자료구조설계: 2013

Topological Sort

강지훈
jhkang@cnu.ac.kr
충남대학교 컴퓨터공학과

Activity Networks

Activity Networks

- AOV-network (Activity-On-Vertex)
  - It is a digraph such that
    - Vertex : task or activity
    - Edge : precedence between tasks
  - When $i \rightarrow j$ is a directed path in AOV-network,
    - i is called a predecessor of j.
    - j is called a successor of i.
  - When $< i, j >$ is an edge in AOV-network,
    - i is called an immediate predecessor of j.
    - j is called an immediate successor of i.

Relations

Relations

- The cartesian product $A \times B$ on sets A and B is the set of ordered pairs $<a,b>$ such that if $a \in A$ and $b \in B$ then $<a,b> \in A \times B$.
  - $A \times B = \{<a,b> \mid a \in A \text{ and } b \in B\}$
  - Note that $<a,b> \neq <b,a>$.
- A (binary) relation R on sets A and B is a subset of the cartesian product $A \times B$ on the sets A and B.
- A binary relation R on a set S is a subset of the cartesian product $S \times S$ on the set S and S.
- Notation:
  - $<a,b> \in R \Leftrightarrow a \; R \; b$
  - $<a,b> \in \bullet \Leftrightarrow a \bullet b$
  - $<a,b> \in < \; \Leftrightarrow a < b$

Exmaple

- Let $S = \{a,b,c\}$.
- The cartesian product $S \times S$ on the set S is:

  $S \times S = \{<a,a>, <a,b>, <a,c>, <b,a>, <b,b>, <b,c>, <c,a>, <c,b>, <c,c>\}$
- Any subset of $S \times S$ is a binary relation on S.
  - The number of elements of $S \times S$ is $9(=3^2)$.
  - There are $2^9(=512)$ subsets of $S \times S$.
    - $R1=\{<a,a>,<b,b>,<c,c>\}$ is a relation on S since $R1$ is a subset of $S \times S$.
    - $R2=\{<a,b>,<c,a>\}$ is a relation on S since $R2$ is a subset of $S \times S$.
  - We are interested in some relations with some specific properties, but not in all the 512 relations.

Example: '<' (less than) relation

- '<' (less than) relation on the nonnegative integer set I
  - $< = \{<0,1>,<0,2>,<0,3>,<0,4>, \cdots\cdots$
    $<1,2>,<1,3>,<1,4>,<1,5>, \cdots\cdots$
    $<2,3>,<2,4>,<2,5>,<2,6>, \cdots\cdots$
    $\cdots\cdots$
    $\}$
  - $<5,129> \in <$ $(5 < 129)$ since 5 is less than 129.
  - $<7,7> \notin <$ $(7 < 7)$ since 7 is not less than 7.

Partial Order

Partial Order

- A (binary) relation $\bullet$ on a set S is irreflexive (on S) iff for any element $x \in S$, $x \bullet x$ is false.
  (i.e., for any $x \in S$, $<x,x> \notin \bullet$).
  - (Eg.) Let be $<$ (the 'less than' relation) on the set I of nonnegative integers. Then $x < x$ is not true for any nonnegative integer in I. That is, (x,x) is not a member of the relation $<$ for any x in I. So, $<$ is irreflexive (on I).
- A relation $\bullet$ on a set S is transitive iff $i \bullet j$ and $j \bullet k \Rightarrow i \bullet k$ for all i, j, k in S.
  - (Eg.) Let $<$ and I be the same as above.
    Then $i < j$ and $j < k \Rightarrow i < k$ for all integer i, j, k .
    So, $<$ is transitive.
- A relation $\bullet$ is a partial order on a set S iff $\bullet$ is both irreflexive and transitive on S.
  - (Eg) The above $<$ relation is a partial order on S.

Example of a partial order

$S = \{1, 2, 3\}$

The power set $\wp$ of S:

$\wp(S) = \{\phi,\{1\},\{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$

Let $\subset$ be the proper subset relation.

- Example:

  $<\{1\},\{1,2,3\}> \in \subset$ since $\{1\} \subset \{1,2,3\}$ is true.

Then $\subset$ is a partial order on $\wp(S)$.

- A $\not\subset$ A for any A $\in \wp(S)$.
  So, $\subset$ is irreflexive.
- A $\subset$ B and B $\subset$ C implies A $\subset$ C for any A, B, C $\in \wp(S)$.
  So, $\subset$ is transitive.

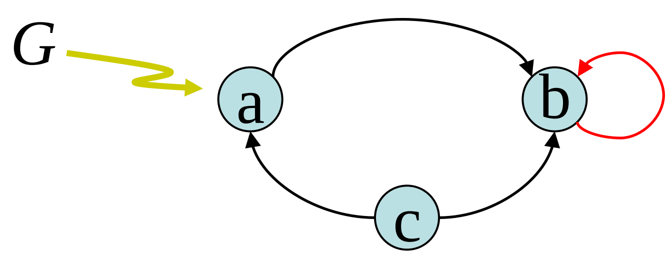
Graph representation of Binary relations

- Any binary relation can be mapped to a directed graph, and vice versa.
- Let R be a binary relation on a set S.
  Then, the corresponding directed graph G is just (S,R) such that
  - Every element of S becomes a vertex of G.
  - Every pair $<a,b>$ in R becomes a directed edge from the vertex a to the vertex b.
- Example:

  Let $S = \{a,b,c\}$ and

  $R = \{<b,b>, <a,b>, <c,a>, <c,b>\}$ be a relation on S.

