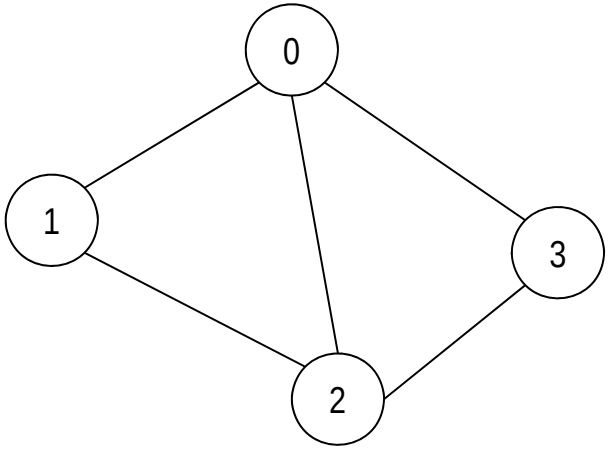
그래프

그래프

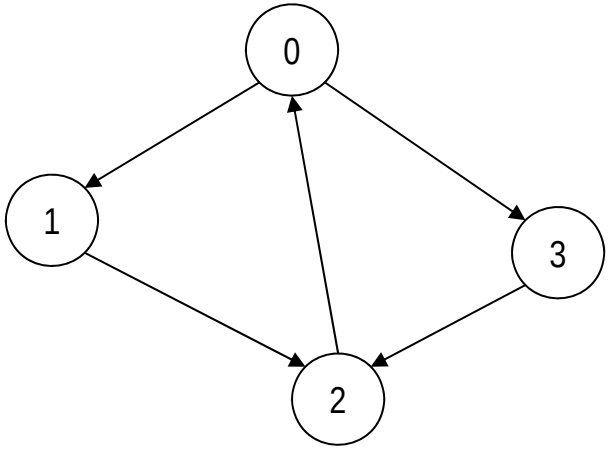
- 그래프(graph)는 정점(vertex)들의 집합 V와 간선(edge)들의 집합 E로 정의된다.
- 그래프는 간선의 방향성 유무에 따라 무방향 그래프와 방향 그래프로 구분된다.

V = { 0, 1, 2, 3 }
E = { (0,1), (0,2), (0,3), (1,2), (2,3) }



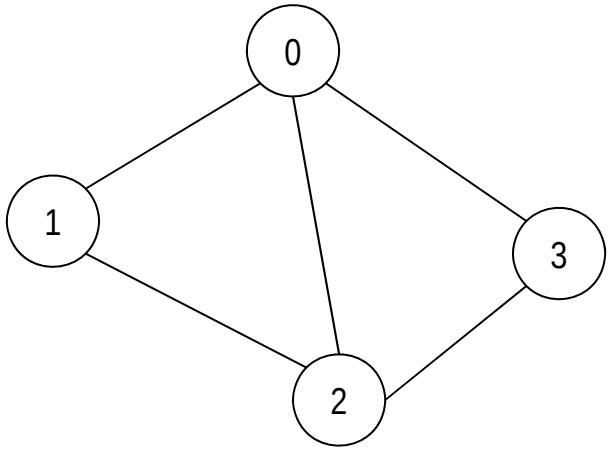
무방향 그래프(undirected graph)

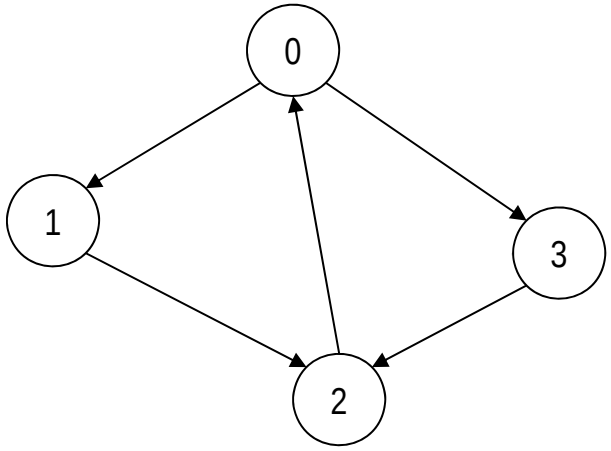
V = { 0, 1, 2, 3 }
E = { <0,1>, <0,3>, <1,2>, <2,0>, <3,2>}



방향 그래프(directed graph)

그래프 표현: 인접행렬(adjacency matrix)

