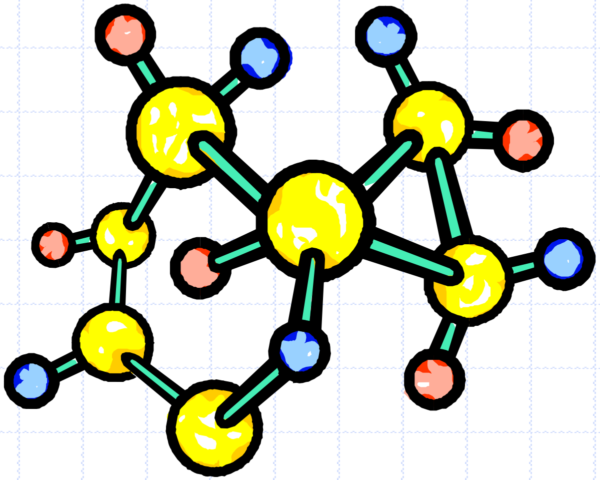
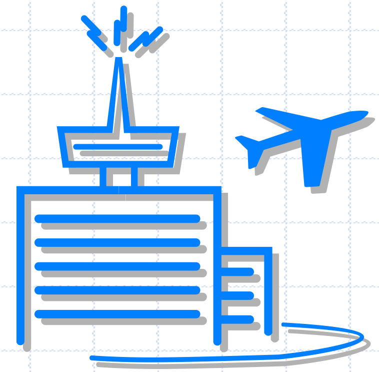
그래프



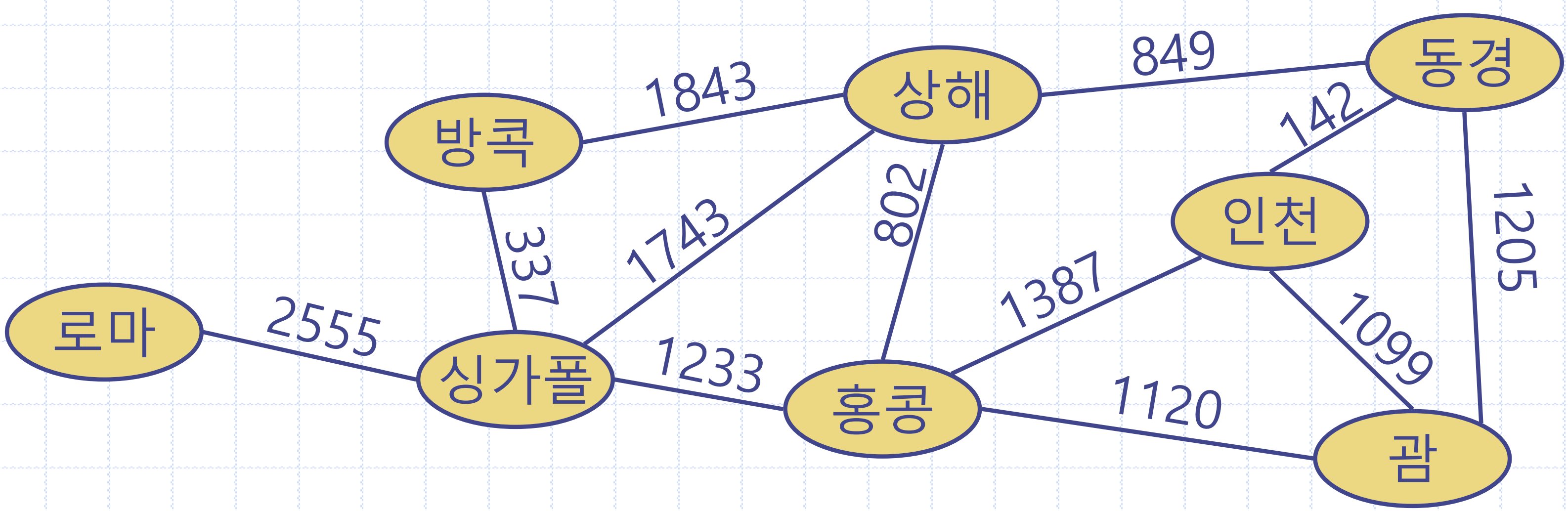
Outline

- 13.1 그래프 ADT
- 13.2 그래프 주요 개념
- 13.3 그래프 ADT 메쏘드
- 13.4 그래프 ADT 구현과 성능
- 13.5 응용문제



그래프 ADT

- **그래프**(graph): (*V*, *E*) 쌍 – 여기서
  - *V*: **정점**(vertex)이라 불리는 노드의 집합
  - *E*: **간선**(edge)이라 불리는 정점쌍들의 집합
  - 정점과 간선은 **원소**, 즉 **정보**를 저장
- **예**
  - 아래 예에서 **정점**은 공항을 표현하며 공항도시 이름을 저장
  - **간선**은 두 공항 사이의 항로를 표현하며 항로의 거리(mile)를 저장



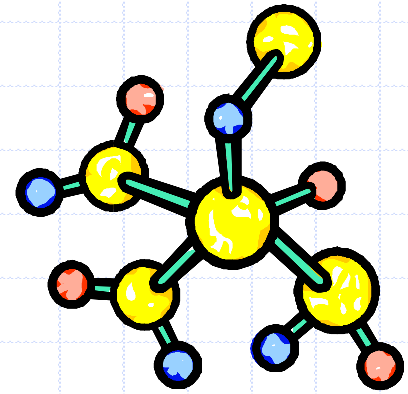
간선에 따른 그래프 유형

- **방향간선**(directed edge)
  - 정점들의 순서쌍 **(*u*, *v*)**
  - ***u***: **시점**(origin)
  - ***v***: **종점**(destination)
  - **예:** 항공편(flight)
- **방향그래프**(directed graph)
  - 모든 간선이 방향간선인 그래프
  - **예:** 항공편망(flight network)
- **무방향간선**(undirected edge)
  - 정점들의 무순쌍 **(*u*, *v*)**
  - **예:** 항로
- **무방향그래프**(undirected graph)
  - 무방향간선으로 이루어진 그래프
  - **예:** 항로망(flight route network)

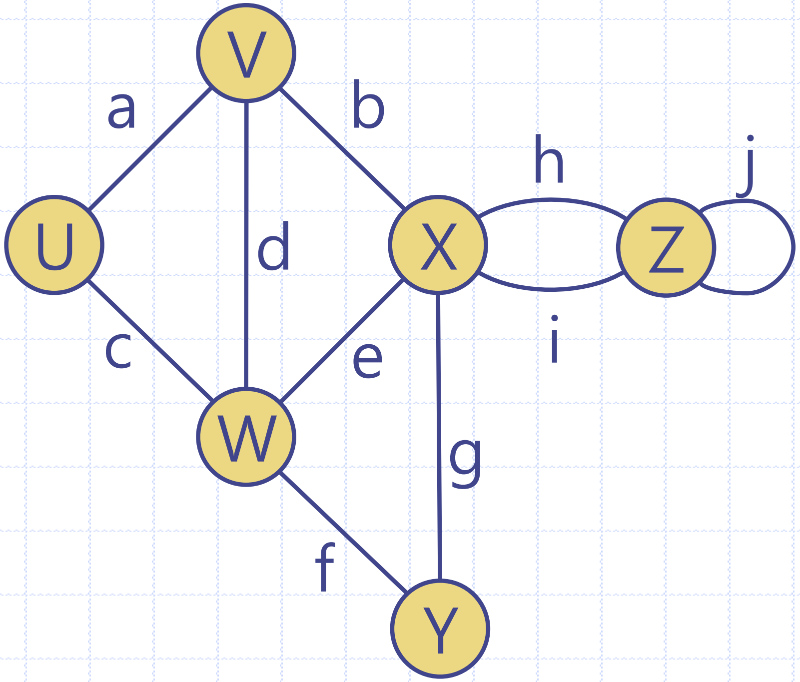
그래프 응용

- 전자회로
  - 인쇄회로기판(printed circuit board, PCB)
  - 집적회로(integrated circuit, IC)
- 교통망
  - 고속도로망
  - 항공노선망
- 컴퓨터 네트워크
  - LAN(local area network)
  - 인터넷
  - 웹
- 데이터베이스
  - 개체-관계 다이어그램(entity-relationship diagram)

