

# Graphs

## Lecture 12

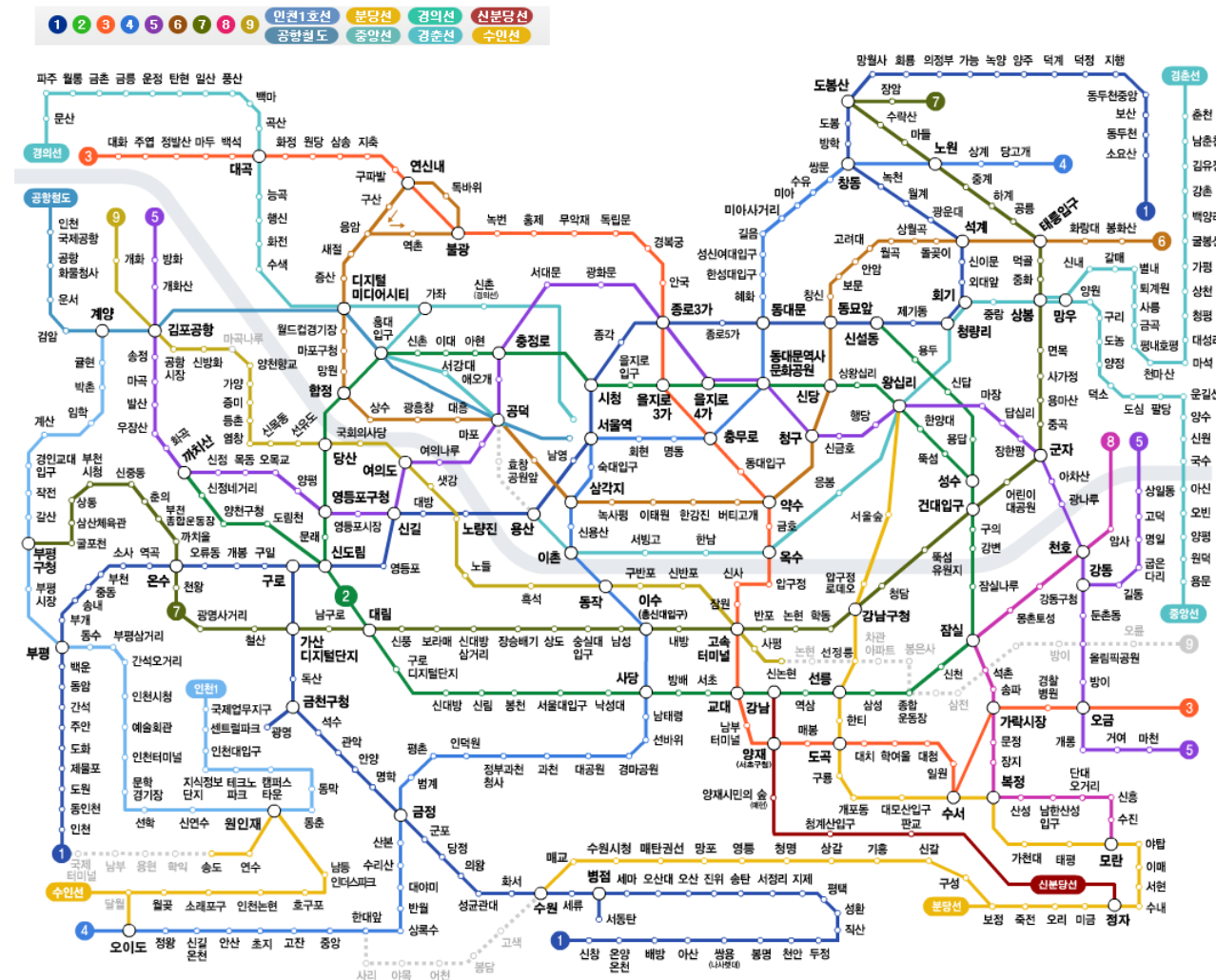
Hyung-Sin Kim



SNU Graduate School of Data Science

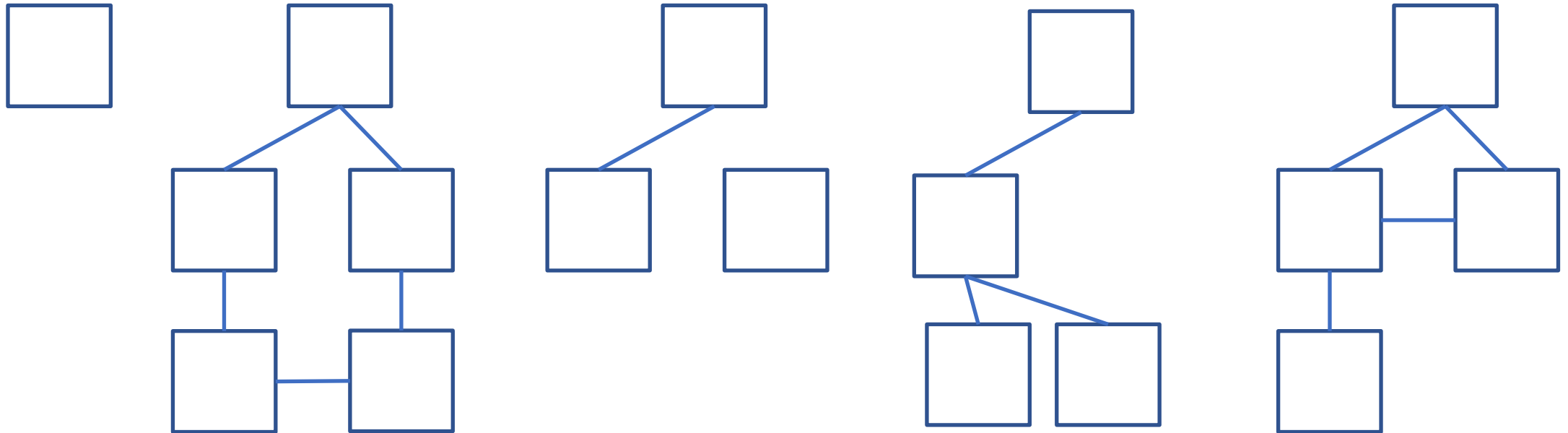
# Trees are Great but...

- Trees are good for representing data with hierarchy
- Is Seoul Metro map hierarchical?
- When you want to travel from station A to station B, is there usually only one path?



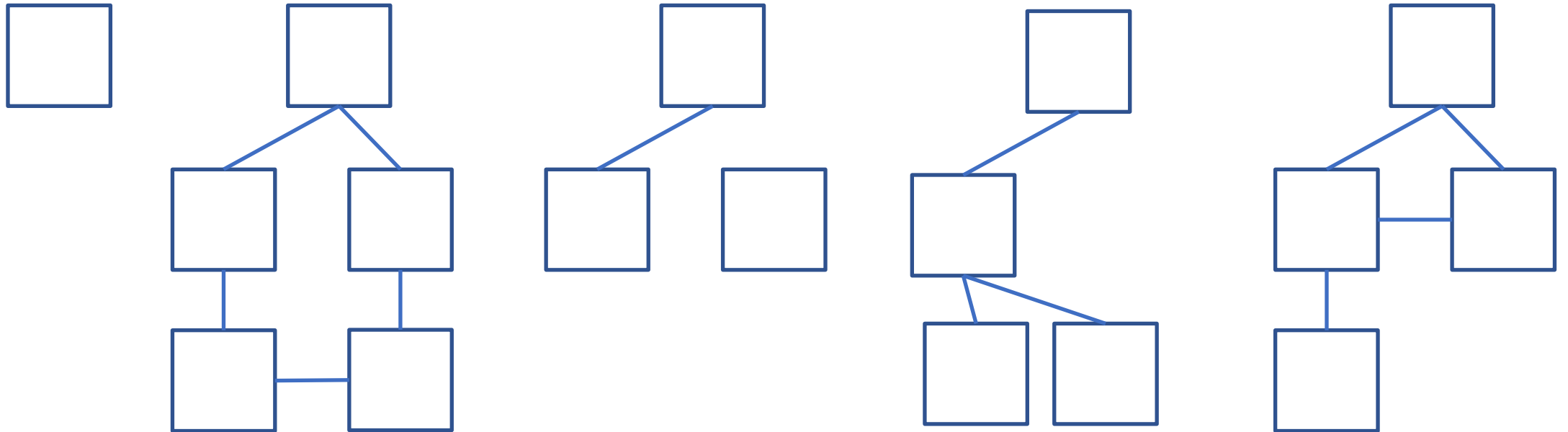
# Trees

- A tree comprises a set of **nodes** that are **connected (linked)** to each other
- There is **only one path** between two nodes in a tree
- Choose trees!



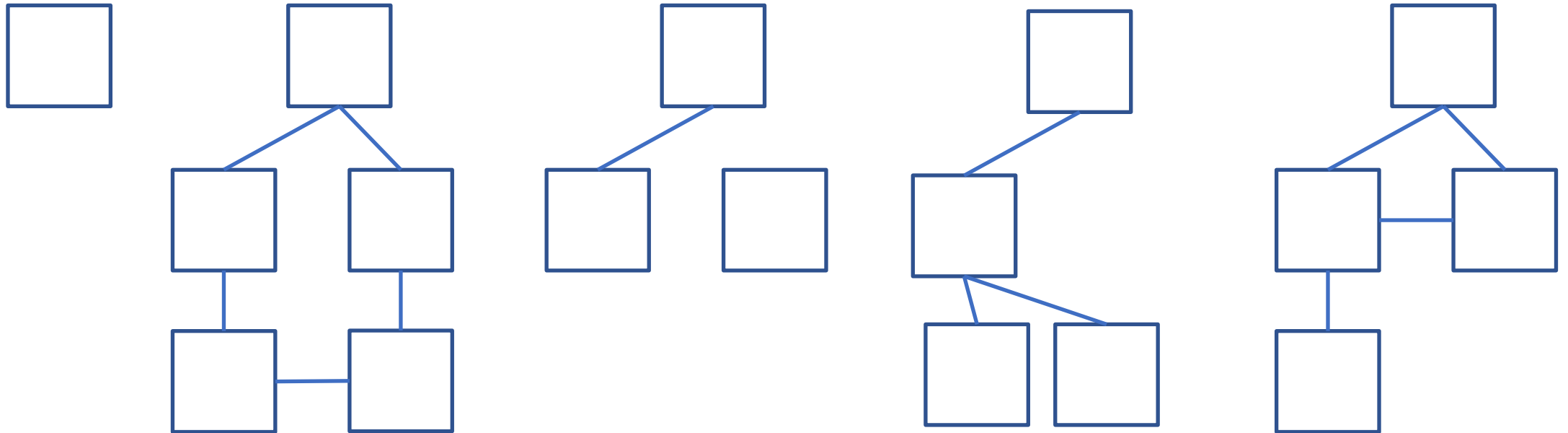
# Graphs

- A graph comprises a set of **nodes** and a set of **edges** (including zero edge), each of which connects two nodes
- Choose graphs!



# Graphs

- A graph comprises a set of **nodes** and a set of **edges** (including zero edge), each of which connects two nodes
- Choose graphs!
  - Yes, graph is more general than tree (all trees are graphs)

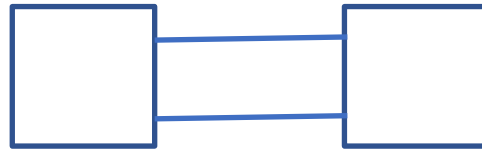


# Simple Graphs

- A **simple** graph is a graph that does not have...
  - **Loop**: an edge that connects a node to itself node, such as  $(v,v)$



- **Parallel edge**: two edges that connects the same nodes, such as  $e1 = (v,w)$  and  $e2 = (v,w)$



- We only focus on simple graphs today 😊

# Terms and Terms ...

- **Vertex:** a node
  - $v$
- **Edge:** a pair of vertices
  - $(v,w)$  where  $v$  and  $w$  are vertices
- Two vertices which have an edge between are **adjacent**
  - If a graph has an edge  $(v,w)$ , vertices  $v$  and  $w$  are adjacent
- A **path** is a sequence of vertices connected by edges
  - If there are three edges in a graph,  $(v,w)$ ,  $(w,y)$ , and  $(y,z)$ , the graph has a path  $(v,w,y,z)$

# Terms and Terms ...

- A path with unique (non-repeated) vertices is a **simple** path
  - $(v,w,y,v,z)$  is not simple, but  $(v,w,y,z)$  is simple
- A **cycle** is a path where the first and last vertices are the same
  - A path  $(v,w,y,v)$  is a cycle. Another path  $(v,w,y,z)$  is not a cycle
  - A graph having a cycle is a **cyclic** graph
  - A graph without a cycle is an **acyclic** graph
- If there is a path between two vertices, the two vertices are **connected**
  - If there is a path  $(v,w,y,z)$ , vertices  $v$  and  $z$  are connected

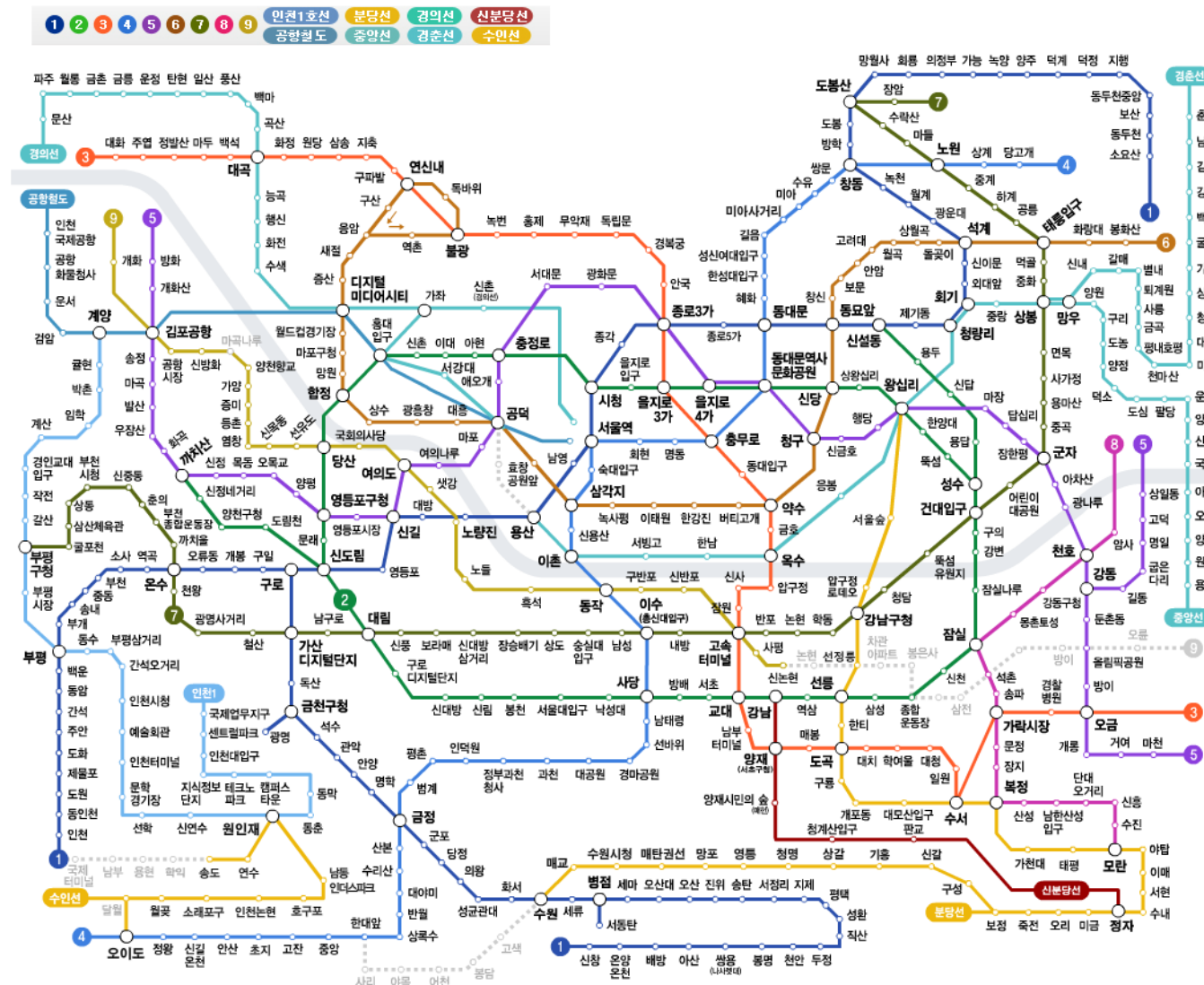


# Terms and Terms ...

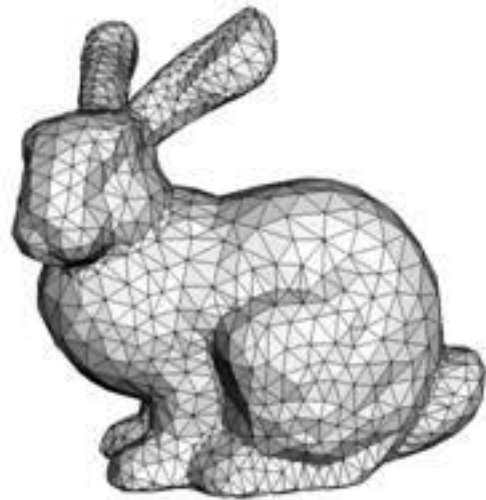
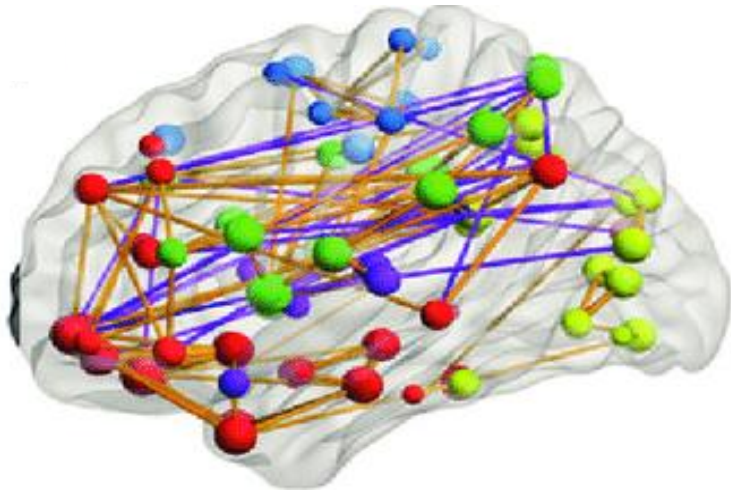
- An edge is either **undirected** (like a line) or **directed** (like an arrow)
  - If an edge  $(v,w)$  is undirected,  $v$  and  $w$  can reach each other
  - If an edge  $(v,w)$  is directed,  $v$  can go to  $w$ , not vice versa
    - We need another edge  $(w,v)$  so that  $w$  can go to  $v$
- A graph with undirected edges is an **undirected graph**
- A graph with directed edges is a **directed graph**

# So... What is this?

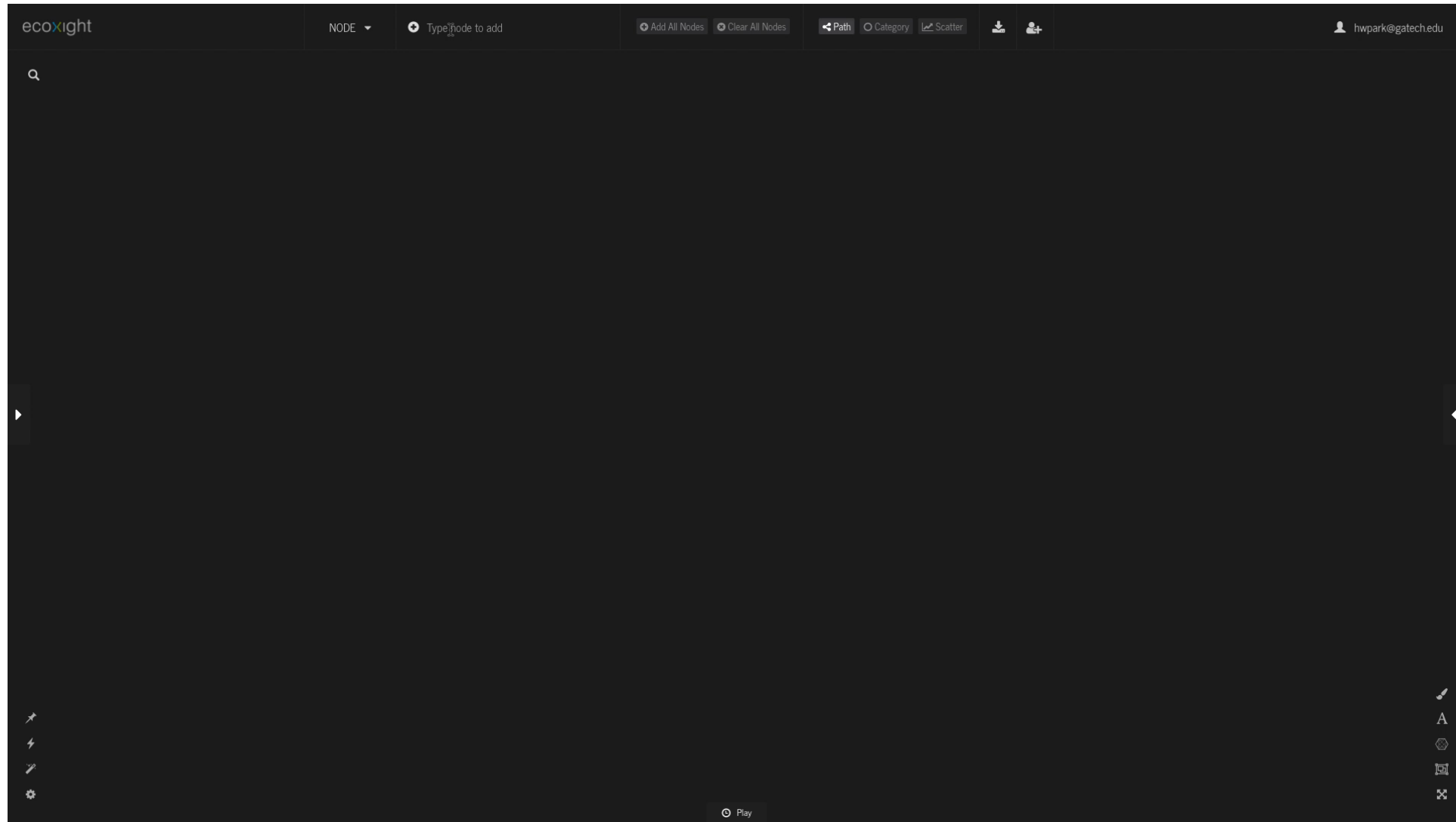
- Tree or graph?
- Connected or not?
- Cyclic or acyclic?
- Undirected or directed?