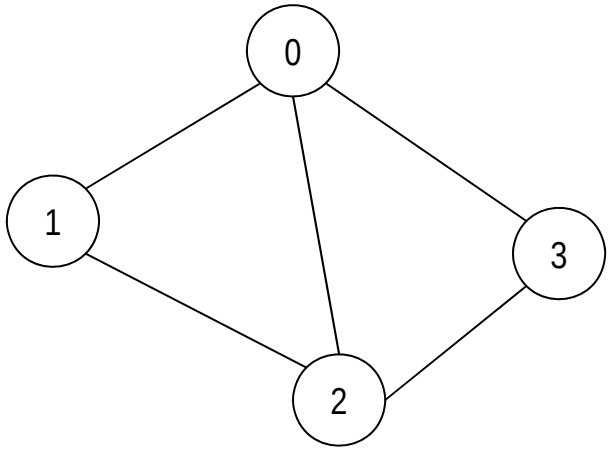


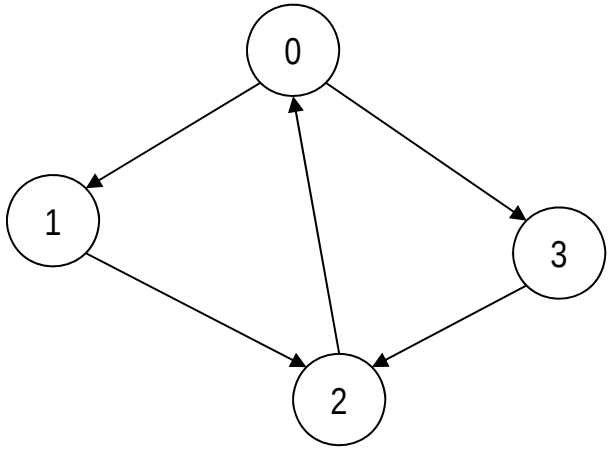


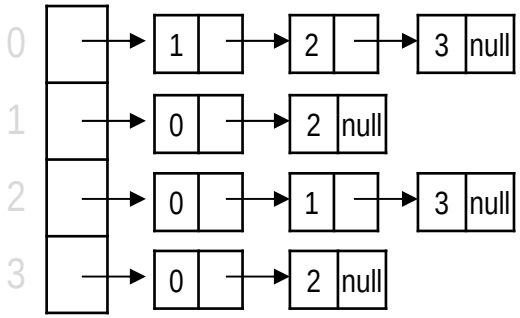
	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	1
3	1	0	1	0

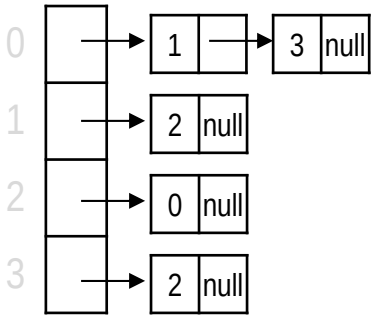
	0	1	2	3
0	0	1	0	1
1	0	0	1	0
2	1	0	0	0
3	0	0	1	0

그래프 표현: 인접리스트(adjacency list)



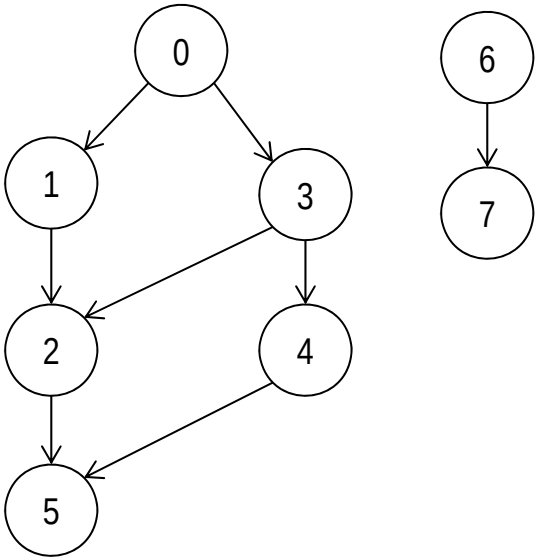




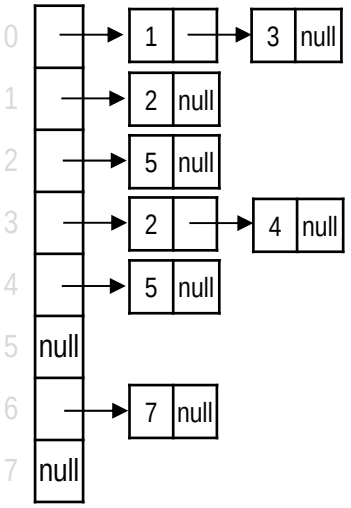


그래프 표현

V = { 0, 1, 2, 3, 4, 5, 6, 7 }
E = { <0,1>,<0,3>,<1,2>,<3,2>,<3,4>,<2,5>,<4,5>,<6,7> }



	0	1	2	3	4	5	6	7
0		1		1				
1			1					
2						1		
3			1		1			
4						1		
5								
6								1
7								

