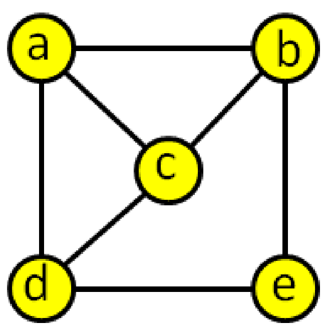
제9장 그래프

그래프(Graph)

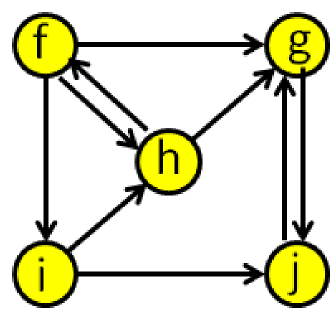
- **인터넷, 도로, 운송, 전력, 상하수도망, 신경망, 화학성분 결합, 단백질 네트워크, 금융 네트워크, 소셜네트워크 분석(Social Network Analysis) 등의 광범위한 분야에서 활용되는 자료구조.**
  - 그래프 용어
  - 깊이우선탐색(DFS)
  - 너비우선탐색(BFS)
  - 연결성분(Connected Components)
  - 이중연결성분(Doubly Connected Components)
  - 강연결성분(Strongly Connected Components)
  - 위상정렬(Topological Sort)
  - 최소신장트리(Minimum Spanning Tree)
  - 최단경로(Shortest Paths)
  - 소셜네트워크 분석(Social Network Analysis)

9.1 그래프

- 그래프는 정점(Vertex)과 간선(Edge)의 집합으로 하나의 간선은 두 개의 정점을 연결
- 그래프는 G=(V, E)로 표현, V=정점의 집합, E=간선의 집합
- 방향그래프(Directed Graph): 간선에 방향이 있는 그래프
- 무방향그래프(Undirected Graph): 간선에 방향이 없는 그래프



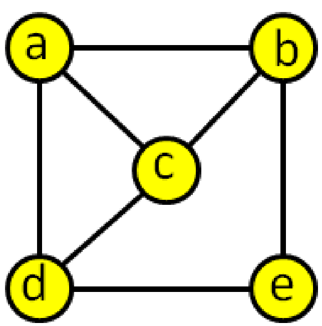
(a) 무방향그래프



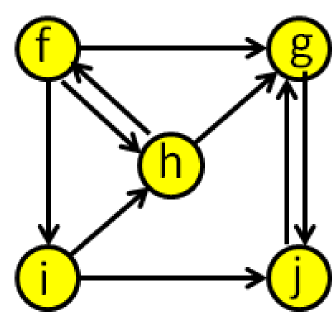
(b) 방향그래프

9.1.1 그래프 용어

- **정점 a와 b를 연결하는 간선을 (a, b)로 표현**
- **정점 a에서 b로 간선의 방향이 있는 경우 <a, b>로 표현**
- **차수(Degree): 정점에 인접한 정점의 수**
  - 방향그래프에서는 차수를 진입차수(In-degree)와 진출차수(Out-degree)로 구분
  - 그림(a) 정점 a의 차수 = 3, 정점 e의 차수 = 2.
  - 그림(b) 정점 g의 진입차수 = 3, 진출차수 = 1.



(a) 무방향그래프

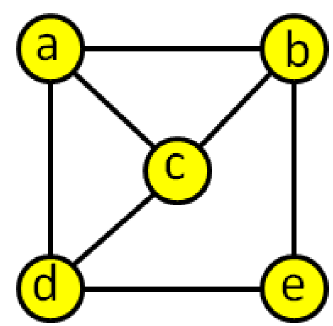


(b) 방향그래프

9.1.1 그래프 용어

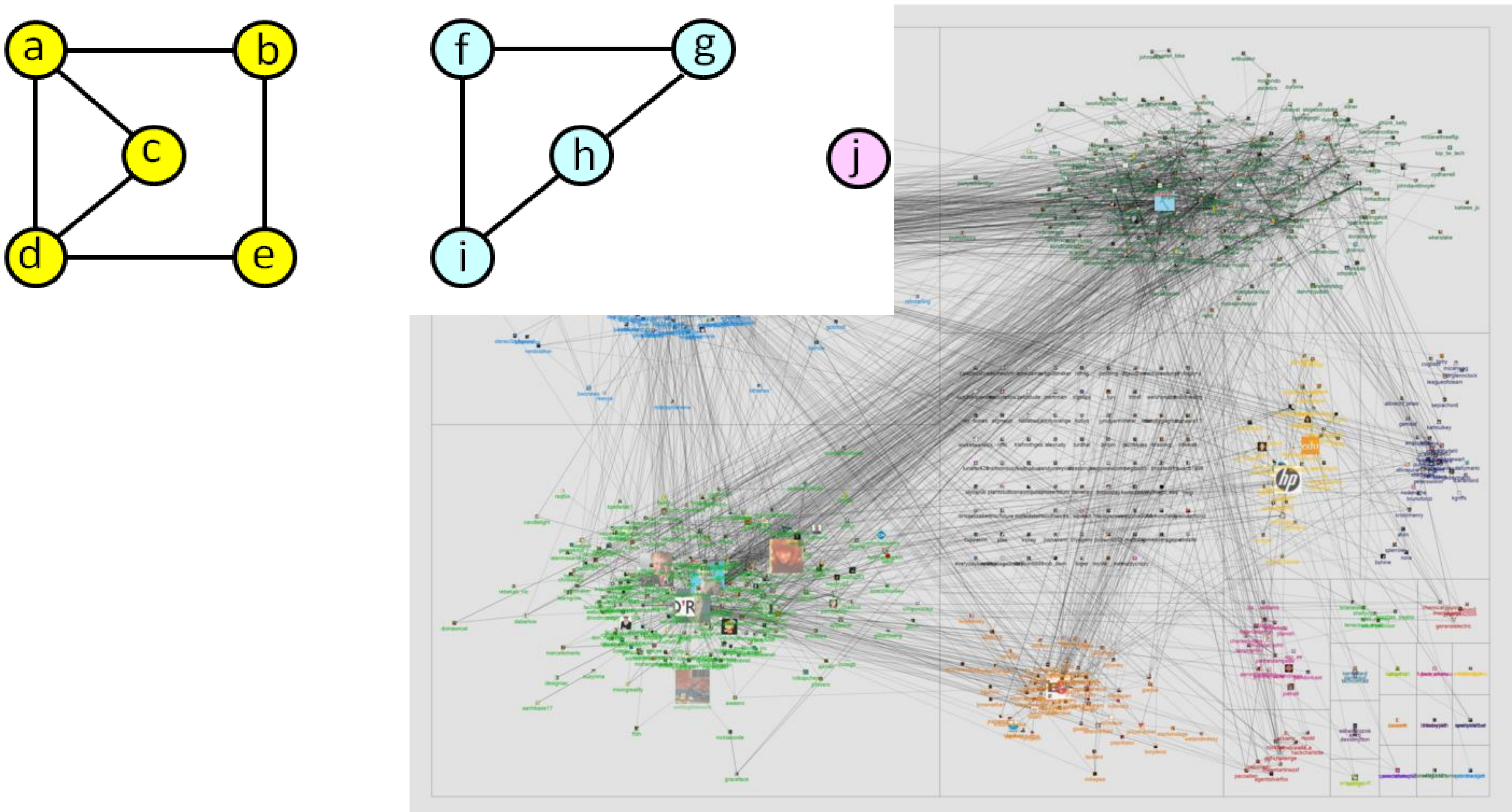
- **경로(Path)는 시작 정점 u부터 도착점 v까지의 정점들을 나열하여 표현**
  - [a, c, b, e]: 정점 a로부터 도착점 e까지의 여러 경로들 중 하나

  - 단순경로(Simple Path): 경로 상의 정점들이 모두 다른 경로
  - ‘일반적인’ 경로: 동일한 정점을 중복하여 방문하는 경우를 포함
    - [a, b, c, b, e]: 정점 a로부터 도착점 e까지의 경로
  - 싸이클(Cycle): 시작 정점과 도착점이 동일한 단순경로
    - [a, b, e, d, c, a]



9.1.1 그래프 용어

- **연결성분(Connected Component)**
  - 그래프에서 정점들이 서로 연결되어 있는 부분
  - 아래의 그래프는 3개의 연결성분, [a, b, c, d, e], [f, g, h, i], [j]로 구성



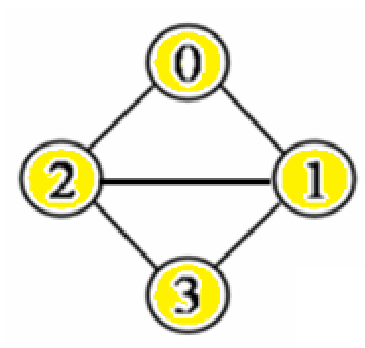
9.1.1 그래프 용어

- **가중치(Weighted) 그래프: 간선에 가중치가 부여된 그래프**
  - 가중치는 두 정점 사이의 거리, 지나는 시간이 될 수도 있음.
  - 또한 음수인 경우도 존재
- **부분그래프(Subgraph): 주어진 그래프의 정점과 간선의 일부분(집합)으로 이루어진 그래프**
  - 부분그래프는 원래의 그래프에 없는 정점이나 간선을 포함하지 않음
- **트리(Tree): 싸이클이 없는 그래프**
- **신장트리(Spanning Tree): 주어진 그래프가 하나의 연결성분으로 구성되어 있을 때, 그래프의 모든 정점들을 싸이클 없이 연결하는 부분그래프**

9.1.2 그래프 자료구조

- **그래프를 자료구조로서 저장하는 방법**
  - 인접행렬(Adjacency Matrix)
  - 인접리스트(Adjacency List)

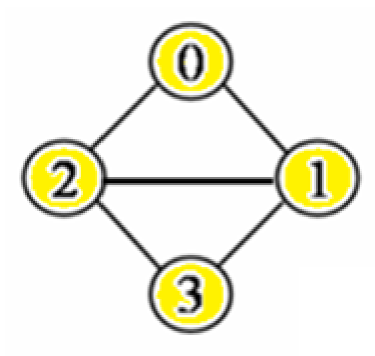
- **N개의 정점을 가진 그래프의 인접행렬은 2차원 NxN 배열에 저장**
  - 배열이 a라면, 정점들을 0, 1, 2,…, N-1로 하여, 정점 i와 j 사이에 간선이 없으면 a[i][j] = 0, 간선이 있으면 a[i][j] = 1로 표현
  - 가중치그래프는 1 대신 가중치 저장

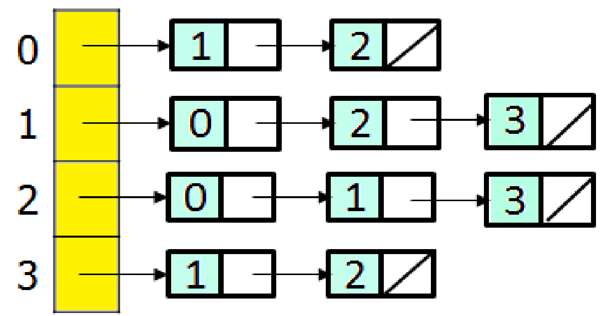


$$\begin{array}{c c} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

9.1.2 그래프 자료구조

- **인접리스트는 각 정점마다 1 개의 단순연결리스트를 이용하여 인접한 각 정점을 노드에 저장**






```
public class Edge {
    int adjvertex;  // 간선의 다른쪽 정점
    public Edge(int v) { // 생성자
        adjvertex = v;
    }
}
```

- Edge 객체는 간선의 다른 쪽 정점만을 가짐
- 인접리스트 adjList는 List배열로 선언
- List의 각 원소는 LinkedList로 선언하여 단순연결리스트의 각 노드에 인접한 간선(다른 쪽 정점)을 가진 Edge 객체를 저장

9.1.2 그래프 자료구조

- **인접리스트를 만드는 프로그램**

```
List<Edge>[] adjList = new List[N];
for (int i = 0; i < N; i++) {
    adjList[i] = new LinkedList<>();
    for (int j = 0; j < N; j++) {
        if (/*정점 i와 j사이에 간선이 존재하면*/) {
            Edge e = new Edge(j);
            adjList[i].add(e);
        }
    }
}
```

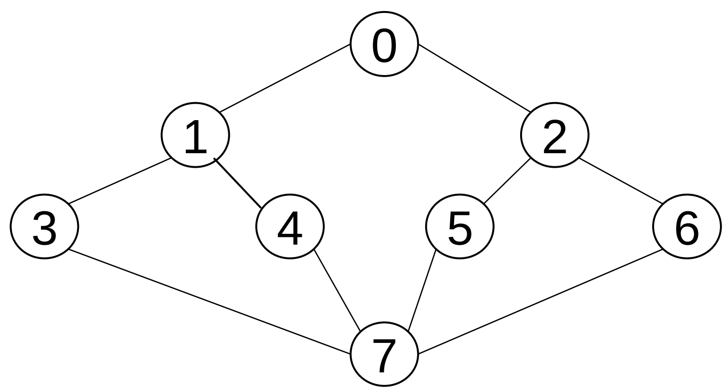
- 인접리스트 adjList는 List배열로 선언
- List의 각 원소는 LinkedList로 선언하여 단순연결리스트의 각 노드에 인접한 간선(다른 쪽 정점)을 가진 Edge 객체를 저장

9.1.2 그래프 자료구조

- **실세계의 그래프는 대부분 정점의 평균 차수가 작은 희소그래프(Sparse Graph)이다.**
- **희소그래프의 간선 수는 최대 간선 수인 N(N-1)/2보다 훨씬 작으므로 인접리스트에 저장하는 것이 매우 적절**
  - 무방향그래프를 인접리스트를 사용하여 저장할 경우 간선 1 개당 2개의 Edge 객체를 저장하고, 방향그래프의 경우 간선 1 개당 1개의 Edge 객체만 저장하기 때문
- **조밀그래프(Dense Graph): 간선의 수가 최대 간선 수에 근접한 그래프**

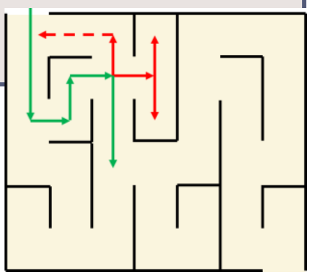
9.2 그래프 탐색

- **그래프에서는 두 가지 방식으로 모든 정점을 방문**
  - 깊이우선탐색(DFS; Depth First Search)
  - 너비우선탐색(BFS; Breadth First Search)



9.2.1 깊이우선탐색(DFS)

[핵심 아이디어] DFS는 실타래를 가지고 미로에서 출구를 찾는 것과 유사. 새로운 곳으로 갈 때는 실타래를 풀면서 진행하고, <u>길이 막혀 진행할 수 없을 때에는 실타래를 되감으며</u> 왔던 길을 되돌아가 같은 방법으로 다른 경로를 탐색하여 출구를 찾는다.



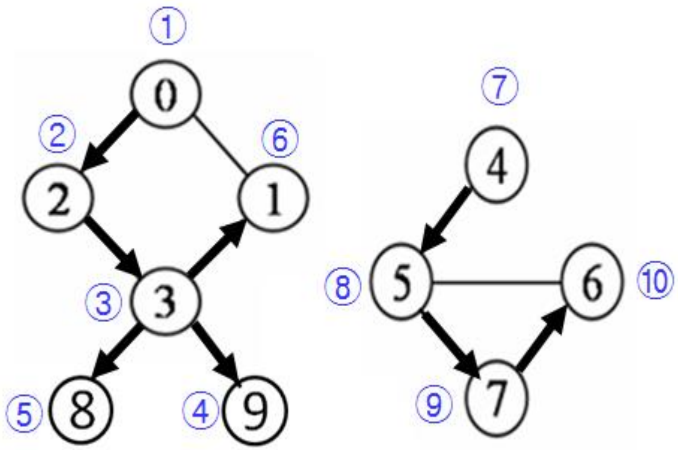
- 그래프에서의 DFS는 임의의 정점에서 시작하여 이웃하는 하나의 정점을 방문하고,
- 방금 방문한 정점의 이웃 정점을 방문하며,
- 이웃하는 정점들을 모두 방문한 경우에는 이전 정점으로 되돌아 가서 탐색을 수행하는 방식으로 진행

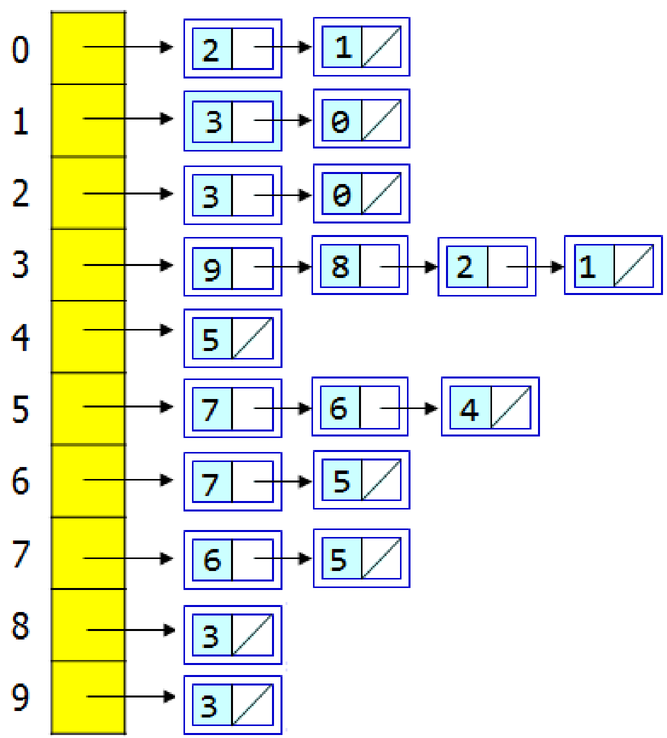
```java
import java.util.List;
public class DFS {
    int N;    // 그래프 정점의 수
    List<Edge>[] graph;
    private boolean[ ] visited; // DFS 수행 중 방문한 정점을 true로 만든다.
    public DFS(List<Edge>[] adjList) { // 생성자
        N = adjList.length;
        graph = adjList;
        visited = new boolean [N];
        for (int i = 0; i < N; i++) visited[i] = false;  // 배열 초기화
        for (int i = 0; i < N; i++) if (!visited[i]) dfs(i);
    }
    private void dfs(int i) {
        visited[i] = true;          // 정점 i가 방문되어 visited[i]를 true로 만든다.
        System.out.print(i+" ");   // 정점 i가 방문되었음을 출력한다.
        for (Edge e: graph[i]) {   // 정점 i에 인접한 각 정점에 대해
            if (!visited[e.adjvertex]) { // 정점 i에 인접한 정점이 방문 안되었으면 재귀호출
                dfs(e.adjvertex);
            }
        }
    }
}
```

9.2.1 깊이우선탐색(DFS)

- Line 10: for-루프에서 visited 배열을 false로 초기화, 정점 i를 방문하면 visited[i] = true로 만들어 한번 방문한 정점을 다시 방문하는 것을 방지
  - 단, 방문은 정점을 출력하는 것으로 가정
- Line 11: for-루프에서는 0부터 N-1까지의 정점에 대해 dfs() 메소드를 호출
  - 그래프가 여러 개의 연결성분으로 구성된 경우 정점 0으로부터 방문을 시작하여 계속해서 인접한 정점을 방문하다 보면 정점 0이 속한 연결성분의 정점들만 방문하고, 다른 연결성분의 정점들은 방문할 수 없기 때문
- Line 13의 dfs() 메소드: line 14~15에서 visited[i]를 true로 만들고 i를 출력
- Line 16: for-루프는 방금 방문한 정점 i에 인접한 정점(w.adjvertex)이 아직 방문되지 않은 경우 line 18에서 dfs()를 재귀호출

DFS 수행 과정

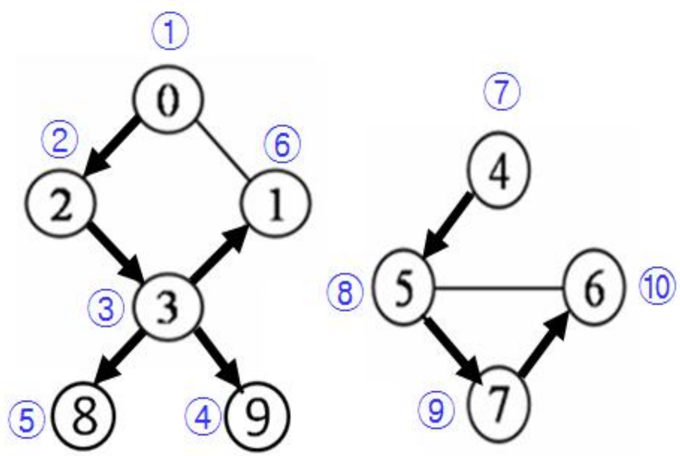


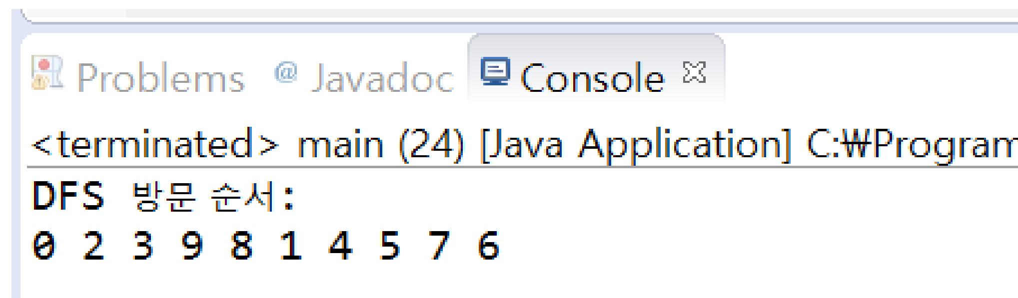


방문순서	dfs()호출	visited[]	출력
①	dfs(0)	visited[0] = true	0
②	dfs(2)	visited[2] = true	2
③	dfs(3)	visited[3] = true	3
④	dfs(9)	visited[9] = true	9
⑤	dfs(8)	visited[8] = true	8
⑥	dfs(1)	visited[1] = true	1
⑦	dfs(4)	visited[4] = true	4
⑧	dfs(5)	visited[5] = true	5
⑨	dfs(7)	visited[7] = true	7
⑩	dfs(6)	visited[6] = true	6

9.2.1 깊이우선탐색(DFS)

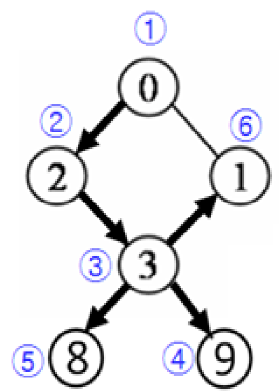
- DFS 클래스의 line 11에 있는 for-루프에서 i가 각각 1, 2, 3일 때 visited[i] = true이므로 dfs()가 호출되지 않음
- 그러나 i = 4일 때에는 visited[4] = false이므로, dfs(4)를 호출하면서 정점4는 7 번째로 방문되며 나머지 정점들도 차례로 방문



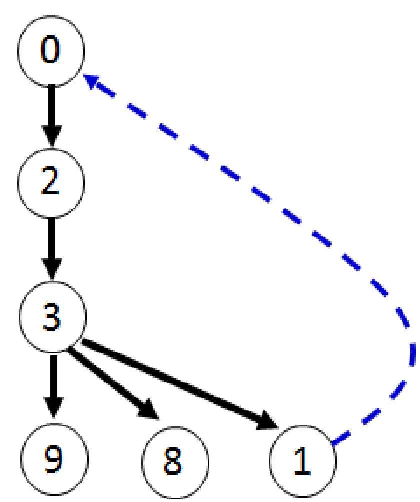


9.2.1 깊이우선탐색(DFS)

- **(a)의 DFS 방문순서대로 정점 0부터 위에서 아래방향으로 정점들을 그리면 (b)와 같은 트리가 만들어진다.**
  - 실선은 탐색하며 처음 방문할 때 사용된 간선
  - 점선은 뒷간선(Back Edge)으로서 탐색 중 이미 방문 된 정점에 도달한 경우
  - 그래프가 1개의 연결성분으로 되어있을 때 DFS를 수행하며 만들어지는 트리를 깊이우선 신장트리(Depth First Spanning Tree)라 함.



(a)



(b)

수행시간

- DFS의 수행시간은 탐색이 각 정점을 한번씩 방문하며, 각 간선을 한번씩만 사용하여 탐색하기 때문에O(N+M)
- N은 그래프의 정점의 수이고, M은 간선의 수

9.2.2 너비우선탐색(BFS)

- BFS는 임의의 정점 s에서 시작하여 s의 모든 이웃하는 정점들을 방문하고, 방문한 정점들의 이웃 정점들을 모두 방문하는 방식으로 그래프의 모든 정점을 방문

[핵심 아이디어] BFS는 연못에 돌을 던져서 만들어지는 동심원의 물결이 퍼져나가는 것 같이 정점들을 방문한다.