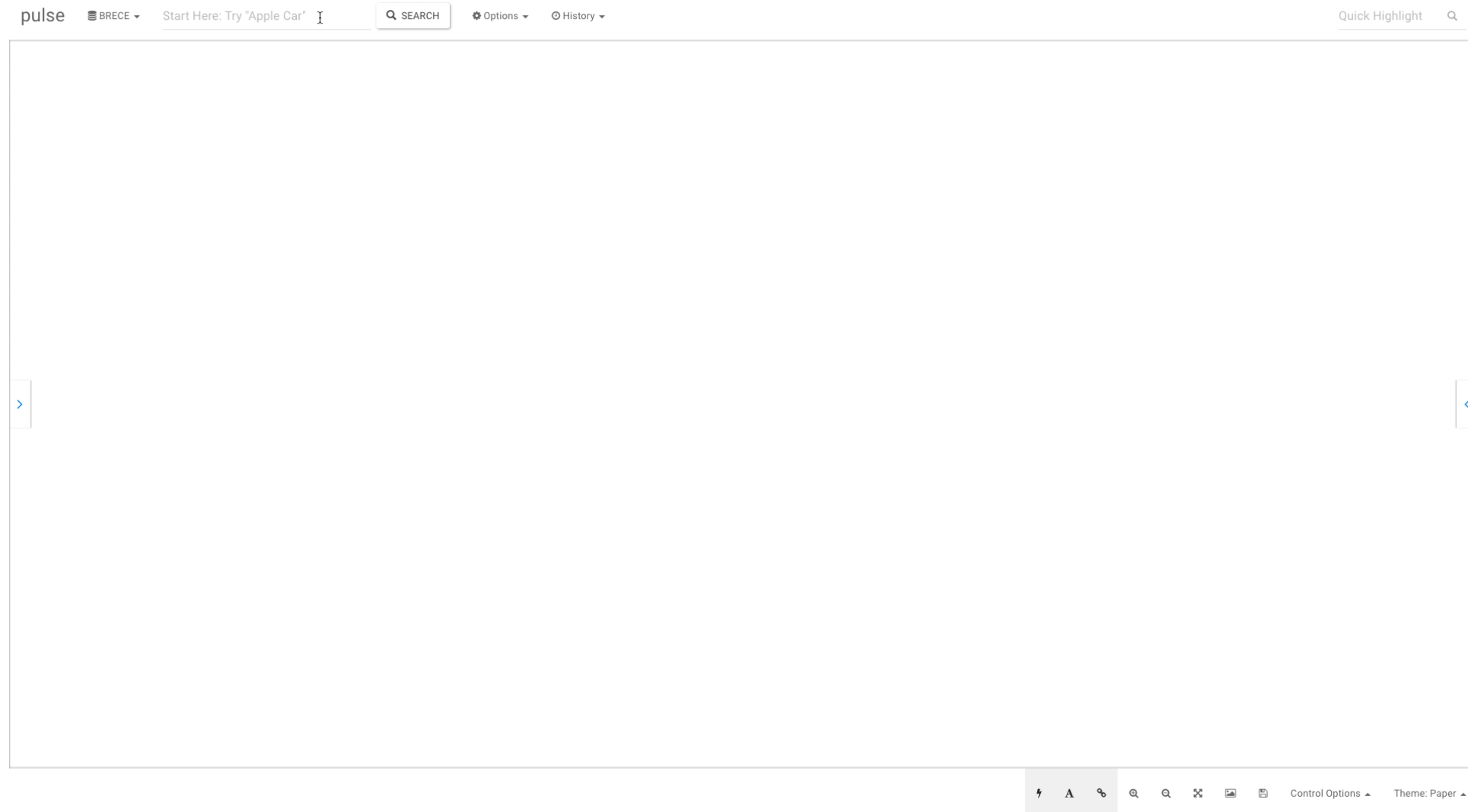
# Graphs in Practice



# Graph Visualization (Prof. Hyunwoo Park)



# Graph Visualization (Prof. Hyunwoo Park)

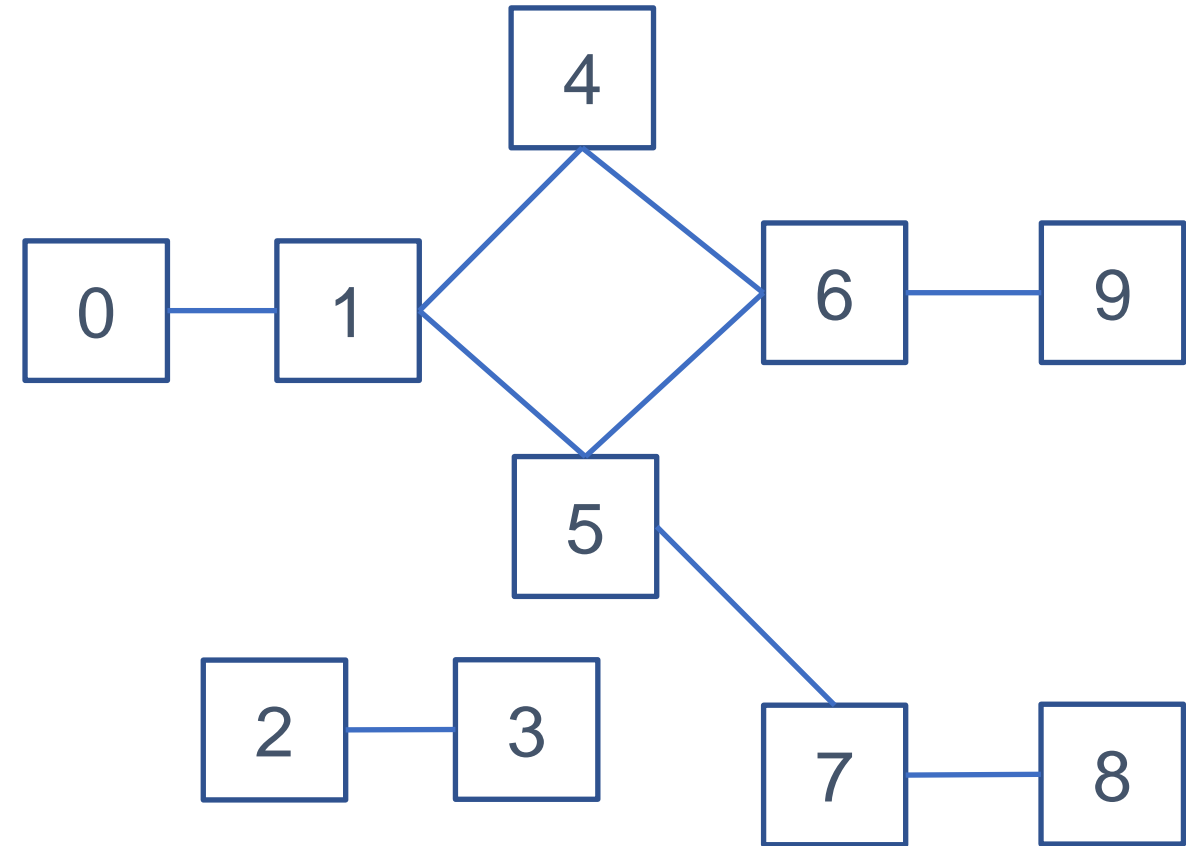


# Graph Construction

- Given vertices  $V$  and edges  $E$ , build relationship between the vertices
  - `class undi_graph():`
  - `def __init__(self, V: list, E: list) -> None:`
  - `self.V = V[:]`
  - `self.neighbor = {}`
  - `for v in V:`
  - `self.neighbor[v] = []`
  - `for (v, w) in E:`
  - `self.neighbor[v].append(w)`
  - `self.neighbor[w].append(v)`

# Graph Traversals

- Graphs have **cycles** and can be **disconnected**
- We still should visit **all nodes**



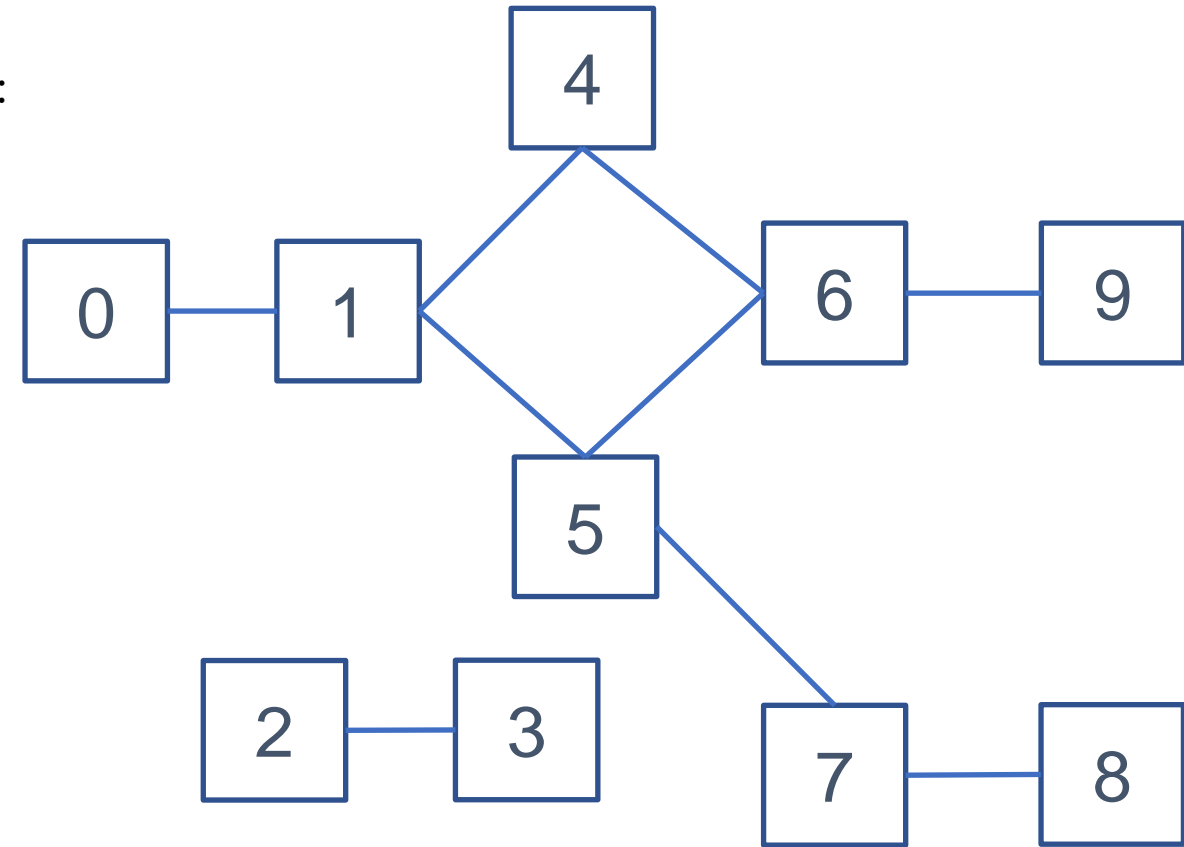
Good news!

We can still use depth-first traversals and bread-first traversals,  
but a little bit differently 😊



# Depth-First Traversal – Preorder

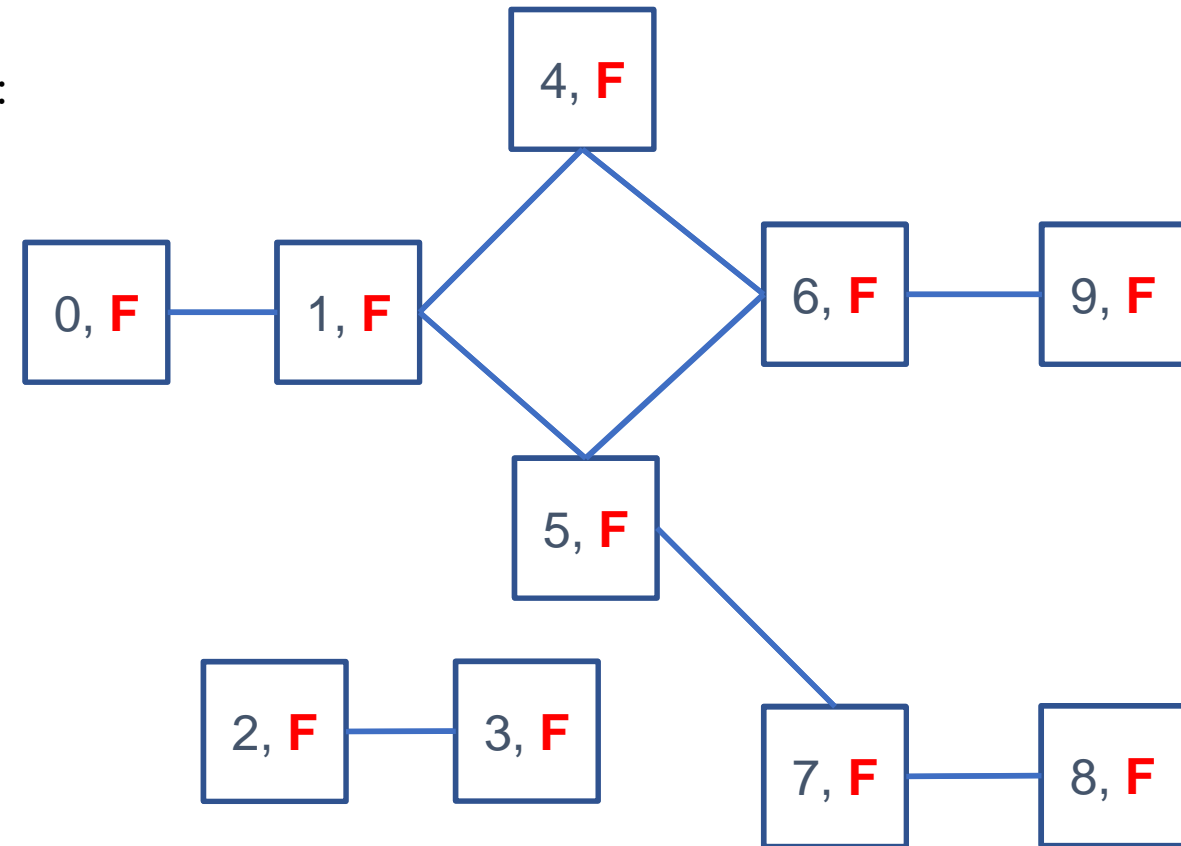
```
• class undi_graph():
•     def __DFTHelp(self, visited: list, v: int) -> None:
•         if not visited[v]:
•             visited[v] = True
•             print(v)
•             for w in self.neighbor[v]:
•                 self.__DFTHelp(visited, w)
•
•     def DFT(self) -> None:
•         if self.V:
•             visited = {}
•             for v in self.V:
•                 visited[v] = False
•                 for v in self.V:
•                     self.__DFTHelp(visited, v)
```



1. Now we should **mark** the nodes we already visited to handle **cycles**
2. Now one \_\_DFTHelp call is **not enough** to traversal a **disconnected** graph

# Depth-First Traversal – Preorder

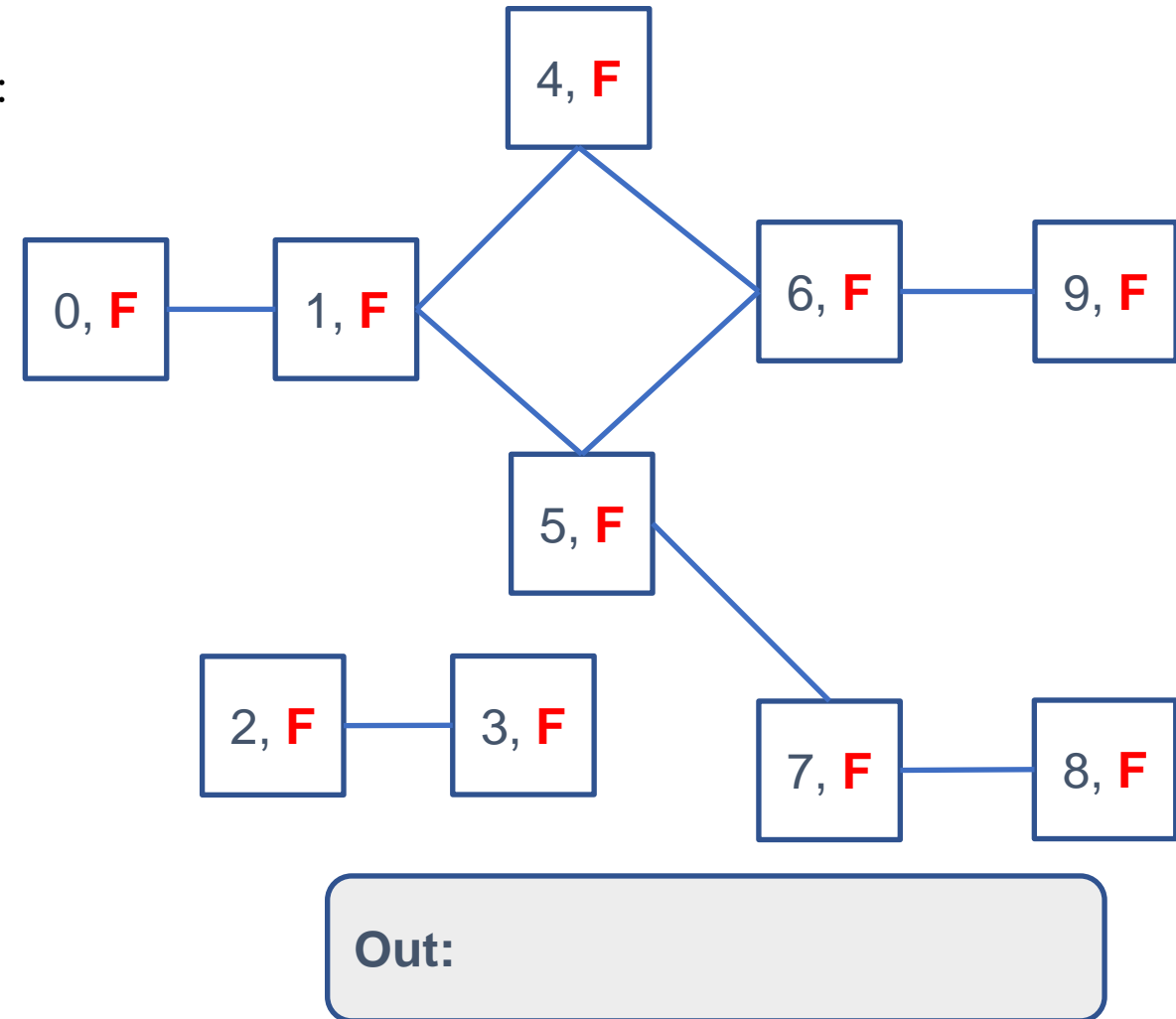
```
class undi_graph():  
    def __DFTHelp(self, visited: list, v: int) -> None:  
        if not visited[v]:  
            visited[v] = True  
            print(v)  
            for w in self.neighbor[v]:  
                self.__DFTHelp(visited, w)  
  
    def DFT(self) -> None:  
        if self.V:  
            visited = {}  
            for v in self.V:  
                visited[v] = False  
            for v in self.V:  
                self.__DFTHelp(visited, v)
```



1. Now we should **mark** the nodes we already visited to handle **cycles**
2. Now one \_\_DFTHelp call is **not enough** to traversal a **disconnected** graph

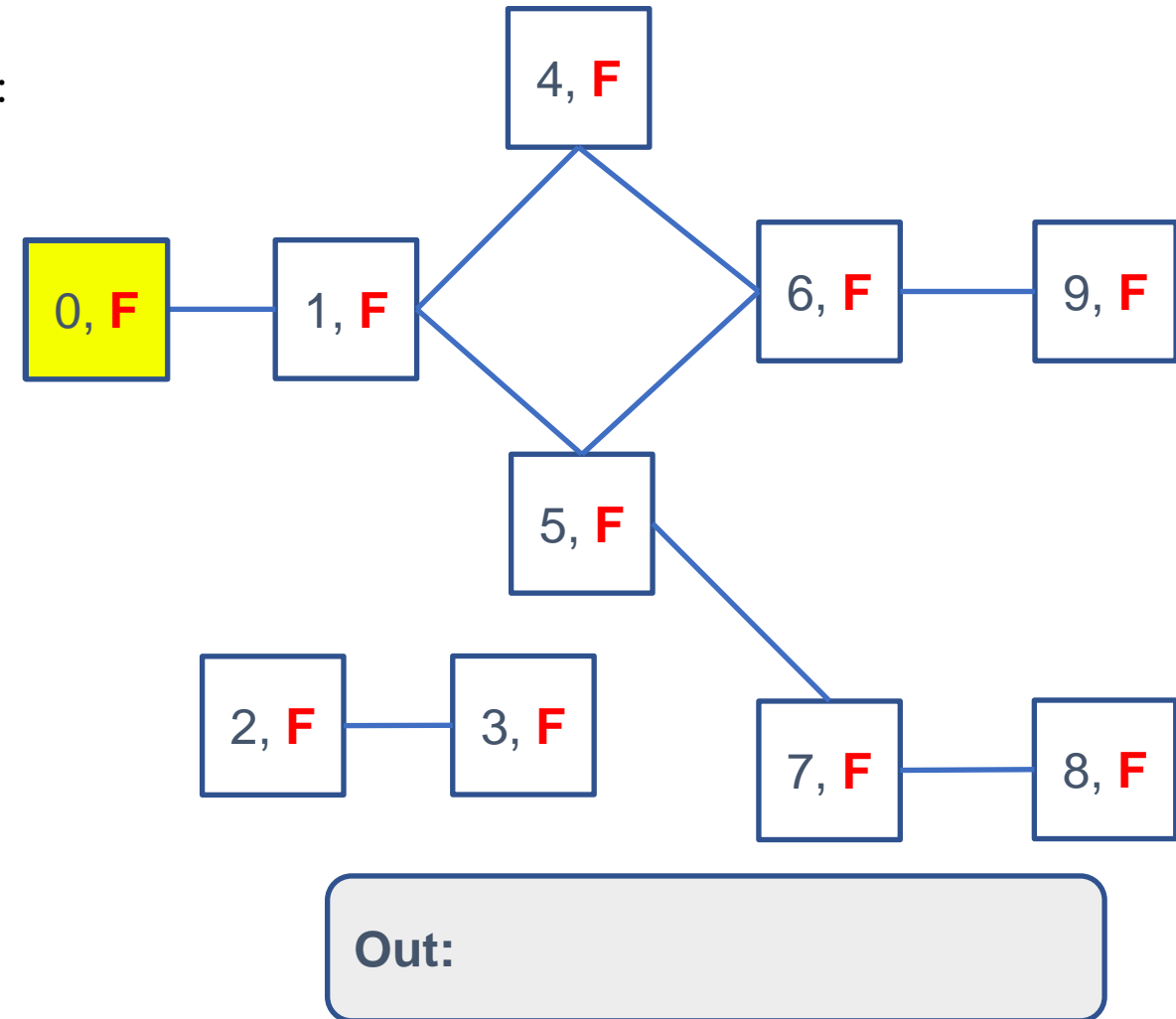
# Depth-First Traversal – Preorder

```
• class undi_graph():
•     def __DFTHelp(self, visited: list, v: int) -> None:
•         if not visited[v]:
•             visited[v] = True
•             print(v)
•             for w in self.neighbor[v]:
•                 self.__DFTHelp(visited, w)
•
•     def DFT(self) -> None:
•         if self.V:
•             visited = {}
•             for v in self.V:
•                 visited[v] = False
•             for v in self.V:
•                 self.__DFTHelp(visited, v)
```