그래프 용어

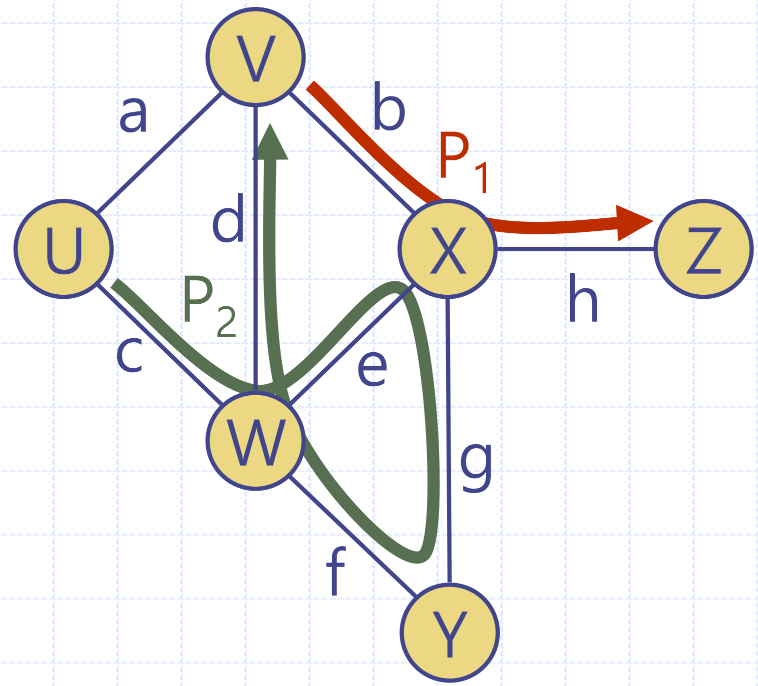


- 간선의 **끝점**(end vertex, 또는 endpoint)
  - 정점 ***U***와 ***V***는 ***a***의 양끝점
- 정점의 **부착**(incident) 간선
  - ***a***, ***d***, ***b***는 ***V***에 부착한다
- 정점의 **인접**(adjacent) 정점
  - ***U***와 ***V***는 인접하다
- 정점의 **차수**(degree)
  - ***X***의 차수는 5다
- **병렬 간선**(parallel edges)
  - ***h***와 ***i***는 병렬 간선
- **루프**(loop 또는 self-loop)
  - ***j***는 루프다



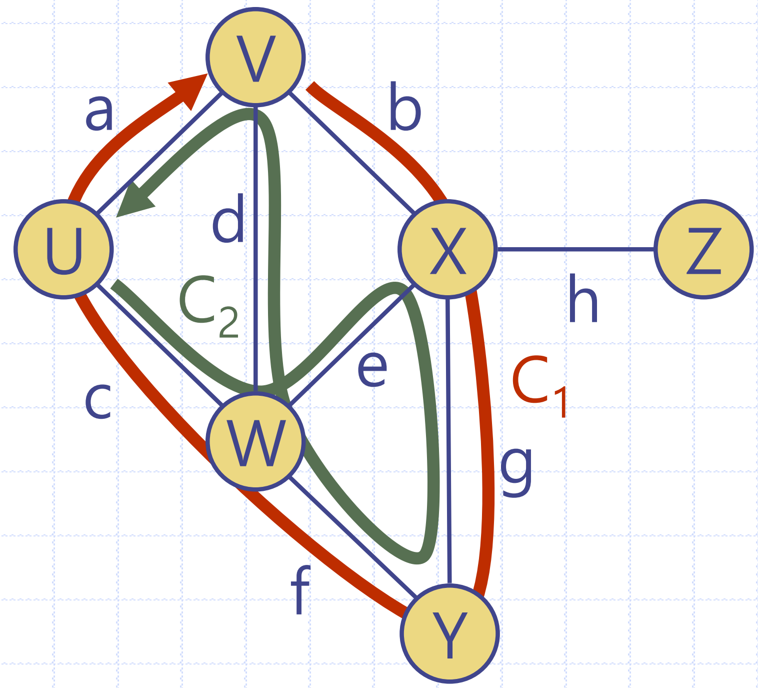
그래프 용어 (conti.)

- **경로**(path)
  - 정점과 간선의 교대열
  - 정점으로 시작하여 정점으로 끝난다
  - 각 간선은 그 양끝점으로 시작하고 끝난다
- **단순경로**(simple path)
  - 모든 정점과 간선이 유일한 경로
- **예**
  - P_1=(V,b,X,h,Z)은 단순경로
  - P_2=(U,c,W,e,X,g,Y,f,W,d,V)는 비단순경로



그래프 용어 (conti.)

- **싸이클**(cycle)
  - 정점과 간선이 교대하는 원형 열
  - 각 간선은 그 양끝점으로 시작하고 끝난다
- **단순싸이클**(simple cycle)
  - 모든 정점과 간선이 유일한 싸이클
- **예**
  - C_1=(V,b,X,g,Y,f,W,c,U,a)는 단순싸이클
  - C_2=(U,c,W,e,X,g,Y,f,W,d,V,a)는 비단순싸이클



속성

속성 1

$\Sigma_v \, deg(v) = 2m$

증명: 각 간선이 두 번 세어진다

속성 2

루프와 병렬 간선이 없는 무방향그래프에서,

$m \le n(n-1)/2$

증명: 각 정점의 최대 차수는 $(n-1)$

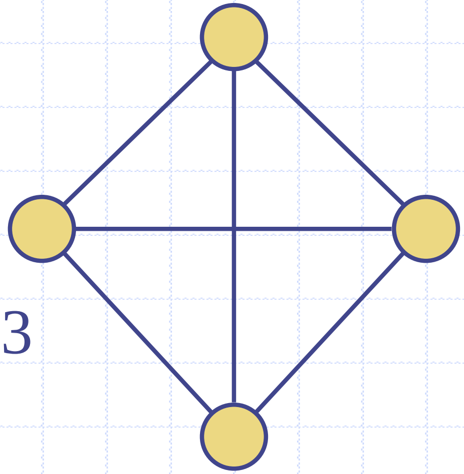
방향그래프에서 m의 상한은?

