16. 그래프

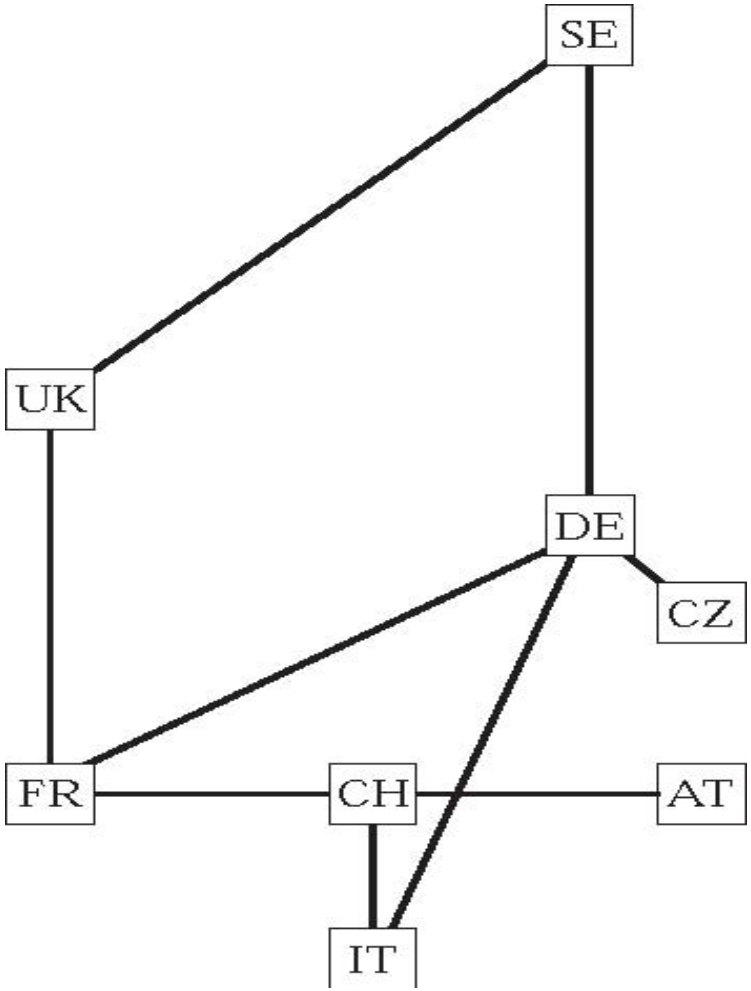
16.1 그래프

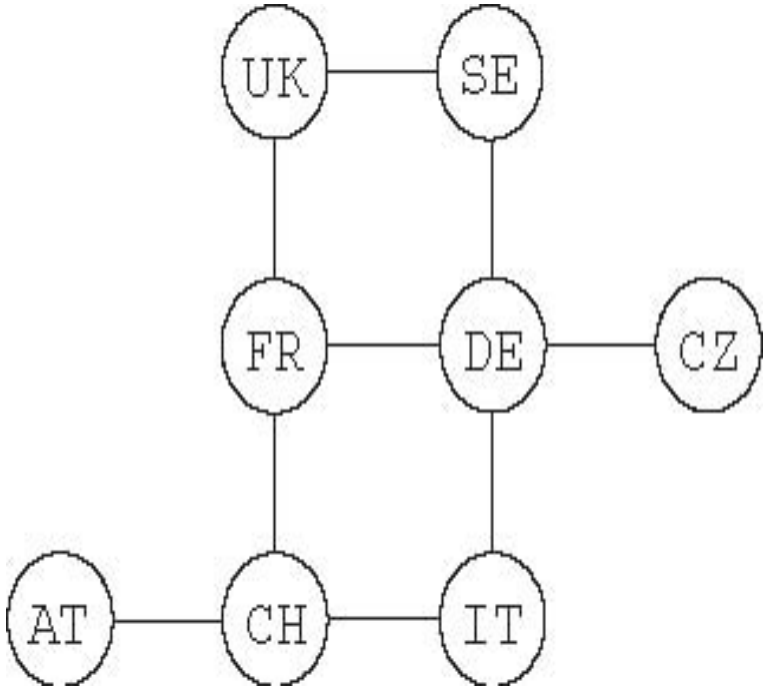
- 선형 자료 구조
  - 배열, 리스트, 스택, 큐를 사용하여 해결할 수 있다.

- 비선형 자료 구조
  - 트리, 이진 트리로 복잡한 문제를 해결한다.
  - *그래프(graph)* ; 인터넷과 같은 통신 네트워크나 고속도로나 철도와 같은 교통 네트워크에 적합하다.

- *그래프(graph)*
  - 링크에 의해 연결되어 있는 노드들로 구성된 구조이다.
    노드를 *정점(vertex)*이라고 부르고, 링크를 *간선(edge)*이라고 부른다.
  - 그래프는 V와 E가 집합이고, E의 모든 원소가 V의 두-원소 부분집합일 때 쌍 G=(V, E)로 표현된다. V의 원소를 정점이라고 부르고, E의 원소를 간선이라고 부른다.

유럽 인터넷 연결망과 동등한 그래프





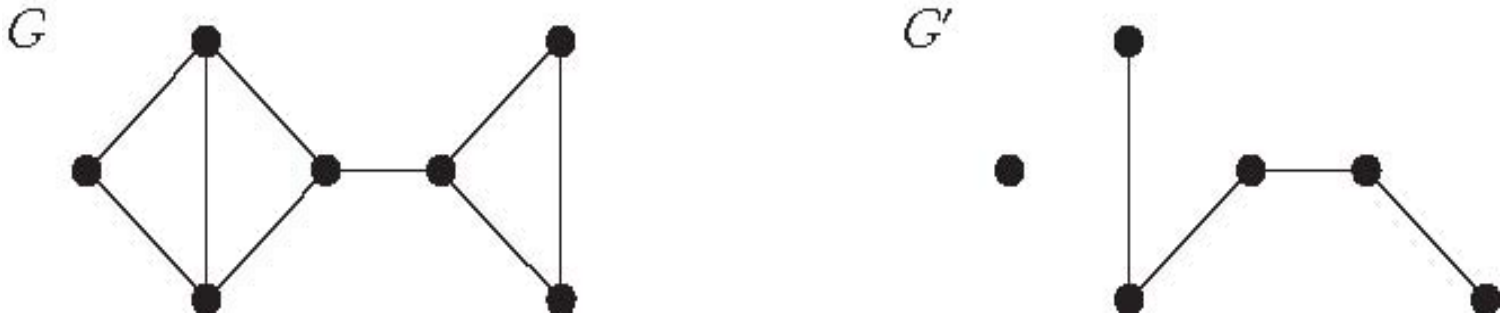
용어

- 그래프는 노드의 지리적인 위치가 중요하지 않기 때문에 *위상기하학(topology)*이라고도 한다.
- 정점의 개수를 그래프의 *크기(size)*라 한다.
- 두 정점을 연결하는 간선이 있다면 두 정점을 *인접(adjacent)*하다고 한다.
- 정점 v_0부터 v_k 까지의 *경로(path)*는 i=1,...,k에 대해 $(v_{i-1}, v_i) \in E$ 일 때, $(v_0, v_1, ..., v_k)$ 의 시퀀스이다.
- 경로의 *길이(length)*는 간선의 개수 k이다.
- 모든 정점이 서로 다른 것이라면 경로 p를 단순 경로(simple path) 라 한다.
- *사이클(cycle)*은 하나의 노드에서 다시 자기 자신으로 돌아오는 경로이다. *단순 사이클(simple cycle)*은 처음과 마지막 정점만 같은 사이클이다
- 사이클이 없는 그래프를 비순환(acyclic) 그래프라 한다.