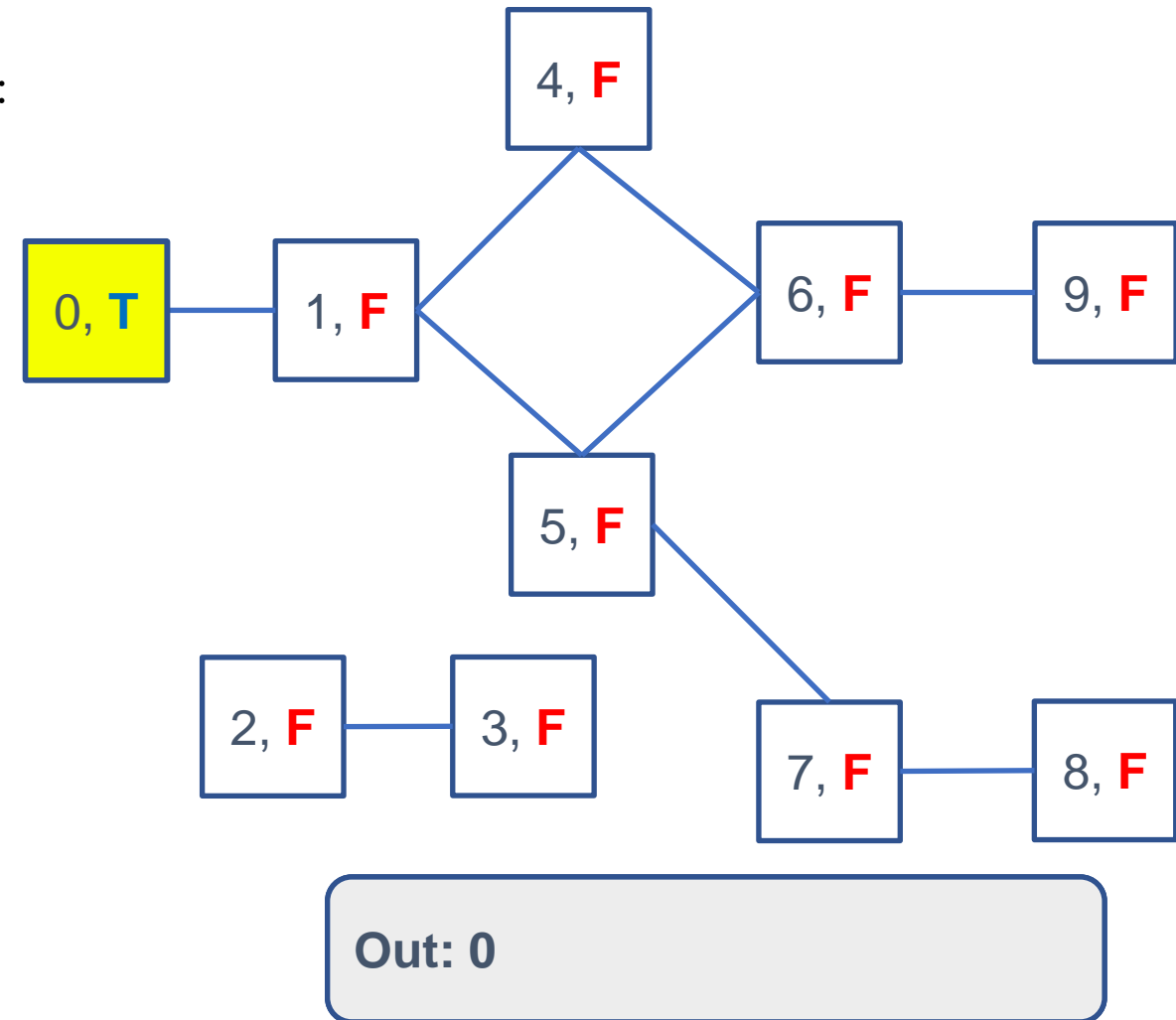
# Depth-First Traversal – Preorder

- `class undi_graph():`
- `def __DFTHelp(self, visited: list, v: int) -> None:`
- `if not visited[v]:`
- `visited[v] = True`
- `print(v)`
- `for w in self.neighbor[v]:`
- `self.__DFTHelp(visited, w)`
- `def DFT(self) -> None:`
- `if self.V:`
- `visited = {}`
- `for v in self.V:`
- `visited[v] = False`
- `for v in self.V:`
- `self.__DFTHelp(visited, v)`



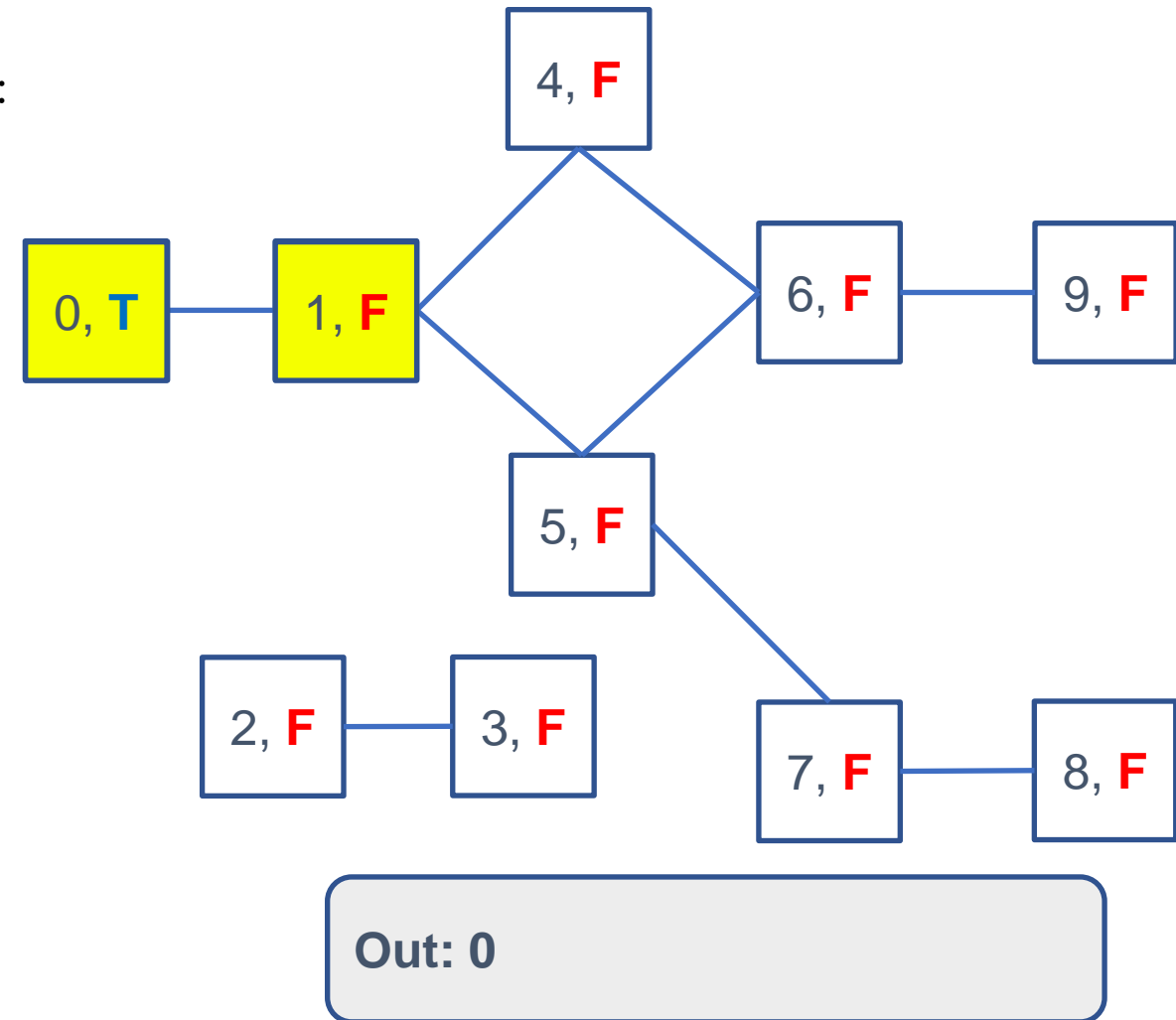
# Depth-First Traversal – Preorder

```
class undi_graph():  
    def __DFTHelp(self, visited: list, v: int) -> None:  
        if not visited[v]:  
            visited[v] = True  
            print(v)  
            for w in self.neighbor[v]:  
                self.__DFTHelp(visited, w)  
  
    def DFT(self) -> None:  
        if self.V:  
            visited = {}  
            for v in self.V:  
                visited[v] = False  
            for v in self.V:  
                self.__DFTHelp(visited, v)
```



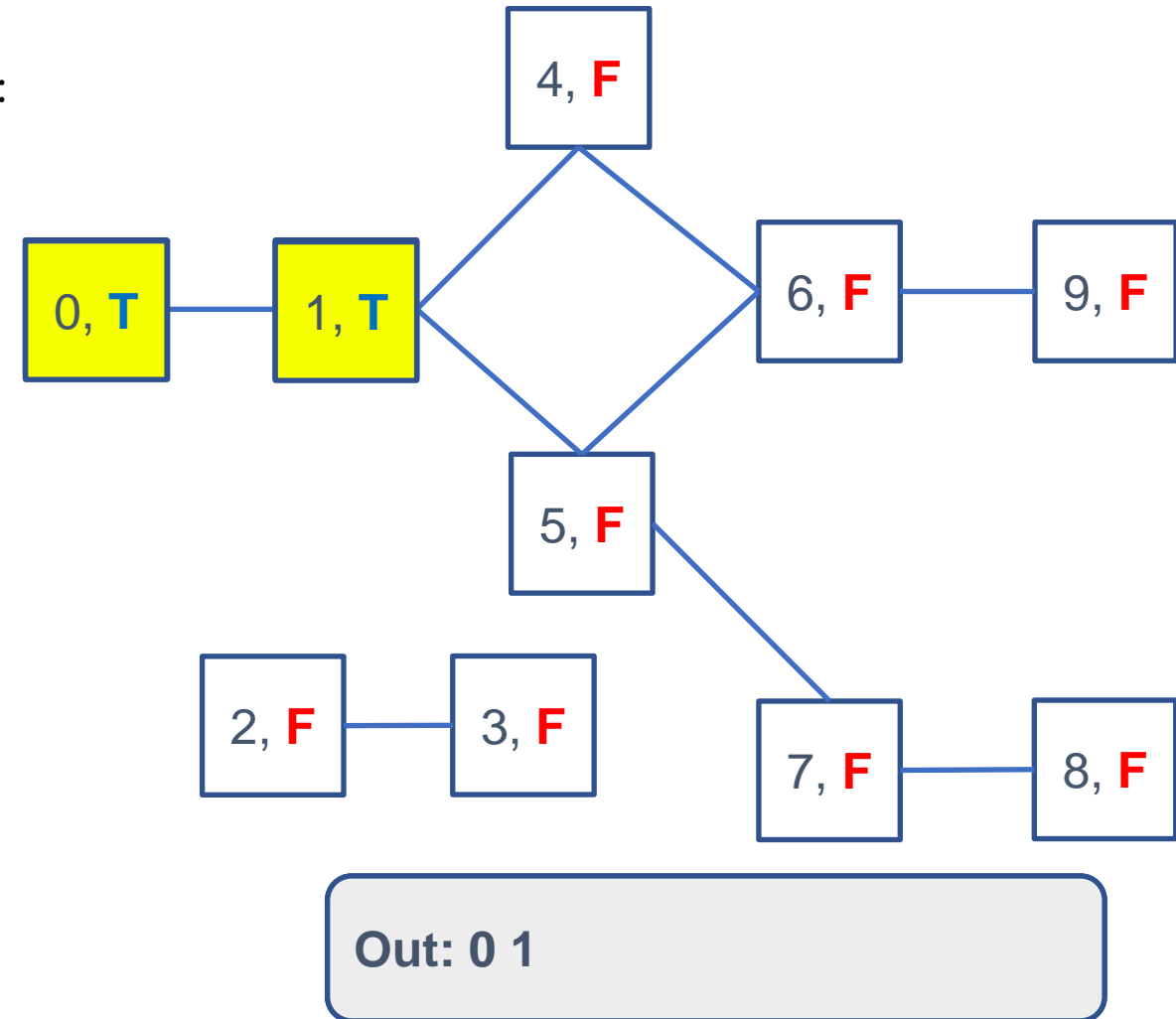
# Depth-First Traversal – Preorder

```
class undi_graph():  
    def __DFTHelp(self, visited: list, v: int) -> None:  
        if not visited[v]:  
            visited[v] = True  
            print(v)  
            for w in self.neighbor[v]:  
                self.__DFTHelp(visited, w)  
  
    def DFT(self) -> None:  
        if self.V:  
            visited = {}  
            for v in self.V:  
                visited[v] = False  
            for v in self.V:  
                self.__DFTHelp(visited, v)
```



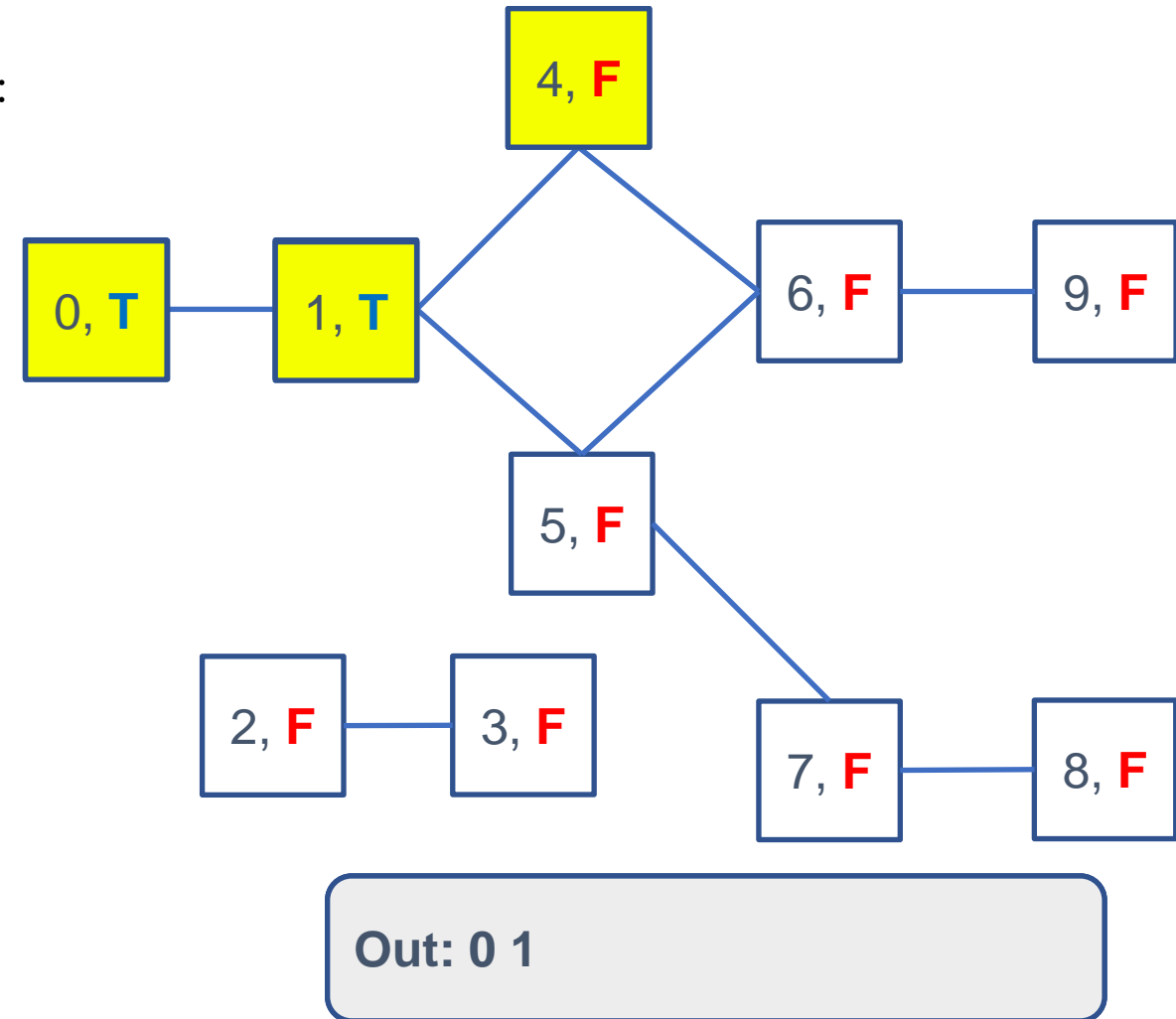
# Depth-First Traversal – Preorder

```
class undi_graph():  
    def __DFTHelp(self, visited: list, v: int) -> None:  
        if not visited[v]:  
            visited[v] = True  
            print(v)  
            for w in self.neighbor[v]:  
                self.__DFTHelp(visited, w)  
  
    def DFT(self) -> None:  
        if self.V:  
            visited = {}  
            for v in self.V:  
                visited[v] = False  
            for v in self.V:  
                self.__DFTHelp(visited, v)
```



# Depth-First Traversal – Preorder

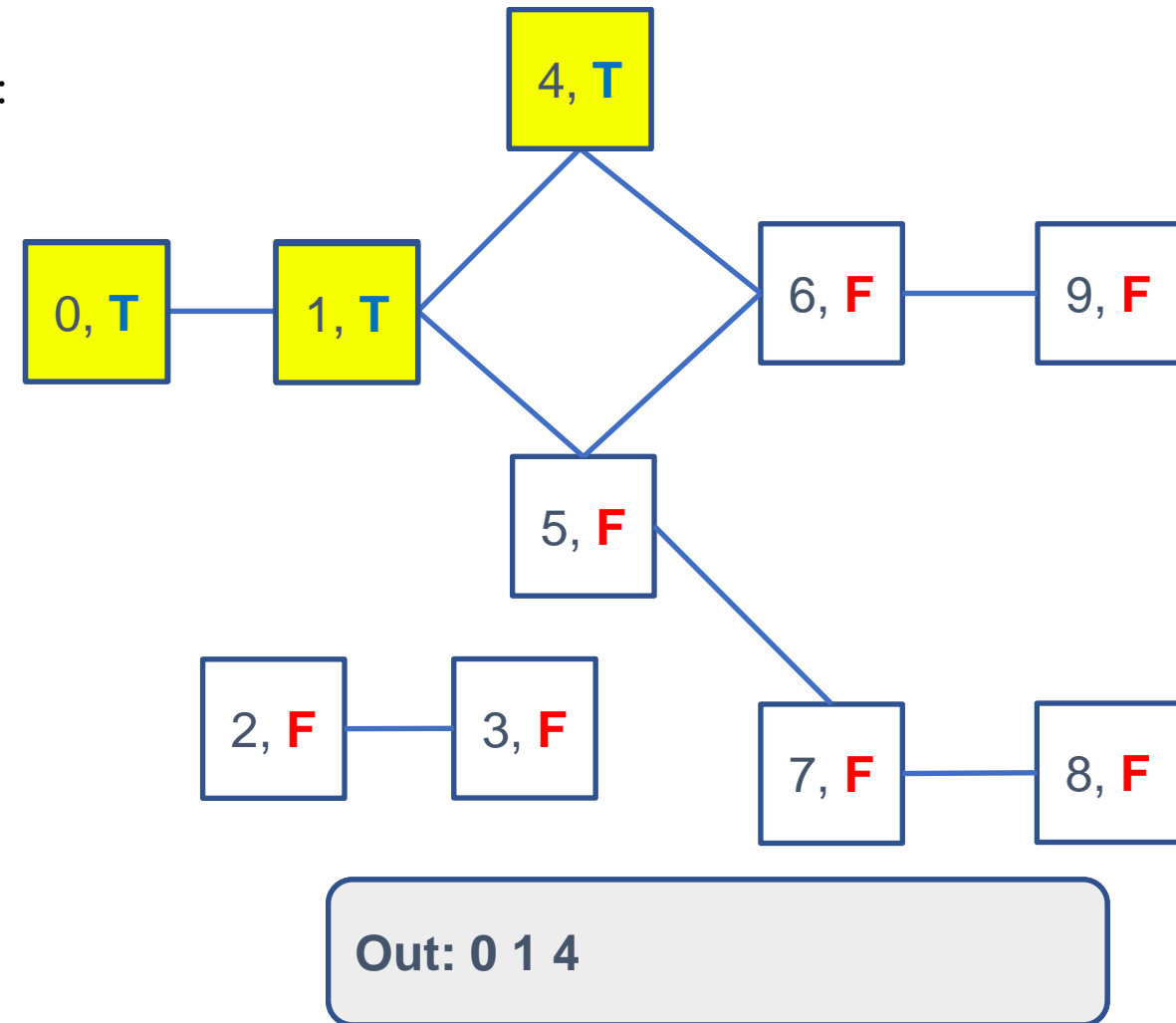
```
• class undi_graph():
•     def __DFTHelp(self, visited: list, v: int) -> None:
•         if not visited[v]:
•             visited[v] = True
•             print(v)
•             for w in self.neighbor[v]:
•                 self.__DFTHelp(visited, w)
•
•     def DFT(self) -> None:
•         if self.V:
•             visited = {}
•             for v in self.V:
•                 visited[v] = False
•             for v in self.V:
•                 self.__DFTHelp(visited, v)
```





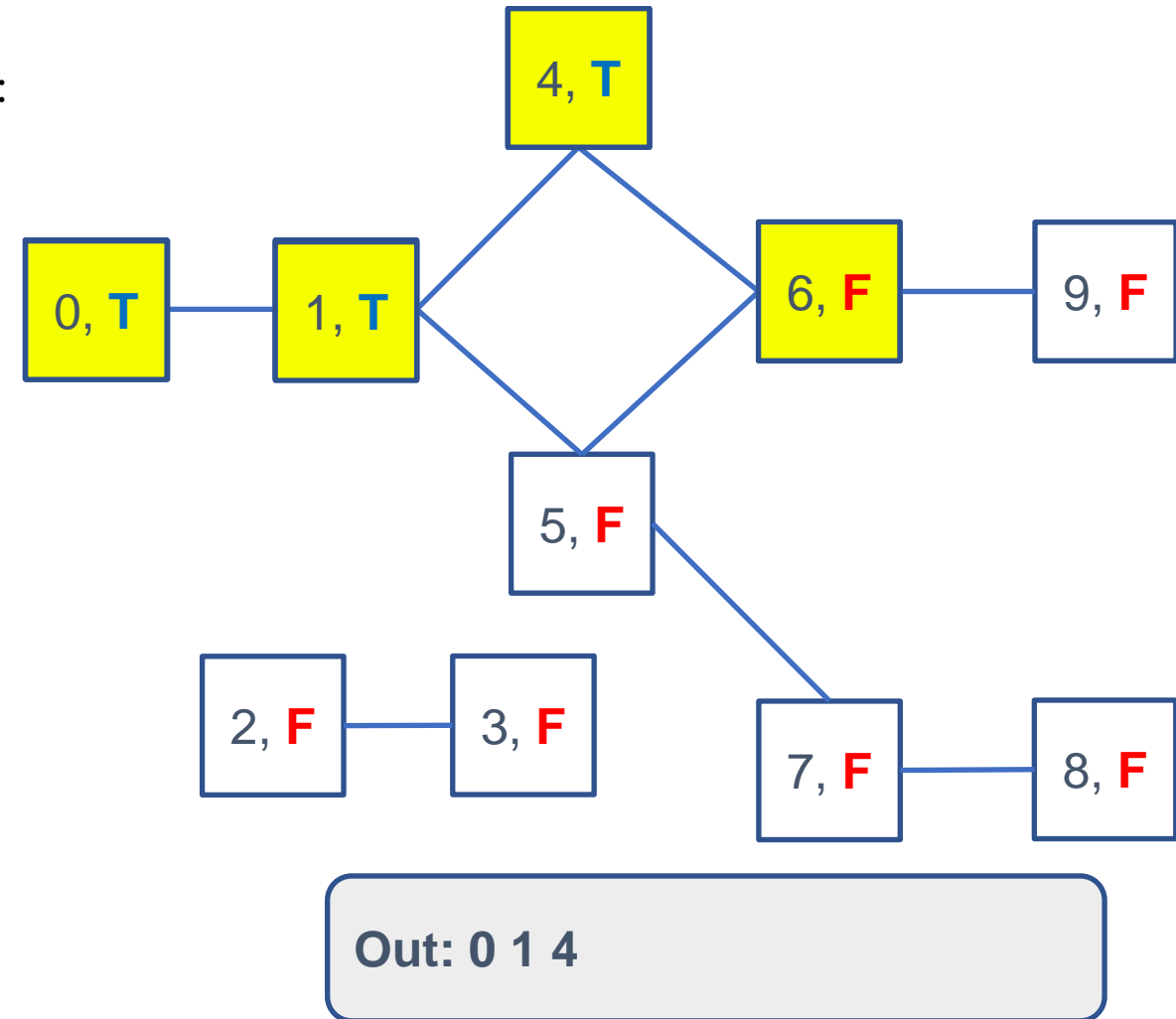
# Depth-First Traversal – Preorder

```
class undi_graph():  
    def __DFTHelp(self, visited: list, v: int) -> None:  
        if not visited[v]:  
            visited[v] = True  
            print(v)  
            for w in self.neighbor[v]:  
                self.__DFTHelp(visited, w)  
  
    def DFT(self) -> None:  
        if self.V:  
            visited = {}  
            for v in self.V:  
                visited[v] = False  
            for v in self.V:  
                self.__DFTHelp(visited, v)
```