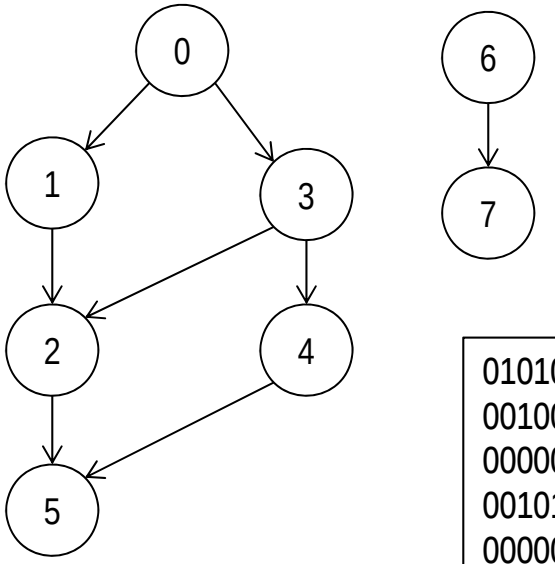


그래프 표현: 인접행렬

```
public class Test {
	public static void main(String[] args) {

		int	V=8;
		String	input="0 1 0 3 1 2 3 2 3 4 2 5 4 5 6 7";

		int	adjMat[][]=new int[V][V];

		String	s[]=input.split("\\s+");
		for (int i = 0; i < s.length; i+=2){
			int	v1=Integer.parseInt(s[i]);
			int	v2=Integer.parseInt(s[i+1]);
			adjMat[v1][v2]=1;
		}
		for (int i = 0; i < adjMat.length; i++) {
			for (int j = 0; j < adjMat[i].length; j++) {
				System.out.print(adjMat[i][j]);
			}
			System.out.println();
		}
	}
}
```

V = { 0, 1, 2, 3, 4, 5, 6, 7 }

E = { <0,1>,<0,3>,<1,2>,<3,2>,<3,4>,<2,5>,<4,5>,<6,7> }




```
01010000
00100000
00000100
00101000
00000100
00000000
00000001
00000000
```

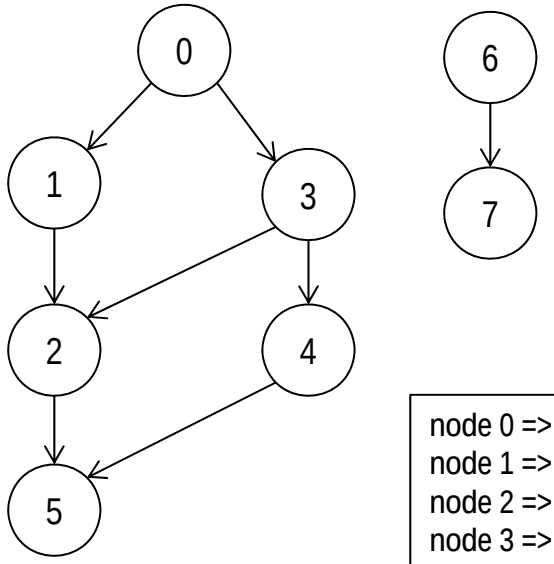
실습: 무방향 그래프로 처리하도록 코드를 수정하시오.

그래프 표현: 인접리스트

```
public class Test {
	public static void main(String[] args) {

		int	V=8;
		String	input="0 1 0 3 1 2 3 2 3 4 2 5 4 5 6 7";

		LinkedList<Integer> adjList[]=new LinkedList[V];
		for (int i = 0; i < adjList.length; i++){
			adjList[i]=new LinkedList<>();
		}
		String	s[]=input.split("\\s+");
		for (int i = 0; i < s.length; i+=2){
			int	v1=Integer.parseInt(s[i]);
			int	v2=Integer.parseInt(s[i+1]);
			adjList[v1].add(v2);
		}
		for (int i = 0; i < adjList.length; i++){
			System.out.println("node "+i+" => "+adjList[i]);
		}
	}
}
```

V = { 0, 1, 2, 3, 4, 5, 6, 7 }

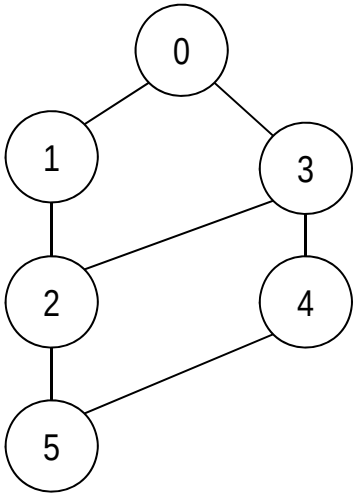
E = { <0,1>,<0,3>,<1,2>,<3,2>,<3,4>,<2,5>,<4,5>,<6,7> }



