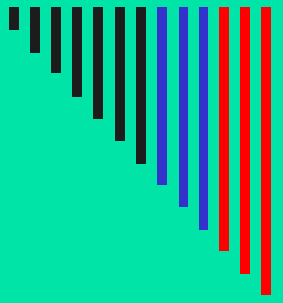


Chapter 11

Graphs

Objectives

Upon completion you will be able to:

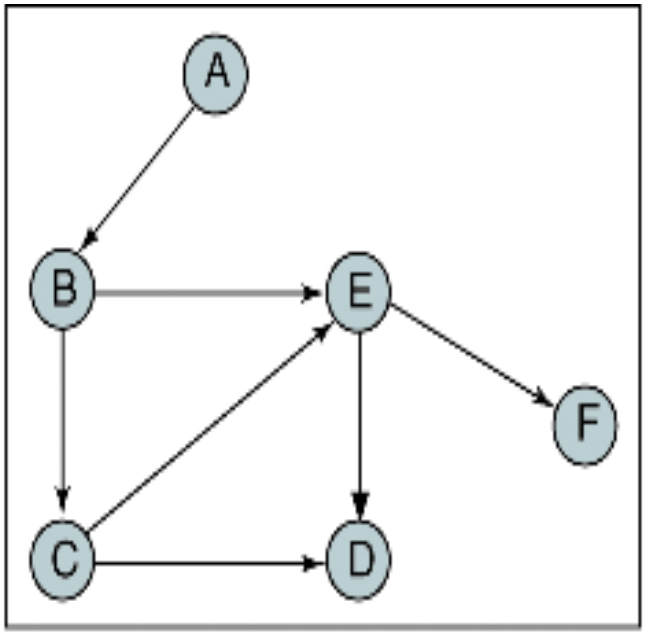
- Understand and use basic graph terminology and concepts
- Define and discuss basic graph and network structures
- Design and implement graph and network applications
- Design and implement applications using the graph ADT
- Define and discuss Dijkstra's shortest path algorithm

11-1 Basic Concepts

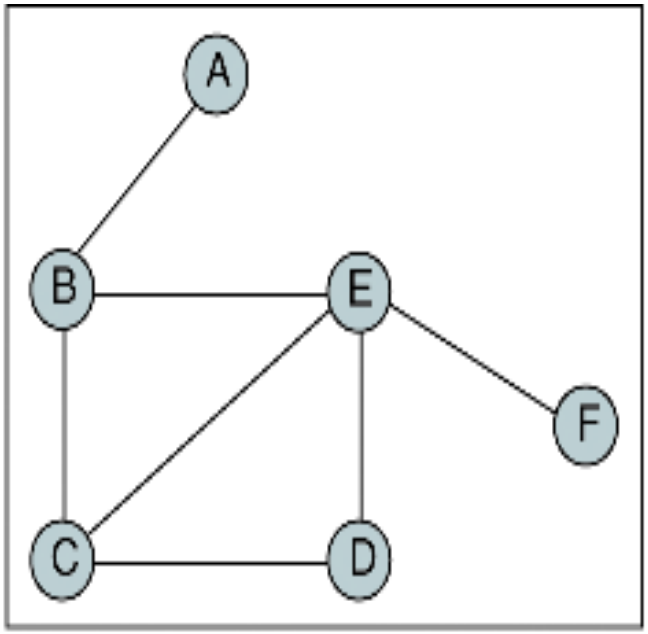
A graph is a collection of nodes, called vertices, and a collection of segments, called lines, connecting pairs of vertices. In Basic Concepts we develop the terminology and structure for graphs. In addition to the graph definitions, discussion points include:

- **Directed and Undirected Graphs**
- **Cycles and Loops**
- **Connected and Disjoint Graphs**