  Then $G = (S,R)$ is :

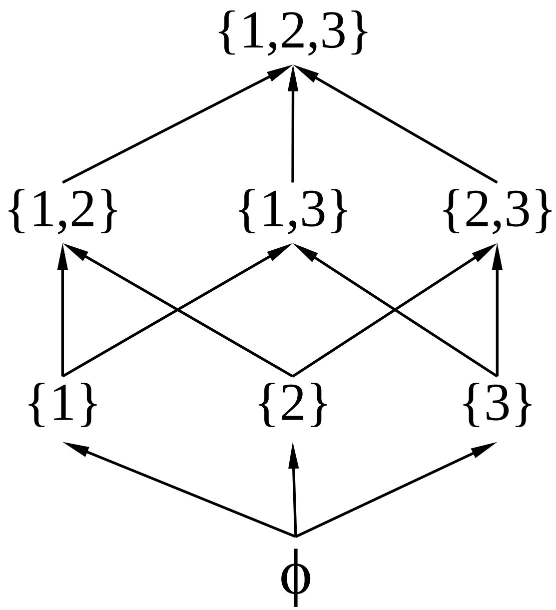


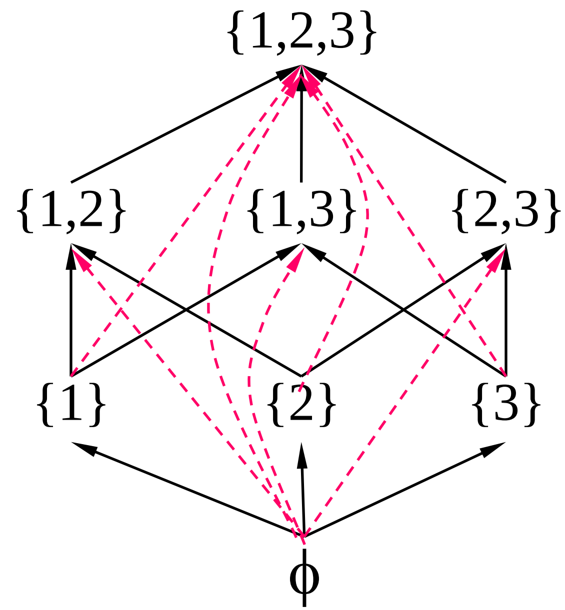
Dag (directed acyclic graph)

- A dag (directed acyclic graph) is a directed graph with no cycles.
- Dags are useful in representing partial orders.
  - Every partial order is represented by a dag, but not vice versa.
    - If E is a partial order on a set V, then the digraph $G = (V,E)$ is a dag.
    - If $G = (V, E)$ is a dag and E^+ is a transitive closure of E, then E^+ is a partial order on V.

Example: Representation of Partial Order by dag

- Let $S = \{1, 2, 3\}$ and let $\subset$ be the proper subset relation on $\wp(S)$.



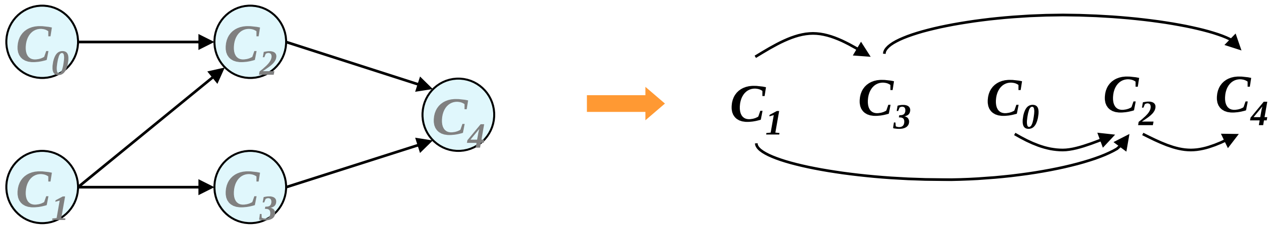


- Let $R1 = \{\text{all} \rightarrow\}$ and $R2 = \{\text{all} \rightarrow \text{ and } \rightarrow\}$. Then $R2$ is $\subset$ and $R2 = R1^+$.
  - $G1 = (\wp(S), R1)$ is just a dag, but not represents a partial order.
  - $G2 = (\wp(S), R2) = (\wp(S), \subset)$ is a dag that represents a partial order $\subset$ on $\wp(S)$.

Topological Sort

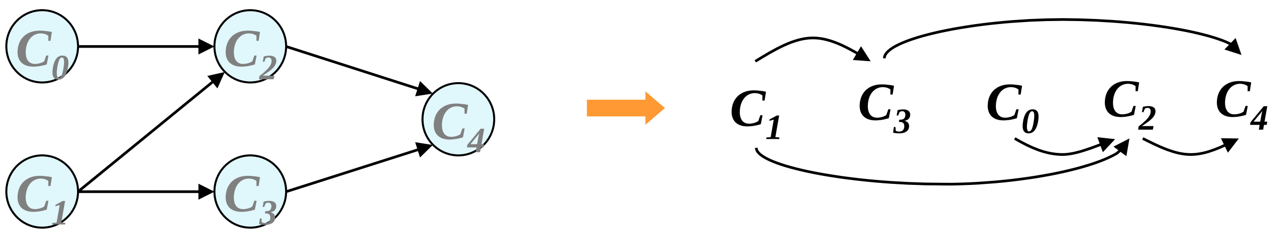
Topological Sort

- A topological order is a linear order of the vertices of a graph such that,
  for any vertices i and j, if i is a predecessor of j in the network then i precedes j in the linear order.
- The topological sort is a transformation from a given partial order into a linear (topological) order.