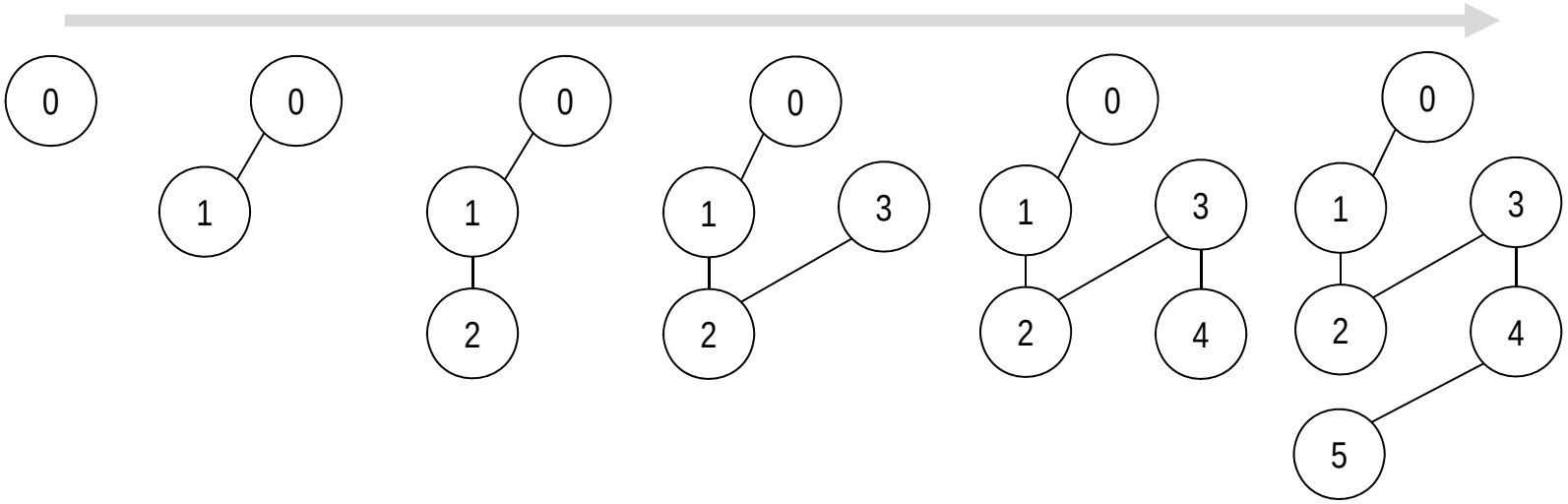

```
node 0 => [1, 3]
node 1 => [2]
node 2 => [5]
node 3 => [2, 4]
node 4 => [5]
node 5 => []
node 6 => [7]
node 7 => []
```

실습: 무방향 그래프로 처리하도록 코드를 수정하시오.

깊이우선탐색(depth first search, dfs)

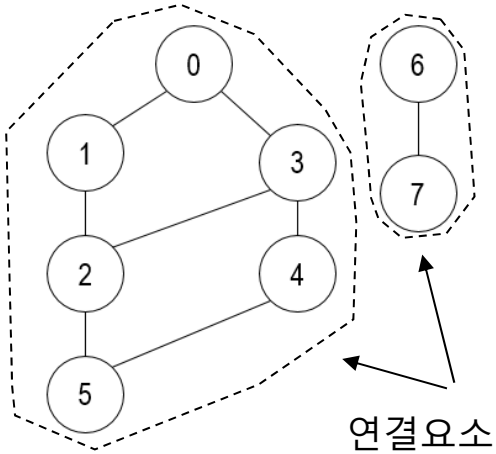


- **깊이우선탐색**(dfs): 현재 노드를 방문(표시) 및 처리하고 현재 노드의 미방문 인접노드에 대해 깊이우선탐색(dfs)을 다시 적용한다.
- **시간복잡도**: 인접리스트 활용 시 O(E), 인접행렬의 경우 O(V^2)



깊이우선탐색(depth first search, dfs)

```
public class Test {
	public static void main(String[] args) {
		String input="0 1 0 3 1 2 3 2 3 4 2 5 4 5 6 7"; // 무방향 간선으로 해석
		int	V=8;
		 LinkedList<Integer> adjList[]=new LinkedList[V];
		... // input으로부터 인접리스트 생성 (무방향그래프로 처리)
		connected(adjList, V);
	}
	private static void connected(LinkedList<Integer>[] adjList, int V) {
		Boolean	visited[]=new Boolean[V];
		for (int I = 0; i < visited.length; i++) {
			if(visited[i]==false){
				dfs(adjList, V, visited, i);
				System.out.println();
			}
		}
	}
	private static void dfs(LinkedList<Integer>[] adjList, int V, Boolean[] visited, int v) {
		visited[v]=true;
		System.out.print(v+ "    " );
		for (Integer i : adjList[v])
			if(visited[i]==false) dfs(adjList, V, visited, i);
		}
	}
}
```