Directed and Undirected Graphs

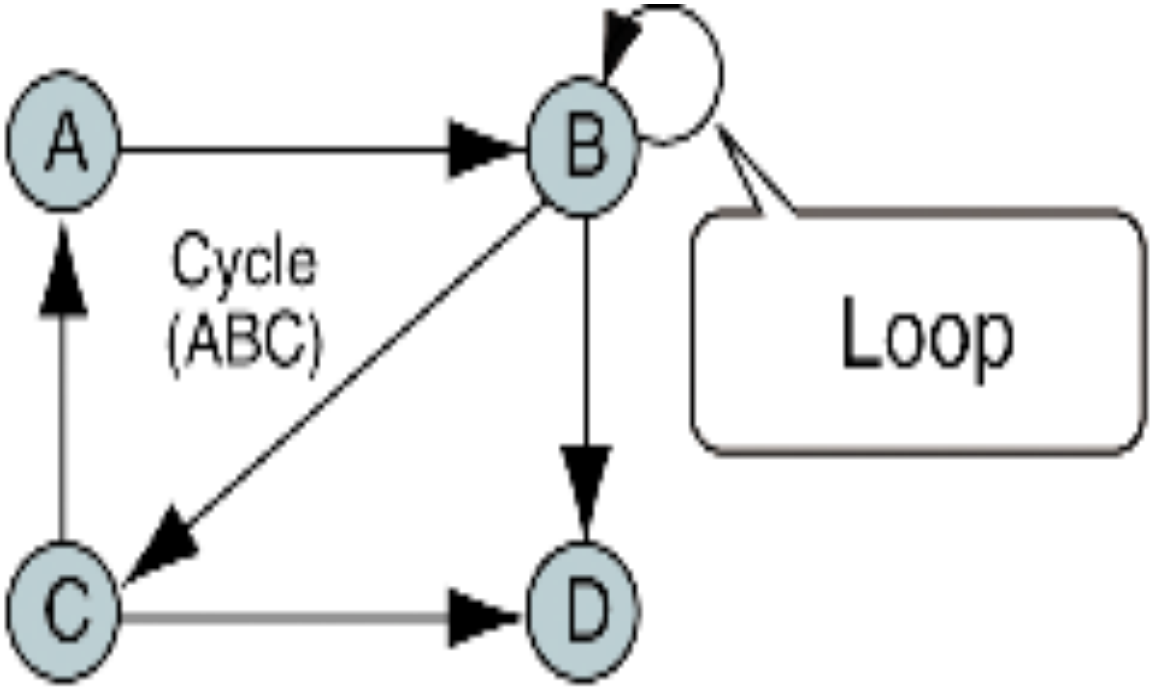


(a) Directed graph

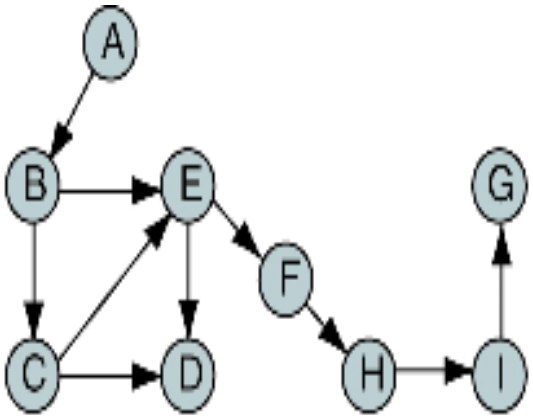


(b) Undirected graph

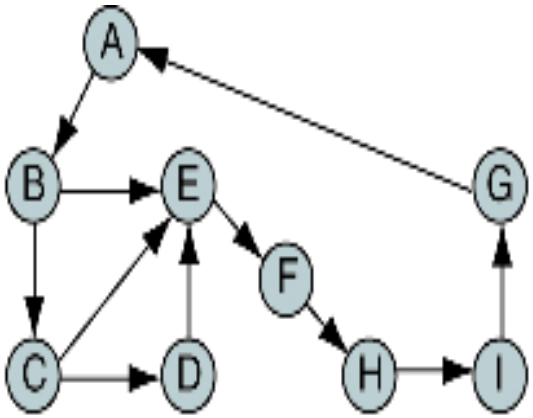
Cycles and Loops



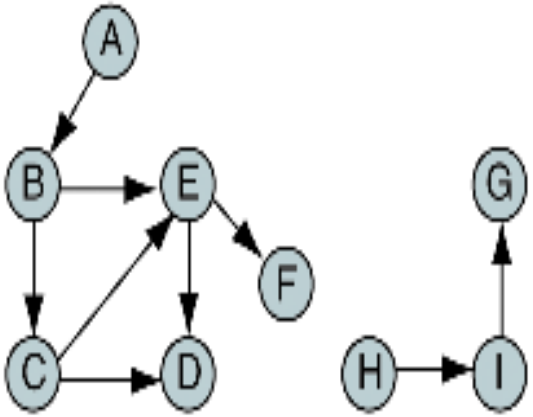
Connected and Disjoint Graphs



(a) Weakly connected



(b) Strongly connected



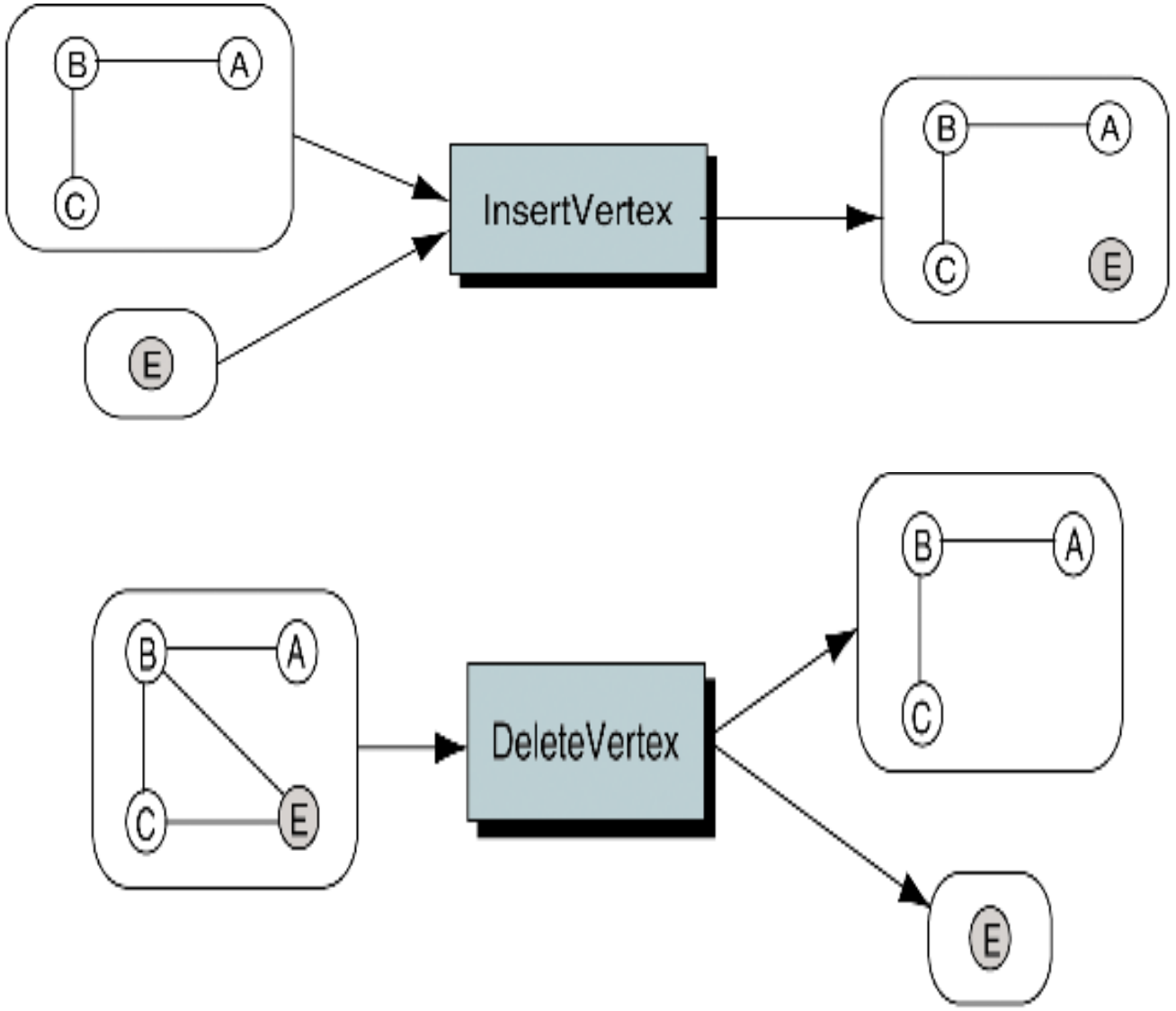
(c) Disjoint graph

11-2 Operations

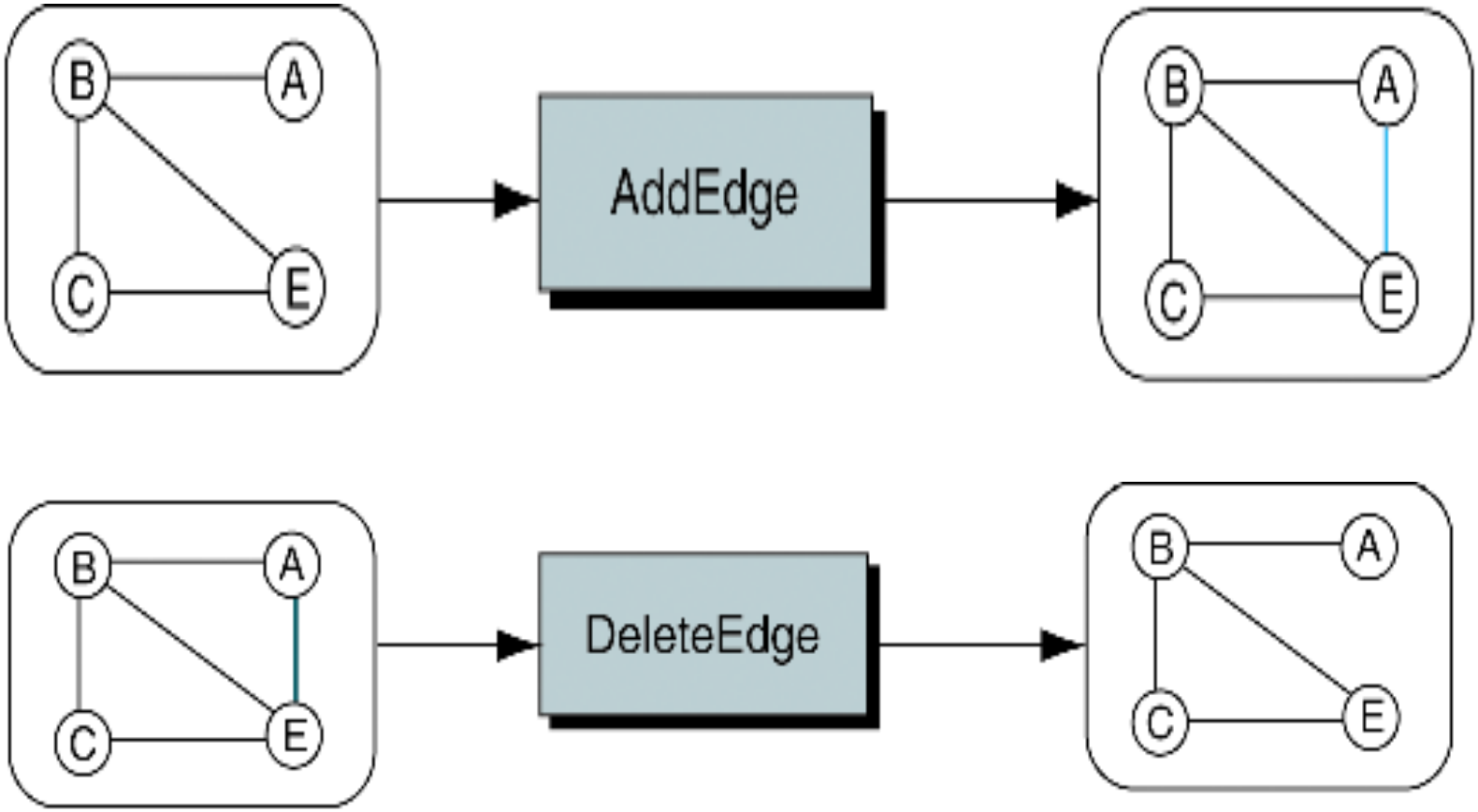
We define and discuss the six primitive graph operations required to maintain a graph.

- **Insert Vertex** 頂點
- **Delete Vertex**
- **Add Edge**
- **Delete Edge**
- **Find Vertex**
- **Traverse Graph**

