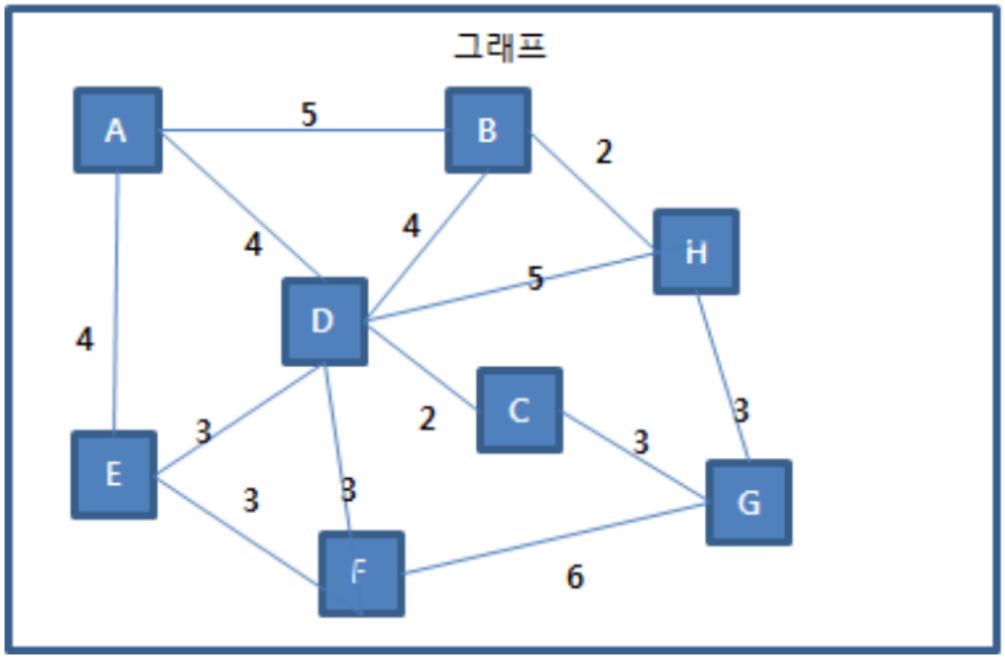
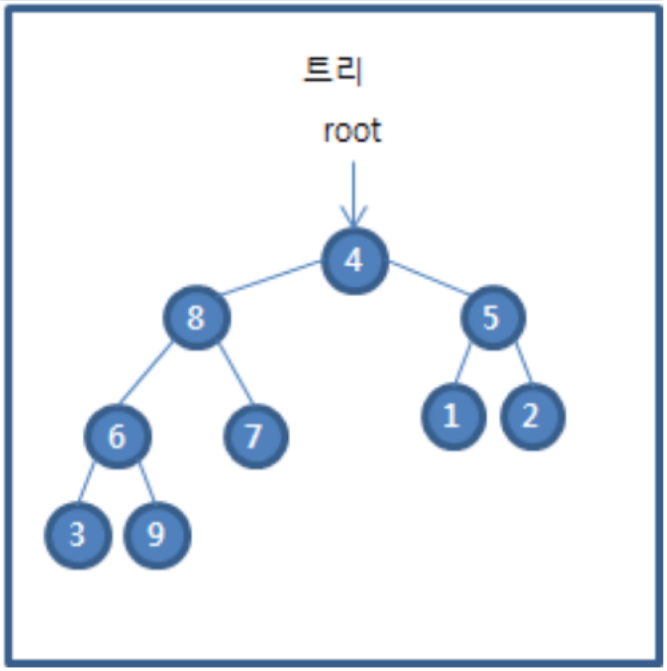
비선형 자료구조

비선형 자료구조

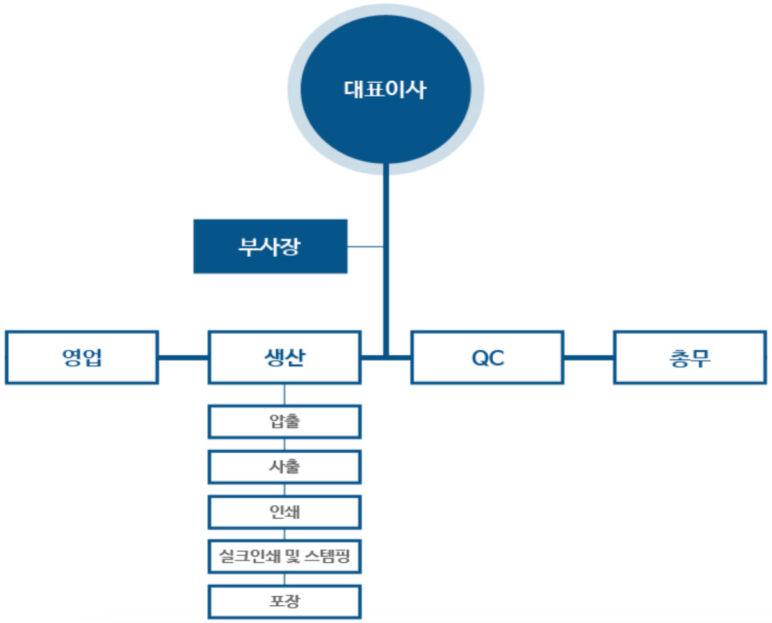
- 비선형 자료구조
  - ✓ 하나의 자료 뒤에 여러 개의 자료가 존재할 수 있는 구조
  - ✓ 비선형 자료구조의 종류
    - 트리: 방향성 있고 사이클이 존재하지 않으며 고립 영역이 없는 그래프
    - 그래프: 정점과 간선의 집합

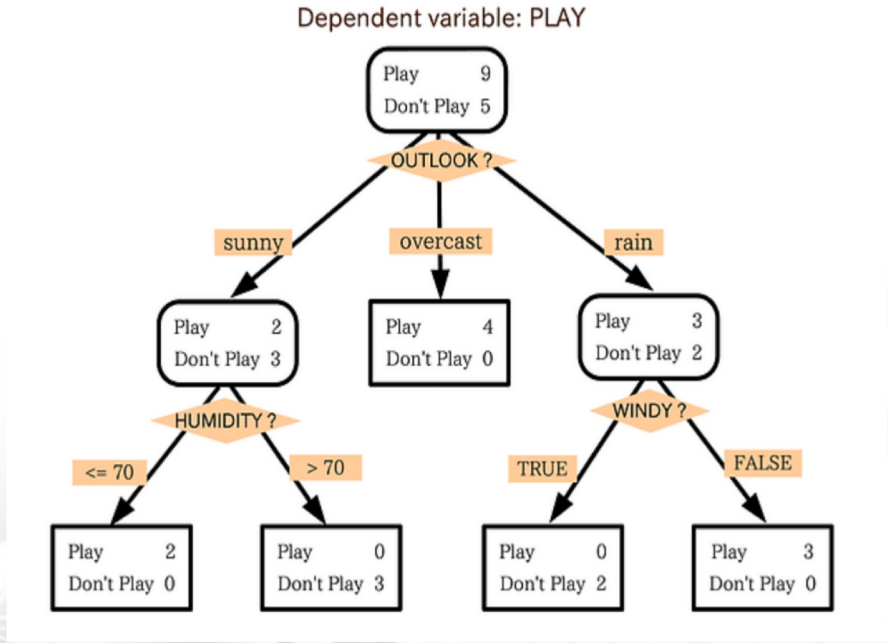




Tree

- Tree
  - ✓ 계층 구조로 자료를 저장하는 구조
  - ✓ 트리를 구성하는 노드가 부모-자식(Parent – Child) 관계라는 의미
  - ✓ 부모 노드 하나에 여러 개의 자식 노드들이 연결되는 구조
  - ✓ 회사의 조직도, 컴퓨터의 디렉토리 구조 등이 대표적이며 머신러닝에 사용되는 의사결정 트리 등이 있음



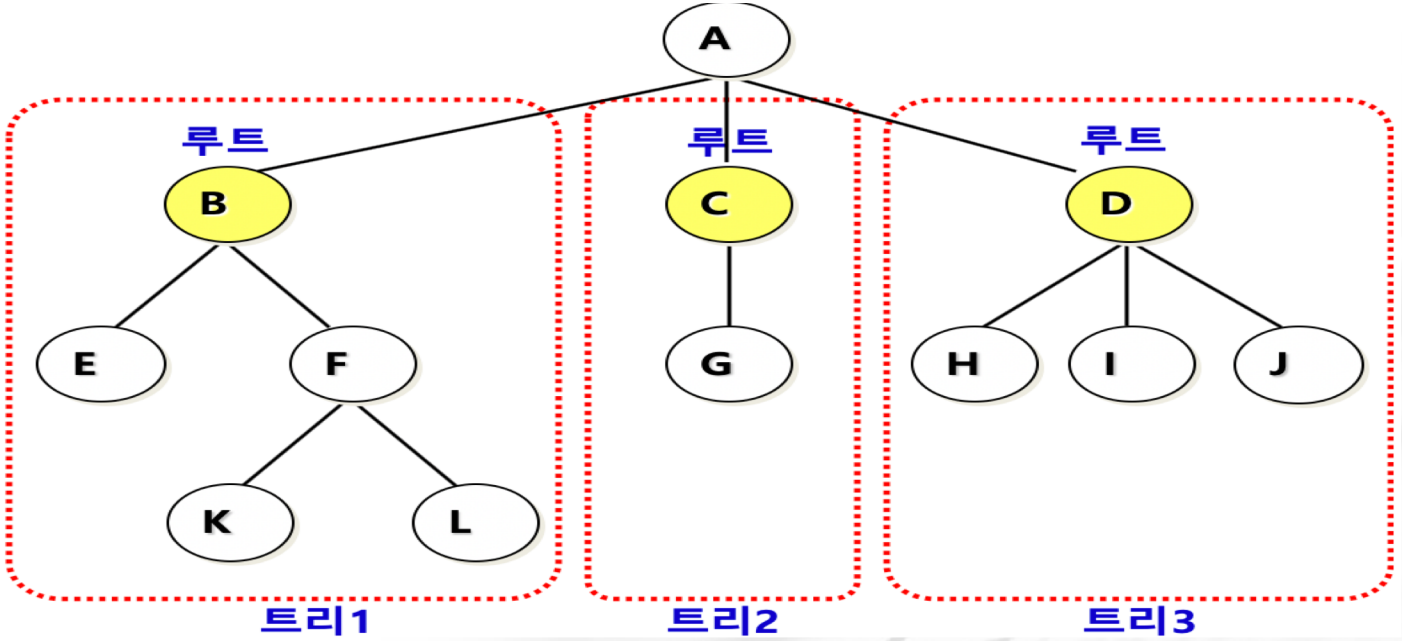


Tree

- Tree
  - ✓ 노드(Node)와 간선(Edge)의 집합
  - ✓ 조직도에서는 부서가 노드가 되며 의사결정 트리에서는 의사결정을 위한 조건이 노드
  - ✓ 간선은 이러한 노드들 사이의 부모-자식 관계를 정의
  - ✓ 노드의 종류
    - Root: 트리의 첫번째 노드
    - Leaf(Terminal): 자식 노드가 없는 노드
    - Internal: 자식 노드가 있는 노드
  - ✓ 노드 사이의 관계
    - Parent: 간선으로 연결된 상위 노드
    - Child: 간선으로 연결된 하위 노드
    - Ancestor(선조): root에서 parent 까지의 경로 상에 있는 모든 노드
    - Descendant(후손): 특정 노드의 아래에 있는 모든 노드
    - Sibling(형제): 같은 부모 노드의 자식 노드

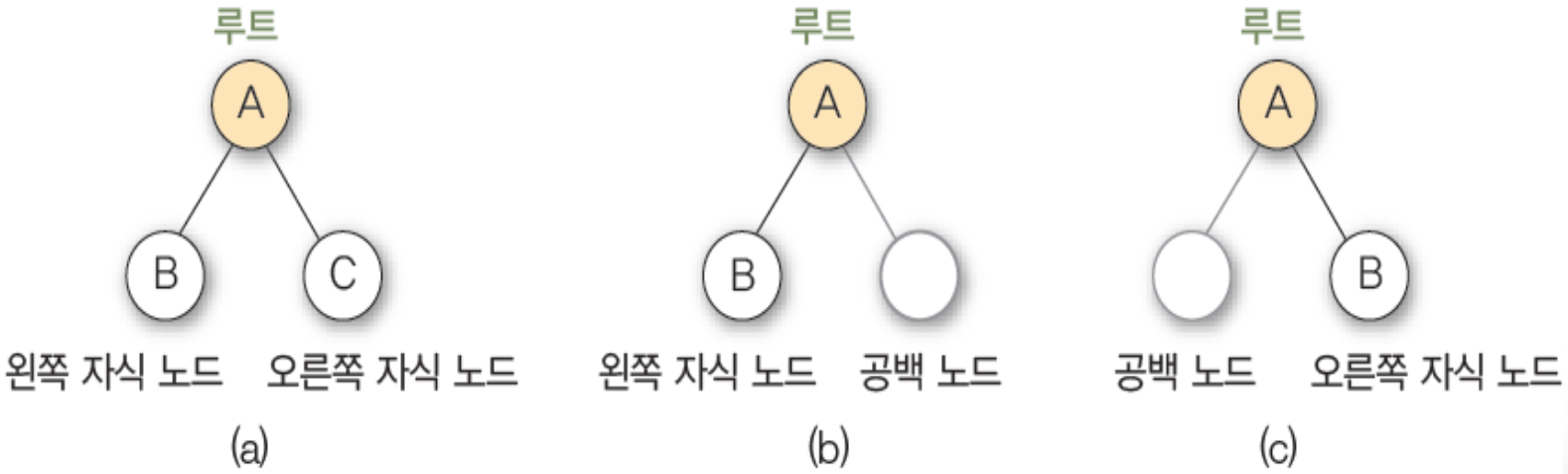
Tree

- Tree
  - ✓ 노드의 속성
    - Level: 루트 노드부터의 거리
    - Height: 루트 노드로부터 가장 먼 거리에 있는 자식 노드의 높이에 1을 더한 값
    - Degree: 한 노드가 가지는 자식 노드의 개수
  - ✓ Subtree: 트리에 속한 노드들의 부분 집합
  - ✓ Forest: 특정 노드가 제거되면서 만들어지는 Subtree 의 집합