용어(계속)

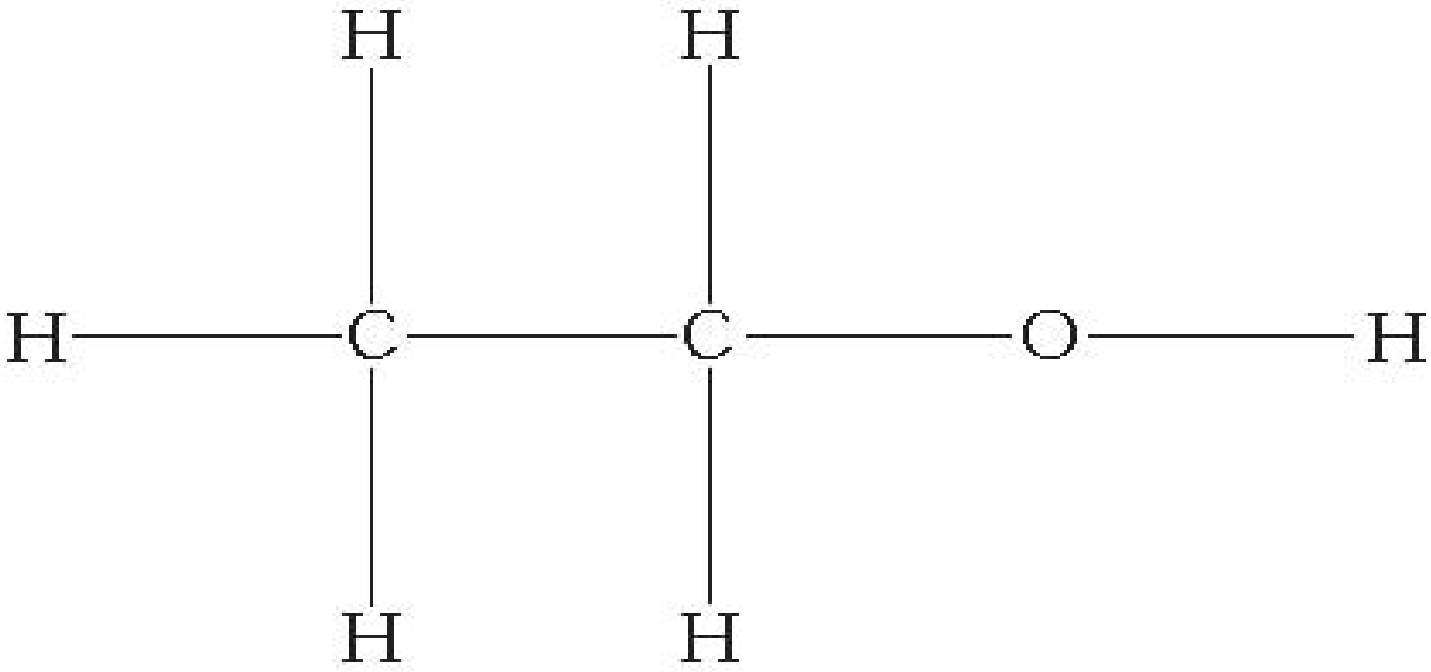
- G=(V, E)가 그래프이고, V′⊂V 이고 E′⊂E 일 때 G′=(V′, E′)은 G의 서브그래프(subgraph)이다.
- 어떤 정점이 다른 모든 정점으로부터 도달가능하다면, 이것을 연결 그래프라고 한다.
- 그래프 G의 *연결 요소(connected component)* G'은 최대 연결 서브그래프이다.
- 연결된 비순환 그래프를 *자유 트리(free tree)*라고 부른다.
  - 트리는 특별한 종류의 그래프로 보아야 한다.

서브그래프 G'을 가지고 있는 그래프 G



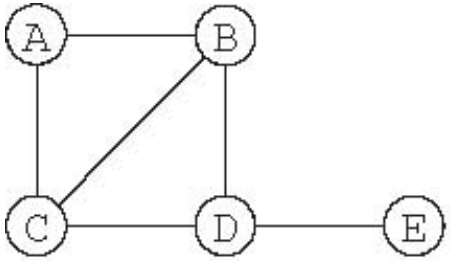
4개의 요소를 가지고 있는 그래프

에탄올을 표현하는 자유 트리



16.2 인접 행렬 구현

- 그래프를 저장하는 두 가지 일반적인 방법으로 인접 행렬 (adjacency matrix) 과 인접 리스트가 있다.
- 그래프를 위한 인접 행렬은 boolean 원소의 2차원 배열이다



	A	B	C	D	E
A	false	true	true	false	false
B	true	false	true	true	false
C	true	true	false	true	false
D	false	true	true	false	true
E	false	false	false	true	false

그래프와 인접 행렬

인접 행렬을 사용한 그래프의 저장

- LISTING 16.1: Storing a Graph with an Adjacency Matrix

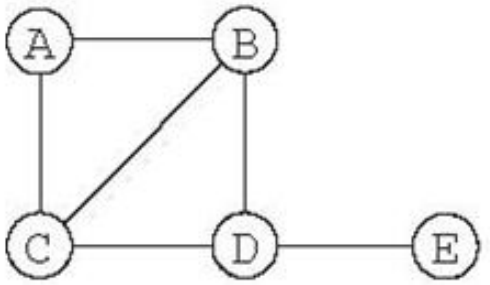
```
1 class Graph {
2       int size;
3       String[] vertices;
4       boolean[][] a; // adjacency matrix
5
6       public Graph(String[] args) {
7          size = args.length;
8          vertices = new String[size];
9          System.arraycopy(args, 0, vertices, 0, size);
10         a = new boolean[size][size];
11      }
13      public void add(String v, String w) {
14         int i = index(v), j = index(w);
15         a[i][j] = a[j][i] = true;
16      }
```

```
    private int index(String v) {
        for (int i = 0; i < size; i++)
            if (vertices[i].equals(v)) return i;
        return a.length;
    }
    public String toString() {
        if (size == 0) return "{ }";
        StringBuffer buf = new StringBuffer("{" + vertex(0));
        for (int i = 1; i < size; i++)
            buf.append("," + vertex(i));
        return buf + "}";
    }
    private String vertex(int i) {
        StringBuffer buf = new StringBuffer(vertices[i] + ":");
        for (int j = 0; j < size; j++)
            if ( a[i][j] ) buf.append(vertices[j]);
        return buf + "";
    }
}
```