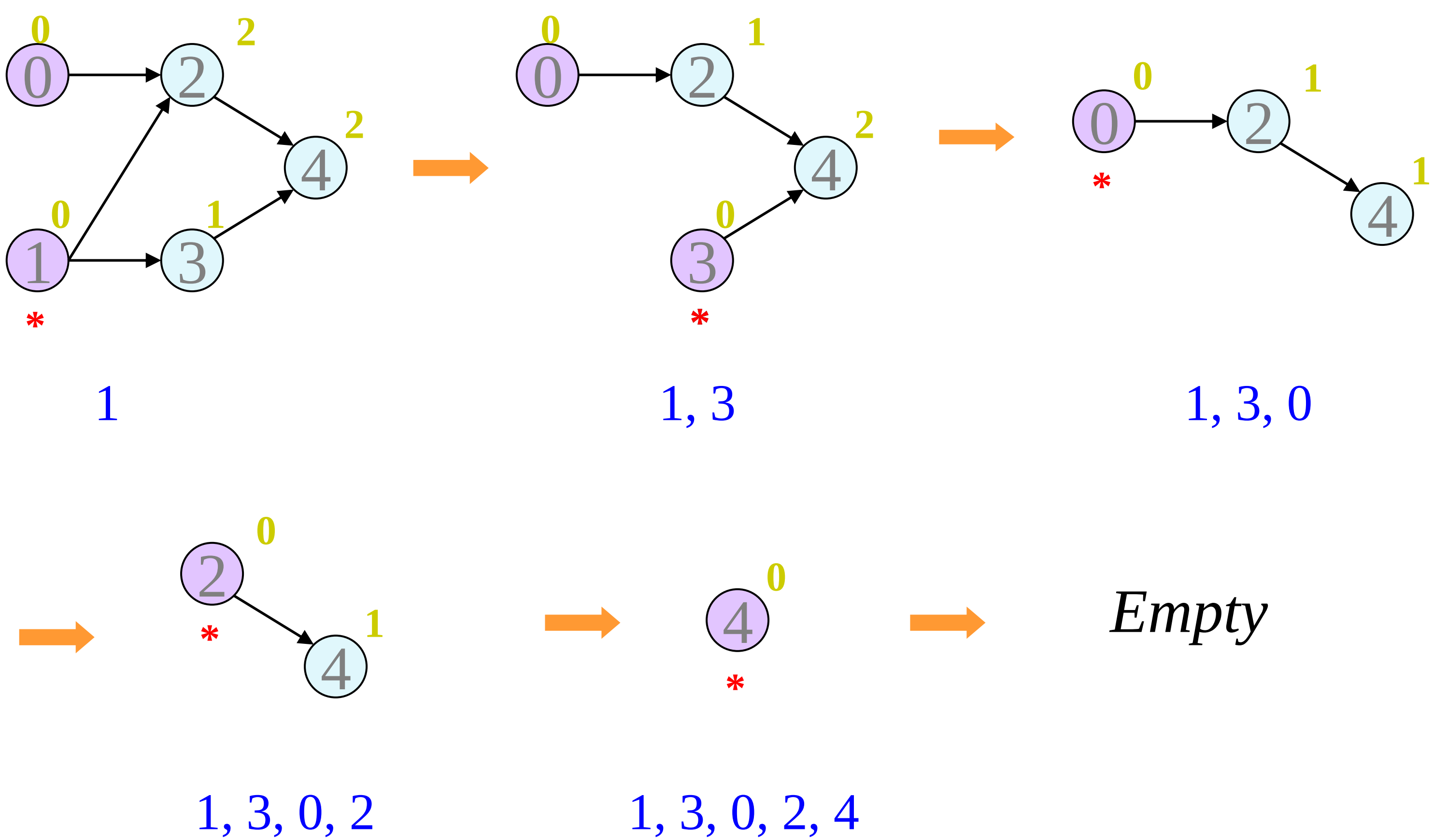


Basic Idea of Topological sort

- Find out a vertex with no immediate predecessors.
- Delete the vertex and all edges coming out from the vertex.
- Repeat these two steps until all the vertices have been listed,
  or all the remaining vertices have predecessors.
- In the latter case, the graph has a cycle.
  So, we cannot get a topological order.



