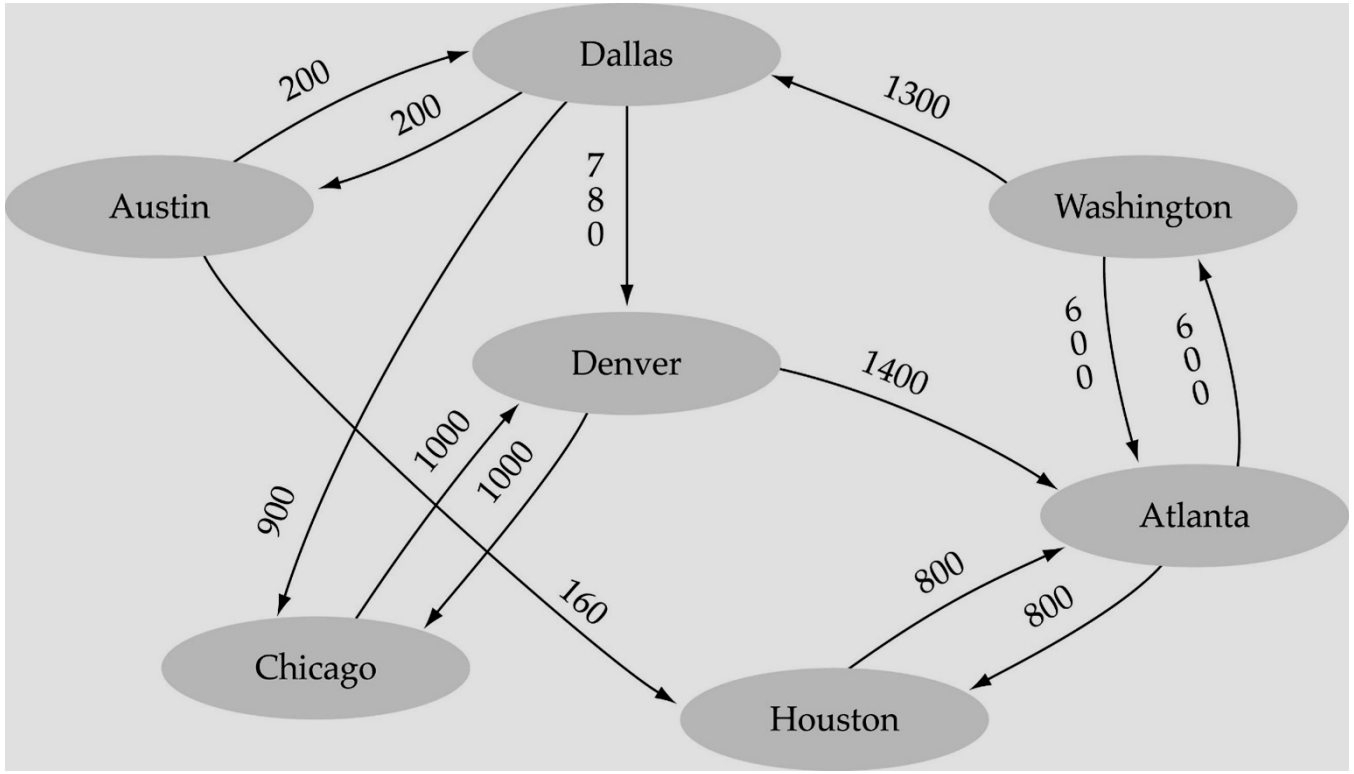
$O(N^2)$



(b) Complete undirected graph.

