Data Structures
資料結構

Image courtesy of www.escpeurope alumni.org

Graphs – Part I

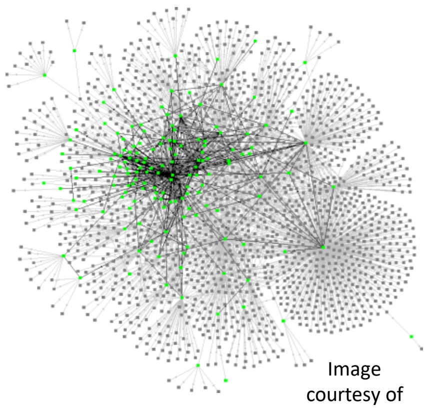


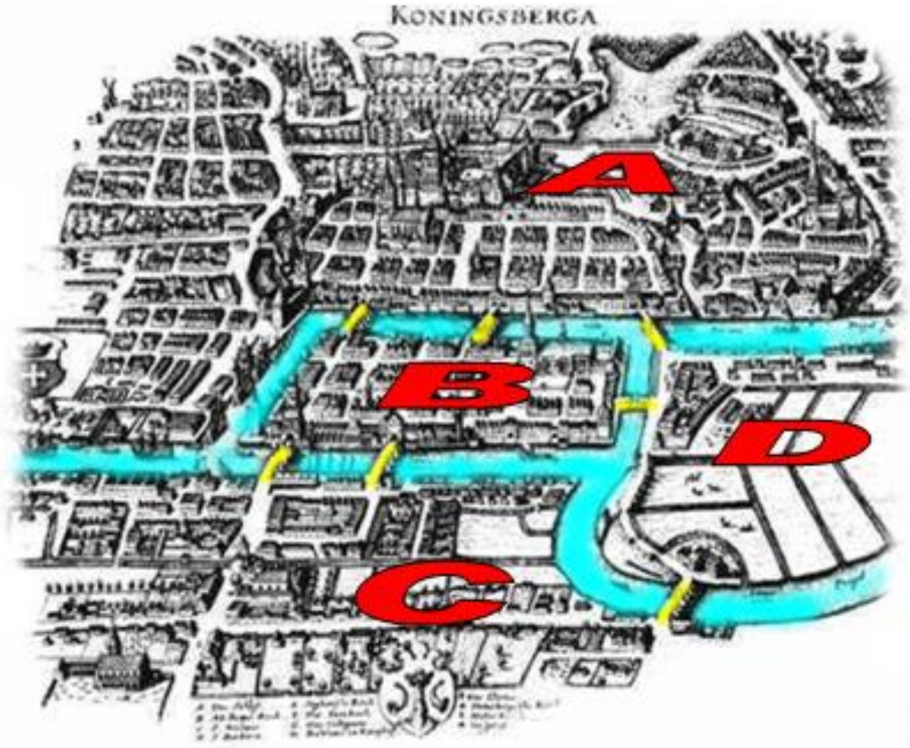
Image courtesy of Ross Mayfield

Department of Computer Science
National Tsing Hua University

Konigsberg Bridge Problem

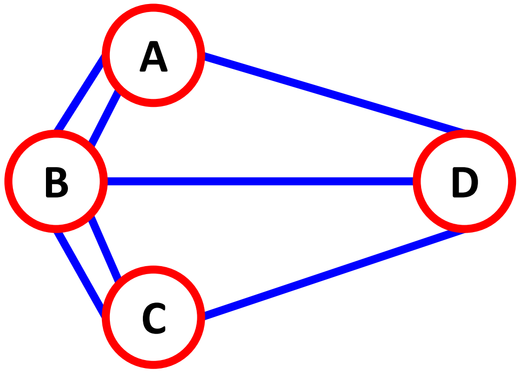
- 4 lands
- 7 bridges
- Problem:

Starting at one land, is it possible to **walk across all the bridges exactly once** and **returning to the starting land**?



Konigsberg Bridge Problem

- Euler formulate the problem as a graph.

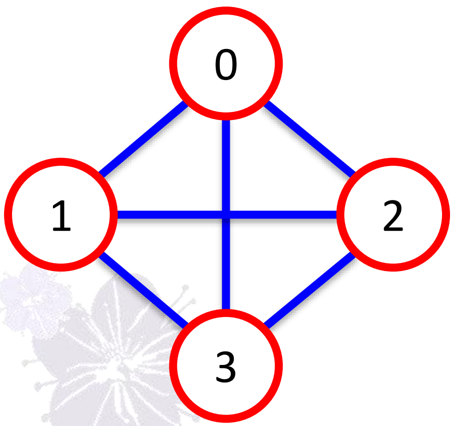


- Prove that the answer to the problem is possible iff **the degree of each vertex is even.**

Graph Definition

- A graph, ***G***, consists of two sets, ***V*** and ***E***.
  - ***G = (V, E)***
  - ***V*** : a set of **vertices**.
  - ***E*** : a set of *pairs of vertices* called ***edges***.
- **Undirected graph** (simply **graph**)
  - (u,v) and (v,u) represent the same edge.
- **Directed graph** (**digraph**)
  - <u,v> ≠ <v,u>
  - <u,v> => u is tail and v is head of edge.

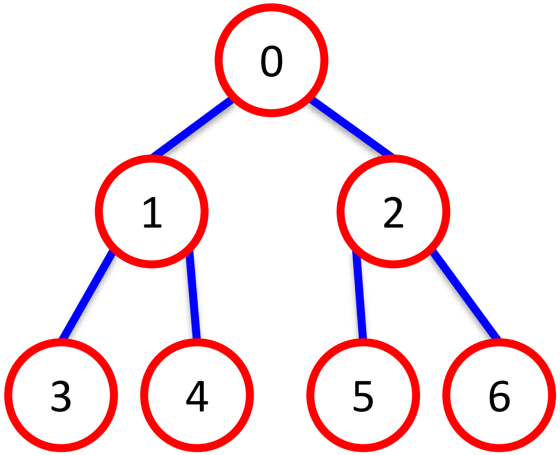
Examples



Undirected Graph

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

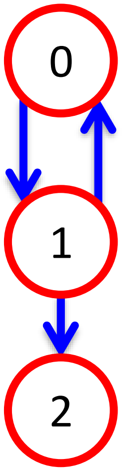
E(G)={(0,1), (0,2), (0,3),
(1,2), (1,3), (2,3)}