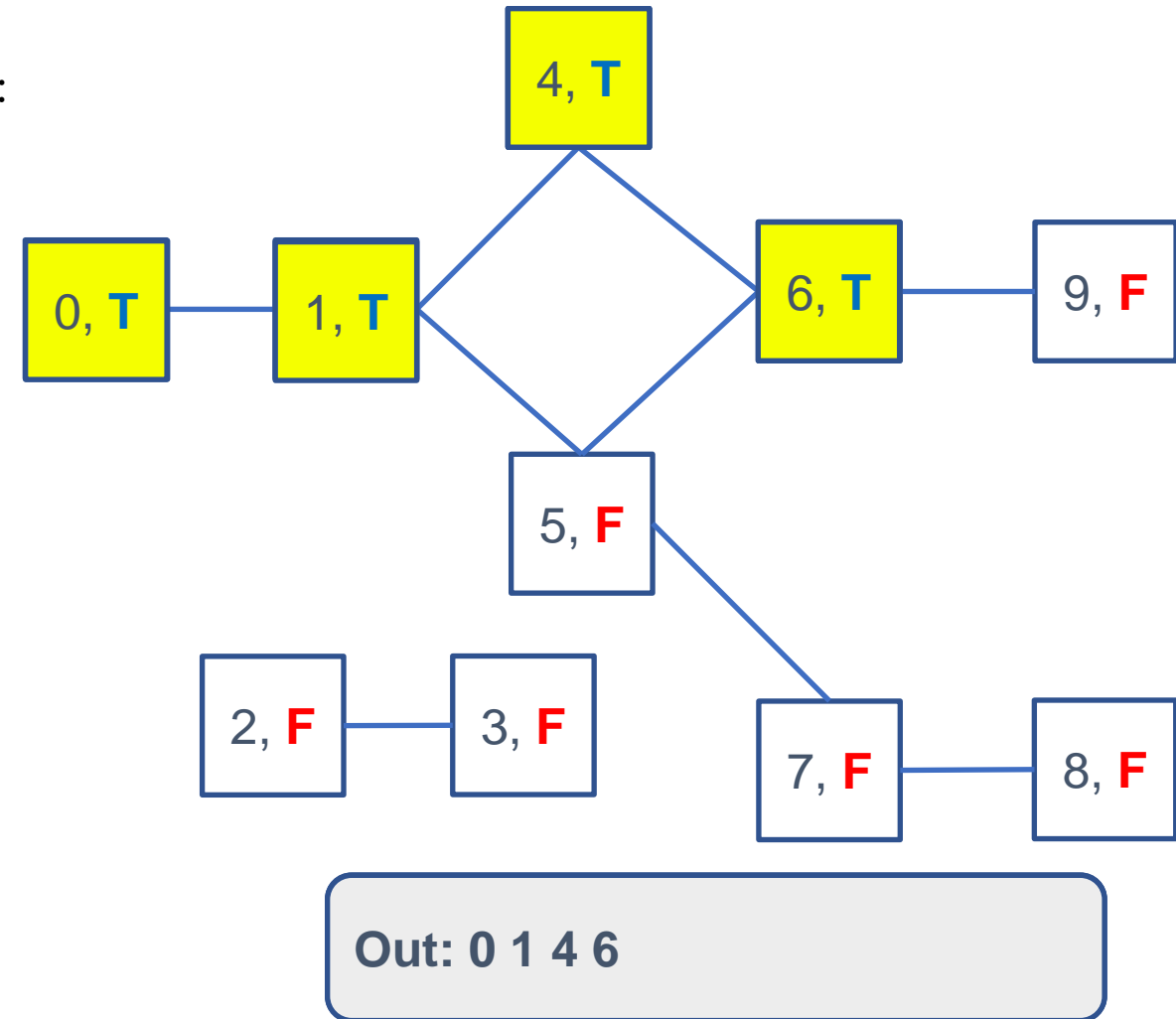
# Depth-First Traversal – Preorder

```
class undi_graph():  
    def __DFTHelp(self, visited: list, v: int) -> None:  
        if not visited[v]:  
            visited[v] = True  
            print(v)  
            for w in self.neighbor[v]:  
                self.__DFTHelp(visited, w)  
  
    def DFT(self) -> None:  
        if self.V:  
            visited = {}  
            for v in self.V:  
                visited[v] = False  
            for v in self.V:  
                self.__DFTHelp(visited, v)
```



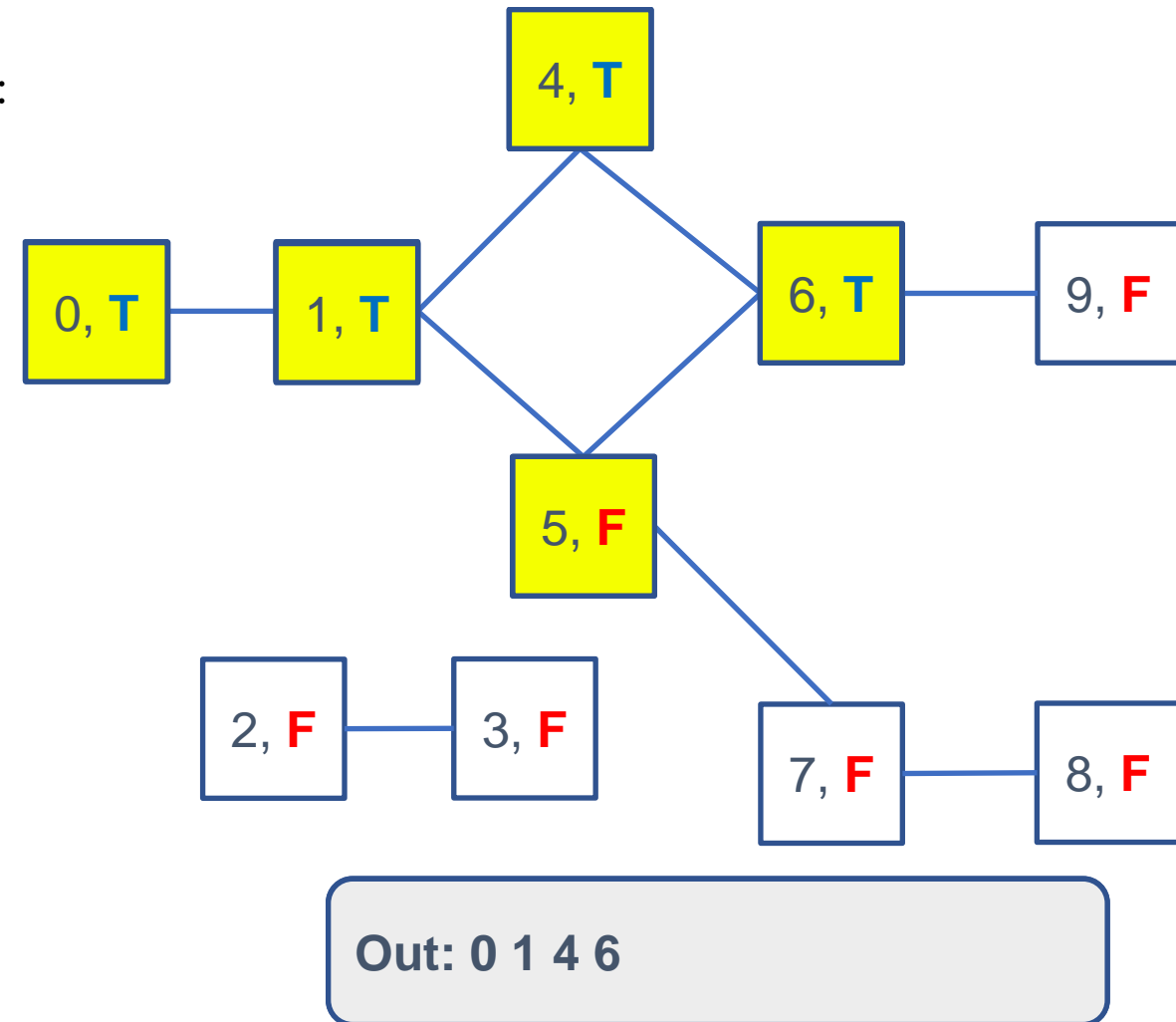
# Depth-First Traversal – Preorder

```
class undi_graph():  
    def __DFTHelp(self, visited: list, v: int) -> None:  
        if not visited[v]:  
            visited[v] = True  
            print(v)  
            for w in self.neighbor[v]:  
                self.__DFTHelp(visited, w)  
  
    def DFT(self) -> None:  
        if self.V:  
            visited = {}  
            for v in self.V:  
                visited[v] = False  
            for v in self.V:  
                self.__DFTHelp(visited, v)
```



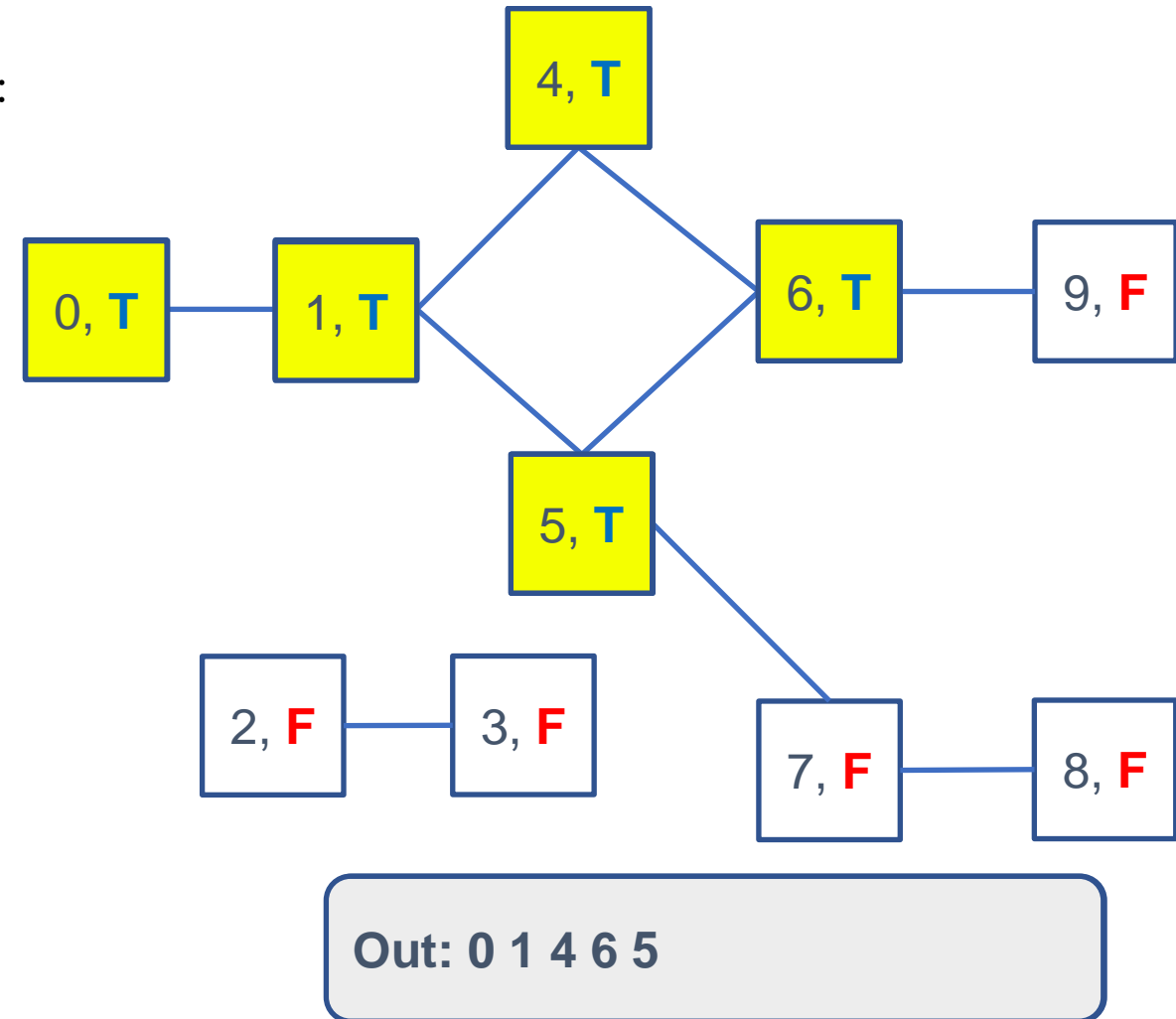
# Depth-First Traversal – Preorder

```
class undi_graph():  
    def __DFTHelp(self, visited: list, v: int) -> None:  
        if not visited[v]:  
            visited[v] = True  
            print(v)  
            for w in self.neighbor[v]:  
                self.__DFTHelp(visited, w)  
  
    def DFT(self) -> None:  
        if self.V:  
            visited = {}  
            for v in self.V:  
                visited[v] = False  
            for v in self.V:  
                self.__DFTHelp(visited, v)
```



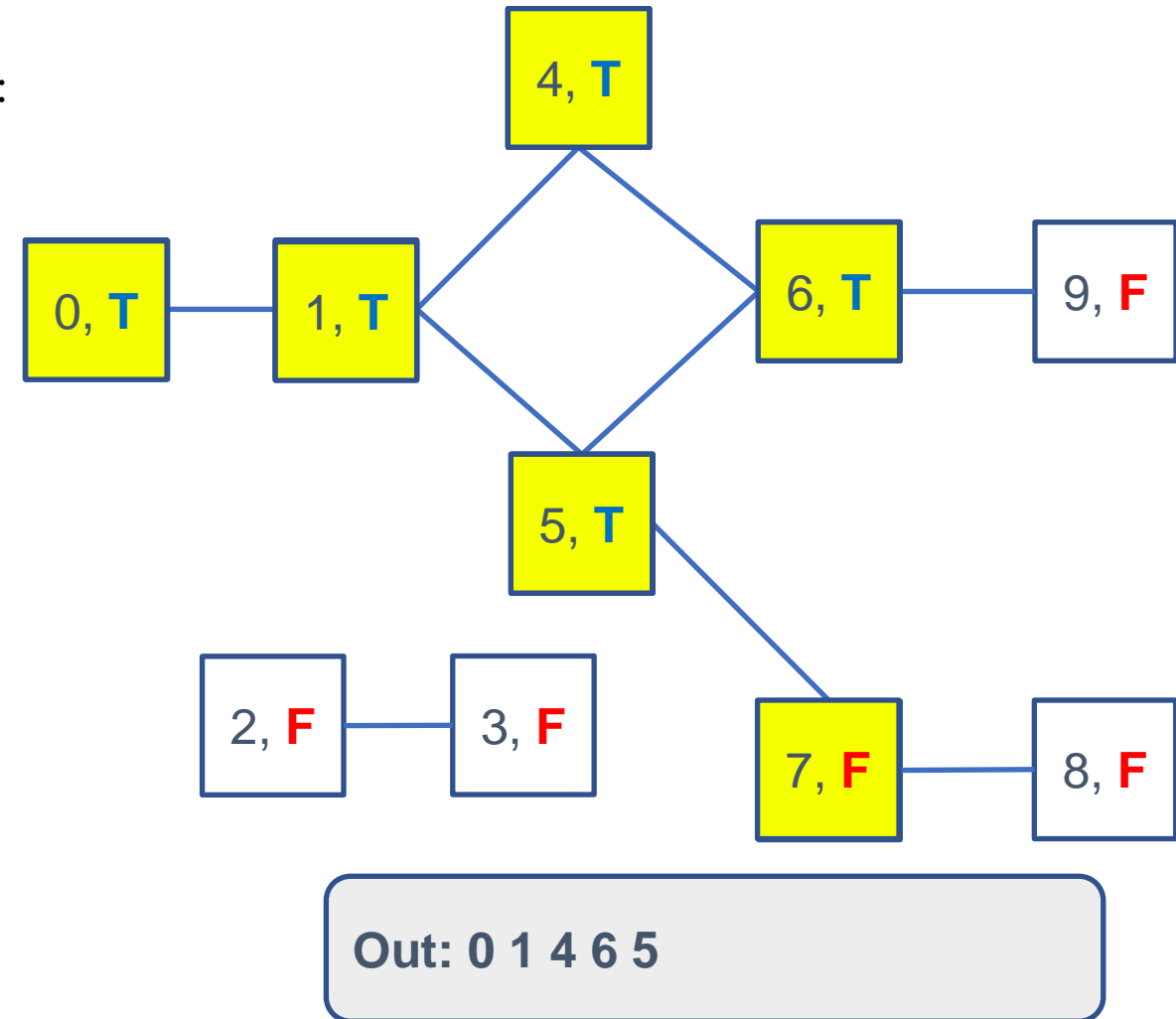
# Depth-First Traversal – Preorder

```
class undi_graph():  
    def __DFTHelp(self, visited: list, v: int) -> None:  
        if not visited[v]:  
            visited[v] = True  
            print(v)  
            for w in self.neighbor[v]:  
                self.__DFTHelp(visited, w)  
  
    def DFT(self) -> None:  
        if self.V:  
            visited = {}  
            for v in self.V:  
                visited[v] = False  
            for v in self.V:  
                self.__DFTHelp(visited, v)
```



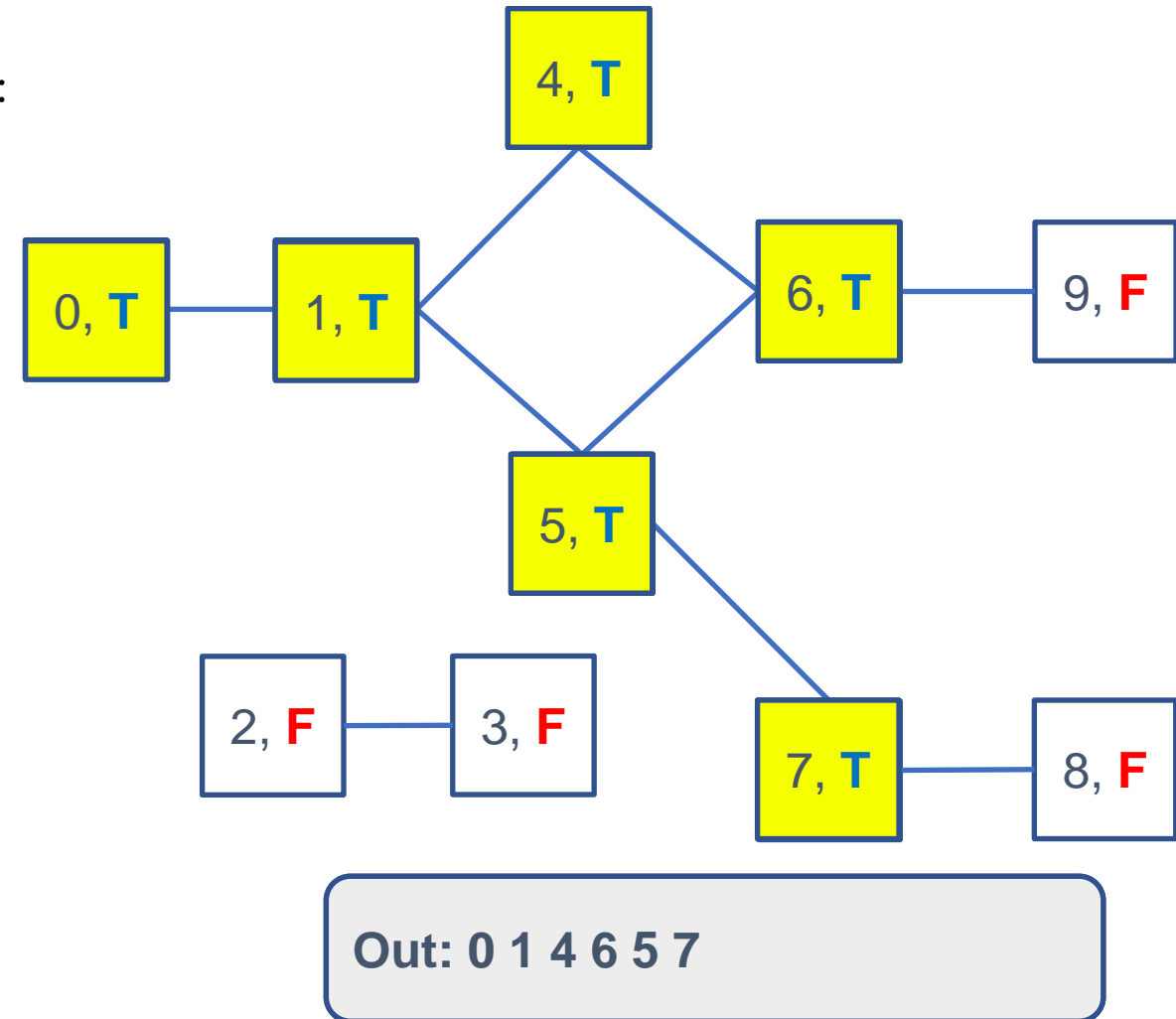
# Depth-First Traversal – Preorder

```
class undi_graph():  
    def __DFTHelp(self, visited: list, v: int) -> None:  
        if not visited[v]:  
            visited[v] = True  
            print(v)  
            for w in self.neighbor[v]:  
                self.__DFTHelp(visited, w)  
  
    def DFT(self) -> None:  
        if self.V:  
            visited = {}  
            for v in self.V:  
                visited[v] = False  
            for v in self.V:  
                self.__DFTHelp(visited, v)
```



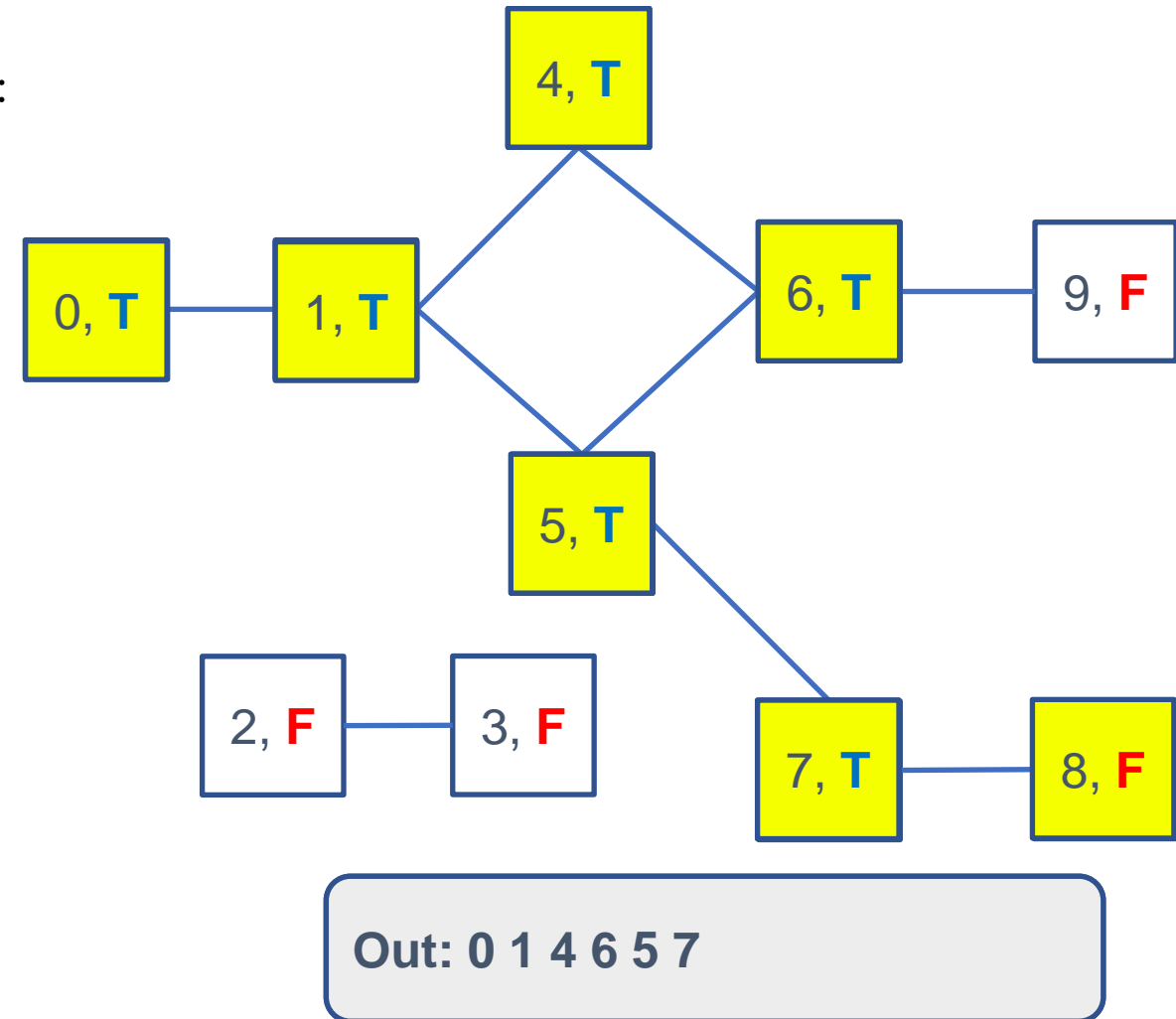
# Depth-First Traversal – Preorder

```
class undi_graph():  
    def __DFTHelp(self, visited: list, v: int) -> None:  
        if not visited[v]:  
            visited[v] = True  
            print(v)  
            for w in self.neighbor[v]:  
                self.__DFTHelp(visited, w)  
  
    def DFT(self) -> None:  
        if self.V:  
            visited = {}  
            for v in self.V:  
                visited[v] = False  
            for v in self.V:  
                self.__DFTHelp(visited, v)
```