Graph 클래스의 테스팅

- LISTING 16.2: Testing the Graph Class

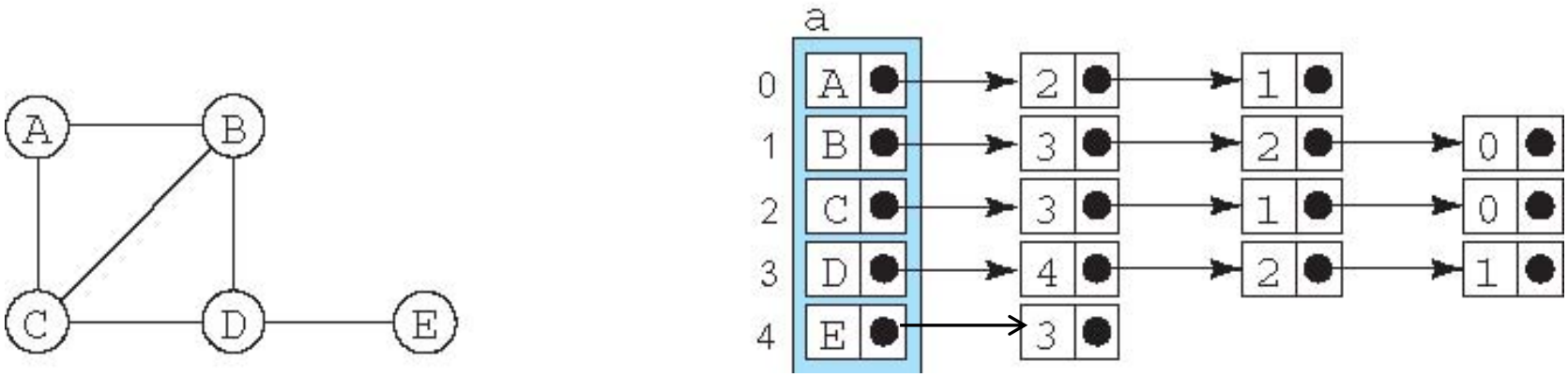
```
1 class TestGraph {
2     public static void main(String[] args) {
3         Graph g = new Graph(new String[]{"A", "B", "C", "D", "E"});
4         System.out.println(g);
5         g.add("A", "B");
6         g.add("A", "C");
7         g.add("B", "C");
8         g.add("B", "D");
9         g.add("C", "D");
10        g.add("D", "E");
11        System.out.println(g);
12    }
13 }
```



실행결과는?

16.3 인접 리스트 구현

- 그래프를 위한 인접 리스트는 각 정점 당 하나의 리스트를 가진 연결 리스트의 배열이다.



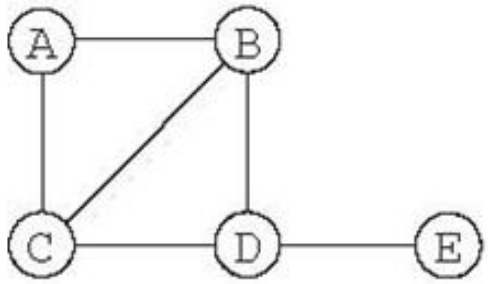
그래프와 인접 리스트

인접 리스트를 사용한 그래프의 저장

- Listing 16.3: Storing a Graph with an Adjacency List

```
1 class Graph {
2       int size;
3       List[] a; // adjacency list
5       public Graph(String[] args) {
6           size = args.length;
7           a = new List[size];
8           for (int i=0; i<size; i++)
9               a[i] = new List(args[i]);
10    }
12    public void add(String v, String w) {
13        a[index(v)].add(index(w));
14        a[index(w)].add(index(v));
15    }
```

```
17    public String toString() {
18        if (size == 0) return "{}";
19        StringBuffer buf = new StringBuffer("{" + a[0]);
20        for (int i = 1; i < size; i++)
21              buf.append("," + a[i]);
22        return buf + "}";
23    }
25    private int index(String v) {
26        for (int i = 0; i < size; i++)
27              if (a[i].vertex.equals(v)) return i;
28        return a.length;
29    }
```

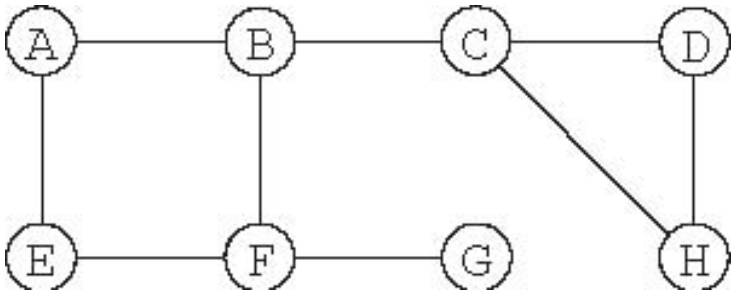


실행결과는?

```
31    private class List {
32        String vertex;      Node edges;
35        List(String vertex) {     this.vertex = vertex;          }
39        public void add(int j) {      edges = new Node(j, edges);    }
43        public String toString() {
44            StringBuffer buf = new StringBuffer(vertex);
45            if (edges != null) buf.append(":");
46            for (Node p = edges; p != null; p = p.next)
47                buf.append(Graph.this.a[p.to].vertex);
48            return buf + "";
49        }
51        private class Node {
52            int to;     Node next;
54             Node(int to, Node next) {
55                this.to = to;       this.next = next;
56             }
57        }
58    }
60 }
```

16.4 너비 우선 탐색

- 그래프에서 정점 A로부터 시작하여 레벨 순서 순회(알고리즘 11.1)을 적용시킨다고 하자.



- A에 인접한 두 개의 정점(B와 E)은 그래프가 루트를 A로 하는 트리라면 A의 자식이 된다. 따라서 레벨 순서 순회에서 첫 번째로 방문하는 세 개의 정점은 A, B, E이다.
- 다음으로 첫 번째 자식 B의 자식들을 방문한다. 그것들은 B와 인접해야 하는 동시에 B보다는 A에서 더 멀리 떨어져 있어야 하므로 C와 F가 되고, 이렇게 되면 정점 E는 자식이 없는 상태가 된다. 따라서 C와 F를 방문하고, C의 자식인 D와 H로 간 다음, 마지막으로 F의 자식인 G를 방문한다. 따라서 레벨 순서 순회는 A, B, E, C, F, D, H, G의 순서로 정점들을 방문한다.

16.4 너비 우선 탐색

- 위에 설명한 순회는 항상 지역적으로 동작하기 때문에 "갈망 알고리즘(greedy algorithm)"이라고 불린다.
- *너비 우선 탐색(breadth-first search: BFS)*
  - 그래프에 적용하기 위해 레벨 순서 순회 알고리즘을 일반화시킨 것

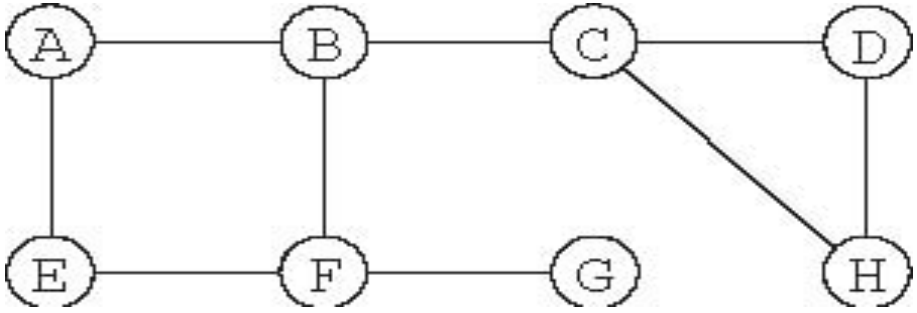
너비 우선 탐색(BFS) 알고리즘

- 너비 우선 탐색(BFS) 알고리즘
  - 입력 : 그래프 $G=(V,E)$와 최초 정점 $v \in V$.
  - 출력 : BFS 순서로 모든 정점을 가지고 있는 리스트 L.