표기

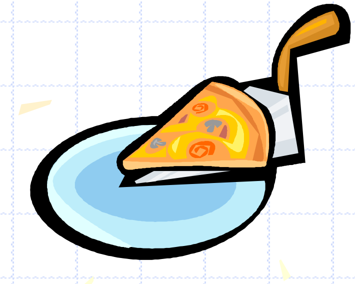
n	정점 수
m	간선 수
$deg(v)$	정점 v의 차수

예

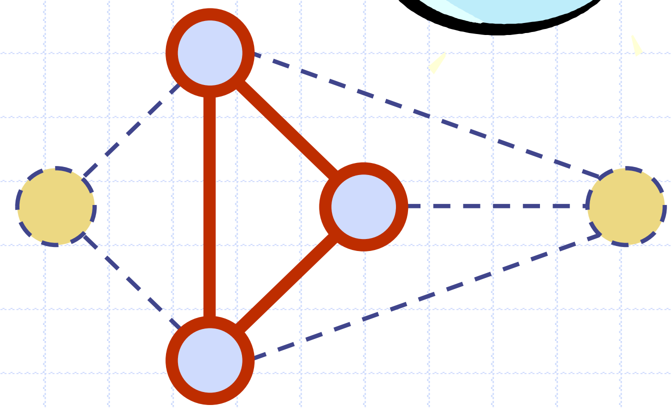
- $n = 4$
- $m = 6$
- $deg(v) = 3$



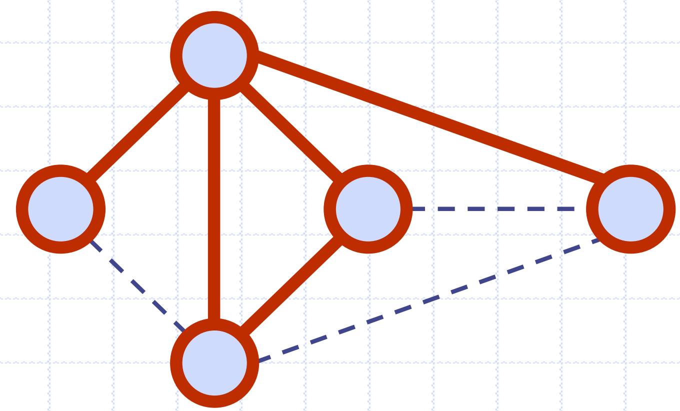
부그래프



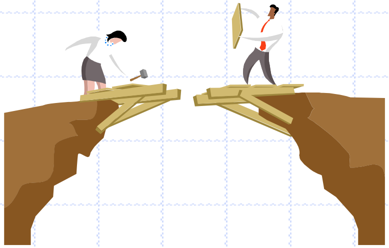
- 그래프 $G = (V, E)$의 **부그래프**(subgraph): 다음 정점과 간선으로 구성된 그래프
  - **정점**: V의 부분집합
  - **간선**: E의 부분집합
- 그래프 $G = (V, E)$의 **신장 부그래프**(spanning subgraph): 다음 정점과 간선으로 구성된 그래프
  - **정점**: V
  - **간선**: E의 부분집합



부그래프

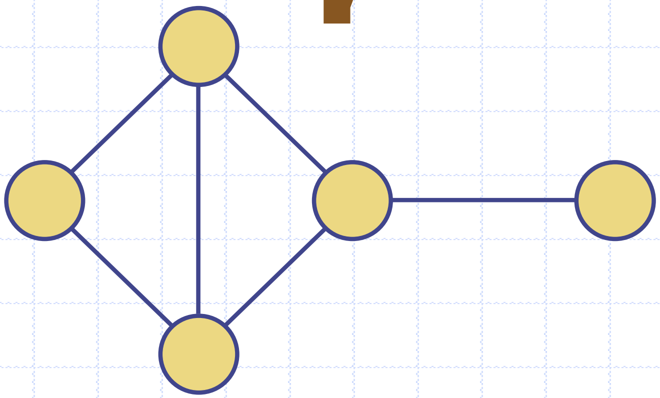


신장 부그래프

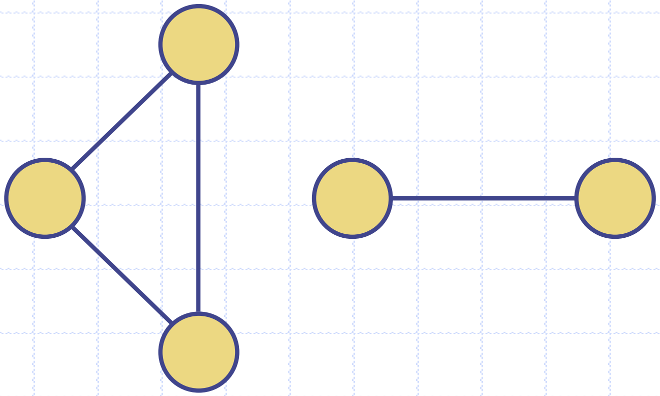


연결성

- 모든 정점쌍에 대해 경로가 존재하면 "그래프가 **연결**(connected)되었다"고 말한다
- 그래프 G의 **연결요소**(connected component): G의 최대 연결 부그래프

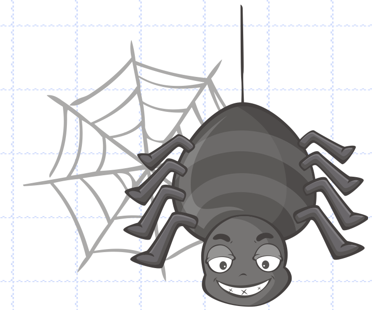


연결그래프

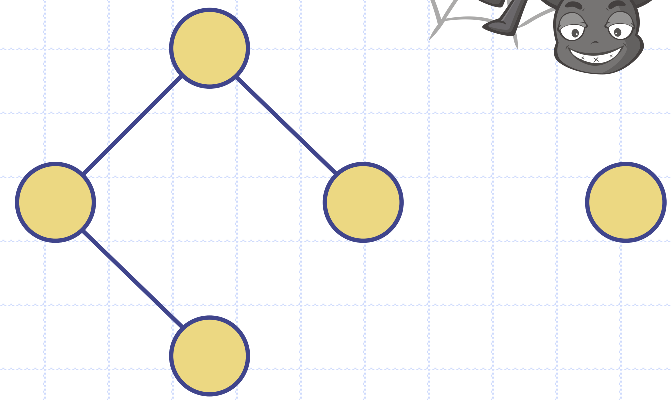


두 개의 연결요소로 구성된 비연결그래프

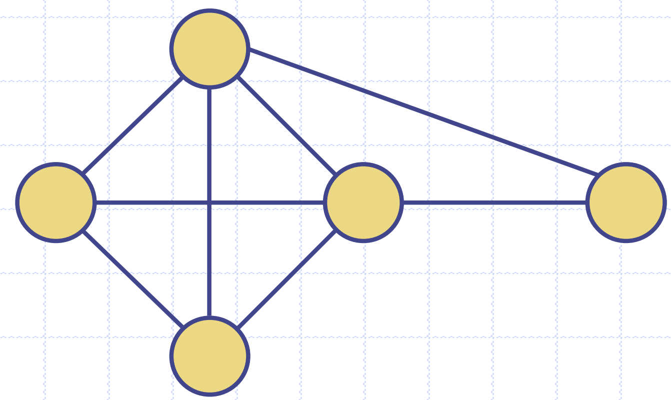
밀집도



- 그래프 알고리즘의 선택은 종종 간선의 **밀집도**에 따라 좌우된다
- **예:** 주어진 그래프 G에 대해, 알고리즘 A와 B가 동일한 문제를 각각 $\mathrm{O}(nm)$ 시간과 $\mathrm{O}(n^2)$ 시간에 해결할 경우,
  - G가 **희소**하다면, 알고리즘 A가 B보다 빠르다
  - G가 **밀집**하다면, 알고리즘 B가 A보다 빠르다