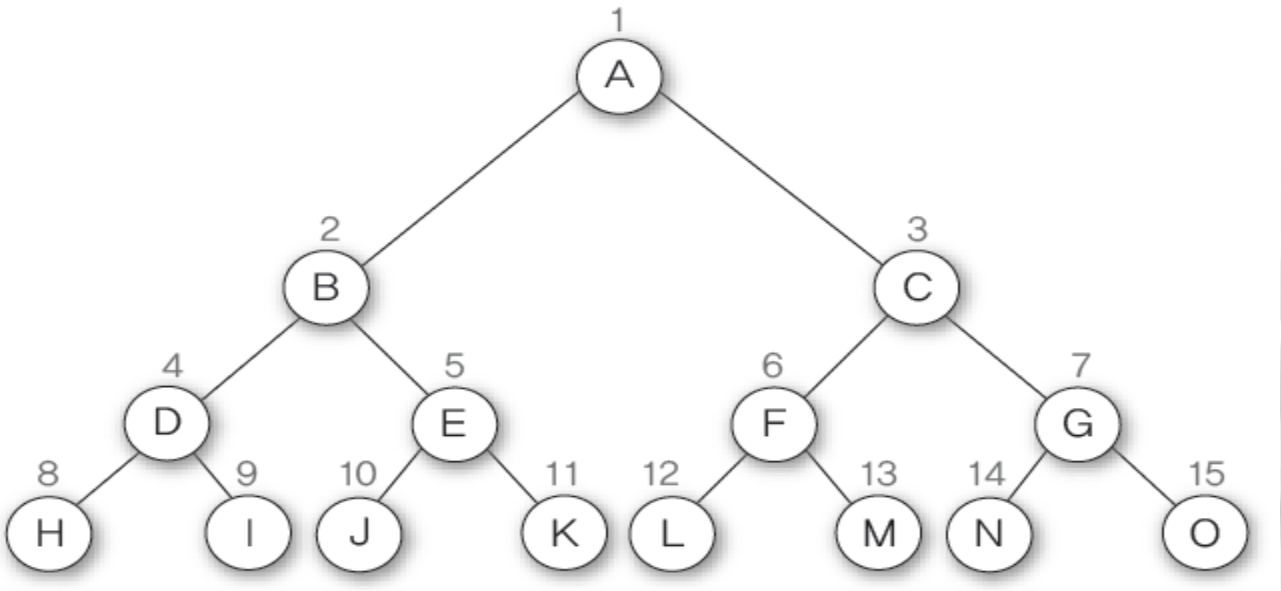
Binary Tree

- Binary Tree
  - ✓ 모든 노드의 차수가 2 이하인 트리



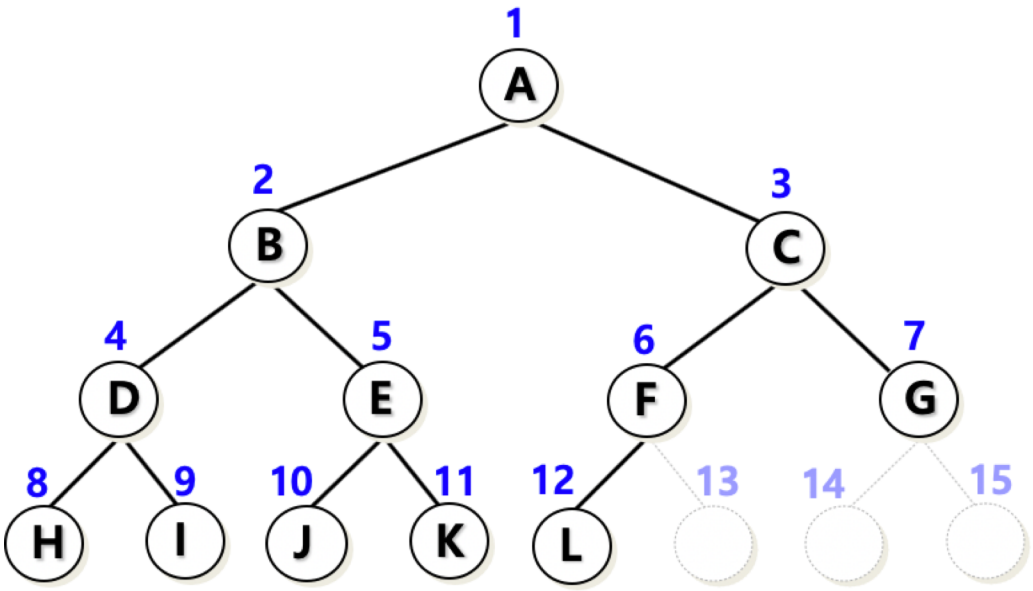
Binary Tree

- Binary Tree
  - ✓ Binary Tree 의 종류
    - Full Binary Tree(포화 이진 트리)
      - 모든 레벨의 노드가 꽉 차있는 이진 트리
      - 높이가 h일 때 최대의 노드 개수인 $(2^{h+1}-1)$의 노드를 가진 이진 트리
      - 루트를 1번으로 하여 $2^{h+1}-1$까지 정해진 위치에 대한 노드 번호를 가짐



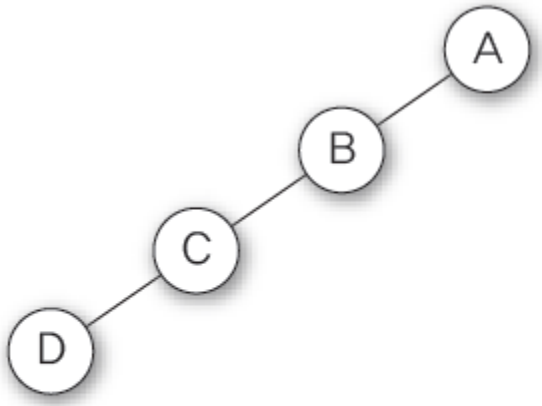
Binary Tree

- Binary Tree
  - ✓ Binary Tree 의 종류
    - Complete Binary Tree(완전 이진 트리)
      - 높이가 h이고 노드 수가 n개일 때 (단, $n < 2^{h+1}-1$)
      - 노드 위치가 포화 이진 트리에서의 노드 1번부터 n번까지의 위치와 완전히 일치하는 이진 트리
      - 완전 이진 트리에서는 (n+1)번부터 ($2^{h+1}-1$)번까지 노드는 모두 공백 노드

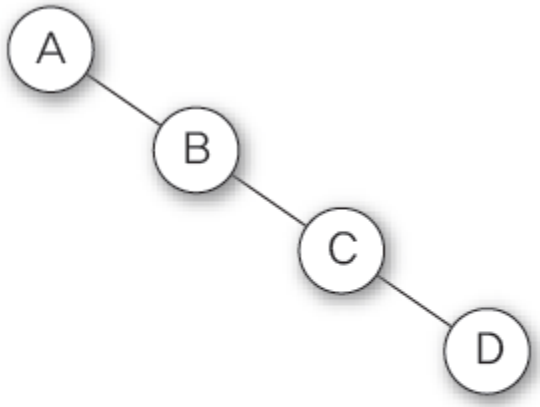


Binary Tree

- Binary Tree
  - ✓ Binary Tree 의 종류
    - Skewed Binary Tree(편향 이진 트리)
      - ◆ 같은 높이의 이진 트리 중에서 최소 개수의 노드 개수를 가지면서 왼쪽 또는 오른쪽 서브트리만을 가진 이진 트리



(a) 왼쪽 편향 이진 트리



(b) 오른쪽 편향 이진 트리

Binary Tree

- Binary Tree 구현
  - ✓ TreeNode

```
class TreeNode {
    int data;
    TreeNode left;
    TreeNode right;
}
```

Binary Tree

- Binary Tree 구현
  - ✓ LinkedTree

```
class LinkedTree {
    private TreeNode root;

    public LinkedTree(int data) {
            root = new TreeNode();
            root.data = data;
            root.left = null;
            root.right = null;
    }

    public TreeNode getRootNode() {
            return root;
    }
```

Binary Tree

- Binary Tree 구현
  - ✓ LinkedTree

```
public void insertLeftChildNode(TreeNode pParentNode, int element) {
        if (pParentNode.left == null) {
                pParentNode.left = new TreeNode();
                pParentNode.left.data = element;
        } else {
                System.out.println("오류, 이미 노드가 존재합니다.");
        }
}
public void insertRightChildNode(TreeNode pParentNode, int element) {
        if (pParentNode.right == null) {
                pParentNode.right = new TreeNode();
                pParentNode.right.data = element;
        } else {
                System.out.println("오류, 이미 노드가 존재합니다.");
        }
}
```

Binary Tree

- Binary Tree 구현
  - ✓ LinkedTree

```
        public TreeNode getLeftChildNode(TreeNode pNode)
        {
                TreeNode pReturn = pNode.left;
                return pReturn;
        }

        public TreeNode getRightChildNode(TreeNode pNode)
        {
                TreeNode pReturn = pNode.right;
                return pReturn;
        }
    }
```

Binary Tree

- Binary Tree 구현
  - ✓ LinkedTreeMain

```
public class LinkedTreeMain {

    public static void main(String[] args) {
        LinkedTree tree = new LinkedTree(100);

        TreeNode root = tree.getRootNode();
        System.out.println(root.data);

        tree.insertLeftChildNode(root, 200);
        tree.insertRightChildNode(root, 300);

        System.out.println(tree.getLeftChildNode(root).data);
        System.out.println(tree.getRightChildNode(root).data);
    }
}
```

Binary Tree

❖ Binary Tree 순회

✓ 전위 순회(preorder traversal)

- D(현재 노드) → L(왼쪽 서브 트리) → R(오른쪽 서브 트리) 순서로 처리

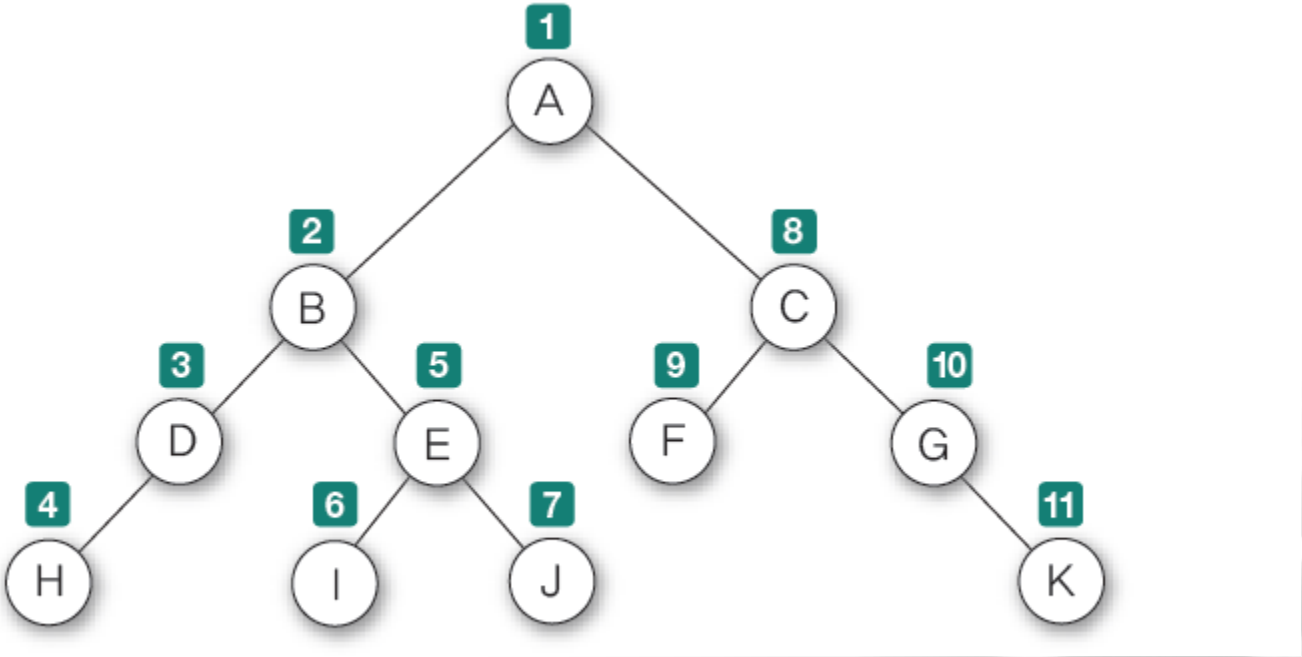
❶ 작업 D : 현재 노드 n을 처리한다.

❷ 작업 L : 현재 노드 n의 왼쪽 서브 트리로 이동한다.

❸ 작업 R : 현재 노드 n의 오른쪽 서브 트리로 이동한다.

Binary Tree

- Binary Tree 순회
  - ✓ 전위 순회(preorder traversal)



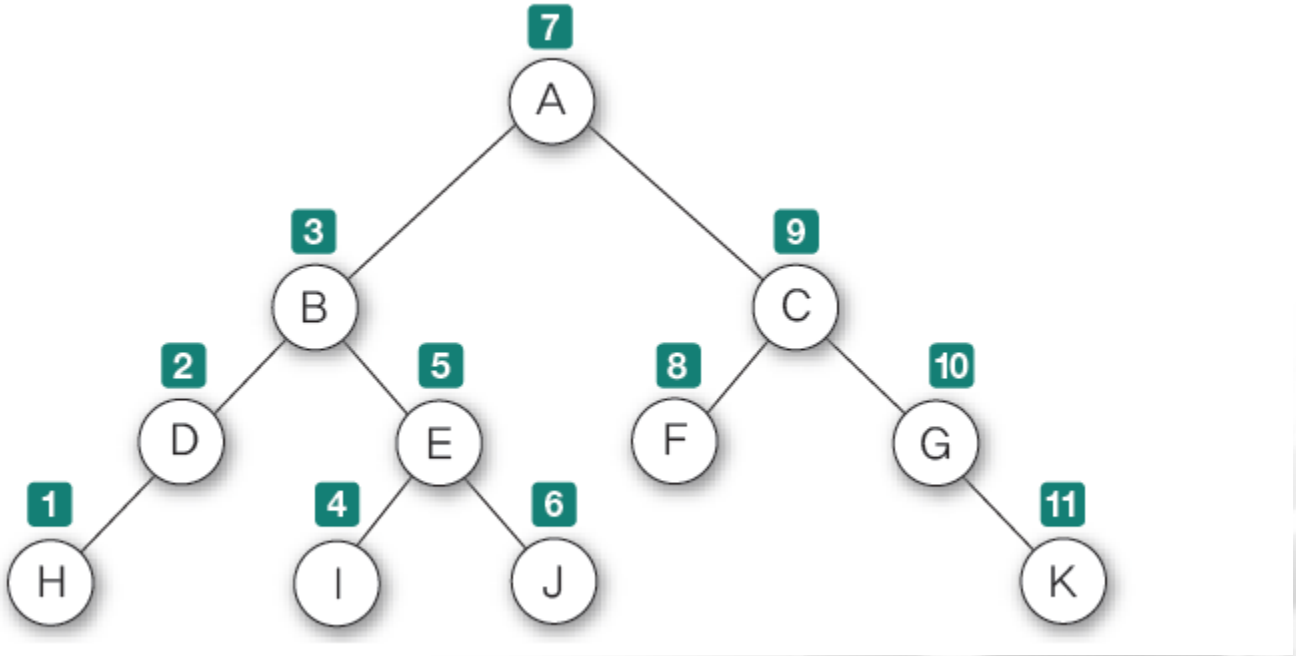
Binary Tree

- Binary Tree 순회
  - ✓ 중위 순회(Inorder traversal)
    - L → D → R 순서로, 현재 노드를 방문하는 작업 D를 작업 L과 작업 R의 중간에 수행

> ❶ 작업 L : 현재 노드 n의 왼쪽 서브 트리로 이동한다.
>
> ❷ 작업 D : 현재 노드 n을 처리한다.
>
> ❸ 작업 R : 현재 노드 n의 오른쪽 서브 트리로 이동한다.

Binary Tree

- Binary Tree 순회
  - ✓ 중위 순회(Inorder traversal)



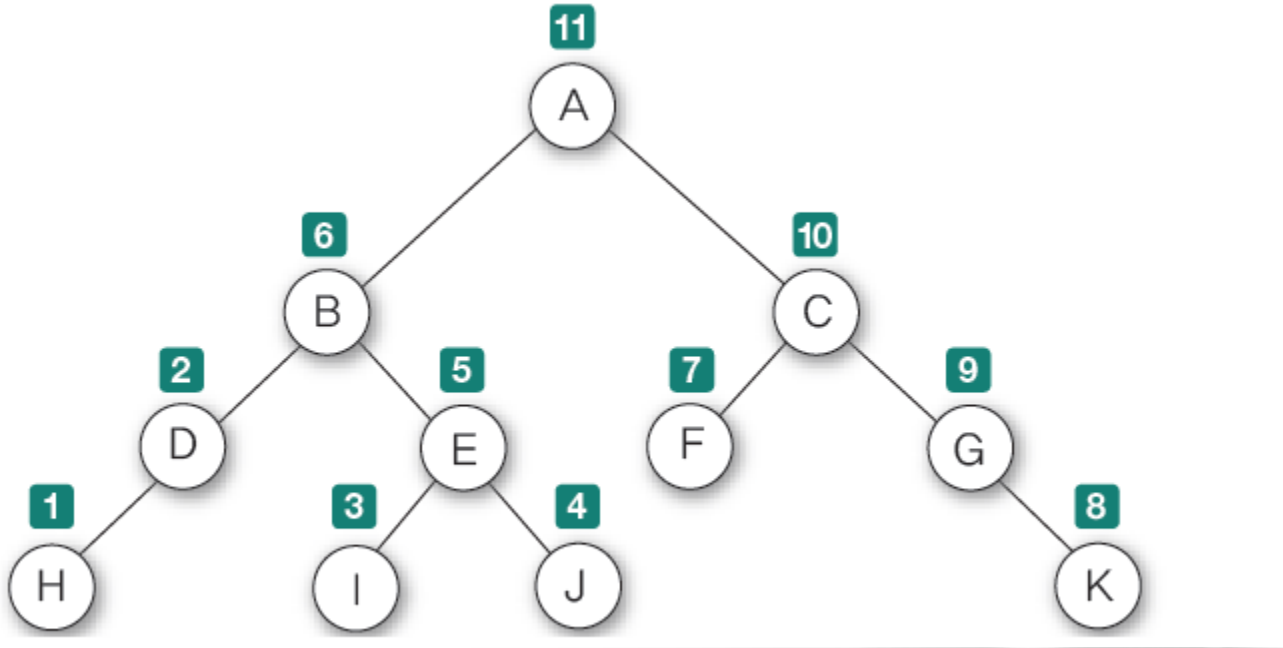
Binary Tree

- Binary Tree 순회
  - ✓ 후위 순회(Postorder traversal)
    - L-R-D 순서로 현재 노드를 방문하는 D 작업을 가장 나중에 수행

> ❶ 작업 L : 현재 노드 n의 왼쪽 서브 트리로 이동한다.
>
> ❷ 작업 R : 현재 노드 n의 오른쪽 서브 트리로 이동한다.
>
> ❸ 작업 D : 현재 노드 n을 처리한다.