Undirected Graph

V(G)={0, 1, 2, 3, 4, 5, 6}

E(G)={(0,1), (0,2), (1,3),
(1,4), (2,5), (2,6)}



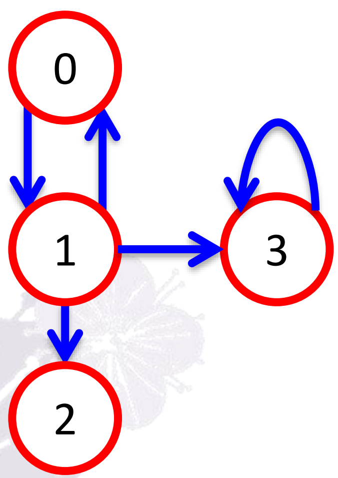
Directed Graph

V(G)={0, 1, 2}

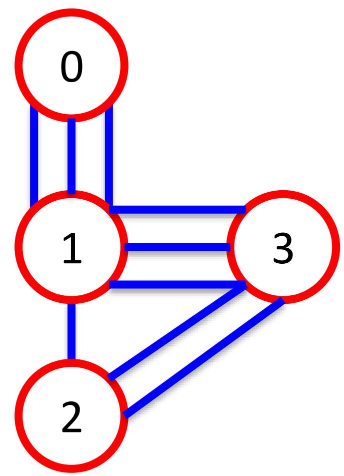
E(G)={<0,1>, <1,0>, <1,2>)}

Restrictions

- ***Self edges*** and ***self loops*** are not permitted!
  - Edges of the form (v, v) and <v, v> are not legal.
- A graph may not have multiple occurrences of the same edge (***multigraph***).



Graph with self edge



Multigraph

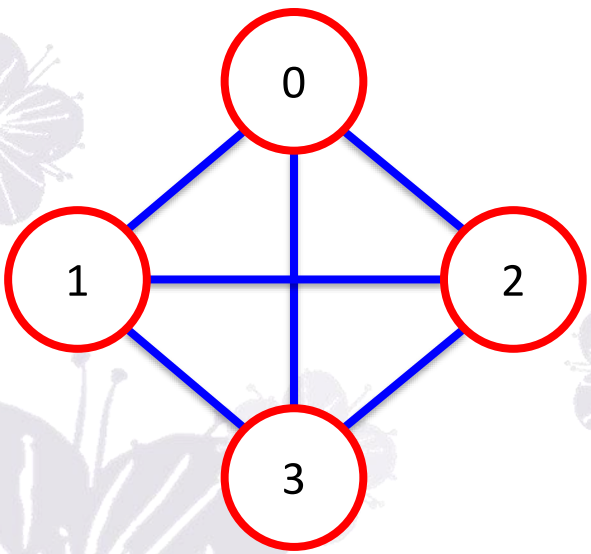
Terminology

- For a graph with n vertices, the maximum # of edges is:
  - $n(n-1)/2$ for undirected graph
  - $n(n-1)$ for directed graph
- Vertices u and v are **adjacent** if (u,v) ∈ E and edge (u,v) is **incident** on vertices u and v.
- For a directed edge <u,v>, we say u is **adjacent to** v and v is **adjacent from** u, and edge <u,v> is **incident** on vertices u and v.

Terminology

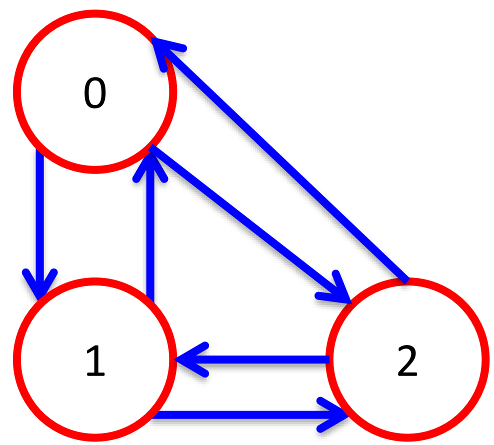
Complete undirected graph

- Graph with n vertices has exactly $n(n-1)/2$ edges.



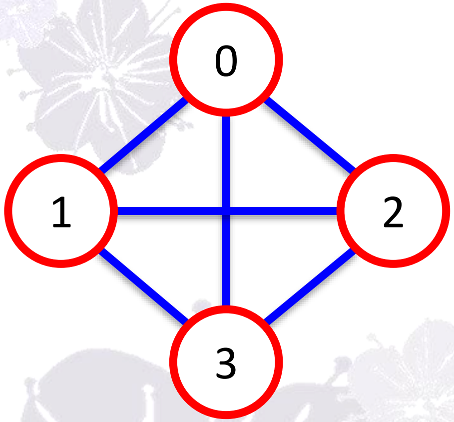
Complete directed graph

- Graph with n vertices has exactly $n(n-1)$ edges.

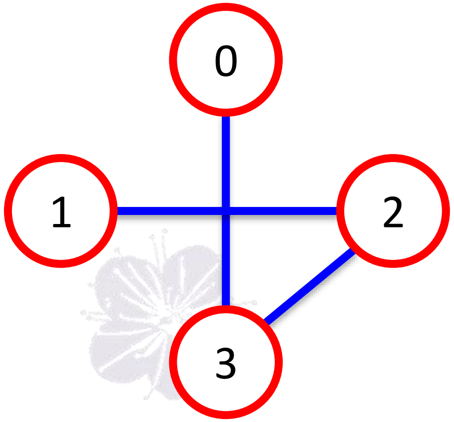


Terminology

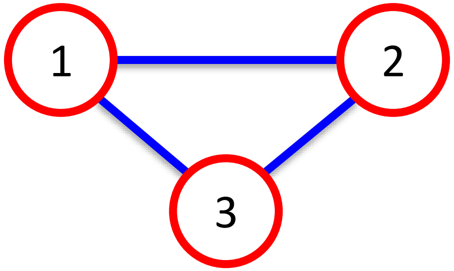
- **Subgraph**:
  - G’ is a subgraph of G
    such that V(G’) $\subseteq$ V(G) and E(G’) $\subseteq$ E(G).