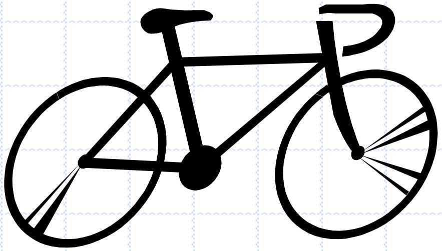


희소그래프



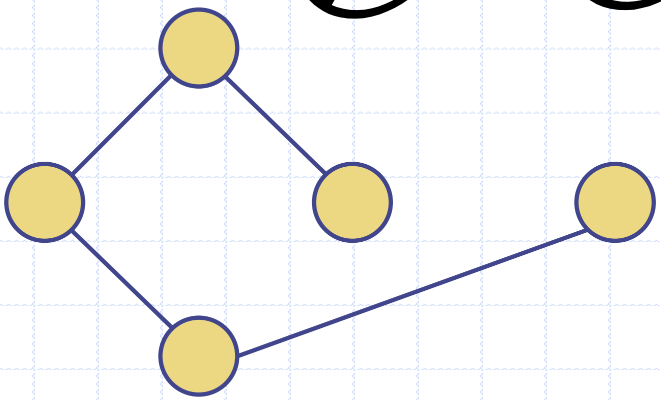
밀집그래프

싸이클

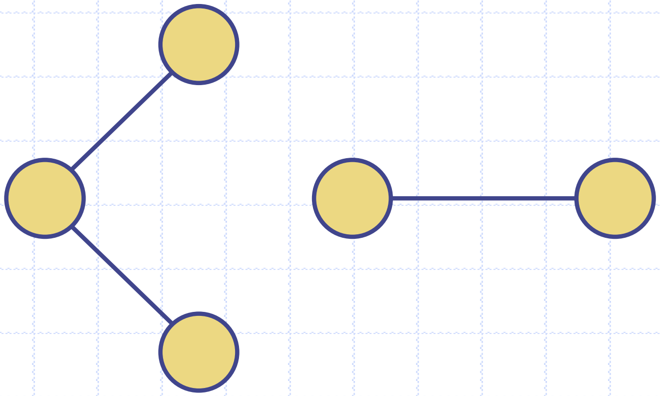


- **자유트리**(free tree), 또는 **트리**: 다음 조건을 만족하는 무방향그래프 T
  - T는 연결됨
  - T에 싸이클이 존재하지 않음

  (위 트리에 대한 정의는 **루트가 있는 트리**에 대한 정의와는 다르다)
- **숲**(forest): 싸이클이 없는 무방향그래프
- 숲의 연결요소는 트리들이다

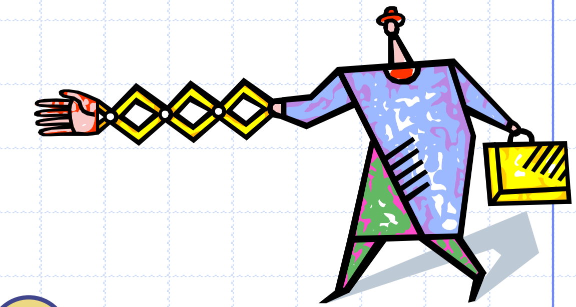


트리

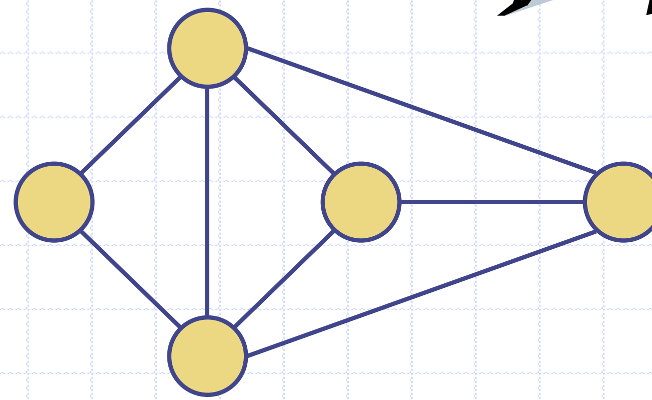


숲

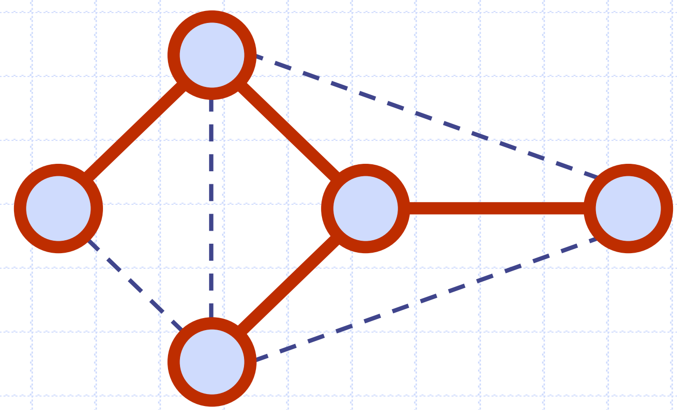
신장



- 연결그래프의 **신장트리**(spanning tree): 신장 부그래프 가운데 트리인 것
- 신장트리는 그래프가 트리가 아닌 한, 유일하지 않다
- 신장트리는 통신망 설계에 응용된다
- 그래프의 **신장숲**(spanning forest): 신장 부그래프 가운데 숲인 것

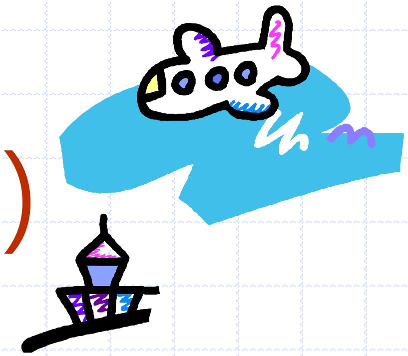


그래프



신장트리

그래프 ADT 메쏘드(공통)



- 정점과 간선들은 **원소**를 저장
- 일반 메쏘드
  - integer numVertices()
  - integer numEdges()
  - iterator vertices()
  - iterator edges()
- 접근 메쏘드
  - vertex aVertex()
- 질의 메쏘드
  - boolean isDirected(e)
- 반복 메쏘드
  - iterator directedEdges()
  - iterator unDirectedEdges()
- 갱신 메쏘드
  - vertex insertVertex(o)
  - removeVertex(v)
  - removeEdge(e)

무방향그래프 ADT 메쏘드

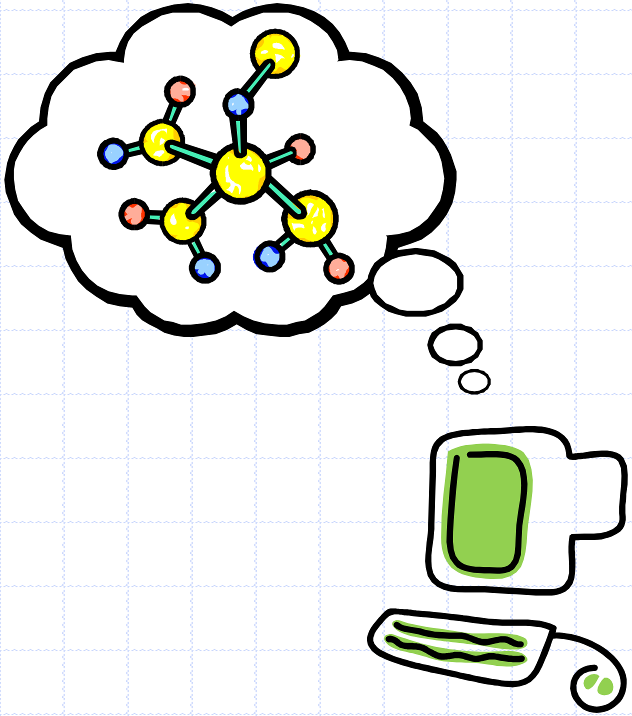
- 접근 메쏘드
  - integer deg(v)
  - vertex opposite(v, e)
- 질의 메쏘드
  - boolean areAdjacent(v, w)
- 반복 메쏘드
  - iterator endVertices(e)
  - iterator adjacentVertices(v)
  - iterator incidentEdges(v)
- 갱신 메쏘드
  - edge insertEdge(v, w, o): 정점 ***v***에서 ***w***로 항목 ***o***를 저장한 무방향간선을 삽입하고 반환