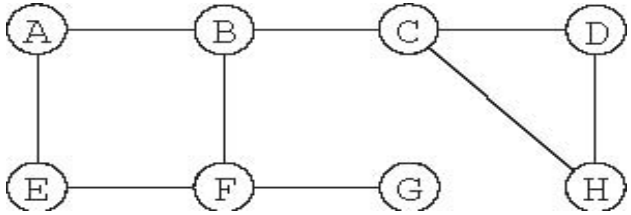
1. 큐,스택? Q와 정점의 출력 리스트 L을 초기화.
2. v를 방문했다고 마크하고 Q에 삽입.
3. Q로부터 x를 삭제하고 L에 삽입.
4. x에 인접한 각각의 정점 y에 대해 단계 5를 반복.
5. 만일 y를 아직 방문하지 않았다면, 이것을 방문했다고 마크하고 Q에 삽입.
6. 만일 Q가 공백이 아니면, 단계 3으로 이동.

너비 우선 탐색의 수행 과정

- 아래 그래프에 대해 정점 A에서 시작하는 경우, 알고리즘 수행 과정을 살펴보자. 정점들은 앞에서 설명한 것과 같이 A, B, E, C, F, D, H, G의 순서로 방문되는데, 이것은 출력 리스트 L에 삽입되는 순서이다.
- 방문되는 정점들이 넓게 퍼지기 때문에 "너비 우선"이라는 용어를 사용한다. 이것은 그림 16.10을 보면 잘 알 수 있다.
- 이것은 방문된 정점(녹색으로 표시)의 집합의 경계 상에 있는 정점들을 각각 방문하는 일련의 단계를 통해 진행된다. 이 과정을 통해 방문되는 마지막 정점은 시작 정점인 A로부터 가장 멀리 떨어져 있다.



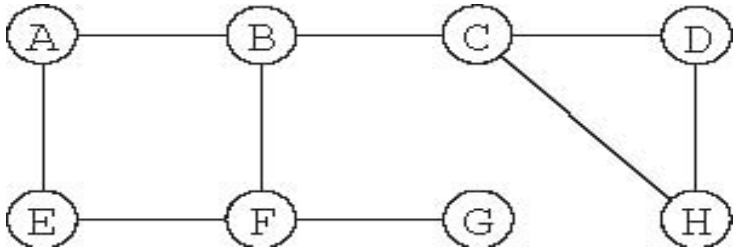
너비 우선 탐색 알고리즘 수행 과정

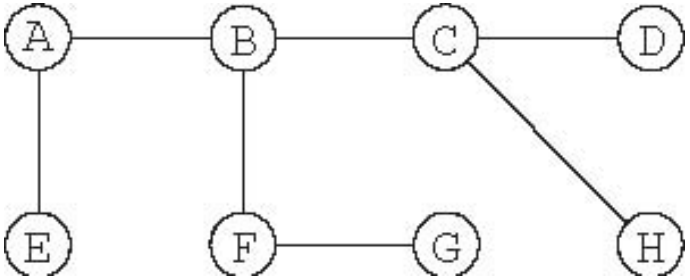


Q	*x*	*L*
A		
	A	A
B		
BE		
E	B	AB
EC		
C	E	ABE
CF		
F	C	ABEC
FD		
FDH		
DH	F	ABECF
DHG		
HG	D	ABECFD
G	H	ABECFDH
	G	ABECFDHG

16.5 신장 트리

- 그래프를 위한 *신장 트리(spanning tree)*
  - 그래프의 모든 정점을 연결하는 서브트리이다.
  - 연결 순환 그래프가 몇 개의 신장 트리를 갖는 것은 쉽게 보일 수 있다. 아래 오른쪽(그림 16.10) 서브트리는 왼쪽(그림 16.8)에 보인 그래프를 위한 신장 트리이다.





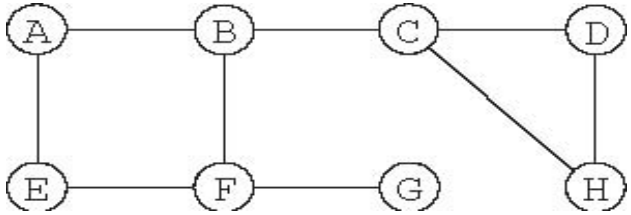
- 신장 트리의 유용성
  - 그림 16.1에 보인 유럽 인터넷 연결망에서, 신장 트리는 하나의 노드에서 다른 노드로 가는 유일한 통신선을 정의하고 있다.

BFS 신장 트리 알고리즘

- BFS 신장 트리 알고리즘
  - 입력 : 그래프 G=(V,E)와 최초 정점 v∈V.
  - 출력 : G를 위한 신장 트리 T.

1. 지역 큐 Q와 정점의 출력 트리 T를 초기화.
2. v를 방문했다고 마크하고 Q에 삽입한 다음,
   그것을 T의 루트로 삽입.
3. Q로부터 x를 삭제.
4. x에 인접한 각각의 정점 y에 대해 단계 5를 반복.
5. 만일 y를 아직 방문하지 않았다면, 이것을 방문했다고
   마크하고 Q에 삽입한 다음 x의 다음 자식으로 T에 삽입.
6. 만일 Q가 공백이 아니면, 단계 3으로 이동.

너비 우선 탐색과 BFS 신장 트리



Q	*x*	*L*	*y*
A			
	A	A	B
B			E
BE			
E	B	AB	C
EC			
C	E	ABE	F
CF			
F	C	ABEC	D
FD			H
FDH			
DH	F	ABECF	G
DHG			
HG	D	ABECFD	
G	H	ABECFDH	
	G	ABECFDHG	

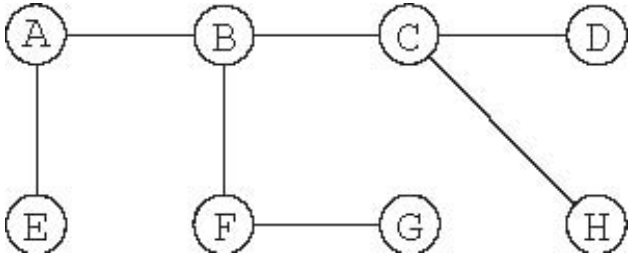
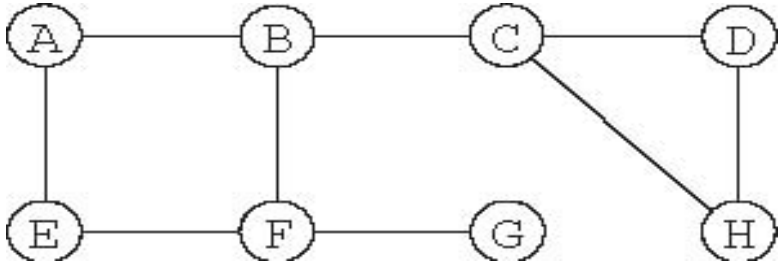


그림 16.11 BFS 신장 트리

16.6 깊이 우선 탐색

- 정점 A에서 시작하여 첫 번째로 만나는 자식은 정점 B이고, 이것의 첫 번째 자식은 정점 C가 되고, 이것의 첫 번째 자식은 정점 D가 되고, 이것의 첫 번째 자식은 정점 H가 된다.
- 이런 다음 방문하는 다음 정점은 마지막으로 방문한 부모가 자식을 더 가지고 있을 경우 다음 자식이 된다. 이것은 부모 B에 갈 때까지 되추적(backtrack)되는데, 이것의 다음 자식은 정점 F가 되고, 이것의 첫 번째 자식은 정점 G가 된다.
- 마지막으로 정점 E가 A의 두 번째 자식으로 방문된다. 따라서 완전한 순회는 A, B, C, D, H, F, G, E의 순서로 정점들을 방문한다.