Example: Topological Sort

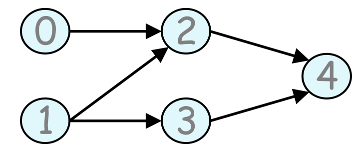


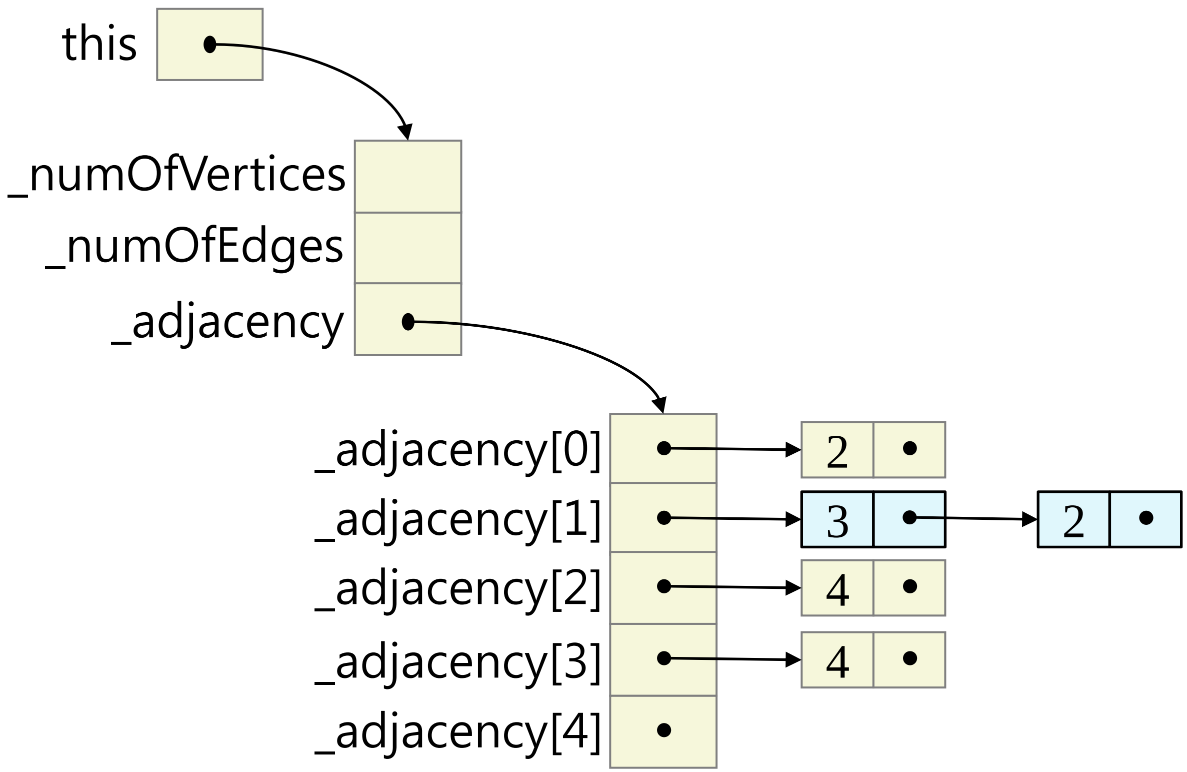
Algorithm

```
1.  Input the AOV network.
    (Let N be # of vertices)
2.  i = 0 ;
3.  while ( (i < N) && (there is a vertex with no predecessors) ) {
4.          pick a vertex v which has no predecessors ;
5.          output v ;
6.          delete v and all edges adjacent from v ;
7.          i++ ;
8.  }
9.  if ( i < N) {
10.         ...... // We cannot find topological order since the graph has a cycle.
11. }
12. else {
13.         ...... // We have got a topological order
14. }
```