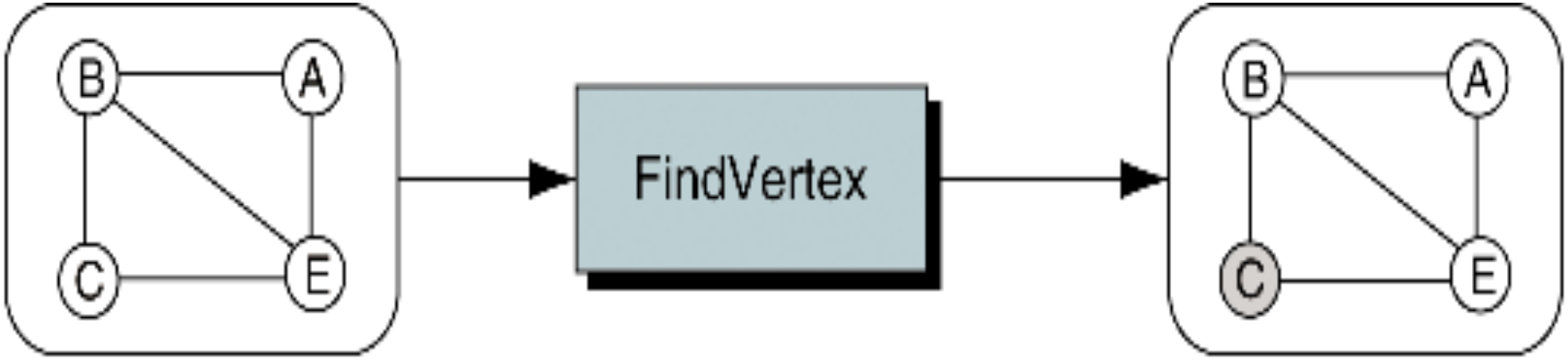
Insert and Delete Vertex



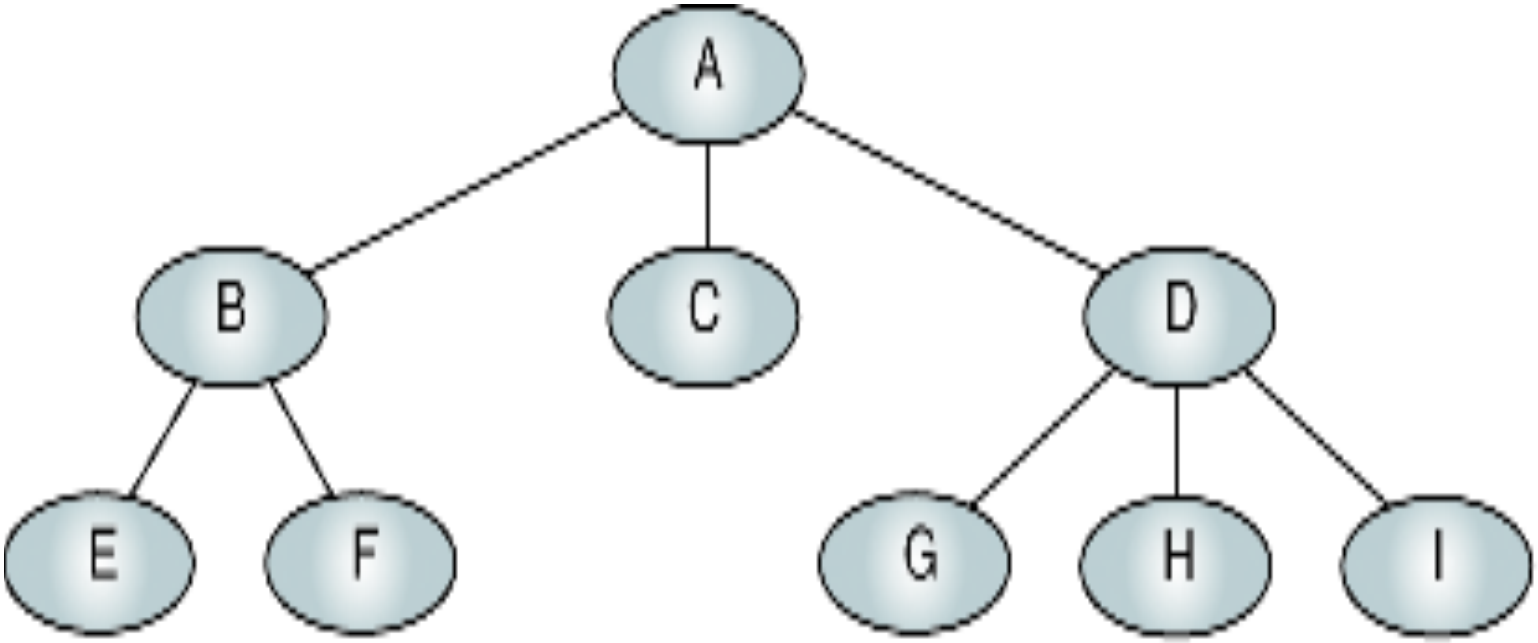
Add and Delete Edge



Find Vertex

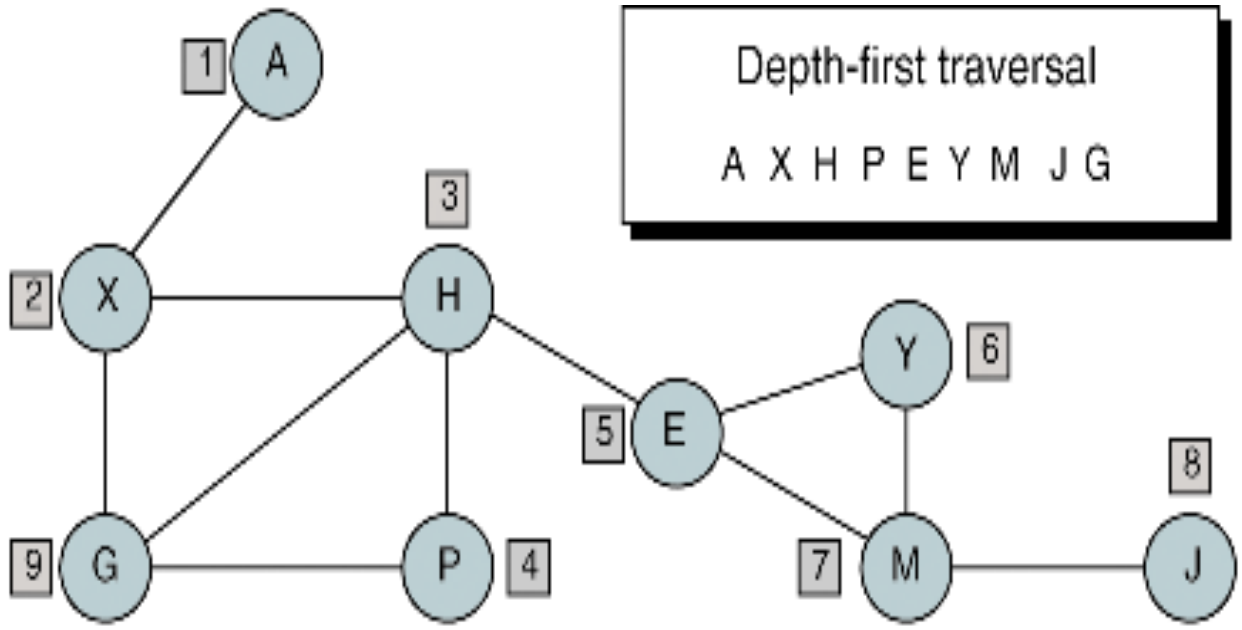


Depth-first Traversal of Tree

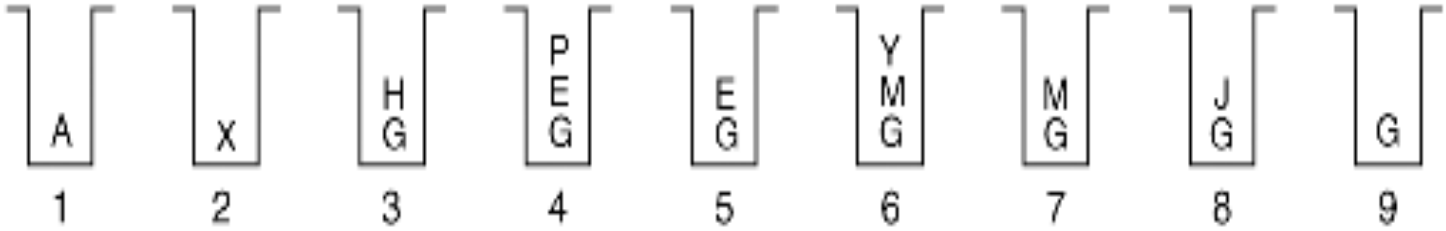


Depth-first traversal: A B E F C D G H I

Depth-first Traversal of a Graph

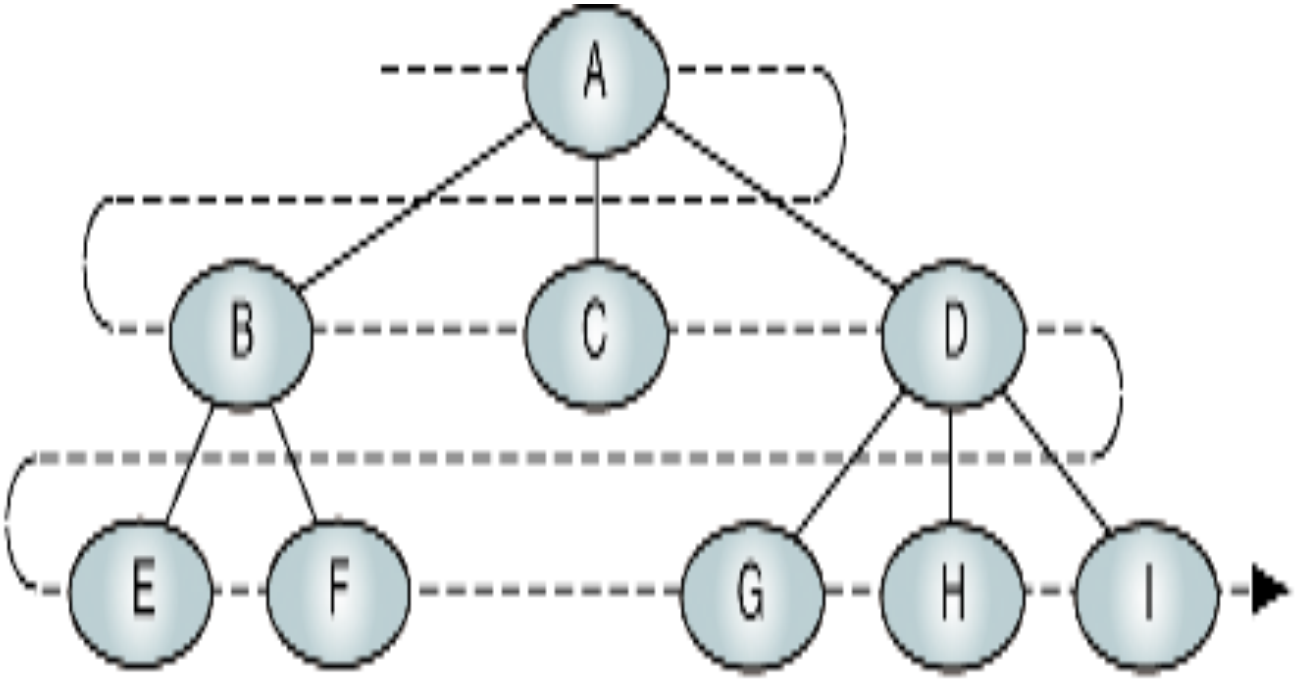


(a) Graph



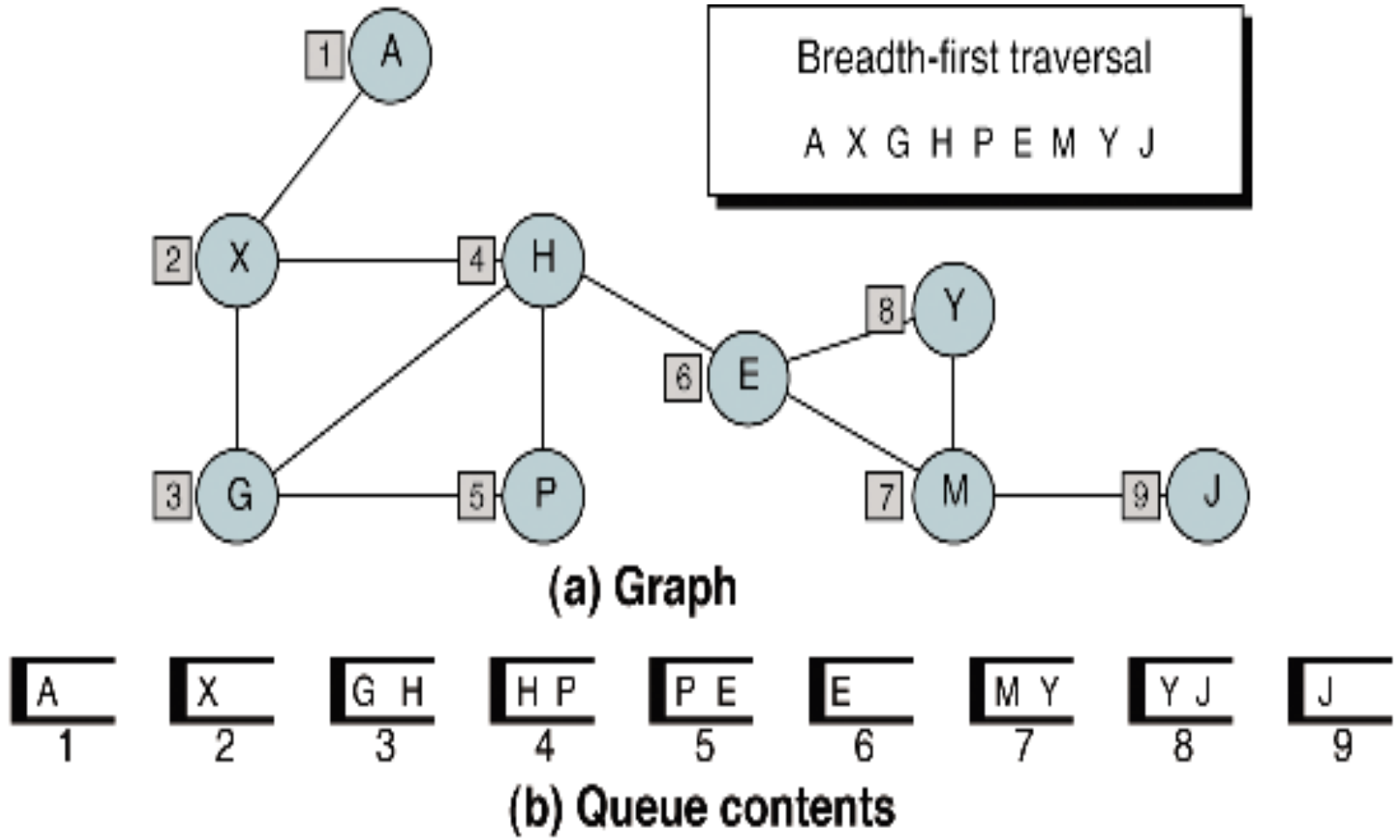
(b) Stack contents

Breadth-first Traversal of a Tree



Breadth-first traversal: A B C D E F G H I

Breadth-first Traversal of a Graph

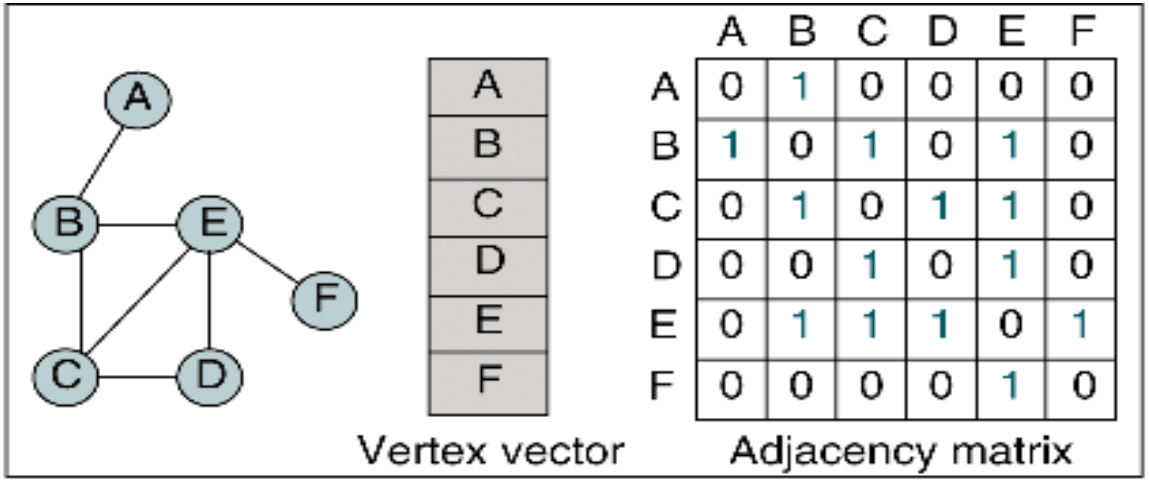


11-3 Graph Storage Structures

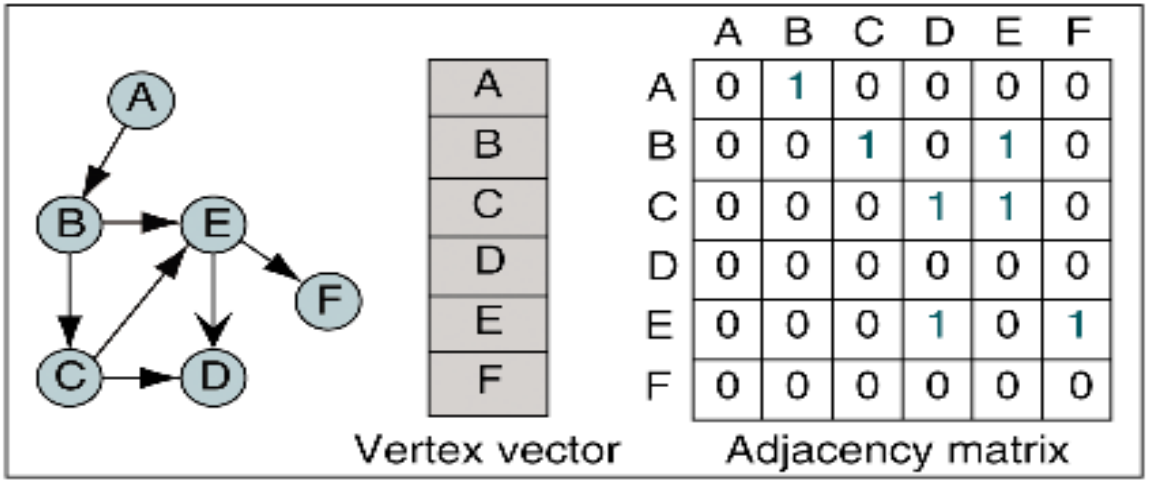
To represent a graph, we need to store two sets. The first set represents the vertices of the graph, and the second set represents the edges or arcs. The two most common structures used to store these sets are arrays and linked lists.