16.6 깊이 우선 탐색

- 각각의 노드를 역순(오른쪽에서 왼쪽)으로 처리하는 "역(reverse)" 전위 순회라는 것을 제외하고는 동일한 프로세스를 적용해 보자. 이 버전의 순회는 A, E, F, G, B, C, H, D의 순서로 정점을 방문한다. 이것은 알고리즘 16.1에 큐 대신 스택을 사용한 경우와 정확하게 일치하는데, 이것을 깊이 우선 탐색(depth-first search)라고 부른다.

깊이 우선 탐색(DFS) 알고리즘

- 깊이 우선 탐색(DFS) 알고리즘
  - 입력 : 그래프 G=(V,E)와 최초 정점 v∈V.
  - 출력 : DFS 순서로 모든 정점을 가지고 있는 리스트 L.

1. 지역 스택 S와 정점의 출력 리스트 L을 초기화.
2. v를 방문했다고 마크하고 S에 삽입.
3. S로부터 x를 삭제하고 L에 삽입.
4. x에 인접한 각각의 정점 y에 대해 단계 5를 반복.
5. 만일 y를 아직 방문하지 않았다면, 이것을 방문했다고 마크하고 S에 삽입.
6. 만일 S가 공백이 아니면, 단계 3으로 이동.

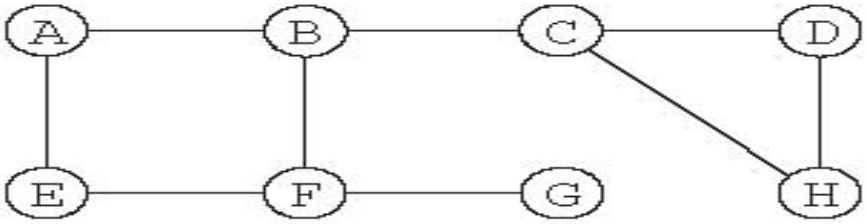
깊이 우선 탐색의 수행 과정

- 그림 16.12는 이전과 동일한 그래프에 대한 수행 과정을 보이고 있다. 출력 리스트는 A, E, F, G, B, C, H, D를 가지고 있는데, 이것은 이전에 살펴본 역 전위 순회의 경우와 같다.

- "깊이 우선"이라는 용어는 순회가 다른 시이퀀스를 따라 되추적되기 전에 정점들이 갈 수 있는 만큼 깊이 들어가는 하나의 시이퀀스를 따라간다는 사실을 반영하고 있다.

- 따라서, 첫 번째 시이퀀스 {A, E, F, G}이 되고, 다음 시이퀀스는 {B, C, H, D}가 된다.

깊이 우선 탐색과 DFS 신장 트리



S	*x*	*L*	*y*
A			
	A	A	B
B			E
BE			
B	E	AE	F
BF			
B	F	AEF	G
BG			
B	G	AEFG	
	B	AEFGB	C
C	C	AEFGBC	D
D	F		H
DH			
D	H	AEFGBCH	
	D	AEFGBCHD	

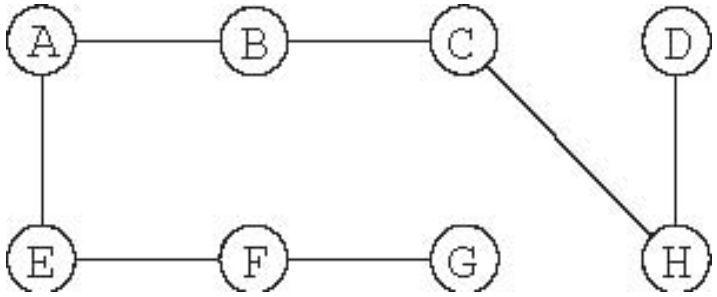


그림 16.13 DFS 신장 트리

순환 깊이 우선 탐색(DFS) 알고리즘

- 순환 깊이 우선 탐색(DFS) 알고리즘
  - 입력 : 그래프 G=(V,E)와 최초 정점 v∈V.
  - 출력 : DFS 순서로 모든 정점을 가지고 있는 리스트 L.

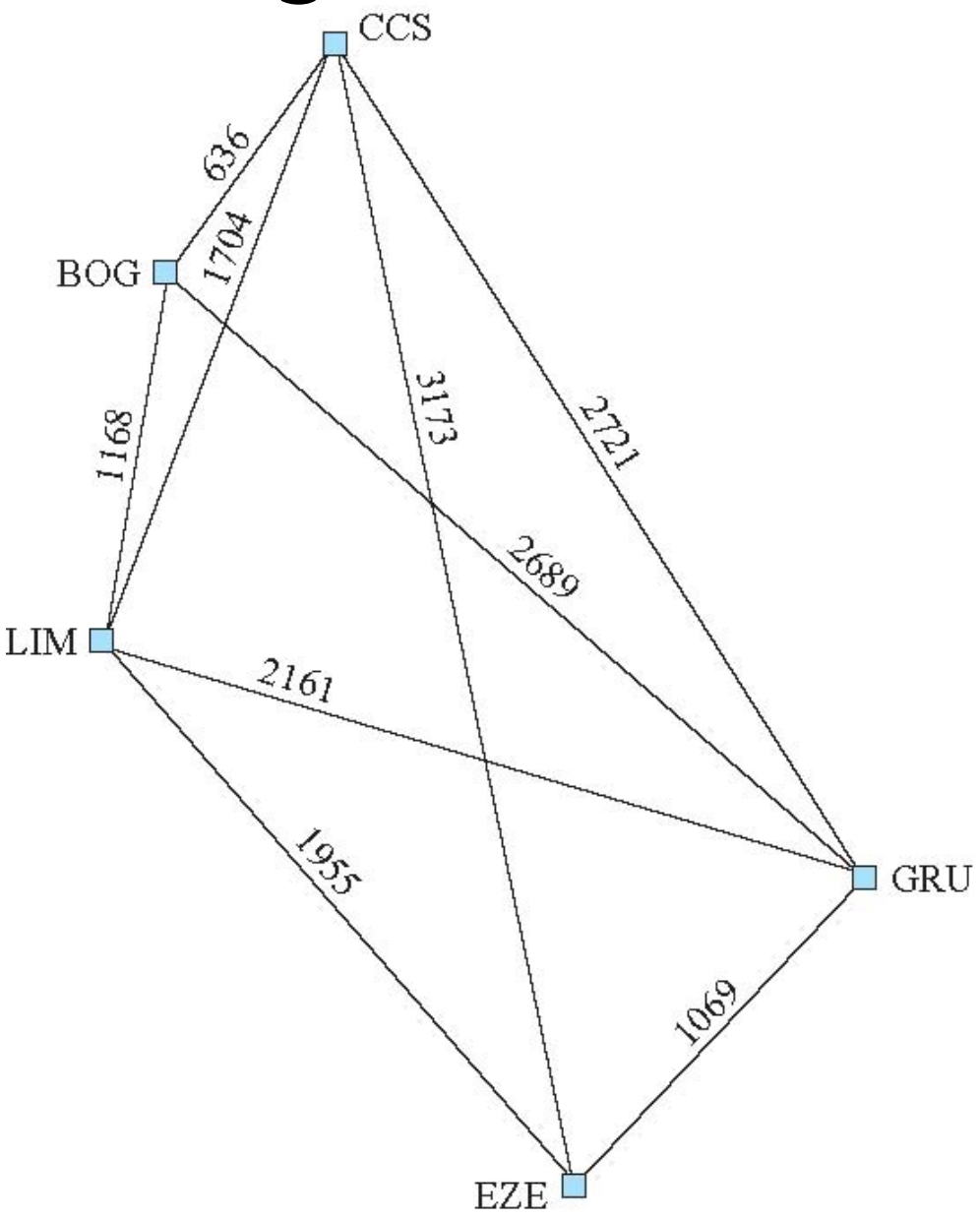
1. v를 방문했다고 마크하고 L에 삽입.
2. v에 인접한 각각의 정점 y에 대해 단계 3을 반복.
3. 만일 y를 아직 방문하지 않았다면,
   L=depthFirstSearch(G,y,L)로 설정.