# Depth-First Traversal – Preorder

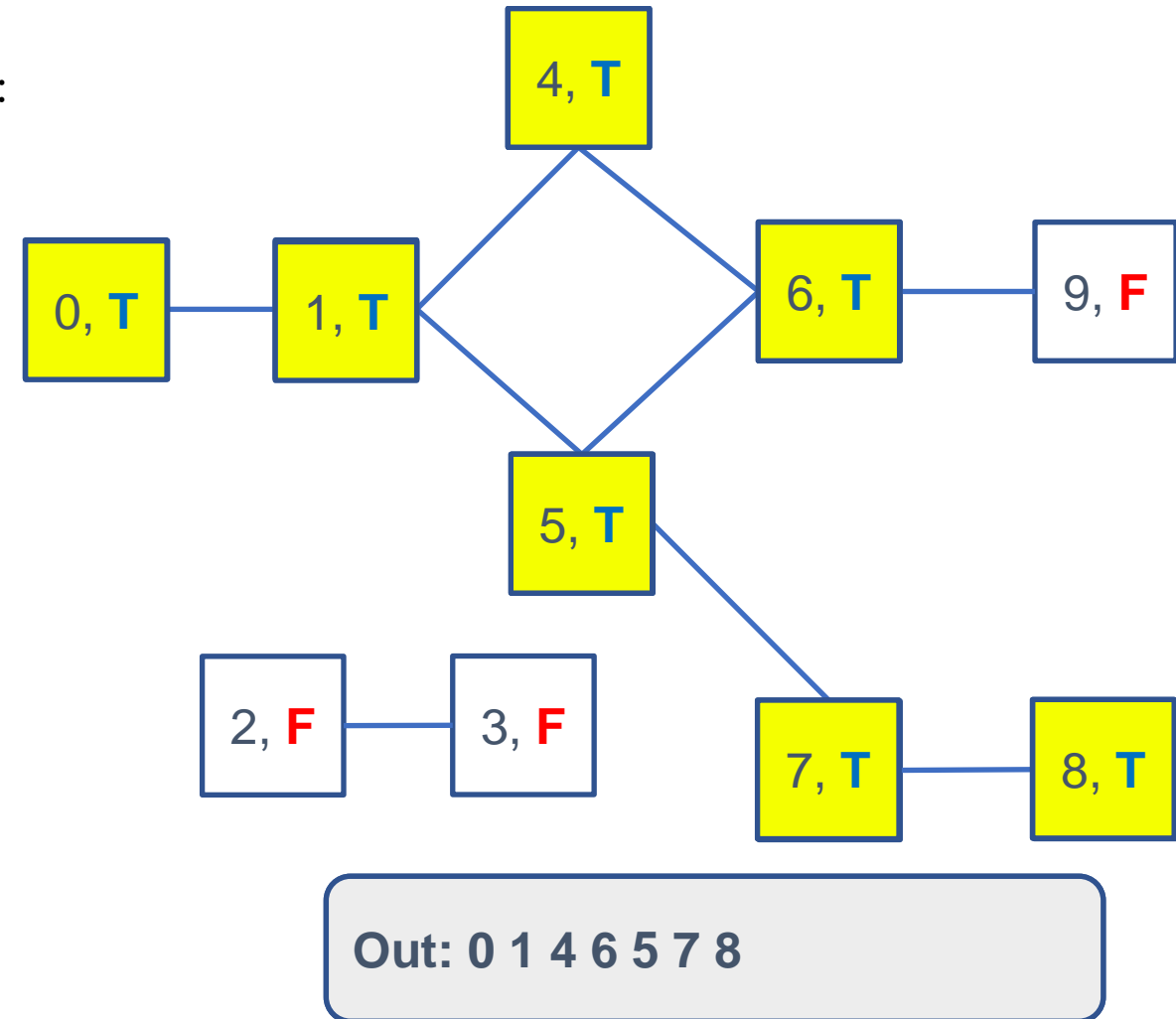
```
class undi_graph():  
    def __DFTHelp(self, visited: list, v: int) -> None:  
        if not visited[v]:  
            visited[v] = True  
            print(v)  
            for w in self.neighbor[v]:  
                self.__DFTHelp(visited, w)  
  
    def DFT(self) -> None:  
        if self.V:  
            visited = {}  
            for v in self.V:  
                visited[v] = False  
            for v in self.V:  
                self.__DFTHelp(visited, v)
```





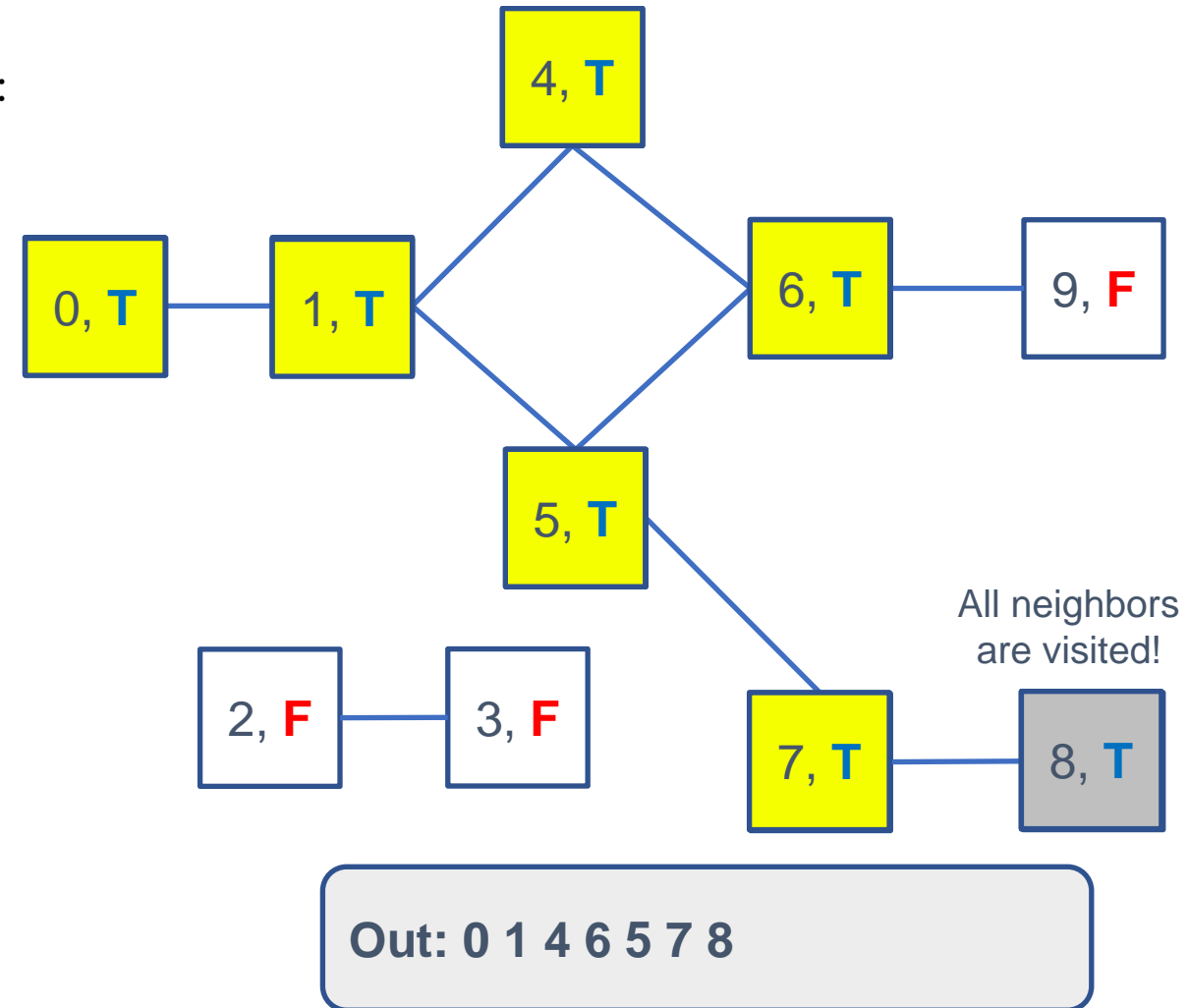
# Depth-First Traversal – Preorder

```
class undi_graph():  
    def __DFTHelp(self, visited: list, v: int) -> None:  
        if not visited[v]:  
            visited[v] = True  
            print(v)  
            for w in self.neighbor[v]:  
                self.__DFTHelp(visited, w)  
  
    def DFT(self) -> None:  
        if self.V:  
            visited = {}  
            for v in self.V:  
                visited[v] = False  
            for v in self.V:  
                self.__DFTHelp(visited, v)
```



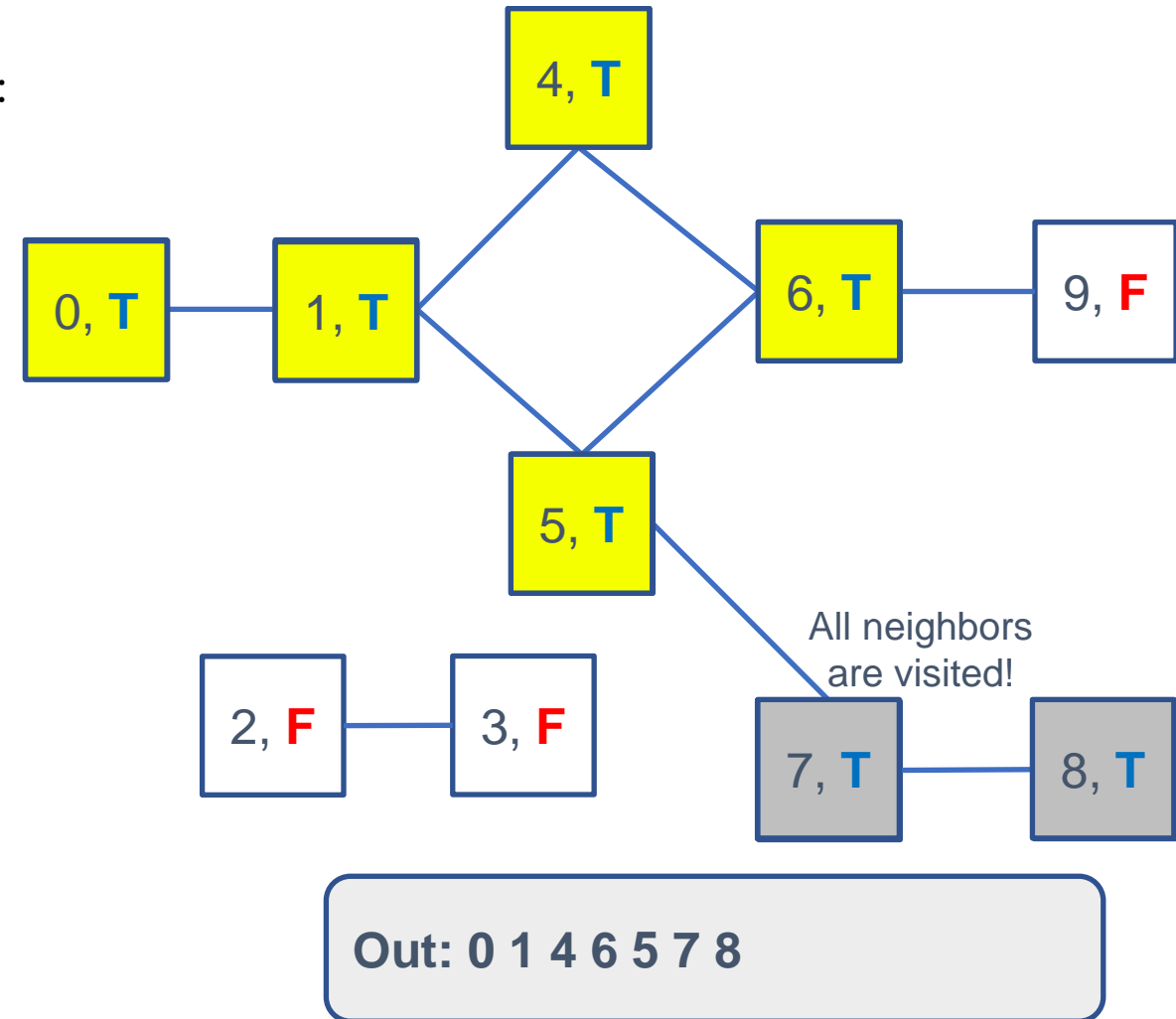
# Depth-First Traversal – Preorder

```
class undi_graph():  
    def __DFTHelp(self, visited: list, v: int) -> None:  
        if not visited[v]:  
            visited[v] = True  
            print(v)  
            for w in self.neighbor[v]:  
                self.__DFTHelp(visited, w)  
  
    def DFT(self) -> None:  
        if self.V:  
            visited = {}  
            for v in self.V:  
                visited[v] = False  
            for v in self.V:  
                self.__DFTHelp(visited, v)
```



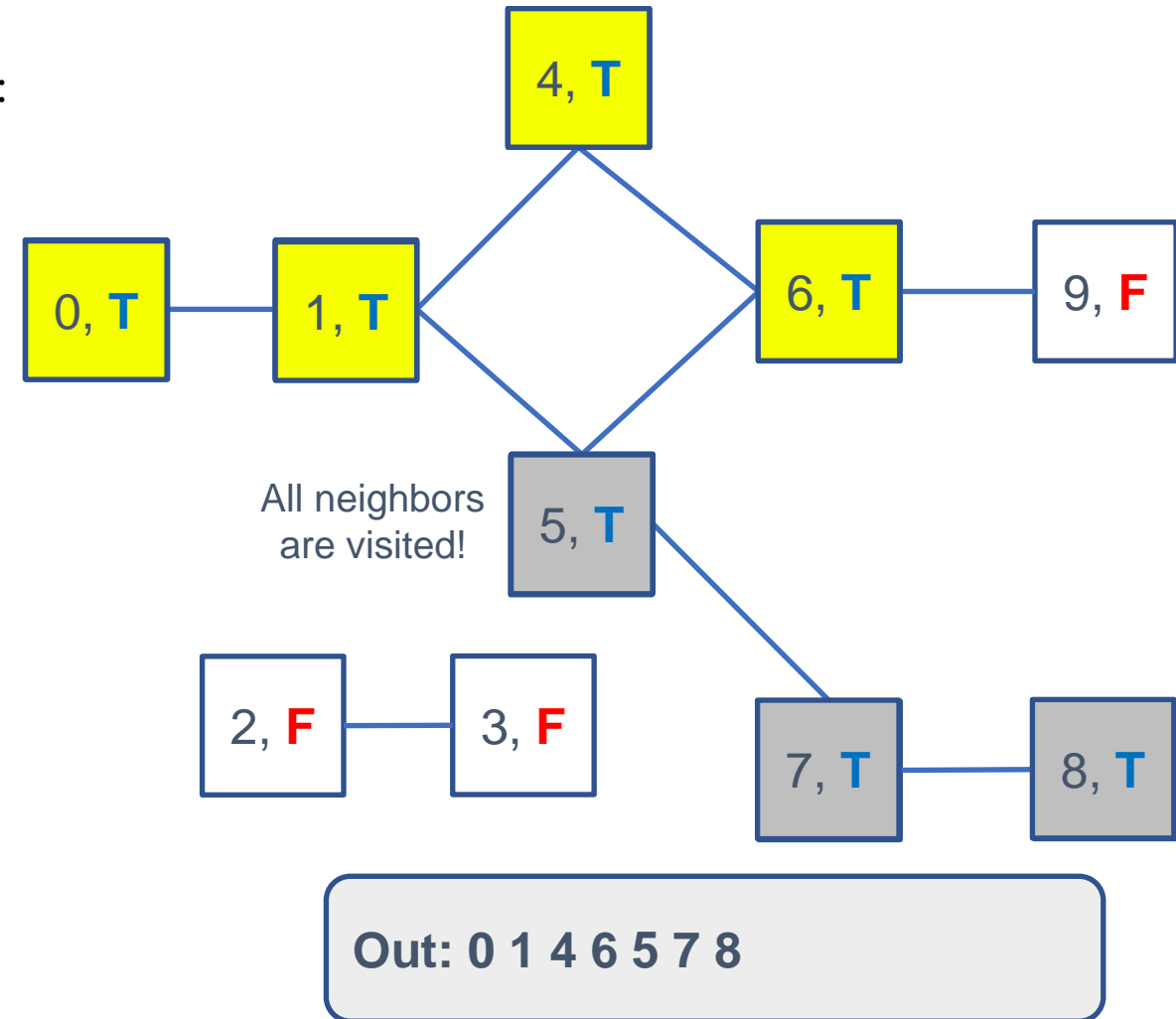
# Depth-First Traversal – Preorder

```
class undi_graph():  
    def __DFTHelp(self, visited: list, v: int) -> None:  
        if not visited[v]:  
            visited[v] = True  
            print(v)  
            for w in self.neighbor[v]:  
                self.__DFTHelp(visited, w)  
  
    def DFT(self) -> None:  
        if self.V:  
            visited = {}  
            for v in self.V:  
                visited[v] = False  
            for v in self.V:  
                self.__DFTHelp(visited, v)
```



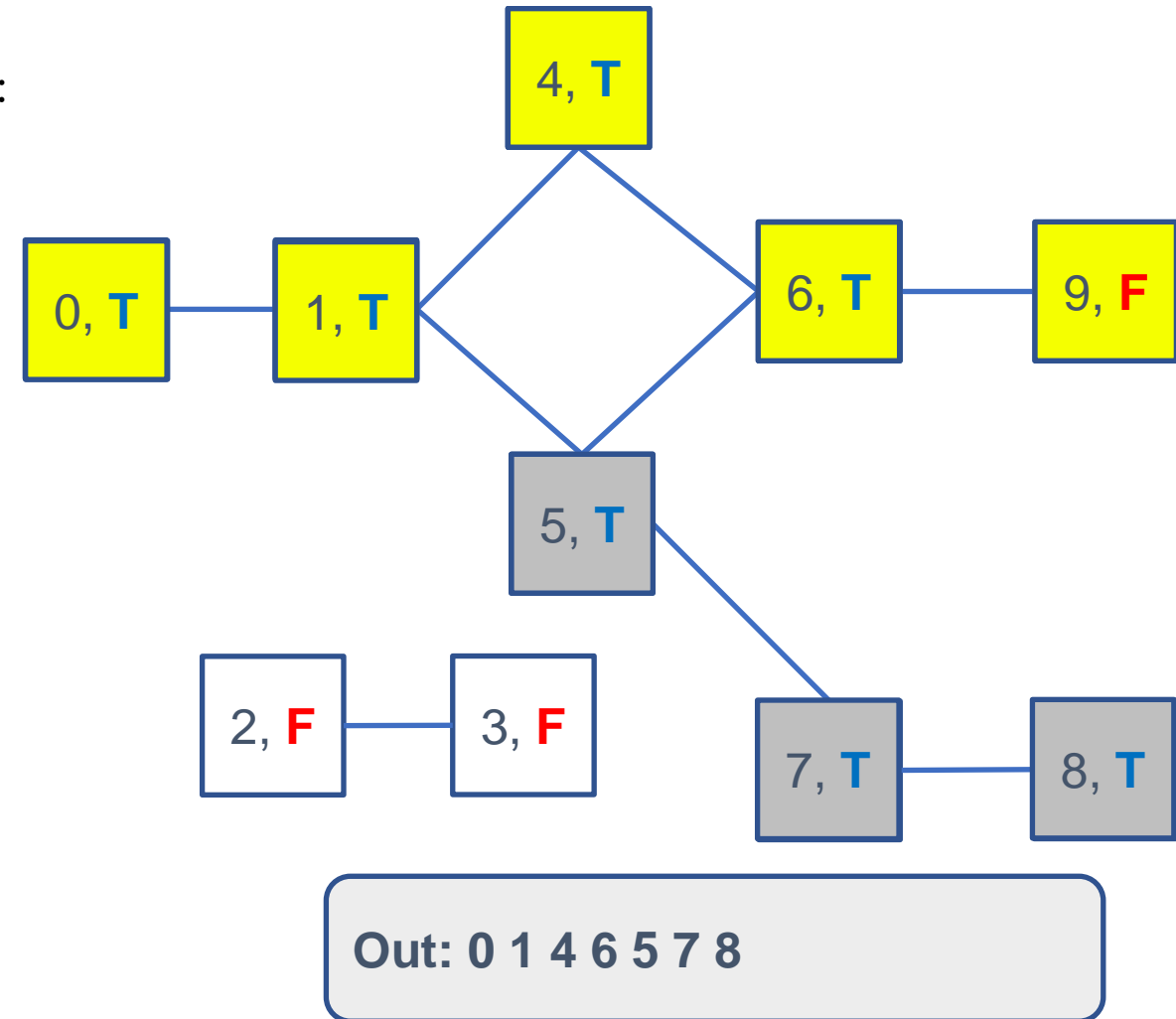
# Depth-First Traversal – Preorder

```
class undi_graph():  
    def __DFTHelp(self, visited: list, v: int) -> None:  
        if not visited[v]:  
            visited[v] = True  
            print(v)  
            for w in self.neighbor[v]:  
                self.__DFTHelp(visited, w)  
  
    def DFT(self) -> None:  
        if self.V:  
            visited = {}  
            for v in self.V:  
                visited[v] = False  
            for v in self.V:  
                self.__DFTHelp(visited, v)
```