- Adjacency Matrix
- Adjacency List

Adjacency Matrix

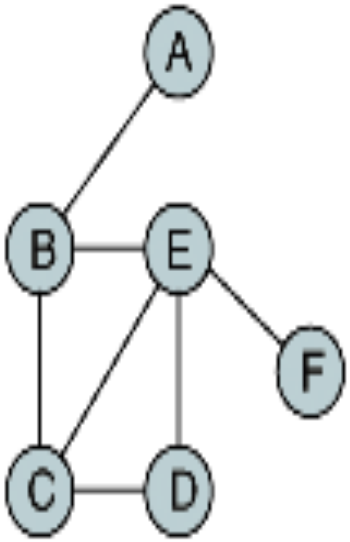


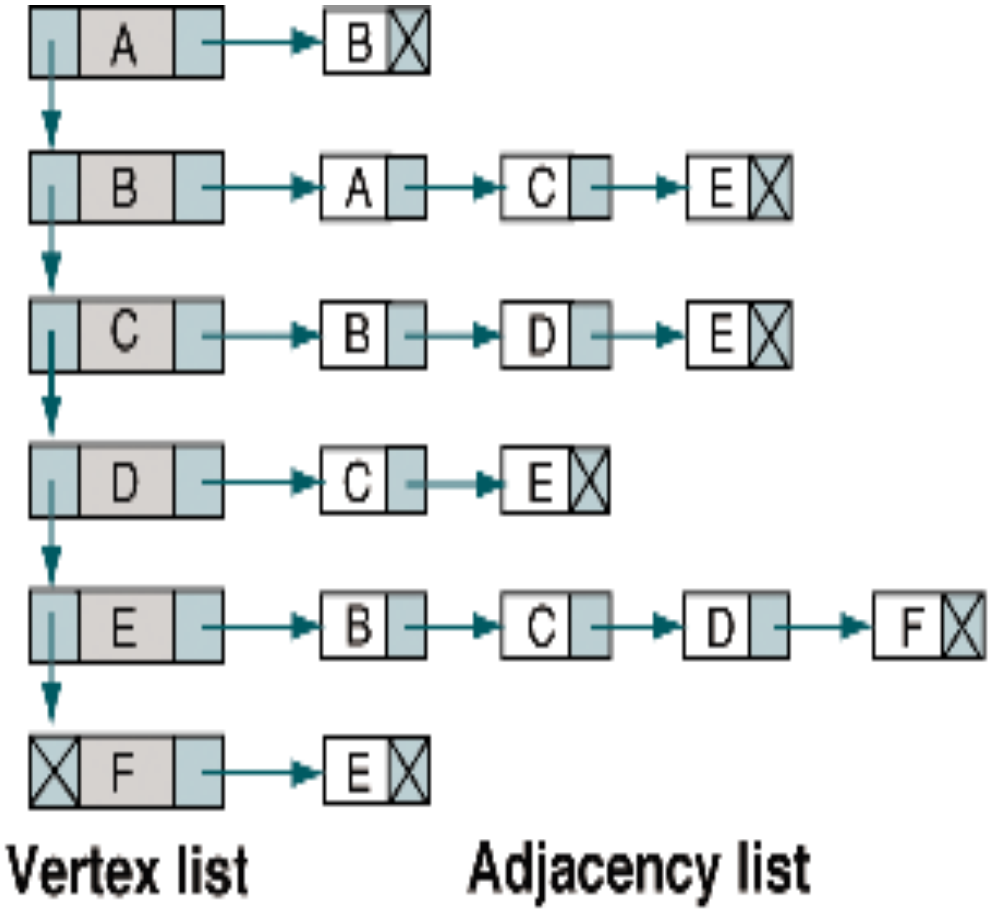
(a) Adjacency matrix for nondirected graph



(b) Adjacency matrix for directed graph

Adjacency List



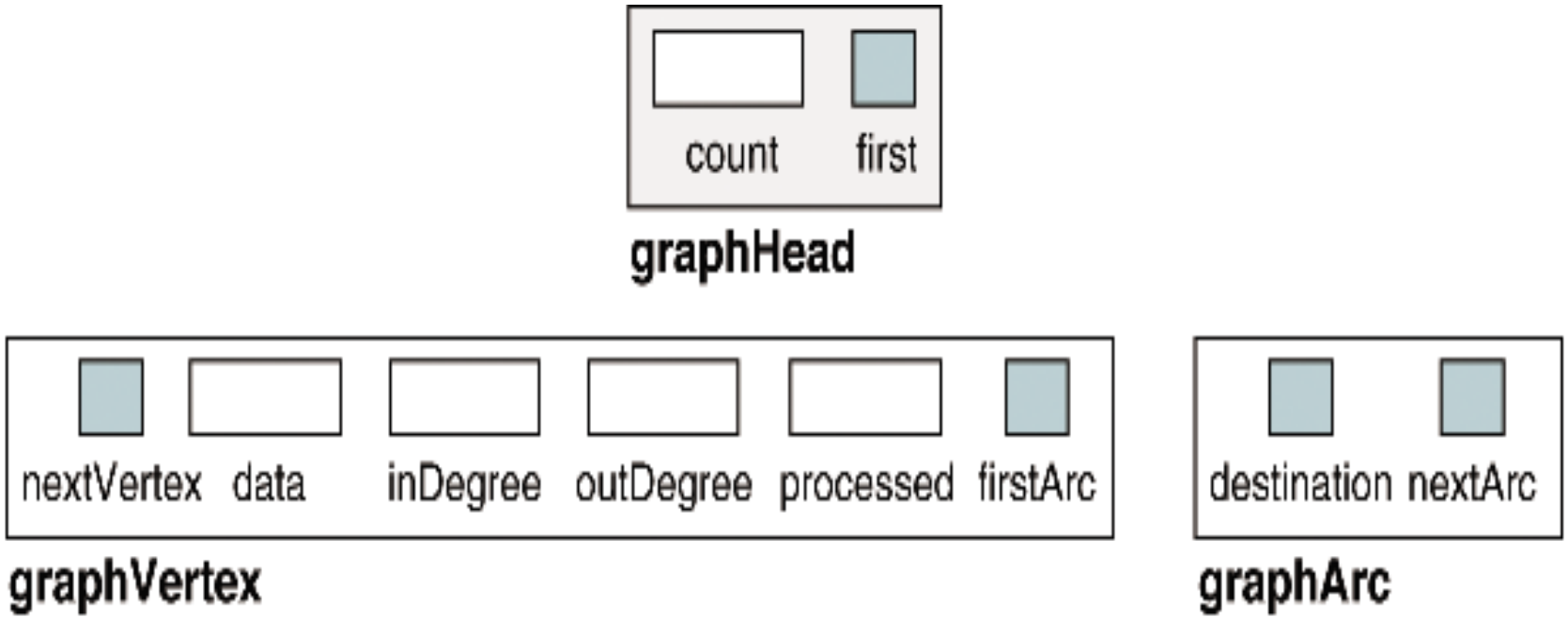


11-4 Graph Algorithms

In this section we develop a minimum set of algorithms that are needed to create and maintain a directed graph.

- Create Graph
- Insert Vertex
- Delete Vertex
- Insert Arc
- Delete Arc
- Retrieve Vertex
- Depth-first Traversal
- Breadth-first Traversal
- Destroy Graph

Graph Data Structure



Create Graph

```
Algorithm createGraph
Initializes the metadata elements of a graph structure.
      Pre  graph is a reference to metadata structure
      Post metadata elements have been initialized
1  set count to 0
2  set first to null
3  return graph head
end createGraph
```

Insert Vertex

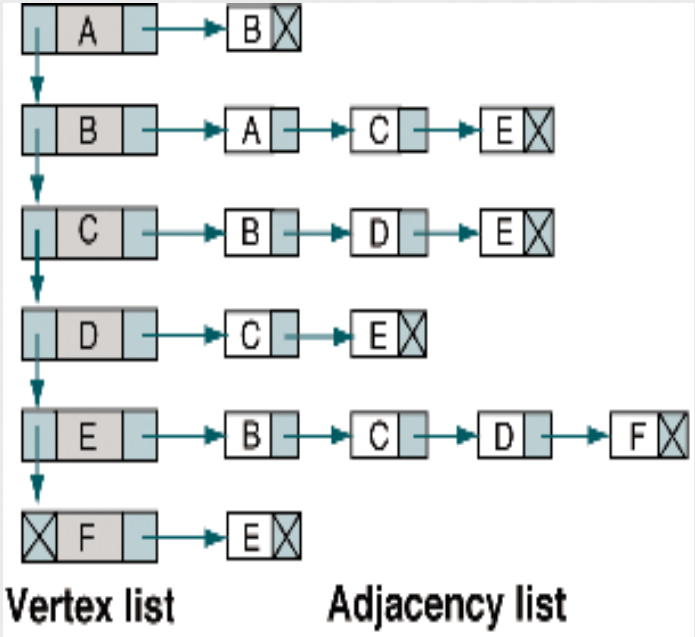
```
Algorithm insertVertex (graph, dataIn)
Allocates memory for a new vertex and copies the data to it.
   Pre    graph is a reference to graph head structure
          dataIn contains data to be inserted into vertex
   Post   new vertex allocated and data copied
 1 allocate memory for new vertex
 2 store dataIn in new vertex
 3 initialize metadata elements in newNode
 4 increment graph count
   Now find insertion point
 5 if (empty graph)
   1  set graph first to newNode
 6 else
   1  search for insertion point
   2  if (inserting before first vertex)
      1  set graph first to new vertex
   3  else
      1  insert new vertex in sequence
 7 end if
end insertVertex
```

Delete Vertex

```
Algorithm deleteVertex (graph, key)
Deletes an existing vertex only if its degree is 0.
   Pre    graph is reference pointer to graph head
          key is the key of the vertex to be deleted
   Post   vertex deleted (if degree zero)
   Return +1 if successful
          -1 if degree not zero
          -2 if key not found
 1 if (empty graph)
   1   return -2
 2 end if
   Locate vertex to be deleted
 3 search for vertex to be deleted
 4 if (not found)
   1   return -2
 5 end if
   Found vertex to be deleted. Test degree.
 6 if (vertex inDegree > 0 OR outDegree > 0)
   1   return -1
 7 end if
   Okay to delete vertex
 8 delete vertex
 9 decrement graph count
10 return 1
end deleteVertex
```

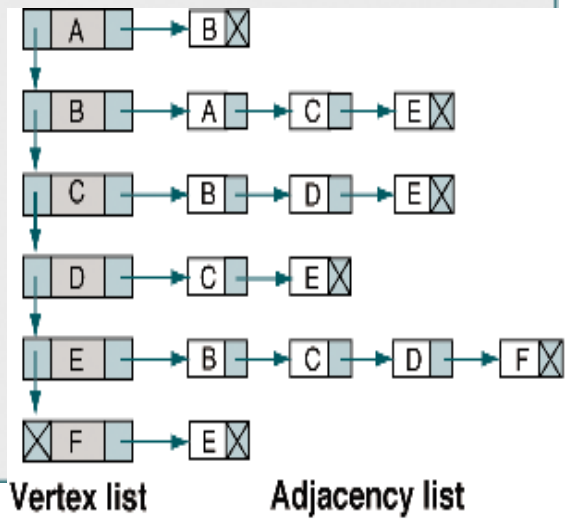
Insert Arc

```
Algorithm insertArc (graph, fromKey, toKey)
Adds an arc between two vertices.
   Pre    graph is reference to graph head structure
          fromKey is the key of the originating vertex
          toKey is the key of the destination vertex
   Post   arc added to adjacency list
   Return +1 if successful
          -2 if fromKey not found
          -3 if toKey not found
 1 allocate memory for new arc
   Locate source vertex
 2 search and set fromVertex
 3 if (from vertex not found)
   1   return -2
 4 end if
   Now locate to vertex
 5 search and set toVertex
 6 if (to vertex not found)
   1   return -3
```



Insert Arc (cont.)

```
 7 end if
   From and to vertices located. Insert new arc.
 8 increment fromVertex outDegree
 9 increment toVertex   inDegree
10 set arc destination to toVertex
11 if (fromVertex arc list empty)
       Inserting first arc
   1   set fromVertex firstArc to new arc
   2   set new arc nextArc to null
   3   return 1
12 end if
   Find insertion point in adjacency (arc) list
13 find insertion point in arc list
14 if (insert at beginning of arc list)
       Insertion before first arc
   1   set fromVertix firstArc to new arc
15 else
   1   insert in arc list
16 end if
17 return 1
end insertArc
```