16.7 가중치 그래프

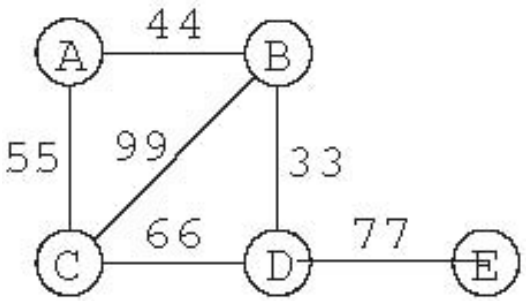
- 가중치 그래프
  - (V, E)가 그래프이고, w:E→R은 각각의 간선 e∈E에 이 간선의 가중치(비용 또는 길이)라고 불리는 숫자 w(e)를 할당하는 함수일 때, 삼원소쌍 G=(V, E, w)이다.
  - 만일 p=(v0, v1, v2, ..., vk)가 가중치 그래프의 경로라고 하면, 경로 w(p)의 가중치(비용 또는 길이)는 경로 상에 있는 모든 간선들의 가중치의 합으로 정의된다.

예 16.3: Air Mileages (비행 거리)

남미 도시 간의 비행 거리

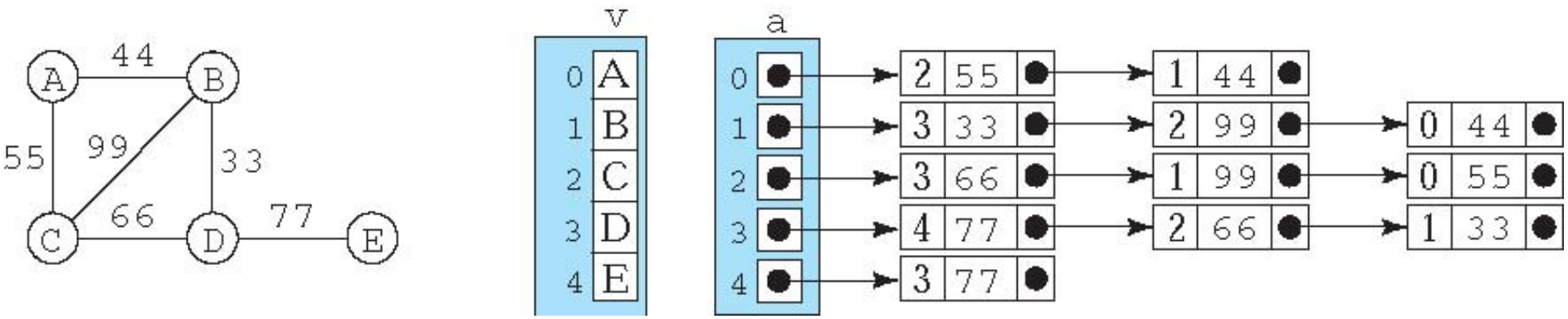


가중치 그래프를 위한 인접 행렬



	A	B	C	D	E
A	0	44	55	∞	∞
B	44	0	99	33	∞
C	55	99	0	66	∞
D	∞	33	66	0	77
E	∞	∞	∞	77	0

가중치 그래프를 위한 인접 리스트

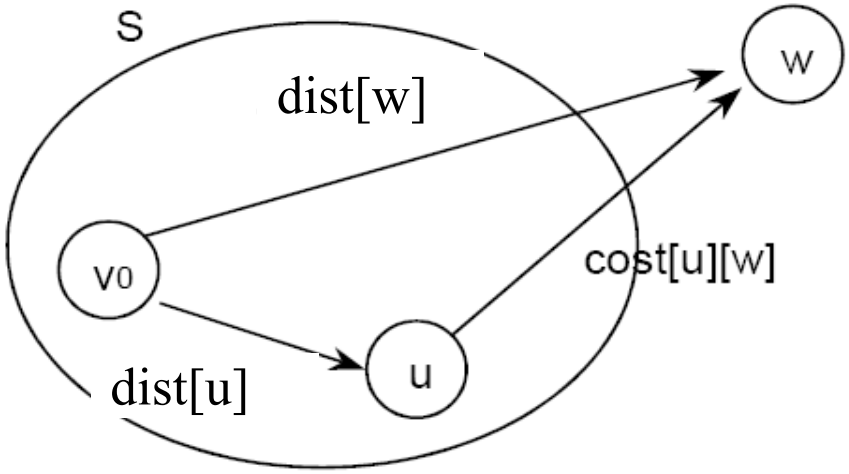


16.8 Dijkstra의 알고리즘

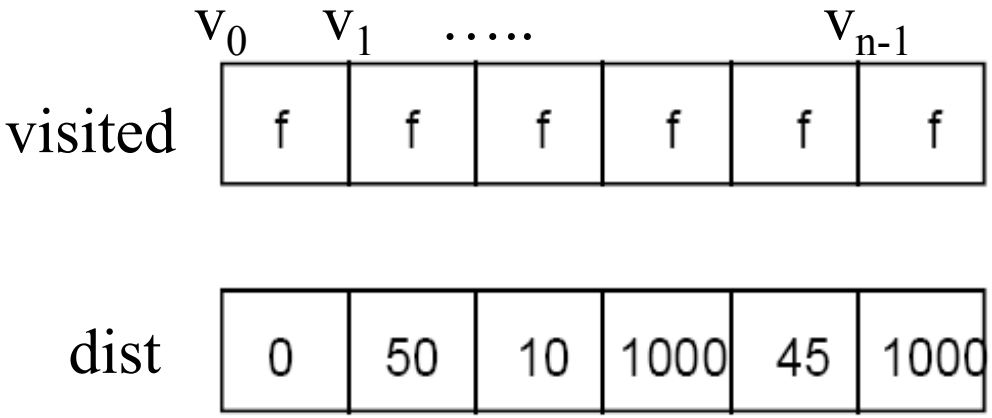
- 가중치 그래프에서 하나의 정점으로부터 다른 정점으로 가는 최단 경로를 계산하는 알고리즘
- 1959년에 Edsger Dijkstra는 e가 간선의 개수이고 n이 그래프에 있는 정점의 개수일 때, *O(e logn)* 시간 내에 하나의 정점에서 다른 각각의 정점으로 가는 최단 경로를 찾는 알고리즘을 발견하였다.
- 이것은 각각의 정점에서 boolean visited 필드 외에 prev 필드와 dist 필드 등 두 개의 필드를 더 필요로 한다. prev 필드는 시작 정점으로부터 최단 거리에 있는 이전 정점을 가리키고, dist 필드는 최단 거리의 길이를 가지고 있다. 또한, 이 알고리즘은 최소 dist가 가장 높은 우선순위를 가지는 정점들의 우선순위 큐를 사용한다.