- BFS는 이진트리에서의 레벨순회와 유사

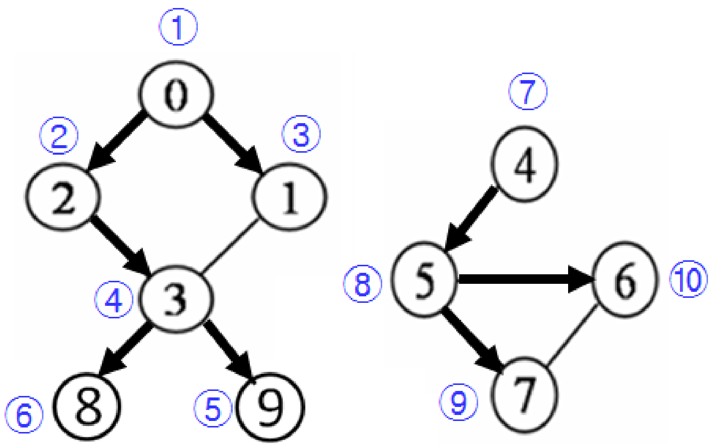
BFS클래스

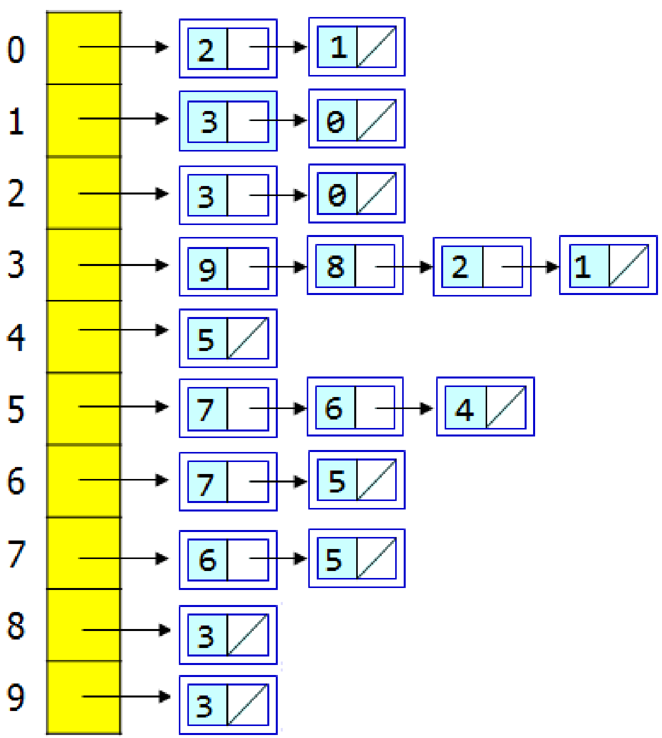
```java
import java.util.*;
public class BFS {
    int N;   // 그래프 정점의 수
    List<Edge>[] graph;
    private boolean[ ] visited;  // BFS 수행 중 방문한 정점의 원소를 true로 만든다.
    public BFS(List<Edge>[] adjList) { // 생성자
        N = adjList.length;
        graph = adjList;
        visited = new boolean [N];
        for (int i = 0; i < N; i++) visited[i] = false;    // 배열 초기화
        for (int i = 0; i < N; i++) if (!visited[i]) bfs(i);
    }
    private void bfs(int i) {
        Queue<Integer> q = new LinkedList<Integer>();  // 큐 선언
        visited[i] = true;
        q.add(i); //큐에 시작 정점 s를 삽입
        while (!q.isEmpty()) {
            int j = q.remove();          // 큐에세 정점 j를 가져옴
            System.out.print(j+" ");
            for (Edge e: graph[j]) {     // 정점 j에 인접한 정점들 중 방문안된 정점 하나씩 방문
                if (!visited[e.adjvertex]) {
                    visited[e.adjvertex] = true;
                    q.add(e.adjvertex);  // 새로이 방문된 정점을 큐에 삽입
                }
            }
        }
    }
}
```

9.2.2 너비우선탐색(BFS)

- Line 10: for-루프에서 visited 배열을 false로 초기화하고, 정점 i를 방문하면 visited[i]를 true로 만들어 한번 방문한 정점을 다시 방문하는 것을 방지
- Line 11: for-루프는0부터 N-1까지의 정점에 대해 bfs() 메소드를 호출하여 그래프의 모든 정점 방문
- Line 13의 bfs() 메소드: line 15~16에서 visited[i]를 true로 만들고, i를 큐에 삽입
- Line 17: while-루프는 큐가 empty가 되면 종료되고, 루프가 처음 시작하여 끝날 때까지 연속적으로 방문된 정점들이 1개의 연결성분 구성
- Line 18~19: 큐에서 다음 방문할 정점 j를 삭제한 후, j를 출력하고, line 20의 for-루프는 정점 j에 인접해 있지만 아직 방문 안된 정점들을 큐에 삽입

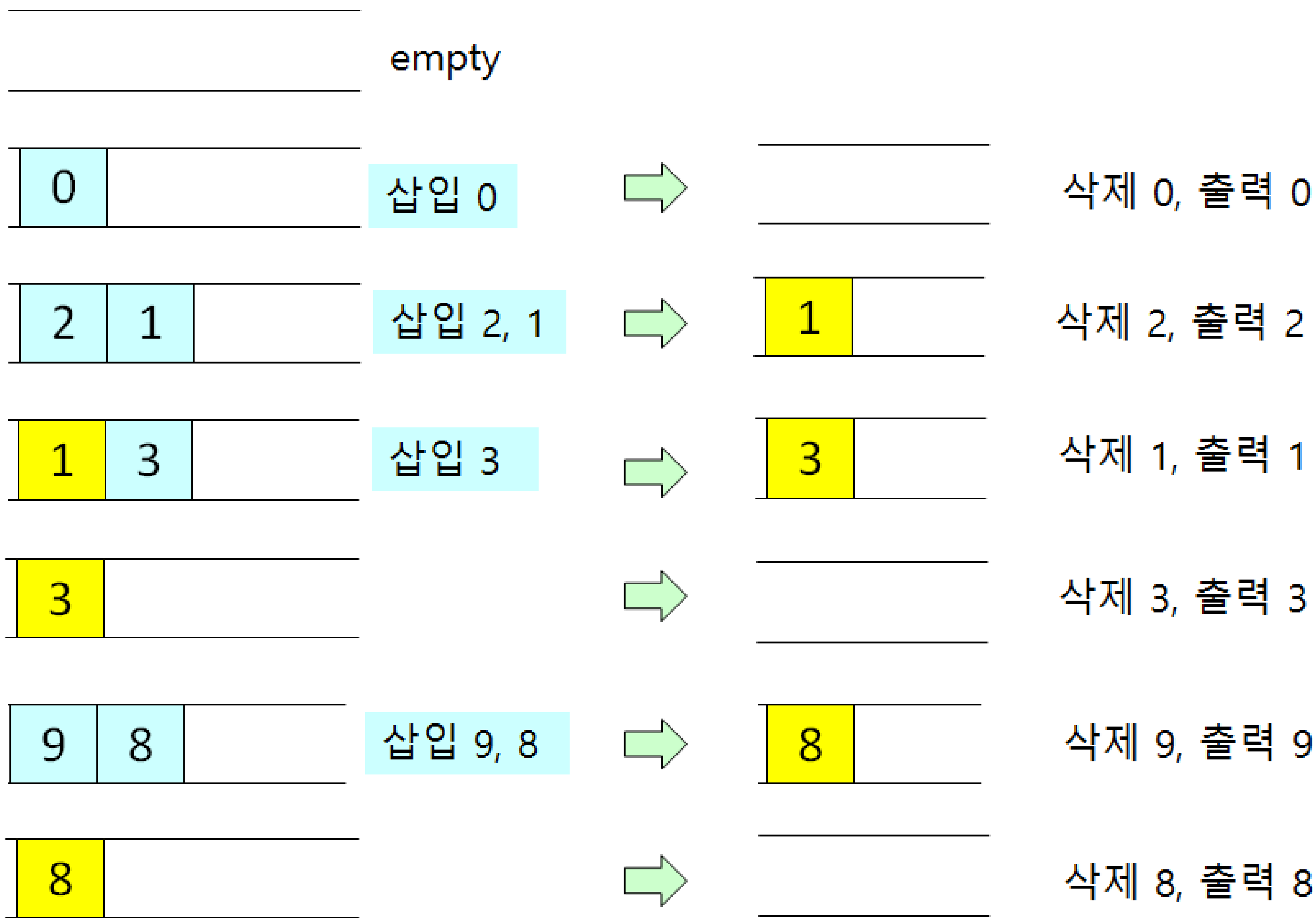
BFS 수행과정





방문순서	visited[]	출력
①	visited[0] = true	0
②	visited[2] = true	2
③	visited[1] = true	1
④	visited[3] = true	3
⑤	visited[9] = true	9
⑥	visited[8] = true	8
⑦	visited[4] = true	4
⑧	visited[5] = true	5
⑨	visited[7] = true	7
⑩	visited[6] = true	6

- bfs(0)부터 BFS 클래스가 수행되며 첫번째 연결성분의 정점들을 모두 방문할 때까지 큐에 정점들이 삽입, 삭제되며 정점들이 출력(방문)될 때의 큐의 상태



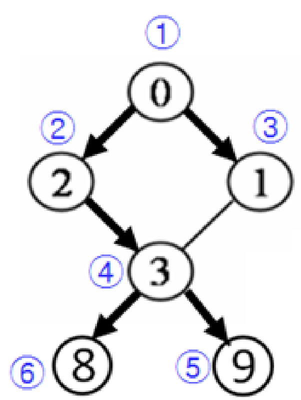
프로그램 수행결과

Problems @ Javadoc Console

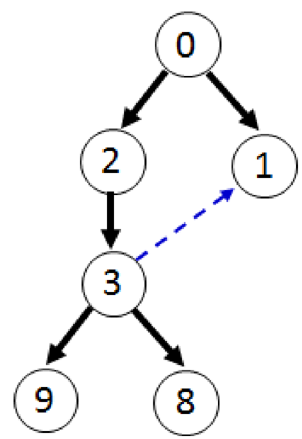
```
<terminated> main (26) [Java Application] C:₩Program Files₩Java₩jdk1.8.0_40₩bin₩javaw.exe
BFS(0) 방문 순서:
0 2 1 3 9 8 4 5 7 6
```

9.2.2 너비우선탐색(BFS)

- **(a)의 그래프에서 BFS 방문순서대로 정점 0부터 위에서 아래방향으로 그려보면 (b)와 같은 트리가 만들어짐**
  - 실선은 탐색하며 처음 방문할 때 사용된 그래프의 간선이고, 점선은 교차간선(Cross Edge)으로서 탐색 중 이미 방문된 정점에 도달한 경우를 나타냄
- **그래프가 1개의 연결성분으로 되어 있을 때 BFS를 수행하며 만들어지는 트리: 너비우선 신장트리 (Breadth First Spanning Tree)**



(a)



(b)

수행시간

- BFS는 각 정점을 한번씩 방문하며, 각 간선을 한번씩만 사용하여 탐색하기 때문에O(N+M)의 수행시간이 소요
- BFS와 DFS는 정점의 방문순서나 간선을 사용하는 순서만 다를 뿐이다.

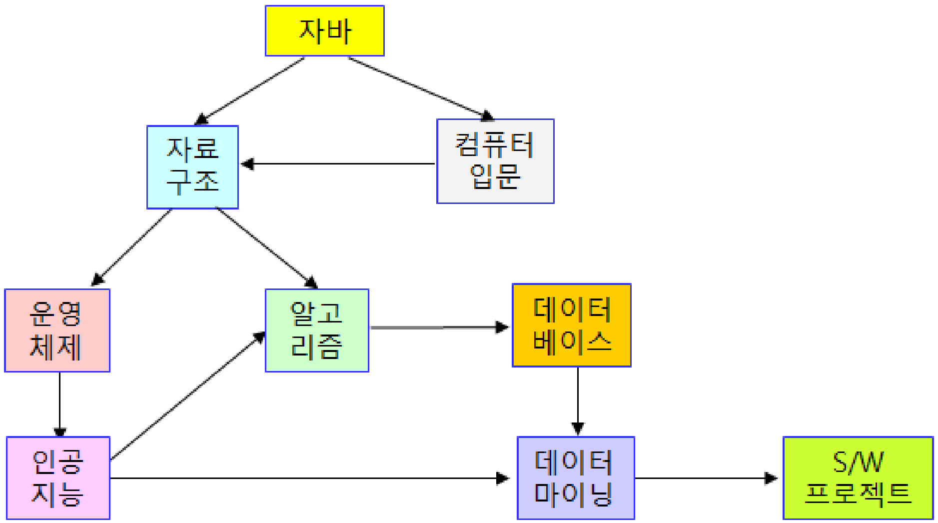
DFS와 BFS로 수행 가능한 그래프 응용

응용	DFS	BFS
신장트리, 연결성분, 경로, 싸이클	√	√
최소 선분을 사용하는 경로		√
위상정렬, 이중연결성분, 강연결성분	√	

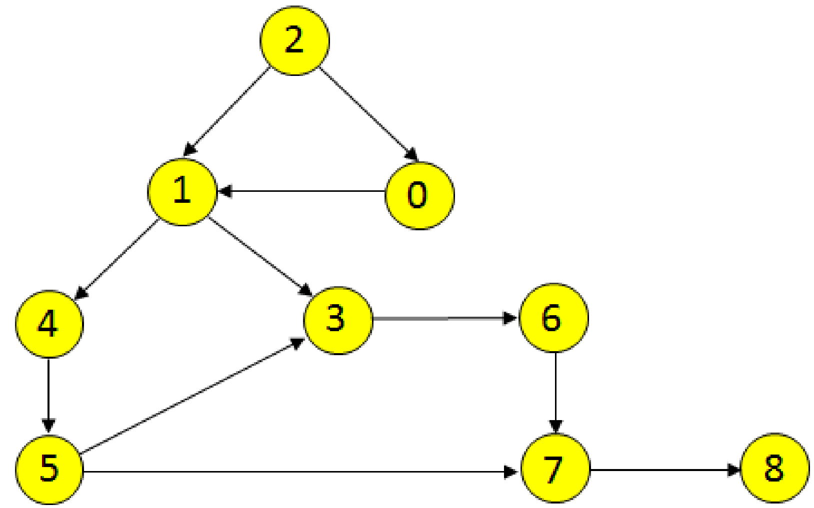
9.3 기본적인 그래프 알고리즘

- **9.3.1 위상정렬(Topological Sort)**
  - 싸이클이 없는 방향그래프(Directed Acyclic Graph, DAG)에서 정점을 선형순서(즉, 정점들을 일렬)로 나열하는 것
- **위상정렬 결과는 그래프의 각 간선 <u, v>에 대해 u가 v보다 반드시 앞서 나열되어야 함**

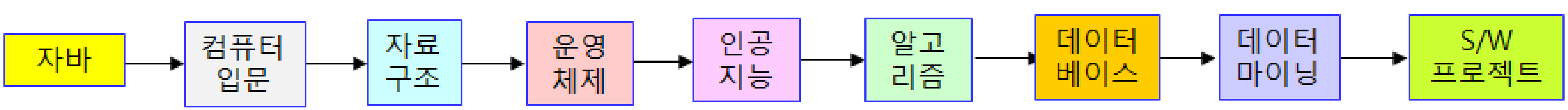
[예제] (a)는 교과과정의 선수과목 관계도이고, (c)는 교과목 수강순서도이다.



(a) 선수과목 관계도



(b) 그래프



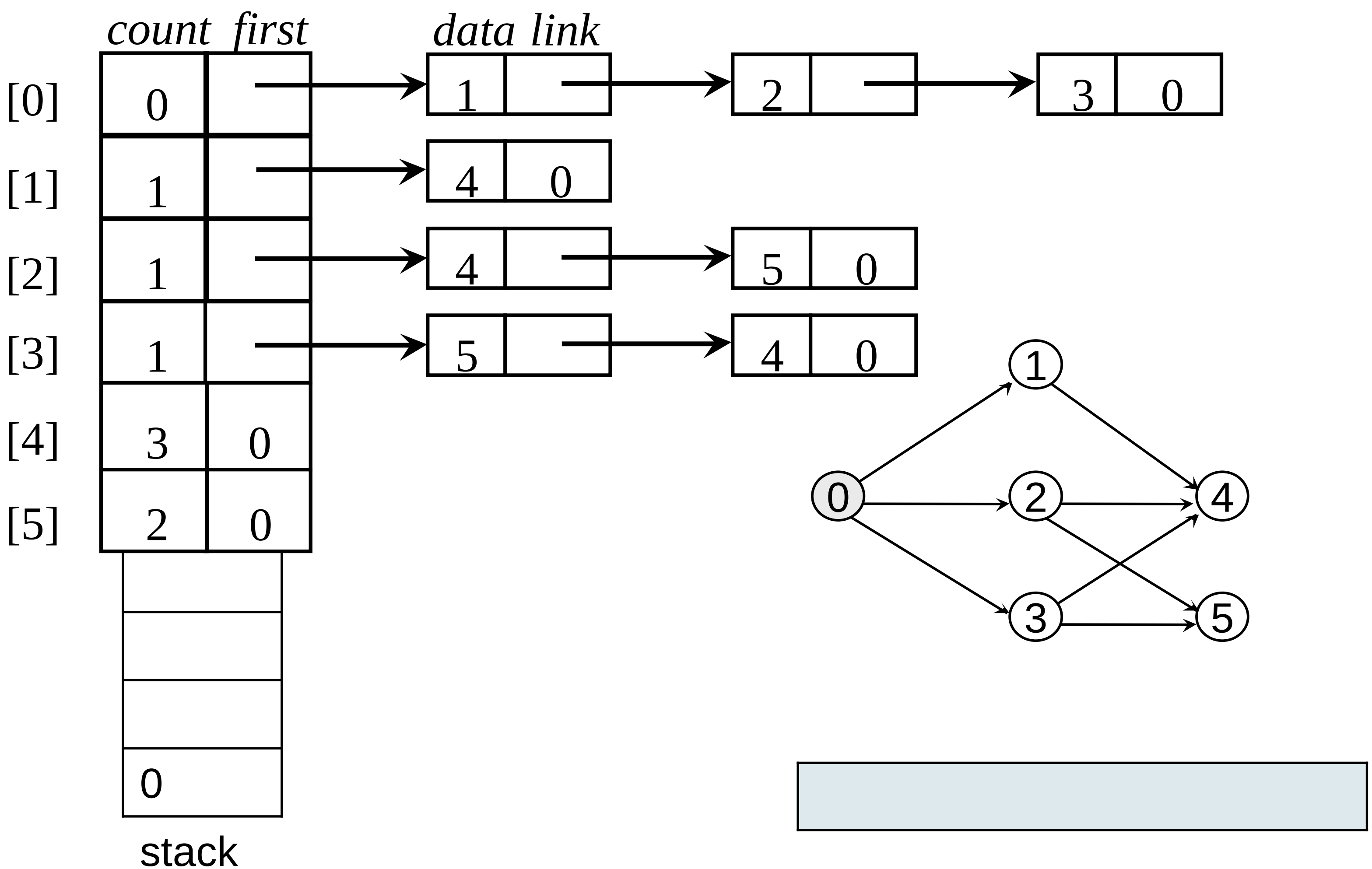
(c) 교과목 수강순서

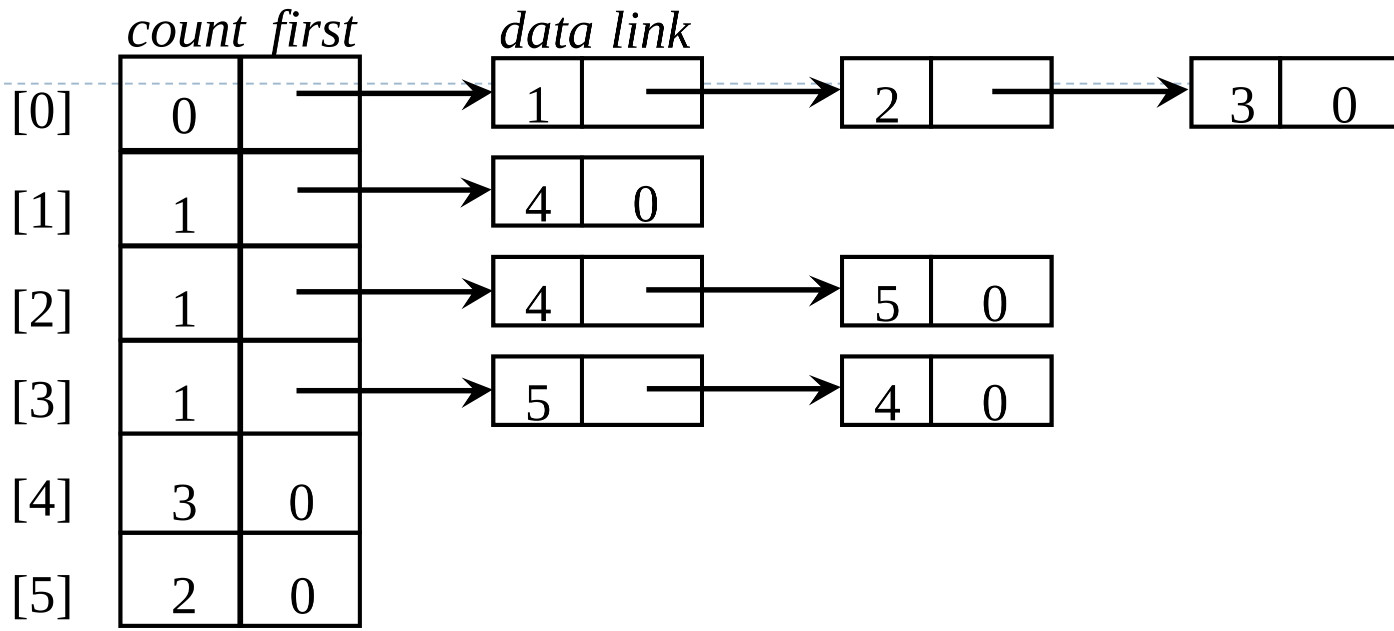
9.3.1 위상 정렬

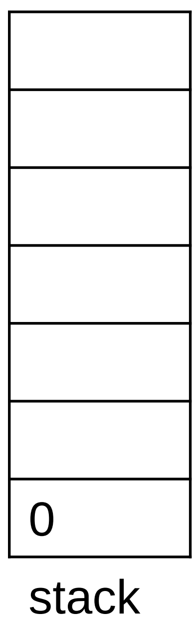
- **주어진 그래프에 따라 여러 개의 위상정렬이 존재할 수 있음**
- **일반적으로 작업(Task)들 사이에 의존관계가 존재할 때 수행 가능한 작업 순서를 도식화하는데에 위상정렬을 사용**
- **위상정렬 찾기**
  - 그래프에서 진입차수가 0인 정점 v로부터 시작하여 v를 출력하고 v를 그래프에서 제거하는 과정을 반복하는 순방향 방법
  - 진출차수가 0인 정점 v를 출력하고 v를 그래프에서 제거하는 과정을 반복하여 얻은 출력 리스트를 역순으로 만들어 결과를 얻는 역방향 방법