Delete Arc

```
Algorithm  deleteArc (graph, fromKey, toKey)
Deletes an arc between two vertices.
   Pre    graph is reference to a graph head structure
          fromKey is the key of the originating vertex
          toKey is the key of the destination vertex
   Post   vertex deleted
   Return +1 if successful
          -2 if fromKey not found
          -3 if toKey not found
 1 if (empty graph)
   1   return -2
 2 end if
   Locate source vertex
 3 search and set fromVertex to vertex with key equal to
      fromKey
 4 if (fromVertex not found)
   1   return -2
 5 end if
   Locate destination vertex in adjacency list
```

Delete Arc (cont.)

```
 6 if (fromVertex arc list null)
   1   return -3
 7 end if
 8 search and find arc with key equal to toKey
 9 if (toKey not found)
       1  return -3
10 end if
   fromVertex, toVertex, and arc all located. Delete arc.
11 set toVertex to arc destination
12 delete arc
13 decrement fromVertex outDegree
14 decrement toVertex   inDegree
15 return 1
end deleteArc
```

Retrieve Vertex

```
Algorithm retrieveVertex (graph, key, dataOut)
Data contained in vertex identified by key passed to caller.
   Pre    graph is reference to a graph head structure
          key is the key of the vertex data
          dataOut is reference to data variable
   Post   vertex data copied to dataOut
   Return +1 if successful
          -2 if key not found
1  if (empty graph)
   1   return -2
2  end if
3  search for vertex
4  if (vertex found)
   1   move locnPtr data to dataOut
   2   return 1
5  else
   1   return -2
6  end if
end retrieveVertex
```

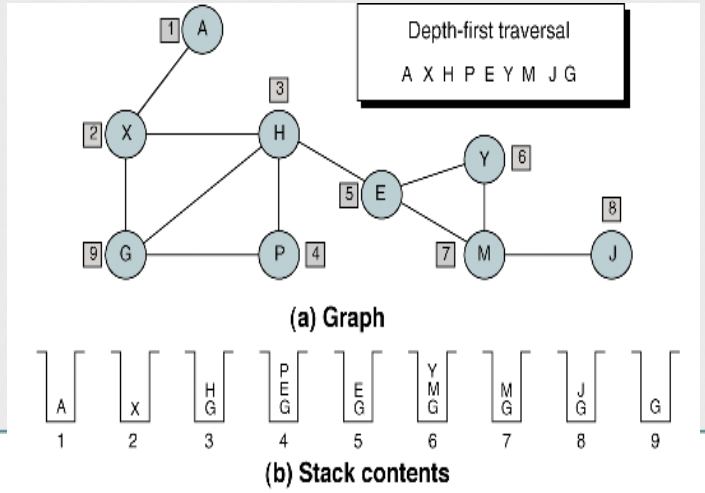
Depth-first Traversal

```
Algorithm depthFirst (graph)
Process the keys of the graph in depth-first order.
   Pre  graph is a pointer to a graph head structure
   Post vertices "processed"
1   if (empty graph)
   1   return
   Set processed flags to not processed
2  set walkPtr to graph first
3  loop (through all vertices)
   1   set processed to 0
4  end loop
   Process each vertex in list
5  createStack (stack)
```

Depth-first Traversal (cont.)

```
6  loop (through vertix list)
   1  if (vertix not processed and not in stack)
         Push and set flag to stack
      1  pushStack (stack, walkPtr)
      2  set walkPtr processed to 1
      3  end if
         Process vertex at stack top
      4  loop (not emptyStack(stack))
         1  set vertex to popStack(stack)
         2  process (vertix)
         3  set vertex to processed
            Push non-processed vertices from adjacency list
         4  loop (through arc list)
            1  if (arc destination not in stack)
               1  pushStack(stack, destination)
               2  set destination to in stack
            2  end if
            3  get next destination
         5  end loop
      5  end loop
   2  end if
   3  get next vertix
 7  end loop
 8  destroyStack(stack)
end depthFirst
```

?



Breadth-first Traversal

```
Algorithm breadthFirst (graph)
Processes the keys of the graph in breadth-first order.
   Pre  graph is pointer to graph head structure
   Post vertices processed
1  if (empty graph)
   1   return
2  end if
   First set all processed flags to not processed
3  createQueue (queue)
4  loop (through all vertices)
   1   set vertix to not processed
5  end loop
   Process each vertex in vertex list
6  loop (through all vertices)
   1   if (vertix not processed)
       1   if (vertix not in queue)
               Enqueue and set process flag to queued (1)
           1   enqueue (queue, walkPtr)
           2   set vertix to enqueued
       2   end if
           Now process descendents of vertex at queue front
       3   loop (not emptyQueue (queue))
           1   set vertex to dequeue (queue)
               Process vertex and flag as processed
```

Breadth-first Traversal (cont.)

```
        2  process (vertex)
        3  set vertex to processed
           Enqueue non-processed vertices from adjacency list
        4  loop (through adjacency list)
           1  if (destination not enqueued or processed)
              1  enqueue (queue, destination)
              2  set destination to enqueued
           2  end if
           3  get next destination
        5  end loop
     4  end loop
  2  end if
  3  get next vertex
7 end loop
8 destroyQueue (queue)
end breadthFirst
```

Destroy Graph

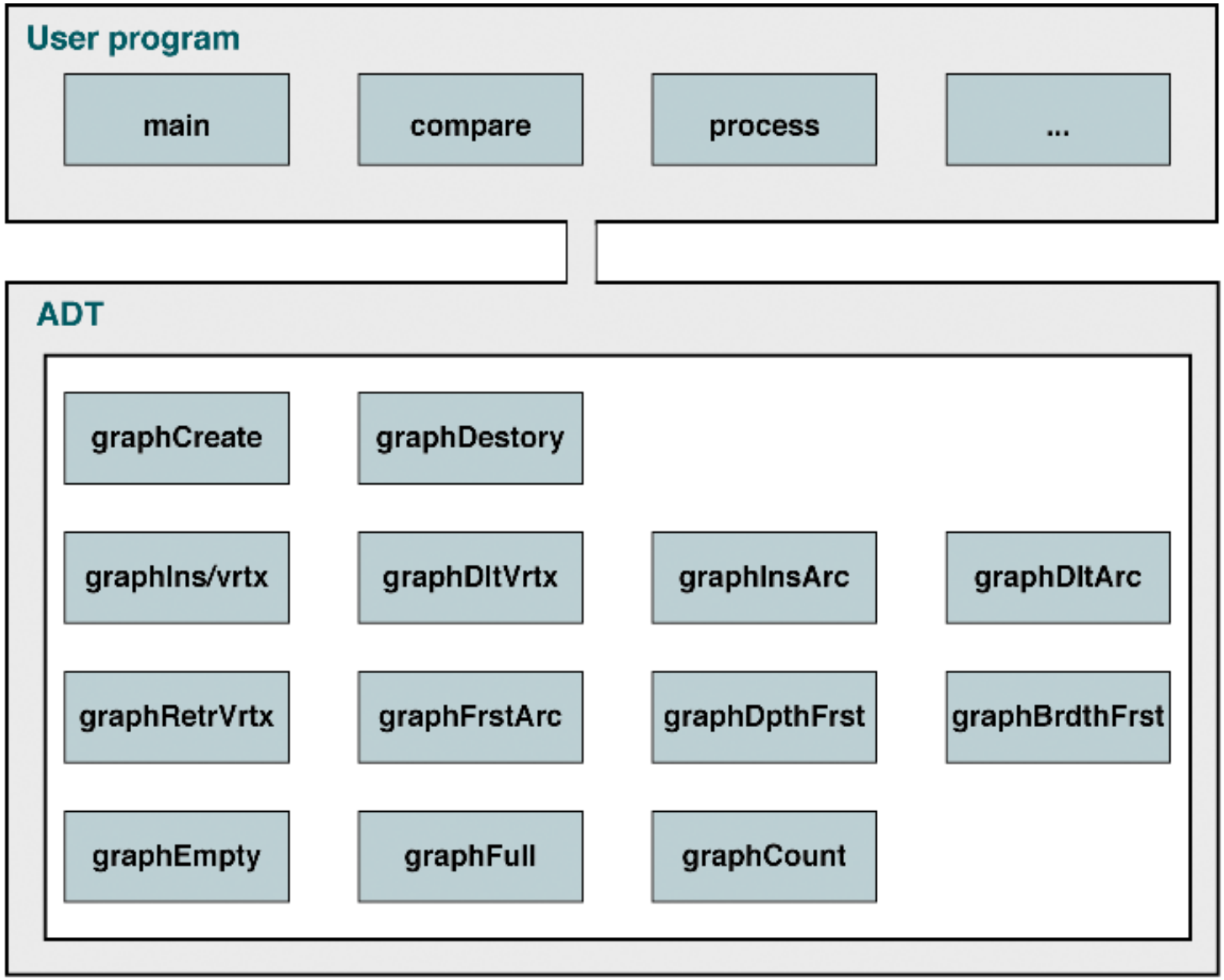
```
Algorithm destroyGraph (graph)
Traverses graph deleting all vertices and arcs.
   Pre  Nothing
   Post All vertices and arcs deleted
1  if (empty graph)
   1   return
2  loop (more vertices)
   1   loop (vertex outDegree > 0)
       1   delete vertex firstArc
       2   subtract 1 from vertex outDegree
   2   end loop
3  end loop
end destroyGraph
```

11-5 Graph ADT

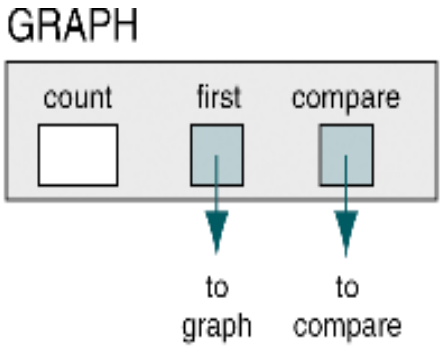
We begin by developing the code for the three graph structures. We then develop the code for the ADT functions.

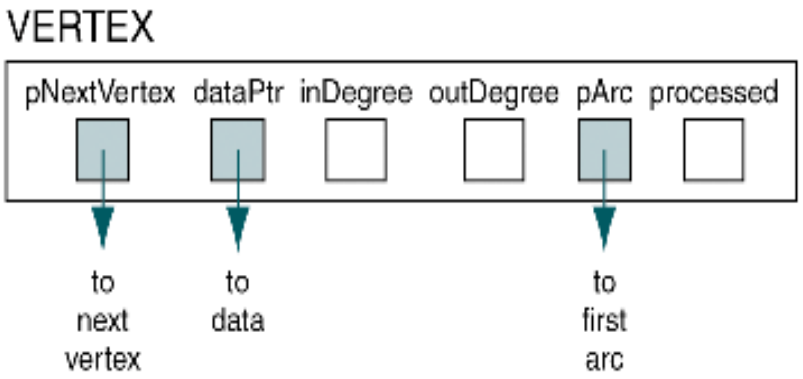
- **Data Structure**
- **Functions**

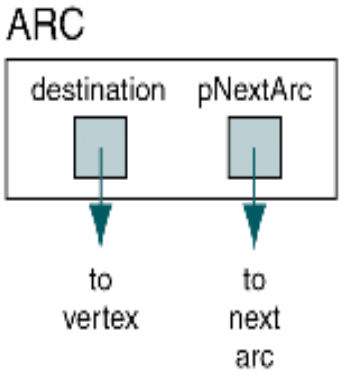
Graph ADT Design



Graph Data Structure








```
typedef struct
   {
    int             count;
    struct vertex* first;
    int (*compare)
          (void* argu1,
           void* argu2);
   } GRAPH;

typedef struct vertex
 {
  struct vertex*  pNextVertex;
  void*           dataPtr;
  int             inDegree;
  int             outDegree;
  short           processed;
  struct arc*     pArc;
 } VERTEX;

typedef struct arc
 {
  struct vertex*  destination;
  struct arc*     pNextArc;
 } ARC;
```

Graph Declarations

```
1  /* ==================== graphs.h ====================
2     This header file contains the declarations and
3     prototype functions for graphs as developed in
4     this text.
5        Written by:
6        Date:
7  */
8  #include "queueADT.h"
9  #include "stackADT.h"
10 #include "stdbool.h"
11
```