16.8 Dijkstra의 알고리즘

- Dijkstra의 알고리즘은 원형적(prototypical) "갈망" 알고리즘이다. 이것은 간선의 가중치 함수에 의해 제어되는 수정된 너비 우선 탐색을 사용한다. 각각의 반복에서, 이것은 처음 정점 v에 가장 가까운 미방문 정점 x를 방문한 다음, x에 인접한 모든 정점 y의 y.prev와 y.dist 필드를 갱신한다.



dist[w]← min{dist[w], dist[u] + cost[u][w]}



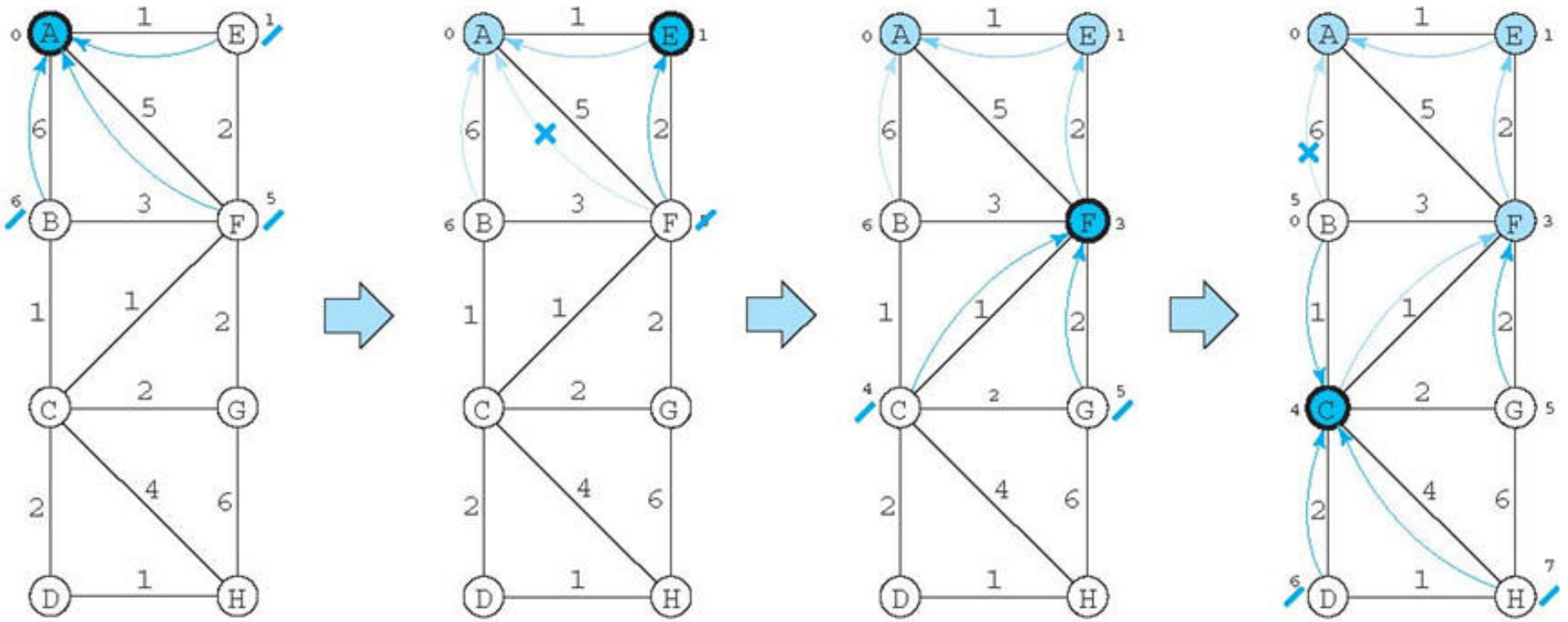
If $v_i \in S$, visited[i]=true
If $v_i \notin S$, visited[i]=false

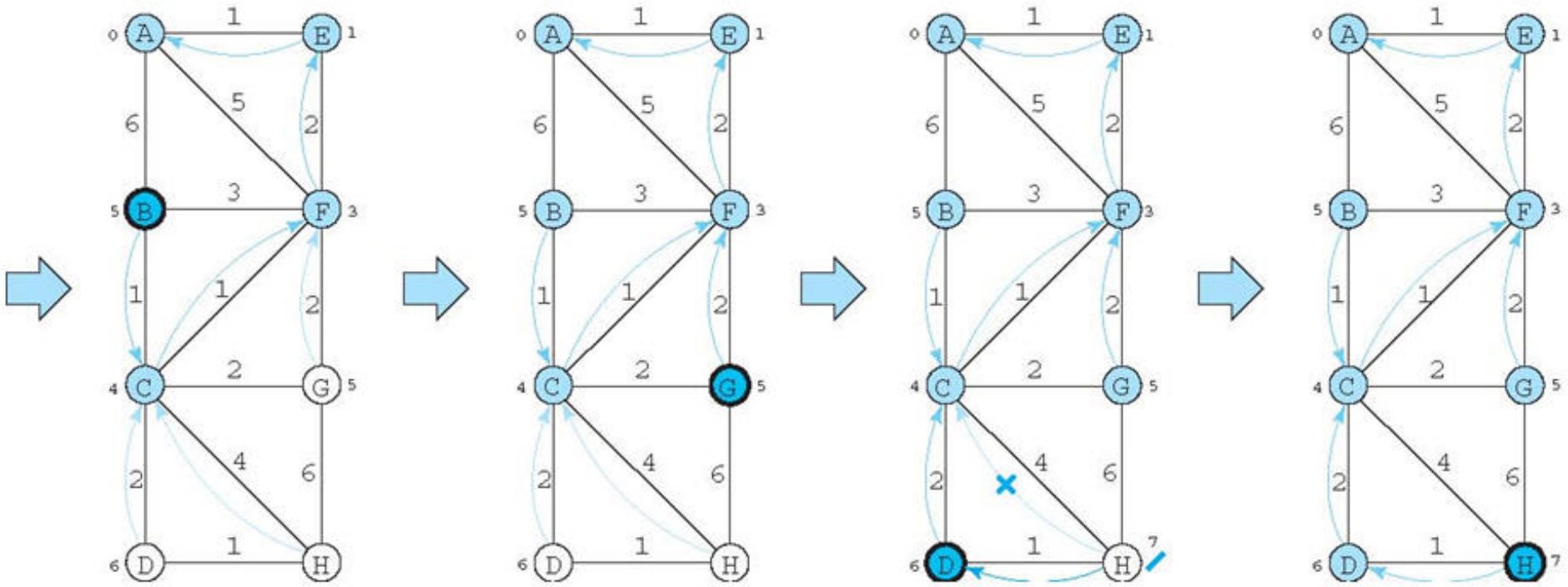
Dijkstra의 최단 경로 알고리즘

- 입력 : 가중치 그래프 G=(V,E,w)와 최초 정점 v0∈V.
- 출력 : prev와 dist 필드 집합을 가지고 있는 정점 집합 V.
- 후조건 : 각각의 정점 v∈V에 대해, prev 포인터에 의해 정의되는 경로는 v0에서 v로 가는 최단 경로이고, v.dist는 이것의 길이가 된다.