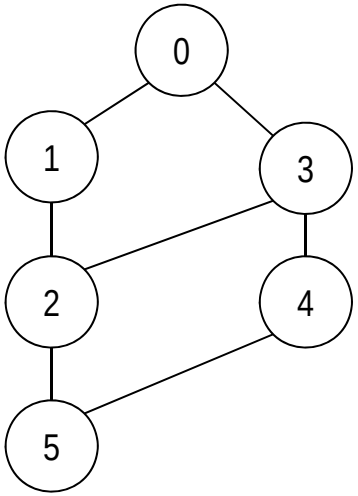


연결요소 2개

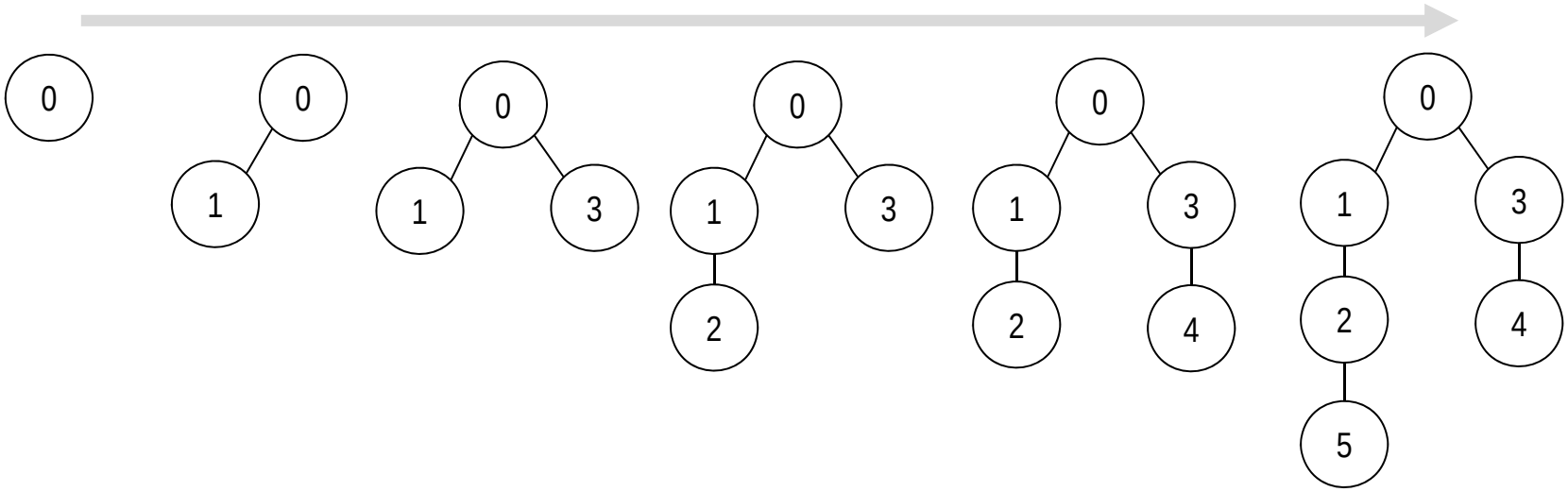
- **연결요소탐색**: 그래프 내 모든 연결요소들을 탐색하기 위해 미방문 노드에 대해 dfs를 반복 호출한다

- **깊이우선탐색**(dfs): 현재 노드를 방문(표시) 및 처리하고 현재 노드의 미방문 인접노드에 대해 깊이우선탐색을 다시 적용한다.

너비우선탐색(breadth first search, bfs)



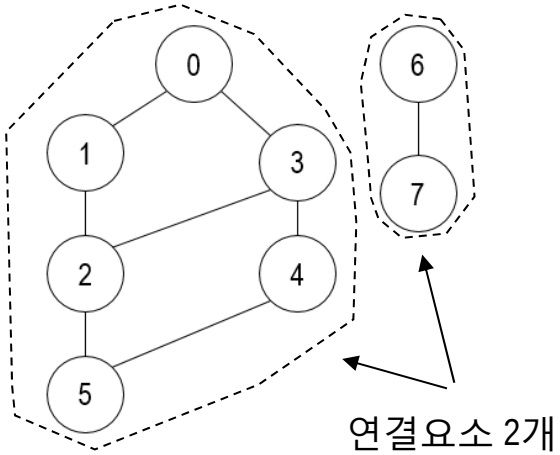
- **너비우선탐색**(bfs): 시작 노드를 방문(표시)하고 큐에 삽입한 다음, 공백 큐가 아닌 동안 "*큐에서 추출된 노드의 미방문 인접 노드들을 방문(표시)하고 큐에 삽입하는 작업*"을 반복한다.
- **시간복잡도**: 인접리스트 활용 시 $O(E)$, 인접행렬의 경우 $O(V^2)$