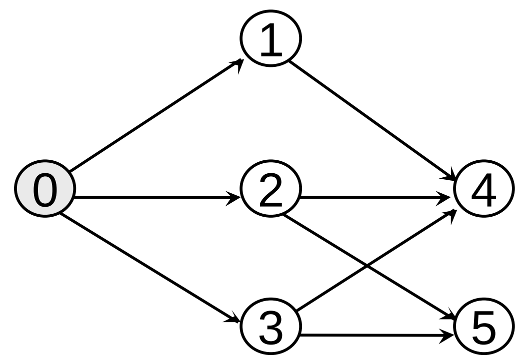
9.3.1 위상 정렬

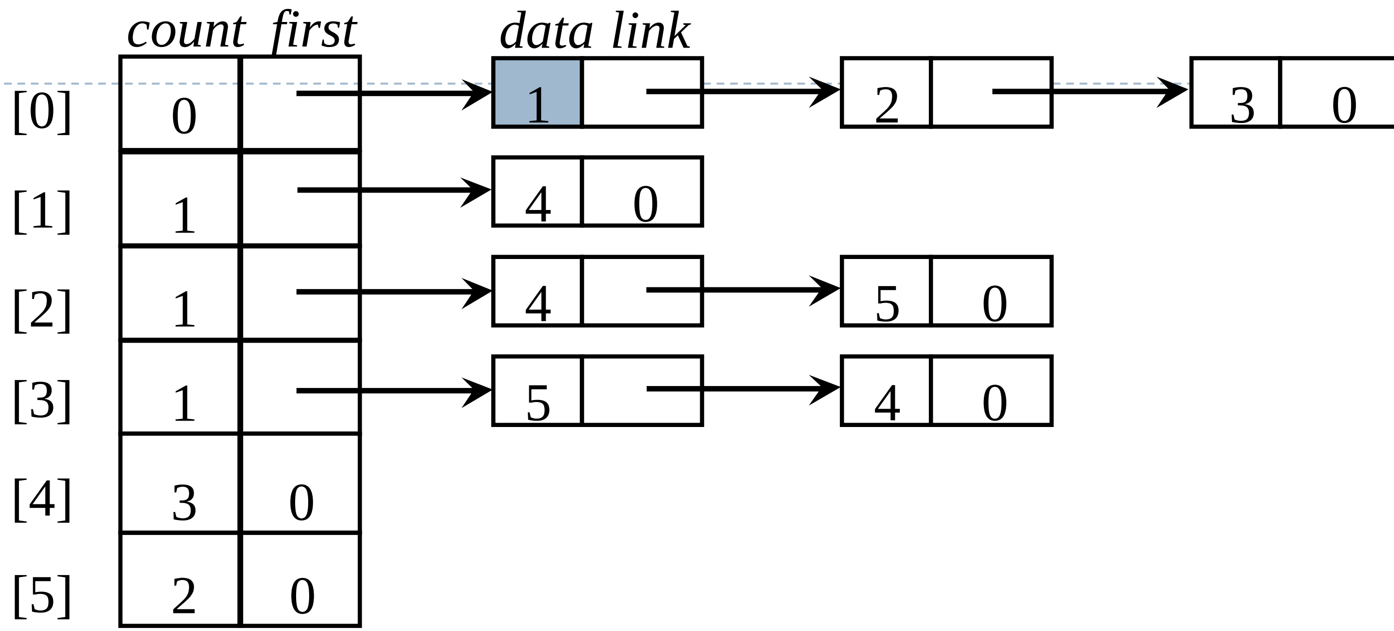
- **순방향 방법**은 각 정점의 진입차수를 알아야 하므로 인접리스트를 각 정점으로 진입하는 정점들의 리스트로 바꾸어야



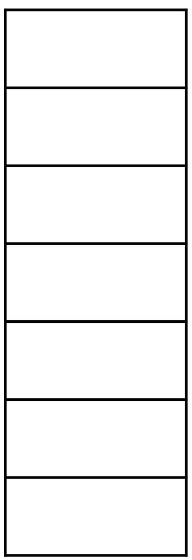

count first
data link
[0] 0
1
2
3 0
[1] 1
4 0
[2] 1
4
5 0
[3] 1
5
4 0
[4] 3 0
[5] 2 0


0
stack

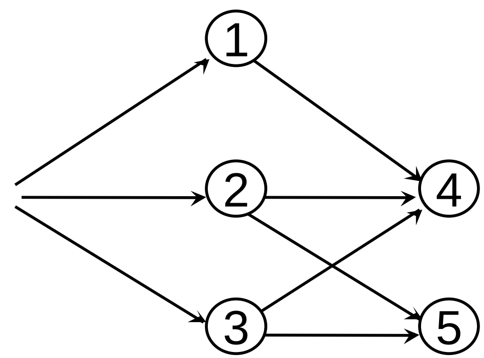

0
1
2
3
4
5



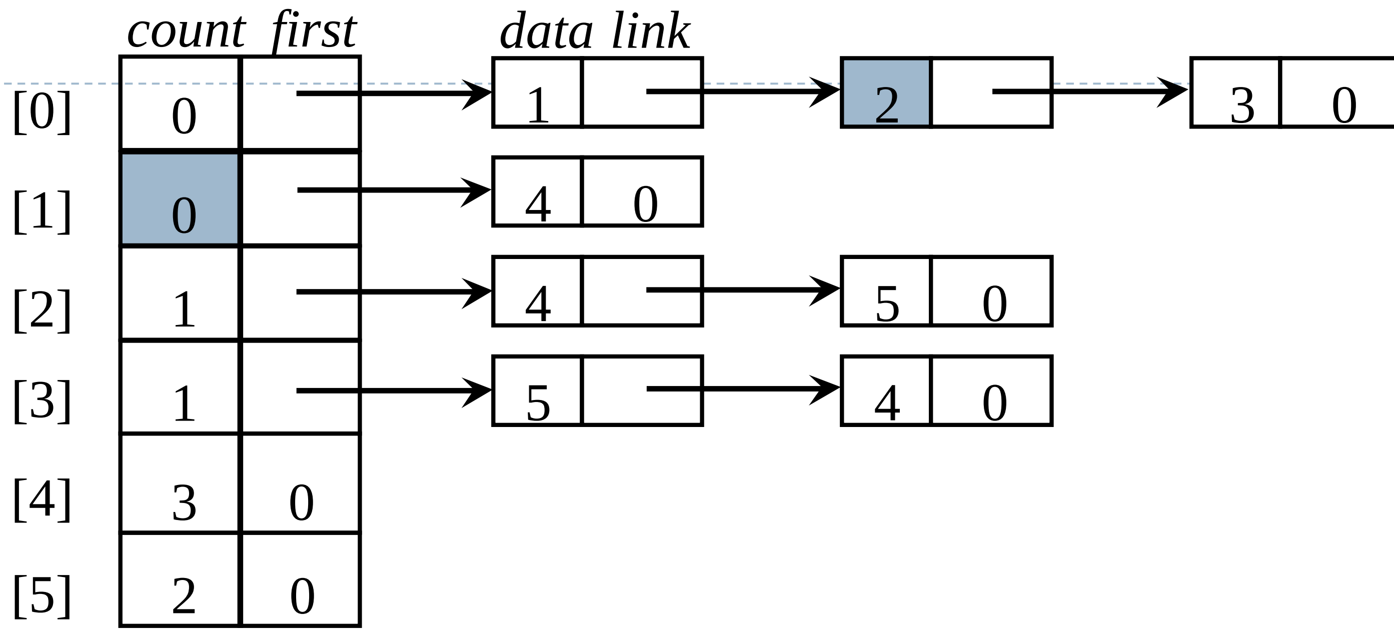
0



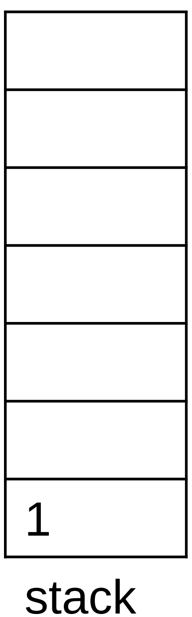
stack

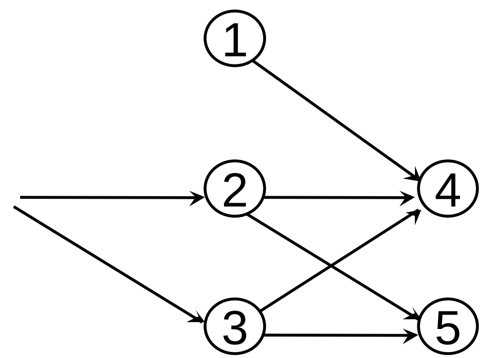


0

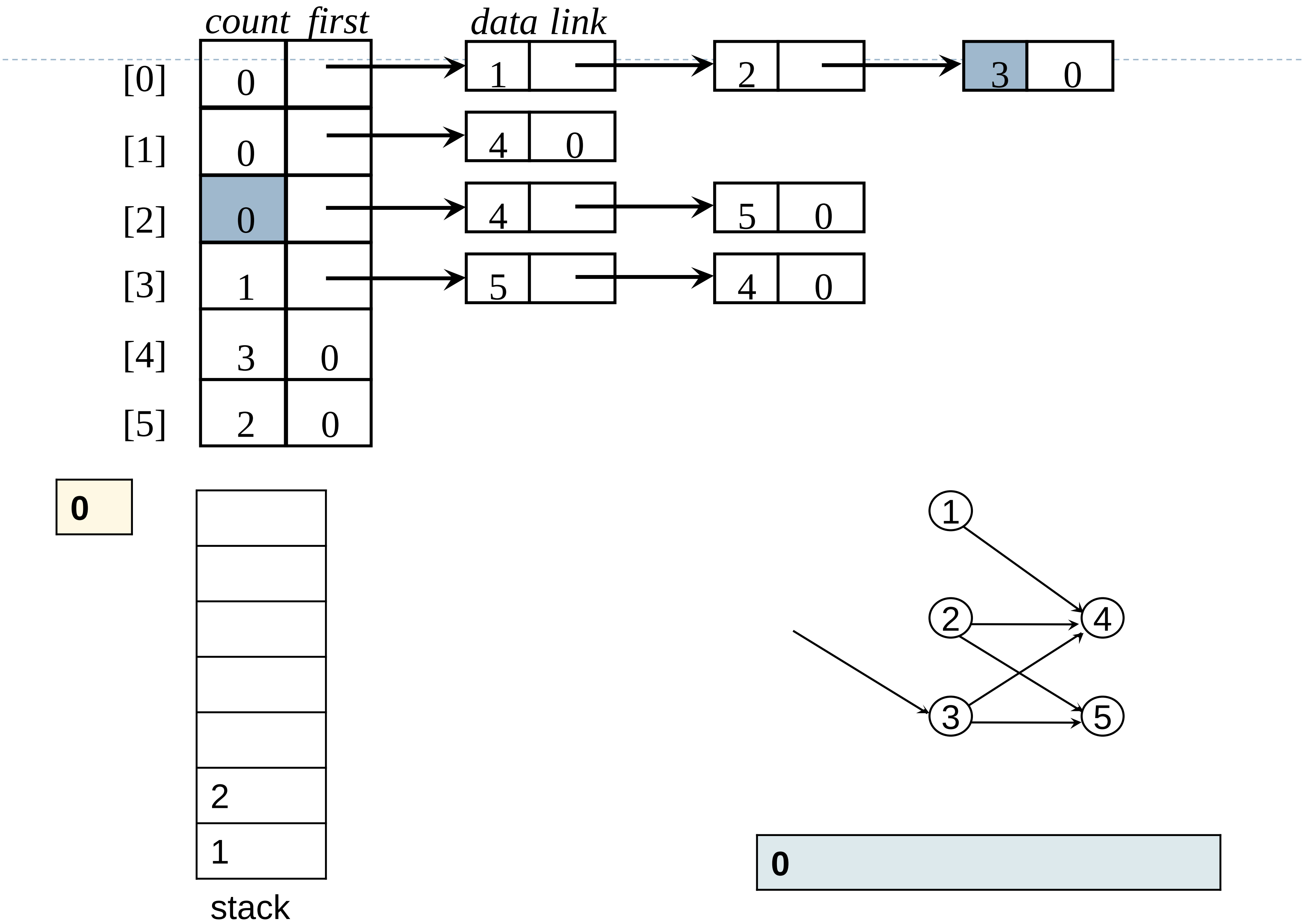

count first
data link
[0] 0
[1] 0
[2] 1
[3] 1
[4] 3 0
[5] 2 0
1
2
3 0
4 0
4
5 0
5
4 0

0

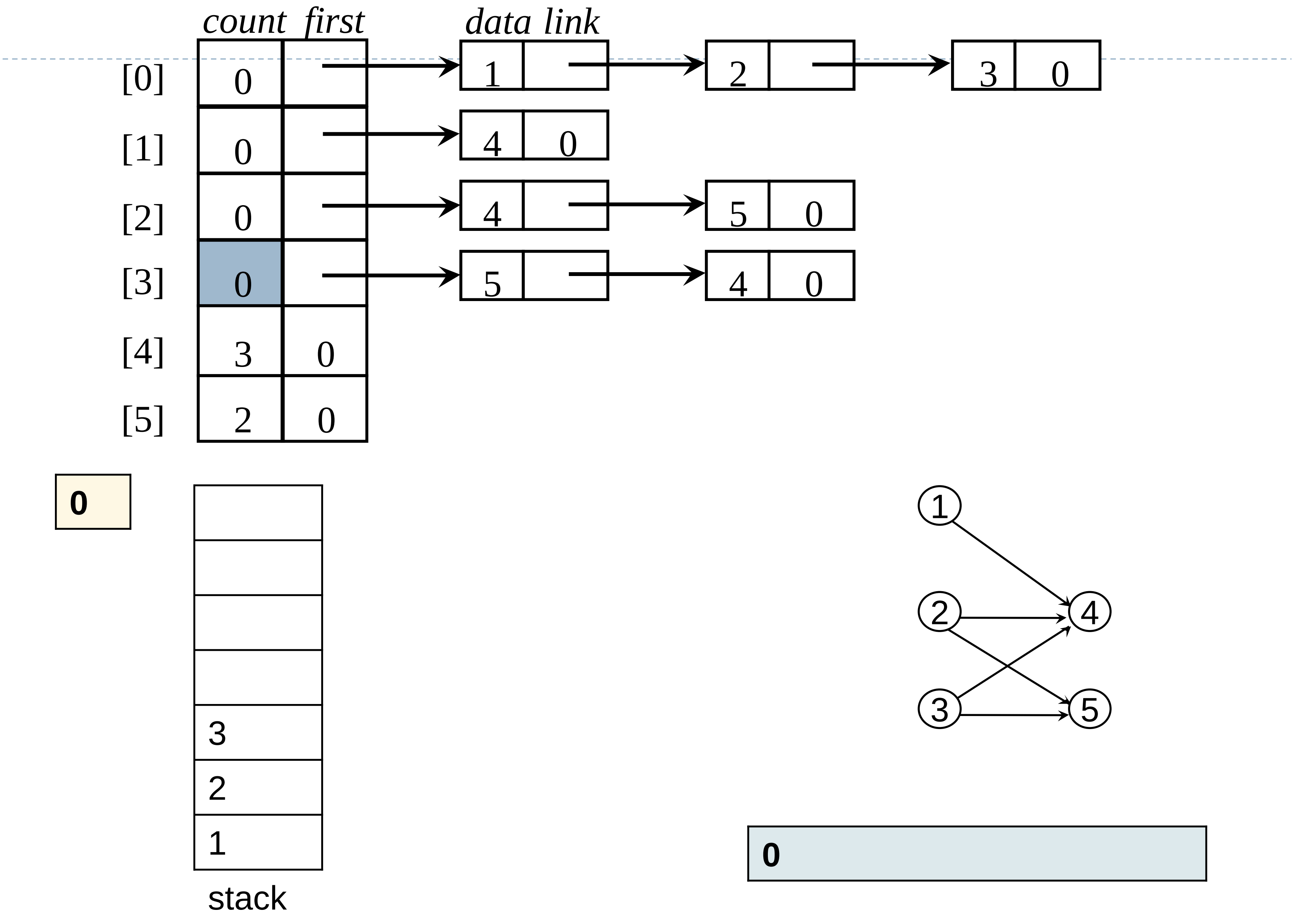

1
stack

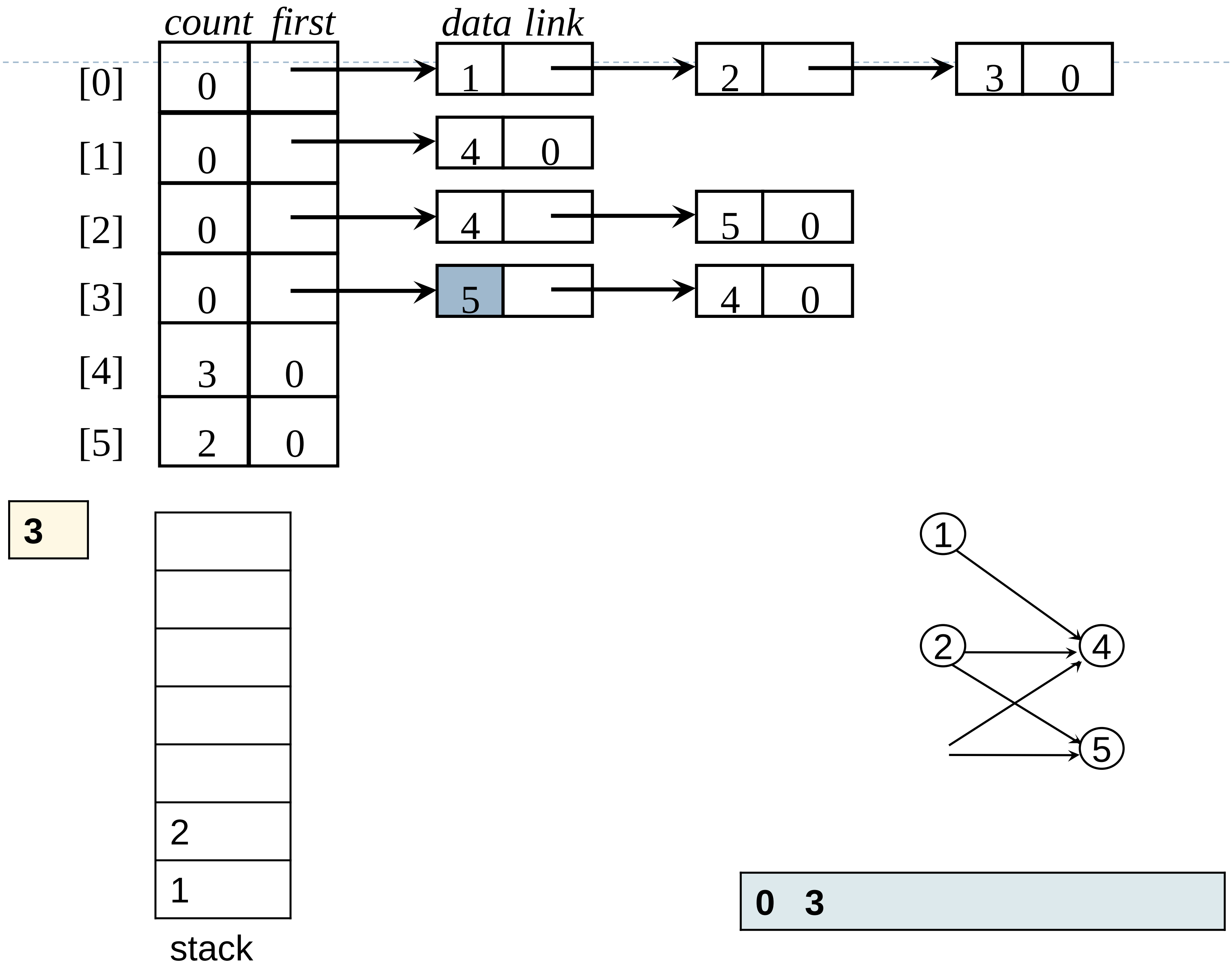

1
2
3
4
5

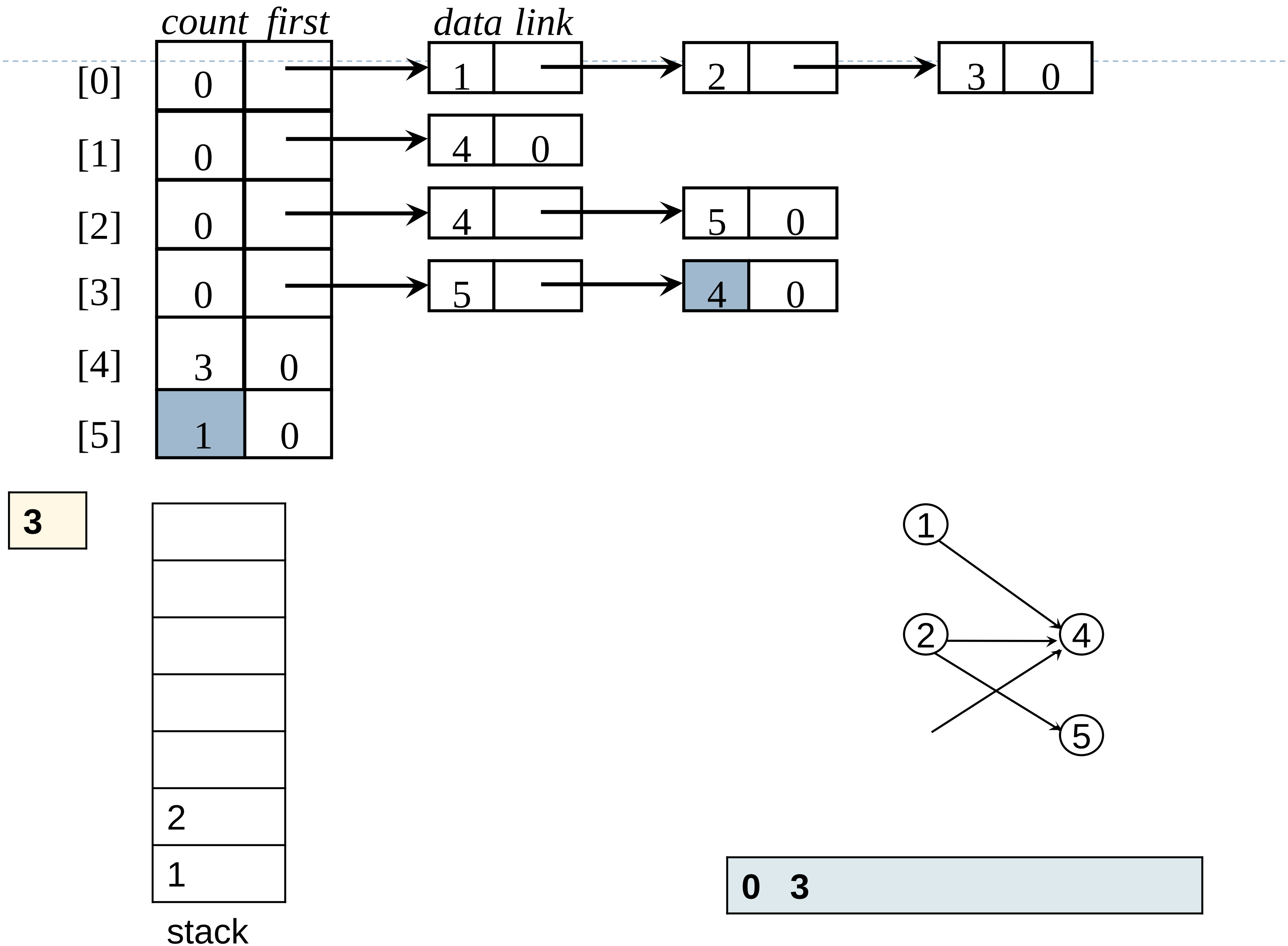
0

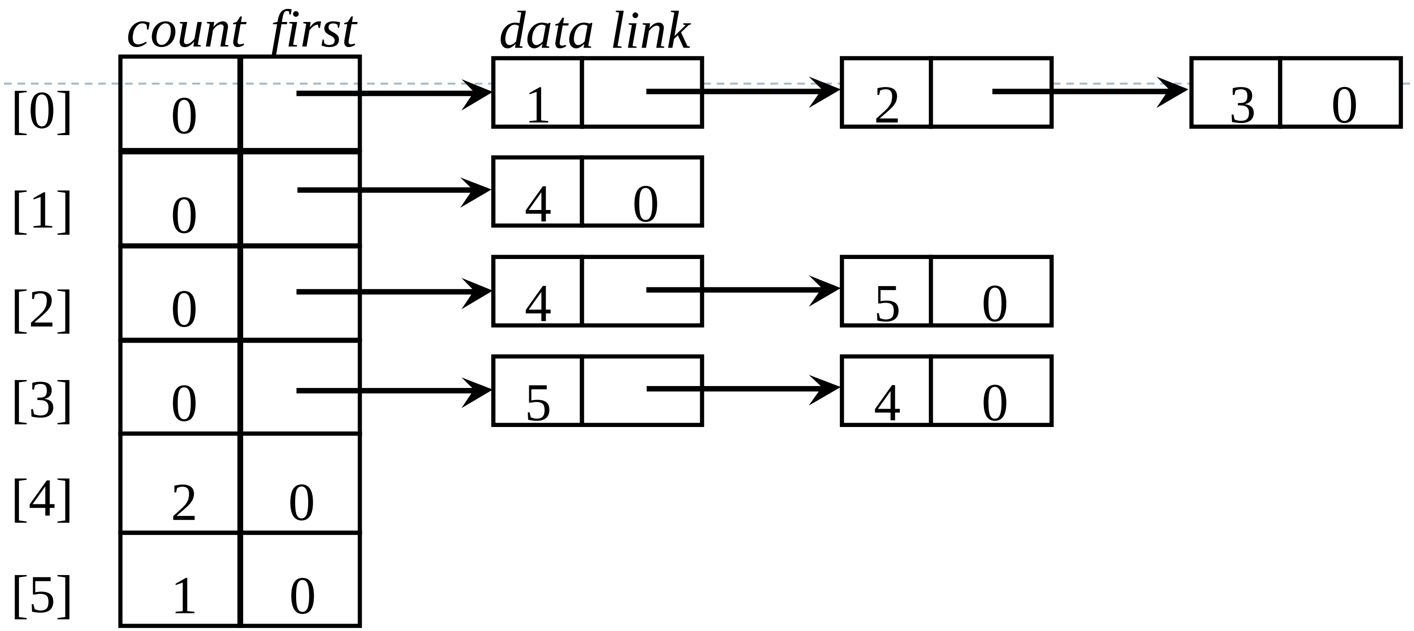


count first
data link
[0]
[1]
[2]
[3]
[4]
[5]
0
stack
0

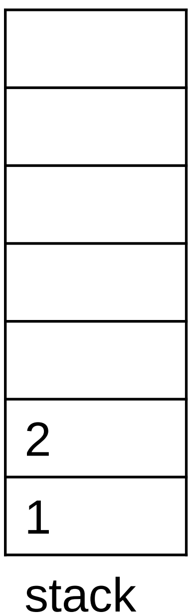

count first
data link
[0] 0
[1] 0
[2] 0
[3] 0
[4] 3 0
[5] 2 0
1
2
3 0
4 0
4
5 0
5
4 0
0
3
2
1
stack
1
2
3
4
5
0