Graph Declarations (cont.)

```
12 // ==================== STRUCTURES ==================
13 typedef struct
14    {
15     int             count;
16     struct vertex* first;
17     int (*compare) (void* argu1, void* argu2);
18    } GRAPH;
19
20 typedef struct vertex
21    {
22     struct vertex*  pNextVertex;
23     void*           dataPtr;
24     int             inDegree;
25     int             outDegree;
26     short           processed;
27     struct arc*     pArc;
28    } VERTEX;
```

Graph Declarations (cont.)

```
29
30  typedef struct arc
31     {
32      struct vertex*  destination;
33      struct arc*     pNextArc;
34     } ARC;
35
36  // ============ Prototype Declarations =============
37
38  GRAPH* graphCreate
39             (int  (*compare) (void* argu1, void* argu2));
40  GRAPH* graphDestroy (GRAPH* graph);
41
```

Graph Declarations (cont.)

```
42  void graphInsVrtx   (GRAPH* graph, void*  dataInPtr);
43  int graphDltVrtx    (GRAPH* graph, void*  dltKey);
44  int graphInsArc     (GRAPH* graph, void*  pFromKey,
45                                     void*  pToKey);
46  int graphDltArc     (GRAPH* graph, void*  pFromKey,
47                                     void*  pToKey);
48
49  int graphRetrVrtx   (GRAPH* graph, void*  pKey,
50                                     void** pDataOut);
51  int graphFrstArc    (GRAPH* graph, void*  pKey,
52                                     void** pDataOut);
53
54  void graphDpthFrst  (GRAPH* graph,
55                        void (*process) (void* dataPtr));
56  void graphBrdthFrst (GRAPH* graph,
57                        void (*process) (void* dataPtr));
58
59  bool graphEmpty (GRAPH* graph);
60  bool graphFull  (GRAPH* graph);
61  int  graphCount (GRAPH* graph);
```

Graph Insert Vertex

```
1  /*  ================ graphInsVrtx =================
2     This function inserts new data into the graph.
3        Pre  graph is pointer to valid graph structure
4        Post data inserted or abort if memory O/F
5  */
6  void graphInsVrtx (GRAPH* graph, void* dataInPtr)
7  {
8  // Local Definitions
9     VERTEX* newPtr;
10    VERTEX* locPtr;
11    VERTEX* predPtr;
12
13 // Statements
14       newPtr = (VERTEX*)malloc(sizeof (VERTEX));
15       if (newPtr)
16           {
17            newPtr->pNextVertex   = NULL;
18            newPtr->dataPtr       = dataInPtr;
19            newPtr->inDegree      = 0;
20            newPtr->outDegree     = 0;
21            newPtr->processed     = 0;
22            newPtr->pArc          = NULL;
23            (graph->count)++;
24          } // if malloc
```

Graph Insert Vertex (cont.)

```
25        else
26            printf("Overflow error 100\a\n"),
27                exit (100);
28
29    // Now find insertion point
30    locPtr = graph->first;
31    if (!locPtr)
32         // Empty graph. Insert at beginning
33         graph->first = newPtr;
34    else
35        {
36         predPtr = NULL;
37         while (locPtr && (graph->compare
38                           (dataInPtr, locPtr->dataPtr) > 0))
39             {
40              predPtr = locPtr;
41              locPtr  = locPtr->pNextVertex;
42             } // while
43         if (!predPtr)
44              // Insert before first vertex
45              graph->first = newPtr;
46         else
47              predPtr->pNextVertex = newPtr;
48         newPtr->pNextVertex = locPtr;
49        } // else
50    return;
51  }   // graphInsVrtx
```

Graph Delete Vertex

```
 1  /* ================== graphDltVrtx ================
 2     Deletes existing vertex only if its degree is zero.
 3        Pre    graph is pointer to graph head structure
 4               dltKey is key of vertex to be deleted
 5        Post   Vertex deleted if degree zero
 6               -or- An error code is returned
 7        Return Success +1 if successful
 8                       -1 if degree not zero
 9                       -2 if dltKey not found
10  */
11  int graphDltVrtx (GRAPH* graph, void* dltKey)
12  {
13  // Local Definitions
14     VERTEX* predPtr;
15     VERTEX* walkPtr;
```

Graph Delete Vertex (cont.)

```
16
17  // Statements
18      if (!graph->first)
19          return -2;
20
21      // Locate vertex to be deleted
22      predPtr = NULL;
23      walkPtr = graph->first;
24      while (walkPtr
25        && (graph->compare(dltKey, walkPtr->dataPtr) > 0))
26          {
27          predPtr = walkPtr;
28          walkPtr = walkPtr->pNextVertex;
29          } // walkPtr &&
30      if (!walkPtr
31        || graph->compare(dltKey, walkPtr->dataPtr) != 0)
32          return -2;
33
34      // Found vertex. Test degree
35      if ((walkPtr->inDegree > 0)
36            || (walkPtr->outDegree > 0))
37          return -1;
38
39      // OK to delete
40      if (!predPtr)
41          graph->first = walkPtr->pNextVertex;
42      else
43        predPtr->pNextVertex = walkPtr->pNextVertex;
44    --graph->count;
45    free(walkPtr);
46    return 1;
47  } // graphDltVrtx
```

Graph Insert Arc

```
1  /* ================== graphInsArc =================
2     Adds an arc vertex between two vertices.
3        Pre    graph is a pointer to a graph
4               fromKey is pointer to start vertex key
5               toKey   is pointer to dest'n vertex key
6        Post   Arc added to adjacency list
7        Return success  +1 if successful
8                        -1 if memory overflow
9                        -2 if fromKey not found
10                       -3 if toKey not found
11 */
12 int graphInsArc (GRAPH* graph, void* pFromKey,
13                  void*  pToKey)
14 {
```

Graph Insert Arc (cont.)

```
15 // Local Definitions
16    ARC*    newPtr;
17    ARC*    arcPredPtr;
18    ARC*    arcWalkPtr;
19    VERTEX* vertFromPtr;
20    VERTEX* vertToPtr;
21
22 // Statements
23    newPtr = (ARC*)malloc(sizeof(ARC));
24    if (!newPtr)
25       return (-1);
26
27    // Locate source vertex
28    vertFromPtr = graph->first;
29    while (vertFromPtr && (graph->compare(pFromKey,
30                                vertFromPtr->dataPtr) > 0))
```

Graph Insert Arc (cont.)

```
31          {
32           vertFromPtr = vertFromPtr->pNextVertex;
33          } // while vertFromPtr &&
34       if (!vertFromPtr || (graph->compare(pFromKey,
35                                vertFromPtr->dataPtr) != 0))
36          return (-2);
37
38       // Now locate to vertex
39       vertToPtr   = graph->first;
40       while (vertToPtr
41           && graph->compare(pToKey, vertToPtr->dataPtr) > 0)
42          {
43           vertToPtr   = vertToPtr->pNextVertex;
44          } // while vertToPtr &&
45       if (!vertToPtr ||
46          (graph->compare(pToKey, vertToPtr->dataPtr) != 0))
47          return (-3);
48
```

Graph Insert Arc (cont.)

```
49      // From and to vertices located. Insert new arc
50      ++vertFromPtr->outDegree;
51      ++vertToPtr  ->inDegree;
52      newPtr->destination = vertToPtr;
53      if (!vertFromPtr->pArc)
54         {
55          // Inserting first arc for this vertex
56          vertFromPtr->pArc = newPtr;
57          newPtr-> pNextArc = NULL;
58          return 1;
59         } // if new arc
60
```

Graph Insert Arc (cont.)

```
61      // Find insertion point in adjacency (arc) list
62      arcPredPtr = NULL;
63      arcWalkPtr = vertFromPtr->pArc;
64      while (arcWalkPtr
65              && graph->compare(pToKey,
66                      arcWalkPtr->destination->dataPtr) >= 0)
67          {
68           arcPredPtr = arcWalkPtr;
69           arcWalkPtr = arcWalkPtr->pNextArc;
70          } // arcWalkPtr &&
71
72      if (!arcPredPtr)
73           // Insertion before first arc
74           vertFromPtr->pArc    = newPtr;
75      else
76           arcPredPtr->pNextArc = newPtr;
77      newPtr->pNextArc = arcWalkPtr;
78       return 1;
79  }   // graphInsArc
```

Graph Delete Arc

```
 1  /* ================== graphDltArc ==================
 2     Deletes an existing arc.
 3        Pre    graph is pointer to graph head structure
 4               fromKey is key of start vertex; toKey is
 5               toKey is key of dest'n of delete vertex
 6        Post   Arc deleted
 7        Return Success +1 if successful
 8                       -2 if fromKey not found
 9                       -3 if toKey not found
10  */
11  int graphDltArc (GRAPH* graph,
12                   void* fromKey, void* toKey)
13  {
14  // Local Definitions
15     VERTEX* fromVertexPtr;
16     VERTEX* toVertexPtr;
17     ARC*    preArcPtr;
18     ARC*    arcWalkPtr;
19
```

Graph Delete Arc (cont.)

```
20  // Statements
21      if (!graph->first)
22          return -2;
23
24      // Locate source vertex
25      fromVertexPtr = graph->first;
26      while (fromVertexPtr && (graph->compare(fromKey,
27                               fromVertexPtr->dataPtr) > 0))
28          fromVertexPtr = fromVertexPtr->pNextVertex;
29
30      if (!fromVertexPtr || graph->compare(fromKey,
31                               fromVertexPtr->dataPtr) != 0)
32         return -2;
33
34      // Locate destination vertex in adjacency list
35      if (!fromVertexPtr->pArc)
36          return -3;
37
38      preArcPtr = NULL;
39      arcWalkPtr = fromVertexPtr->pArc;
```

Graph Delete Arc (cont.)

```
40      while (arcWalkPtr && (graph->compare(toKey,
41                arcWalkPtr->destination->dataPtr) > 0))
42          {
43           preArcPtr  = arcWalkPtr;
44           arcWalkPtr = arcWalkPtr->pNextArc;
45          } // while arcWalkPtr &&
46      if (!arcWalkPtr || (graph->compare(toKey,
47                arcWalkPtr->destination->dataPtr) != 0))
48           return -3;
49      toVertexPtr = arcWalkPtr->destination;
50
51      // from, toVertex & arcPtr located. Delete arc
52      --fromVertexPtr->outDegree;
53      --toVertexPtr -> inDegree;
54      if (!preArcPtr)
55           // Deleting first arc
56           fromVertexPtr->pArc  = arcWalkPtr->pNextArc;
57      else
58           preArcPtr->pNextArc = arcWalkPtr->pNextArc;
59      free (arcWalkPtr);
60      return 1;
61  }  // graphDltArc
```

Graph Depth-first Traversal

```
1  /* ================== graphDpthFrst ================
2     Process data in graph in depth-first order.
3        Pre  graph is the a pointer to graph head
4        Post vertices "processed".
5
6        Processed Flag: 0 = not processed
7                        1 = pushed into stack
8                        2 = processed
9  */
10 void graphDpthFrst (GRAPH* graph,
11                     void (*process) (void* dataPtr))
12 {
```

Graph Depth-first Traversal (cont.)

```
13    // Local Definitions
14       bool    success;
15       VERTEX* walkPtr;
16       VERTEX* vertexPtr;
17       VERTEX* vertToPtr;
18       STACK * stack;
19       ARC*    arcWalkPtr;
20
21    // Statements
22       if (!graph->first)
23           return;
24
25       // Set processed flags to not processed
26       walkPtr = graph->first;
27       while (walkPtr)
28          {
29           walkPtr->processed = 0;
30           walkPtr            = walkPtr->pNextVertex;
31          } // while
```

Graph Depth-first Traversal (cont.)

```
32
33      // Process each vertex in list
34      stack = createStack ();
35      walkPtr = graph->first;
36      while (walkPtr)
37        {
38         if (walkPtr->processed < 2)
39           {
40            if (walkPtr->processed < 1)
41              {
42               // Push & set flag to pushed
43               success = pushStack (stack, walkPtr);
44               if (!success)
45                   printf("\aStack overflow 100\a\n"),
46                         exit (100);
47               walkPtr->processed = 1;
48              } // if processed < 1
49           } // if processed < 2
```

0 = not processed
1 = pushed into stack
2 = processed

?