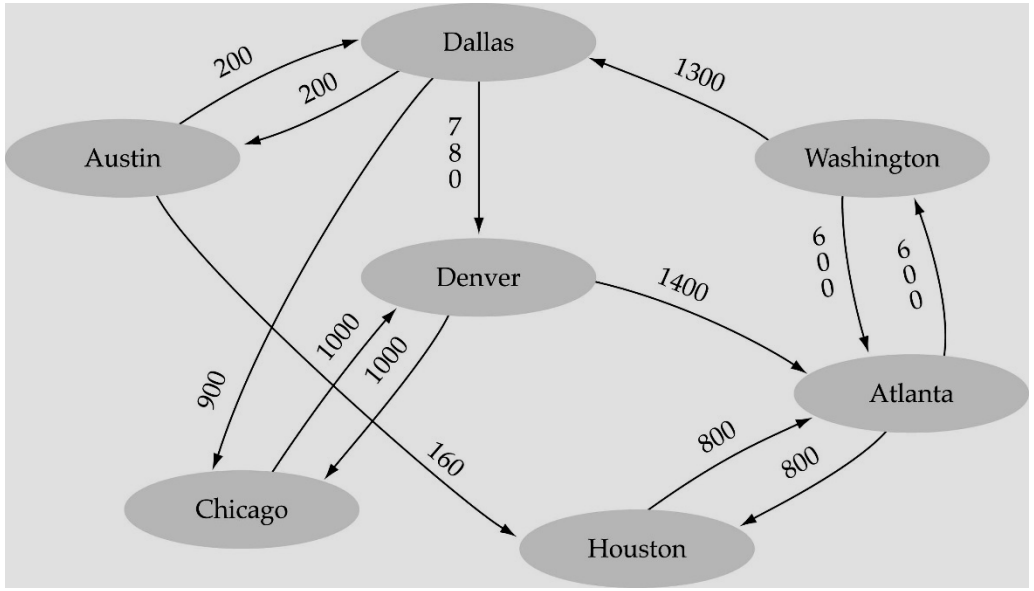
Graph terminology (cont.)

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



Graph implementation

- Adjacency Matrix: Array-based implementation
  - A 1D array is used to represent the vertices
  - A 2D array (adjacency matrix) is used to represent the edges



Array-based implementation

graph

.numVertices 7

.vertices

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

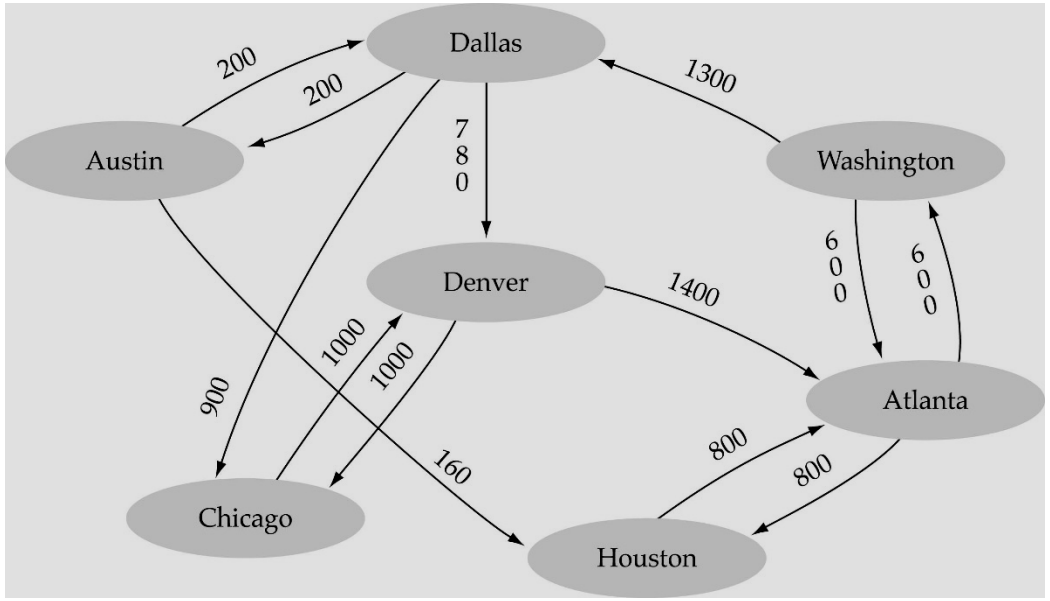
.edges

	[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)