너비우선탐색(breadth first search, bfs)

```
public class Test {
	public static void main(String[] args) {
		String  input="0 1 0 3 1 2 3 2 3 4 2 5 4 5 6 7"; // 무방향 간선으로 해석
		int	V=8;
		 LinkedList<Integer> adjList[]=new LinkedList[V];
		... // input으로부터 인접리스트 생성 (무방향그래프로 처리)
		connected(adjList, V);
	}
	private static void connected(LinkedList<Integer>[] adjList, int V) {
		boolean	visited[]=new boolean[V];
		for (int i = 0; i < visited.length; i++) {
			if(visited[i]==false){
				bfs(adjList, V, visited, i);
				System.out.println();
			}
		}
	}
	private static void bfs(LinkedList<Integer>[] adjList, int V, boolean[] visited, int v) {
		LinkedList<Integer>  queue=new LinkedList<>();
		visited[v]=true;  queue.addLast(v);
		while(!queue.isEmpty()){
			v=queue.removeFirst();
			System.out.print(v+" ");
			for (Integer w : adjList[v]) {
				if(visited[w]==false){ visited[w]=true;  queue.addLast(w); }
			}
		}
	}
}
```

- **연결요소탐색**: 그래프 내 모든 연결요소들을 탐색하기 위해 미방문 노드에 대해 bfs를 반복 호출한다



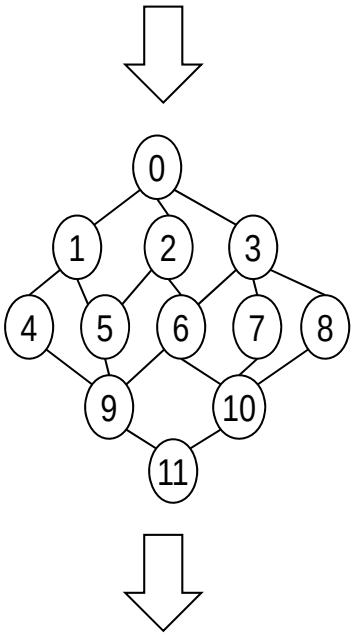
- **너비우선탐색**(bfs): 시작 노드를 방문(표시)하고 큐에 삽입한 다음, 공백 큐가 아닌 동안 "*큐에서 추출된 노드의 미방문 인접 노드들을 방문(표시)하고 큐에 삽입하는 작업*"을 반복한다.

BFS 경로 탐색

Reference: https://algs4.cs.princeton.edu/41graph/BreadthFirstPaths.java.html, GPLv3

```
private static void bfs(LinkedList<Integer>[] adjList, int V, int start){
        boolean     visited[]=new boolean[V];
        double      dist[]=new double[V];
        int     prev[]=new int[V];
        for (int i = 0; i < dist.length; i++) dist[i]=Double.MAX_VALUE;
        for (int i = 0; i < prev.length; i++) prev[i]=-1;
        LinkedList<Integer>    queue=new LinkedList<>();
        visited[start]=true;
        dist[start]=0;
        queue.addLast(start);
        while(!queue.isEmpty()){
                int     v=queue.removeFirst();
                for (Integer w : adjList[v]) {
                        if(visited[w]==false){
                                visited[w]=true;
                                dist[w]=dist[v]+1;
                                prev[w]=v;
                                queue.addLast(w);
                        }
                }
        }
        System.out.println(Arrays.toString(dist));
        System.out.println(Arrays.toString(prev));
}
```

bfs(adjList, 12, 0);



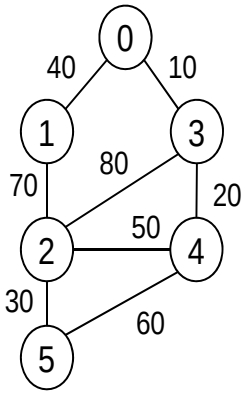
실습: 다음과 같이 시작노드으로부터 나머지 각 노드로의 bfs 경로를 출력하시오

```
0 ~ 0 => [0]
0 ~ 1 => [0, 1]
0 ~ 2 => [0, 2]
0 ~ 3 => [0, 3]
0 ~ 4 => [0, 1, 4]
0 ~ 5 => [0, 1, 5]
0 ~ 6 => [0, 2, 6]
0 ~ 7 => [0, 3, 7]
0 ~ 8 => [0, 3, 8]
0 ~ 9 => [0, 1, 4, 9]
0 ~ 10 => [0, 2, 6, 10]
0 ~ 11 => [0, 1, 4, 9, 11]
```

	0	1	2	3	4	5	6	7	8	9	10	11
dist	0	1	1	1	2	2	2	2	2	3	3	4
prev	-1	0	0	0	1	1	2	3	3	4	6	9