Data Representation

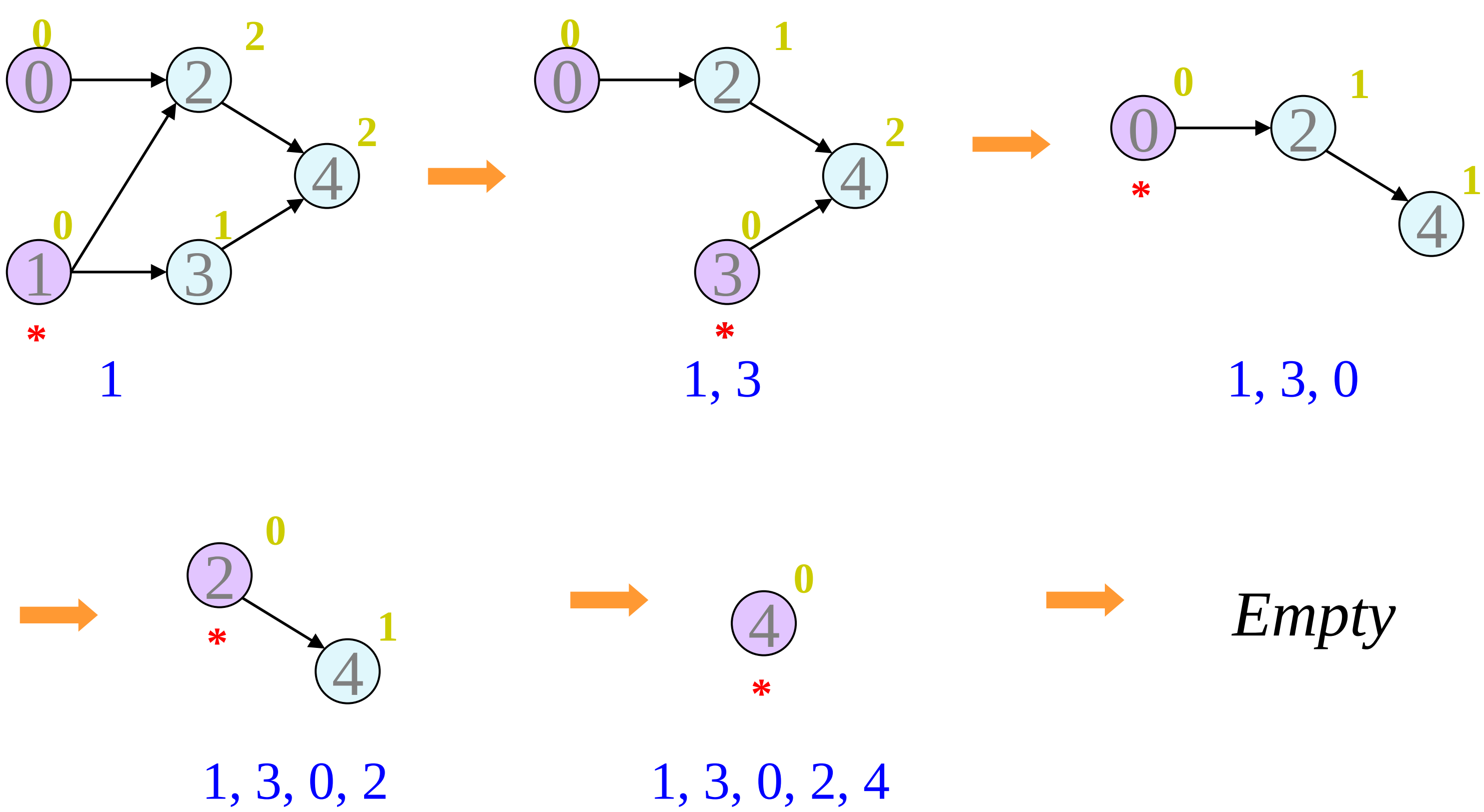
Graph by Adjacency List





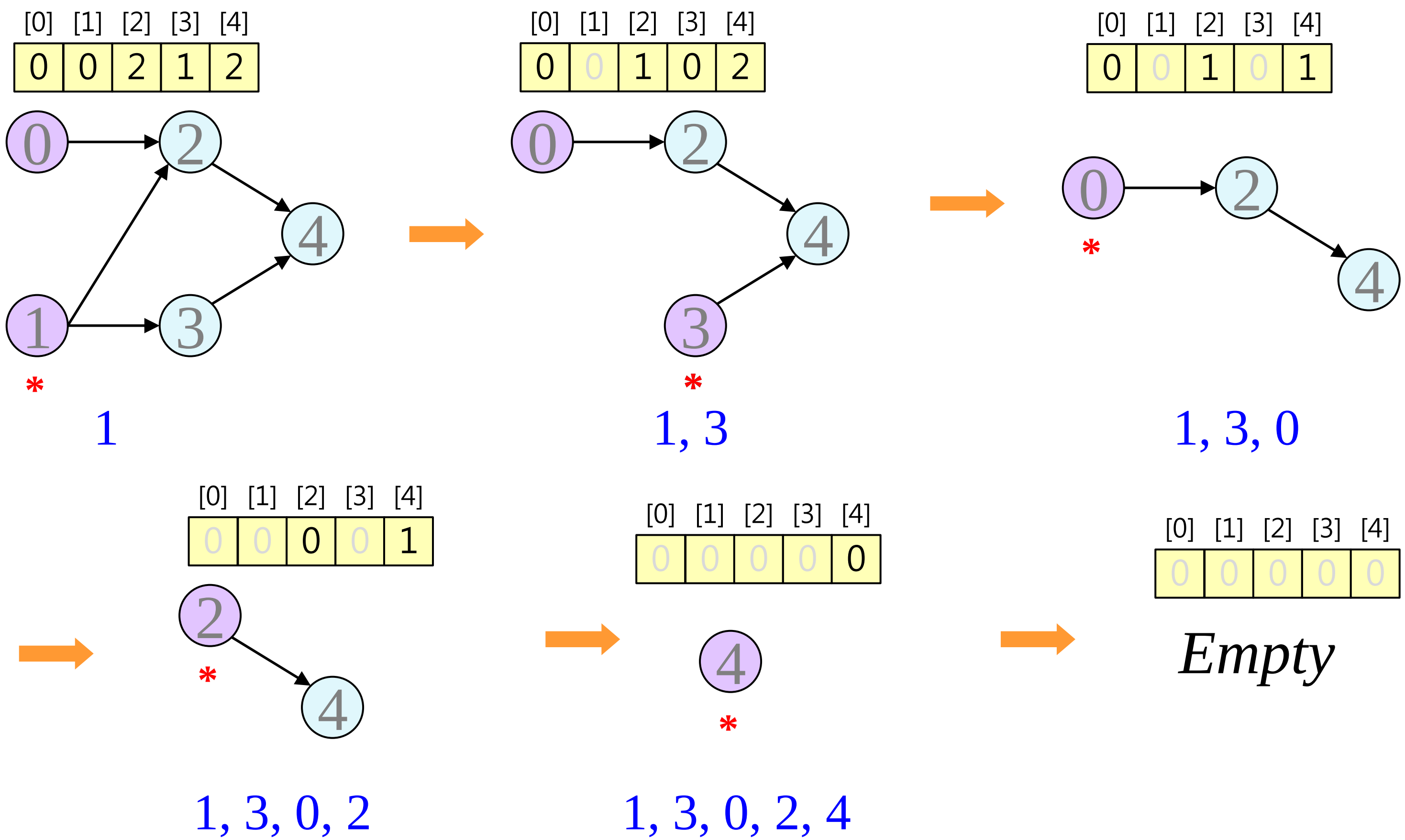
❑ Vertices with No Immediate Predecessors

- How to find and manage the vertices with no immediate predecessors?