방향그래프 ADT 메쏘드

- 접근 메쏘드
  - vertex origin(e)
  - vertex destination(e)
  - integer inDegree(v)
  - integer outDegree(v)
- 반복 메쏘드
  - iterator inIncidentEdges(v)
  - iterator outIncidentEdges(v)
  - iterator inAdjacentVertices(v)
  - iterator outAdjacentVertices(v)
- 갱신 메쏘드
  - edge insertDirectedEdge(v, w, o): 정점 ***v***에서 ***w***로 항목 ***o***를 저장한 방향간선을 삽입하고 반환
  - makeUndirected(e): 간선 ***e***를 무방향으로 전환
  - reverseDirection(e): 방향간선 ***e***를 역행

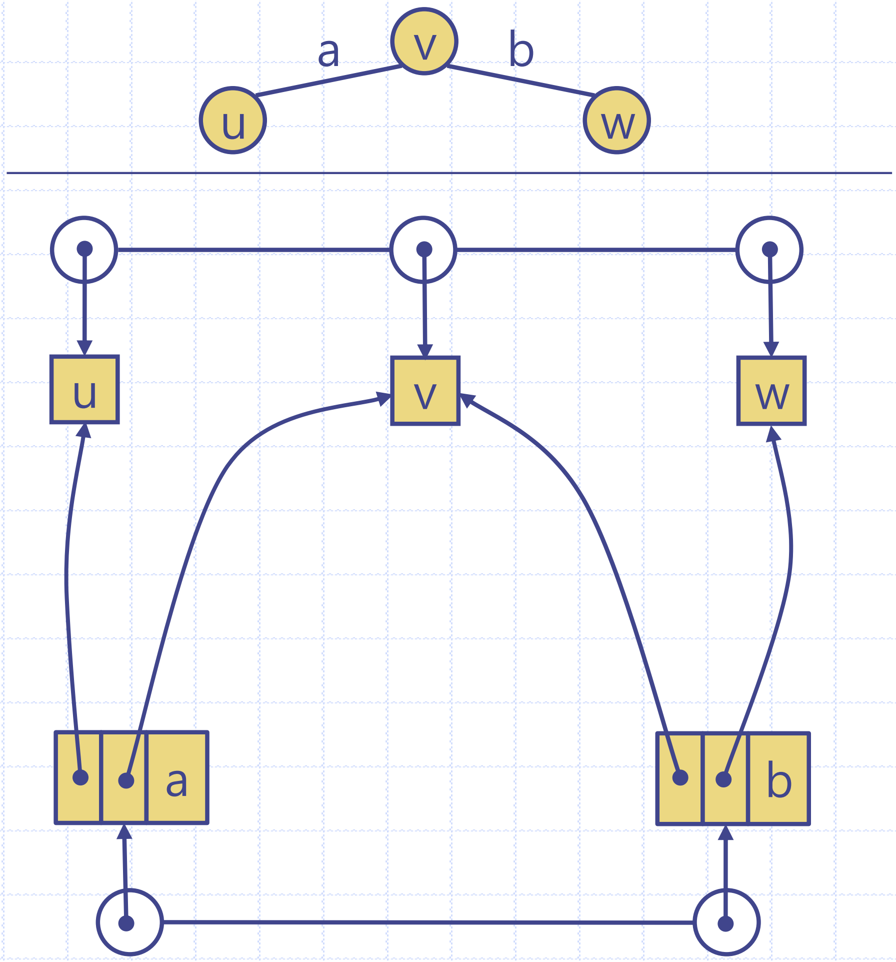
그래프 구현

- 간선리스트(edge list) 구조
- 인접리스트(adjacency list) 구조
- 인접행렬(adjacency matrix) 구조



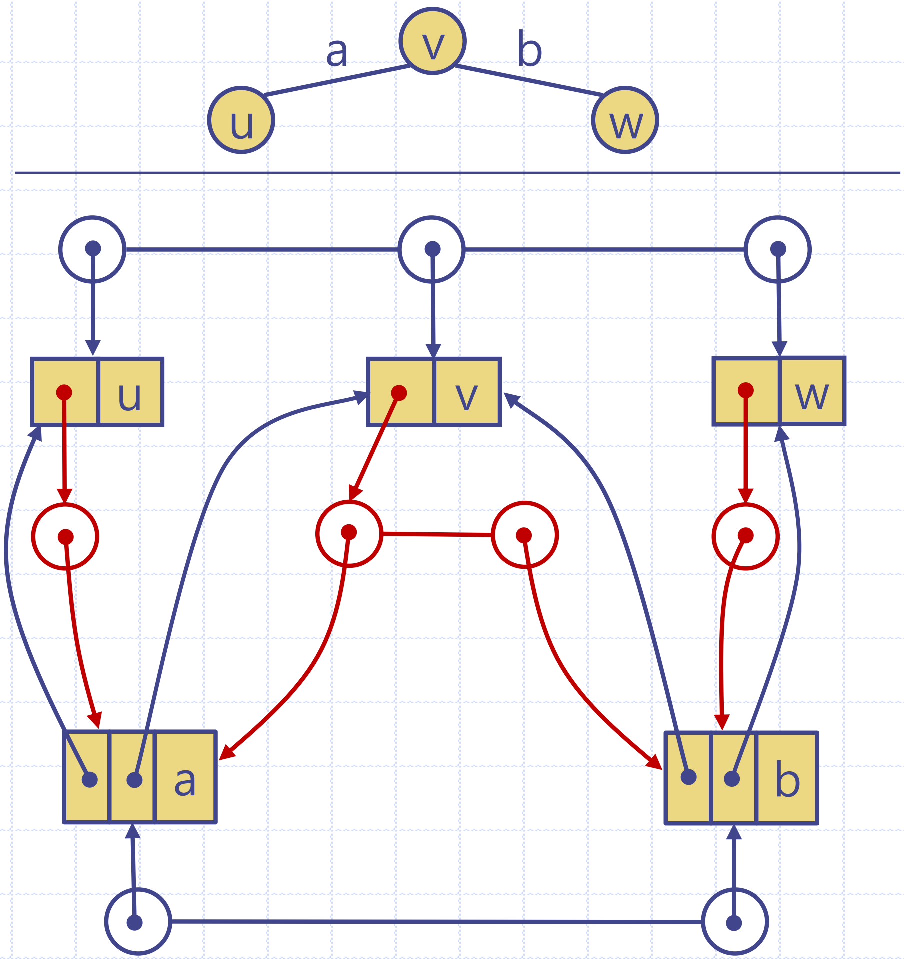
간선리스트 구조

- 정점리스트
  - 정점 노드들에 대한 포인터의 리스트
- 간선리스트
  - 간선 노드들에 대한 포인터의 리스트
- 정점 노드
  - 원소
- 간선 노드
  - 원소
  - 시점 노드
  - 종점 노드