Binary Tree

- Binary Tree 순회
  - ✓ 후위 순회(Postorder traversal)



Binary Tree

- Binary Tree 순회 구현
  - ✓ LinkedTree 에 메서드 추가

```
public void preorder(TreeNode root){
        if(root != null){
                System.out.print(root.data + "   ");
                preorder(root.left);
                preorder(root.right);
        }
}
```

Binary Tree

- Binary Tree 순회 구현
  - ✓ LinkedTree 에 메서드 추가

```
public void inorder(TreeNode root){
        if(root != null){
                inorder(root.left);
                System.out.print(root.data + "   ");
                inorder(root.right);
        }
}
```

Binary Tree

- Binary Tree 순회 구현
  - ✓ LinkedTree 에 메서드 추가

```
public void postorder(TreeNode root){
        if(root != null){
                postorder(root.left);

                postorder(root.right);
                System.out.print(root.data + "   ");
        }
}
```

Binary Tree

- Binary Tree 순회 구현
  - ✓ LinkedTreeMain 수정

```
public class LinkedTreeMain {

    public static void main(String[] args) {
        LinkedTree tree = new LinkedTree(100);

        TreeNode root = tree.getRootNode();
        tree.insertLeftChildNode(root, 200);
        tree.insertRightChildNode(root, 300);

        TreeNode node = tree.getLeftChildNode(root);
        tree.insertLeftChildNode(node, 400);
        tree.insertRightChildNode(node, 500);
```

Binary Tree

- Binary Tree 순회 구현
  - ✓ LinkedTreeMain 수정

```
                tree.inorder(root);
                System.out.println();

                tree.preorder(root);
                System.out.println();

                tree.postorder(root);
                System.out.println();
        }
    }
```

Binary Tree

- Binary Tree 노드 개수 구해주는 메서드 구현
  - ✓ LinkedTree 에 노드의 개수와 단말 노드 개수를 구해주는 메서드 추가

```
//노드 개수를 세는 메서드
public int getNodeCount()
{
        int ret = getNodeCountNode(root);
        return ret;
}

private int getNodeCountNode(TreeNode node)
{
        int ret = 0;
        if(node != null) {
                ret = getNodeCountNode(node.left) +
getNodeCountNode(node.right) + 1;
        }
        return ret;
}
```

Binary Tree

- Binary Tree 노드 개수 구해주는 메서드 구현
  - ✓ LinkedTree 에 노드의 개수와 단말 노드 개수를 구해주는 메서드 추가

```
//단말 노드 개수를 세주는 메서드
public int getLeafNodeCount()
{
        int ret = 0;
        if (root != null) {
                ret = getLeafNodeCountNode(root);
        }
        return ret;
}
```

Binary Tree

- Binary Tree 노드 개수 구해주는 메서드 구현
  - ✓ LinkedTree 에 노드의 개수와 단말 노드 개수를 구해주는 메서드 추가

```
private int getLeafNodeCountNode(TreeNode pSource)
{
        int ret = 0;
        if (pSource != null) {
                if (pSource.left == null && pSource.right == null) {
                        ret = 1;
                }
                else {
                        ret = getLeafNodeCountNode(pSource.left) +
getLeafNodeCountNode(pSource.right);
                }
        }
        return ret;
}
```

Binary Tree

- Binary Tree 순회 구현
  - ✓ LinkedTreeMain 클래스의 main 메서드 하단에 추가

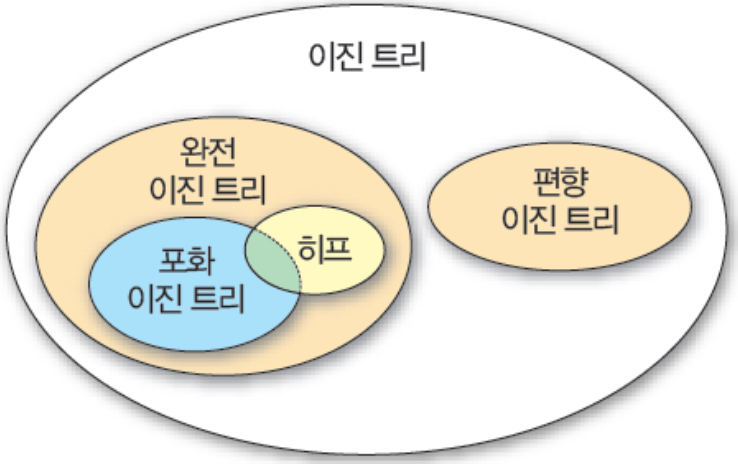
```
System.out.println("노드 개수:" + tree.getNodeCount() + "");
System.out.println("단말 노드 개수:" + tree.getLeafNodeCount() + "");
```

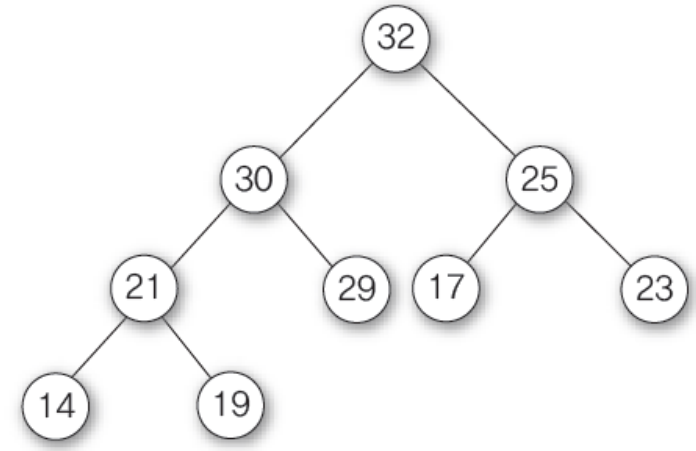
Heap

- Heap
  - ✓ 완전 이진 트리에 있는 노드 중에서 키 값이 가장 큰 노드나 키 값이 가장 작은 노드를 찾기 위해서 만든 자료 구조
  - ✓ 루트 노드가 그 트리의 최댓값 혹은 최솟값을 갖는 트리
  - ✓ 최대 히프(max heap)
    - 키 값이 가장 큰 노드를 찾기 위한 완전 이진 트리
    - {부모 노드의 키 값 ≥ 자식 노드의 키 값}
    - 루트 노드: 키 값이 가장 큰 노드
  - ✓ 최소 히프(min heap)
    - 키 값이 가장 작은 노드를 찾기 위한 완전 이진 트리
    - {부모 노드의 키 값 ≤ 자식 노드의 키 값}
    - 루트 노드: 키값 이 가장 작은 노드

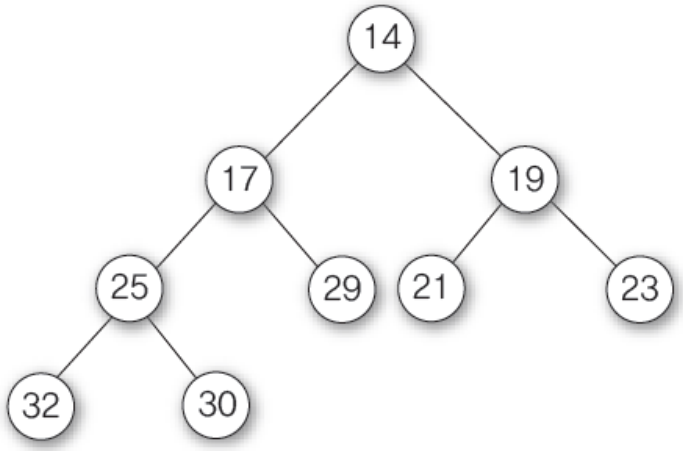
Heap

❖ Heap





(a) 최대 힙



(b) 최소 힙

Heap

- Heap
  - ✓ HEAP의 삽입 연산
    - 트리의 마지막 자리에 임시 저장
    - 부모 노드와 키 값 비교 및 이동
      - 새로 추가한 노드의 키 값이 부모 노드보다 크다면 서로 교환
      - 이 작업을 다시 수행 – 부모 노드보다 작을 때 까지 수행
  - ✓ HEAP의 삭제 연산 – 반드시 루트 노드만 삭제
    - 루트 노드를 삭제
    - 트리의 마지막 노드를 루트로 이동
    - 루트 노드를 자식 노드들과 비교해서 자식 노드가 더 크다면 자식 노드와 위치 변경 – 이 때 2개의 자식 모두 루트보다 크다면 둘 중에 더 큰 노드와 교체
    - 위의 작업을 반복

Heap

- Heap
  - ✓ Heap 클래스

```
class Heap{
    private int heapSize;
    private int itemHeap[];

    public Heap(){
            heapSize = 0;
            itemHeap = new int [50];
    }

    public void insertHeap(int item){
            int i = ++heapSize;
            while((i != 1) && (item > itemHeap[i/2])){
                    itemHeap[i] = itemHeap[i/2];
                    i/=2;
            }
            itemHeap[i] = item;
    }

    public int getHeapSize(){
            return this.heapSize;
    }
```

Heap

- Heap
  - ✓ Heap 클래스

```
public int deleteHeap(){
        int parent, child;
        int item, temp;
        item = itemHeap[1];
        temp = itemHeap[heapSize--];
        parent = 1;    child = 2;

        while(child <= heapSize){
                if((child < heapSize) && (itemHeap[child] < itemHeap[child+1]))
                        child++;
                if(temp >= itemHeap[child]) break;

                itemHeap[parent] = itemHeap[child];
                parent = child;
                child *= 2;
        }
        itemHeap[parent] = temp;
        return item;
}
```

Heap

- Heap
  - ✓ Heap 클래스

```
public void printHeap(){
        System.out.printf("₩nHeap >>> ");
        for(int i=1; i<=heapSize; i++)
                System.out.printf("[%d] ", itemHeap[i]);
    }
}
```

Heap

- Heap
  - ✓ HeapMain 클래스

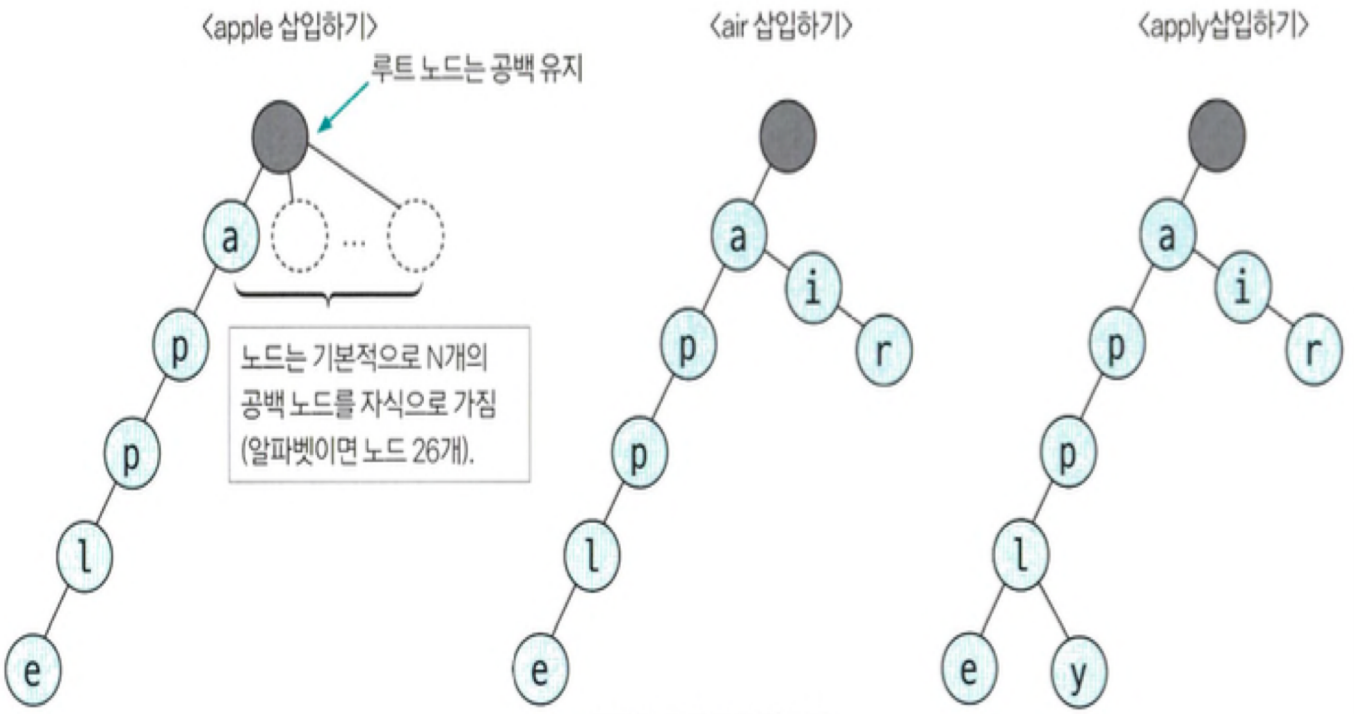
```
public class HeapMain {
    public static void main(String[] args) {
        int n, item;
        Heap h = new Heap();

        h.insertHeap(13);
        h.insertHeap(8);
        h.insertHeap(10);
        h.insertHeap(15);
        h.insertHeap(20);
        h.insertHeap(19);

        h.printHeap();

        n = h.getHeapSize();
        for(int i=1; i<=n; i++){
            item = h.deleteHeap();
            System.out.printf("₩n deleted Item : [%d]", item);
        }
    }
}
```

Trie

- Trie
  - ✓ 문자열 검색을 빠르게 실행할 수 있도록 설계한 트리 형태의 자료구조
  - ✓ 특징
    - N진 트리
      - 문자 종류의 개수에 따라 N이 결정
      - 알파벳은 26개의 문자로 이뤄져 있으므로 26진 트리로 구성
    - 루트 노드는 항상 빈 문자열을 의미하는 공백 상태를 유지

Trie

- Trie
  - ✓ apple, air, apply를 삽입

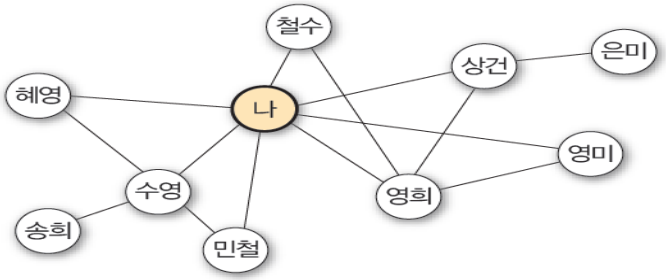


Trie

- Trie
  - ✓ apple, air, apply를 삽입
    - 루트 노드는 공백을 유지하고 apple의 각 알파벳에 해당하는 노드를 생성
    - air를 삽입할 때는 루트 노드에서부터 검색하는데 a 노드는 공백 상태가 아니므로 이동하고 i와 r은 공백 상태이므로 신규 노드를 생성
    - apply를 삽입할 때도 검색 노드가 공백 상태이면 신규 노드를 생성하고 아니면 이동하는 원리로 트라이 자료구조를 구현

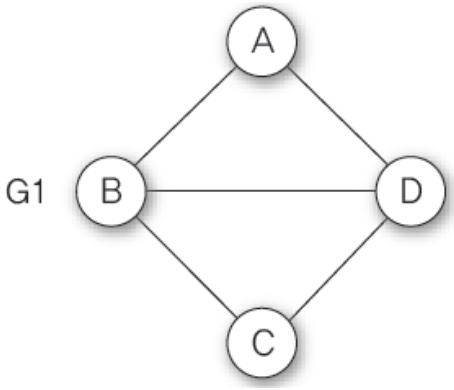
Graph

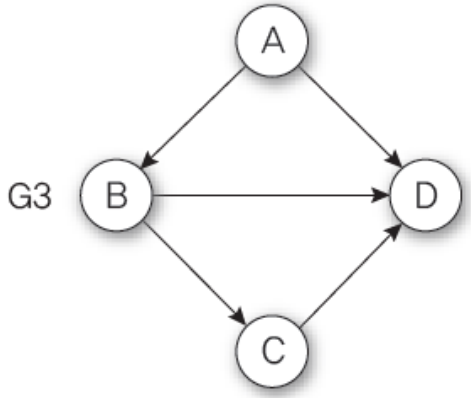
- Graph
  - ✓ 연결되어 있는 원소 사이의 다:다 관계를 표현하는 자료구조
  - ✓ 표현능력이 우수하여 현실 세계의 다양한 문제를 효과적으로 모델링하기 위한 자료구조
  - ✓ 리스트, 스택, 큐 등의 선형자료구조는 데이터를 순차적으로 저장하여 효율적으로 데이터를 저장하는 것이 목적이고 트리는 계층 관계를 표현하기 위한 자료구조인데 트리는 계층 구조가 아닌 일반적인 관계는 나타낼 수 없음
  - ✓ 트리는 순환 구조도 표현할 수 없음
  - ✓ 객체를 나타내는 정점(vertex)과 객체를 연결하는 간선(edge)의 집합
  - ✓ G = (V, E)
    - V는 그래프에 있는 정점들의 집합
    - E는 정점을 연결하는 간선들의 집합