1. 모든 정점 v≠v0에 대해 v.dist=∞로 설정
2. 모든 정점들을 방문할 때까지 단계 3-6을 반복.
3. x를 최소 dist를 가지는 미방문 정점으로 설정하고, x를 방문했다고 마크.
4. x에 인접한 각각의 미방문 정점 y에 대해 단계 5-6을 반복.
5. dy=x.dist+w(x,y)로 설정.
6. 만일 dy<y.dist이면, y.dist=dy 와 y.prev=x 로 설정.
   (더 짧은 경로가 발견됨.)

Dijkstra 알고리즘의 수행 과정




A
E
B
F
C
G
D
H
0
1
5
6
3
2
1
1
2
2
4
6
1
6
5
1
3
2
1
1
2
2
4
6
1
6
5
1
3
2
1
1
2
2
4
6
1
6
5
1
3
2
1
1
2
2
4
6
1
7

Listing 16.4 : Dijkstra 알고리즘

```
public class WeightedGraph {
    Vertex start;
```

```
private static class Vertex {
    private Object object;
    Edge edges;
    Vertex nextVertex;
    boolean done;
    int dist;
    Vertex back;

    public String toString( ) { ..... }
    void printPath( ) { ....  }
}
```

```
private static class Edge {
    Vertex to;
    int weight;
    Edge nextEdge;

    Edge(vertex to, int weight,
          Edge nextEdge) {
      this.to=to; this.weight=weight;
      this.nextEdge= nextEdge;
    }
    public String toString( ) { ..... }
}
```

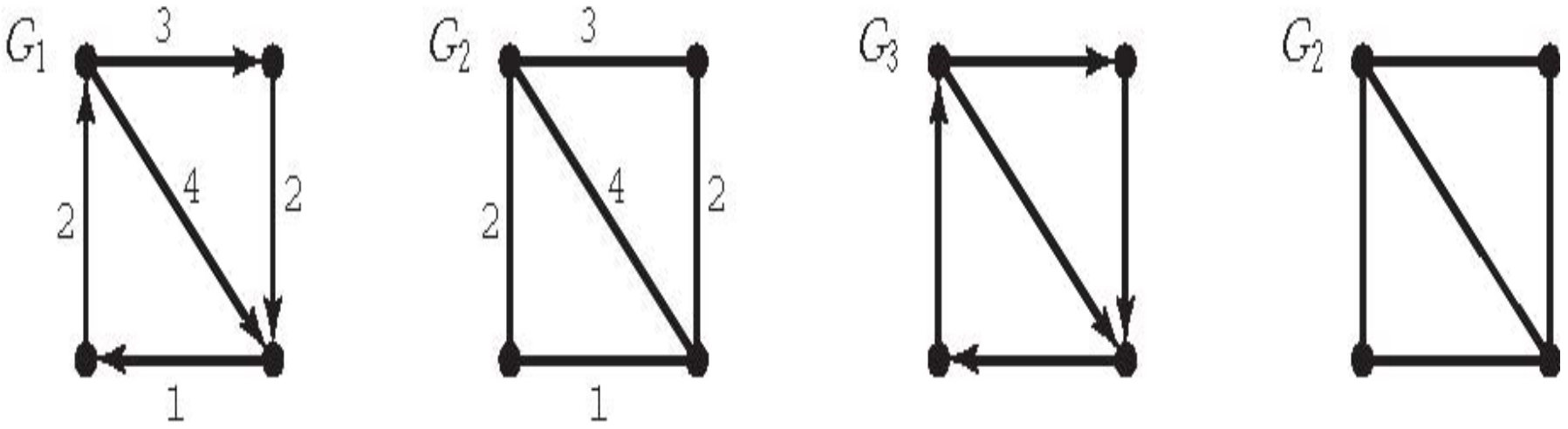
```
    public Weightedgraph(String[ ] args){
       Vertex v = start = new Vertex(args[0]);
       for (int i=1; i<args.length; i++) {
           v = v.nextVertex = new Vertex(args[i]);
           v.dist = Integer.MAX_VALUE; // infinity    }   }
```

```
  public void addEdge(String vString, String wString, int weight) { ...}

  public void findShortestPaths() {
     // implements Dijkstra's Algorithm
    ..... }

  public void printPaths() { .... }

  public String toString() { .... }


  private Vertex find(Object object) {
     // returns the vertex that contains the specified object:
     ..... }


  private Vertex closestVertex() {
     // returns the undone vertex with smallest dist field:
     ..... }


}
```

16.9 다이그래프 (Digraphs)

- 다이그래프(digraph)
  - 방향 그래프(directed graph)라고도 함.
  - 간선들이 하나의 방향을 가지고 있다.
  - 공식적인 정의는 간선을 두 정점의 무순서 집합이 아니라 두 정점의 시이퀀스로 정의한다.
  - *가중치 다이그래프(weighted digraph)*는 각각의 방향 간선에 숫자를 할당하는 가중치 함수를 가지는 다이그래프이다. 가중치 다이그래프를 *네트워크(network)*라고 부른다.

- 가중치 다이그래프
  - 세 가지 종류의 그래프를 생각해 볼 수 있다.
    1. 가중치가 없는 다이그래프(가중치를 무시),
    2. 가중치 그래프(간선들이 양방향이라고 해석),
    3. 그래프 자신이다.

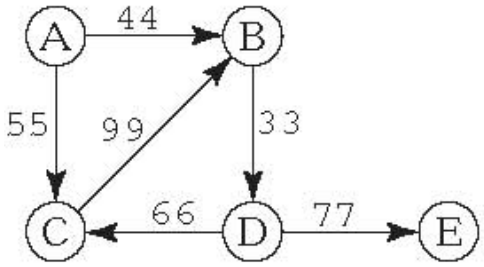


가중치 다이그래프와 내장 구조

- 다이그래프와 네트워크를 저장하는데 사용되는 자료 구조
  - 인접 행렬과 인접 리스트를 사용한다.
  - 다이그래프에서 유일하게 다른 점은 인접 행렬이 대칭일 필요가 없으며, 인접 리스트는 간선당 두 개가 아니라 하나의 리스트 노드만 사용한다는 것이다.

가중치 다이그래프를 위한 인접 행렬과 인접 리스트

– 그래프는 여섯 개의 방향 간선을 가지고 있으므로, 인접 행렬은 양의 유한 항목을 여섯 개 가지고 있고, 인접 리스트는 여섯 개의 노드만을 가지고 있다.



	A	B	C	D	E
A	0	44	55	∞	∞
B	∞	0	∞	33	∞
C	∞	99	0	∞	∞
D	∞	∞	66	0	77
E	∞	∞	∞	∞	0