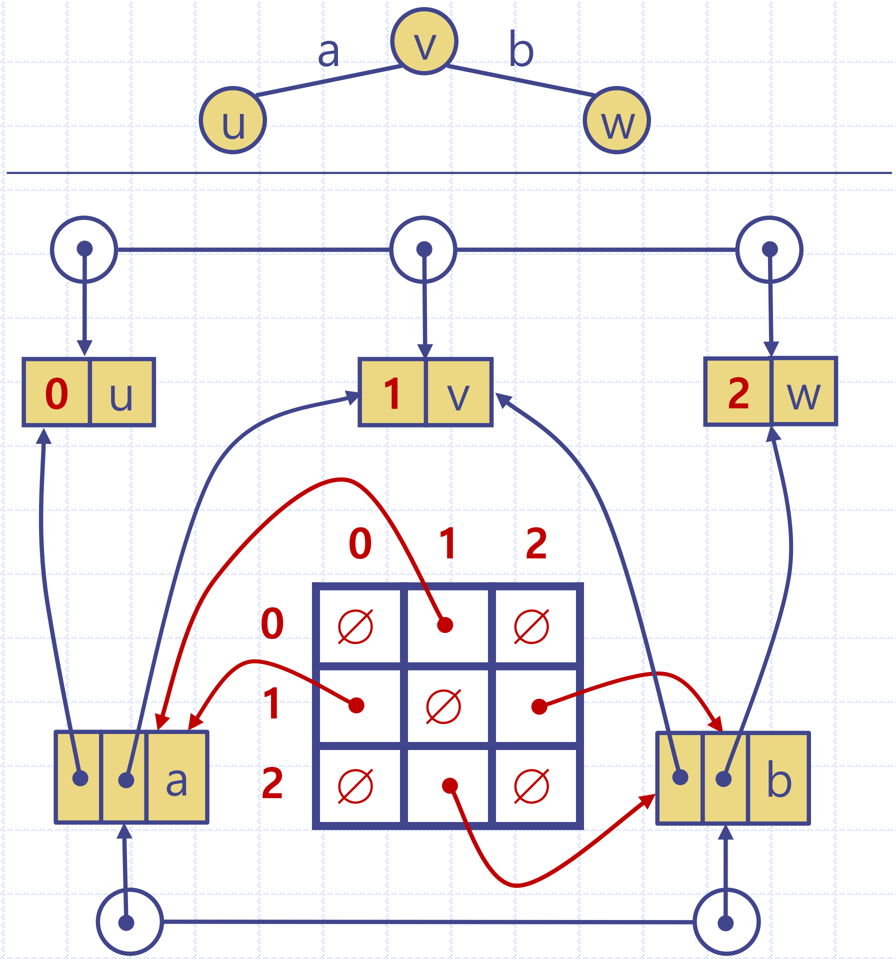
인접리스트 구조

- 간선리스트 구조 + α
- 각 정점에 대한 **부착리스트**
  - 각 정점의 부착간선들을 간선 노드에 대한 참조들의 리스트로 표시

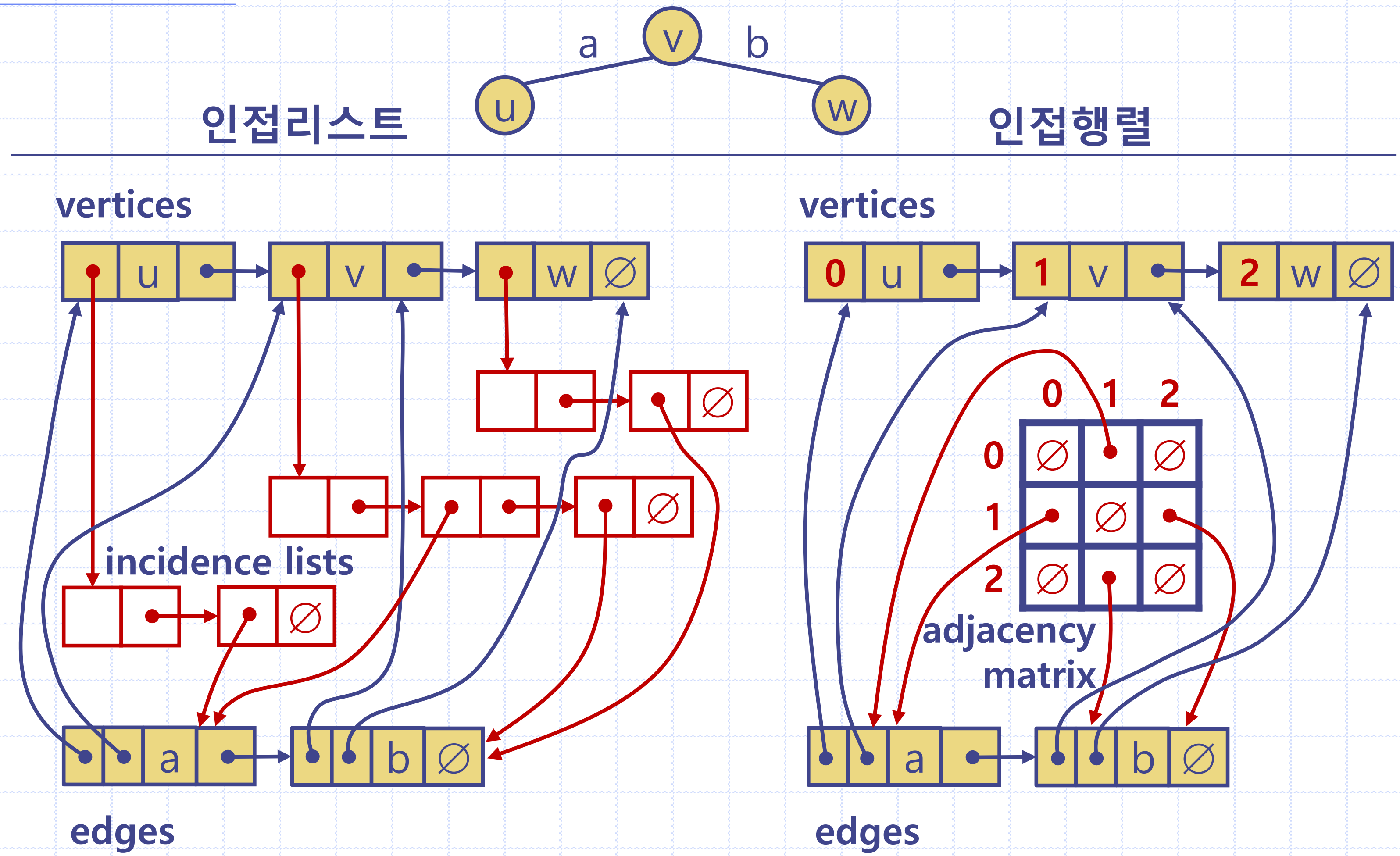


인접행렬 구조

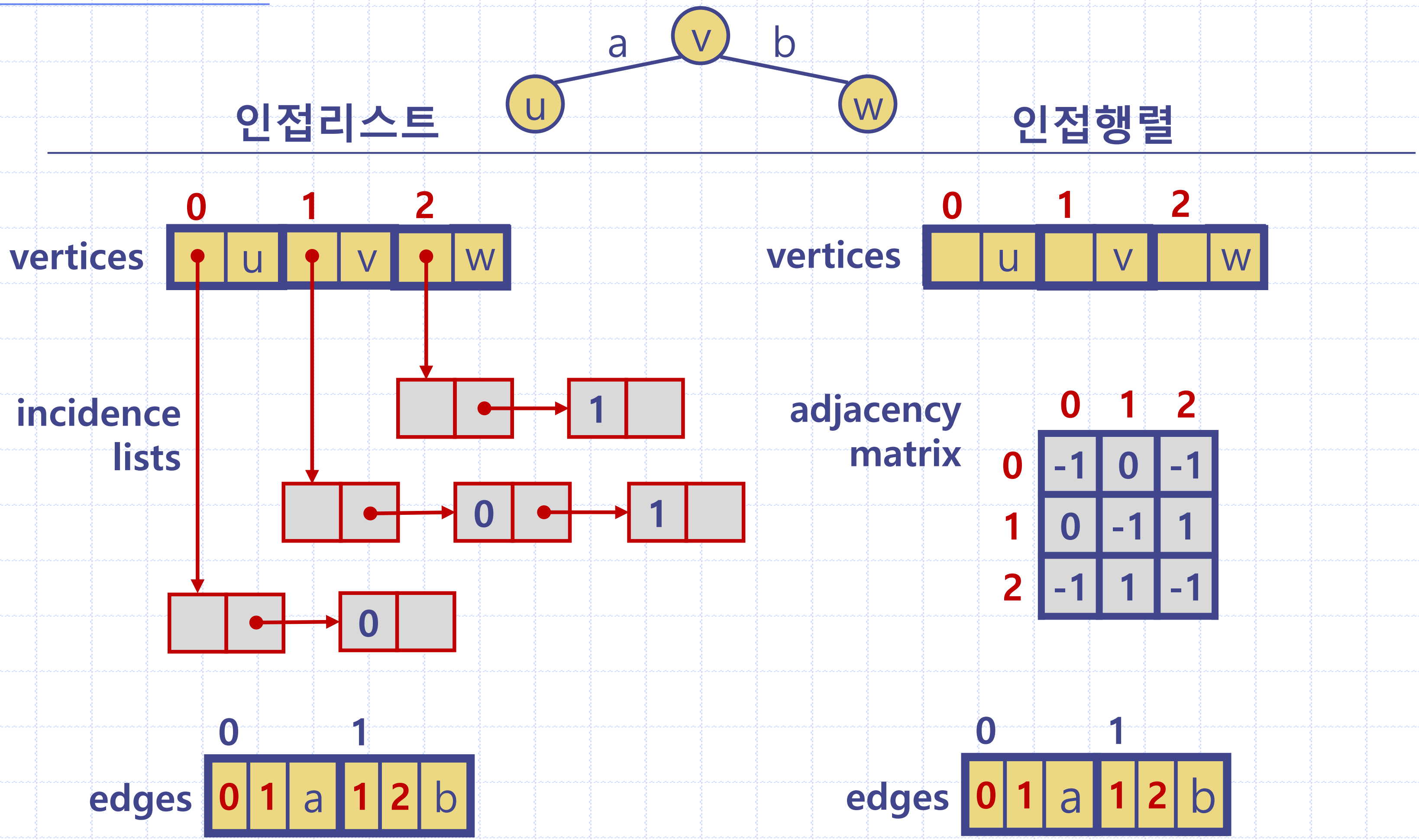
- 간선리스트 구조 + α
- 정점 개체에 대한 확장
  - 정점에 해당하는 정수 키(첨자)
- **인접행렬**
  - n x n 배열
  - 인접정점 쌍에 대응하는 간선 노드들에 대한 참조
  - 비인접정점 쌍에 대한 널 정보
- "구식 버전"은 간선의 존재여부만을 1(간선 존재)과 0(간선 부존재)으로 표시함



연결리스트를 이용한 상세 구현



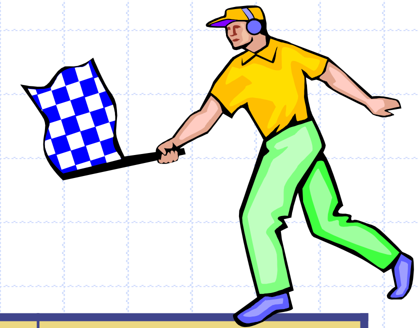
배열을 이용한 상세 구현


a
v
b
u
w
인접리스트
인접행렬
vertices
incidence lists
edges
vertices
adjacency matrix
edges

그래프 상세 구현 비교

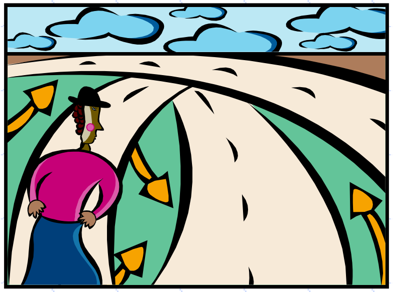
		인접리스트	인접행렬
연결리스트	정점리스트, 간선리스트	동적메모리 노드의 연결리스트	
	정점, 간선	동적메모리 노드	
	인접 정보	포인터의 연결리스트	2D 포인터 배열
	장점	동적 그래프에 사용 시 유리	
	단점	다수의 포인터 사용으로 복잡	
배열	정점리스트, 간선리스트	구조체 배열	
	정점, 간선	구조체	
	인접 정보	첨자의 연결리스트	2D 첨자 배열
	장점	다수의 포인터를 첨자로 대체하여 단순	
	단점	동적 그래프에 사용 시 불리	

점근 성능 비교



◆ n 정점과 m 간선 ◆ 병렬 간선 없음 ◆ 루프 없음 ◆ "big-Oh" 한계임	간선 리스트	인접리스트	인접행렬
공간	$n + m$	$n + m$	n^2
incidentEdges(v)	m	$deg(v)$	n
adjacentVertices(v)	m	$deg(v)$	n