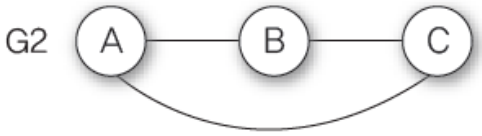


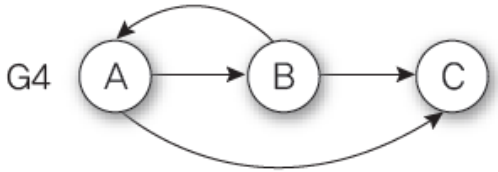
Graph

- Graph
  - ✓ 그래프의 종류
    - 간선의 방향성
      - 무방향 그래프: 간선에 방향이 없는 그래프
      - 방향 그래프: 간선에 방향이 있는 그래프



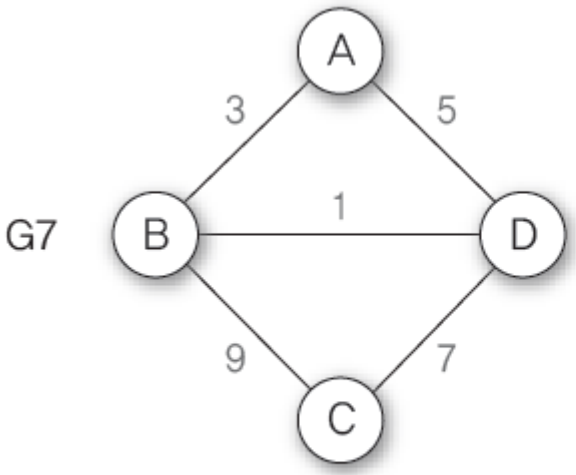


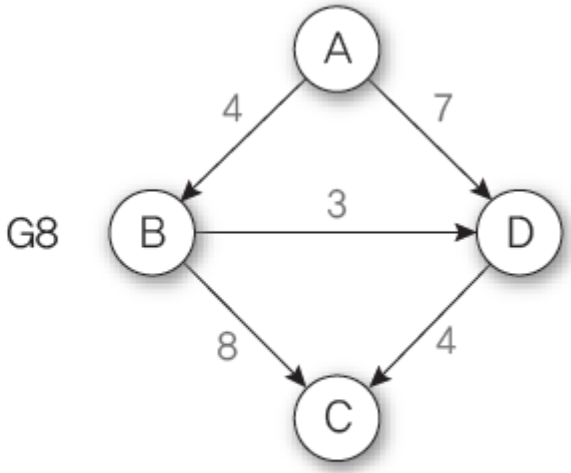




Graph

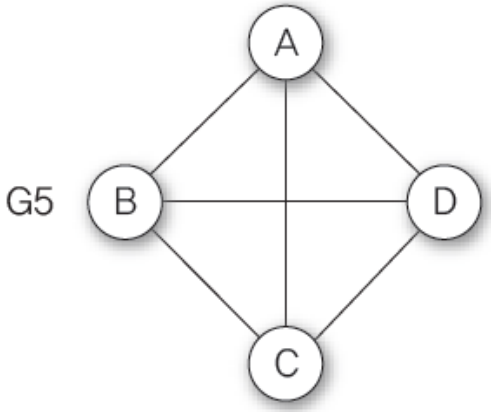
- Graph
  - ✓ 그래프의 종류
    - 가중 그래프: 간선에 가중치가 할당된 그래프

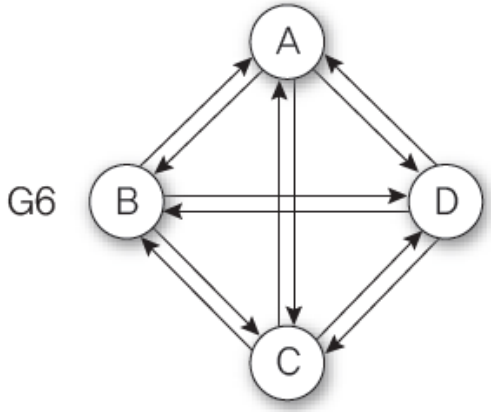




Graph

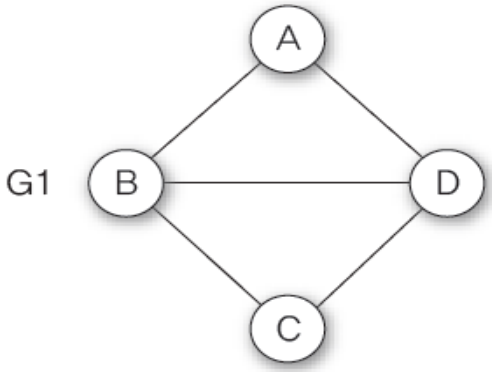
- Graph
  - ✓ 그래프의 종류
    - ● 구조적 특징
      - ◆ 완전 그래프: 연결 가능한 최대 간선 수를 가진 그래프

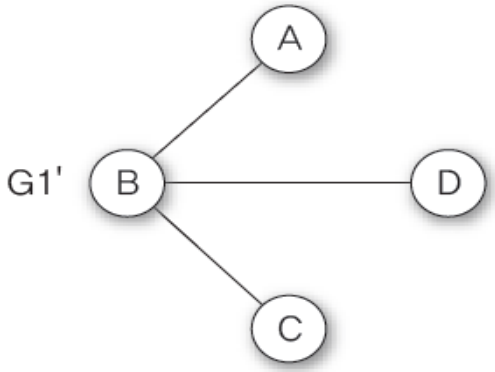


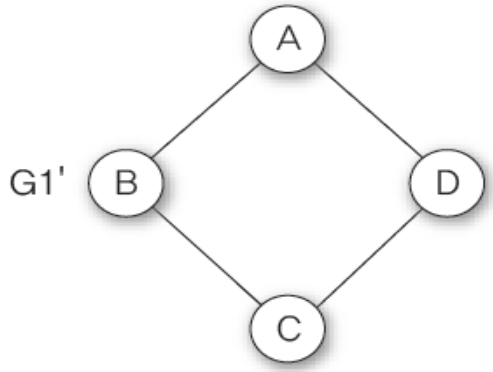


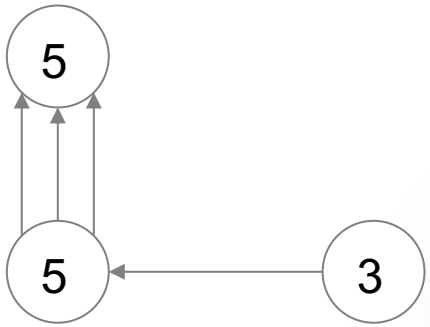
Graph

- Graph
  - ✓ 그래프의 종류
    - 구조적 특징
      - 부분 그래프: 원래의 그래프에서 일부의 노드나 간선을 제외하여 만든 그래프
      - 다중 그래프: 중복된 간선을 포함하는 그래프









Graph

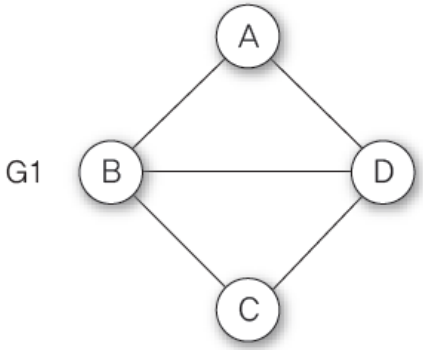
- Graph
  - ✓ 용어
    - Adjacent(인접): 두 개의 노드를 연결하는 간선이 존재하는 경우
    - Incident(부속): 간선 (Vi, Vj)는 정점 Vi와 Vj에 부속(incident)되어 있다고 함
    - Degree(차수): 노드에 부속된 간선의 수
    - Path(경로): 그래프에서 간선을 따라 갈 수 있는 길을 순서대로 나열한 것으로 정점 Vi에서 Vj까지 간선으로 연결된 정점을 순서대로 나열한 리스트
    - 그래프의 동일성: 그래프의 모양은 다르더라도 노드와 간선의 집합이 같으면 동일한 그래프
    - Loop: 그래프의 임의의 노드에서 자기 자신으로 이어지는 간선

Graph

- Graph 구현
  - ✓ 순차 자료구조를 이용한 그래프의 구현: 인접 행렬
    - 행렬에 대한 2차원 배열을 사용하는 순차 자료구조 방법
    - 그래프의 두 정점을 연결한 간선의 유무를 행렬로 저장
    - n개의 정점을 가진 그래프 : n x n 정방 행렬
    - 행렬의 행 번호 와 열 번호 : 그래프의 정점
    - 행렬 값 : 두 정점이 인접되어 있으면 1, 인접되어 있지 않으면 0
    - 무방향 그래프의 인접 행렬
      - 행 i의 합 = 열 i의 합 = 정점 i의 차수
    - 방향 그래프의 인접 행렬
      - 행 i의 합 = 정점 i의 진출 차수
      - 열 i의 합 = 정점 i의 진입 차수

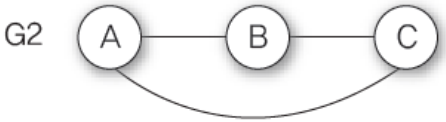
Graph

- Graph 구현
  - ✓ 순차 자료구조를 이용한 그래프의 구현 : 인접 행렬



	A	B	C	D
A	0	1	0	1
B	1	0	1	1
C	0	1	0	1
D	1	1	1	0

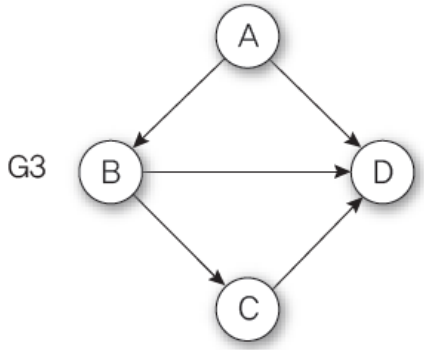
1+0+1+1=3 정점 B의 차수



	A	B	C
A	0	1	1
B	1	0	1
C	1	1	0

Graph

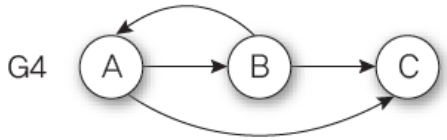
- Graph 구현
  - ✓ 순차 자료구조를 이용한 그래프의 구현 : 인접 행렬



	A	B	C	D
A	0	1	0	1
B	0	0	1	1
C	0	0	0	1
D	0	0	0	0

0+0+1+1=2 정점 B의 진출 차수

1+0+0+0=1 정점 B의 진입 차수



	A	B	C
A	0	1	1
B	1	0	1
C	0	0	0

Graph

- Graph 구현
  - ✓ 순차 자료구조를 이용한 그래프의 구현 – AdjMatrix 클래스

```
class AdjMatrix{
    //배열의 간선을 저장할 배열
    private int matrix[][];
    //그래프의 방향성을 저장할 변수
    private boolean isDirect;
    //그래프의 크기를 저장할 변수
    private int size;

    //생성자 - 그래프의 크기와 방향성을 매개변수로 받아서 생성
    public AdjMatrix(int size, boolean isDirect) {
            matrix = new int[size][size];
            this.isDirect = isDirect;
            this.size = size;
    }
```

Graph

- Graph 구현
  - ✓ 순차 자료구조를 이용한 그래프의 구현 – AdjMatrix 클래스

```
//간선을 추가하는 메서드
public void insertEdge(int v1, int v2){
        if(v1>=size || v2>=size) {
                System.out.println("그래프에 없는 정점입니다!!");
        }
        else {
                if(isDirect == true) {
                        matrix[v1][v2] = 1;
                }else {
                        matrix[v1][v2] = 1;
                        matrix[v2][v1] = 1;
                }
        }
}
```

Graph

- Graph 구현
  - ✓ 순차 자료구조를 이용한 그래프의 구현 – AdjMatrix 클래스

```
//간선을 제거하는 메서드
public void removeEdge(int v1, int v2){
        if(v1>=size || v2>=size) {
                System.out.println("그래프에 없는 정점입니다!!");
        }
        else {
                if(isDirect == true) {
                        matrix[v1][v2] = 0;
                }else {
                        matrix[v1][v2] = 0;
                        matrix[v2][v1] = 0;
                }
        }
}
```

Graph

- Graph 구현
  - ✓ 순차 자료구조를 이용한 그래프의 구현 – AdjMatrix 클래스

```
//배열의 내용을 출력하는 메서드
public void printAdjMarix() {
        for(int i=0; i<size; i++) {
                for(int j=0; j<size; j++) {
                        System.out.print(matrix[i][j] + " ");
                }
                System.out.println();
        }
    }
}
```

Graph

- Graph 구현
  - ✓ 순차 자료구조를 이용한 그래프의 구현 – AdjMatrix 클래스

```
public class AdjMatrixMain {
    public static void main(String[] args) {
        System.out.println("----방향성 그래프----");
        AdjMatrix graph = new AdjMatrix(5, true);
        graph.printAdjMarix();
        System.out.println();

        graph.insertEdge(0, 1);
        graph.insertEdge(2, 3);
        graph.printAdjMarix();
        System.out.println();

        graph.removeEdge(2, 3);
        graph.printAdjMarix();
        System.out.println();
```

Graph

- Graph 구현
  - ✓ 순차 자료구조를 이용한 그래프의 구현 – AdjMatrix 클래스

```
        System.out.println("----무방향성 그래프----");
        graph = new AdjMatrix(5, false);
        graph.printAdjMarix();
        System.out.println();

        graph.insertEdge(0, 1);
        graph.insertEdge(2, 3);
        graph.printAdjMarix();
        System.out.println();

        graph.removeEdge(2, 3);
        graph.printAdjMarix();
        System.out.println();
    }
}
```

Graph

- Graph 구현
  - ✓ 연결 리스트를 이용한 그래프의 구현 : 인접 리스트
    - n개의 정점과 e개의 간선을 가진 무방향 그래프의 인접 리스트
      - 헤드 노드 배열의 크기 : n
      - 연결하는 노드의 수 : 2e
      - 각 정점의 헤드에 연결된 노드의 수 : 정점의 차수
    - n개의 정점과 e개의 간선을 가진 방향 그래프의 인접 리스트
      - 헤드 노드 배열의 크기 : n
      - 연결하는 노드의 수 : e
      - 각 정점의 헤드에 연결된 노드의 수 : 정점의 진출 차수

Graph

- Graph 구현
  - ✓ 연결 리스트를 이용한 그래프의 구현 – GraphNode 클래스

```
class GraphNode {
    int vertex;
    GraphNode link;
}
```

Graph

- Graph 구현
  - ✓ 연결 리스트를 이용한 그래프의 구현 – AdjList 클래스

```
class AdjList {
    private GraphNode head[];;
    private int size;
    private boolean isDirect;

    public AdjList(int size, boolean isDirect) {
            head = new GraphNode[size];
            for (int i = 0; i < size; i++) {
                    head[i] = new GraphNode();
            }
            this.isDirect = isDirect;
            this.size = size;
    }
```

Graph

- Graph 구현
  - ✓ 연결 리스트를 이용한 그래프의 구현 – AdjList 클래스

```
public void insertEdge(int v1, int v2) {
        if (v1 >= size || v2 >= size) {
                System.out.println("그래프에 없는 정점입니다!!");
        } else {
                if (head[v1].link == null) {
                        GraphNode next = new GraphNode();
                        next.vertex = v2;
                        head[v1].link = next;
                } else {
                        GraphNode next = new GraphNode();
                        next.vertex = v2;
                        GraphNode temp = head[v1].link;
                        GraphNode prev = head[v1];
                        boolean flag = false;
```

Graph

- Graph 구현
  - ✓ 연결 리스트를 이용한 그래프의 구현 – AdjList 클래스

```
while (true) {
    if (v2 == temp.vertex) {
        System.out.println("이미 존재하는 간선입니다.");
        return;
    } else if (v2 > temp.vertex) {
        if (temp.link == null) {
            break;
        }
        prev = temp;
        temp = temp.link;
    } else {
        flag = true;
        break;
    }
}
```