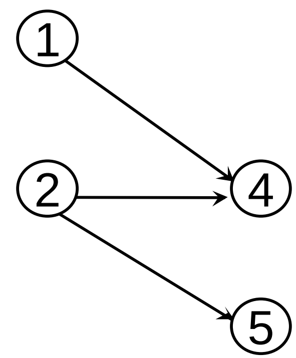

count first
data link
[0]
[1]
[2]
[3]
[4]
[5]
3
2
1
stack
0 3


count first
data link
[0] 0
1
2
3 0
[1] 0
4 0
[2] 0
4
5 0
[3] 0
5
4 0
[4] 3 0
[5] 1 0
3
2
1
stack
1
2
4
5
0 3

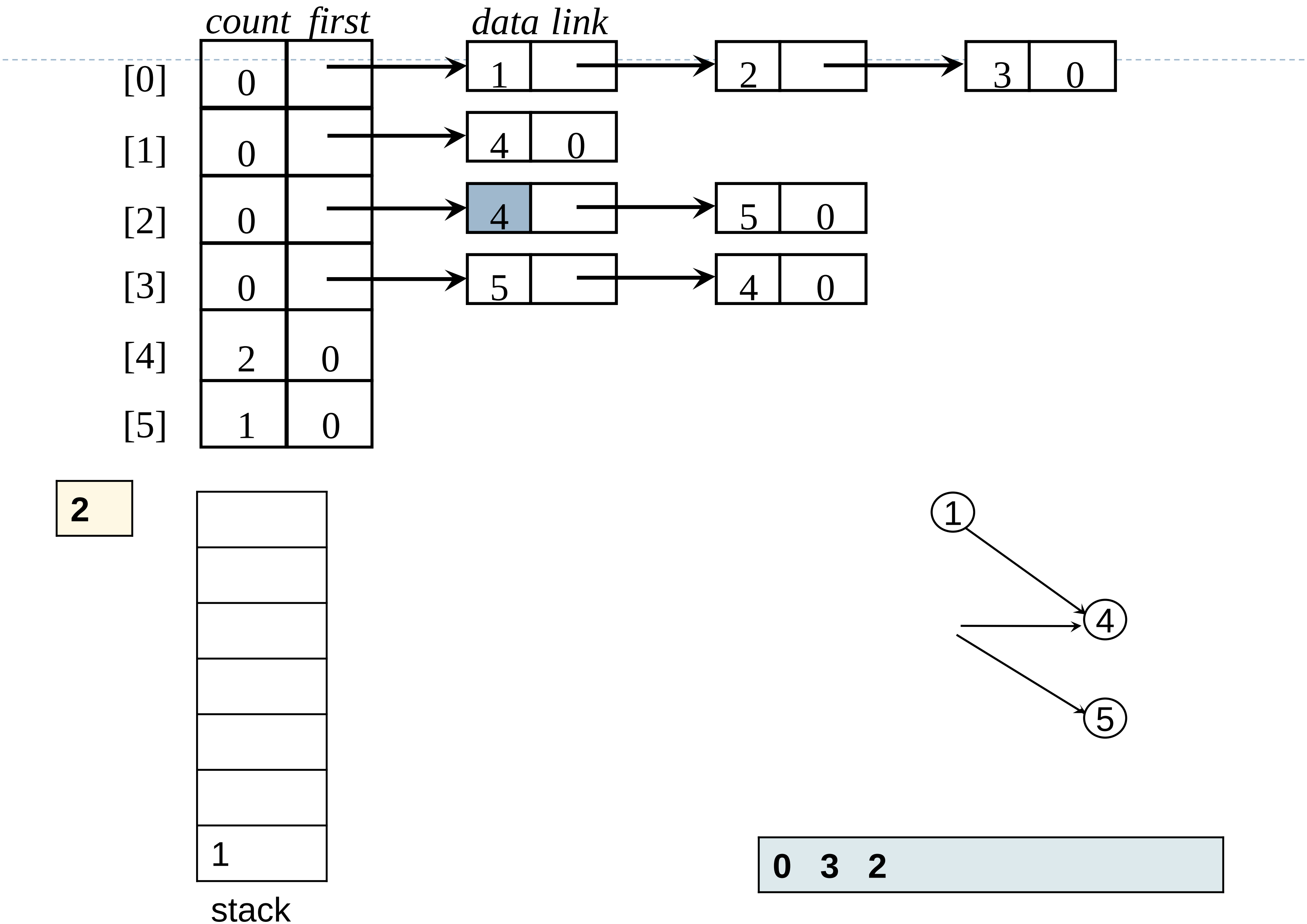

count first
data link
[0] 0
[1] 0
[2] 0
[3] 0
[4] 2 0
[5] 1 0
1
2
3 0
4 0
4
5 0
5
4 0

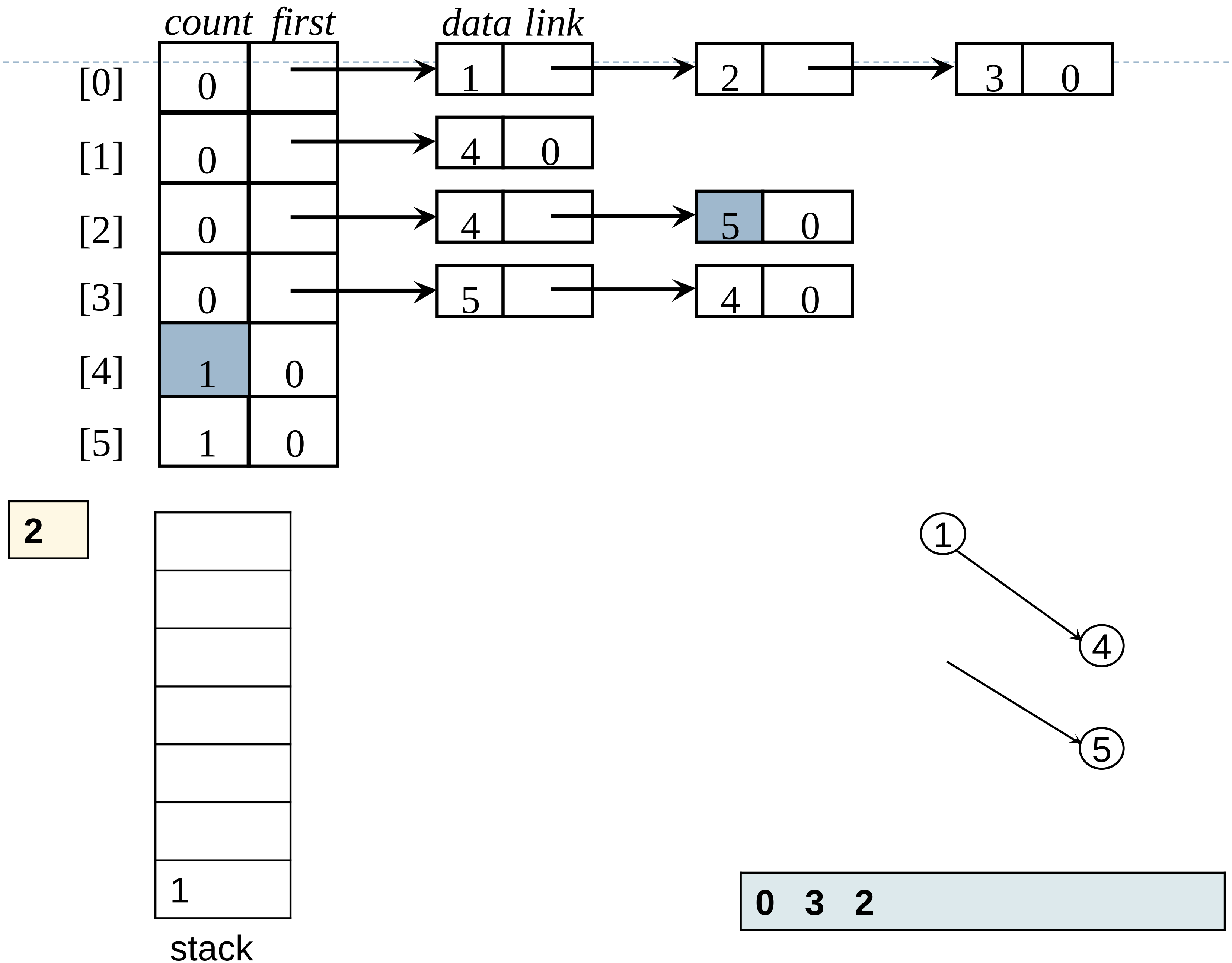
3

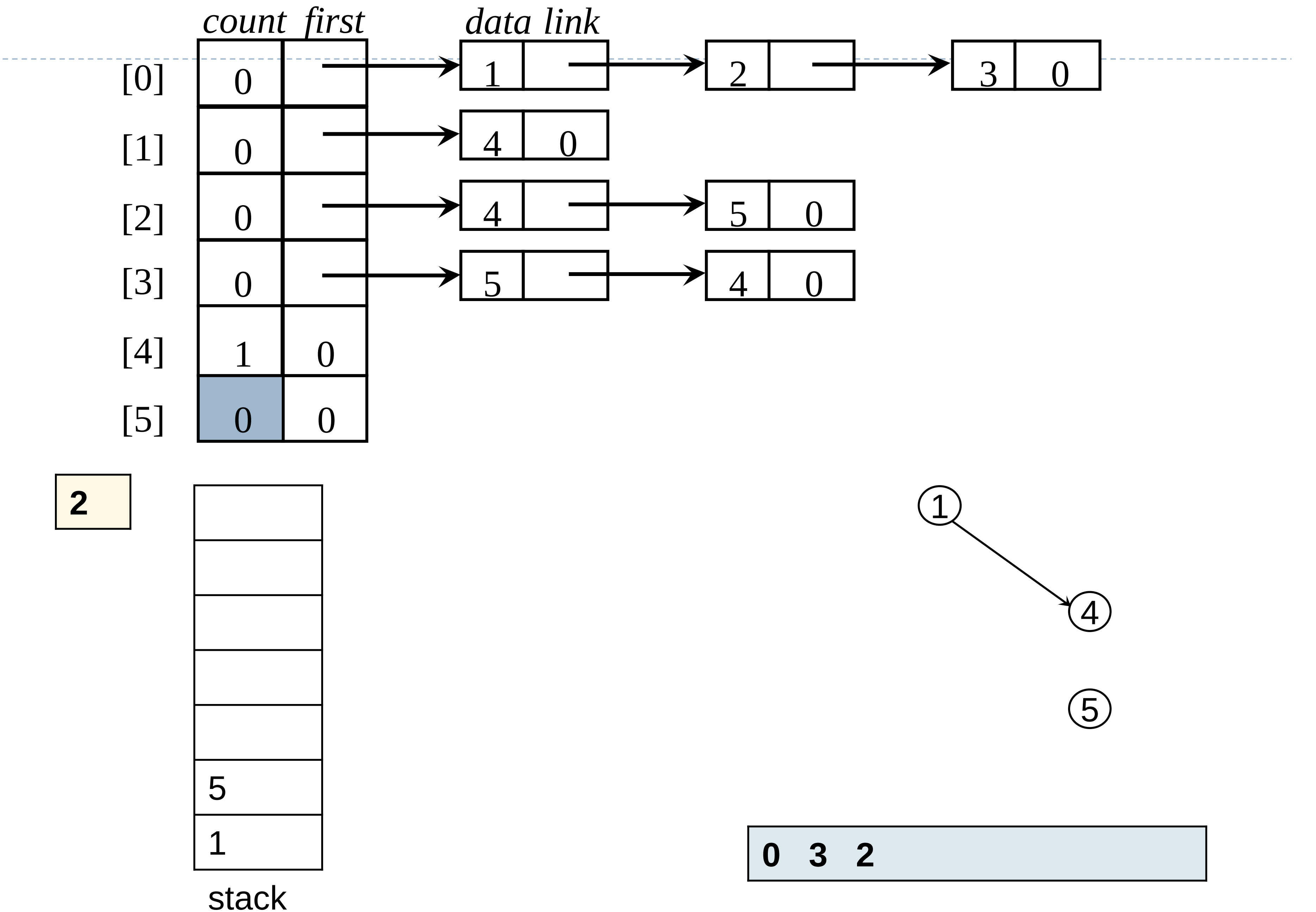

2
1
stack

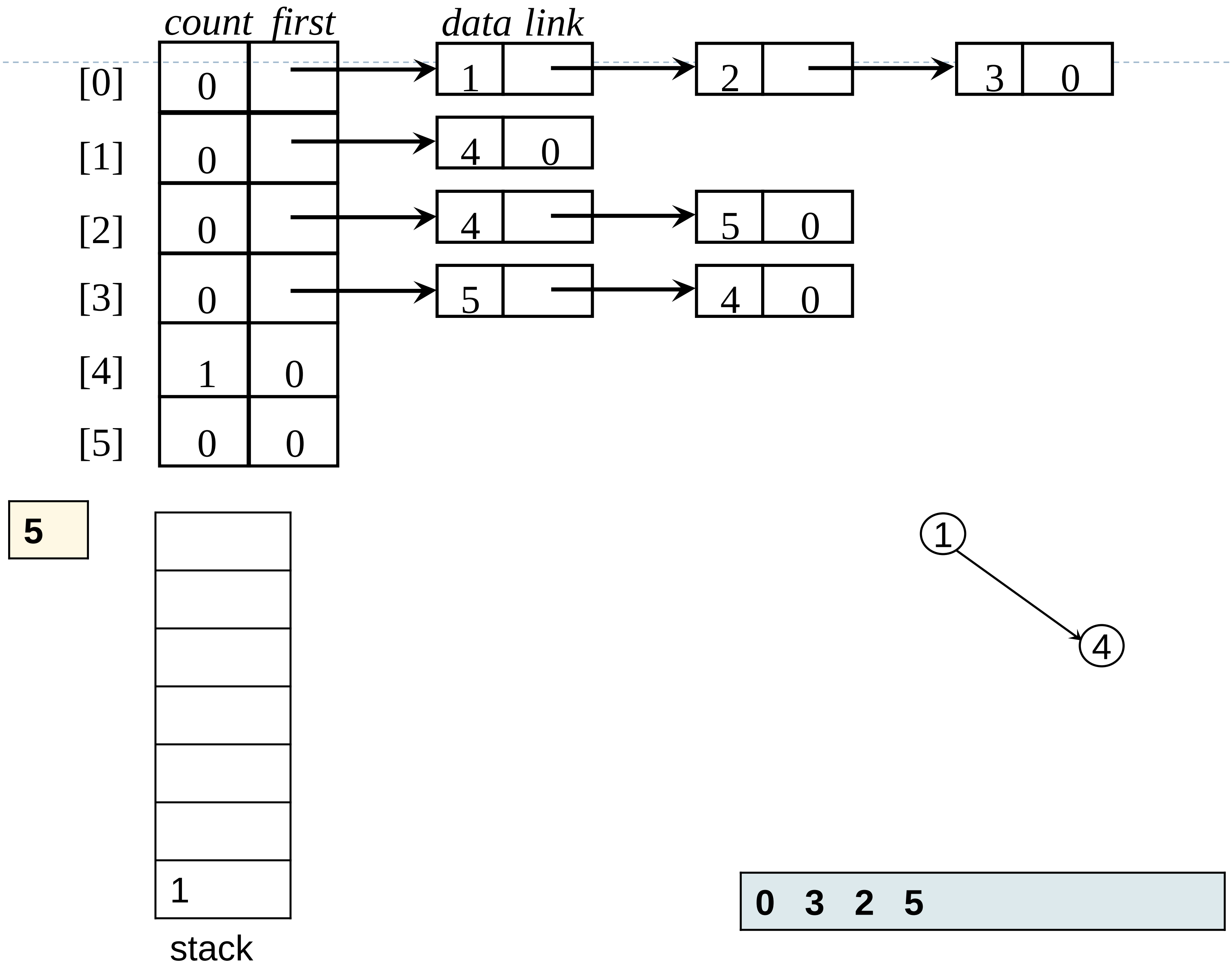

1
2
4
5

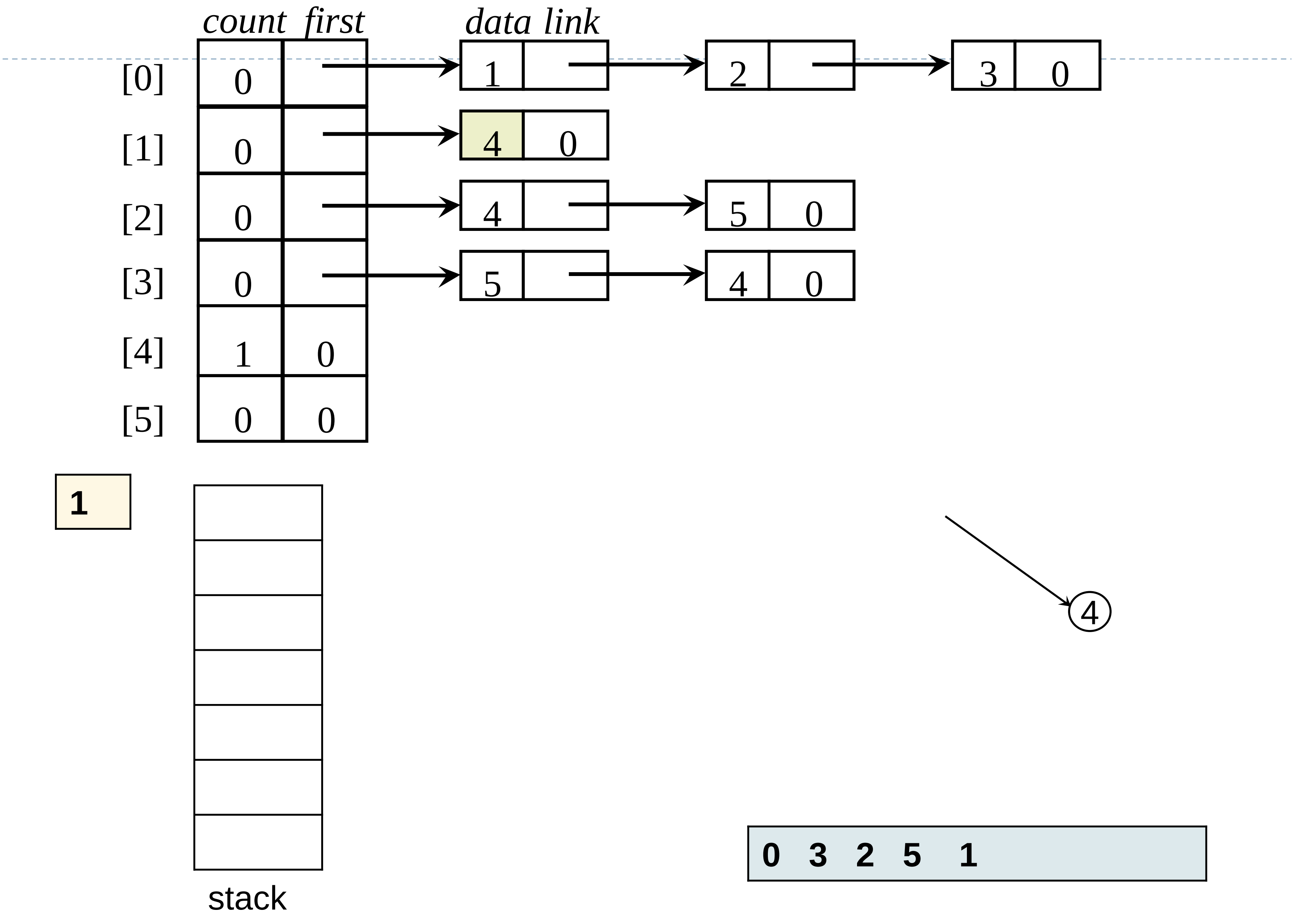
0 3

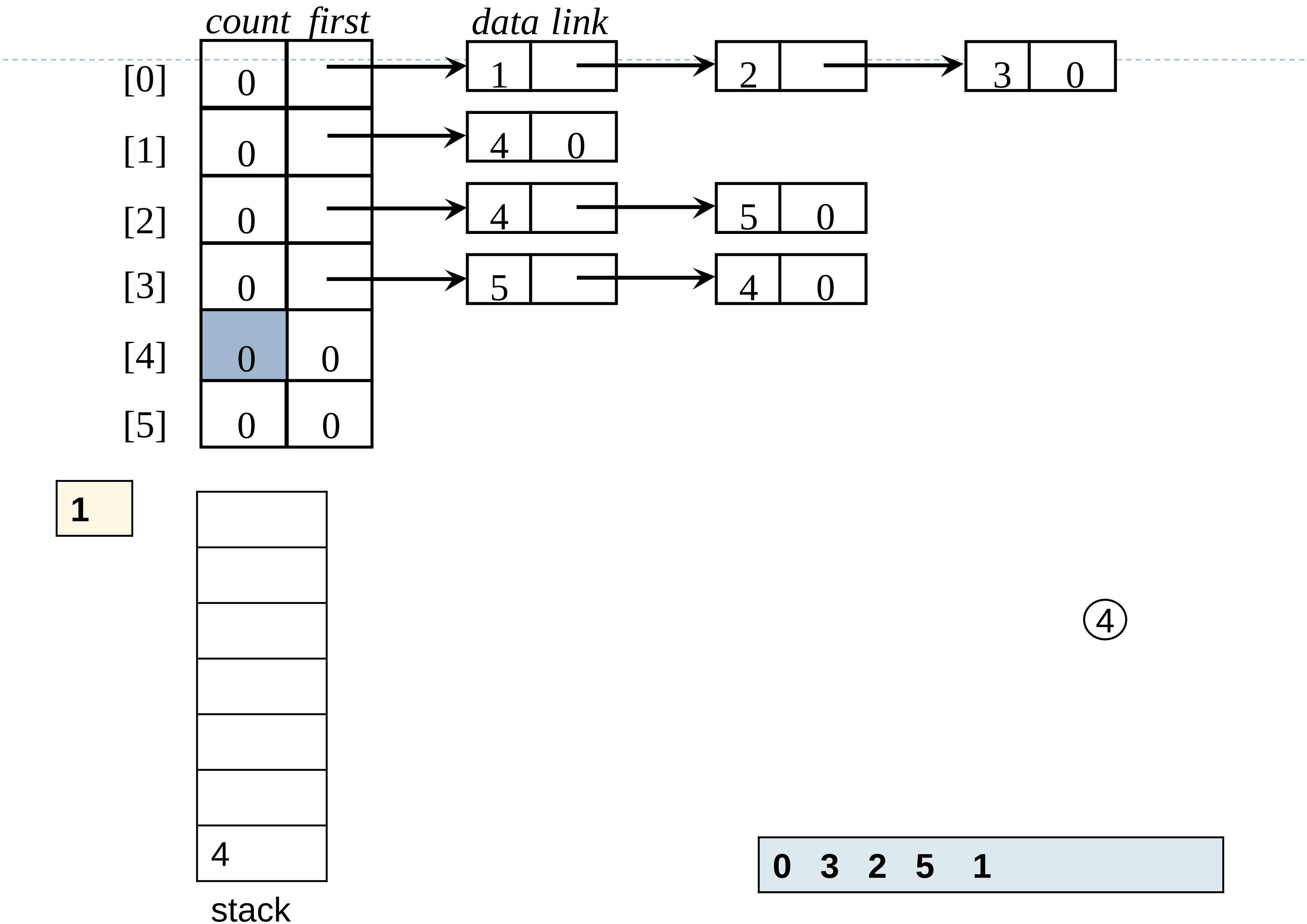

count first
data link
[0]
[1]
[2]
[3]
[4]
[5]
0
0
0
0
2 0
1 0
1
2
3 0
4 0
4
5 0
5
4 0
2
1
stack
1
4
5
0 3 2