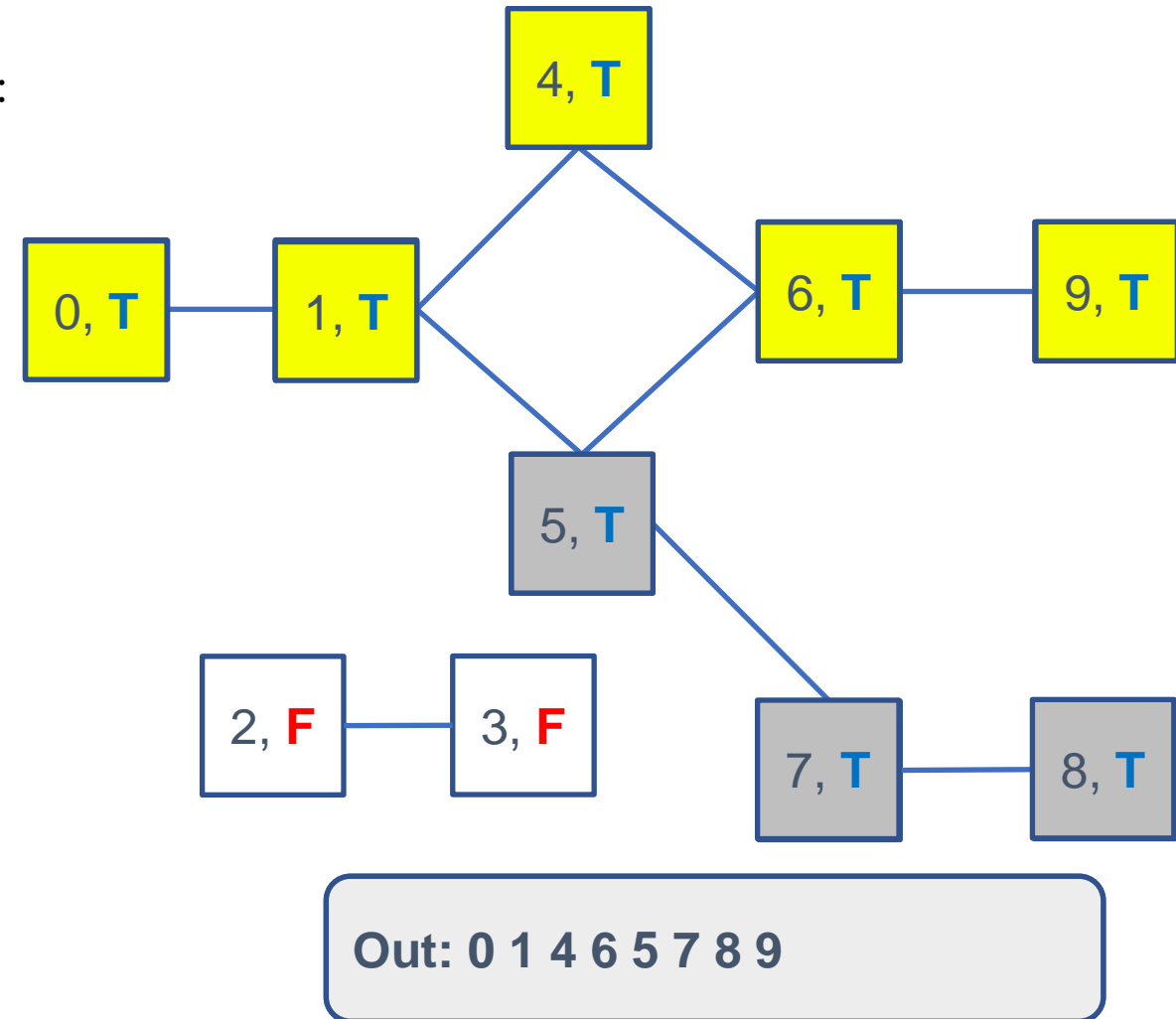
# Depth-First Traversal – Preorder

```
class undi_graph():  
    def __DFTHelp(self, visited: list, v: int) -> None:  
        if not visited[v]:  
            visited[v] = True  
            print(v)  
            for w in self.neighbor[v]:  
                self.__DFTHelp(visited, w)  
  
    def DFT(self) -> None:  
        if self.V:  
            visited = {}  
            for v in self.V:  
                visited[v] = False  
            for v in self.V:  
                self.__DFTHelp(visited, v)
```



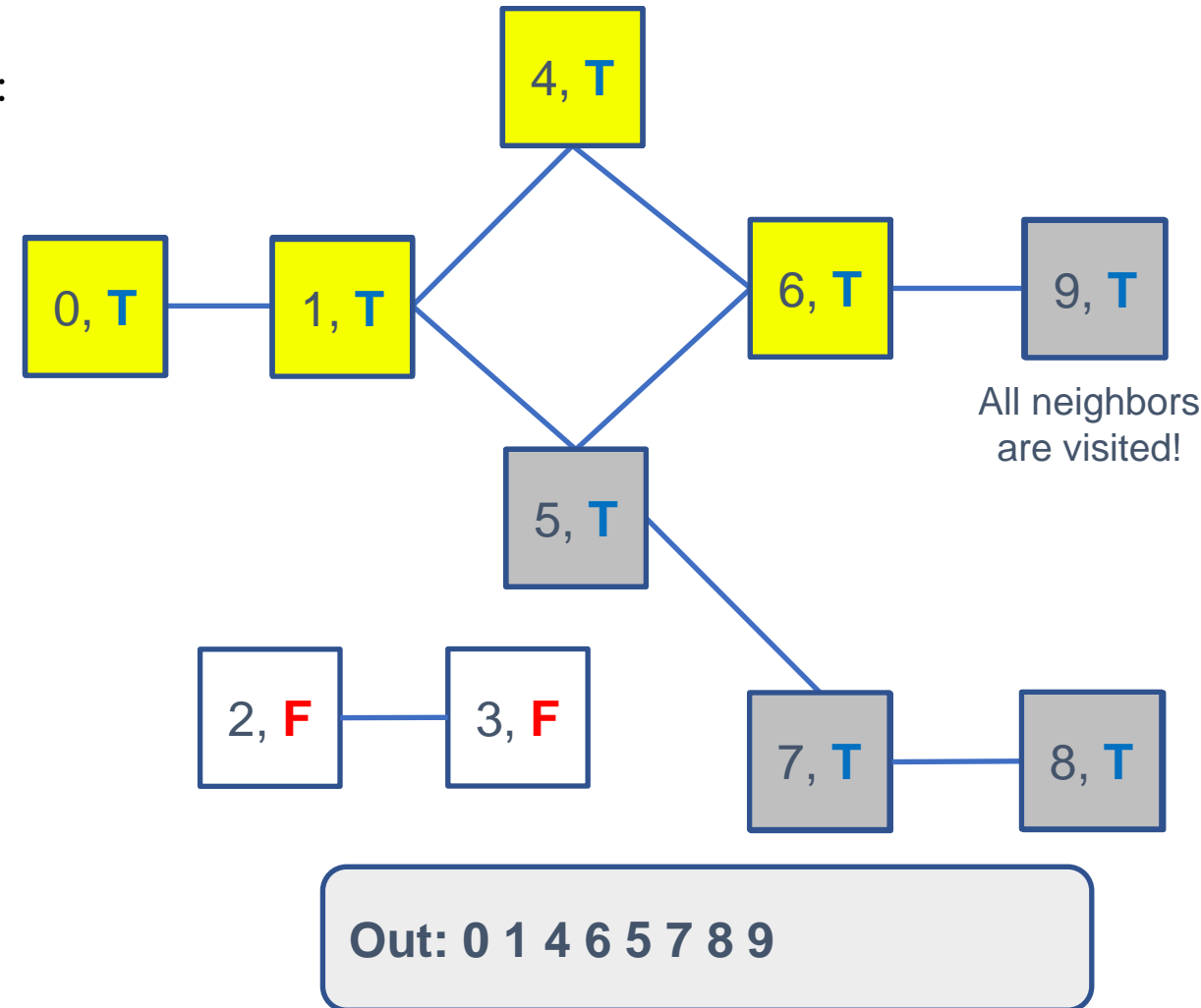
# Depth-First Traversal – Preorder

```
class undi_graph():  
    def __DFTHelp(self, visited: list, v: int) -> None:  
        if not visited[v]:  
            visited[v] = True  
            print(v)  
            for w in self.neighbor[v]:  
                self.__DFTHelp(visited, w)  
  
    def DFT(self) -> None:  
        if self.V:  
            visited = {}  
            for v in self.V:  
                visited[v] = False  
            for v in self.V:  
                self.__DFTHelp(visited, v)
```



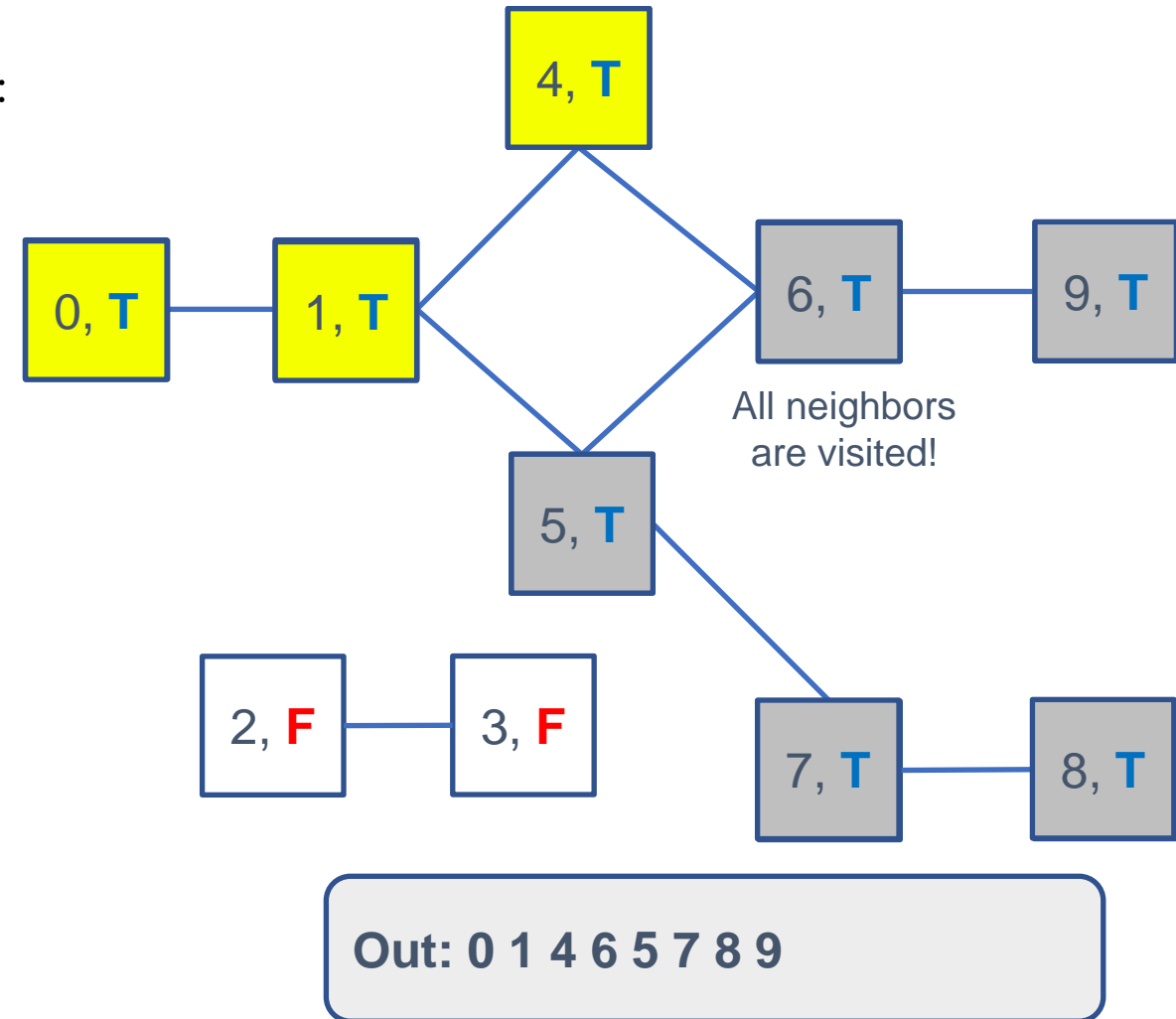
# Depth-First Traversal – Preorder

```
class undi_graph():  
    def __DFTHelp(self, visited: list, v: int) -> None:  
        if not visited[v]:  
            visited[v] = True  
            print(v)  
            for w in self.neighbor[v]:  
                self.__DFTHelp(visited, w)  
  
    def DFT(self) -> None:  
        if self.V:  
            visited = {}  
            for v in self.V:  
                visited[v] = False  
            for v in self.V:  
                self.__DFTHelp(visited, v)
```



# Depth-First Traversal – Preorder

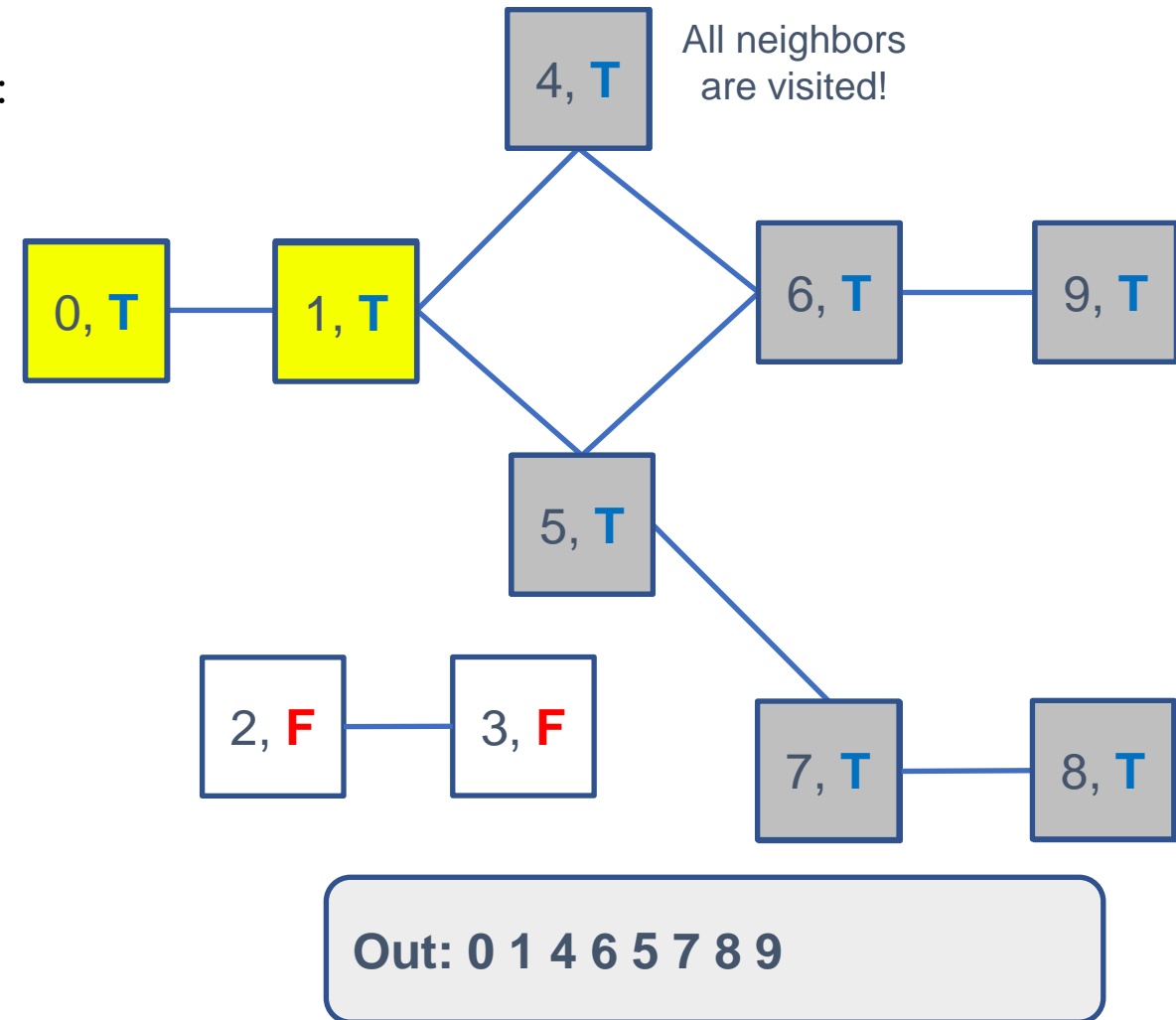
```
• class undi_graph():  
•     def __DFTHelp(self, visited: list, v: int) -> None:  
•         if not visited[v]:  
•             visited[v] = True  
•             print(v)  
•             for w in self.neighbor[v]:  
•                 self.__DFTHelp(visited, w)  
•  
•     def DFT(self) -> None:  
•         if self.V:  
•             visited = {}  
•             for v in self.V:  
•                 visited[v] = False  
•             for v in self.V:  
•                 self.__DFTHelp(visited, v)
```





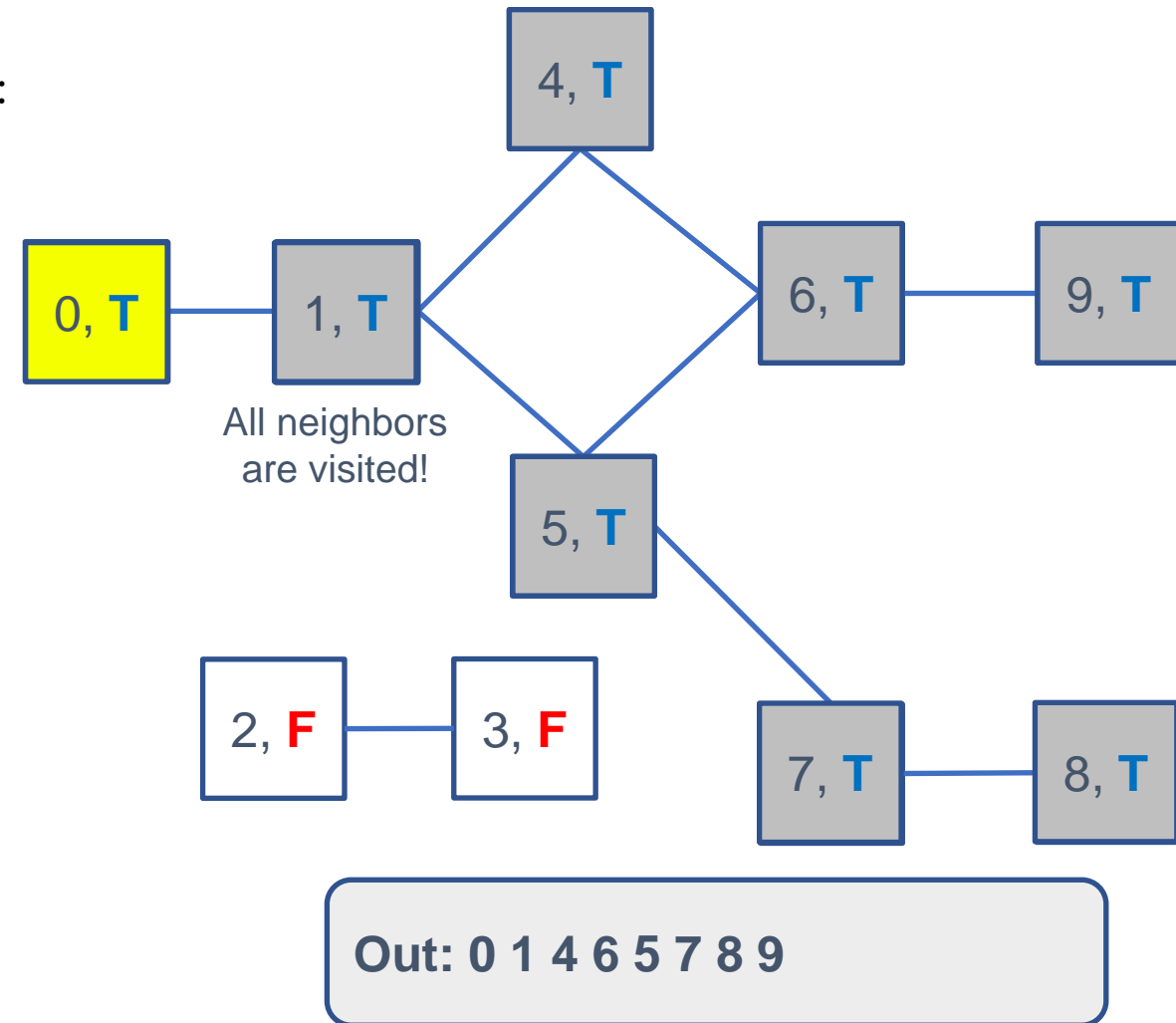
# Depth-First Traversal – Preorder

```
• class undi_graph():
•     def __DFTHelp(self, visited: list, v: int) -> None:
•         if not visited[v]:
•             visited[v] = True
•             print(v)
•             for w in self.neighbor[v]:
•                 self.__DFTHelp(visited, w)
•
•     def DFT(self) -> None:
•         if self.V:
•             visited = {}
•             for v in self.V:
•                 visited[v] = False
•             for v in self.V:
•                 self.__DFTHelp(visited, v)
```



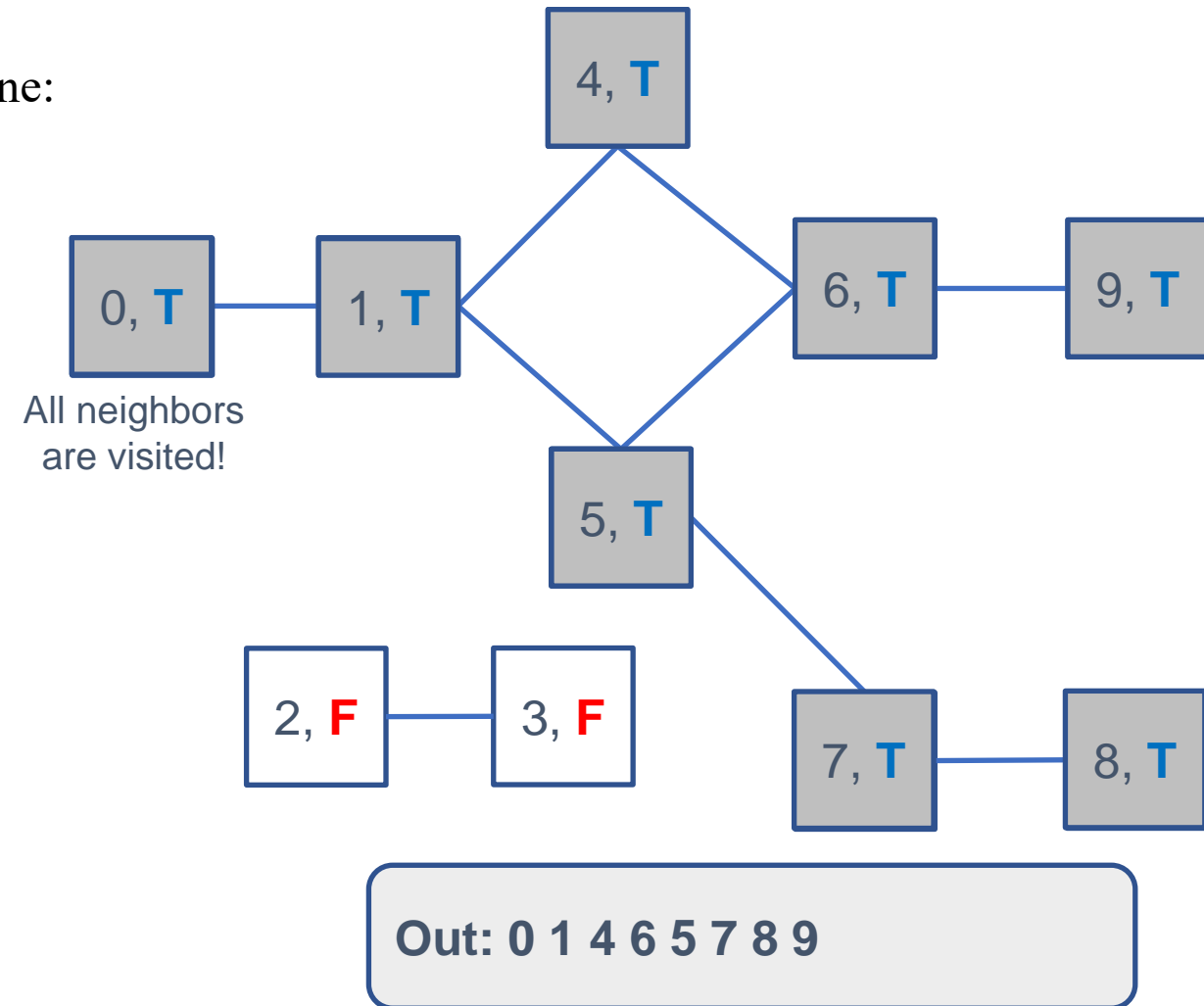
# Depth-First Traversal – Preorder

```
class undi_graph():  
    def __DFTHelp(self, visited: list, v: int) -> None:  
        if not visited[v]:  
            visited[v] = True  
            print(v)  
            for w in self.neighbor[v]:  
                self.__DFTHelp(visited, w)  
  
    def DFT(self) -> None:  
        if self.V:  
            visited = {}  
            for v in self.V:  
                visited[v] = False  
            for v in self.V:  
                self.__DFTHelp(visited, v)
```