Graph

- Graph 구현
  - ✓ 연결 리스트를 이용한 그래프의 구현 – AdjList 클래스

```
                    if (flag == false) {
                        next.link = temp.link;
                        temp.link = next;
                    } else {
                        prev.link = next;
                        next.link = temp;
                    }
                }
```

Graph

- Graph 구현
  - ✓ 연결 리스트를 이용한 그래프의 구현 – AdjList 클래스

```
if (isDirect == false) {
        int imsi = v1;
        v1 = v2;
        v2 = imsi;
        if (head[v1].link == null) {
                GraphNode next = new GraphNode();
                next.vertex = v2;
                head[v1].link = next;
        } else {
                GraphNode next = new GraphNode();
                next.vertex = v2;
                GraphNode temp = head[v1].link;
                GraphNode prev = head[v1];
                boolean flag = false;
```

Graph

- Graph 구현
  - ✓ 연결 리스트를 이용한 그래프의 구현 – AdjList 클래스

```
while (true) {
    if (v2 == temp.vertex) {
        System.out.println("이미 존재하는 간선입니다.");
        return;
    } else if (v2 > temp.vertex) {
        if (temp.link == null) {
            break;
        }
        prev = temp;
        temp = temp.link;
    } else {
        flag = true;
        break;
    }
}
```

Graph

- Graph 구현
  - ✓ 연결 리스트를 이용한 그래프의 구현 – AdjList 클래스

```
                                        if (flag == false) {
                                                next.link = temp.link;
                                                temp.link = next;
                                        } else {
                                                prev.link = next;
                                                next.link = temp;
                                        }
                                }
                        }
                }
        }
```

Graph

- Graph 구현
  - ✓ 연결 리스트를 이용한 그래프의 구현 – AdjList 클래스

```
public void removeEdge(int v1, int v2) {
        if (v1 >= size || v2 >= size) {
                System.out.println("그래프에 없는 정점입니다!!");
        } else {
                if (head[v1].link == null) {
                        System.out.println("간선이 없어서 삭제할 수
없습니다!!");
                } else {
                        GraphNode temp = head[v1].link;
                        GraphNode prev = head[v1];
```

Graph

- Graph 구현
  - ✓ 연결 리스트를 이용한 그래프의 구현 – AdjList 클래스

```
while (true) {
    if (v2 == temp.vertex) {
        prev.link = temp.link;
        temp = null;
        break;
    } else {
        if (temp.link == null) {

System.out.println("존재하지 않는 간선입니다.");
            break;
        }
        prev = temp;
        temp = temp.link;
    }
}
```

Graph

- Graph 구현
  - ✓ 연결 리스트를 이용한 그래프의 구현 – AdjList 클래스

```
if(isDirect == false) {
        int imsi = v1;
        v1 = v2;
        v2 = imsi;

        if (head[v1].link == null) {
                System.out.println("간선이 없어서 삭제할
수 없습니다!!");
        } else {
                temp = head[v1].link;
                prev = head[v1];
```

Graph

- Graph 구현
  - ✓ 연결 리스트를 이용한 그래프의 구현 – AdjList 클래스

```
                while (true) {
                        if (v2 == temp.vertex) {
                                prev.link = temp.link;
                                        temp = null;
                                        return;
                        } else {
                                        if (temp.link == null){
System.out.println("존재하지 않는 간선입니다.");
return;
                }
                                        prev = temp;
                                        temp = temp.link;
```

Graph

- Graph 구현
  - ✓ 연결 리스트를 이용한 그래프의 구현 – AdjList 클래스

```
                                                                    }
                                                                }
                                                            }
                                                        }
                                                    }
                                                }
                                            }
```

Graph

- Graph 구현
  - ✓ 연결 리스트를 이용한 그래프의 구현 – AdjList 클래스

```
public void printAdjList() {
        for (int i = 0; i < size; i++) {
                if (head[i].link == null) {
                        break;
                } else {
                        GraphNode temp = head[i].link;
```

Graph

- Graph 구현
  - ✓ 연결 리스트를 이용한 그래프의 구현 – AdjList 클래스

```
                                        while (true) {
                                                System.out.print("(" + i + "," + temp.vertex
        + ")");
                                                if (temp.link == null) {
                                                        break;
                                                } else {
                                                        System.out.print(",");
                                                        temp = temp.link;
                                                }
                                        }
                                        System.out.println();
                                }
                        }
                }
        }
```

Graph

- Graph 구현
  - ✓ 연결 리스트를 이용한 그래프의 구현 – AdjListMain 클래스

```
public class AdjListMain {
    public static void main(String[] args) {
        System.out.println("---방향성---");
        System.out.println("------간선 추가------");
        AdjList graph = new AdjList(5, true);
        graph.printAdjList();
        graph.insertEdge(0, 0);
        graph.insertEdge(0, 2);
        graph.insertEdge(0, 1);
        graph.insertEdge(1, 2);
        graph.insertEdge(2, 3);
        graph.insertEdge(2, 2);
        graph.insertEdge(2, 1);
        graph.printAdjList();
```

Graph

- Graph 구현
  - ✓ 연결 리스트를 이용한 그래프의 구현 – AdjListMain 클래스

```
System.out.println("------간선 삭제------");
graph.removeEdge(0, 1);
graph.printAdjList();
System.out.println();
```

Graph

- Graph 구현
  - ✓ 연결 리스트를 이용한 그래프의 구현 – AdjListMain 클래스

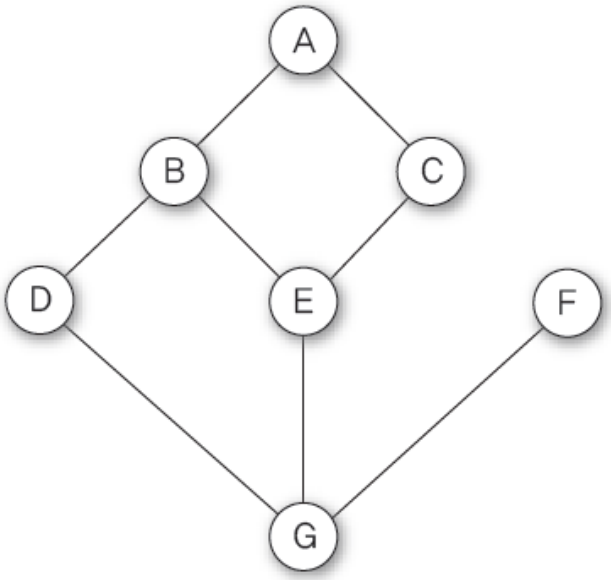
```
                System.out.println("---무방향성---");
                System.out.println("------간선 추가------");
                graph = new AdjList(5, false);
                graph.printAdjList();
                graph.insertEdge(0, 0);
                graph.insertEdge(0, 2);
                graph.insertEdge(0, 1);
                graph.insertEdge(1, 2);
                graph.insertEdge(2, 3);
                graph.insertEdge(2, 2);
                graph.insertEdge(2, 1);
                graph.printAdjList();
        System.out.println("------간선 삭제------");
                graph.removeEdge(0, 1);
                graph.printAdjList();
        }
    }
```

Graph

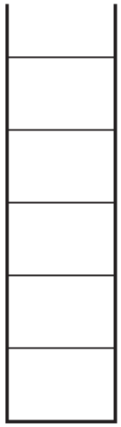
- Graph 탐색
  - ✓ 깊이 우선 탐색(DFS – Depth First Search)
    - 시작 정점의 한 방향으로 갈 수 있는 경로가 있는 곳까지 깊이 탐색해 가다가 더 이상 갈 곳이 없으면, 가장 마지막에 만났던 갈림길 간선이 있는 정점으로 되돌아와 다른 방향의 간선으로 탐색을 계속 반복하여 결국 모든 정점을 방문하는 순회 방법
    - 가장 마지막에 만났던 갈림길 간선의 정점으로 가장 먼저 되돌아가서 다시 깊이 우선 탐색을 반복해야 하므로 후입 선출 구조의 스택 사용

Graph

- Graph 탐색
  - ✓ 깊이 우선 탐색(DFS – Depth First Search)
    - 알고리즘에 따라 그래프 G9를 깊이 우선 순회하는 과정
      - 그래프 G9의 깊이 우선 탐색을 위한 초기 상태: 배열 visited를 false로 초기화하고 공백 스택 생성



G9



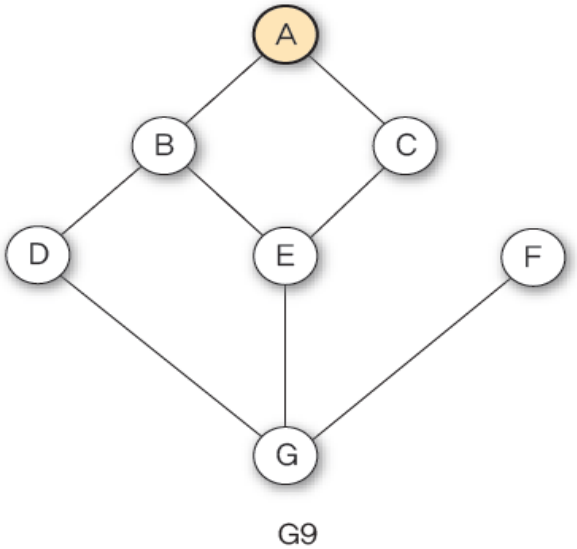
stack

정점	A	B	C	D	E	F	G
	[0]	[1]	[2]	[3]	[4]	[5]	[6]
visited	F	F	F	F	F	F	F

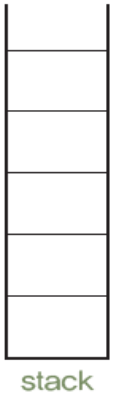
Graph

- Graph 탐색
  - ✓ 깊이 우선 탐색(DFS – Depth First Search)
    - 알고리즘에 따라 그래프 G9를 깊이 우선 순회하는 과정
      - 정점 A를 시작으로 깊이 우선 탐색을 시작

```
visited[A] ← true;
A 방문;
```



G9



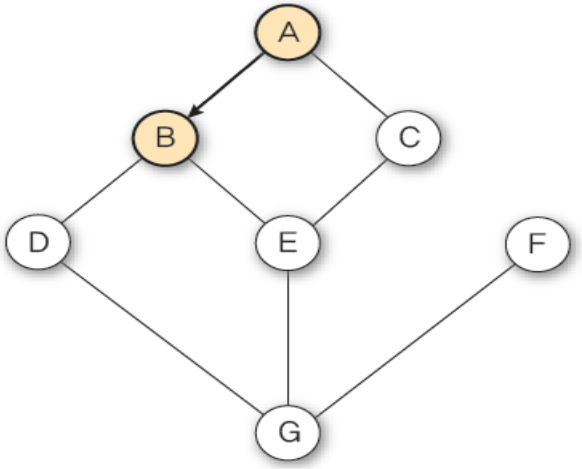
stack

정점	A	B	C	D	E	F	G
	[0]	[1]	[2]	[3]	[4]	[5]	[6]
visited	T	F	F	F	F	F	F

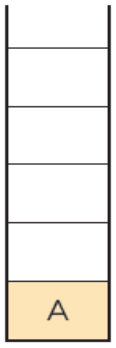
Graph

- Graph 탐색
  - ✓ 깊이 우선 탐색(DFS – Depth First Search)
    - 알고리즘에 따라 그래프 G9를 깊이 우선 순회하는 과정
      - 정점 A에 방문하지 않은 정점 B, C가 있으므로 A를 스택에 push하고 인접 정점 B와 C 중에서 오름차순에 따라 B를 선택하여 탐색을 계속

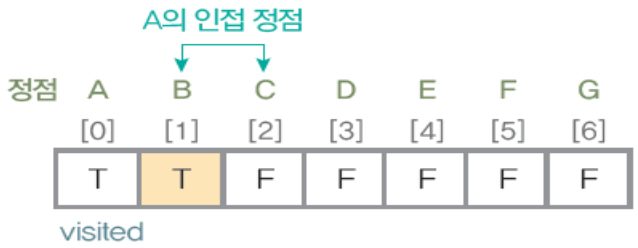
```
push(stack, A);
visited[B] ← true;
B 방문;'
```



G9



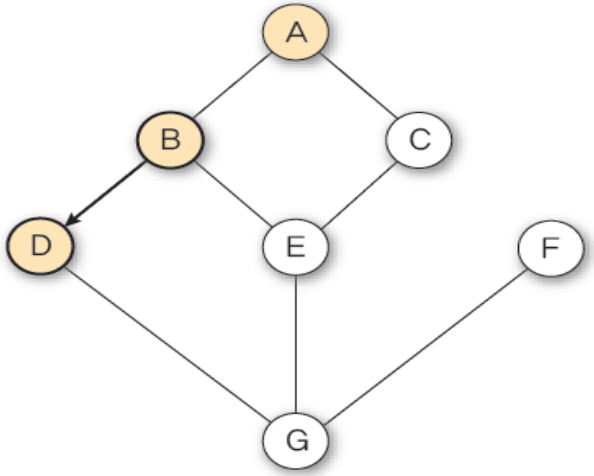
stack

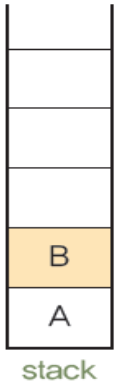


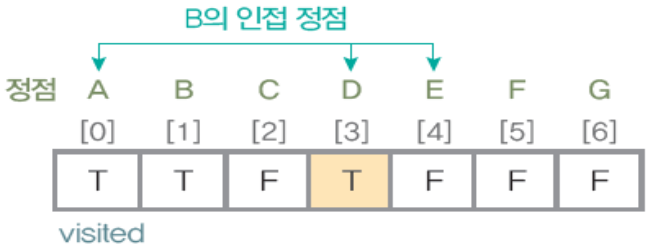
Graph

- ❖ Graph 탐색
  - ✓ 깊이 우선 탐색(DFS – Depth First Search)
    - ● 알고리즘에 따라 그래프 G9를 깊이 우선 순회하는 과정
      - ◆ 정점 B에 방문하지 않은 정점 D, E가 있으므로 B를 스택에 push하고, 방문하지 않은 인접 정점 D와 E 중에서 오름차순에 따라 D를 선택하여 탐색을 계속

```
push(stack, B);
visited[D] ← true;
D 방문;
```



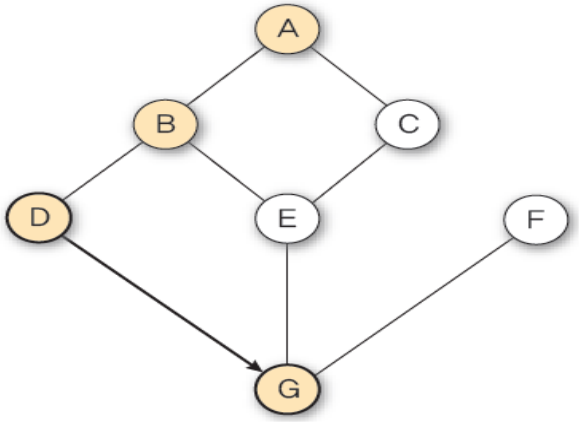


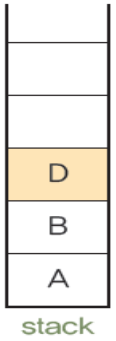


Graph

- ❖ Graph 탐색
  - ✓ 깊이 우선 탐색(DFS – Depth First Search)
    - ● 알고리즘에 따라 그래프 G9를 깊이 우선 순회하는 과정
      - ◆ 정점 D에 방문하지 않은 정점 G가 있으므로 D를 스택에 push하고 인접 정점 G를 선택하여 탐색을 계속

```
push(stack, D);
visited[G] ← true;
G 방문;
```