Graph



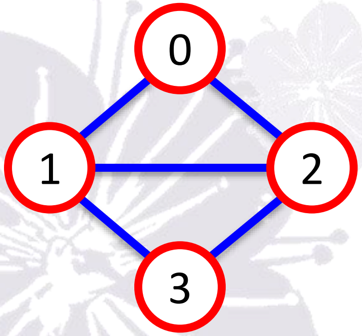
Subgraph



Subgraph

Terminology

- **Path**:
  - A path from **u** to **v** represents a sequence of vertices $\mathbf{u, i_1, i_2, \ldots, i_k, v}$ such that $(\mathbf{u, i_1}), (\mathbf{i_1, i_2}), \ldots, (\mathbf{i_k, v})$ are edges in graph.
- **Simple path:**
  - A simple path is a path in which all vertices except possibly the first and the last are distinct.



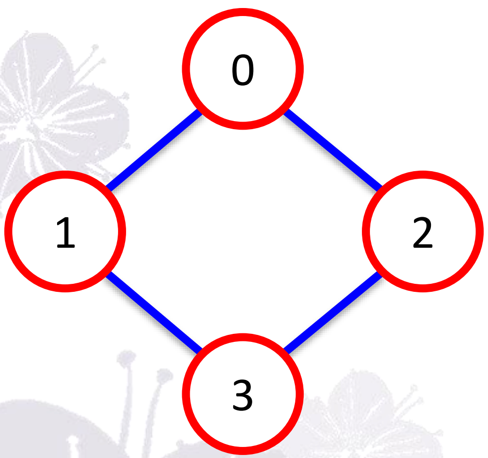
Sequence	Path?	Simple path?
0,1,3,2	Yes	Yes
0,2,0,1	Yes	No
0,3,2,1	No	No

Terminology

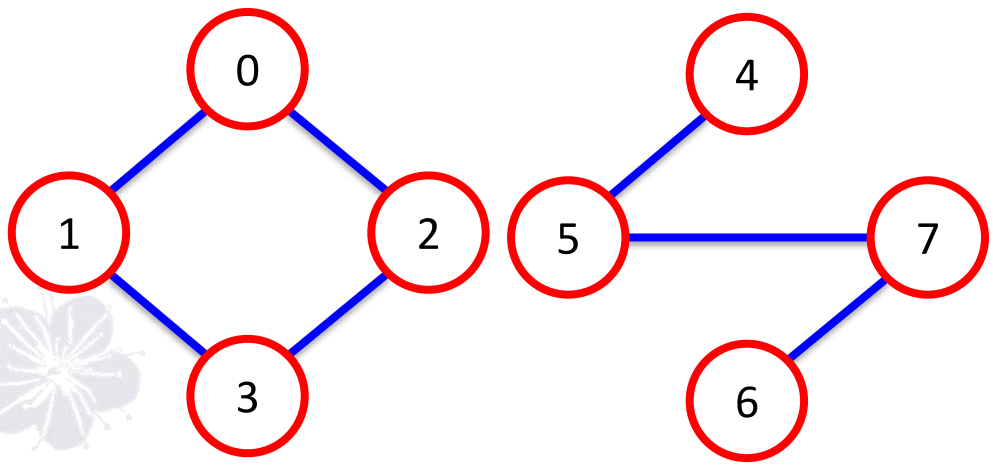
- **Cycle**:
  - A cycle is a simple path which the first and the last vertices are the same.
- Notes: if the graph is a directed graph, we usually add the prefix “directed” to above terms:
  - Directed path
  - Directed simple path
  - Directed cycle

Terminology

- Undirected graph G is said to be **connected** iff for **every pair of distinct vertices u and v**, there is a **path from u to v** in G.



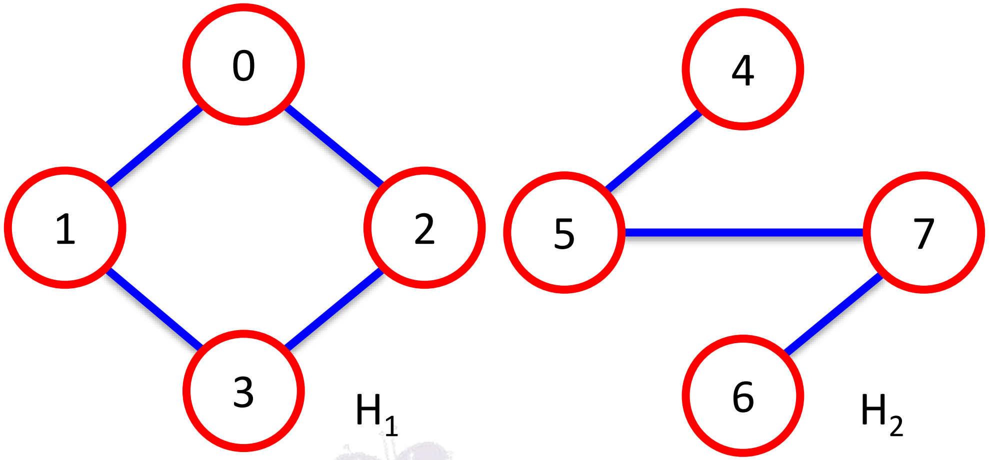
Connected graph



Not a connected graph

Terminology

- A **connected component**, H, of an undirected graph is a **maximal connected subgraph.**

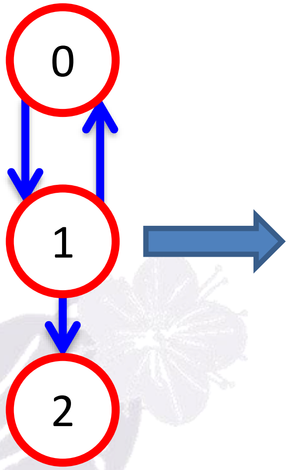


Graph with two connected components

- **Tree:**
  - A connected acyclic graph.