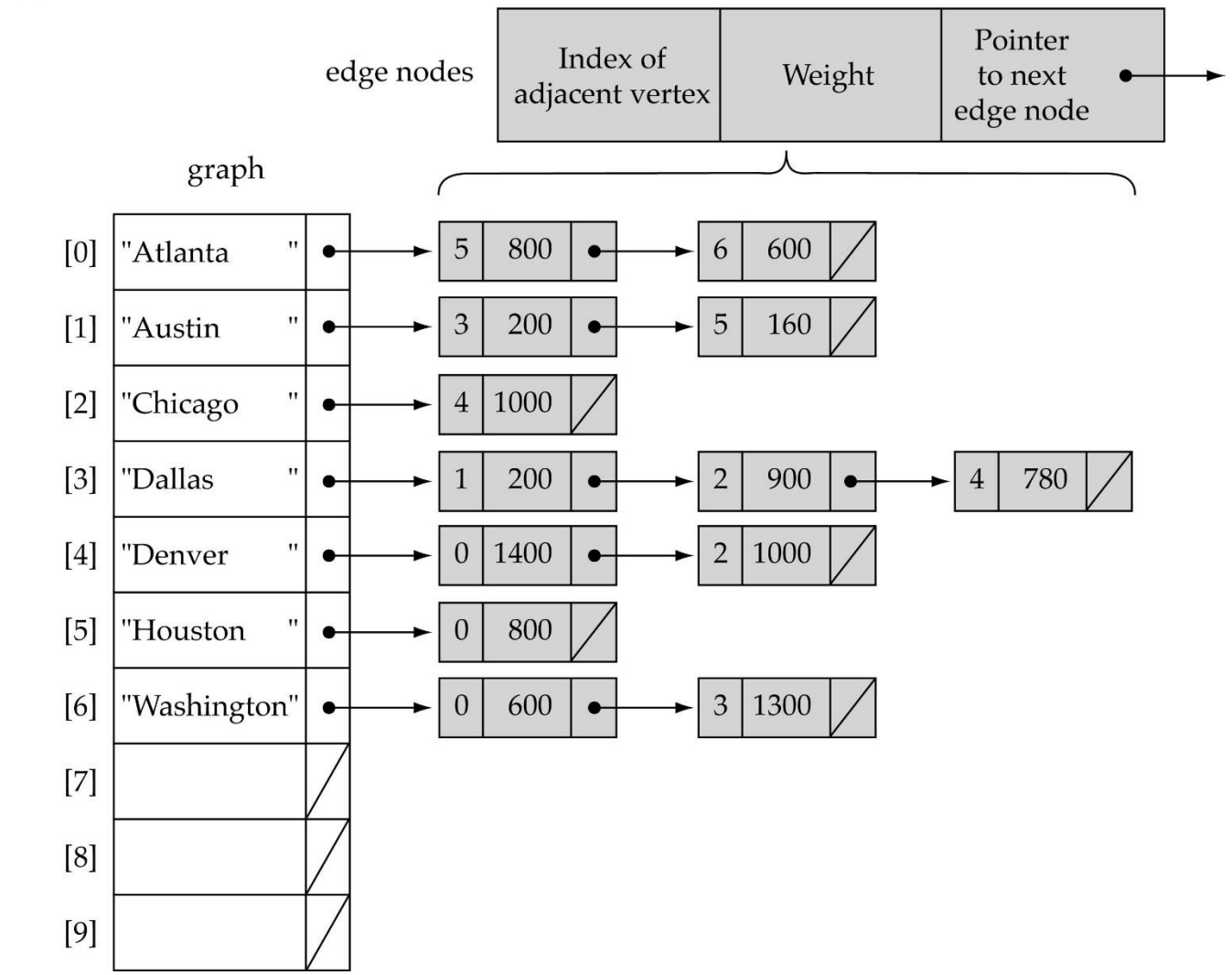
Graph implementation (cont.)

- Adjacency List: Linked-list implementation
  - A 1D array is used to represent the vertices
  - A list is used for each vertex *v* which contains the vertices which are adjacent from *v*



Linked-list implementation

(a)



Adjacency matrix vs. adjacency list representation

- **Adjacency matrix**
  - Good for dense graphs --$|E| \sim O(|V|^2)$
  - Memory requirements: $O(|V| + |E|) = O(|V|^2)$
  - Connectivity between two vertices can be tested quickly

- **Adjacency list**
  - Good for sparse graphs -- $|E| \sim O(|V|)$
  - Memory requirements: $O(|V| + |E|) = O(|V|)$
  - Vertices adjacent to another vertex can be found quickly

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;
};
```

(continues)

```
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;
}
```

(continues)

```
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++;
}

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;
}
```

(continues)

```
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 searching

- *Problem*: find a path between two nodes of the graph (e.g., Austin and Washington)

- *Methods*: Depth-First-Search (DFS) or Breadth-First-Search (BFS)

Depth-First-Search (DFS)