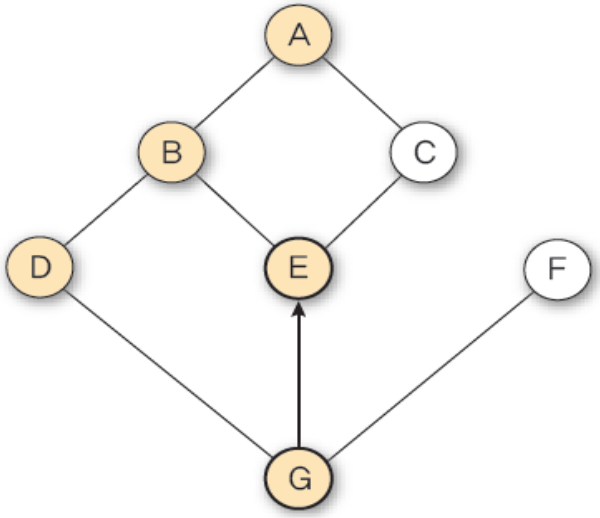


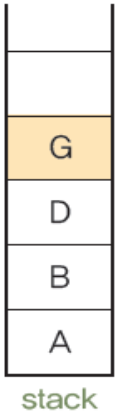


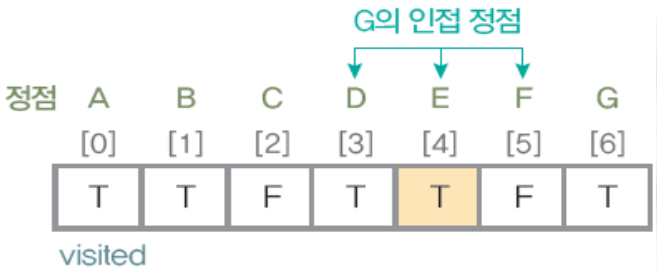
Graph

- Graph 탐색
  - ✓ 깊이 우선 탐색(DFS – Depth First Search)
    - 알고리즘에 따라 그래프 G9를 깊이 우선 순회하는 과정
      - 정점 G에 방문하지 않은 정점 E, F가 있으므로 G를 스택에 push하고 방문하지 않은 인접 정점 E와 F중에서 오름차순에 따라 E를 선택하여 탐색을 계속

```
push(stack, G);
visited[E] ← true;
E 방문;
```



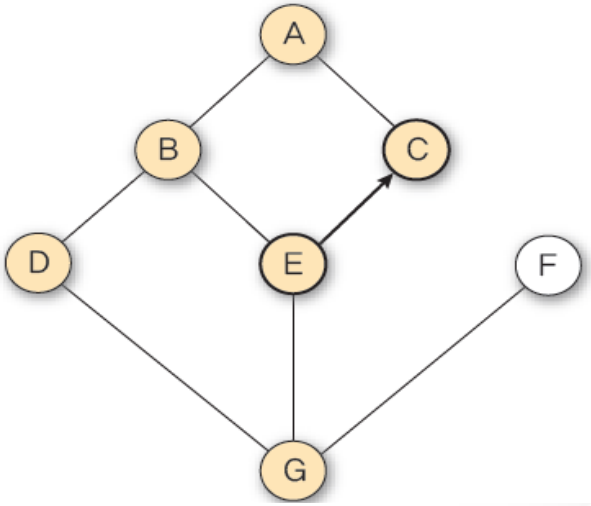


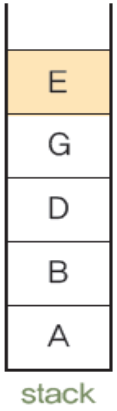


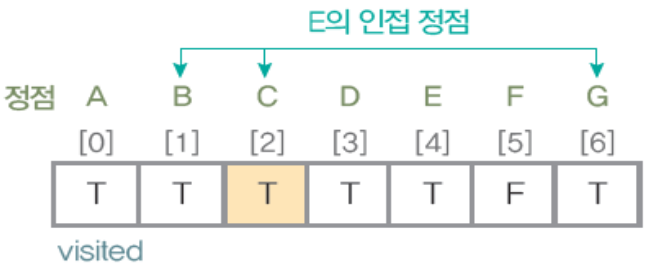
Graph

- ❖ Graph 탐색
  - ✓ 깊이 우선 탐색(DFS – Depth First Search)
    - ● 알고리즘에 따라 그래프 G9를 깊이 우선 순회하는 과정
      - ◆ 정점 E에 방문하지 않은 정점 C가 있으므로 E를 스택에 push하고 방문하지 않은 인접 정점 C를 선택하여 탐색을 계속

```
push(stack, E);
visited[C] ← true;
C 방문;
```



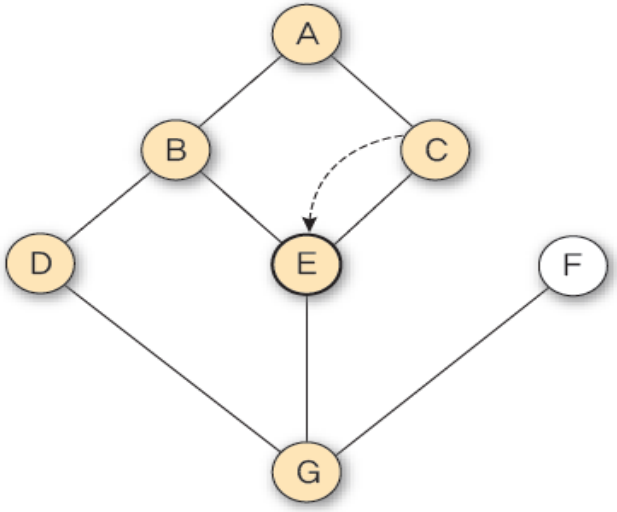


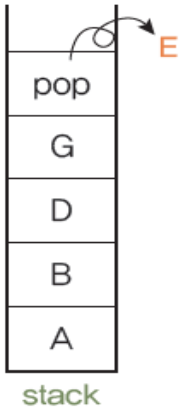


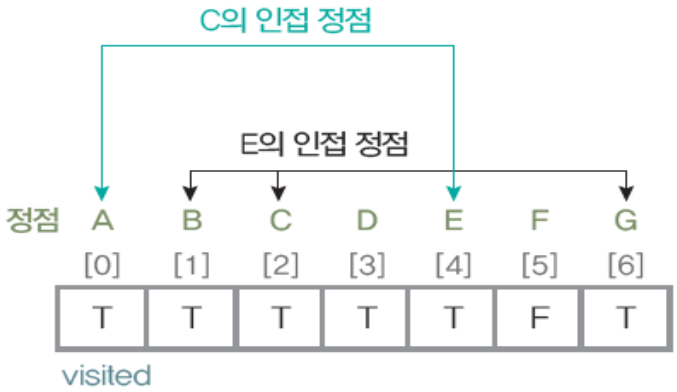
Graph

- Graph 탐색
  - ✓ 깊이 우선 탐색(DFS – Depth First Search)
    - 알고리즘에 따라 그래프 G9를 깊이 우선 순회하는 과정
      - 정점 C에서 방문하지 않은 인접 정점이 없으므로 마지막 정점으로 돌아가기 위해 스택을 pop하여 받은 정점 E에 대해서 방문하지 않은 인접 정점이 있는지 확인
      - 정점 E는 방문하지 않은 인접 정점이 없음

```
pop(stack);
```



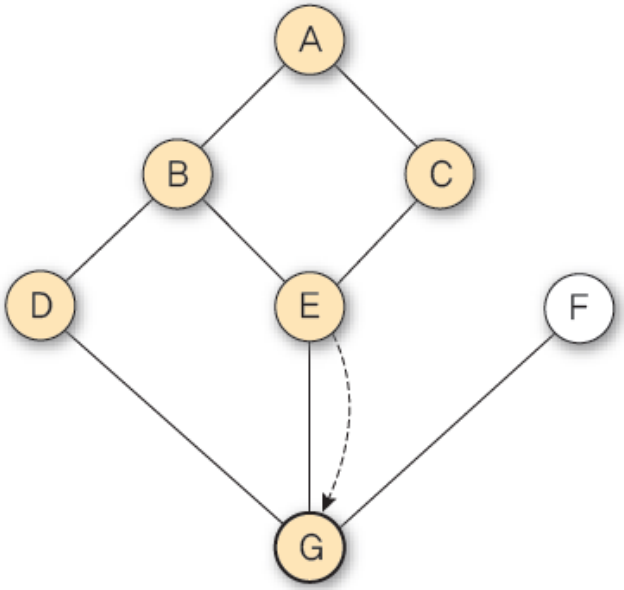


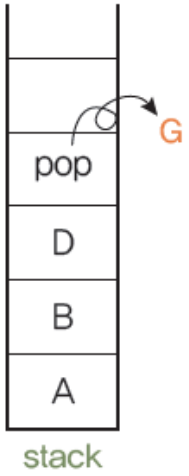


Graph

- Graph 탐색
  - ✓ 깊이 우선 탐색(DFS – Depth First Search)
    - 알고리즘에 따라 그래프 G9를 깊이 우선 순회하는 과정
      - 현재 정점 E에서 방문할 수 있는 인접 정점이 없으므로 다시 스택을 pop하여 받은 정점 G에 대해서 방문하지 않은 인접 정점이 있는지 확인

```
pop(stack);
```



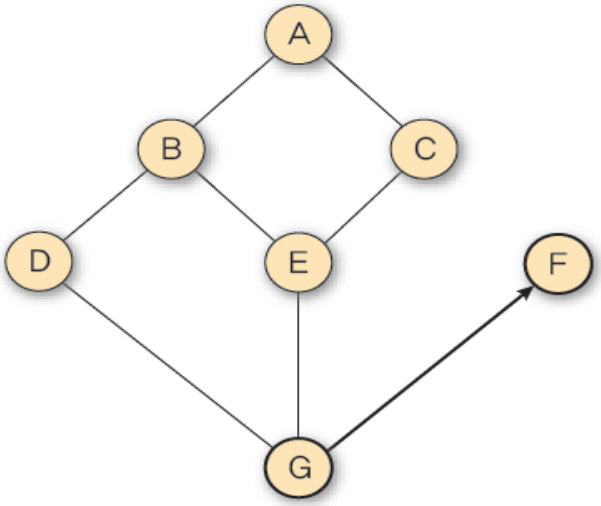


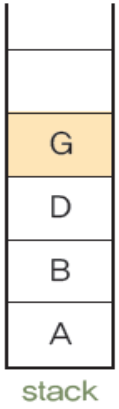
정점	A	B	C	D	E	F	G
	[0]	[1]	[2]	[3]	[4]	[5]	[6]
visited	T	T	T	T	T	F	T

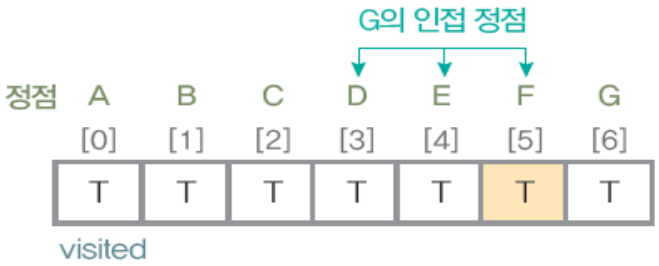
Graph

- Graph 탐색
  - ✓ 깊이 우선 탐색(DFS – Depth First Search)
    - 알고리즘에 따라 그래프 G9를 깊이 우선 순회하는 과정
      - 현재 정점 G에서 방문하지 않은 정점 F가 있으므로 G를 스택에 push하고 인접 정점 F를 선택하여 탐색을 계속

```
push(stack, G);
visited[F] ← true;
F 방문;
```







Graph

- Graph 탐색
  - ✓ 깊이 우선 탐색(DFS – Depth First Search) 구현 – Node 와 Stack 클래스

```
class GraphNode {
    int vertex;
    GraphNode link;
}

class StackNode{
    int data;
    StackNode link;
}
```

Graph

- Graph 탐색
  - ✓ 깊이 우선 탐색(DFS – Depth First Search) 구현 – DFS 메서드를 AdjList 클래스에 추가

```
public void DFS(int v){
        GraphNode w = new GraphNode();
        LinkedStack S = new LinkedStack();;
        boolean visited[] = new boolean[size];
        S.push(v);
        visited[v] = true;
        System.out.print(v + " ");
```

Graph

- Graph 탐색
  - ✓ 깊이 우선 탐색(DFS – Depth First Search) 구현 – DFS 메서드를 AdjList 클래스에 추가

```
            while(S.top != null){
                    w = head[v];
                    while(w != null){
                            if(visited[w.vertex]==false){
                                    S.push(w.vertex);
                                    visited[w.vertex]=true;
                                    System.out.print(w.vertex +" ");
                                    v = w.vertex;
                                    w = head[v];
                            }
                            else w = w.link;
                    }
                    v = S.pop();
            }
            System.out.println();
    }
```

Graph

- Graph 탐색
  - ✓ 깊이 우선 탐색(DFS – Depth First Search) 구현 – main 메서드에 추가

```
System.out.println("---깊이 우선 탐색---");
graph.DFS(0);
```

Graph

- Graph 탐색
  - ✓ 깊이 우선 탐색(DFS – Depth First Search) 구현 – main 메서드에 추가
    - 깊이 우선 탐색은 실제 구현 시 재귀 함수를 이용하므로 스택 오버플로에 유의
    - 깊이 우선 탐색을 응용하여 풀 수 있는 문제는 단절점 찾기, 단절선 찾기, 사이클 찾기, 위상 정렬 등이 있음

Graph

❖ Graph 탐색

✓ 너비 우선 탐색(BFS – Breadth First Search)

❶ 시작 정점 v를 결정하여 방문한다.

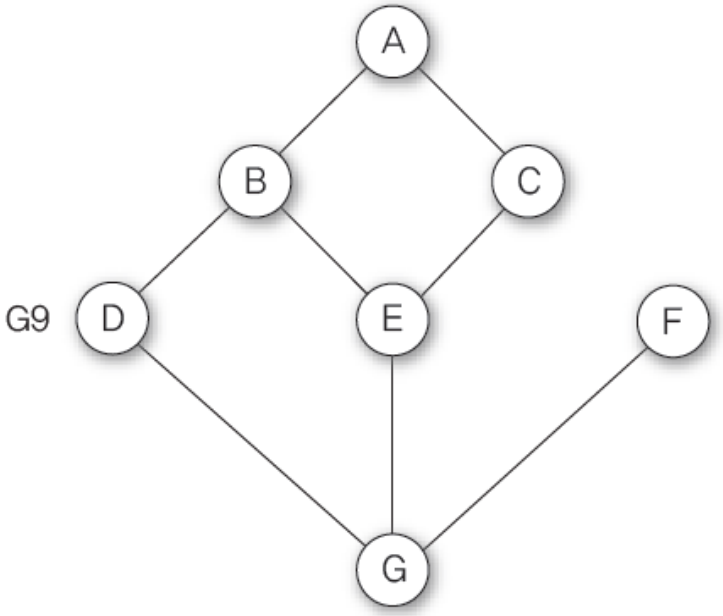
❷ 정점 v에 인접한 정점 중에서 방문하지 않은 정점을 차례로 방문하면서 큐에 enQueue한다.

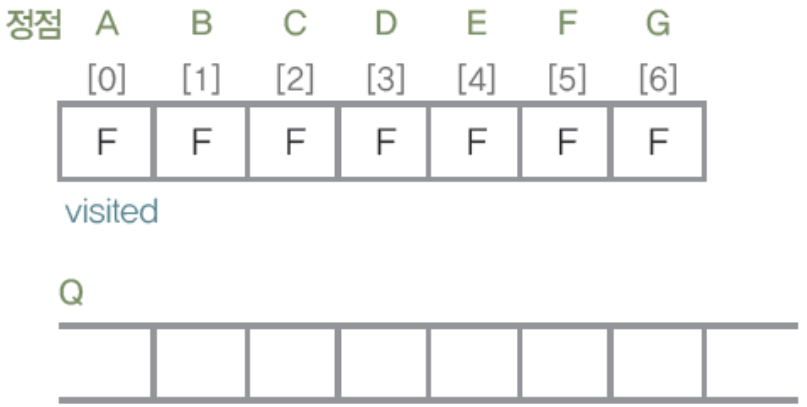
❸ 방문하지 않은 인접한 정점이 없으면, 방문했던 정점에서 인접한 정점을 다시 차례로 방문하기 위해 큐에서 deQueue 하여 받은 정점을 v로 설정하고 ❷를 반복한다.

❹ 큐가 공백이 될 때까지 ❷~❸을 반복한다.