Terminology

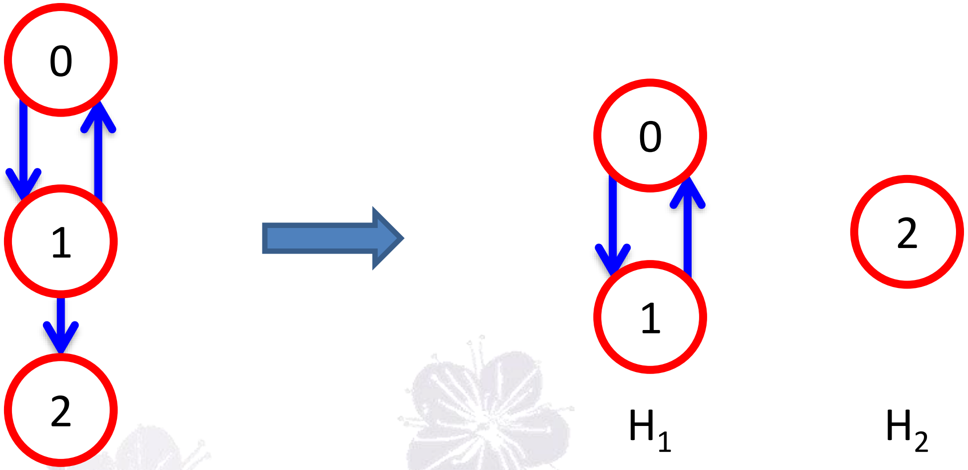
- Directed graph G is said to be **strongly connected** iff for **every pair of distinct vertices u and v**, there is a **directed path from u to v** and **also from v to u** in G.



Not a strongly connected digraph!
There is no directed path from 2 to 0

Terminology

- A **strongly connected component** is a maximal subgraph that is strongly connected.



Two strongly connected components

Terminology

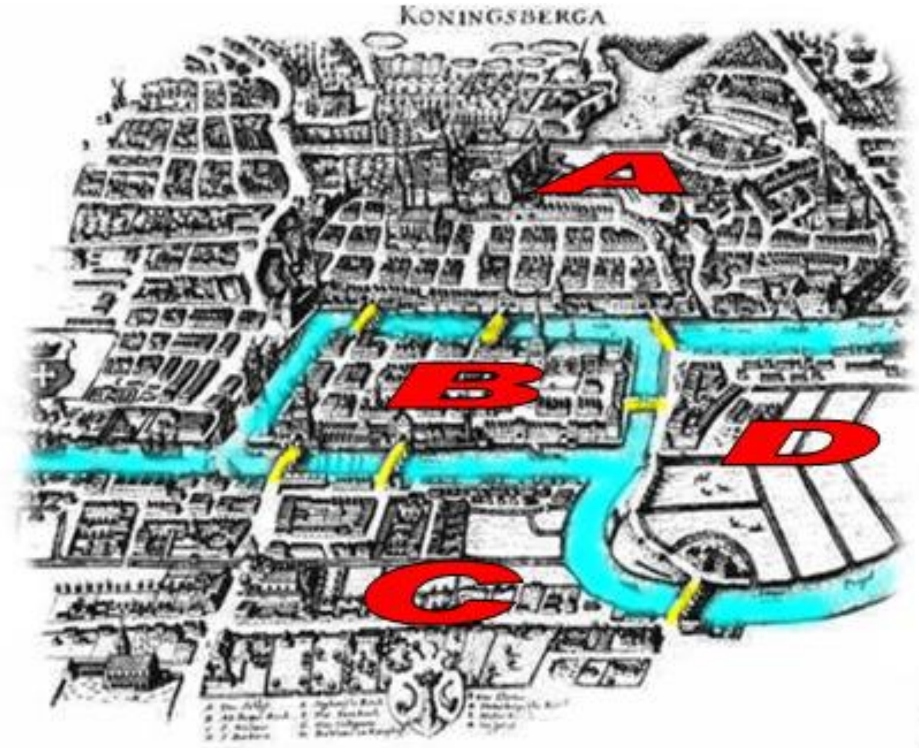
- **Degree** of a vertex v:
  - The # of edges incident to v.
- In a directed graph:
  - **In-degree** of v
    - The # of edges for which v is the head.
  - **Out-degree** of v
    - The # of edges for which v is the tail.
  - **Degree of v = in-degree + out-degree**

Self-Study Topics – some variations to Konigsberg Bridge Problem

- 4 lands
- 7 bridges
- Problem:

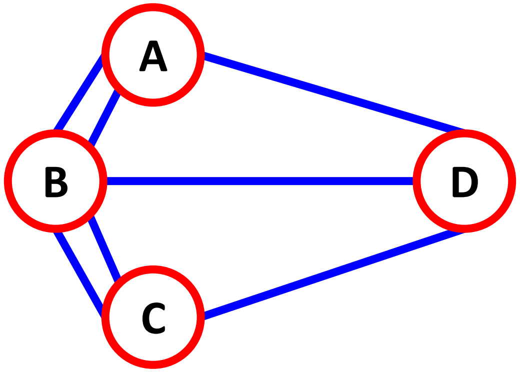
Original Problem:

Starting at one land, is it possible to **walk across all the bridges exactly once** and **returning to the starting land**?



Konigsberg Bridge Problem

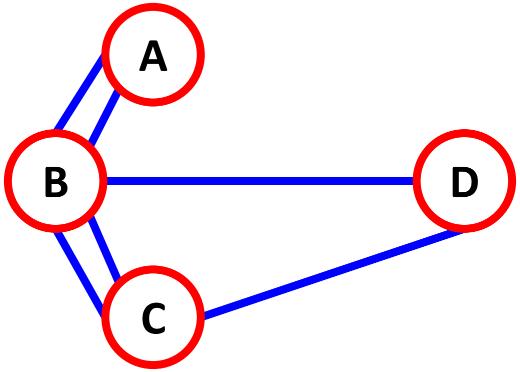
- Euler formulate the problem as a graph.