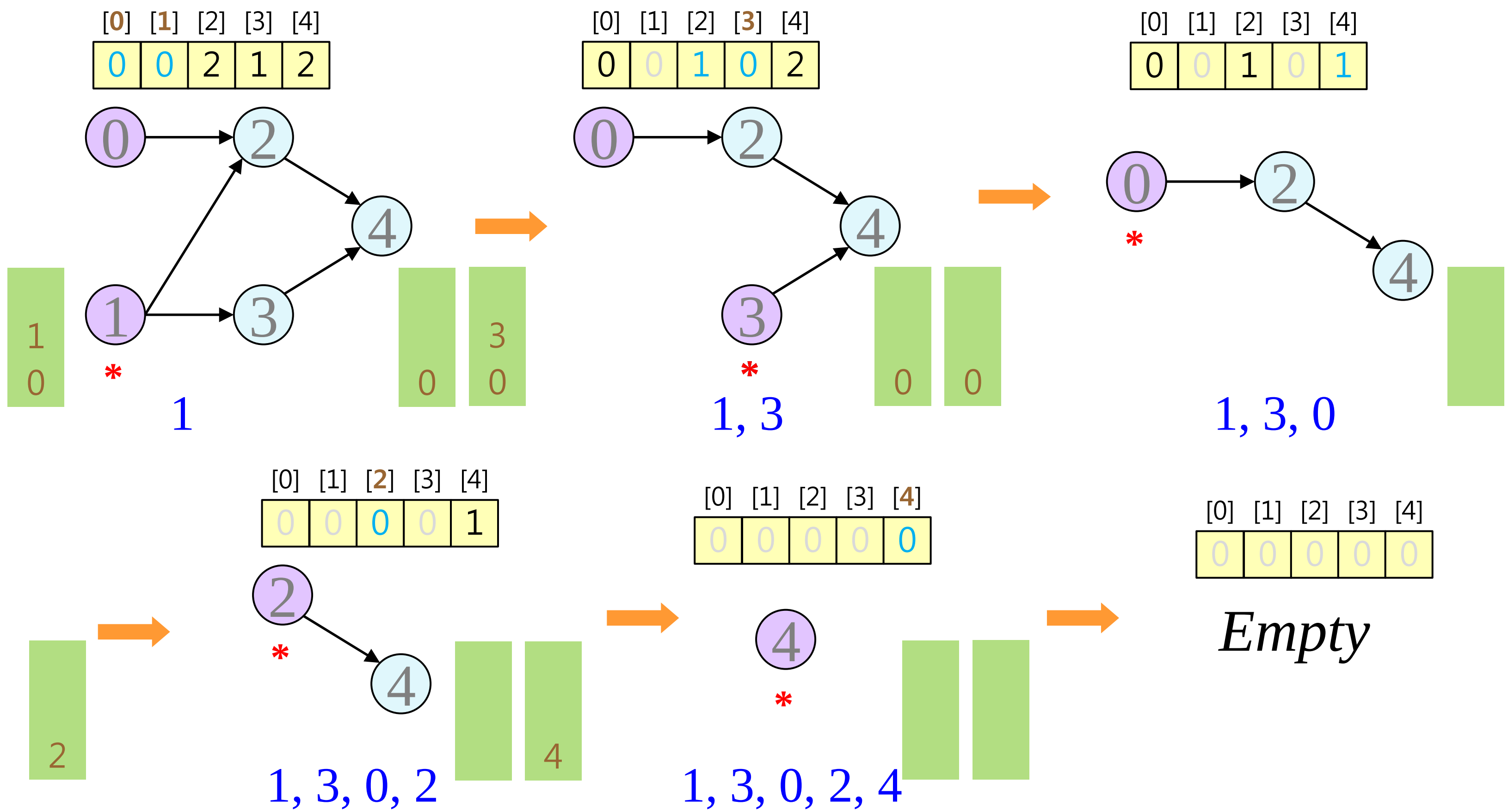
Vertices with No Immediate Predecessors

- How to find and manage the vertices with no immediate predecessors?
  - For each vertex, maintain the number of immediate predecessors.



How to find the vertices with no immediate predecessors efficiently ?

- Use a stack for such vertices! : _zeroCountVertices



Algorithm: More Specific

```
Input an AOV network.
Initialize _predecesorCount[] ;
Initialize Stack _zeroCountVertices ;
i = 0 ;
while ( ! _zeroCountVertices.isEmpty() ) {
        v = _zeroCountVertices.pop() ; // pick a vertex v which has no predecessors ;
        output v ;
        for (each vertex w adjacent from v) { // delete v and all edges adjacent from v
                _predecessorCount[w] -- ;
                if ( _predecessorCount[w] == 0 )
                        _zeroCountVertices.push(w) ;
        }
        i++ ;
}
if ( i < _numOfVertices) {
        ...... // We cannot find topological order since the graph has a cycle.
}
else {
        ...... // We have got a topological order
}
```

Predecessor Count

- int _predecessorCount[] ;
  - size: number of vertices
  - _predecessorCount[i]: number of immediate predecessors of vertex i.

- How to set the initial value of _predecessorCount[] ?
  - Initially, set all zero.
  - When we make a graph by getting edges:

    For each edge (i,j), increment _predecessorCount[j] by 1.