Graph

- ❖ Graph 탐색
  - ✓ 너비 우선 탐색(BFS – Breadth First Search)
    - ● 알고리즘에 따라 그래프 G9를 너비 우선 순회하는 과정
      - ◆ 초기 상태: 배열 visited를 False로 초기화, 공백 큐를 생성

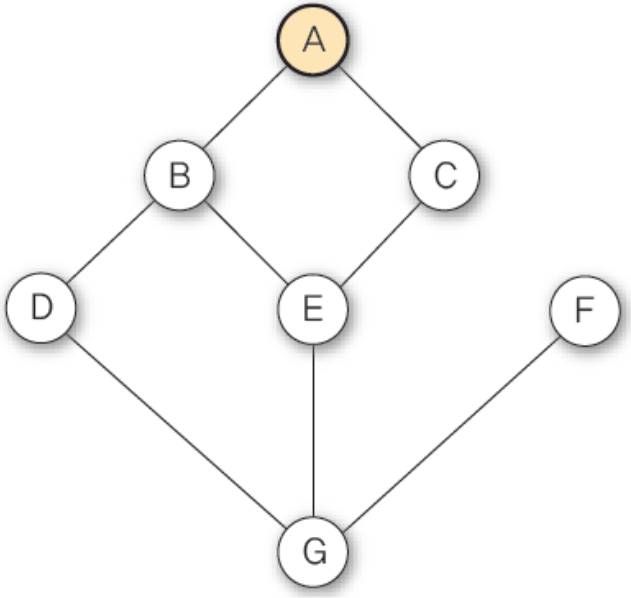


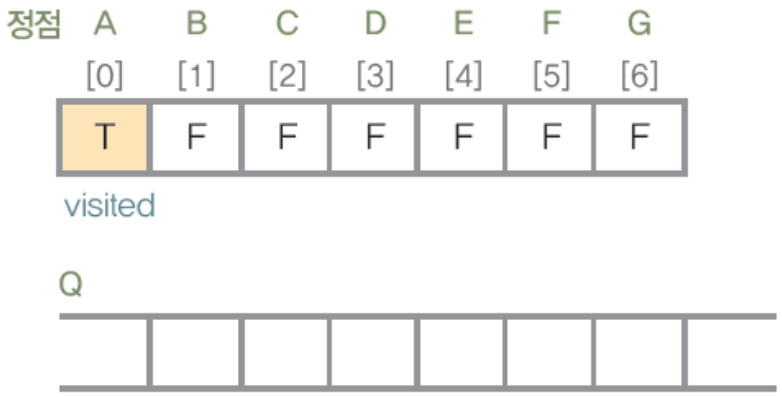


Graph

- Graph 탐색
  - ✓ 너비 우선 탐색(BFS – Breadth First Search)
    - 알고리즘에 따라 그래프 G9를 너비 우선 순회하는 과정
      - 정점 A를 시작으로 너비 우선 탐색을 시작

```
visited[A] ← true;
A 방문;
```

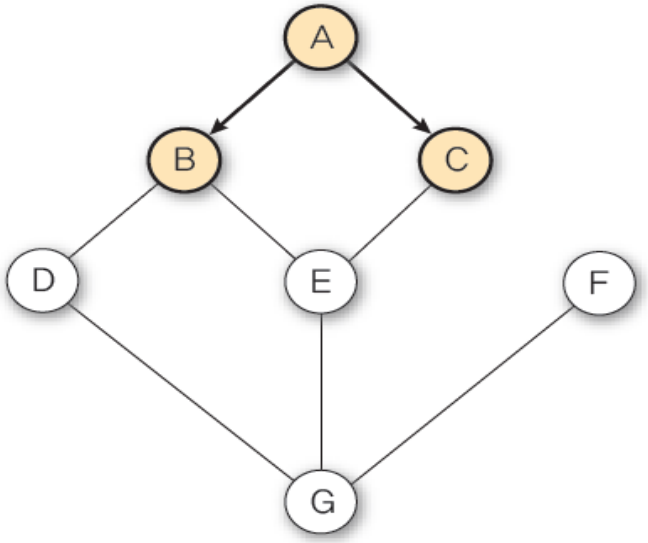


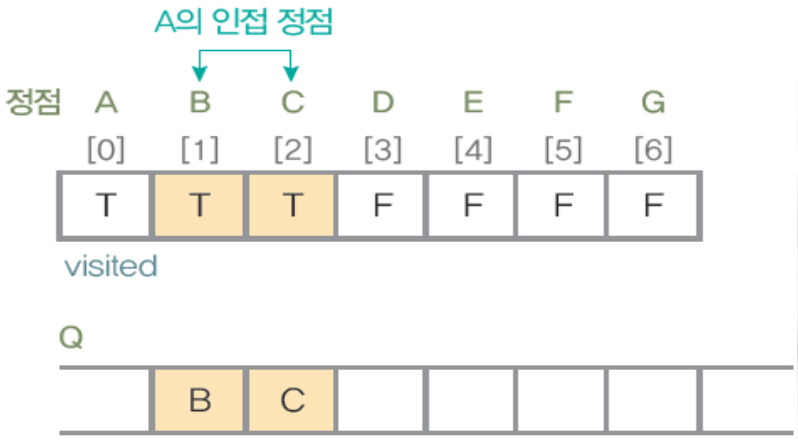


Graph

- Graph 탐색
  - ✓ 너비 우선 탐색(BFS – Breadth First Search)
    - 알고리즘에 따라 그래프 G9를 너비 우선 순회하는 과정
      - 정점 A에서 방문하지 않은 모든 인접 정점 B, C를 방문하고 큐에 enQueue

```
visited[(A가 방문하지 않은 인접 정점 B와 C)] ← true;
(A가 방문하지 않은 인접 정점 B와 C) 방문;
enQueue(Q, (A가 방문하지 않은 인접 정점 B와 C));
```

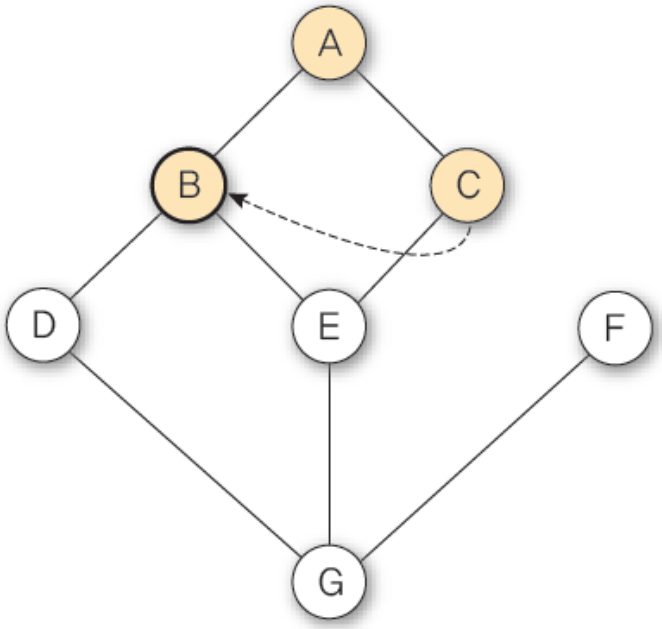


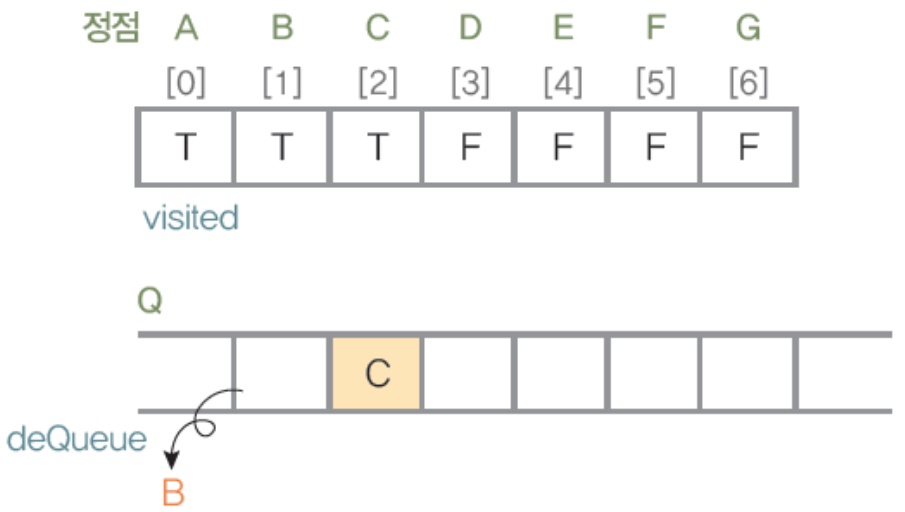


Graph

- Graph 탐색
  - ✓ 너비 우선 탐색(BFS – Breadth First Search)
    - 알고리즘에 따라 그래프 G9를 너비 우선 순회하는 과정
      - 정점 A에 대한 인접 정점들을 처리했으므로 너비 우선 탐색을 계속할 다음 정점을 찾기 위해 큐를 deQueue하여 B를 받음

```
v ← deQueue(Q);
```

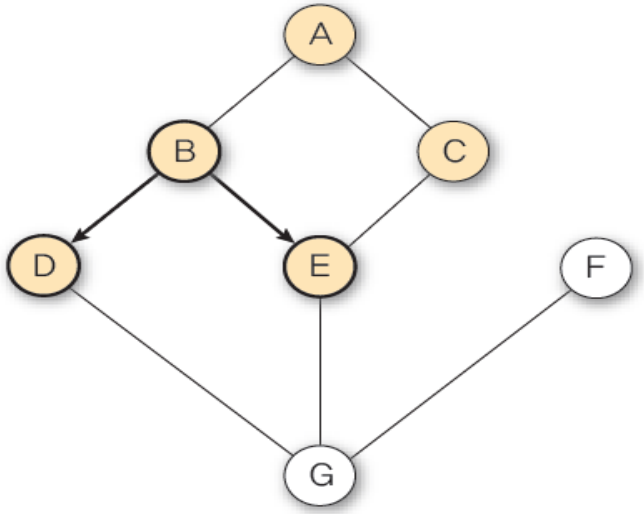


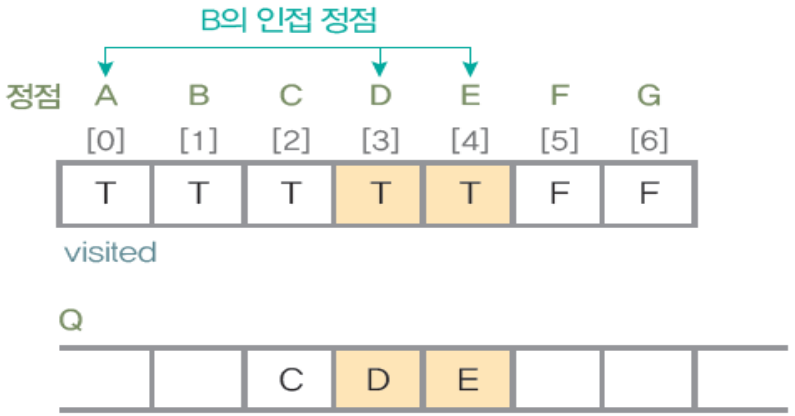


Graph

- Graph 탐색
  - ✓ 너비 우선 탐색(BFS – Breadth First Search)
    - 알고리즘에 따라 그래프 G9를 너비 우선 순회하는 과정
      - 정점 B에서 방문하지 않은 모든 인접 정점 D, E를 방문하고 큐에 enQueue

```
visited[(B가 방문하지 않은 인접 정점 D와 E)] ← true;
(B가 방문하지 않은 인접 정점 D와 E) 방문;
enQueue(Q, (B가 방문하지 않은 인접 정점 D와 E));
```

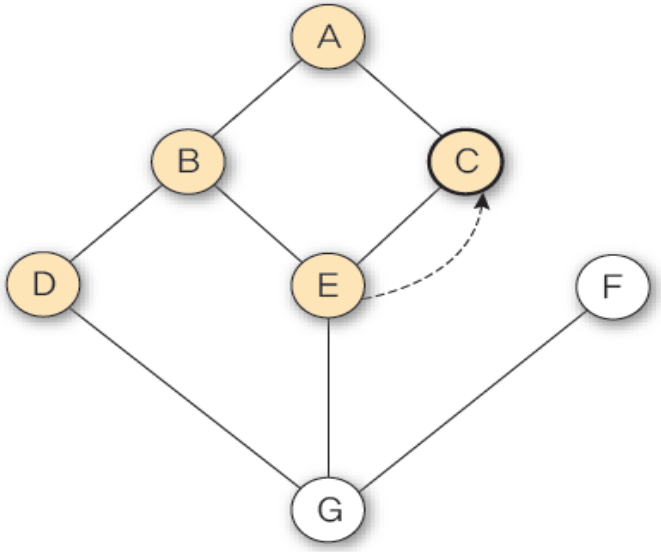


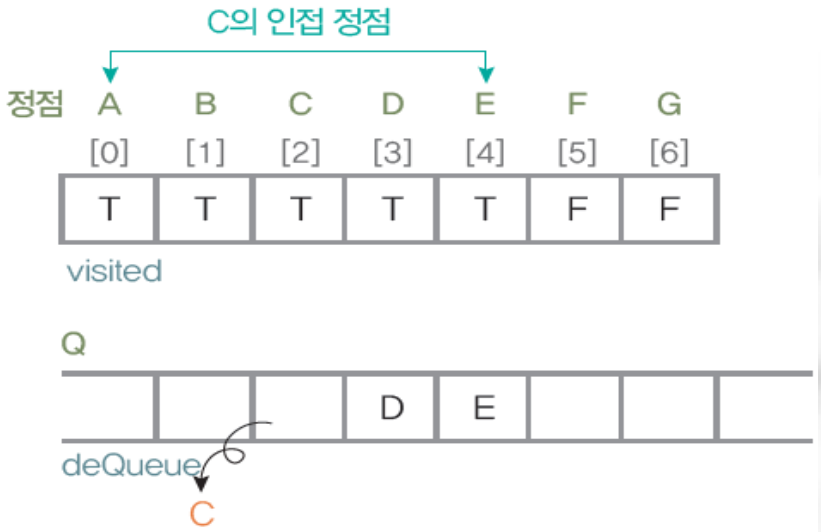


Graph

- Graph 탐색
  - ✓ 너비 우선 탐색(BFS – Breadth First Search)
    - 알고리즘에 따라 그래프 G9를 너비 우선 순회하는 과정
      - 정점 B에 대한 인접 정점들을 처리했으므로 너비 우선 탐색을 계속할 다음 정점을 찾기 위해 큐를 deQueue하여 C를 받음

```
v ← deQueue(Q);
```

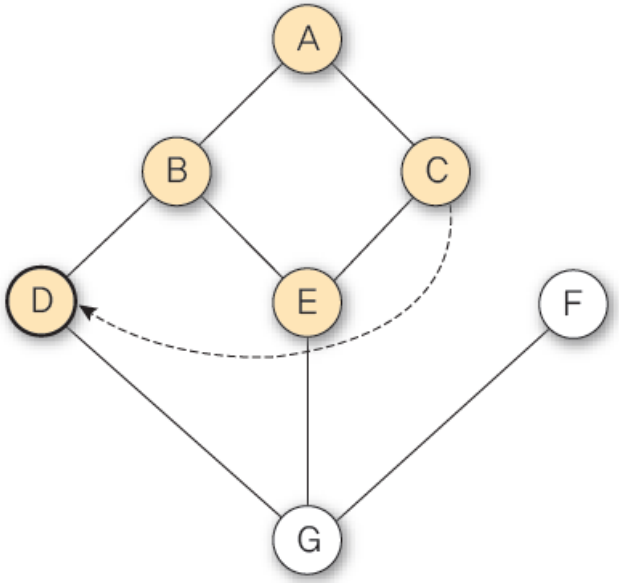


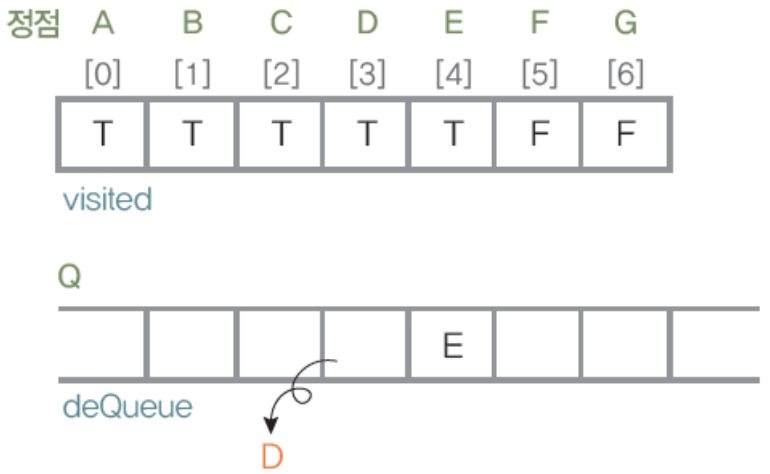


Graph

- ❖ Graph 탐색
  - ✓ 너비 우선 탐색(BFS – Breadth First Search)
    - ● 알고리즘에 따라 그래프 G9를 너비 우선 순회하는 과정
      - ◆ 정점 C에는 방문하지 않은 인접 정점이 없으므로 너비 우선 탐색을 계속할 다음 정점을 찾기 위해 큐를 deQueue하여 D를 받음

```
v ← deQueue(Q);
```