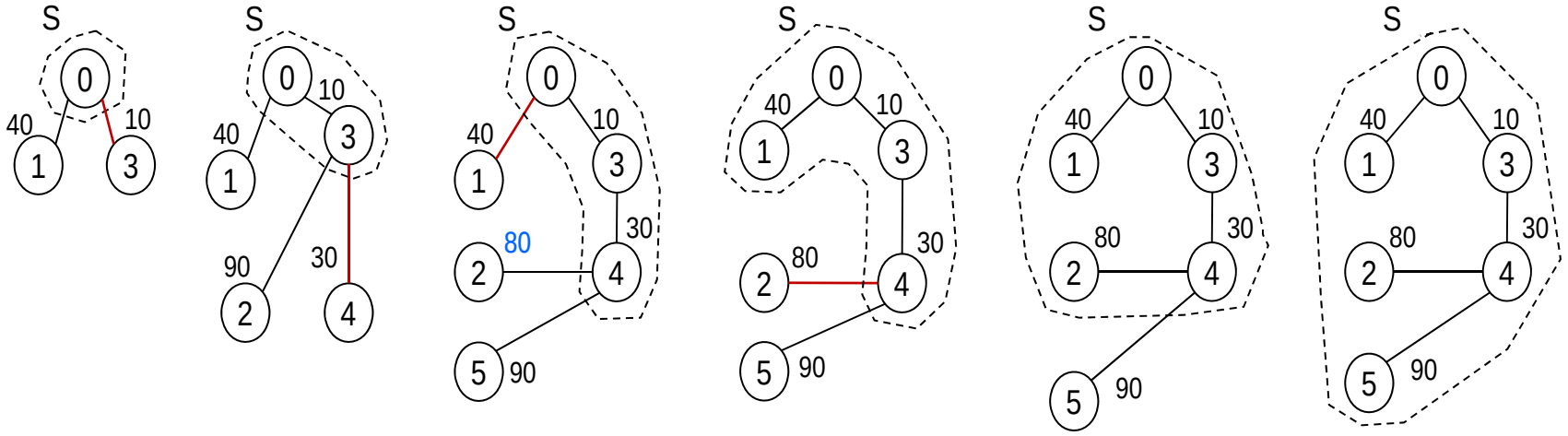
최단경로탐색: Dijkstra's algorithm

Reference: https://en.wikipedia.org/wiki/Dijkstra's_algorithm, CC-BY-SA
Reference: p.299 in (Horowitz et al., 1993)



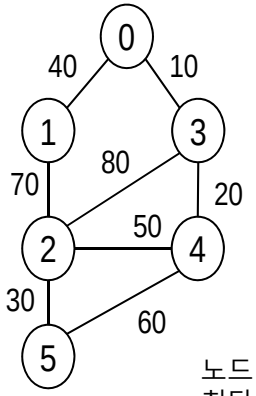
Dijkstra's shortest path algorithm (시작노드 s로부터 다른 모든 노드로의 최단 경로를 탐색)

- S: 시작노드 s로부터의 최단경로가 알려진 노드들의 집합. 최초 S={}
- dist: 크기 V(그래프 노드 개수)의 배열로 dist[v]는 s로부터 v까지의 거리 저장.
- 최초 dist[s]=0, 나머지 각 노드 v는 dist[v]=∞
- *모든 노드의 최단경로가 발견될 때까지 "**최단거리 미결정 노드들 중 s로부터의 최단거리 노드 u를 S에 추가하고, s로부터 u를 거쳐 도달할 수 있는 노드들의 최단거리를 갱신하는 작업**"*을 반복한다.



최단경로탐색: Dijkstra's algorithm

Reference: https://en.wikipedia.org/wiki/Dijkstra's_algorithm, CC-BY-SA
Reference: p.299 in (Horowitz et al., 1993)

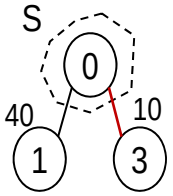


Dijkstra's shortest path algorithm (시작노드 s로부터 다른 모든 노드로의 최단 경로를 탐색)

- S: 시작노드 s로부터의 최단경로가 알려진 노드들의 집합. 최초 S={}
- dist: 크기 V(그래프 노드 개수)의 배열로 dist[v]는 s로부터 v까지의 거리 저장.
- 최초 dist[s]=0, 나머지 각 노드 v는 dist[v]=∞
- *모든 노드의 최단경로가 발견될 때까지 "**최단거리 미결정 노드들 중 s로부터의 최단거리 노드 u를 S에 추가하고, s로부터 u를 거쳐 도달할 수 있는 노드들의 최단거리를 갱신하는 작업**"*을 반복한다.