areAdjacent(v, w)	m	$min(deg(v), deg(w))$	1
insertVertex(o)	1	1	n
insertEdge(v, w, o)	1	1	1
removeVertex(v)	m	$deg(v)$	n
removeEdge(e)	1	1	1

응용문제: 그래프 구현 방식 선택



- 다음 각 경우에 **인접리스트 구조**와 **인접행렬 구조** 둘 중 어느 것을 사용하겠는가? 선택의 이유를 설명하라
  a. 그래프가 10,000개의 정점과 20,000개의 간선을 가지며 가능한 최소한의 공간을 사용하는 것이 중요하다
  b. 그래프가 10,000개의 정점과 20,000,000개의 간선을 가지며 가능한 최소한의 공간을 사용하는 것이 중요하다
  c. 얼마의 공간을 사용하든, areAdjacent 질의에 가능한 빨리 답해야 한다

해결

a. **인접리스트 구조**가 유리 – 실상 **인접행렬 구조**를 쓴다면 많은 공간을 낭비. 왜냐면 20,000개의 간선만이 존재하는데도 100,000,000개의 간선에 대한 공간을 할당하기 때문
b. 일반적으로, 이 경우에는 양쪽 구조 모두가 적합 – 공간 사용량을 고려한다면 확실한 승자는 없다. 두 구조의 정확한 공간 사용량은 상세 구현에 따라 달라진다는 점에 유의. areAdjacent 작업에서는 **인접행렬 구조**가 우월하지만, insertVertex와 removeVertex 작업에서는 **인접리스트 구조**가 우월
c. **인접행렬 구조**가 유리 – 그 이유는 이 구조가 areAdjacent 작업을 정점이나 간선의 개수에 관계없이 **O**(1) 시간에 지원하기 때문

응용문제: 배열을 이용한 그래프 데이터구조

- 그림 13-15에 보인대로 그래프를 **배열**을 이용해 구현하기 위한 데이터구조를 대략 설계하라
- **인접리스트**, **인접행렬** 구조 모두에 공통적인 부분과 차별적인 부분이 잘 나타나도록 작성해야 한다
- 마지막으로, **방향그래프**를 구현하기 위해서는, 위의 설계를 어떻게 변경해야 할지 대강 설명하라