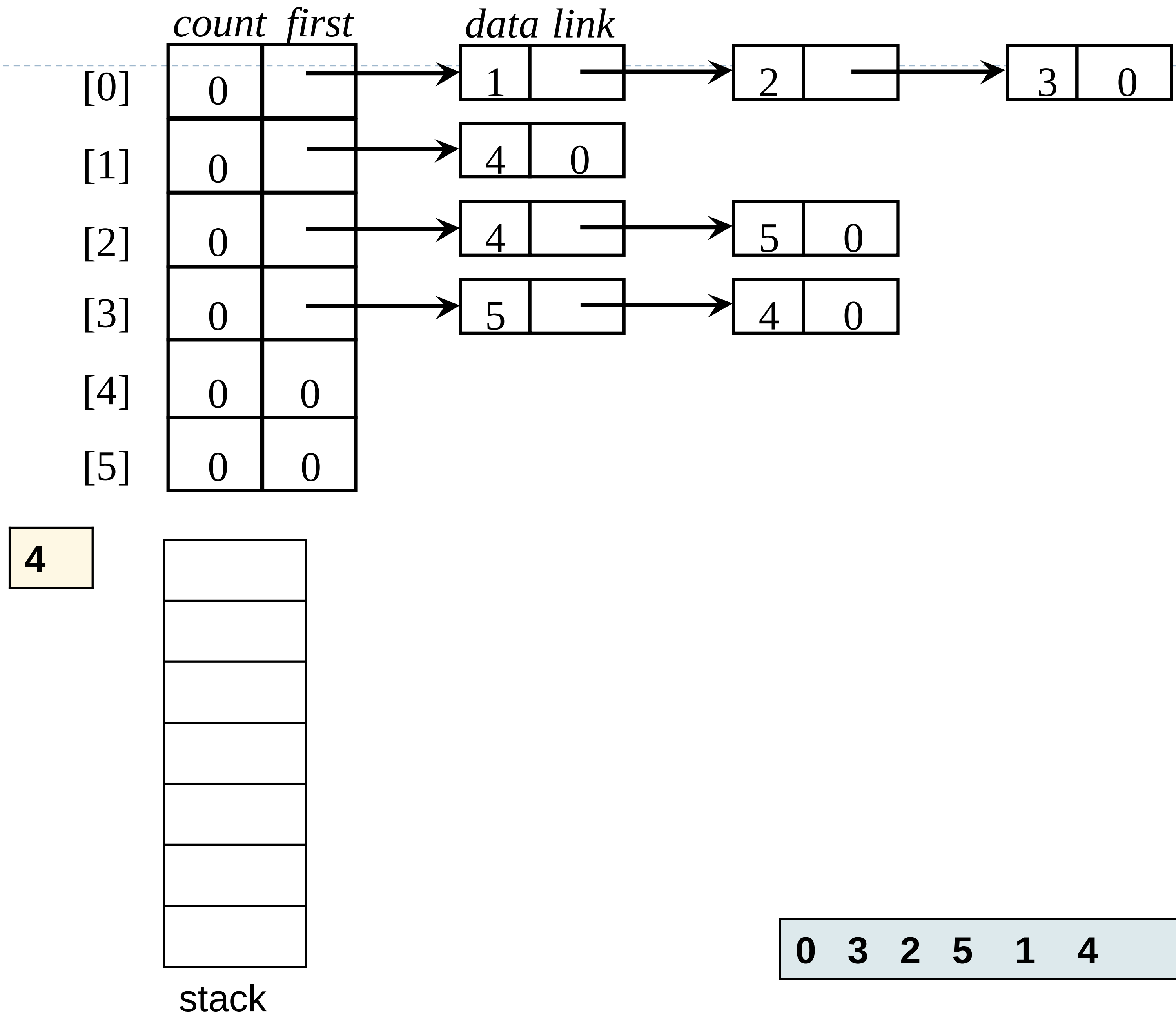

count first
data link
[0]
[1]
[2]
[3]
[4]
[5]
0
0
0
0
1 0
1 0
1
2
3 0
4 0
4
5 0
5
4 0
2
1
stack
1
4
5
0 3 2


count first
data link
[0]
[1]
[2]
[3]
[4]
[5]
0
0
0
0
1 0
0 0
1
2
3 0
4 0
4
5 0
5
4 0
2
5
1
stack
1
4
5
0 3 2


count first
data link
[0]
[1]
[2]
[3]
[4]
[5]
0
0
0
0
1 0
0 0
1
2
3 0
4 0
4
5 0
5
4 0
5
1
stack
1
4
0 3 2 5


count first
data link
[0]
[1]
[2]
[3]
[4]
[5]
0
0
0
0
1 0
0 0
1
2
3 0
4 0
4
5 0
5
4 0
1
stack
4
0 3 2 5 1


count first
data link
[0]
[1]
[2]
[3]
[4]
[5]
0
0
0
0
0 0
0 0
1
2
3 0
4 0
4
5 0
5
4 0
1
4
stack
4
0 3 2 5 1



0 3 2 5 1 4

9.3.1 위상 정렬

- **역방향 방법은 주어진 인접리스트를 입력에 대해 변형된 DFS를 수행하여 출력 리스트를 작성한 후에 리스트를 역순으로 만들어 위상정렬**

[핵심 아이디어] DFS를 수행하며 각 정점 v의 인접한 모든 정점들의 방문이 끝나자마자 v 를 리스트에 추가한다. 리스트가 완성되면 리스트를 역순으로 만든다

- "v의 인접한 모든 정점들의 방문이 끝나자마자 v 를 리스트에 추가한다"

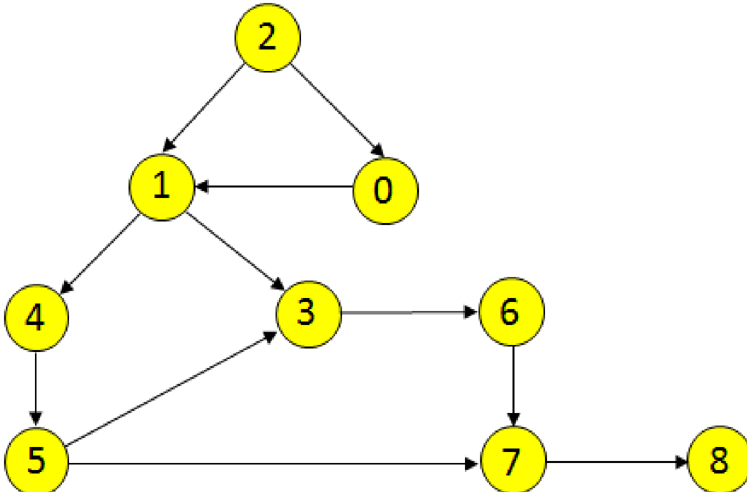
⇨ v가 추가되기 전에 v에 인접한 모든 정점들이 이미 리스트에 추가되어 있음을 뜻함

- 따라서 리스트가 완성되어 이를 역순으로 만들면 위상정렬 결과를 얻음

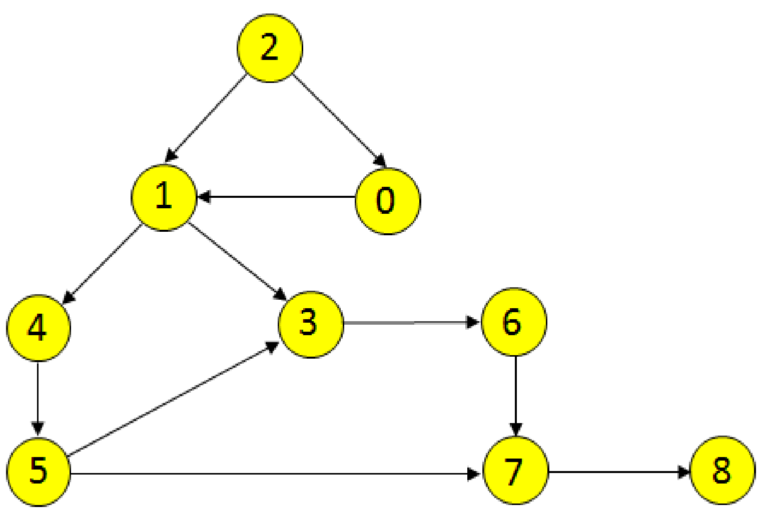
TopologicalSort 클래스

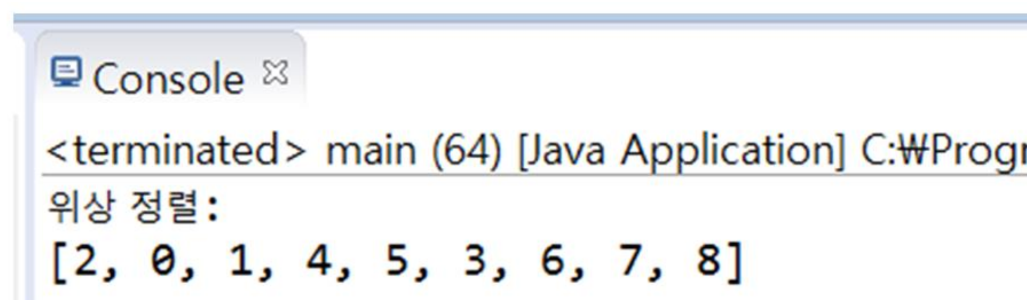
```java
import java.util.*;
public class TopologicalSort {
    int N;                          // 그래프의 정점 수
    boolean[] visited;              // DFS 수행 중 방문여부 체크 용
    List<Integer>[] adjList;        // 인접리스트 형태의 입력 그래프
    List<Integer> sequence;         // 위상 정렬 순서를 담을 리스트
    public TopologicalSort(List<Integer>[] graph) {  //생성자
        N = graph.length;
        visited = new boolean[N];
        adjList = graph;
        sequence = new ArrayList<>();
    }
    public List<Integer> tsort() {       // 위상정렬을 위한 DFS 수행
        for (int i = 0; i < N; i++) if (!visited[i]) dfs(i);
        Collections.reverse(sequence); // sequence를 역순으로 만들기
        return sequence;
    }
    public void dfs(int i) { // DFS 수행
        visited[i] = true;
        for (int v : adjList[i]) { // i 의 방문이 끝나고 앞으로 방문해야하는 각 정점 v에 대해
            if (!visited[v]) dfs(v);
        }
        sequence.add(i);   // i 에서 진출하는 간선이 더 이상 없으므로 i를 sequence에 추가
    }
}
```

- **Line 13의 tsort() 메소드: line 18의 dfs() 메소드를 호출하며, line 15~16과 line 23을 제외하면 DFS 클래스와 거의 동일**
  - 단, DFS클래스에서는 Edge클래스를 사용하여 간선을 나타냈지만, 여기서는 단순히 간선을 인접한 정점(int)으로 나타냄
- **예제**
  - 먼저 Line 14의 dfs(0)으로 시작하여, dfs(1), dfs(3), dfs(6), dfs(7)을 차례로 호출한 후에, dfs(8)이 호출
  - 이때 정점 8에선 더 이상 인접한 정점이 없으므로 line 20의 for-루프가 수행되지 않고 바로 line 23의 sequence.add(8)이 수행되어 '8'이 sequence에 가장 먼저 저장
  - 즉, 위상정렬순서의 가장 마지막 정점을 찾아서 sequence에 저장



- sequence.add(8)이 수행된 후 dfs()메소드가 리턴된 뒤, 정점 7에 대해 line 20의 for-루프에서 정점 7의 인접한 모든 정점들을 이미 방문했으므로, line 23에서 ‘7’을 sequence에 추가
- Line 20의 for-루프에서 더 이상 방문 안된 인접한 정점이 없으면 해당 정점을 sequence에 추가
- 최종적으로 line 15에서 sequence를 역순으로 만들어 line 16에서 위상정렬 결과 리턴



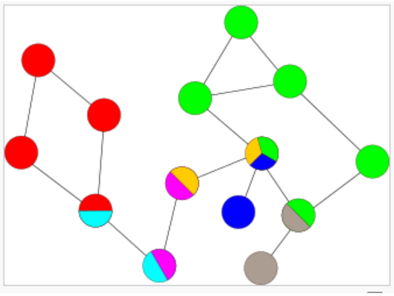


수행시간

- 위상정렬 알고리즘의 수행시간은 DFS의 수행시간과 동일한 O(N+M)
- 기본적으로 DFS를 수행하며 추가로 소요되는 시간은 line 23에서 정점을 리스트에 저장하고, 모든 탐색이 끝나면 리스트를 역순으로 만드는 시간으로 이는 O(N)
- 따라서 위상정렬 알고리즘의 수행시간은 O(N+M) + O(N) = O(N+M)

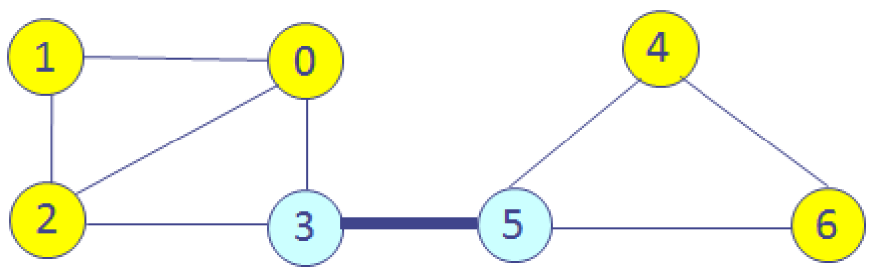
9-3-2 이중연결성분(Biconnected Component)

- **무방향그래프의 연결성분에서 임의의 두 정점들 사이에 적어도 두 개의 단순경로가 존재하는 연결성분**
  - 따라서 하나의 단순경로 상의 어느 정점 하나가 삭제되더라도 삭제된 정점을 거치지 않는 또 다른 경로가 존재하므로 연결성분내에서 정점들 사이의 연결이 유지
- **이중연결성분은 통신 네트워크 보안, 전력 공급 네트워크 등에서 네트워크의 견고성(Robustness)을 분석하는 주된 방법**



9-3-2 이중연결성분(Biconnected Component)

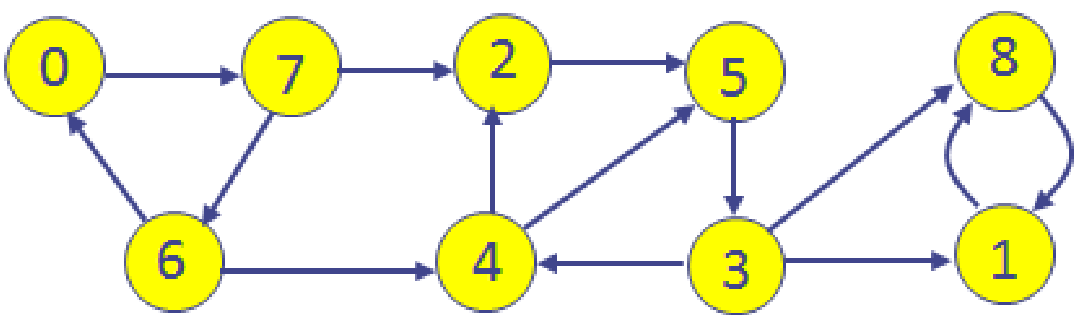
- **단절정점(Articulation Point 또는 Cut Point)**
  - 연결성분의 정점들 중 하나의 정점을 삭제했을 때, 두 개 이상의 연결성분들로 분리될 때 삭제된 정점
- **다리간선(Bridge)**
  - 간선을 제거했을 때 두 개 이상의 연결성분들로 분리될 때 삭제된 간선



- 정점 3과 5는 각각 단절정점
- 간선 (3, 5)는 다리간선
- 위 그래프는 3 개의 이중연결성분, [0, 1, 2, 3], [3, 5], [4, 5, 6]으로 구성
- 단절정점은 이웃한 이중연결성분들에 동시에 속하고, 다리간선은 그 자체로 하나의 이중연결성분

9.3.3 강연결성분(Strongly Connected Component)

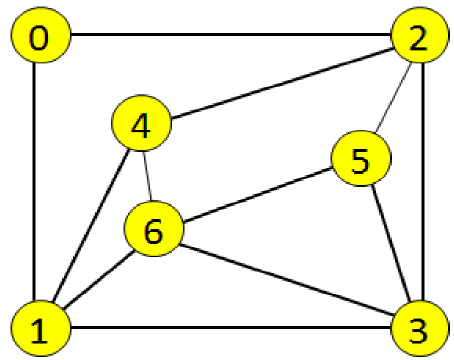
- 방향그래프에서 연결성분 내의 임의의 두 정점 u와 v에 대해 정점 u에서 v로가는 경로가 존재하고 동시에 v에서 u로 돌아오는 경로가 존재하는 연결성분
- 강연결성분은 단절정점이나 다리간선을 포함하지 않는다.
- 강연결성분은 소셜네트워크에서 커뮤니티(Community)를 분석하는데 활용되며, 인터넷의 웹 페이지 분석에도 사용된다.

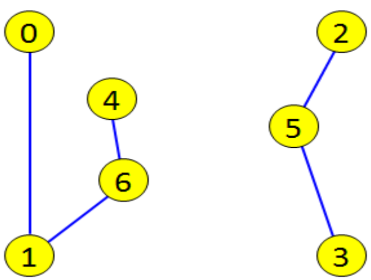


9.4 최소신장트리(Minimum Spanning Tree, MST)

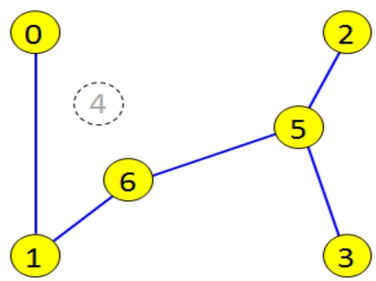
- **최소신장트리는 하나의 연결성분으로 이루어진 무방향 가중치그래프에서 간선의 가중치의 합이 최소인 신장트리**
- **MST를 찾는 대표적인 알고리즘은 Kruskal, Prim, Sollin 알고리즘 - 모두 그리디 (Greedy) 알고리즘**
- **그리디 알고리즘**
  - 최적해(최솟값 또는 최댓값)를 찾는 문제를 해결하기 위한 알고리즘 방식들 중 하나
  - 알고리즘의 선택이 항상 '욕심내어' 지역적인 최솟값(또는 최댓값)을 선택하며, 이러한 부분적인 선택을 축적하여 최적해를 찾음

어느 그래프가 신장트리일까?

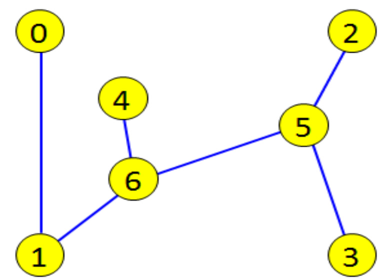




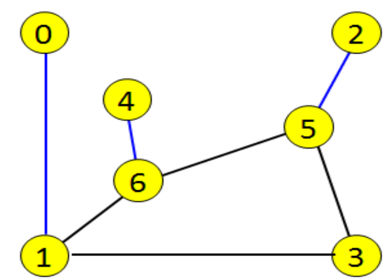
(b)



(c)



(d)



(e)

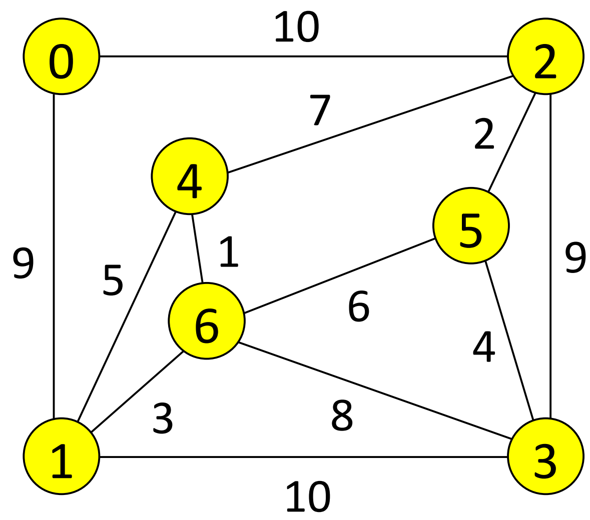
9.4.1 Kruskal 알고리즘

- **간선들을 가중치가 감소하지 않는 순서로 정렬한 후 가장 가중치가 작은 간선을 트리에 추가하되 싸이클을 만들지 않으면 트리 간선으로 선택하고, 싸이클을 만들면 버리는 일을 반복하여 N-1개의 간선이 선택되었을 때 알고리즘을 종료**
  - N은 그래프 정점의 수
- **Kruskal 알고리즘이 그리디 알고리즘인 이유**
  - 남아있는 (정렬된) 간선들 중에서 항상 '욕심 내어' 가중치가 가장 작은 간선을 가져오기 때문

Kruskal 알고리즘

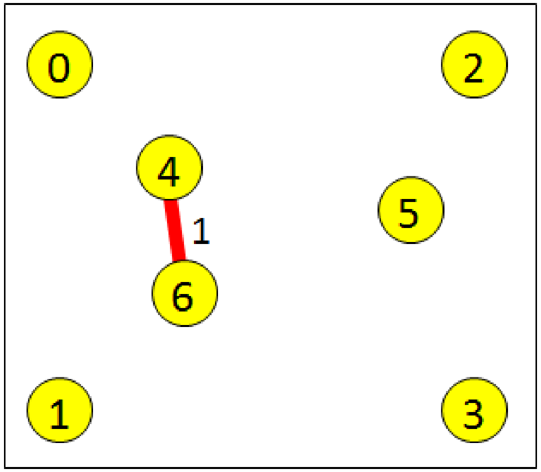
```
[1] 가중치가 감소하지 않는 순서로 간선 리스트 L을 만든다.
[2] while (트리의 간선 수 < N-1)
[3]     L에서 가장 작은 가중치를 가진 간선 e를 가져오고, e
        를 L에서 제거
[4]     if (간선 e가 T에 추가하여 싸이클을 만들지 않으면)
[5]         간선 e를 T에 추가
```

[예제]

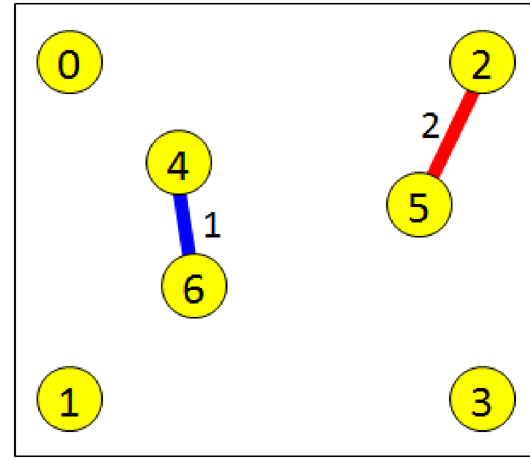


		정렬된 L	
(0, 1)	9	(4, 6)	1
(0, 2)	10	(2, 5)	2
(1, 3)	10	(1, 6)	3
(1, 4)	5	(3, 5)	4
(1, 6)	3	(1, 4)	5
(2, 3)	9	(5, 6)	6
(2, 4)	7	(2, 4)	7
(2, 5)	2	(3, 6)	8
(3, 5)	4	(0, 1)	9
(3, 6)	8	(2, 3)	9
(4, 6)	1	(0, 2)	10
(5, 6)	6	(1, 3)	10

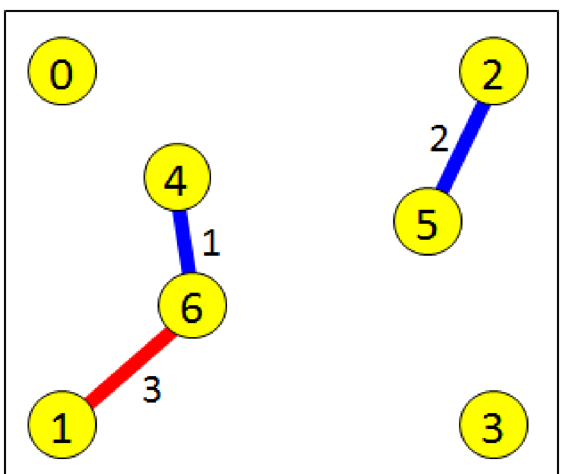
(4, 6)	1
(2, 5)	2
(1, 6)	3
(3, 5)	4
(1, 4)	5
(5, 6)	6
(2, 4)	7
(3, 6)	8
(0, 1)	9
(2, 3)	9
(0, 2)	10
(1, 3)	10