Graph Depth-first Traversal (cont.)

```
50        // Process descendents of vertex at stack top
51        while (!emptyStack (stack))
52          {
53           vertexPtr = popStack(stack);
54           process (vertexPtr->dataPtr);
55           vertexPtr->processed = 2;
56
57           // Push all vertices from adjacency list
58           arcWalkPtr = vertexPtr->pArc;
59           while (arcWalkPtr)
60             {
```

Graph Breadth-first Traversal

```
1  /* ================== graphBrdthFrst ==================
2      Process the data of the graph in breadth-first order.
3         Pre  graph is pointer to graph head structure
4         Post graph has been processed
5         Processed Flag: 0 = not processed
6                         1 = enqueued
7                         2 = processed
8  */
9  void graphBrdthFrst (GRAPH* graph,
10                      void (*process) (void* dataPtr))
11 {
12 // Local Definitions
13    bool    success;
14    VERTEX* walkPtr;
15    VERTEX* vertexPtr;
16    VERTEX* vertToPtr;
17    QUEUE*  queue;
18    ARC*    arcWalkPtr;
19
```

Graph Breadth-first Traversal (cont.)

```
20  // Statements
21     if (!graph->first)
22         return;
23
24     // Set processed flags to not processed
25     walkPtr = graph->first;
26     while (walkPtr)
27        {
28         walkPtr->processed = 0;
29         walkPtr            = walkPtr->pNextVertex;
30        } // while
31
32     // Process each vertex in list
33     queue = createQueue ();
34     walkPtr = graph->first;
35     while (walkPtr)
36        {
37         if (walkPtr->processed < 2)
38            {
39             if (walkPtr->processed < 1)
40                {
```

Graph Breadth-first Traversal (cont.)

```
41                  // Enqueue & set flag to queue
42                  success = enqueue(queue, walkPtr);
43                  if (!success)
44                     printf("\aQueue overflow 100\a\n"),
45                     exit (100);
46                  walkPtr->processed = 1;
47                } // if processed < 1
48            } // if processed < 2
49          // Process descendents of vertex at que front
50          while (!emptyQueue (queue))
51              {
52               dequeue(queue, (void**)&vertexPtr);
53               process (vertexPtr->dataPtr);
54               vertexPtr->processed = 2;
55
56               // Enqueue vertices from adjacency list
57               arcWalkPtr = vertexPtr->pArc;
58               while (arcWalkPtr)
59                  {
60                   vertToPtr = arcWalkPtr->destination;
61                   if (vertToPtr->processed == 0)
62                     {
63                      success = enqueue(queue, vertToPtr);
64                      if (!success)
65                        printf("\aQueue overflow 101\a\n"),
66                           exit (101);
67                      vertToPtr->processed = 1;
```

Graph Breadth-first Traversal (cont.)

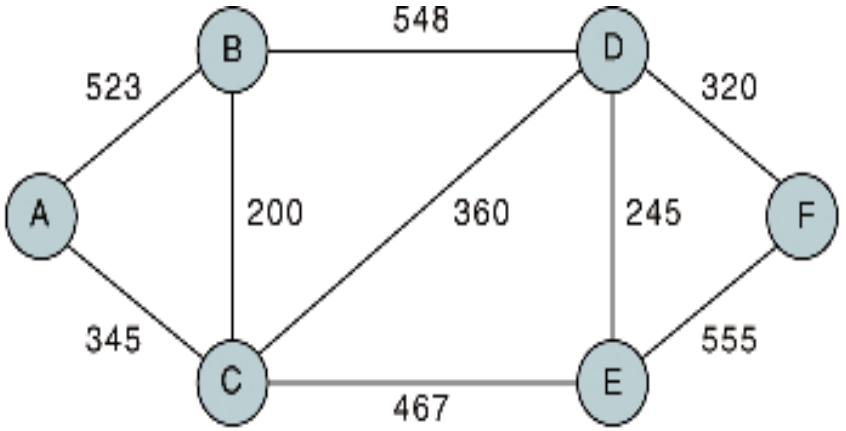
```
68                  } // if vertToPtr
69                arcWalkPtr = arcWalkPtr->pNextArc;
70              } // while pWalkArc
71            } // while !emptyQueue
72          walkPtr = walkPtr->pNextVertex;
73        } // while walkPtr
74    destroyQueue(queue);
75    return;
76  } // graphBrdthFrst
```

11-6 Networks

A network is a graph whose lines are weighted. It is also known as a weighted graph. Included in this section are two graph applications that process networks.

- **Minimum Spanning Tree**
- **Shortest Path Algorithm**

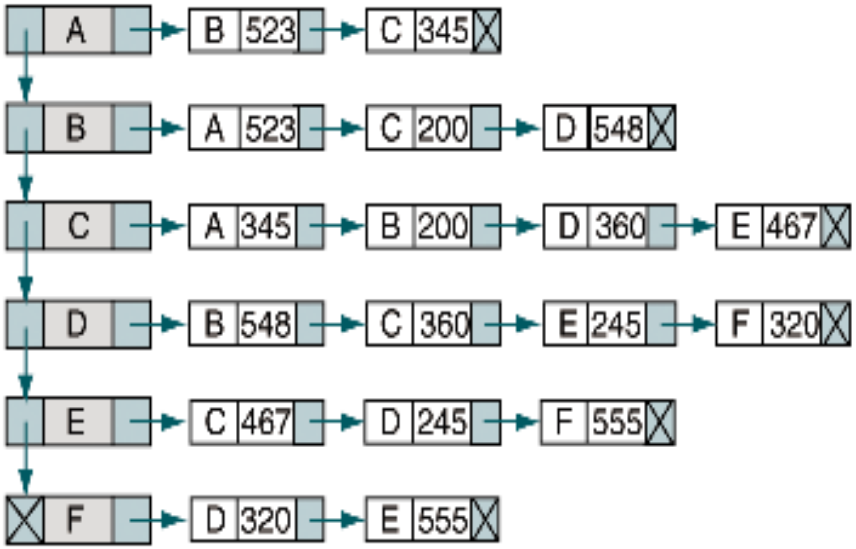
Storing Weights in Graph Structures



City Network

Vertex vector		A	B	C	D	E	F
A	A	0	523	345	0	0	0
B	B	523	0	200	548	0	0
C	C	345	200	0	360	467	0
D	D	0	548	360	0	245	320
E	E	0	0	467	245	0	555
F	F	0	0	0	320	555	0

Adjacency matrix

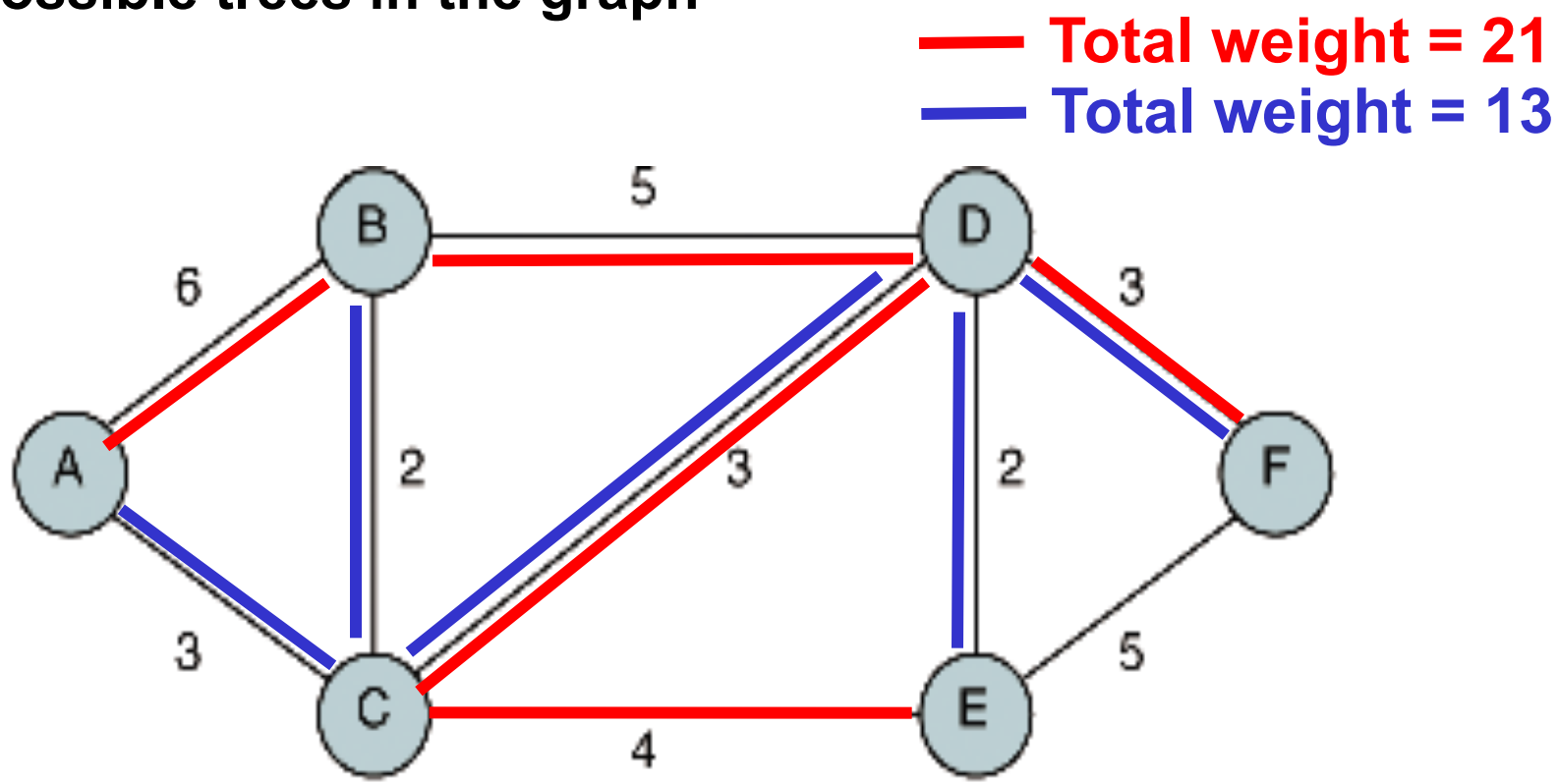


Vertex list

Adjacency list

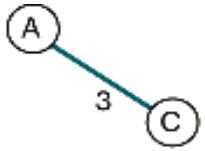
Spanning Tree of a Graph

- A spanning tree contains all of the vertices in a graph
- A minimum spanning tree is a spanning tree in which the total weight of the lines is quaranteed to be the minimum of all possible trees in the graph

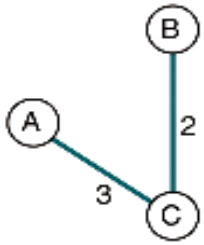


Develop Minimum Spanning Tree of Graph

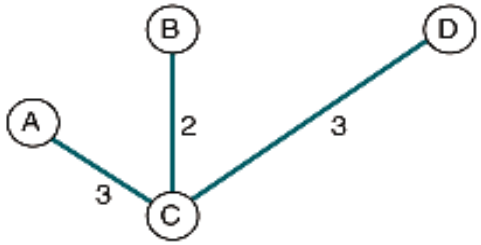
(a) Insert first vertex



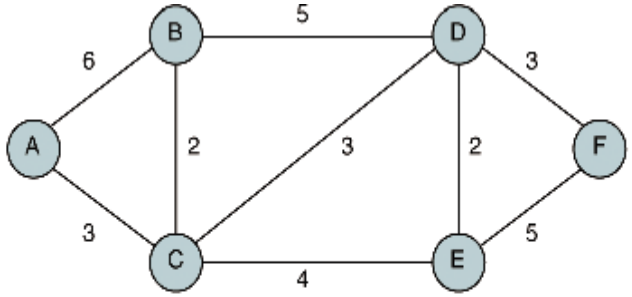
(b) Insert edge AC

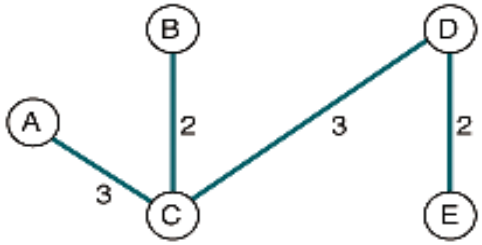


(c) Insert edge BC

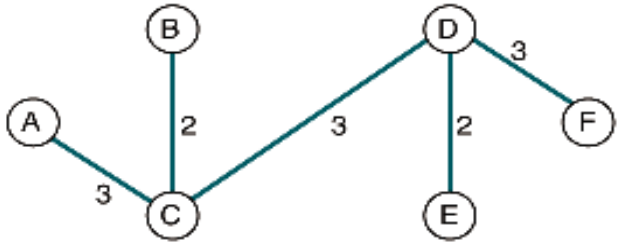


(d) Insert edge CD

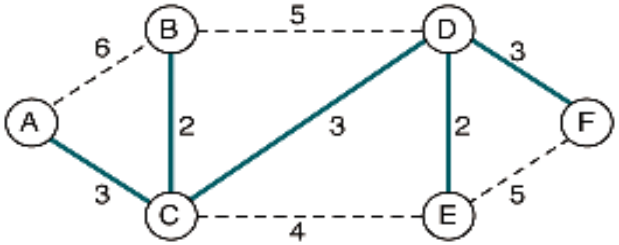




(e) Insert edge DE



(f) Insert edge DF



(g) Final tree in the graph

Data Structure for Spanning Tree

```
graphHead
    count
    first
end graphHead

graphVertex
    nextVertex
    data
    inDegree
    outDegree
    inTree
    firstEdge
end graphVertex

graphEdge
    destination
    weight
    nextEdge
    inTree
end graphEdge
```