- What is the idea behind DFS?
  - Visit all nodes in a branch to its deepest point before moving up
  - Travel as far as you can down a path
- DFS can be implemented efficiently using a *stack*

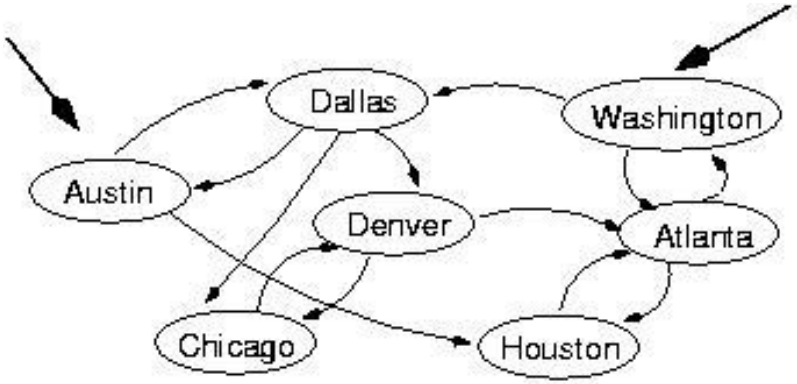
Depth-First-Search (DFS) *(cont.)*

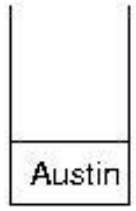
```
Set found to false
stack.Push(startVertex)
DO
  stack.Pop(vertex)
  IF vertex == endVertex
    Set found to true
  ELSE
    Push all adjacent vertices onto stack
WHILE !stack.IsEmpty() AND !found

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

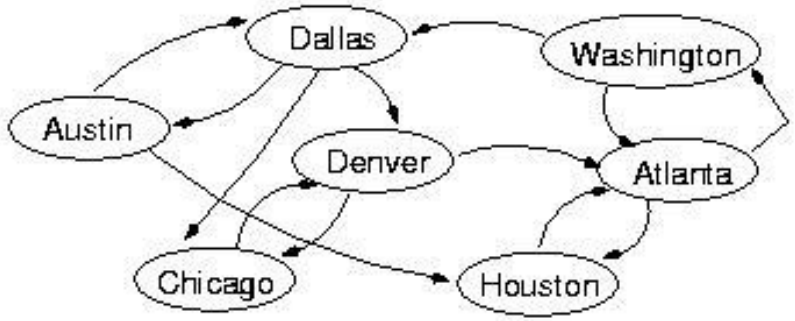
start

end

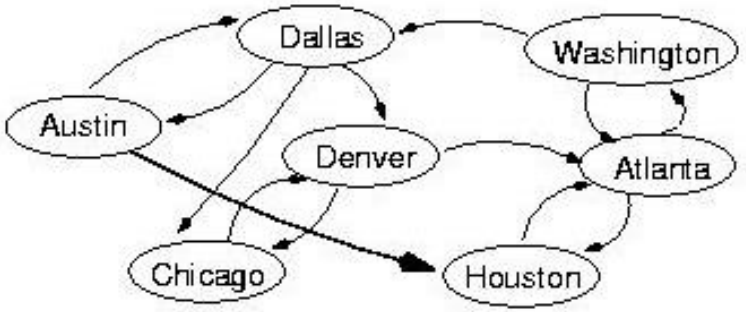




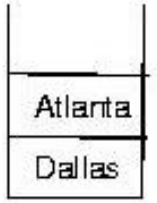
(initialization)

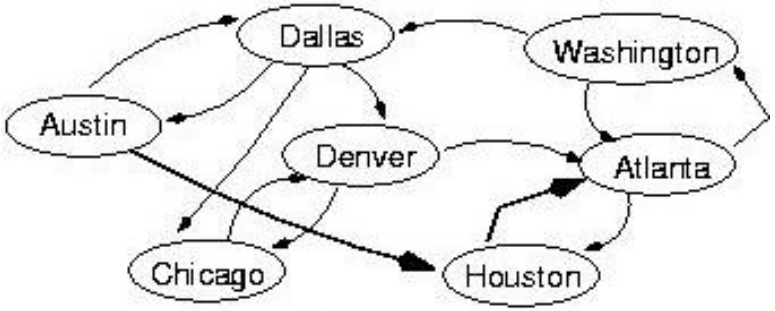


pop Austin

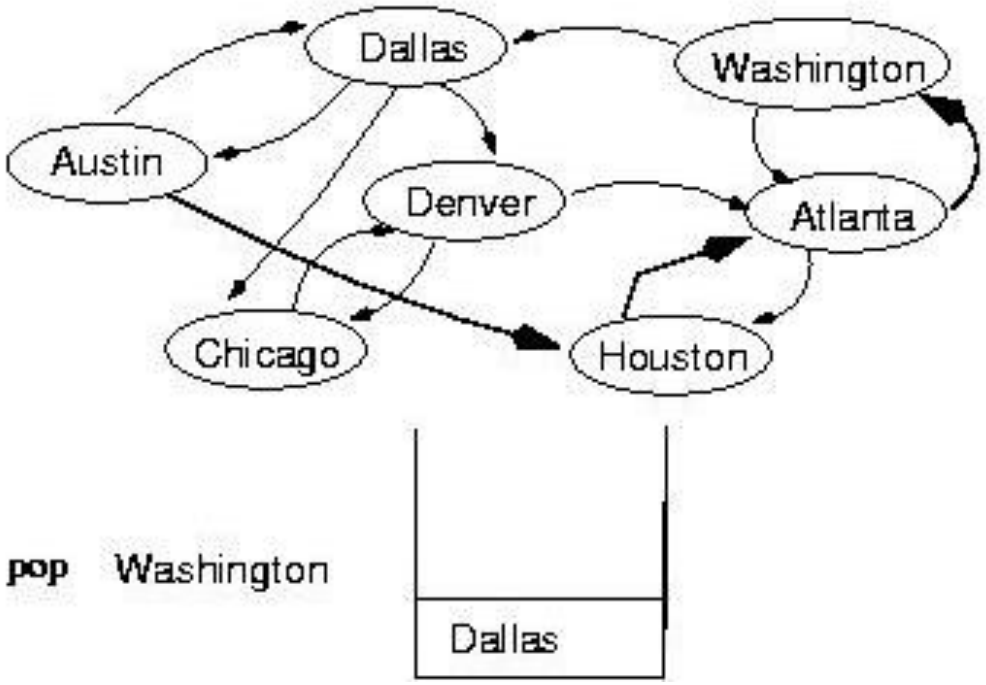


pop Houston





pop Atlanta


Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston
pop Washington
Dallas