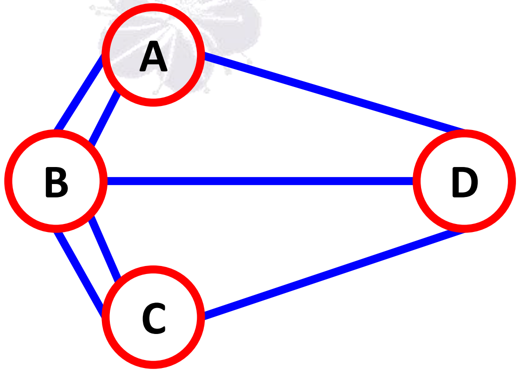
- Prove that the answer to the problem is possible iff **the degree of each vertex is even.**

What if the degrees are not all even? (Self-Study)

- Only a pair of vertices with odd degree



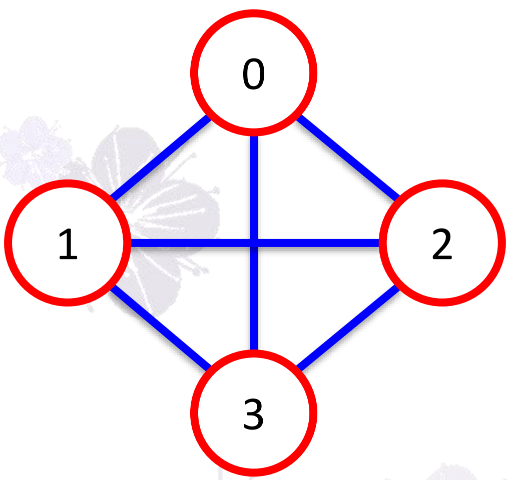
- Multiple pairs of odd-degree vertices
  Chinese Postman Problem (中國郵差問題)



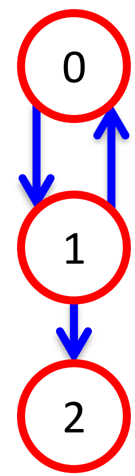
GRAPH REPRESENTATION

Adjacency Matrix

- A two dimensional array with the property that a[i][j] = 1 iff the edge (i,j) or <i,j> is in E(G).



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

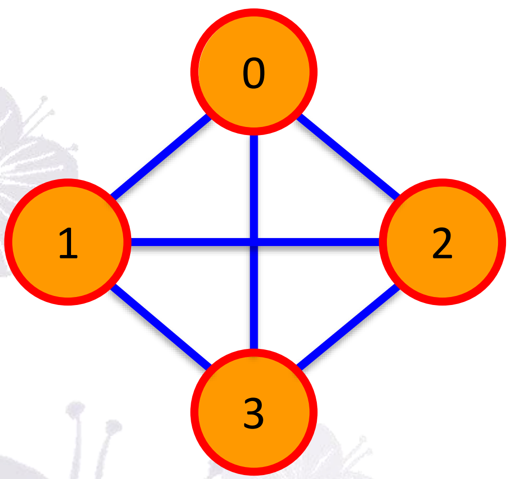


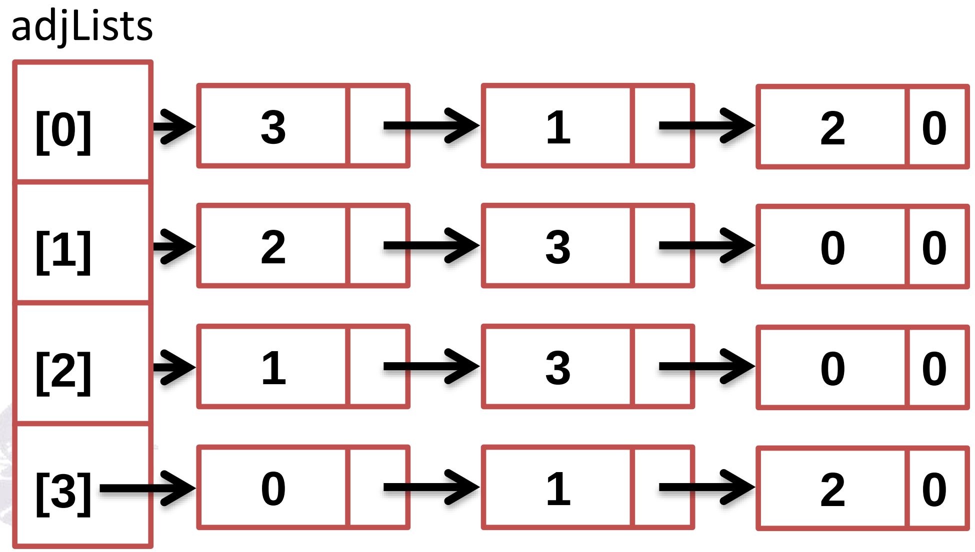
$$\begin{array}{c c} & \begin{array}{ccc} 0 & 1 & 2 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \end{array} & \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \end{array}$$

- Waste of memory when a graph is sparse
  - Storage $O(n^2)$

Adjacency Lists

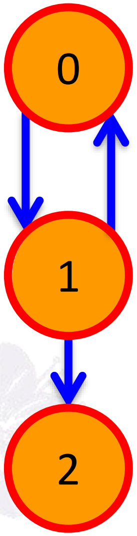
- Undirected graph: Use a chain to represent each vertex and its **adjacent** vertices.

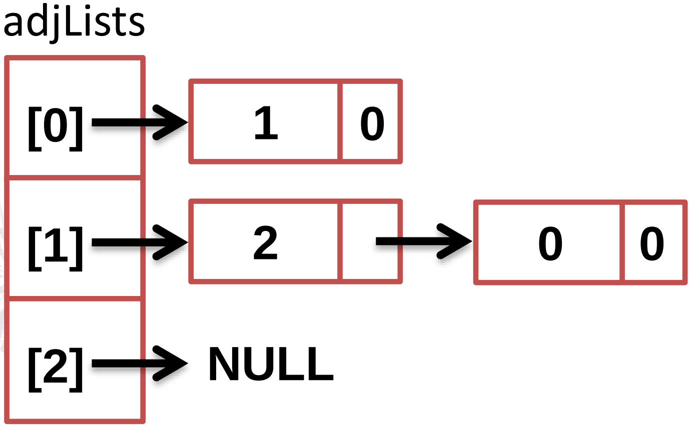




Adjacency Lists

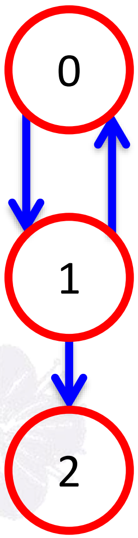
- Directed graph: Use a chain to represent each vertex and its **adjacent to** vertices.
  – Length of list = Out-degree of v