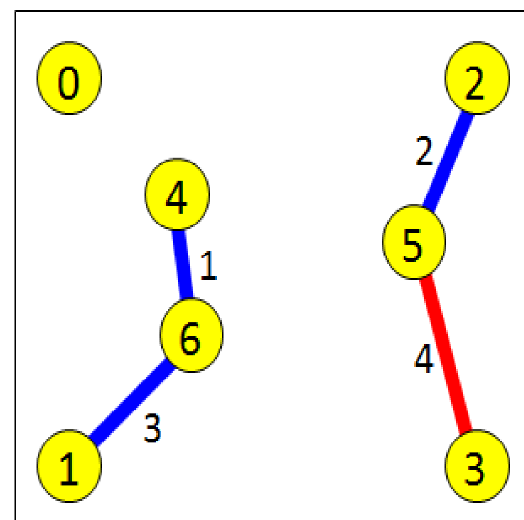
(2, 5)	2
(1, 6)	3
(3, 5)	4
(1, 4)	5
(5, 6)	6
(2, 4)	7
(3, 6)	8
(0, 1)	9
(2, 3)	9
(0, 2)	10
(1, 3)	10



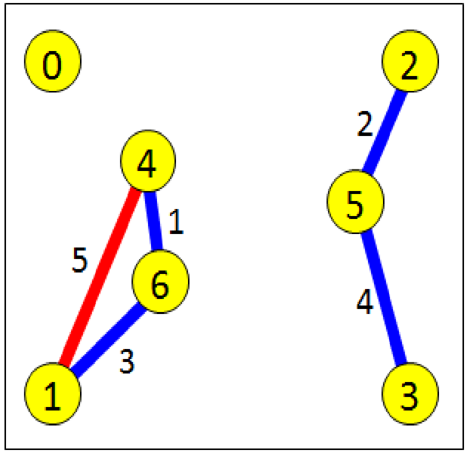
(1, 6)	3
(3, 5)	4
(1, 4)	5
(5, 6)	6
(2, 4)	7
(3, 6)	8
(0, 1)	9
(2, 3)	9
(0, 2)	10
(1, 3)	10



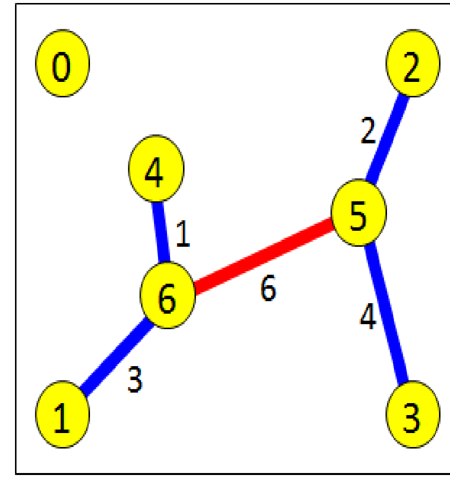
(3, 5)	4
(1, 4)	5
(5, 6)	6
(2, 4)	7
(3, 6)	8
(0, 1)	9
(2, 3)	9
(0, 2)	10
(1, 3)	10



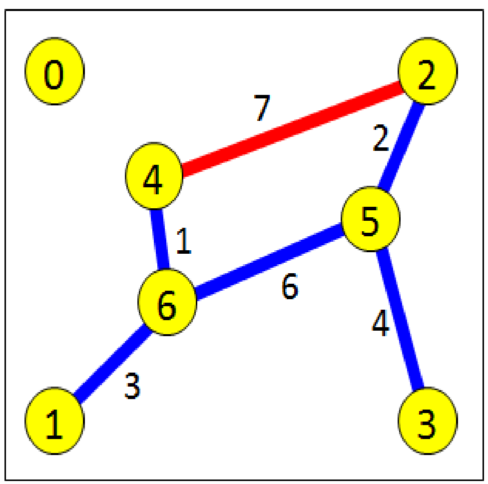
(1, 4) 5
(5, 6) 6
(2, 4) 7
(3, 6) 8
(0, 1) 9
(2, 3) 9
(0, 2) 10
(1, 3) 10



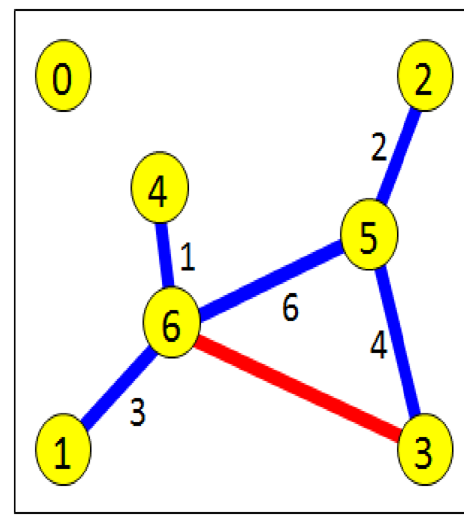
(5, 6) 6
(2, 4) 7
(3, 6) 8
(0, 1) 9
(2, 3) 9
(0, 2) 10
(1, 3) 10

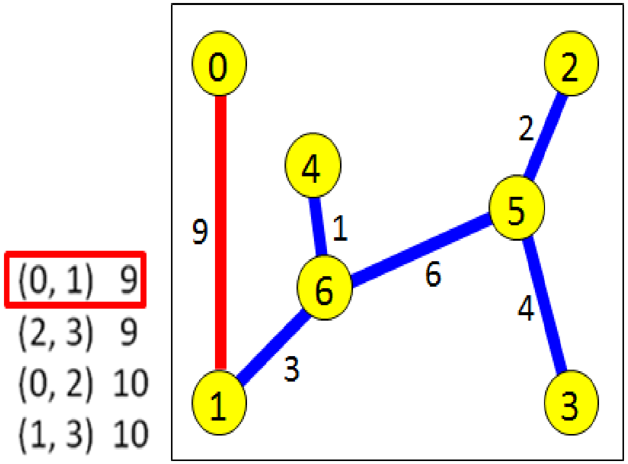


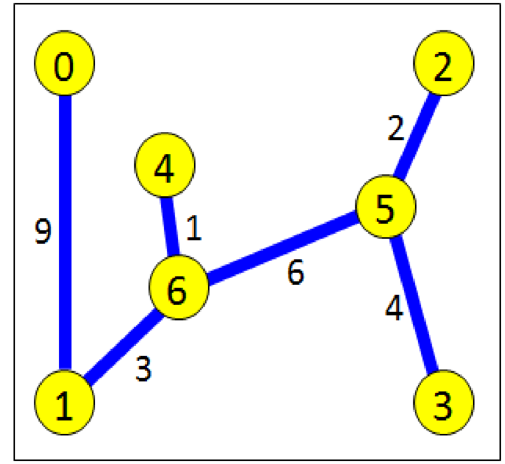
(2, 4) 7
(3, 6) 8
(0, 1) 9
(2, 3) 9
(0, 2) 10
(1, 3) 10



(3, 6) 8
(0, 1) 9
(2, 3) 9
(0, 2) 10
(1, 3) 10







최소신장트리

최소신장트리의 간선의 가중치의 합 = 1 + 2 + 3 + 4 + 6 + 9 = 25

KruskalMST클래스

```java
01 import java.util.*;
02 public class KruskalMST {
03     int N, M;       // 그래프 정점, 간선의 수
04     List<Edge>[] graph;
05     UnionFind uf; // Union-Find 연산을 사용하기 위해
06     Edge[] tree;
07     static class Weight_Comparison implements Comparator<Edge> { //weight를 기준으로 우선순위큐를  사용하기 위해
08         public int compare(Edge e, Edge f) {
09             if(e.weight > f.weight)
10                 return 1;
11             else if(e.weight < f.weight)
12                 return -1;
13             return 0;
14         }
15     }
16     public KruskalMST(List<Edge>[] adjList, int numOfEdges) {
17         N = adjList.length;
18         M = numOfEdges;
19         graph = adjList;
20         uf = new UnionFind(N);   // Union-Find 연산을 사용하기 위해
21         tree = new Edge[N-1];
22     }
23     public Edge[] mst() {  // Kruskal 알고리즘
24         Weight_Comparison BY_WEIGHT = new Weight_Comparison();  // 우선순위큐를 weight 기준으로 구성하기 위해
25         PriorityQueue<Edge> pq = new PriorityQueue<Edge>(M, BY_WEIGHT);  // 자바 라이브러리의 우선순위큐 사용
26                                  // 우선순위큐의 크기로 M(간선의 수)을 지정, BY_WEIGHT는 line 24의 comparator
27         for (int i = 0; i < N; i++){
28             for (Edge e : graph[i]){
29                 pq.add(e);  // edgeArray의 간선 객체들을 pq에 삽입
30             }
31         }
32         int count = 0;
33         while (!pq.isEmpty() && count < N-1) {
34             Edge e = pq.poll();          // 최소 가중치를 가진 간선를 pq에서 제거하고 가져옴
35             int u = e.vertex;            // 가져온 간선의 한 쪽 정점
36             int v = e.adjvertex;         // 가져온 간선의 다른 한 쪽 정점
37             if (!uf.isConnected(u, v)) { // v와 w가 각각 다른 집합에 속해 있으면
38                 uf.union(u, v);          // v가 속한 집합과 u가 속한 집합의 합집합 수행
39                 tree[count++] = e;       // e를 MST의 간선으로서 tree에 추가
40             }
41         }
42         return tree;
43     }
44 }
```

```java
public class Edge {
    int vertex, adjvertex; // 간선의 양끝 정점들
    int weight;            // 간선의 가중치
    public Edge(int u, int v, int wt) {
        vertex    = u;
        adjvertex = v;
        weight    = wt;
    }
}
```

```java
public boolean isConnected(int i, int j) {
    return find(i) == find(j);
}
```

- 간선들을 가중치로 정렬하는 대신에 line 07에서 Weight_Comparison 클래스를 선언하여 mst() 메소드 내의 line 24에서 BY_WEIGHT라는 객체를 만들어 간선의 가중치를 기준으로 간선을 비교하여 line 27~31에서 우선순위큐(최소힙)에 간선들을 저장
- 우선순위큐는 line 25에서 자바의 PriorityQueue를 사용하며 (M, BY_WEIGHT)에서 M은 우선순위큐의 크기(size)이고, BY_WEIGHT는 우선순위 비교기준을 의미
- Kruskal 알고리즘에서 추가하려는 간선이 싸이클을 만드는 간선인지 검사하기 위해, Union-Find 클래스를 활용
- 이를 위해 line 20에서 UnionFind 객체를 생성,

- Line 37에서 UnionFind 클래스에 아래와 같이 선언된 isConnected() 메소드를 이용하여 간선의 양쪽 끝 정점들이 동일한 집합에 속해 있는지 검사

```java
public boolean isConnected(int i, int j) {
    return find(i) == find(j);
}
```

- 만약 양쪽 끝 정점이 다른 집합에 속하면, line 38에서 두 집합에 대해 union() 메소드를 호출하여 합집합을 수행하고, line 39에서 간선을 트리에 추가
- 만약 양쪽 끝 정점이 동일한 집합에 속할 경우, 추가하려는 간선은 무시되고,
- 다음의 루프 수행을 위해 line 33의 while-루프의 조건을 검사

```java
public class UnionFind {
    protected  int[] p;     // 배열 크기는 정점의 수 N이고 p[i]는 i의 부모 원소를 저장한다.
    protected  int[] rank;

    public UnionFind(int N) {
        p = new int[N];
        rank = new int[N];
        for (int i = 0; i < N; i++) {
            p[i]    = i;  // 초기엔 N개의 트리가 각각 i 자기 자신이 부모이기 때문에
            rank[i] = 0;  // 초기엔 N개의 트리 각각의 rank를 0으로 초기화
        }
    }
    //i가 속한 집합의 루트 노드를 재귀적으로 찾고 최종적으로 경로상의 각 원소의 부모를 루트 노드로 만든다.
    protected int find(int i) { // 경로 압축
        if ( i != p[i])
            p[i] = find(p[i]);  //리턴하며 경로상의 각 노드의 부모가 루트가되도록 만든다.
        return p[i];
    }
    //i와 j가 같은 트리에 있는지를 검사한다.
    public boolean isConnected(int i, int j) {
        return find(i) == find(j);
    }
    public void union(int i, int j) {  // Union 연산
        int iroot = find(i);
        int jroot = find(j);
        if (iroot == jroot) return;   // 루트 노드가 동일하면 더이상의 수행없이 그대로 리턴
        // rank가 높은 루트 노드가 승자로 union을 수행한다.
        if (rank[iroot] > rank[jroot])
            p[jroot] = iroot;                 // iroot가 승자
        else if (rank[iroot] < rank[jroot])
            p[iroot] = jroot;                 // jroot가 승자
        else {
            p[jroot] = iroot;  // 둘중에 하나 임의로 승자
            rank[iroot]++;     // iroot의 rank 1 증가
        }
    }
}
```

```java
import java.util.*;
public class main {
    public static void main(String[] args) {
        int[][] weight = {  // [그림 9-4-2](a)의 그래프
                    {  0,  9, 10,  0, 0, 0, 0},
                    {  9,  0,  0 ,10, 5, 0, 3},
                    { 10,  0,  0,  9, 7, 2, 0},
                    {  0, 10,  9,  0, 0, 4, 8},
                    {  0,  5,  7,  0, 0, 0, 1},
                    {  0,  0,  2,  4, 0, 0, 6},
                    {  0,  3,  0,  8, 1, 6, 0},
        };
        int N = weight.length;
        int M = 0;  // 그래프 간선의 수
        List<Edge>[] adjList = new List[N];
        for (int i = 0; i < N; i++) {
            adjList[i] = new LinkedList<>();
            for (int j = 0; j < N; j++) {
                if (weight[i][j] != 0) {
                    Edge e = new Edge(i,j, weight[i][j]);
                    adjList[i].add(e);
                    M++;
                }
            }
        }

        KruskalMST k = new KruskalMST(adjList, M);  // KruskalMST 객체 생성
        Edge[] tree = new Edge[N-1];                 // 최소신장트리의 간선을 출력하기 위해

        System.out.print("최소신장트리 간선: ");
        tree = k.mst();    // mst() 메소드 호출

        int sum = 0;
        for (int i = 0; i < tree.length; i++) {
            System.out.print("("+tree[i].vertex + "," +tree[i].adjvertex+ ") ");
            sum += tree[i].weight;
        }
        System.out.printf("\n\n");
        System.out.println("최소신장트리의 간선 가중치 합 = "+sum);
    }
}
```

프로그램 수행 결과

```
Console
<terminated> main (44) [Java Application] C:₩Program Files₩Java₩jdk1.8.0_40₩bin₩javaw.exe
최소신장트리 간선: (4,6) (5,2) (1,6) (5,3) (5,6) (0,1)

최소신장트리의 간선 가중치 합 = 25
```

수행시간

- 간선을 정렬(또는 우선순위큐의 삽입과 삭제)하는데 소요되는 시간 O(MlogM) = O(MlogN)
- 신장트리가 만들어질 때까지 간선에 대해 isConnected()와 union()을 수행하는 시간 O((M+N)log*N)
- O(MlogN) + O((M+N)log*N) = O(MlogN)
- 4.4절의 상호배타적 집합을 위한 트리 연산의 [수행시간] 참조

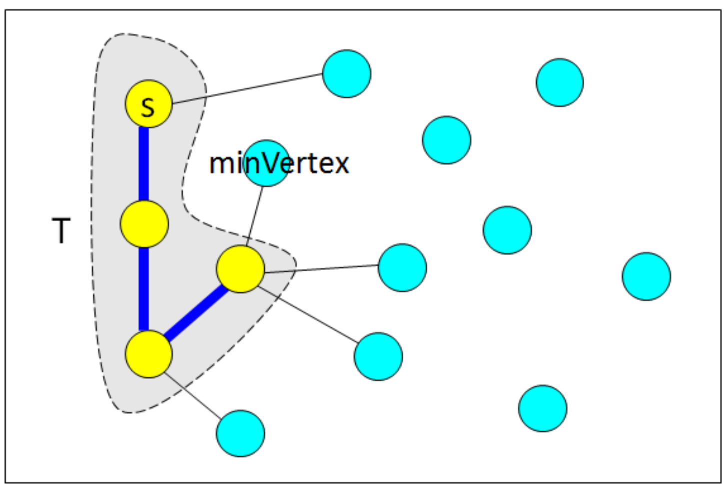
9.4.2 Prim알고리즘

- Prim 알고리즘은 임의의 시작 정점에서 가장 가까운 정점을 추가하여 간선이 하나의 트리를 만들고,
  만들어진 트리에 인접한 가장 가까운 정점을 하나씩 추가하여 최소신장트리를 만든다.
- Prim의 알고리즘에서는 초기에 트리 T는 임의의 정점 s만을 가지며, 트리에 속하지 않은 각 정점과 T의 정점(들)에 인접한 간선들 중에서 가장 작은 가중치를 가진 간선의 끝점을 찾기 위해 배열 D를 사용

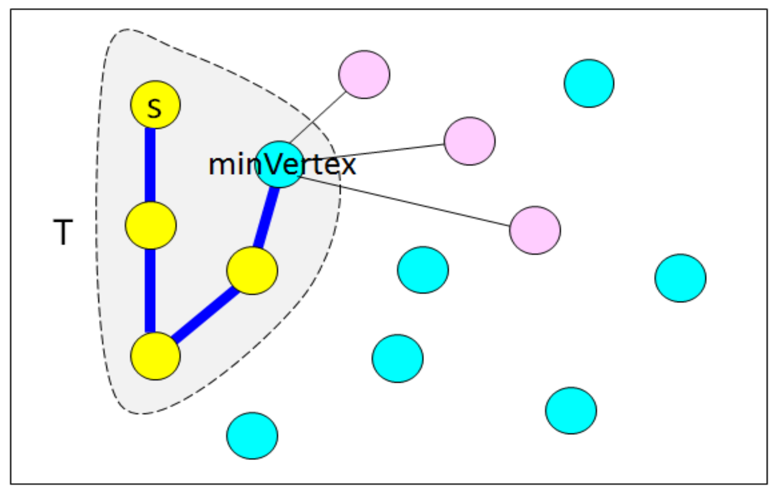
Prim 알고리즘

```
[1] 배열D를 ∞로 초기화한다. 시작정점 s의 D[s] = 0

[2] while (T의 정점 수 < N)

[3]     T에 속하지 않은 각 정점 i에 대해 D[i]가 최소인
          정점 minVertex를 찾아 T에 추가

[4]     for (T에 속하지 않은 각 정점 w에 대해서)
[5]         if (간선 (minVertex, w)의 가중치 < D[w])
[6]             D[w] = 간선 (minVertex, w)의 가중치
```

- **Prim 알고리즘의 step [3] ~ [6]**
  - (a) 트리에 가장 가까운 정점 minVertex를 찾아(트리 밖에 있는 정점들의 배열 D의 원소들 중에서 최솟값을 찾아)
  - (b) 트리에 추가한 후, 정점 minVertex에 인접하면서 트리에 속하지 않은 각 정점의 D 원소가 이전 값보다 작으면 갱신

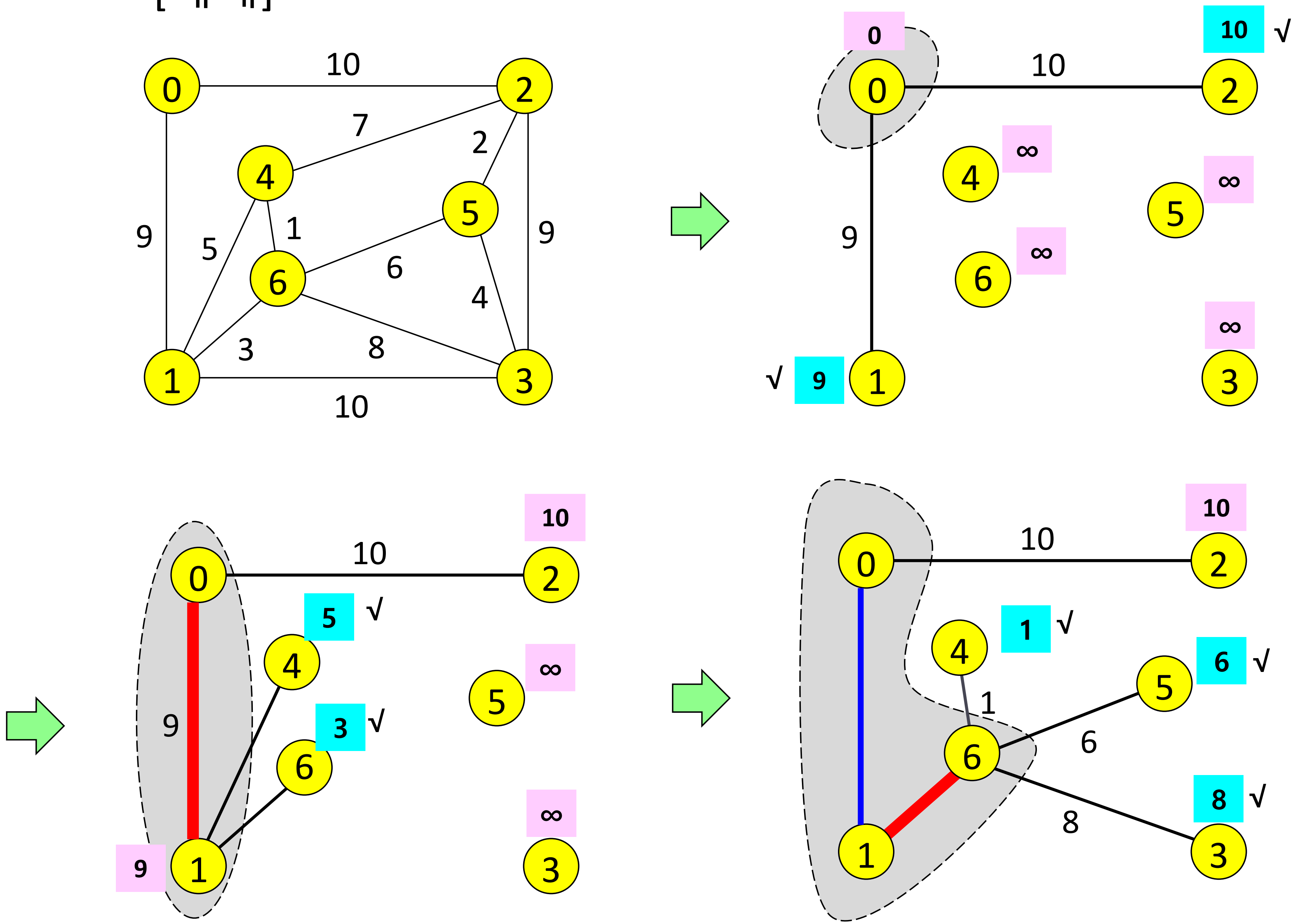


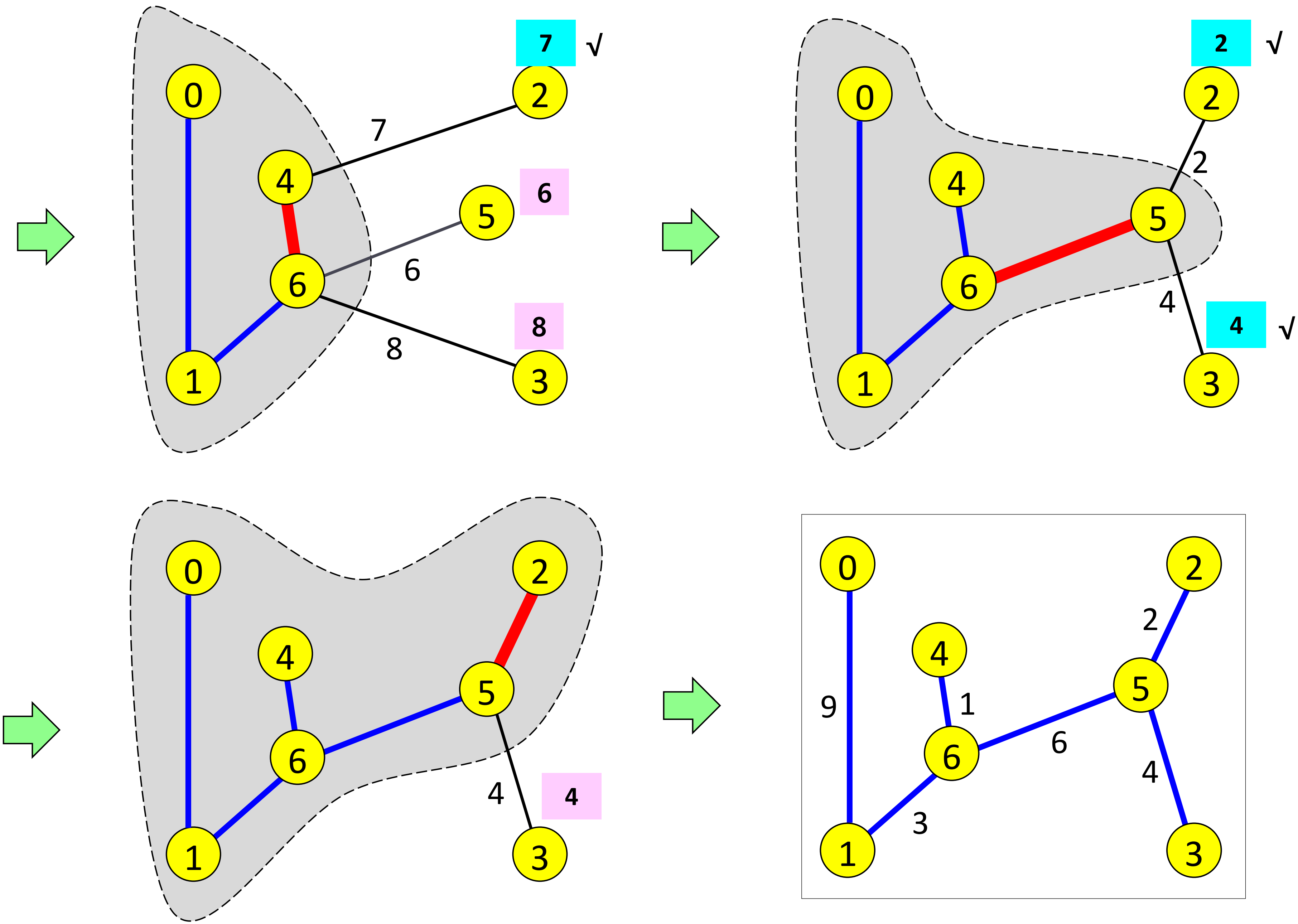
(a)