- How to maintain?
  - Whenever we delete a vertex v with no immediate predecessors from the graph:

    For each edge (v,**w**), decrement _predecessorCount[**w**] by 1.

Algorithm: More Specific

(Assume there are n vertices and e edges in the given dag.)

```
Input the AOV network.
Initialize _predecesorCount[] ;
Initialize Stack ._zeroCountVertices ;
i = 0 ;
while ( ! _zeroCountVertices.isEmpty() ) {
        v = _zeroCountVertices.pop() ; // pick a vert
        output v ;
        for (each vertex w adjacent from v) { // dele
                _predecessorCount[w] -- ;
                if ( _predecessorCount[w] == 0 )
                        _zeroCountVertices.push(w) ;
        }
        i++ ;
}
if ( i < _numOfVertices ) {
        ...... // We cannot find topological order since the graph has a cycle.
}
else {
        ...... // We have got a topological order
}
```

$O(n+e)$

$O(n)$

$O(n)$ for total push & pop

Let d_v be the out-degree of vertex v.
For each v, there are $O(d_v)$ edges.

$$\sum_{v=0}^{n-1} O(d_v) = O(e)$$

So, totaly $O(n+e)$.

Implementation of Topological Sort

Application의 run()

```
public void run() {
    if ( ! this.inputAndMakeGraph() ) {
        System.out.println("오류: 그래프가 생성되지 않았습니다. 프로그램을 종료합니다.");
    }
    else {
        this.showGraph() ;
        TopSort  topSort = new TopSort(graph) ;
        List<Integer> topSortedList = topSort.perform() ;
        if ( topSortedList !=null ) {
            ...... //  위상정렬 결과인 topSortedList 를 출력한다
        }
        else {
            System.out.println("오류: 그래프에 사이클이 존재합니다.") ;
        }
    }
    System.out.println("위상정렬을 종료합니다.") ;
} //end of run ()
```

Class "TopSort"
for
Topological Sorting

Public Functions

- TopSort (Graph givenGraph)

- public List<Integer> perform()
  - Topological Sorting을 실행한다.
  - 결과로 위상정렬된 vertex 순서를 리스트로 얻는다
    - cycle이 존재하여 정상 종료가 되지 않으면 null을 얻는다.

Class “TopSort” 의 구현

Private Attributes

```
public class TopSort {
    private Graph  _graph ; // 위상정렬을 하도록 주어진 그래프
    private int  _predecessorCount ; // 각 vertex의 직전 선행자의 개수
    private Stack<Integer> _zeroCountVertices ;  // 선행자가 없는 vertex의 리스트
    private List<Integer> _topSortedList ;  // sort된 vertex 순서를 저장하는 리스트
    ......
}
```

생성자

- TopSort(Graph givenGraph)
  - 주어진 그래프 graph 로 위상 정렬할 준비를 한다.
  - graph는 이미 만들어져 있는 것이 주어진다.
  - 각 속성들을 초기화 한다.

```
public TopSort (Graph givenGraph)
{
    int   numOfVertices ;
    this._graph = givenGraph ; // 주어진 그래프를 위상정렬 할 그래프로 설정
    this.initPredecessorCount() ;
    numOfVertices = this._graph.numOfVertices() ;
    this._zeroCountVertices = new Stack<Integer>(numOfVertices) ;
    this._topSortedList = new List<Integer>(numOfVertices) ;
}// end of TopSort()
```

Other Public Functions

- public List<Integer> perform()
  - 위상정렬을 실행
  - 결과로 위상정렬된 vertex 순서를 리스트로 돌려준다
    - cycle이 존재하여 정상 종료가 되지 않으면 null을 돌려준다.
  - 사이클 검사
    - 맨 마지막에 다음 검사를 실행: true 이면 사이클이 존재

```
if ( this._topSortedList.size() < this._graph.numOfVertices() )  {
    this._topSortedList = null ; // 사이클이 존재
}
```

Private functions

- private void initPredecessorCount()
  - 배열 this._predecessorCount[]를 동적 할당을 실행
  - 배열의 크기 계산:
    - 원소의 type: int
    - 원소의 개수: this._graph.numOfVertices();

- private void setPredecessorCount ()
  - 먼저 배열 this._predecessorCount[]를 초기화 한다.
  - 그래프 전체 edge를 탐색 하면서 Edge e= new Edge(tailVertex,headVertex)에 대해, headVertex의 PredecessorCount, 즉 PredecessorCount[headVertex]를 하나 증가시킨다.

- private void pushVerticesWithZeroCount()
  - 그래프를 탐색하여 설정한 predecessorCount[]에서 값이 0인 vertex, 즉 직전선행자가 존재하지 않는 vertex들을 찾아 stack에 삽입한다.