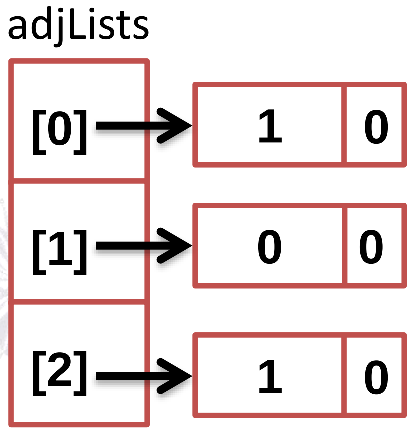




Inverse Adjacency Lists

- Directed graph: Use a chain to represent each vertex and its **adjacent from** vertices
  - Length of list = In-degree of v





Weighted Edges

- Edges of a graph sometimes have weights associated with them.
  - Distance from one vertex to another.
  - Cost of going from one vertex to an adjacent vertex.
- We use additional field in each edge to store the weight.
- A graph with weighted edges is called a **network**.

ADT: Graph

```
class Graph
{// object: A nonempty set of vertices and a set of undirected edges.
public:
  virtual ~Graph() {}                     // virtual destructor
  bool IsEmpty() const{return n == 0};    // return true iff graph has no vertices
  int NumberOfVertices() const{return n}; // return the # of vertices
  int NumberOfEdges() const{return e};    // return the # of edges
  virtual int Degree(int u) const = 0;    // return the degree of a vertex
  virtual bool ExistsEdge(int u, int v) const = 0; // check the existence of edge
  virtual void InsertVertex(int v) = 0;            // insert a vertex v
  virtual void InsertEdge(int u, int v) = 0;       // insert an edge (u, v)
  virtual void DeleteVertex(int v) = 0;            // delete a vertex v
  virtual void DeleteEdge(int u, int v) = 0;       // delete an edge (u, v)
  // More graph operations…
protected:
  int n; // number of vertices
  int e; // number of edges
};
```

Implementation Notes

- To accommodate various graph types, we make the following assumptions:
- Data type of edge weight is **double** (or represented as a template parameter).
- We define operations which are **independent** of specific graph representation in the Graph.
- We assume the **iterator** is used to visit adjacent vertices.

Example: LinkedGraph

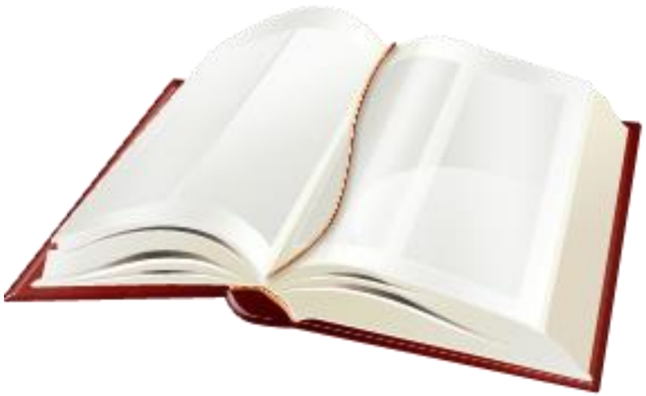
```
void Graph::foo(void){
  // use iterator to visit adjacent vertices of v
  for (each vertex w adjacent to v)…
}
```

```
class LinkedGraph : public Graph
{
public:
  // constructor
  LinkedGraph(const int vertices = 0) : n(vertices), e(0){
     adjLists = new Chain<int>[n];
  }
  // more customized operations…
private:
  Chain<int> *adjLists    // adjacency lists
};
```

ELEMENTARY GRAPH OPERATIONS

Graph Operations

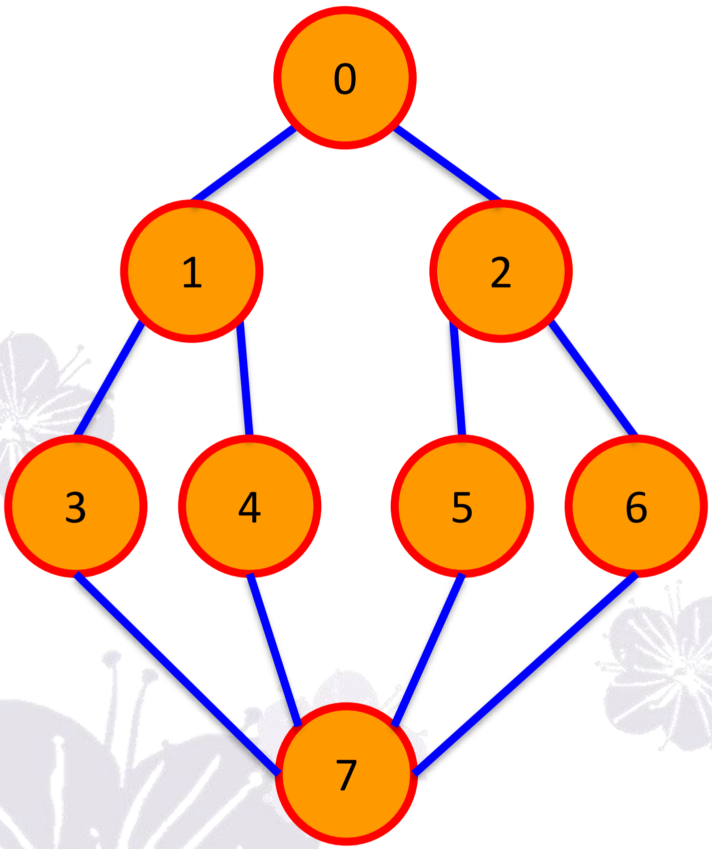
- Graph traversal
  - Depth-first search
  - Breadth-first search
- Connected components
- Spanning trees

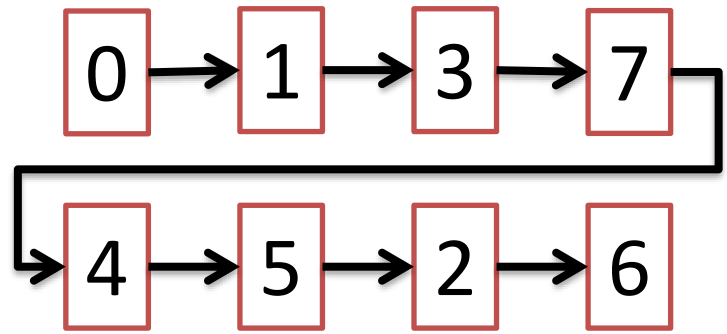


Depth-First Search (DFS)

- Starting from a vertex v
  - Visit the vertex v => DFS(v).
  - For each vertex w adjacent to v, if w is not visited yet, then visit w => DFS(w).
  - If a vertex u is reached such that all its adjacent vertices have been visited, we go back to the last visited vertex.
- The search terminates when no unvisited vertex can be reached from any of the visited vertices.