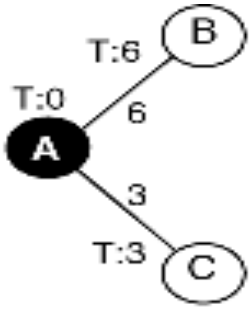
Minimum Spanning Tree of a Graph

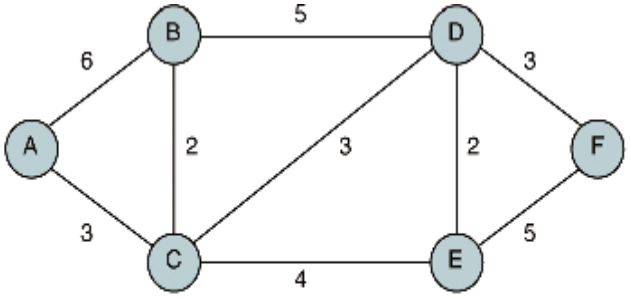
```
Algorithm spanningTree (graph)
Determine the minimum spanning tree of a network.
    Pre  graph contains a network
    Post spanning tree determined
1  if (empty graph)
   1   return
2  end if
3  loop (through all vertices)
       Set inTree flags false.
   1   set vertex inTree flag to false
   2   loop (through all edges)
       1   set edge inTree flag to false
       2   get next edge
   3   end loop
   4   get next vertex
4  end loop
   Now derive spanning tree.
5  set first vertex to in tree
6  set treeComplete to false
7  loop (not treeComplete)
   1   set treeComplete to true
   2   set minEdge to maximum integer
   3   set minEdgeLoc to null
   4   loop (through all vertices)
          Walk through graph checking vertices in tree.
```

Minimum Spanning Tree of a Graph (cont.)

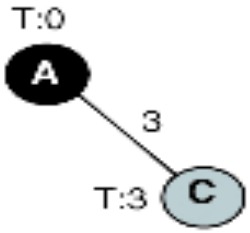
```
      1  if (vertex in tree AND vertex outDegree > 0)
         1  loop (through all edges)
            1  if (destination not in tree)
                    set destination inTree flag to false)
               1  set treeComplete to false
               2  if (edge weight < minEdge)
                  1  set minEdge to edge weight
                  2  set minEdgeLoc to edge
               3  end if
            2  end if
            3  get next edge
         2  end loop
      2  end if
      3  get next vertex
   5  end loop
   6  if (minEdgeLoc not null)
         Found edge to insert into tree.
      1  set minEdgeLoc  inTree flag to true
      2  set destination inTree flag to true
   7  end if
 8  end loop
end spanningTree
```

Determining Shortest Path (Dijkstra)

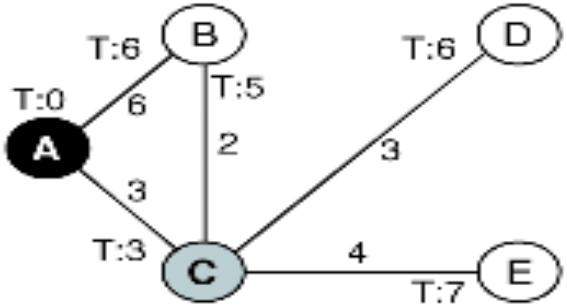




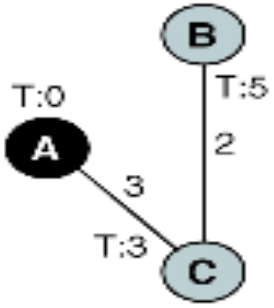
(a1) Possible paths from A1



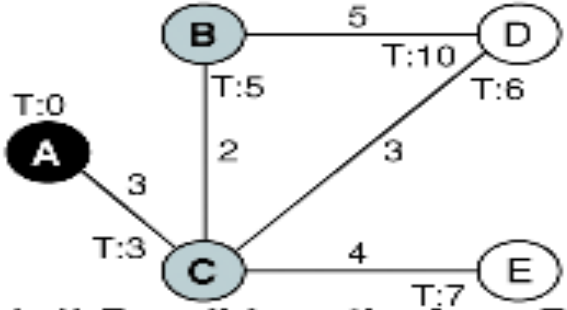
(a2) Tree after insertion of C



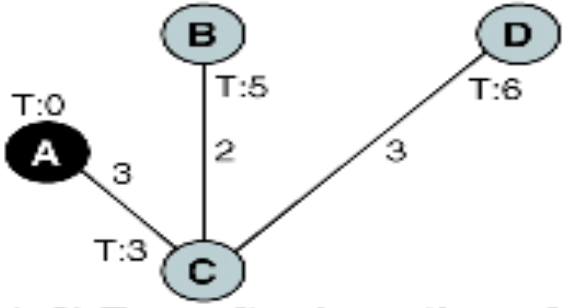
(b1) Possible paths from A and C



(b2) Tree after insertion of B

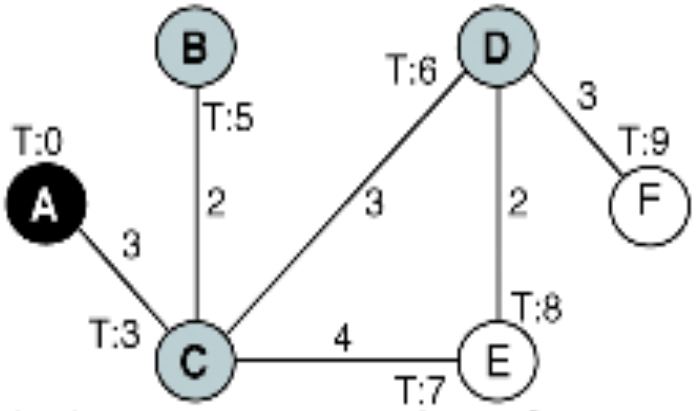


(c1) Possible paths from B and C

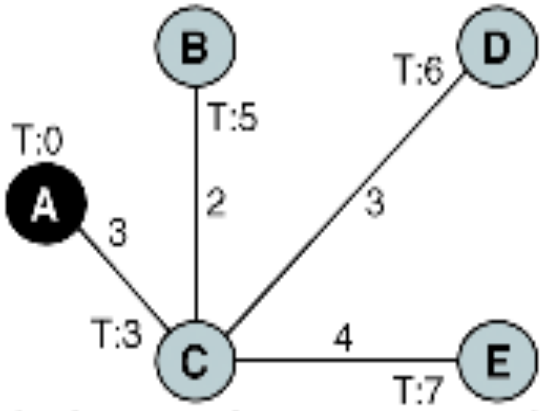


(c2) Tree after insertion of D

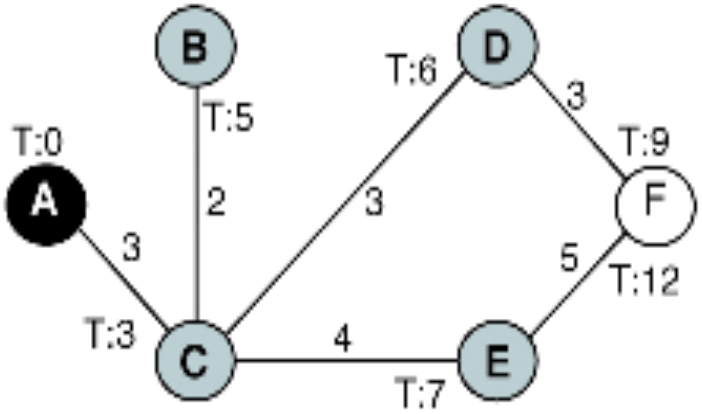
Determining Shortest Path (cont.)



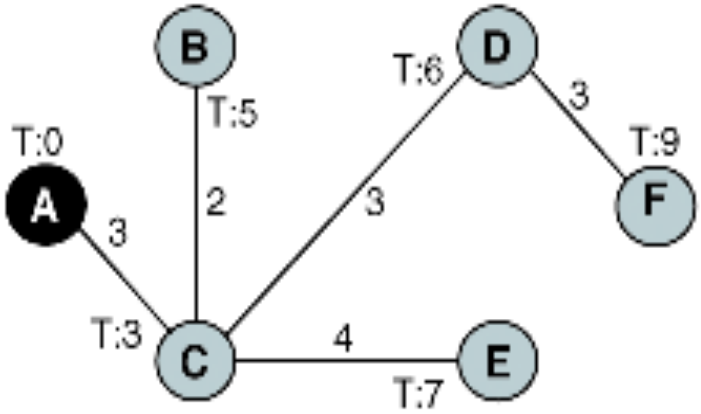
(d1) Possible paths from C and D



(d2) Tree after insertion of E



(e1) Possible paths from D and E



(e2) Tree after insertion of F

T:*n* Total path length from A to node

Shortest Path

```
Algorithm shortestPath (graph)
Determine shortest path from a network vertex to other
vertices.
   Pre    graph is pointer to network
   Post   minimum path tree determined
1  if (empty graph)
   1   return
2  end if
3  loop (through all vertices)
      Initialize inTree flags and path length.
   1   set vertex inTree flag to false
   2   set vertex pathLength to maximum integer
   3   loop (through all edges)
       1   set edge inTree flag to false
       2   get next edge
   4   end loop
   5   get next vertex
4  end loop
   Now derive minimum path tree.
5  set first vertex inTree flag to true
6  set vertex pathLength to 0
7  set treeComplete to false
```

Shortest Path (cont.)

```
8  loop (not treeComplete)
   1   set treeComplete to true
   2   set minEdgeLoc to null
   3   set pathLoc    to null
   4   set newPathLen to maximum integer
   5   loop (through all vertices)
          Walk through graph checking vertices in tree.
       1  if (vertex in tree AND outDegree > 0)
          1  set edgeLoc to firstEdge
          2  set minPath to pathLength
          3  set minEdge to maximum integer
          4  loop (through all edges)
                Locate smallest path from this vertex.
             1  if (destination not in tree)
                1  set treeComplete to false
                2  if (edge weight < minEdge)
                   1  set minEdge to edge weight
                   2  set minEdgeLoc to edgeLoc
                3  end if
             2  end if
             3  set edgeLoc to edgeLoc nextEdge
          5  end loop
             Test for shortest path.
          6  if (minPath + minEdge < newPathLen)
             1  set newPathLen to minPath + minEdge
             2  set pathLoc    to minEdgeLoc
          7  end if
       2  end if
       3  get next vertex
   6   end loop
```

Shortest Path (cont.)

```
   7  if (pathLoc not null)
         Found edge to insert into tree.
      1  set pathLoc                  inTree flag to true
      2  set pathLoc destination inTree flag to true
      3  set pathLoc destination pathLength to newPathLen
   8  end if
 9 end loop
end shortestPath
```