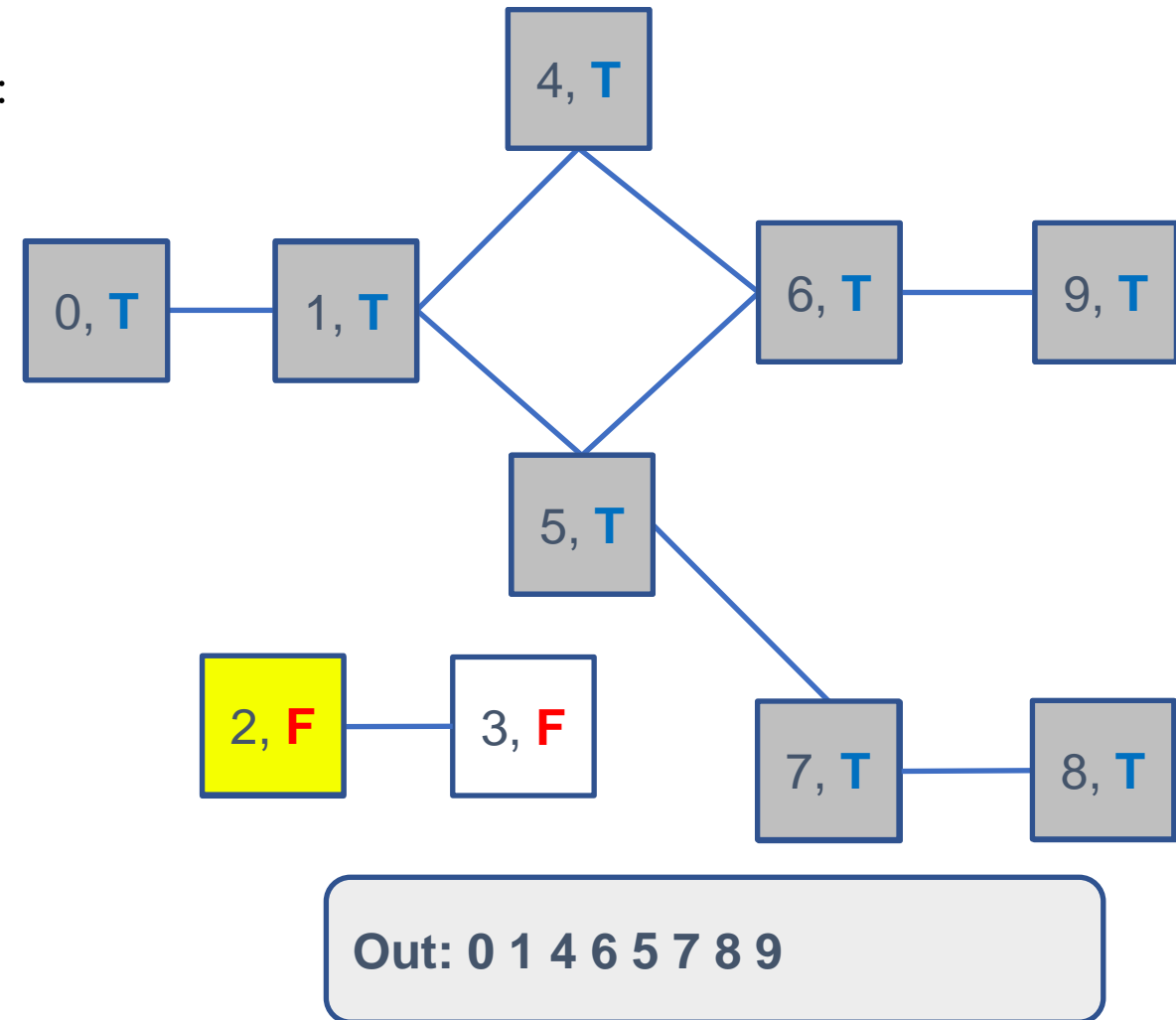
# Depth-First Traversal – Preorder

```
class undi_graph():  
    def __DFTHelp(self, visited: list, v: int) -> None:  
        if not visited[v]:  
            visited[v] = True  
            print(v)  
            for w in self.neighbor[v]:  
                self.__DFTHelp(visited, w)  
  
    def DFT(self) -> None:  
        if self.V:  
            visited = {}  
            for v in self.V:  
                visited[v] = False  
            for v in self.V:  
                self.__DFTHelp(visited, v)
```



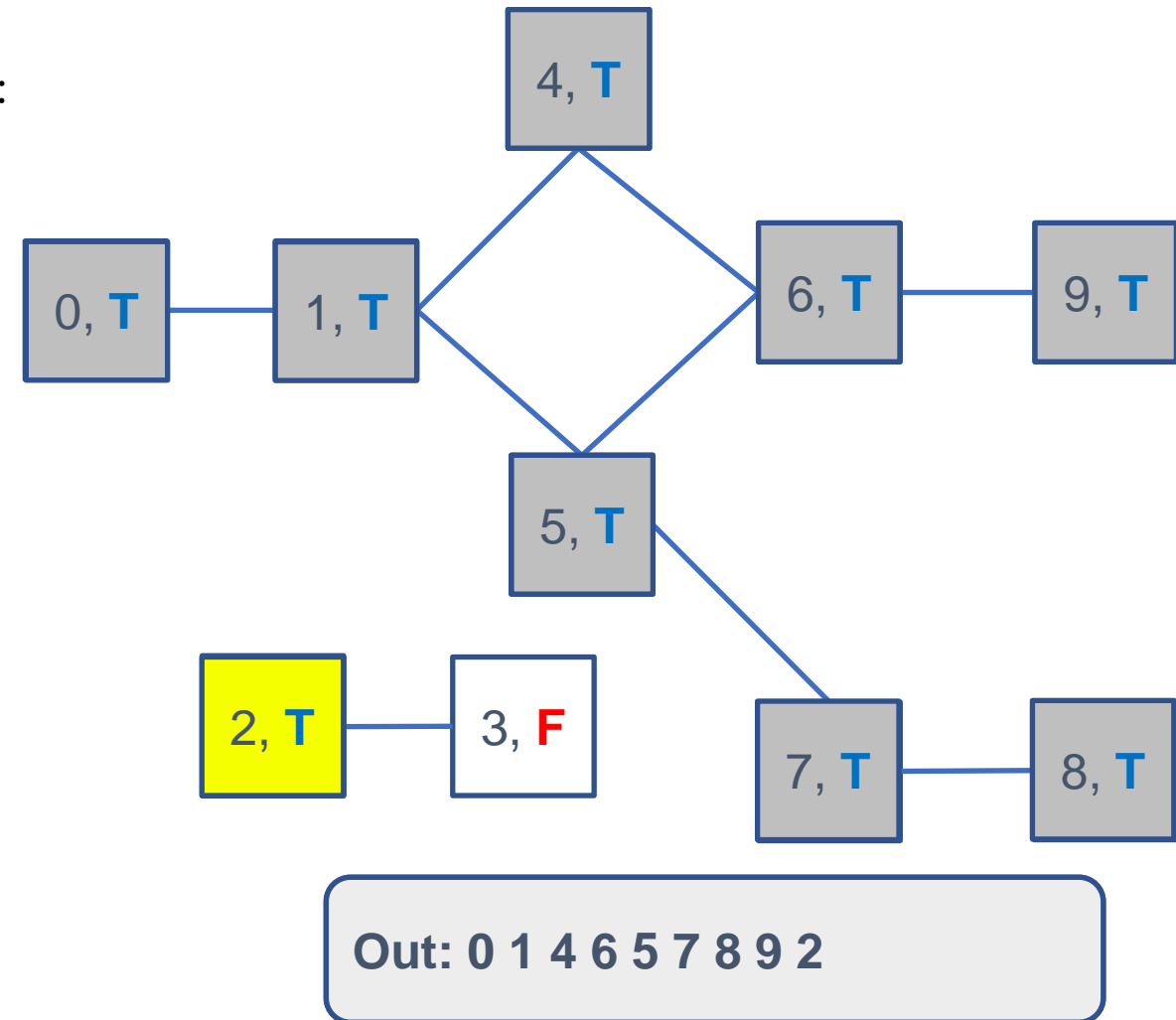
# Depth-First Traversal – Preorder

```
class undi_graph():  
    def __DFTHelp(self, visited: list, v: int) -> None:  
        if not visited[v]:  
            visited[v] = True  
            print(v)  
            for w in self.neighbor[v]:  
                self.__DFTHelp(visited, w)  
  
    def DFT(self) -> None:  
        if self.V:  
            visited = {}  
            for v in self.V:  
                visited[v] = False  
            for v in self.V:  
                self.__DFTHelp(visited, v)
```



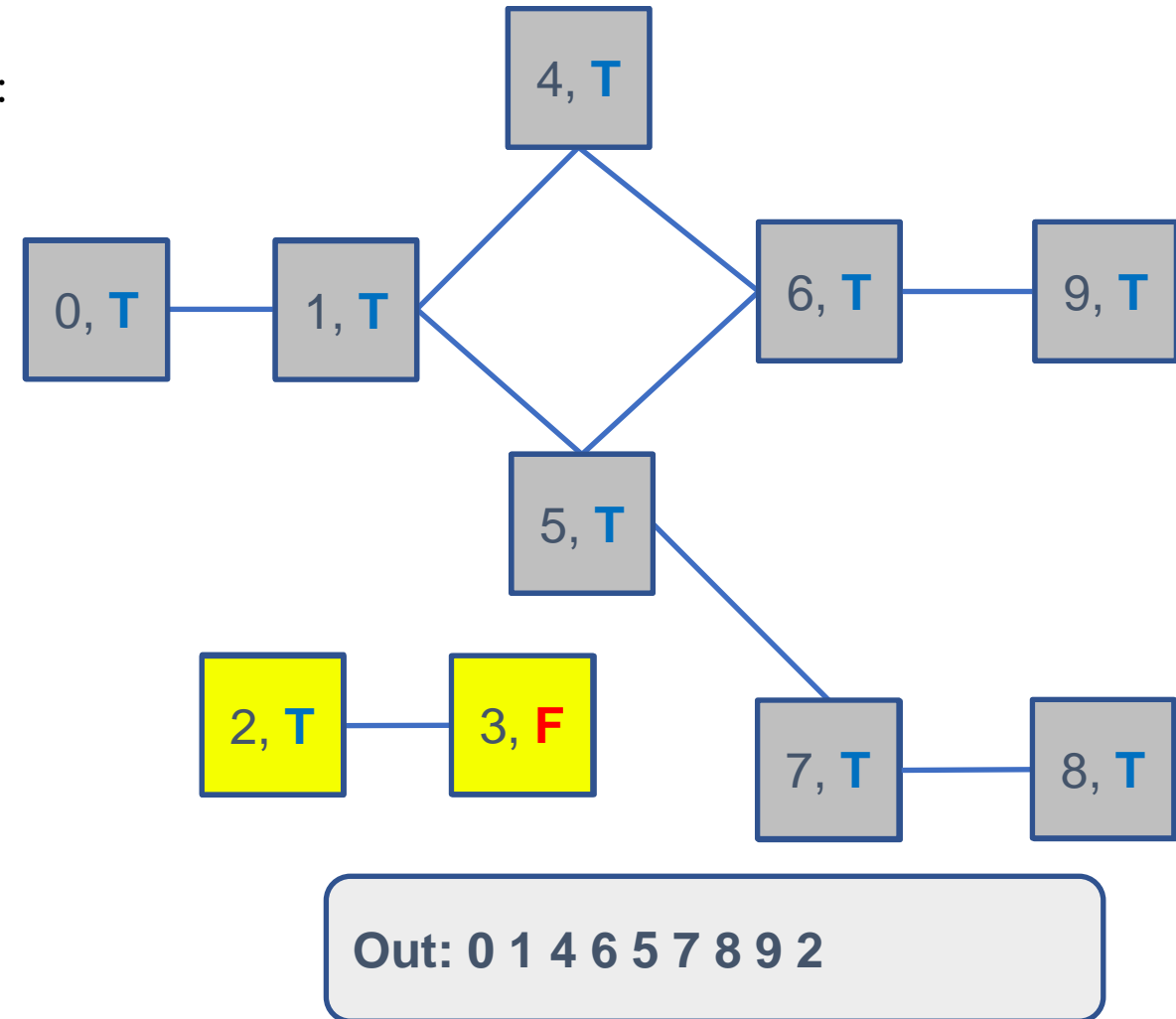
# Depth-First Traversal – Preorder

```
class undi_graph():  
    def __DFTHelp(self, visited: list, v: int) -> None:  
        if not visited[v]:  
            visited[v] = True  
            print(v)  
            for w in self.neighbor[v]:  
                self.__DFTHelp(visited, w)  
  
    def DFT(self) -> None:  
        if self.V:  
            visited = {}  
            for v in self.V:  
                visited[v] = False  
            for v in self.V:  
                self.__DFTHelp(visited, v)
```



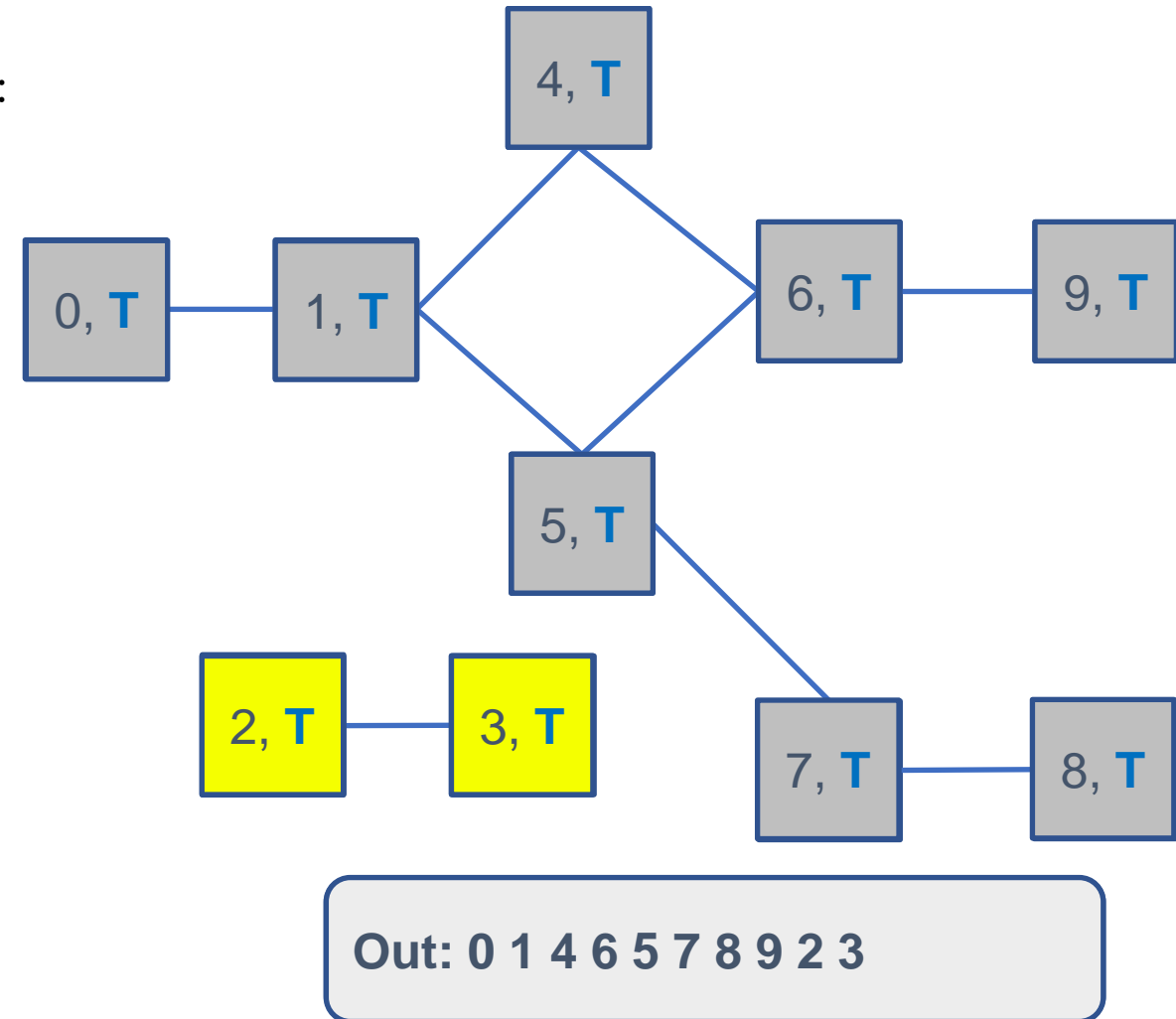
# Depth-First Traversal – Preorder

```
• class undi_graph():
•     def __DFTHelp(self, visited: list, v: int) -> None:
•         if not visited[v]:
•             visited[v] = True
•             print(v)
•             for w in self.neighbor[v]:
•                 self.__DFTHelp(visited, w)
•
•     def DFT(self) -> None:
•         if self.V:
•             visited = {}
•             for v in self.V:
•                 visited[v] = False
•             for v in self.V:
•                 self.__DFTHelp(visited, v)
```



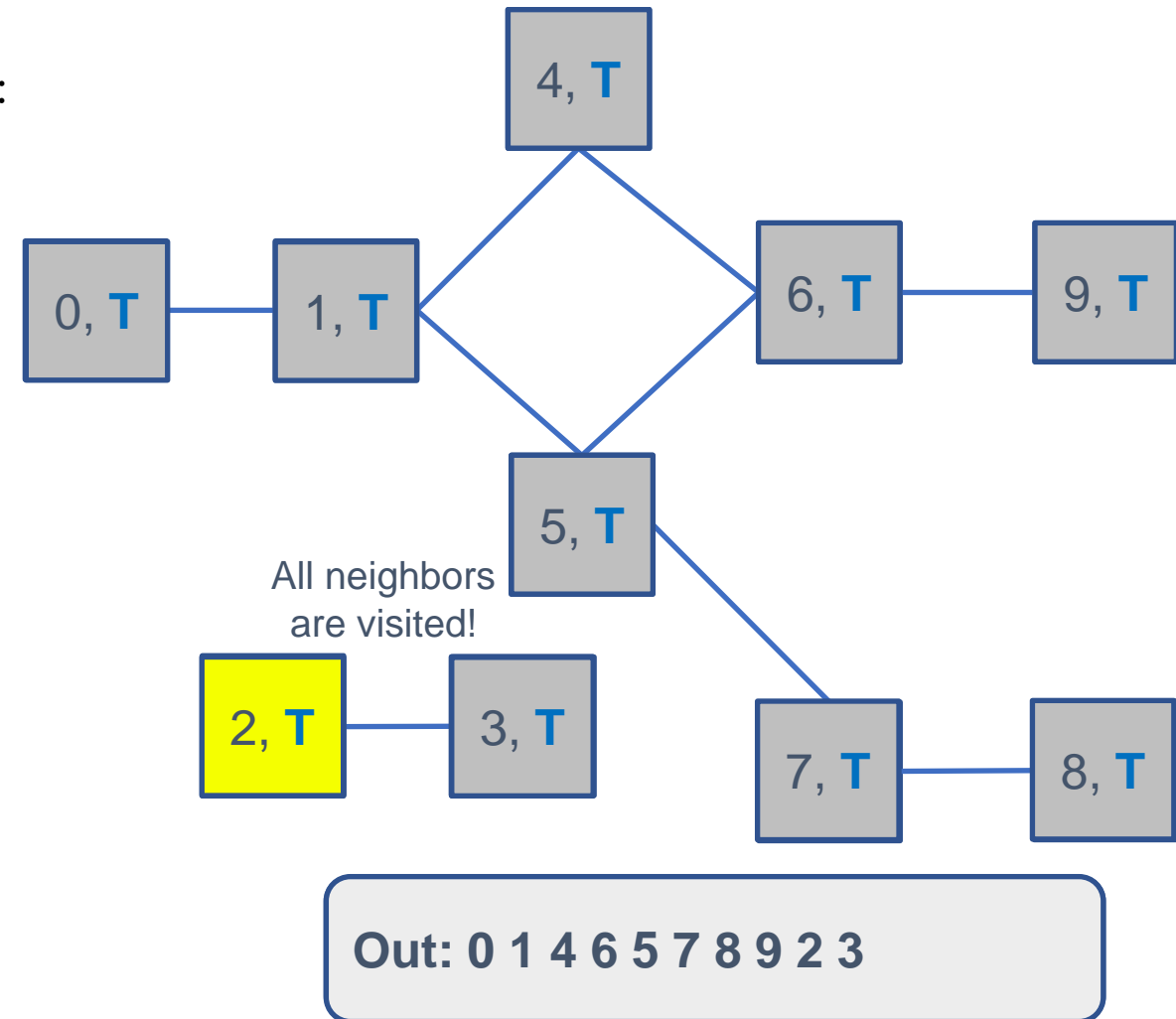
# Depth-First Traversal – Preorder

```
class undi_graph():  
    def __DFTHelp(self, visited: list, v: int) -> None:  
        if not visited[v]:  
            visited[v] = True  
            print(v)  
            for w in self.neighbor[v]:  
                self.__DFTHelp(visited, w)  
  
    def DFT(self) -> None:  
        if self.V:  
            visited = {}  
            for v in self.V:  
                visited[v] = False  
            for v in self.V:  
                self.__DFTHelp(visited, v)
```