Depth-First Search (DFS)





Note that the output order is not unique, there are other possibilities, depending on the graph representation

Recursive DFS

```
void Graph::DFS(void){
   visited = new bool[n]; // this is a data member of Graph
   fill(visited, visited+n, false);
   DFS(0); // start search at vertex 0
   delete [] visited;
}

void Graph::DFS(const int v){
   // visit all previously unvisited vertices that are adjacent to v
   output(v);
   visited[v]=true;
   for(each vertex w adjacent to v)
     if(!visited[w]) DFS(w);
}
```

Non-Recursive DFS

```
void Graph::DFS(int v){
   visited = new bool[n]; // this is a data member of Graph
   fill(visited, visited+n, false);
   Stack<int> s;          // declare and init a stack
   s.Push(v);
   while(!s.IsEmpty()){
      v = s.Top(); s.Pop();
      if(!visited[v]){
         output(v);
         visited[v]=true;
         for(each vertex w adjacent to v)
           if(!visited[w]) s.Push(w);
      }
   }
}
```

DFS Complexity

- Adjacency matrix
  - Time to determine all adjacent vertices: O(n)
  - At most n vertices are visited: $O(n \cdot n) = O(n^2)$
- Adjacency lists
  - There are n+2e chain nodes
  - Each node in the adjacency lists is examined at most once. Time complexity = O(e)

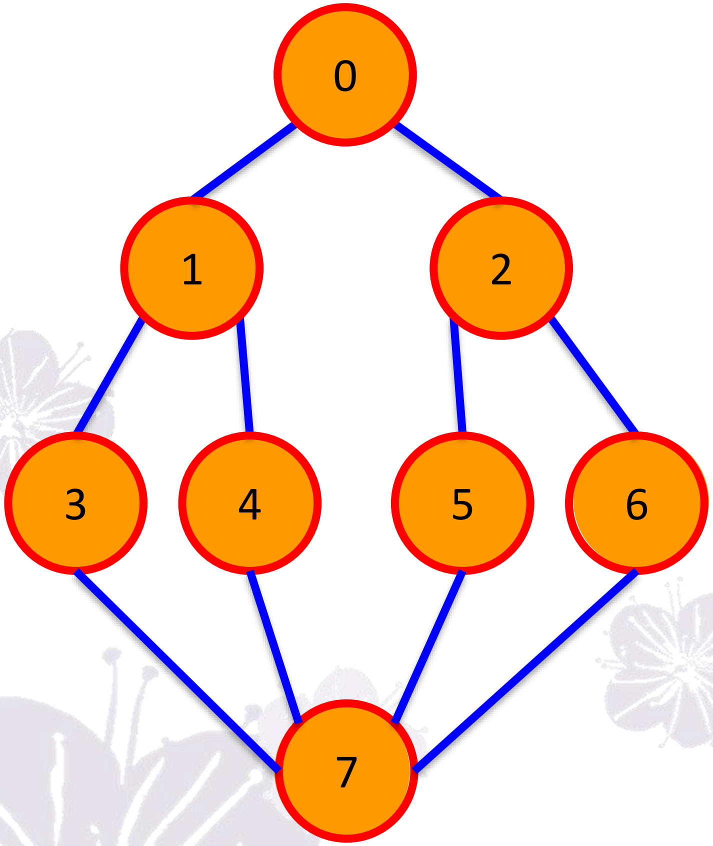
Trees and DFS

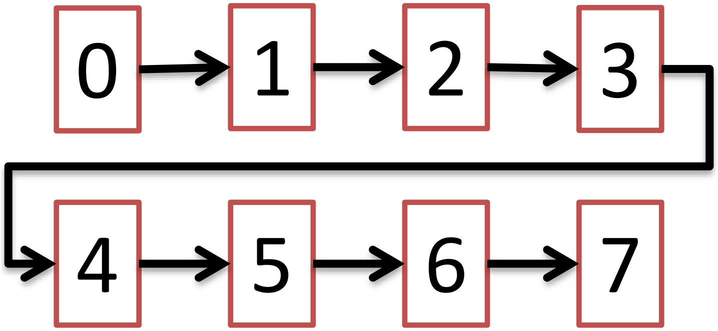
- Inorder, preorder and postorder traversal can be viewed as specialized DFS
  - They differ from each other on when the key is printed
- If tree is represented with linked nodes
  - The time complexity of in-, pre-, post-orders are O(n)
  - Since in trees, |E|=n-1
  - The same time complexity as DFS

Breadth-First Search (BFS)

- Starting from a vertex v
  - Visit the vertex v.
  - Visit all unvisited vertices adjacent to v.
  - Unvisited vertices adjacent to these newly visited vertices are then visited and so on…

Breadth-First Search (BFS)





Note that the output order is not unique, there are other possibilities, depending on the graph representation

BFS: Implementation

```
void Graph::BFS(int v){
   visited = new bool[n]; // this is a data member of Graph
   fill(visited, visited+n, false);
   Queue<int> q;          // declare and init a queue
   q.Push(v);
   visited[v]=true;
   while(!q.IsEmpty()){
      v = q.Front(); q.Pop();
      output(v);
      for(each vertex w adjacent to v){
         if(!visited[w]){
           q.Push(w);
           visited[w]=true;
         }
      }
   }
   delete [] visited;
}
```

Time complexity is the same as DFS

Trees and BFS

- BFS looks familiar, right?