```
template <class ItemType>
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;
}
```

(continues)

```
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]);
}
```

Breadth-First-Searching (BFS)

- What is the idea behind BFS?
  - Visit all the nodes on one level before going to the next level
  - Look at all possible paths at the same depth before you go at a deeper level

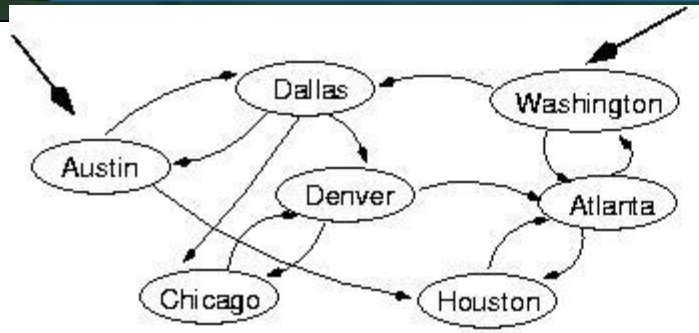
Breadth-First-Searching (BFS) (cont.)

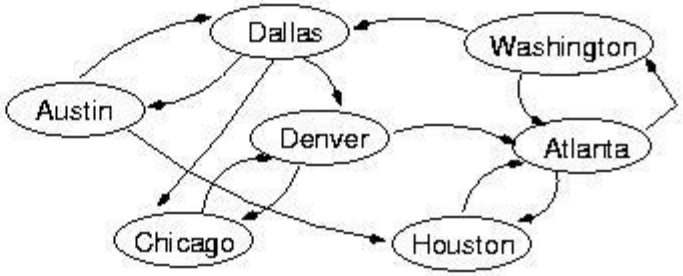
- BFS can be implemented efficiently using a *queue*