Class “Graph”

Public Functions for Graph

- public Graph (int givenNumOfVertices)

- public int numOfVertices ()
- public int numOfEdges ()
- public boolean edgeAlreadyExist (Edge anEdge)

- public boolean addEdge (Edge anEdge)

- public void showGraph ()

Class “Graph”의 구현

Class Graph를 위한 basic Class

- Vertex와 Edge

```
public class Edge {
    int  _tailVertex ;
    int  _headVertex ;
}
```

- Adjacency List에서의 node

```
public class Node {
    private int  _headVertex ;
    private Node<Integer>  _next ;
    ....
}
```

Private Attributes

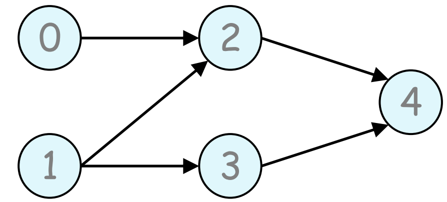
- Adjacency list로 구현한 경우

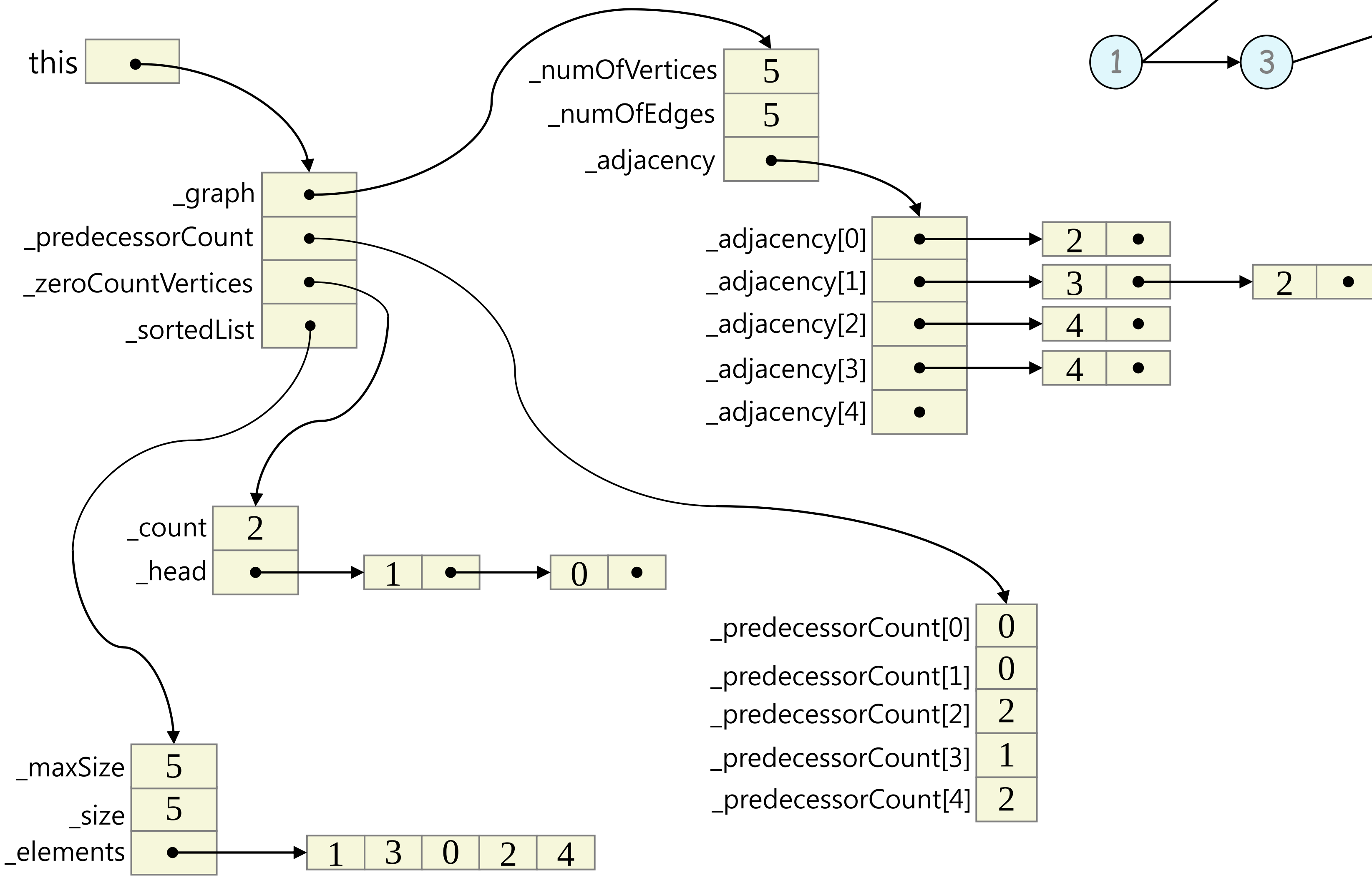
```
public class Graph {
    private int         _numOfVertices ;
    private int         _numOfEdges ;
    private Node[]      _adjacency ; // 각 adjaceny[i]가 linked list
    ...
} //Adjacency list로 구현한 경우
```

- Adjacency matrix로 구현한 경우

```
public class Graph {
    private int         _numOfVertices ;
    private int         _numOfEdges ;
    private int[][]     _adjacency ; // 2차원 배열
    ...
}
```

❑ 예: TopSort 객체


0
2
4
1
3


this
_graph
_predecessorCount
_zeroCountVertices
_sortedList
_numOfVertices
5
_numOfEdges
5
_adjacency
_adjacency[0]
_adjacency[1]
_adjacency[2]
_adjacency[3]
_adjacency[4]
2
3
2
4
4
_count
2
_head
1
0
_predecessorCount[0]
0
_predecessorCount[1]
0
_predecessorCount[2]
2
_predecessorCount[3]
1
_predecessorCount[4]
2
_maxSize
5
_size
5
_elements
1
3
0
2
4

End of
Topological Sort