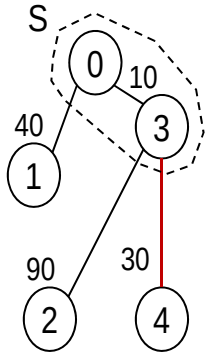
노드 0,1,2,3,4,5 중 최단거리노드 u

	0	1	2	3	4	5
dist	0	∞	∞	∞	∞	∞
dist	0	40	∞	10	∞	∞



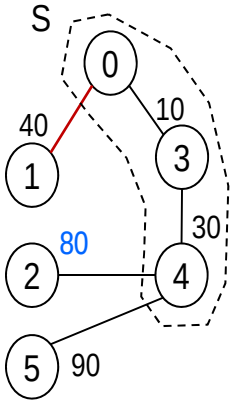
노드 1,2,3,4,5 중 최단거리노드 u

	0	1	2	3	4	5
	0	40	∞	10	∞	∞
	0	40	90	10	30	∞



노드 1,2,4,5 중 최단거리노드 u

	0	1	2	3	4	5
	0	40	90	10	30	∞
	0	40	80	10	30	90



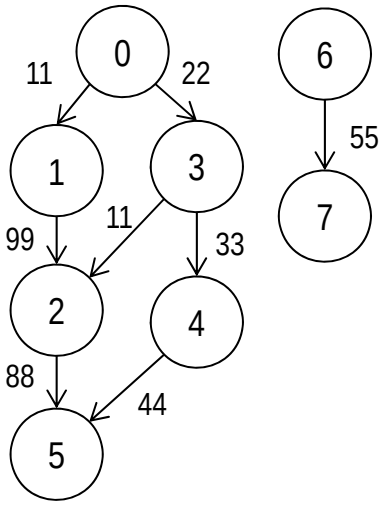
IF dist[v] > dist[u]+adjMat[u][v]
THEN dist[v]=dist[u]+adjMat[u][v]

90 30 50
IF dist[2] > dist[4]+adjMat[4][2]
THEN dist[2]=dist[4]+adjMat[4][2]

최단경로탐색: Dijkstra's algorithm

Reference: https://en.wikipedia.org/wiki/Dijkstra's_algorithm, CC-BY-SA
Reference: p.299 in (Horowitz et al., 1993)

```
public class Test {
	public static void main(String[] args) {
		String	input="0 1 11 0 3 22 1 2 99 3 2 11 3 4 33 2 5 88 4 5 44 6 7 55";
		int	V=8;
		double		adjMat[][]=new double[V][V];
		String	s[]=input.split("\\s+");
		for (int i = 0; i < s.length; i+=3){
			int	v1=Integer.parseInt(s[i]), v2=Integer.parseInt(s[i+1]);
			adjMat[v1][v2]=Double.parseDouble(s[i+2]);
		}
		DijkstraShortestPath(adjMat, V, 0);
	}
	private static void DijkstraShortestPath(double[][] adjMat, int V, int source) {
		boolean		found[]=new boolean[V];
		double dist[]=new double[V];
		int	prev[]=new int[V];
		for (int i = 0; i < V; i++) { dist[i]=Double.MAX_VALUE; prev[i]=-1; }
		dist[source]=0;
		for (int i = 0; i < V; i++) {
			double min=Double.MAX_VALUE;
			int	u=-1; // 최단거리 미결정 노드 중 최단거리 노드
			for (int v = 0; v < V; v++) if(!found[v] && dist[v]<min){ min=dist[v]; u=v; }
			if(u==-1) break;
			found[u]=true;
			for (int v = 0; v < V; v++) {
				if(!found[v] && adjMat[u][v]>0 && dist[v]>dist[u]+adjMat[u][v]){ dist[v]=dist[u]+adjMat[u][v]; prev[v]=u; }
			}
		}
		System.out.println(Arrays.toString(dist)+"\n"+Arrays.toString(prev));
	}
}
```



	0	1	2	3	4	5	6	7
0		11		22				
1			99					
2						88		
3			11		33			
4						44		
5								
6								55
7								

최단거리 미결정 노드들 중 시작노드 source로부터의 최단거리 노드 u를 결정

최단거리 노드 u를 S에 추가

u를 거쳐 도달할 수 있는 노드들의 최단거리 갱신

References

- C로 쓴 자료구조론 (Fundamentals of Data Structures in C, Horowitz et al.). 이석호 역. 사이텍미디어. 1993.
- 쉽게 배우는 알고리즘: 관계 중심의 사고법. 문병로. 한빛아카데미. 2013.
- C언어로 쉽게 풀어 쓴 자료구조. 천인국 외 2인. 생능출판사. 2017.
- 프로그래밍 콘테스트 챌린징, Akiba 등 공저, 로드북, 2011.
- https://introcs.cs.princeton.edu/
- Introduction to Algorithms, Cormen et al., 3rd Edition (The MIT Press)