```
Set found to false
queue.Enqueue(startVertex)
DO
  queue.Dequeue(vertex)
  IF vertex == endVertex
    Set found to true
  ELSE
    Enqueue all adjacent vertices onto queue
WHILE !queue.IsEmpty() AND !found
IF(!found)
  Write "Path does not exist"
```

- Should we mark a vertex when it is enqueued or when it is dequeued ?

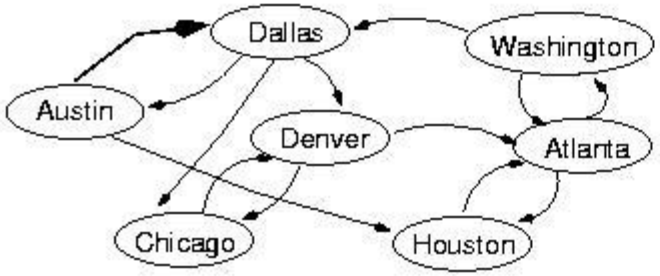
start end

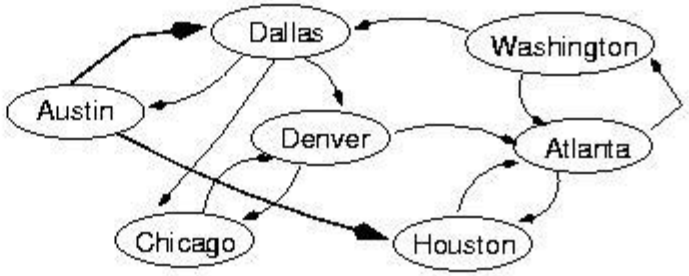




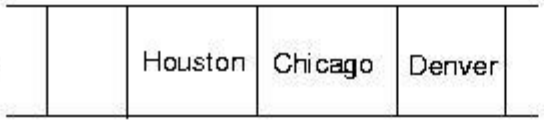
(initialization)

dequeue Austin

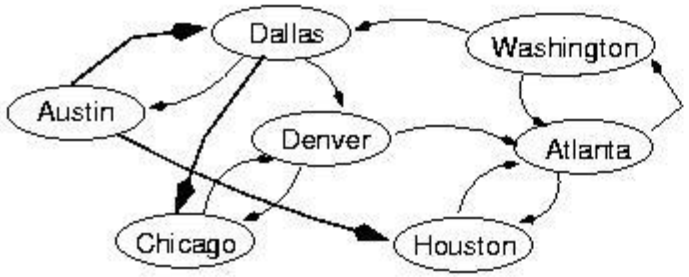




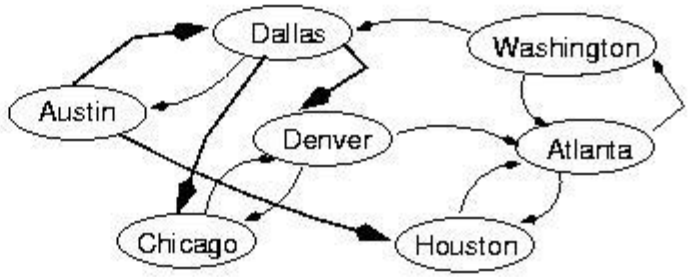
dequeue Dallas



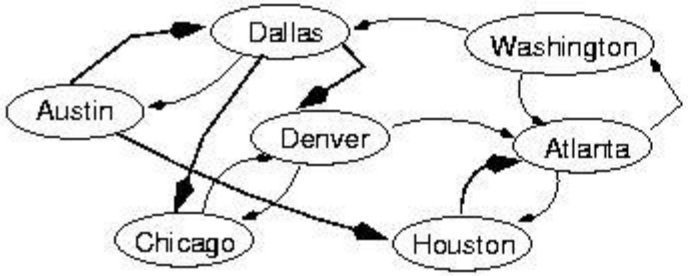
dequeue Houston



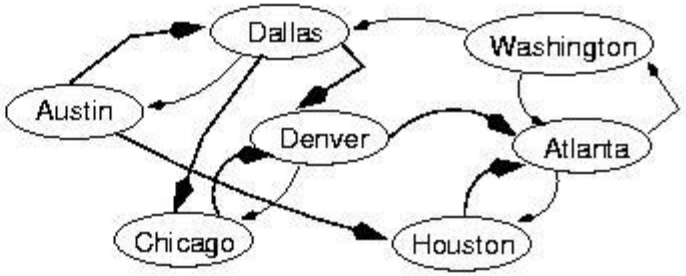
dequeue Chicago | | Denver | Atlanta | Denver |



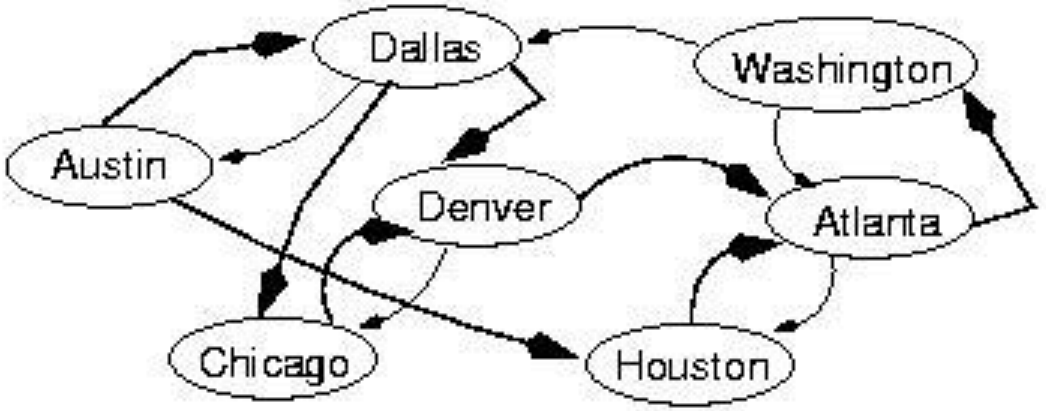
dequeue Denver | | Atlanta | Denver | Atlanta |

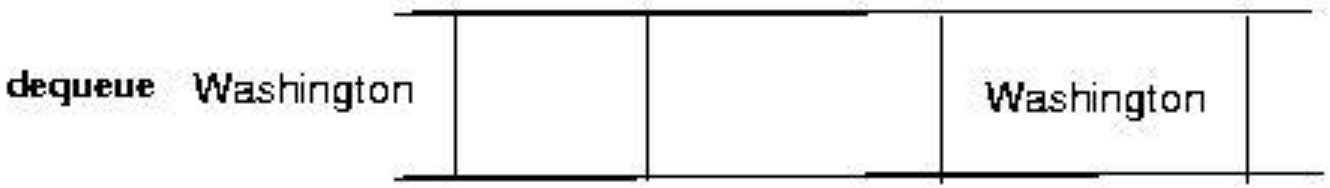


dequeue Atlanta | | Denver | Atlanta | Washington |



dequeue Denver, next: Atlanta | | Washington | Washington |


Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston


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)