(b)

[예제]




0
4
6
1
2
5
3
7 √
6
8
7
6
8
0
4
6
1
5
2
3
2 √
4 √
2
4
0
4
6
1
2
5
3
4
4
0
4
6
1
5
2
3
9
1
3
6
2
4

```java
import java.util.List;
public class PrimMST {
    int N; // 그래프 정점의 수
    List<Edge>[] graph;

    public PrimMST(List<Edge>[] adjList) { // 생성자
        N = adjList.length;
        graph = adjList;
    }

    public int[] mst (int s) { // Prim 알고리즘,s는 시작정점
        boolean[] visited = new boolean[N]; // 방문된 정점은 true로
        int[] D = new int[N];
        int[] previous = new int[N]; // 최소신장트리의 간선으로 확정될 때 간선의 다른 쪽 (트리의)끝점
        for(int i = 0; i < N; i++){  // 초기화
            visited[i] = false;
            previous[i] = -1;
            D[i] = Integer.MAX_VALUE;  // D[i]를 최댓값으로 초기화
        }
        previous[s] = 0;  //시작정점 s의 관련 정보 초기화
        D[s] = 0;

        for(int k = 0; k < N; k++){ // 방문안된 정점들의 D 원소들중 에서 최솟값가진 정점 minVertex 찾기
            int minVertex = -1;
            int min = Integer.MAX_VALUE;
            for(int j=0;j<N;j++){
                if ((!visited[j])&&(D[j] < min)){
                    min = D[j];
                    minVertex = j;
                }
            }
            visited[minVertex] = true;
            for (Edge i : graph[minVertex]){ // minVertex에 인접한 각 정점의 D의 원소 갱신
                if (!visited[i.adjvertex]){  // 트리에 아직 포함 안된 정점이면
                    int currentDist = D[i.adjvertex];
                    int newDist = i.weight;
                    if (newDist < currentDist){
                        D[i.adjvertex] = newDist; // minVertex와 연결된 정점들의 D 원소 갱신
                        previous[i.adjvertex] = minVertex; // 트리 간선 추출을 위해
                    }
                }
            }
        }
        return previous; // 최소신장트리 간선 정보 리턴
    }
}
```

```java
public class Edge {
    int adjvertex; // 간선의 다른쪽 끝 정점
    int weight;    // 간선의 가중치
    public Edge(int v, int wt) {
        adjvertex = v;
        weight    = wt;
    }
}
```

- Line 14: 배열 previous를 선언하여 최소신장트리의 간선을 저장
- 즉, previous[i] = j라면 간선 (i, j)가 트리의 간선
- Line 12~21: 배열 선언 및 초기화
- Line 23의 for-루프: N개의 정점을 트리에 추가한 뒤 종료
- Line 23~31: 트리에서 가장 가까운 정점 minVertex를 찾고
- Line 33~43: minVertex에 인접하면서 트리에 속하지 않은 정점의 D 원소 갱신
- Line 38~39: D 원소를 갱신하고 minVertex를 previous 배열의 해당 원소에 저장
- 마지막으로 배열 previous를 line 44에서 리턴

프로그램 수행 결과

```
Console
<terminated> main (67) [Java Application] C:₩Program Files₩Java₩jdk1.8.0_40₩bin₩javaw.exe
최소신장트리 간선 :
(1,0) (2,5) (3,5) (4,6) (5,6) (6,1)

최소신장트리의 간선 가중치 합 =  25
```

수행시간(1)

- Prim 알고리즘은 N번의 반복을 통해 minVertex를 찾고 minVertex에 인접하면서 트리에 속하지 않은 정점에 해당하는 D 의 원소 값을 갱신
- PrimMST 클래스에서는 minVertex를 배열 D에서 탐색하는 과정에서 O(N) 시간이 소요되고, minVertex에 인접한 정점들을 검사하여 D의 해당 원소를 갱신하므로 O(N) 시간이 소요. 따라서 총 수행시간은 Nx(O(N) +O(N)) = $O(N^2)$
- minVertex 찾기 위해 이진힙(Binary Heap)을 사용하면 각 간선에 대한 D의 원소를 갱신하며 힙 연산을 수행해야 하므로 총 O(MlogN) 시간이 필요

수행시간(2)

- **M은 그래프 간선의 수, 이진힙은 각 정점에 대응되는 D원소를 저장하므로 힙의 최대 크기는 N**
- **또한 가중치가 갱신되어 감소되었을 때의 힙 연산(decrease_key)에는 O(logN) 시간이 소요**
- **입력그래프가 희소그래프라면(예를 들어 M = O(N)이라면) 수행시간이 O(MlogN) = O(NlogN)이 되어 이진힙을 사용하는 것이 매우 효율적**
  - minVertex 찾기에 피보나치힙(Fibonacci Heap) 자료구조를 사용하면 O(NlogN + M) 시간에 Prim 알고리즘 수행
  - 피보나치힙은 복잡하고 구현도 쉽지 않아서 이론적인 자료구조임

9.4.3 Sollin 알고리즘

- **Sollin 알고리즘은 각 정점을 독립적인 트리로 간주하고, 각 트리에 연결된 간선들 중에서 가장 작은 가중치를 가진 간선을 선택.**
  이때 선택된 간선은 2 개의 트리를 1개의 트리로 만든다.
- **같은 방법으로 한 개의 트리가 남을 때까지 각 트리에서 최소 가중치 간선을 선택하여 연결**
- **Sollin 알고리즘은 병렬알고리즘(Parallel Algorithm)으로 구현이 쉽다는 장점을 가짐**

Sollin 알고리즘

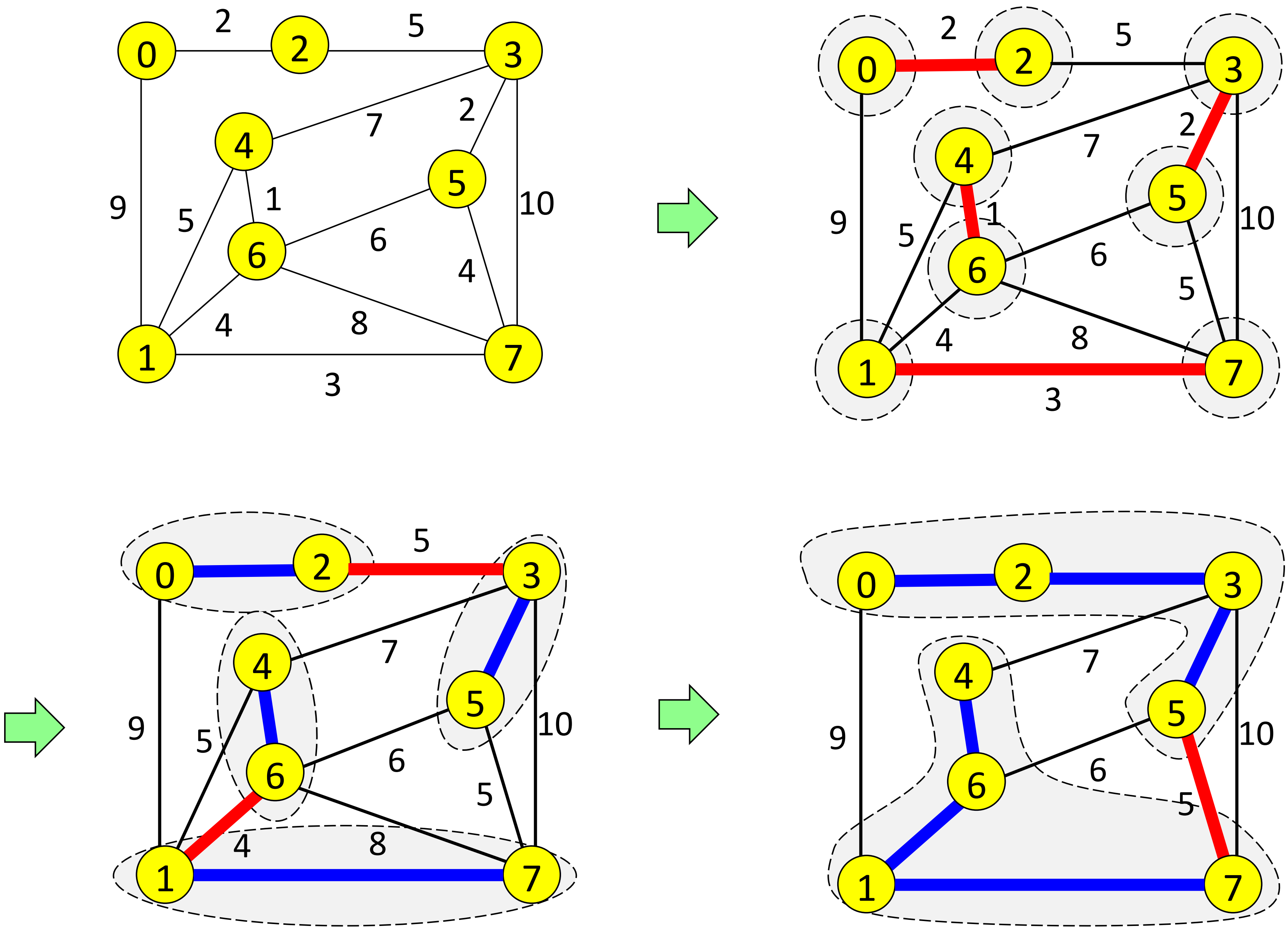
[1] 각 정점은 독립적인 트리이다.

[2] repeat

[3] 각 트리에 닿아 있는 간선들 중에서 가중치가 가장 작은 간선을 선택하여 트리를 합친다.

[4] until (1개의 트리만 남을 때까지)

[예제]



수행시간

- Sollin 알고리즘에서 repeat-루프가 예제와 같이 각 쌍의 트리가 서로 연결된 간선을 선택하는 경우 최대 logN번 수행
- 루프 내에서는 각 트리가 자신에 닿아 있는 모든 간선들을 검사하여 최소 가중치를 가진 간선을 선택하므로 O(M) 시간이 소요
- 따라서 알고리즘의 수행시간은 O(MlogN)

9.5 최단경로 알고리즘

- Dijkstra 알고리즘
- Bellman-Ford
- Floyd-Warshall 알고리즘

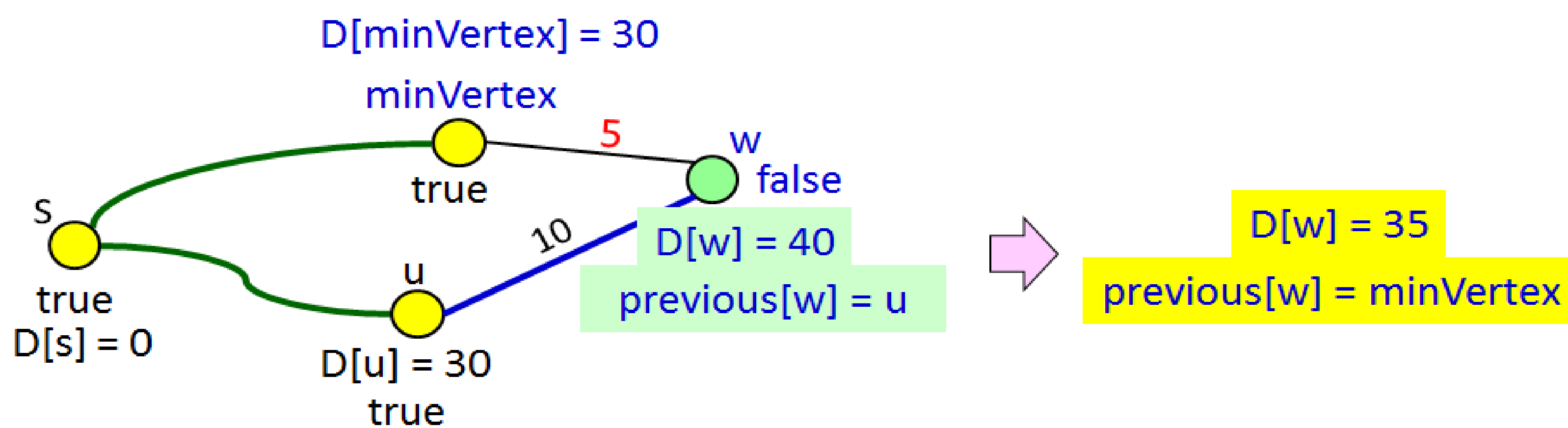
9.5.1 Dijkstra 알고리즘

- **최단경로(Shortest Path) 찾기는 주어진 가중치그래프에서 출발점으로부터 도착점까지의 최단경로를 찾는 문제**
- **Dijkstra 알고리즘: 출발점으로부터 각 정점까지의 최단거리 및 경로를 계산**
  - Dijkstra 알고리즘은 Prim의 MST 알고리즘과 매우 유사
- **차이점**
  - Dijkstra 알고리즘은 출발점이 주어지지지만 Prim알고리즘에서는 출발점이 주어지지 않는다는 것
  - Prim 알고리즘에서는 배열 D의 원소에 간선의 가중치가 저장되지만, Dijkstra 알고리즘에서는 D의 원소에 출발점으로부터 각 정점까지의 경로의 길이가 저장된다는 것

Dijkstra 알고리즘

```
배열 D를 ∞로 초기화시킨다. 단, D[s]=0이다.
for (k = 0; k < N; k++)
    방문 안된 각 정점i에 대해 D[i]가 최소인 정점 minVertex를 찾고, 방문한다.
    for (minVertex에 인접한 각 정점 w에 대해서)
        if (w 가 방문 안된 정점이면)
            if (D[minVertex] + 간선 (minVertex, w)의 가중치 < D[w])
                D[w] = D[minVertex] + 간선 (minVertex, w)의 가중치    // 간선완화
                previous[w] = minVertex
```

- Step [7]의 간선완화(Edge Relaxation)는 minVertex가 step [3]에서 선택된 후에 s로부터 minVertex를 경유하여 정점 w까지의 경로의 길이가 현재의 D[w]보다 더 짧아지면 짧은 길이로 D[w]를 갱신하는 것을 의미
- 그림은 D[w]가 minVertex 덕분에 40에서 35로 완화된 것을 나타냄

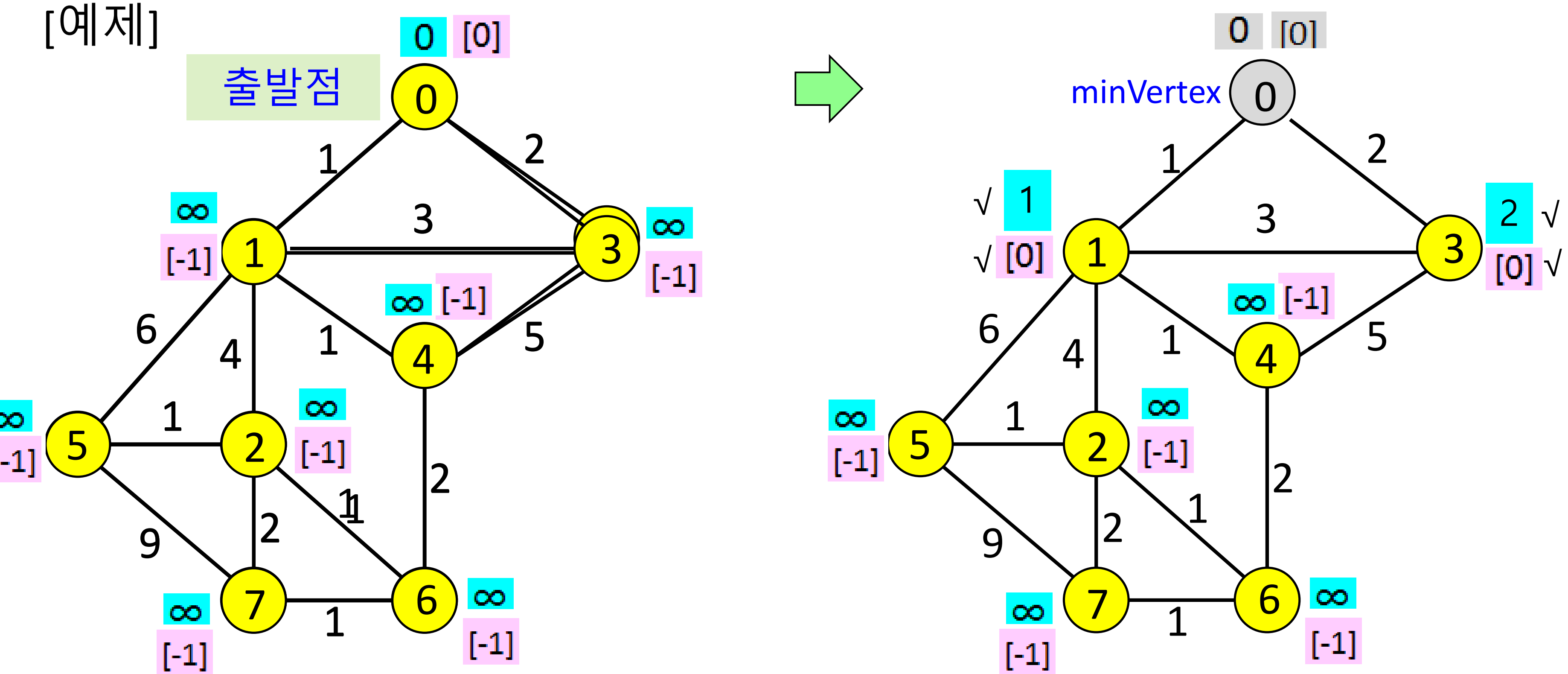


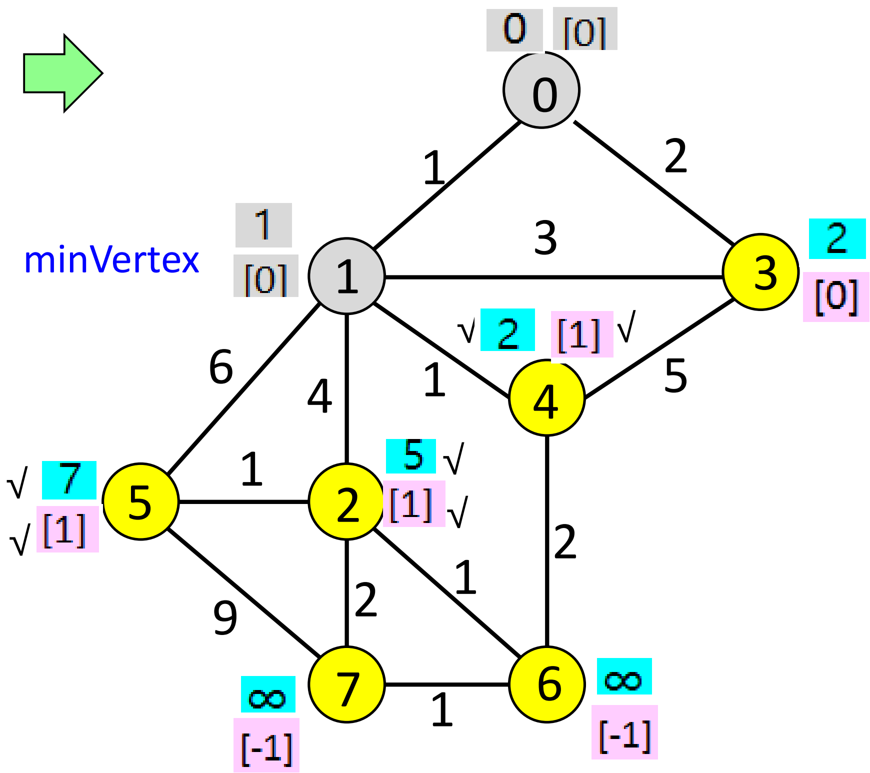
[핵심 아이디어]

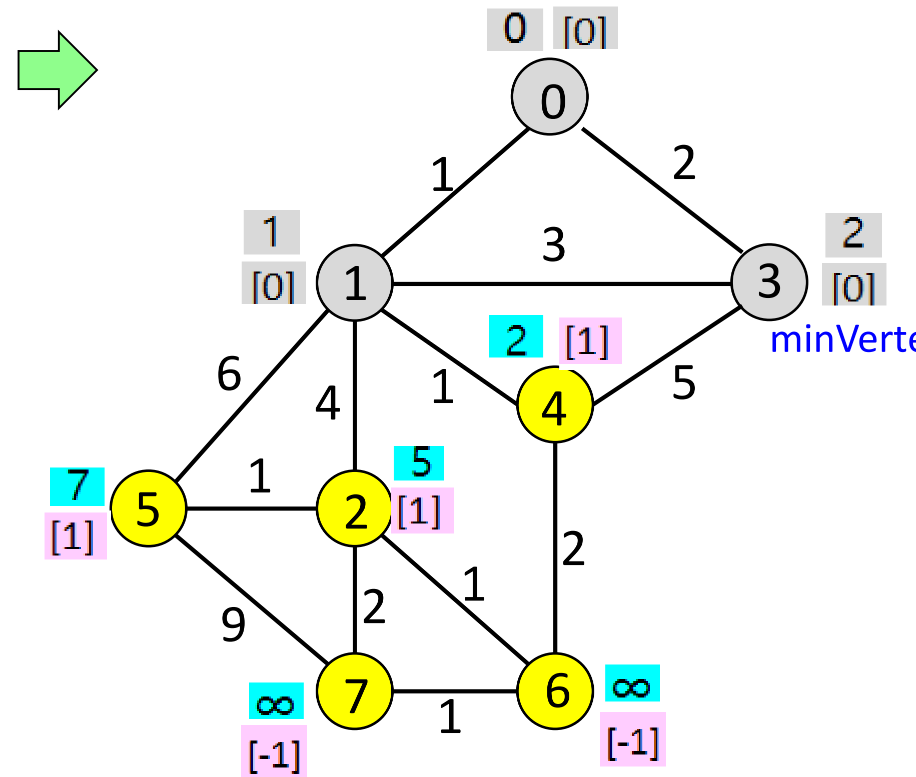
그리디하게 정점을 선택하여 방문하고, 선택한 정점의 방문 안된 인접한 정점들에 대한 간선완화를 수행한다.