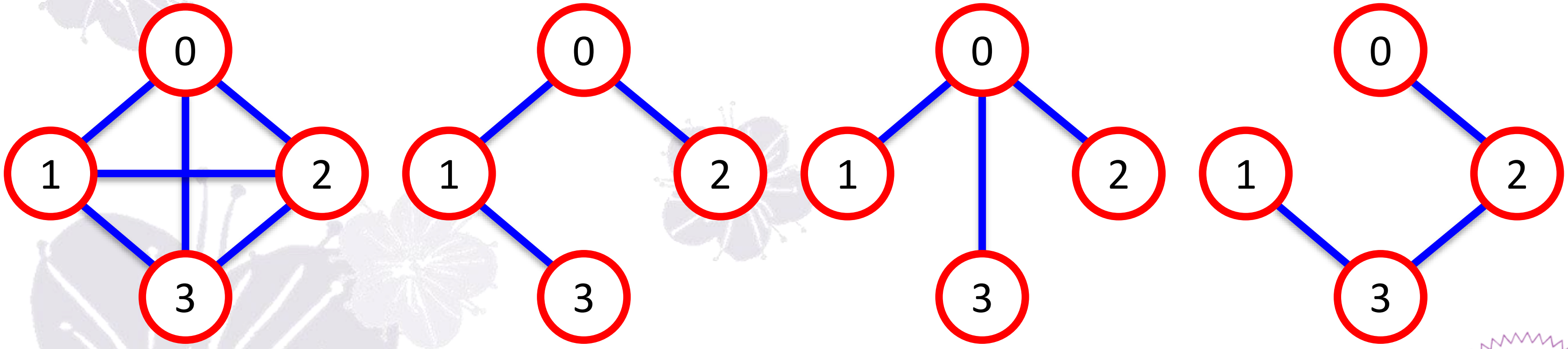
- Level-order traversal is a specialized BFS
  - Visiting order from left to right

Connected Components

- How to determine whether a graph is connected or not?
  - Call DFS or BFS once and check if there is any unvisited vertices, if Yes, then the graph is not connected.
- How to identify connected components
  - Call DFS or BFS repeatly.
  - Each call will output a connected component.
  - Start next call at an unvisited vertex.

Spanning Trees

- Definition: Any tree consists of solely of edges in E(G) and including all vertices of V(G).
- Number of tree edges is **n-1**.
- Add a non-tree edge will create a **cycle.**

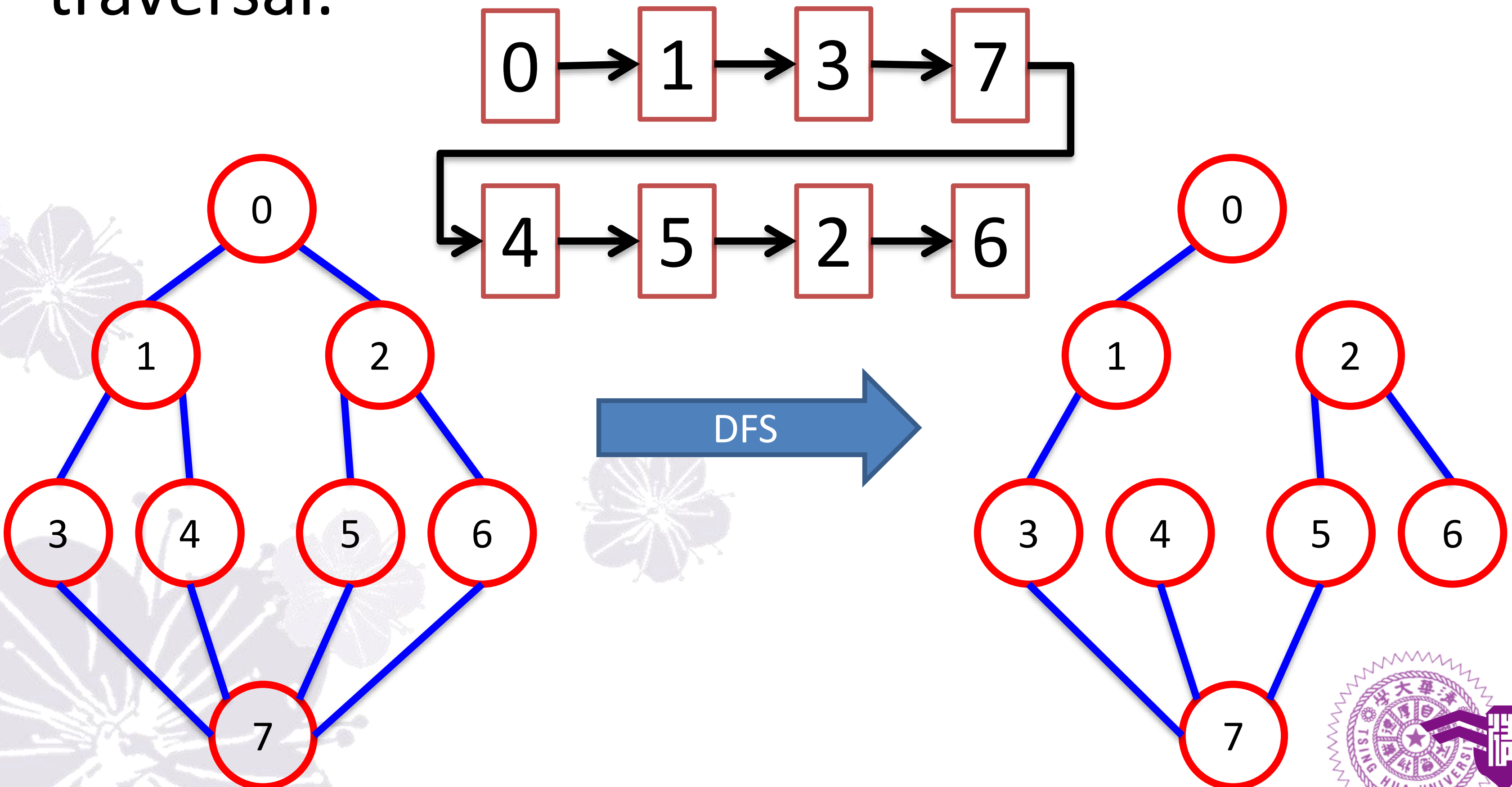


Complete graph

Possible spanning trees

DFS Spanning Tree

- Tree edges are those edges met during the traversal.



BFS Spanning Tree

- Tree edges are those edges met during the traversal.