```
    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;
}
```

Single-source shortest-path problem

- There are multiple paths from a source vertex to a destination vertex
- *Shortest path*: the path whose total weight (i.e., sum of edge weights) is minimum
- Examples:
  - Austin->Houston->Atlanta->Washington: 1560 miles
  - Austin->Dallas->Denver->Atlanta->Washington: 2980 miles

Single-source shortest-path problem (cont.)

- Common algorithms: *Dijkstra's* algorithm, *Bellman-Ford* algorithm
- BFS can be used to solve the shortest graph problem when the graph is weightless or all the weights are the same

(mark vertices before Enqueue)

ADT Set Definitions

Base type: The type of the items in the set

***Cardinality*:** The number of items in a set

***Cardinality of the base type*:** The number of items in the base type

Union of two sets: A set made up of all the items in either sets

***Intersection of two sets*:** A set made up of all the items in both sets

***Difference of two sets*:** A set made up of all the items in the first set that are not in the second set

Beware: At the Logical Level

- Sets can not contain duplicates. Storing an item that is already in the set does not change the set.
- If an item is not in a set, deleting that item from the set does not change the set.
- Sets are not ordered.

Implementing Sets

Explicit implementation (Bit vector)

Each item in the base type has a representation in each instance of a set. The representation is either true (item is in the set) or false (item is not in the set).

Space is proportional to the cardinality of the base type.

Algorithms use Boolean operations.

Implementing Sets (cont.)

Implicit implementation (List)

The items in an instance of a set are on a list that represents the set. Those items that are not on the list are not in the set.

Space is proportional to the cardinality of the set instance.

Algorithms use ADT List operations.

Explain:

Although sets are not ordered, why is the SortedList ADT a better choice as the implementation structure for the implicit representation?