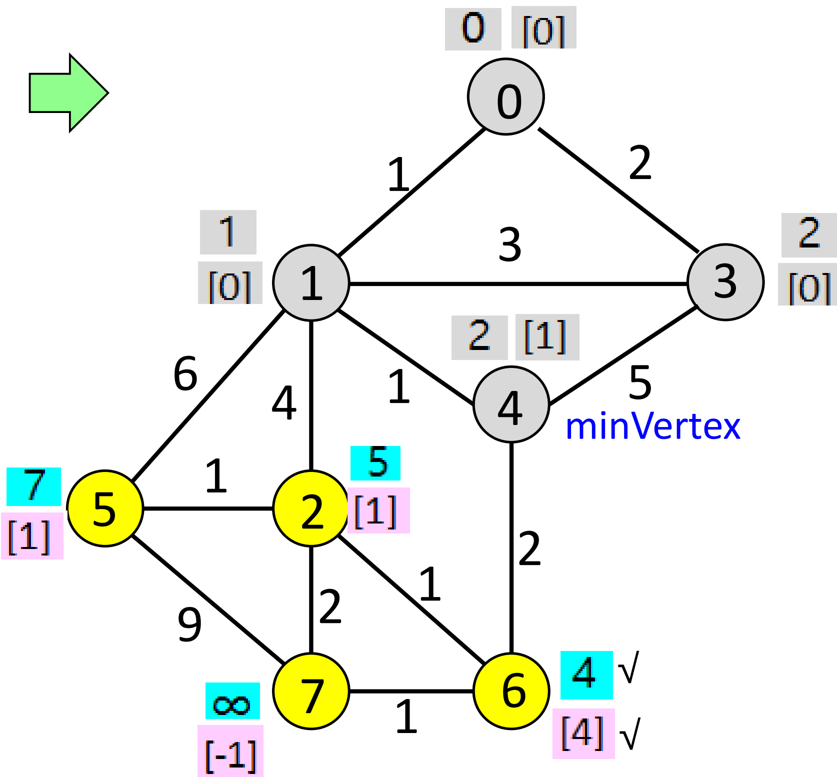
한번 방문된 정점의 D원소 값은 변하지 않는다.

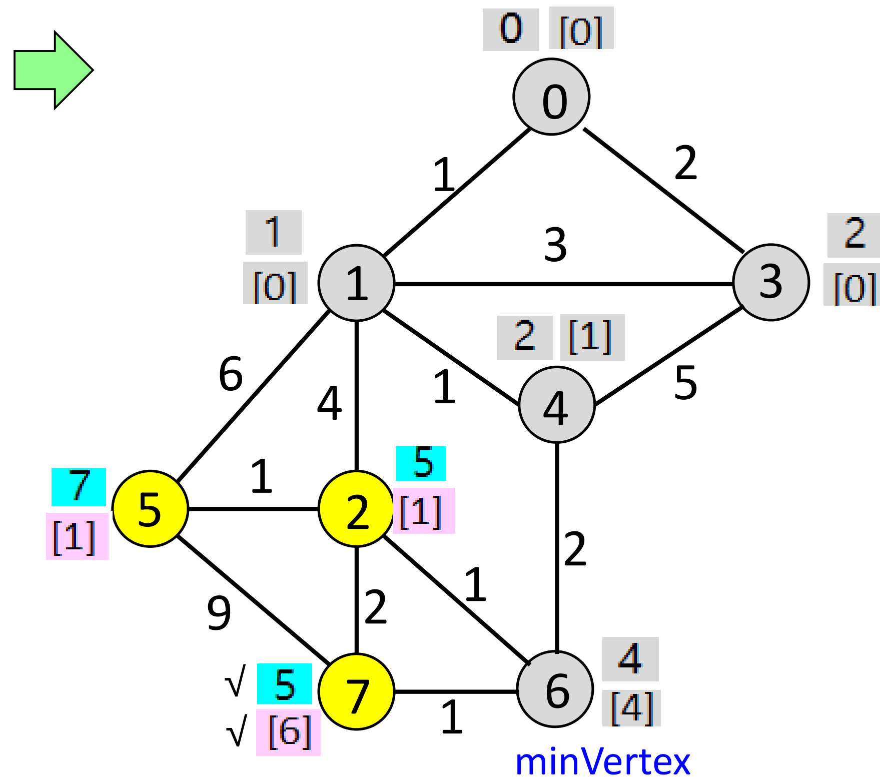
[예제]

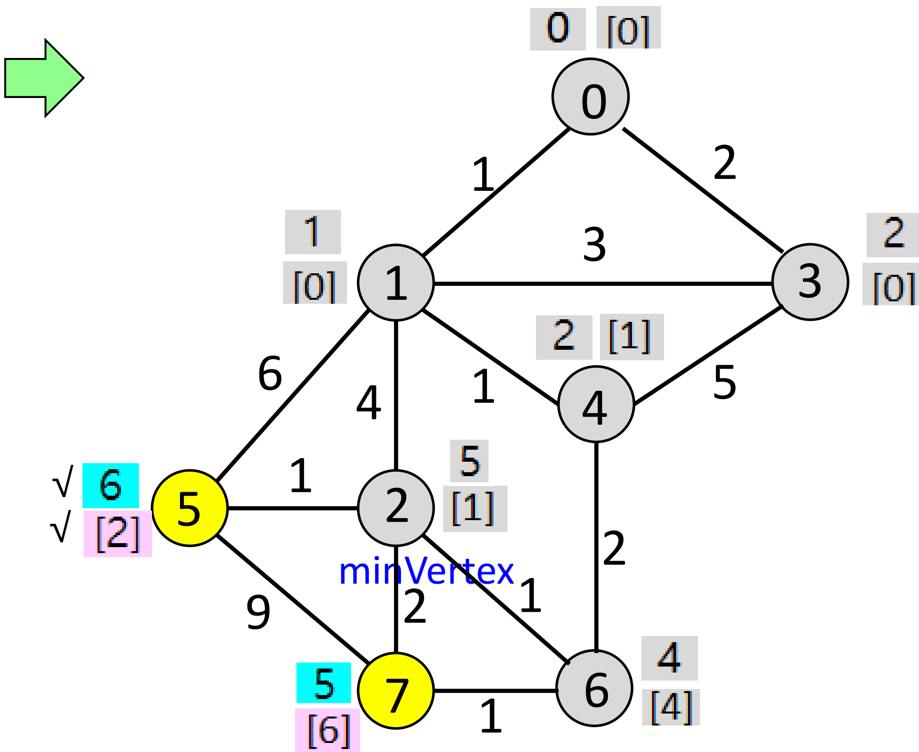


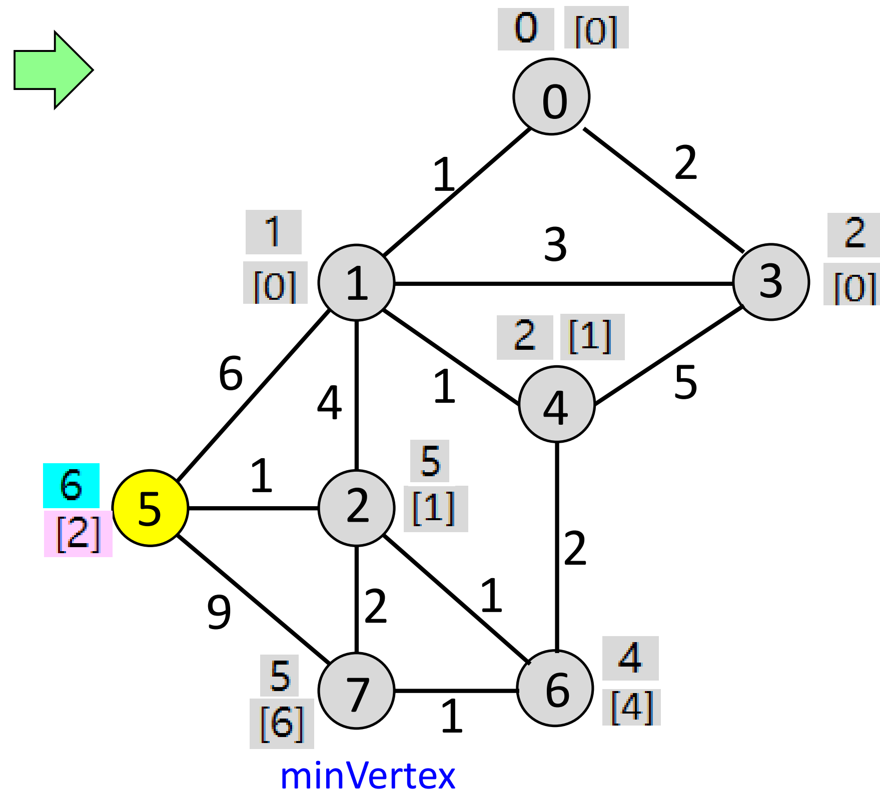

minVertex

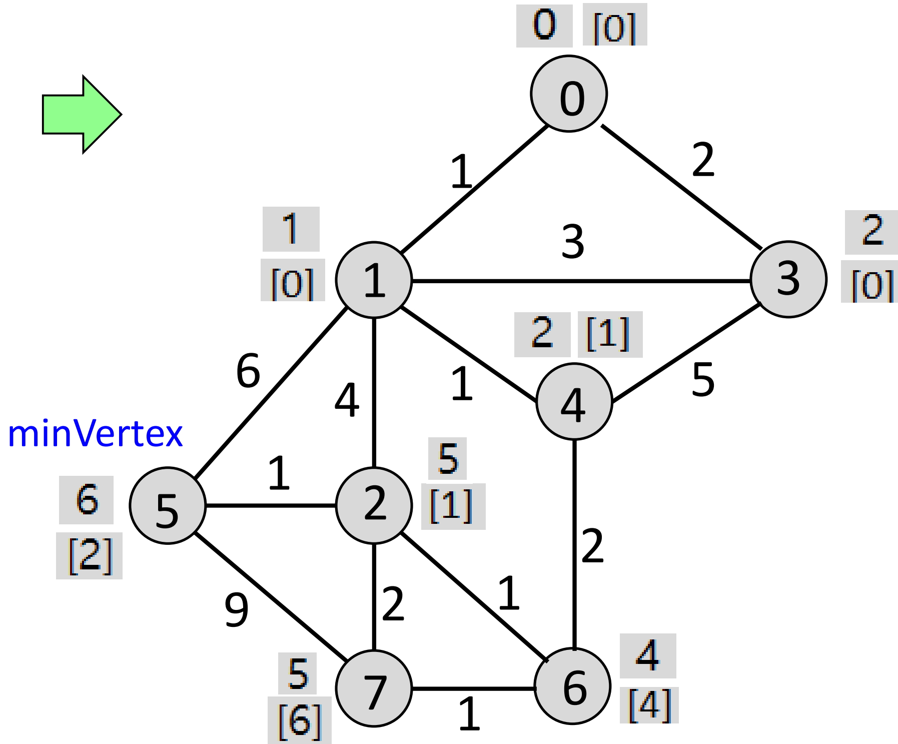

minVerte


0 [0]
0
1
2
1
[0]
1
3
3
2
[0]
2 [1]
6
4
1
4
5
minVertex
7
[1]
5
1
2
5
[1]
2
1
9
2
∞
[-1]
7
1
6
4 √
[4] √


0 [0]
0
1
2
1
[0]
1
3
3
2
[0]
2 [1]
6
4
1
4
5
7
[1]
5
1
2
5
[1]
2
1
9
2
√ 5
√ [6]
7
1
6
4
[4]
minVertex


0
[0]
0
1
2
1
[0]
1
3
3
2
[0]
2
[1]
6
4
1
4
5
√ 6
√ [2]
5
1
2
5
[1]
minVertex
9
2
1
2
5
[6]
7
1
6
4
[4]


0
[0]
0
1
2
1
[0]
1
3
3
2
[0]
2
[1]
6
4
1
4
5
6
[2]
5
1
2
5
[1]
9
2
1
2
5
[6]
7
1
6
4
[4]
minVertex




```
정점 0으로부터의 최단 거리
[0, 1] = 1
[0, 2] = 5
[0, 3] = 2
[0, 4] = 2
[0, 5] = 6
[0, 6] = 4
[0, 7] = 5
```

```
정점 0으로부터의 최단 경로
1<-0
2<-1<-0
3<-0
4<-1<-0
5<-2<-1<-0
6<-4<-1<-0
7<-6<-4<-1<-0
```

Edge 클래스

```java
public class Edge {
    int vertex;        // 간선의 한쪽 끝 정점
    int adjvertex;     // 간선의 다른쪽 끝 정점
    int weight;        // 간선의 가중치

    public Edge(int u, int v, int wt) {
        vertex    = u;
        adjvertex = v;
        weight    = wt;
    }
}
```

```java
import java.util.List;
public class DijkstraSP{
    public int N;           // 그래프 정점의 수
    List<Edge>[] graph;
    public int[] previous;  // 최단경로상 이전 정점을 기록하기 위해
    public DijkstraSP(List<Edge>[] adjList) {
        N = adjList.length;
        previous = new int[N];
        graph = adjList;
    }
    public int[] shortestPath (int s){
        boolean[] visited = new boolean[N];
        int[] D = new int[N];
        for(int i = 0; i < N; i++){   //초기화
            visited[i] = false;
            previous[i] = -1;
            D[i] = Integer.MAX_VALUE;
        }
        previous[s] = 0;  // 시작점 s의 관련 정보 초기화
        D[s]= 0;
        for(int k = 0; k < N; k++){        // 방문 안된 정점들 중에서
            int minVertex = -1;            // D원소 값이 최소인 minVertex 찾기
            int min = Integer.MAX_VALUE;
            for(int j = 0; j < N; j++){
                if ((!visited[j]) && (D[j] < min)){
                    min = D[j];
                    minVertex = j;
                }
            }
            visited[minVertex] = true;
            for (Edge e: graph[minVertex]){  // minVertex에 인접한 각 정점에 대해
                if (!visited[e.adjvertex]){  // 아직  방문 안된 정점에 대해
                    int currentDist = D[e.adjvertex];
                    int newDist = D[minVertex] + e.weight;
                    if (newDist < currentDist){
                        D[e.adjvertex] = newDist;            // 간선완화
                        previous[e.adjvertex] = minVertex;   // 최종 최단경로를 '역 방향으로'추출
                    }
                }
            }
        }
        return D;
    }
}
```

```java
import java.util.*;
public class main {
    public static void main(String[] args) {
        int[][] weight = { // [그림 9-5-2](a)의 입력그래프
                { 0, 1, 0, 2, 0, 0, 0, 0},
                { 1, 0, 4 ,3, 1, 6, 0, 0},
                { 0, 4, 0, 0, 0, 1, 1, 2},
                { 2, 3, 0, 0, 5, 0, 0, 0},
                { 0, 1, 0, 5, 0, 0, 2, 0},
                { 0, 6, 1, 0, 0, 0, 0, 9},
                { 0, 0, 1, 0, 2, 0, 0, 1},
                { 0, 0, 2, 0, 0, 9, 1, 0}
        };
        int N = weight.length;
        List<Edge>[] adjList = new List[N];
        for (int i = 0; i < N; i++) {  // 인접리스트 만들기
            adjList[i] = new LinkedList<>();
            for (int j = 0; j < N; j++) {
                if (weight[i][j] != 0) {
                    Edge e = new Edge(i,j, weight[i][j]);
                    adjList[i].add(e);
                }
            }
        }

        DijkstraSP d = new DijkstraSP(adjList);
```

```java

        System.out.println("정점 0으로부터의 최단거리");
        int[] distance = d.shortestPath(0);

        for (int i = 0; i < distance.length; i++) {
            if (distance[i] == Integer.MAX_VALUE)
                System.out.println("0과 " + i +" 사이에 경로 없음.");
            else
                System.out.println("[0, " + i + "] = " +distance[i]);
        }

        System.out.printf("\n정점 0으로부터의 최단 경로\n");
        for (int i = 1; i < d.N; i++){
            int back = i;
            System.out.print(back);
            while (back!= 0) {
                System.out.print("<-"+d.previous[back]);
                back = d.previous[back];
            }
            System.out.println();
        }
    }
}
```

프로그램 수행 결과

Console

```
<terminated> main (68) [Java Application] C:₩Program Files₩Java₩jdk1.8.0_40₩bin₩javaw.exe
```

```
정점 0으로부터의 최단거리
[0, 0] = 0
[0, 1] = 1
[0, 2] = 5
[0, 3] = 2
[0, 4] = 2
[0, 5] = 6
[0, 6] = 4
[0, 7] = 5
```

```
정점 0으로부터의 최단 경로
1<-0
2<-1<-0
3<-0
4<-1<-0
5<-2<-1<-0
6<-4<-1<-0
7<-6<-4<-1<-0
```

- DijkstraSP 클래스의 line 12~20은 배열 선언 및 초기화
- Line 21: for-루프는 N개의 정점을 방문한 후 종료
- Line 22~29: 출발점에서 아직 방문 안된 정점들 중에서 가장 가까운 정점 minVertex를 찾고,
- Line 31~40: minVertex에 인접하면서 방문 안된 정점에 대해 간선완화 수행
- Line 37: D의 원소가 갱신될 때 minVertex를 previous의 원소에 저장해둔 뒤, 나중에 최단 경로 추출에 이용
- 마지막으로 배열 D를 line 42에서 리턴

수행시간(1)

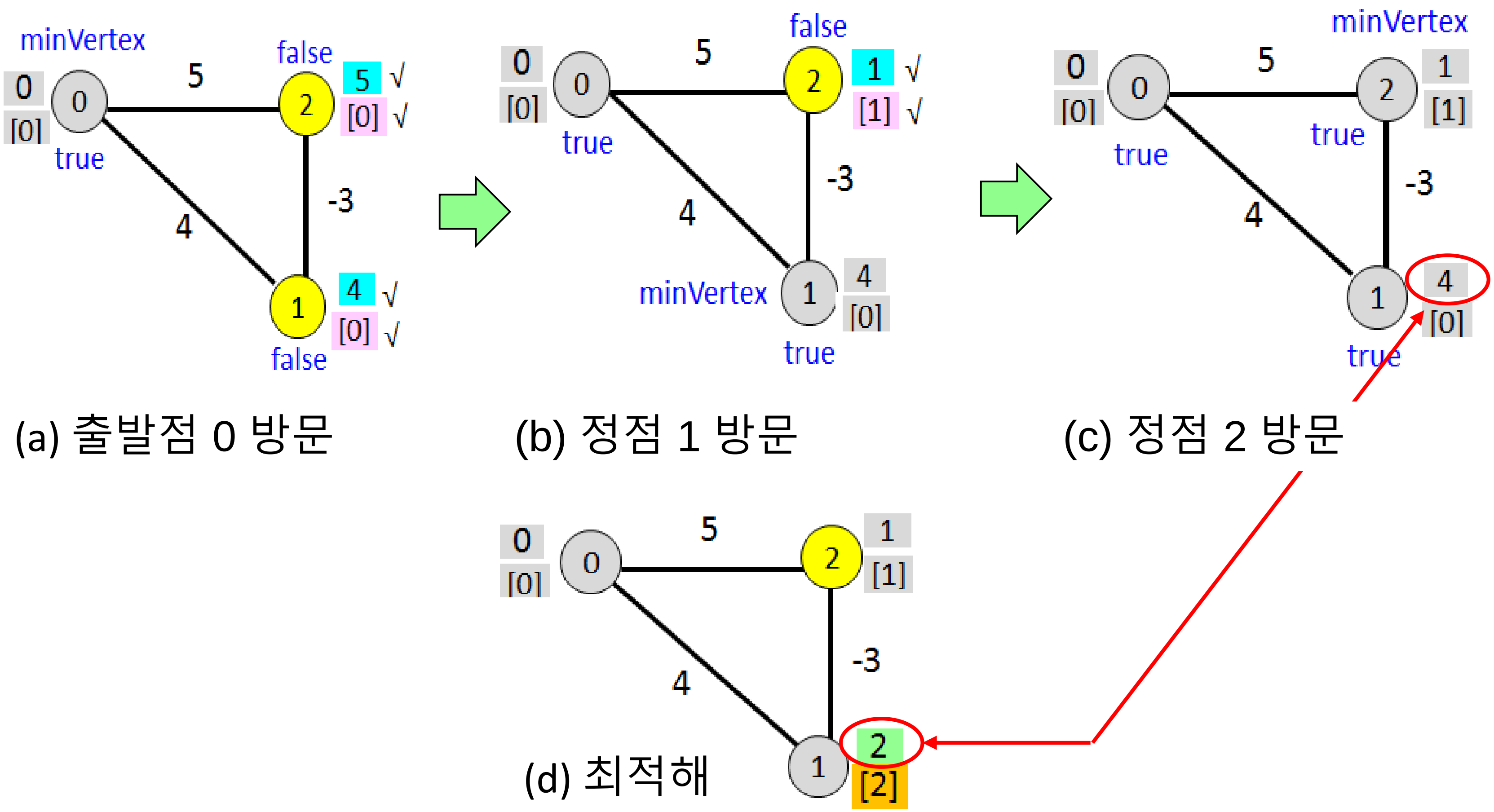
- **Dijkstra 알고리즘은 N번의 반복을 거쳐 minVertex를 찾고 minVertex에 인접하면서 방문되지 않은 정점들에 대한 간선완화를 시도**
  - 이후 배열 D에서 minVertex를 탐색하는데 O(N) 시간이 소요되고, minVertex에 인접한 정점들을 검사하여 D의 원소들을 갱신하므로 추가로 O(N) 시간이 소요
  - 따라서 총 수행시간은 Nx(O(N) +O(N)) = $O(N^2)$
- **minVertex를 찾기 위해 이진힙(Binary Heap)을 사용하면 각 정점의 D의 원소를 힙에 저장하므로 힙 크기는 N**
  - 또한 minVertex찾기는 delete_min연산으로 수행하고, 간선완화는 decrease_key 연산을 수행한다. 이때 각 연산에 O(logN) 시간이 소요

수행시간(2)

- 알고리즘은 minVertex를 N번 찾고, 최대 M번의 간선완화를 수행하므로 총 O(NlogN+MlogN) = O((M+N)logN) 시간이 필요
- 만약 입력그래프가 희소그래프라면, 예를 들어, M = O(N)이라면, 수행시간이 O(MlogN) = O(NlogN)이 되어 이진힙을 사용하는 것이 매우 효율적
- minVertex를 찾기 위해 피보나치힙(Fibonacci Heap)을 사용하면 O(NlogN + M) 시간에 Dijkstra 알고리즘을 수행
- Dijkstra알고리즘과Prim알고리즘은 동일한 수행시간을 가짐

- Dijkstra 알고리즘은 입력그래프에 음수가중치가 있으면 최단경로 찾기에 실패하는 경우가 발생

- Dijkstra 알고리즘이 최적해를 찾지 못하는 반례



(a) 출발점 0 방문

(b) 정점 1 방문

(c) 정점 2 방문

(d) 최적해

- (a) 출발점이 방문되어 visited[0] = true
- 이후 D[1] = 4, previous[1] = 0 그리고 D[2] = 5, previous[1] = 0으로 각각 갱신
- (b) D[1]이 최솟값이므로 정점 1이 방문되고, D[2] = 1, previous[1] = 1로 갱신
- (c) 마지막으로 방문되지 않은 정점 2가 방문되고 알고리즘이 종료
- 그러나 (d)를 보면 출발점 0에서 정점 1까지 최단경로는 [0-2-1]이고, 경로의 길이는 2
- [이러한 문제점이 발생한 이유] Dijkstra 알고리즘이 D의 원소 값의 증가 순으로 minVertex를 선택하고, 한번 방문된 정점의 D 원소를 다시 갱신하지 않기 때문

요약

- 그래프를 자료구조로서 저장하기 위해 인접행렬과 인접리스트가 주로 사용
- 그래프는 깊이우선탐색(DFS)과 너비우선탐색(BFS)으로 그래프의 모든 정점들을 방문하며, DFS는 스택을사용하고, BFS는 큐 자료구조를 사용
- 위상정렬 알고리즘은DFS를 수행하며 각 정점 v의 인접한 모든 정점들의 방문이 끝나자마자 v 를 리스트에 추가한다. 리스트의 역순이 위상정렬이다.

요약

- Kruskal 알고리즘은 간선들을 가중치로 정렬한 후에, 가장 가중치가 작은 간선이 트리에 싸이클을 만들지 않으면 트리 간선으로 선택하고, 만들면 버리는 일을 반복하여 N-1개의 간선을 선택
- Prim 알고리즘은 트리에 인접한 가장 가까운 정점을 하나씩 추가하여 최소신장트리를 만든다.
- Sollin 알고리즘은 각 트리에서 트리에 연결된 간선들 중에서 가장 작은 가중치를 가진 간선을 선택한다. 이때 선택된 간선은 두 개의 트리를 하나의 트리로 합친다. 이와 같은 방식으로 하나의 트리가 남을 때까지 각 트리에서 최소 가중치 간선을 선택하여 연결
- Dijkstra 알고리즘은 출발점으로부터 방문 안된 정점들 중에서 가장 가까운 거리의 정점을 방문하고 방문한 정점을 기준으로 간선완화를 수행하여 최단경로를 계산