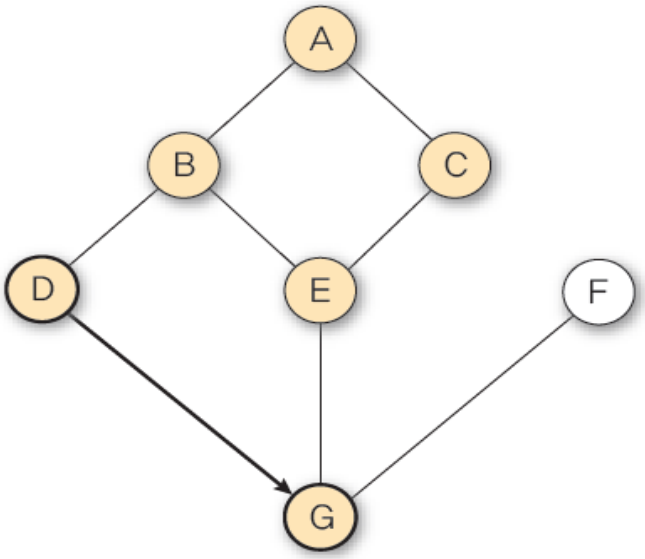


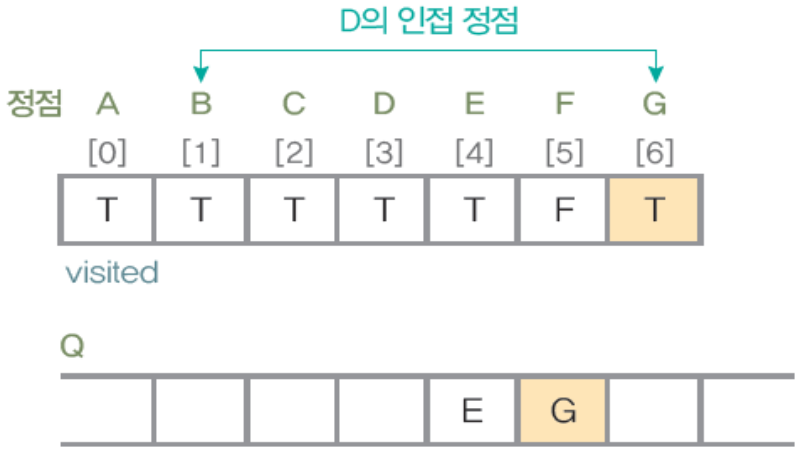


Graph

- Graph 탐색
  - ✓ 너비 우선 탐색(BFS – Breadth First Search)
    - 알고리즘에 따라 그래프 G9를 너비 우선 순회하는 과정
      - 정점 D에서 방문하지 않은 인접 정점 G를 방문하고 큐에 enQueue

```
visited[(D가 방문하지 않은 인접 정점 G)] ← true;
(D가 방문하지 않은 인접 정점 G) 방문;
enQueue(Q, (D가 방문하지 않은 인접 정점 G));
```

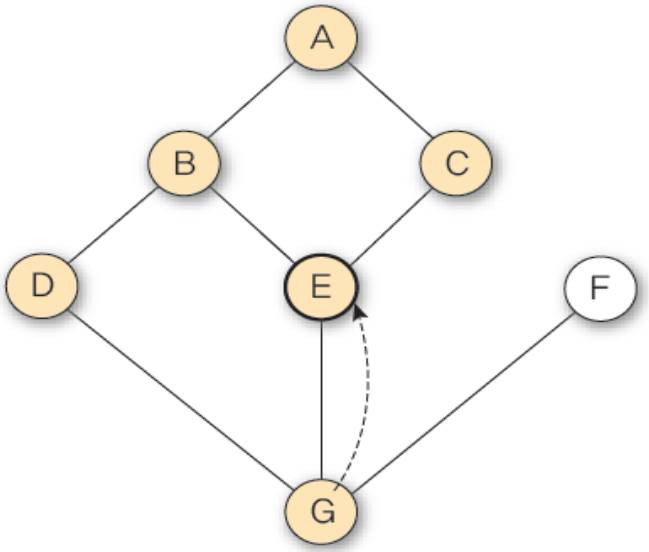


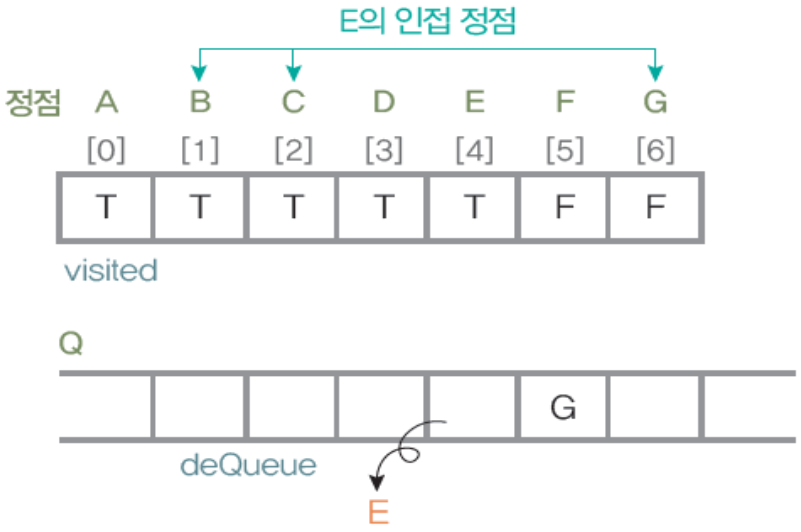


Graph

- Graph 탐색
  - ✓ 너비 우선 탐색(BFS – Breadth First Search)
    - 알고리즘에 따라 그래프 G9를 너비 우선 순회하는 과정
      - 정점 D에 대한 인접 정점들을 처리했으므로 너비 우선 탐색을 계속할 다음 정점을 찾기 위해 큐를 deQueue하여 E를 받음

```
v ← deQueue(Q);
```

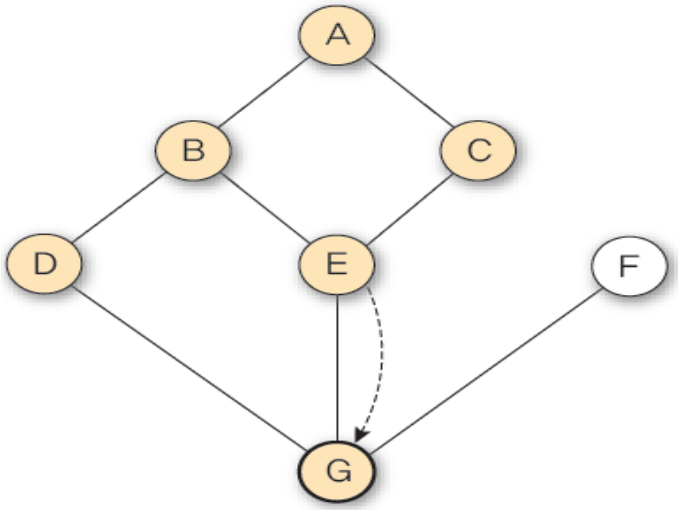


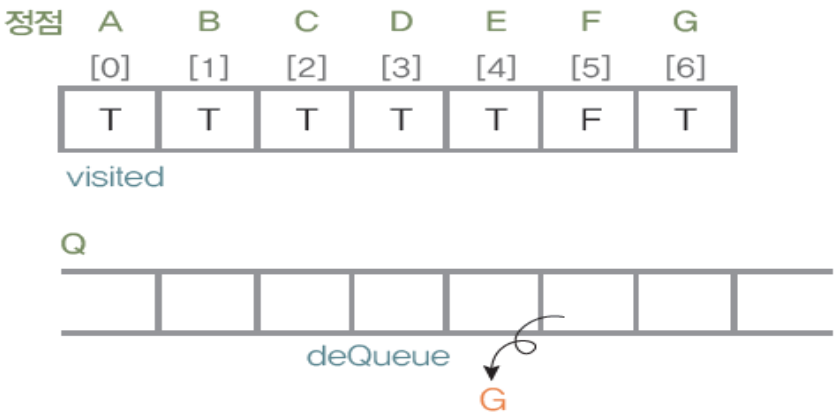


Graph

- Graph 탐색
  - ✓ 너비 우선 탐색(BFS – Breadth First Search)
    - 알고리즘에 따라 그래프 G9를 너비 우선 순회하는 과정
      - 정점 E에는 방문하지 않은 인접 정점이 없으므로 너비 우선 탐색을 계속할 다음 정점을 찾기 위해 큐를 deQueue하여 G를 받음

```
v ← deQueue(Q);
```

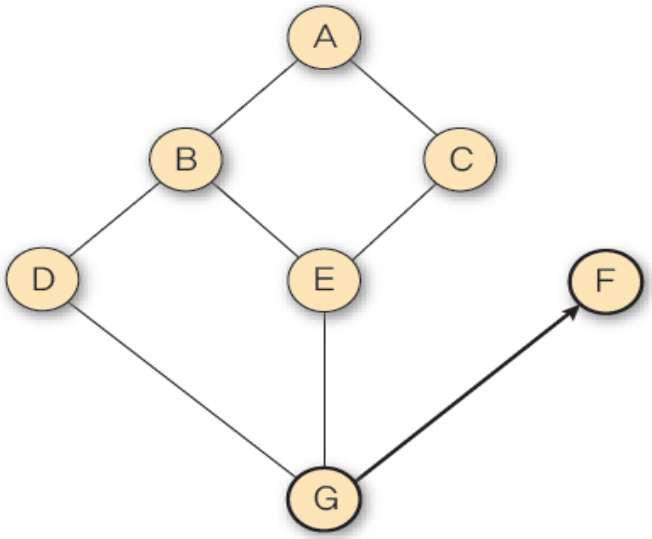




Graph

- Graph 탐색
  - ✓ 너비 우선 탐색(BFS – Breadth First Search)
    - 알고리즘에 따라 그래프 G9를 너비 우선 순회하는 과정
      - 정점 G에서 방문하지 않은 인접 정점 F를 방문하고 큐에 enQueue

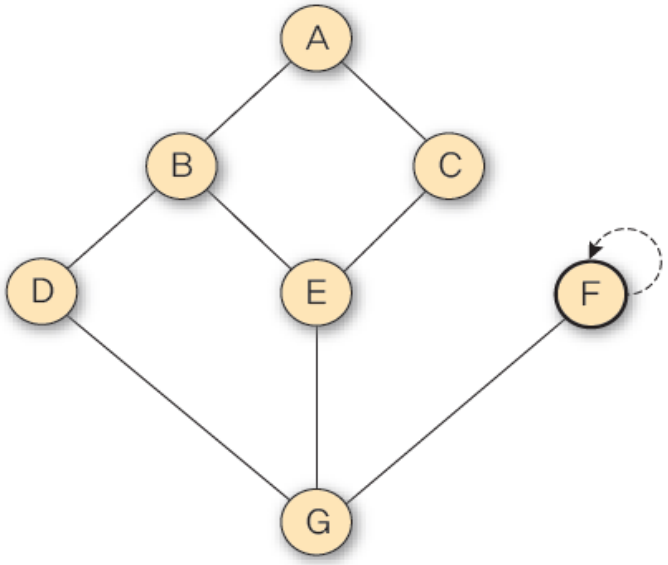
```
visited[(G가 방문하지 않은 인접 정점 F)] ← true;
(G가 방문하지 않은 인접 정점 F) 방문;
enQueue(Q, (G가 방문하지 않은 인접 정점 F));
```

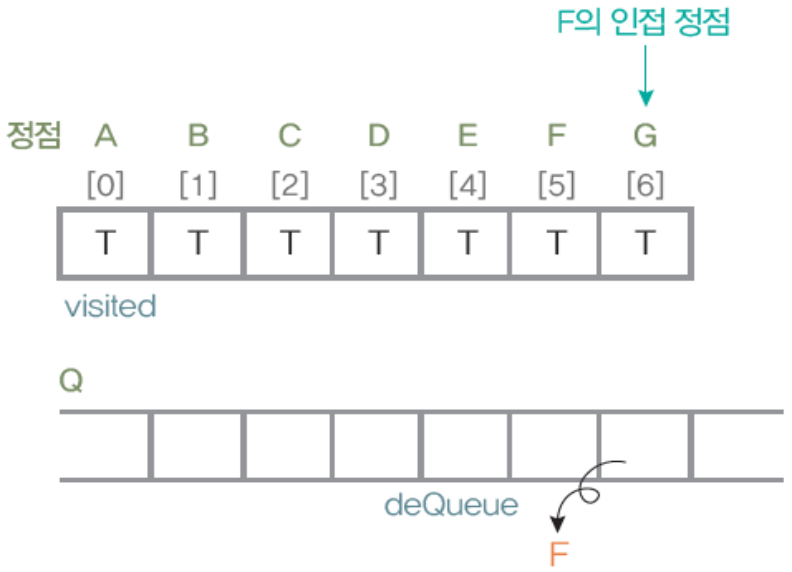


Graph

- ❖ Graph 탐색
  - ✓ 너비 우선 탐색(BFS – Breadth First Search)
    - ● 알고리즘에 따라 그래프 G9를 너비 우선 순회하는 과정
      - ◆ 정점 G에 대한 인접 정점들을 처리했으므로 너비 우선 탐색을 계속할 다음 정점을 찾기 위해 큐를 deQueue하여 F를 받음

```
v ← deQueue(Q);
```





Graph

- Graph 탐색
  - ✓ 너비 우선 탐색(BFS – Breadth First Search)
    - 알고리즘에 따라 그래프 G9를 너비 우선 순회하는 과정
      - 정점 F는 모든 인접 정점을 방문했으므로 너비 우선 탐색을 계속할 다음 정점을 찾기 위해 큐를 deQueue를 하는데 큐가 공백이므로 너비 우선 탐색을 종료

Graph

- Graph 탐색
  - ✓ 너비 우선 탐색(BFS – Breadth First Search) 구현 – Node 클래스

```
class QNode{
    int data;
    QNode link;
}
```

Graph

- Graph 탐색
  - ✓ 너비 우선 탐색(BFS – Breadth First Search) 구현 – LinkedQueue클래스

```
class LinkedQueue{
    QNode front;
    QNode rear;

    public LinkedQueue(){
            front = null;
            rear = null;
    }

    public boolean isEmpty(){
            return (front == null);
    }
```

Graph

- Graph 탐색
  - ✓ 너비 우선 탐색(BFS – Breadth First Search) 구현 – LinkedQueue클래스

```
public void enQueue(int item){
        QNode newNode = new QNode();
        newNode.data = item;
        newNode.link = null;
        if(isEmpty()){
                front = newNode;
                rear = newNode;
        }
        else {
                rear.link = newNode;
                rear = newNode;
        }
}
```

Graph

- Graph 탐색
  - ✓ 너비 우선 탐색(BFS – Breadth First Search) 구현 – LinkedQueue클래스

```
public int deQueue(){
        if(isEmpty()) {
                System.out.println("Deleting fail! Linked Queue is empty!!");
                return 0;
        }
        else{
                int item = front.data;
                front = front.link;
                if(front == null)
                        rear = null;
                return item;
        }
    }
}
```

Graph

- Graph 탐색
  - ✓ 너비 우선 탐색(BFS – Breadth First Search) 구현 – LinkedQueue클래스

```
public void BFS(int v){
        GraphNode w = new GraphNode();
        LinkedQueue Q = new LinkedQueue();
        boolean visited[] = new boolean[10];
        visited[v] = true;
        System.out.print(v + "  ");
        Q.enQueue(v);
```

Graph

- Graph 탐색
  - ✓ 너비 우선 탐색(BFS – Breadth First Search) 구현 – LinkedQueue클래스

```
                while(! Q.isEmpty()){
                        v = Q.deQueue();
                        for(w=head[v]; w != null; w=w.link){
                                if(visited[w.vertex] == false){
                                        visited[w.vertex] = true;
                                        System.out.print(w.vertex + "   ");
                                        Q.enQueue(w.vertex);
                                }
                        }
                }
                System.out.println();
        }
```

Graph

- Graph 탐색
  - ✓ 너비 우선 탐색(BFS – Breadth First Search) 구현 – main 메서드에 추가

```
System.out.println("---너비 우선 탐색---");
graph.BFS(0);
```

Graph

- Graph 탐색
  - ✓ 너비 우선 탐색(BFS – Breadth First Search) 구현 – main 메서드에 추가
    - 너비 우선 탐색은 선입 선출 방식으로 탐색하므로 큐를 이용해 구현
    - 너비 우선 탐 색은 탐색 시작 노드와 가까운 노드를 우선하여 탐색하므로 목표 노드에 도착하는 경로가 여러 개 일 때 최단 경로를 보장