해결: 공통 사항

- **인접리스트, 인접행렬** 구조 공통 사항
- **그래프**를 다음 필드로 구성되는 **레코드**(즉, 구조체)로 정의
  - **vertices:** 정점 레코드의 배열[0:n – 1]
  - **edges:** 간선 레코드의 배열[0:m – 1]
- **정점**을 다음 필드로 구성되는 **레코드**로 정의
  - **name:** 식별자
- **간선**을 다음 필드로 구성되는 **레코드**로 정의
  - **name:** 식별자
  - **endpoints:** 정점 인덱스 1, 정점 인덱스 2의 집합

해결 (conti.): 차별 사항

- **(인접리스트 구조)** 정점 레코드에 다음 필드를 추가
  - **incidentEdges:** 부착간선 인덱스의 헤더연결리스트 (작업 효율을 위해 헤더노드를 추가함)
- **(인접행렬 구조)** 그래프 레코드에 다음 필드를 추가
  - **adjacencyMatrix:** 간선 인덱스의 2D 배열[0:n – 1, 0:n – 1] (간선이 존재하지 않는 경우 불법 인덱스 –1을 저장)

해결 (conti.): 정점 레코드 확장

- 참고로, 필요하다면 **정점 레코드**에 다음 필드를 추가 사용 가능
  - **label:** boolean
    - **순회** 알고리즘에서 **Fresh**, **Visited** 등의 값을 저장
  - **distance:** number
    - **최단경로** 찾기 알고리즘에서 출발점으로부터 이 정점까지의 거리, 또는 **최소신장트리** 찾기 알고리즘에서 배낭으로부터 이 정점까지의 거리를 저장
  - **locator:** integer
    - **최단경로** 또는 **최소신장트리** 찾기 알고리즘에서 이 정점의 우선순위 큐에서의 위치
  - **parent:** 간선 인덱스
    - **최단경로트리** 또는 **최소신장트리**에서 이 정점의 부모로 향하는 간선을 저장

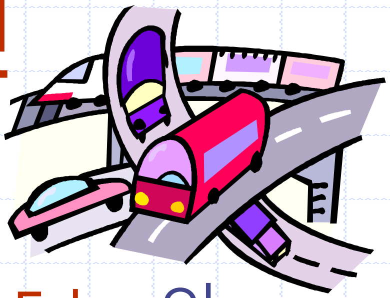
해결 (conti.): 간선 레코드 확장

- 마찬가지로, 필요하다면 **간선 레코드**에 다음 필드를 추가해 사용할 수 있다
  - **label:** integer or string
    - **순회** 알고리즘에서 **Fresh**, **Tree**, **Back**, **Cross** 등의 값을 저장
  - **weight:** number
    - **가중그래프**에서 간선의 무게를 표현

해결 (conti.): 방향그래프

- 마지막으로, **방향그래프**의 구현을 위해서는 **간선 레코드**의 **endpoints** 필드 대신 다음 두 개의 필드를 사용
  - **origin:** 출발정점 인덱스
  - **destination:** 도착정점 인덱스
- **(인접리스트 구조) 정점 레코드**의 **incidentEdges** 필드를 다음 두 개의 필드로 구분하여 사용
  - **outEdges:** 진출 부착간선 인덱스의 헤더연결리스트
  - **inEdges:** 진입 부착간선 인덱스의 헤더연결리스트

응용문제: 정점 또는 간선 삭제 작업의 성능



- 표13-2에 제시된 removeVertex와 removeEdge의 성능을 구현할 수 있는 구체적 방안을 설명하라

해결: 개요

- 우선 **정점**이나 **간선**을 **"실제"** 삭제함으로써 초래되는 "느린" 수행 성능을 살펴보고, 이에 대한 대안으로 해시테이블에서 사용했던 **"비활성화"** 방식의 삭제를 채택함으로써 제시된 성능을 구현할 수 있음을 보인다

해결: "실제" 삭제

- removeVertex(v)
  - **인접리스트** 구조의 경우, 정점 $\boldsymbol{v}$의 모든 부착간선 $\boldsymbol{e} = (\boldsymbol{v}, \boldsymbol{w})$에 대해 $\boldsymbol{v}$와 $\boldsymbol{w}$의 부착간선리스트에서 $\boldsymbol{e}$ 노드 삭제, $\boldsymbol{E}$(간선리스트)에서 $\boldsymbol{e}$ 삭제 후, $\boldsymbol{V}$(정점리스트)에서 $\boldsymbol{v}$ 삭제 – $\mathbf{O}(\boldsymbol{m})$ 시간 소요
  - **인접행렬** 구조의 경우, 행렬 내 $\boldsymbol{v}$행과 $\boldsymbol{v}$열에 저장된 모든 간선 $\boldsymbol{e}$를 $\boldsymbol{E}$에서 삭제 후, $\boldsymbol{V}$ 에서 $\boldsymbol{v}$ 삭제, 그리고 행렬의 $\boldsymbol{v}$행과 $\boldsymbol{v}$열 삭제 및 나머지 정점의 행렬첨자 조정 – $\mathbf{O}(\boldsymbol{n}^2)$ 시간 소요
- removeEdge(e)
  - **인접리스트** 구조의 경우, $\boldsymbol{e} = (\boldsymbol{u}, \boldsymbol{w})$에 대해 $\boldsymbol{u}$와 $\boldsymbol{w}$의 부착간선리스트에서 $\boldsymbol{e}$ 노드 삭제 후 $\boldsymbol{E}$에서 $\boldsymbol{e}$ 삭제 – $\mathbf{O}(\boldsymbol{deg}(\boldsymbol{u}) + \boldsymbol{deg}(\boldsymbol{w}))$ 시간 소요
  - **인접행렬** 구조의 경우, $\boldsymbol{e} = (\boldsymbol{u}, \boldsymbol{w})$에 대해 행렬원소 $[\boldsymbol{u}, \boldsymbol{w}]$와 $[\boldsymbol{w}, \boldsymbol{u}]$를 널로 치환한 후 $\boldsymbol{E}$에서 $\boldsymbol{e}$ 삭제 – $\mathbf{O}(1)$ 시간 소요

해결: "실제" 삭제 (conti.)

- **주의:** V와 E를 **배열**로 구현한 경우 배열첨자 조정에 따른 추가 작업 필요, 즉:
  - 정점이나 간선 삭제 후 남은 정점들과 간선들의 배열첨자를 변경
  - 변경된 값을 사용하여 간선리스트, 부착간선리스트, 인접행렬 등을 갱신

해결: "비활성화" 방식 삭제

- removeVertex(v)
  - **인접리스트** 구조의 경우, ***v***의 부착간선들을 모두 비활성화 한 후 ***v***를 비활성화한다 – **O(*deg*(*v*))** 시간 소요
  - **인접행렬** 구조의 경우, 행렬의 ***v***행과 ***v***열에 저장된 모든 간선 ***e***를 비활성화한 후 ***v***를 비활성화한다 – **O(*n*)** 시간 소요
- removeEdge(e)
  - **인접리스트** 구조, **인접행렬** 구조 두 경우 모두 간선 ***e***를 비활성화한다 – **O(1)** 시간 소요

해결: “비활성화” 방식 삭제 (conti.)

- **참고**
  - *V*와 *E*를 **배열**로 구현한 경우라도 배열첨자 조정 또는 이에 따르는 추가 작업이 없음
  - 삭제 이후 그래프 작업에서, 비활성 정점이나 간선에 대해서는 존재하지 않는 것으로 취급
  - 하지만 사용환경에 따라서는, 빈번한 삭제로 인해 그래프가 비활성 정점과 간선으로 포화되어 시스템 수행 성능이 저하될 수 있다 – 이 경우 주기적인 유지보수를 통해 비활성 정점과 간선을 **“실제”** 삭제하고 관련 데이터구조를 갱신하여 메모리를 회수함과 동시에 시스템 성능을 제고