# Depth-First Traversal – Preorder

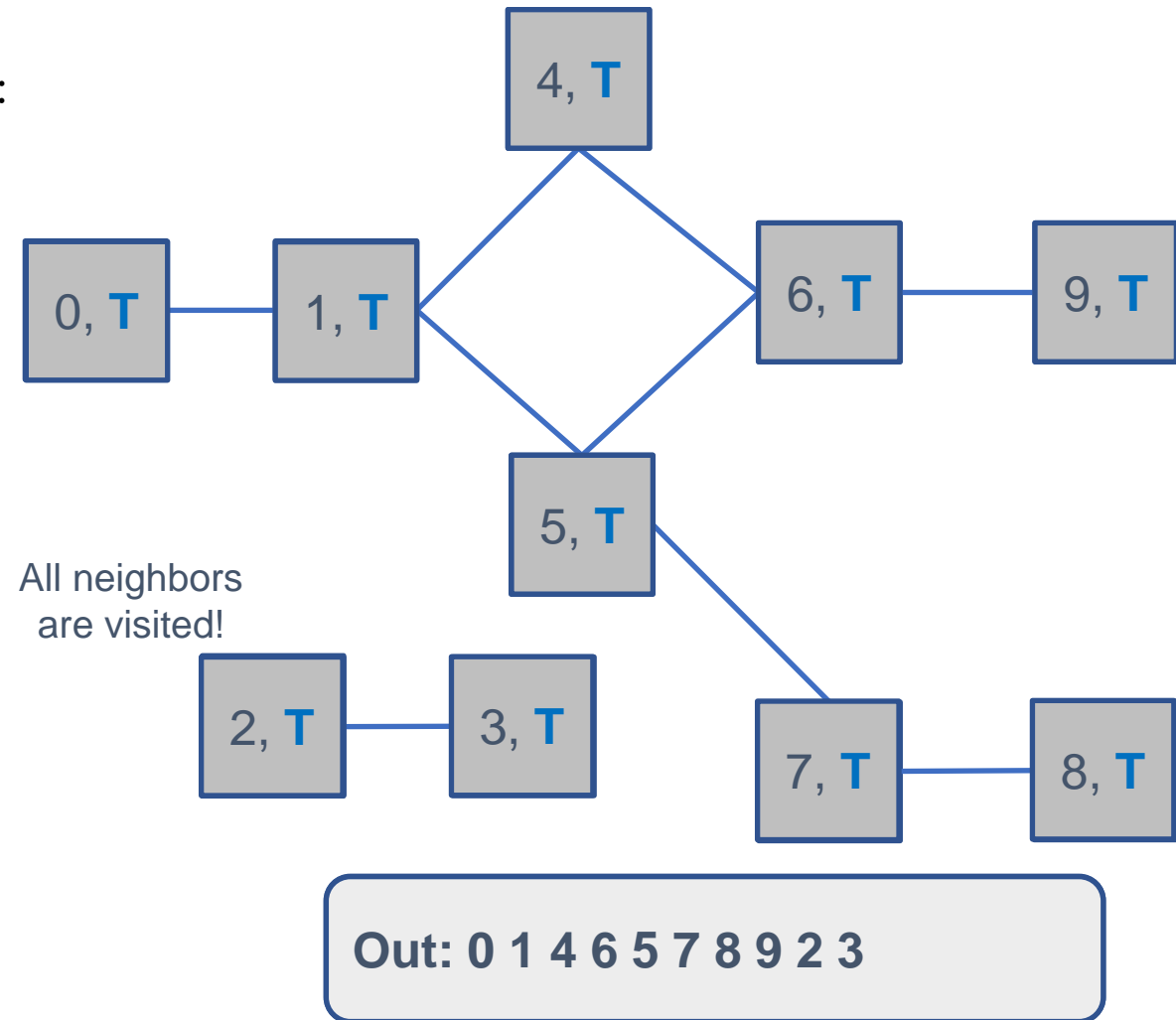
```
• class undi_graph():  
•     def __DFTHelp(self, visited: list, v: int) -> None:  
•         if not visited[v]:  
•             visited[v] = True  
•             print(v)  
•             for w in self.neighbor[v]:  
•                 self.__DFTHelp(visited, w)  
•  
•     def DFT(self) -> None:  
•         if self.V:  
•             visited = {}  
•             for v in self.V:  
•                 visited[v] = False  
•             for v in self.V:  
•                 self.__DFTHelp(visited, v)
```





# Depth-First Traversal – Preorder

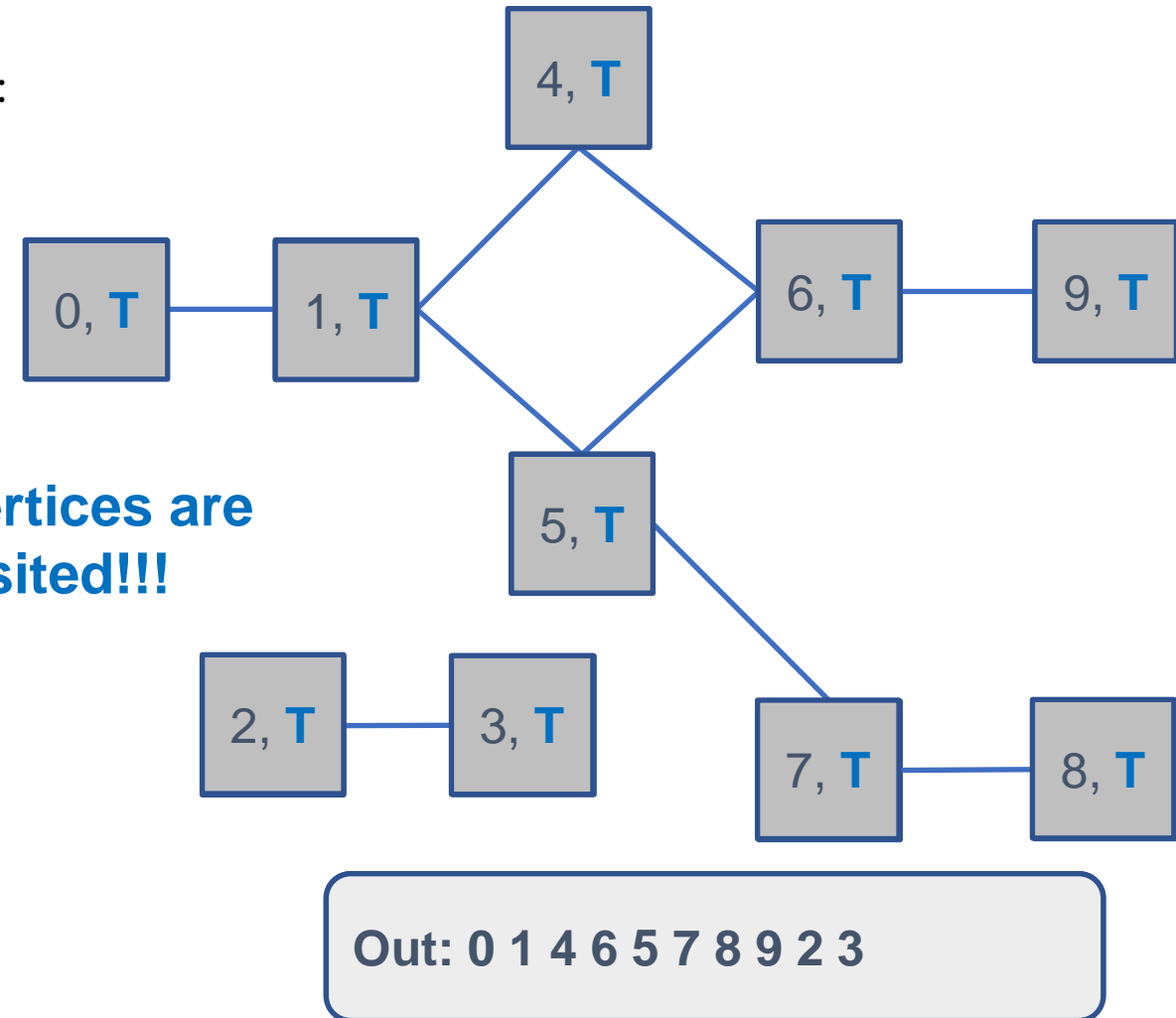
```
class undi_graph():  
    def __DFTHelp(self, visited: list, v: int) -> None:  
        if not visited[v]:  
            visited[v] = True  
            print(v)  
            for w in self.neighbor[v]:  
                self.__DFTHelp(visited, w)  
  
    def DFT(self) -> None:  
        if self.V:  
            visited = {}  
            for v in self.V:  
                visited[v] = False  
            for v in self.V:  
                self.__DFTHelp(visited, v)
```



# Depth-First Traversal – Preorder

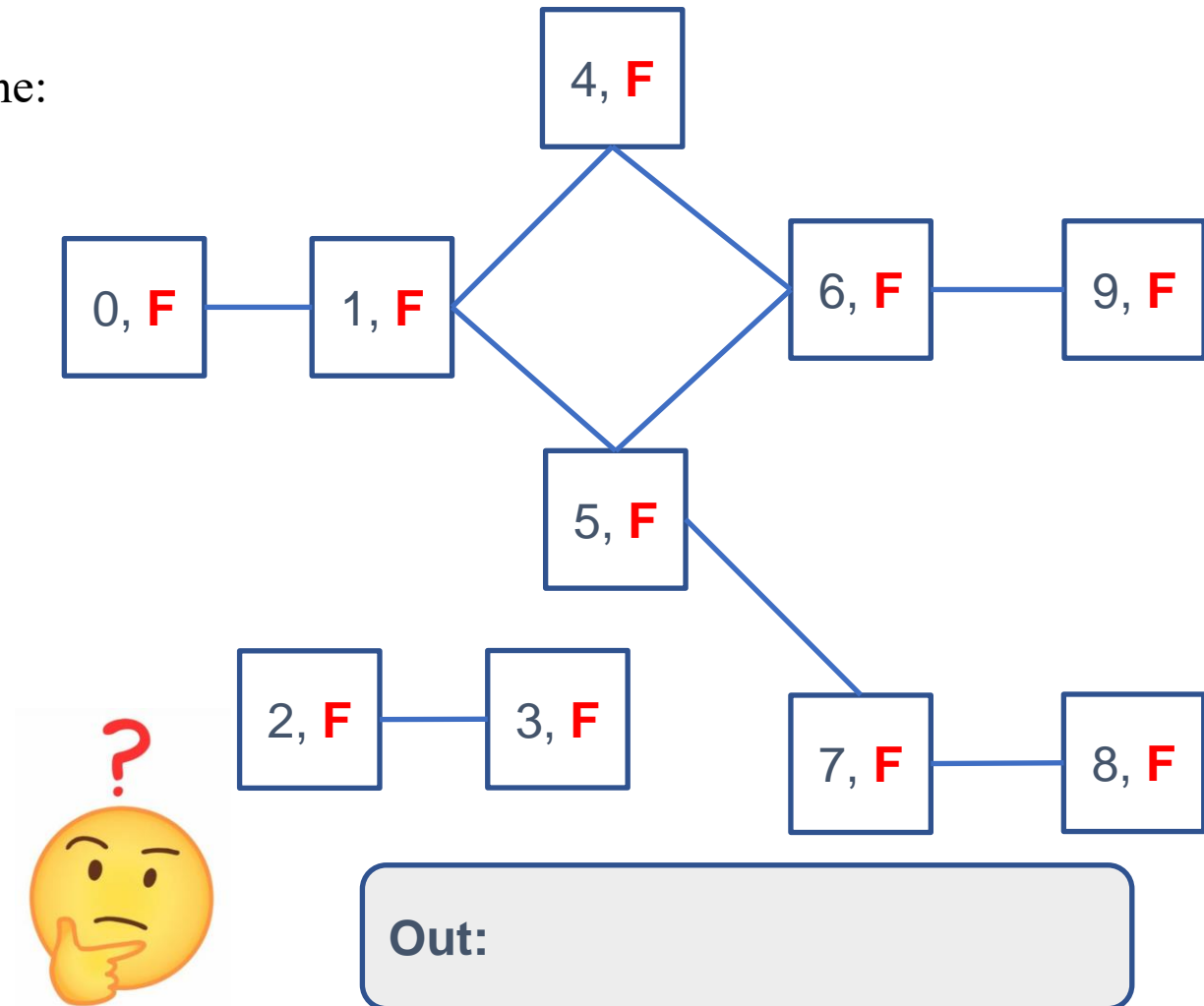
```
• class undi_graph():
•     def __DFTHelp(self, visited: list, v: int) -> None:
•         if not visited[v]:
•             visited[v] = True
•             print(v)
•             for w in self.neighbor[v]:
•                 self.__DFTHelp(visited, w)
•
•     def DFT(self) -> None:
•         if self.V:
•             visited = {}
•             for v in self.V:
•                 visited[v] = False
•                 for v in self.V:
•                     self.__DFTHelp(visited, v)
```

**All vertices are  
visited!!!**



# Depth-First Traversal – Postorder

```
class undi_graph():  
    def __DFTHelp(self, visited: list, v: int) -> None:  
        if not visited[v]:  
            visited[v] = True  
            for w in self.neighbor[v]:  
                self.__DFTHelp(visited, w)  
            print(v)  
  
    def DFT(self) -> None:  
        if self.V:  
            visited = {}  
            for v in self.V:  
                visited[v] = False  
            for v in self.V:  
                self.__DFTHelp(visited, v)
```

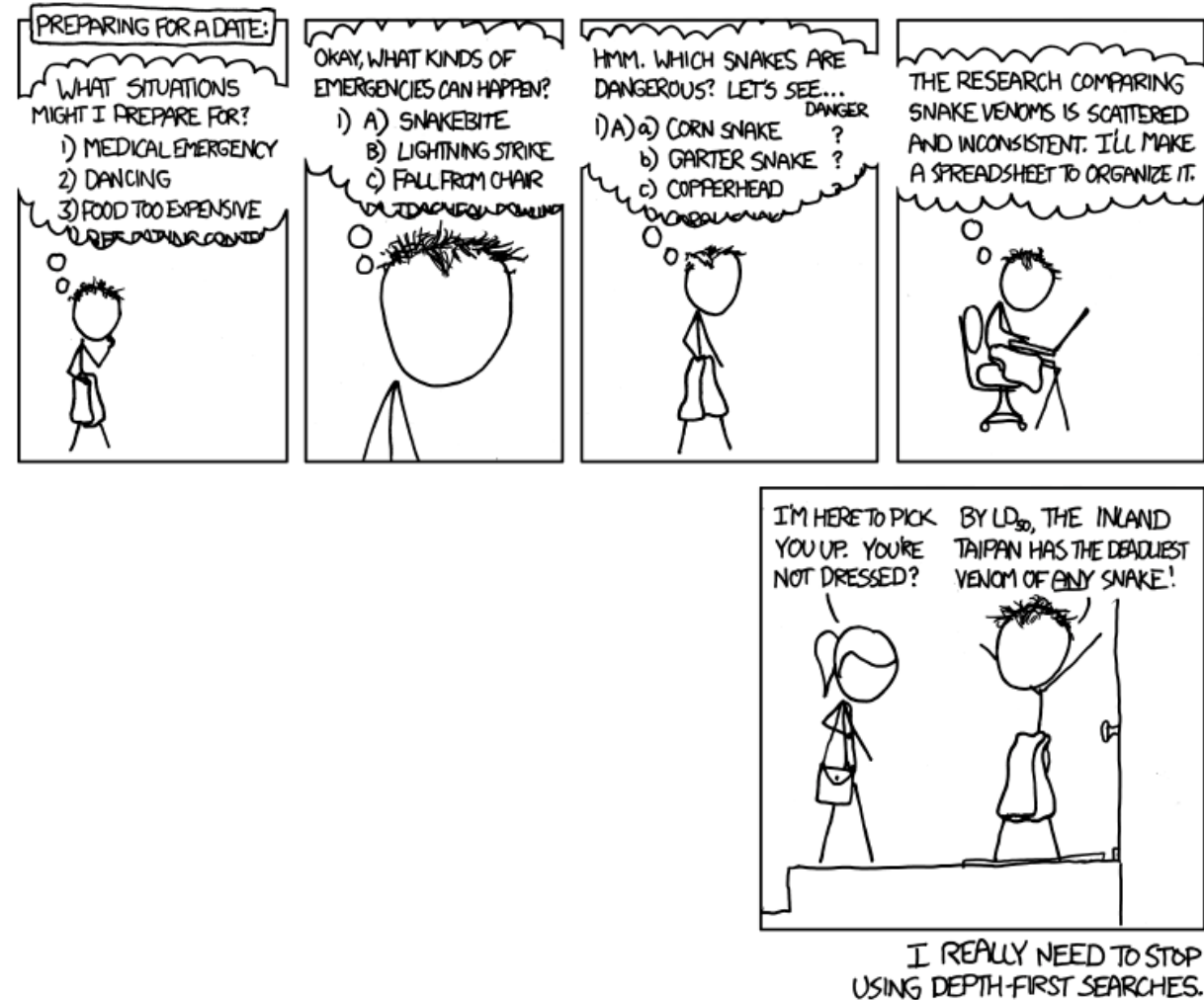


# Depth-First Traversal

- When you want to know if two vertices are **connected**
- When you want to know if all vertices in a graph are **connected**
- When you want to know how many **disjoint islands** are in a graph
- When you want to know if a graph has a **cycle**

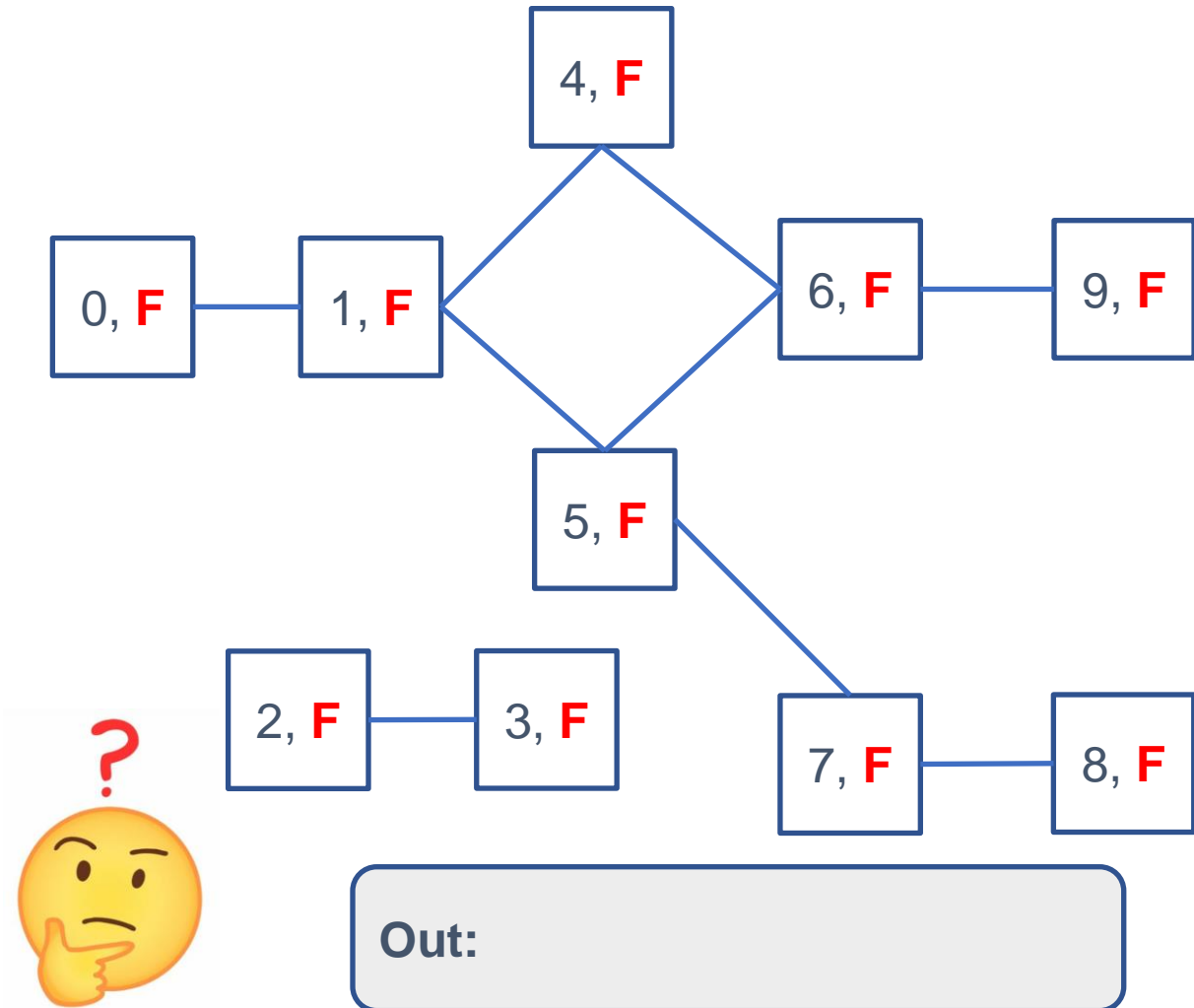
# Depth-First Traversal

- <https://xkcd.com/761/>



# Breadth-First Traversal

```
• class undi_graph():  
•     def BFT(self) -> None:  
•         if self.V:  
•             visited = {}  
•             for v in self.V:  
•                 visited[v] = False  
•             q = deque([])  
•             for v in self.V:  
•                 q.append(v)  
•                 while q:  
•                     v = q.popleft()  
•                     if not visited[v]:  
•                         visited[v] = True  
•                         for w in self.neighbor[v]:  
•                             q.append(w)
```

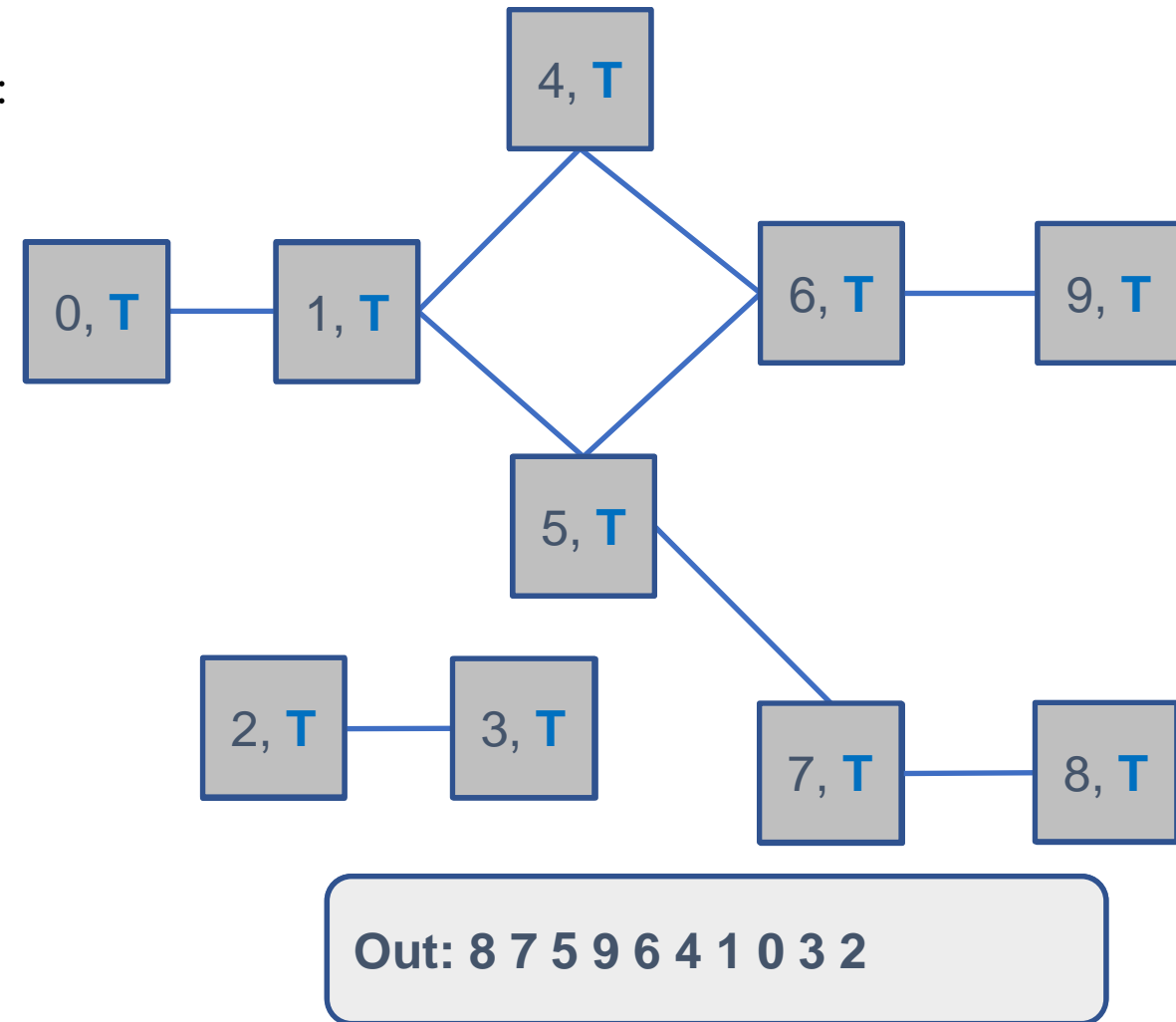


Discuss with your peers!

- DFT postorder and BFT -

# Depth-First Traversal – Postorder

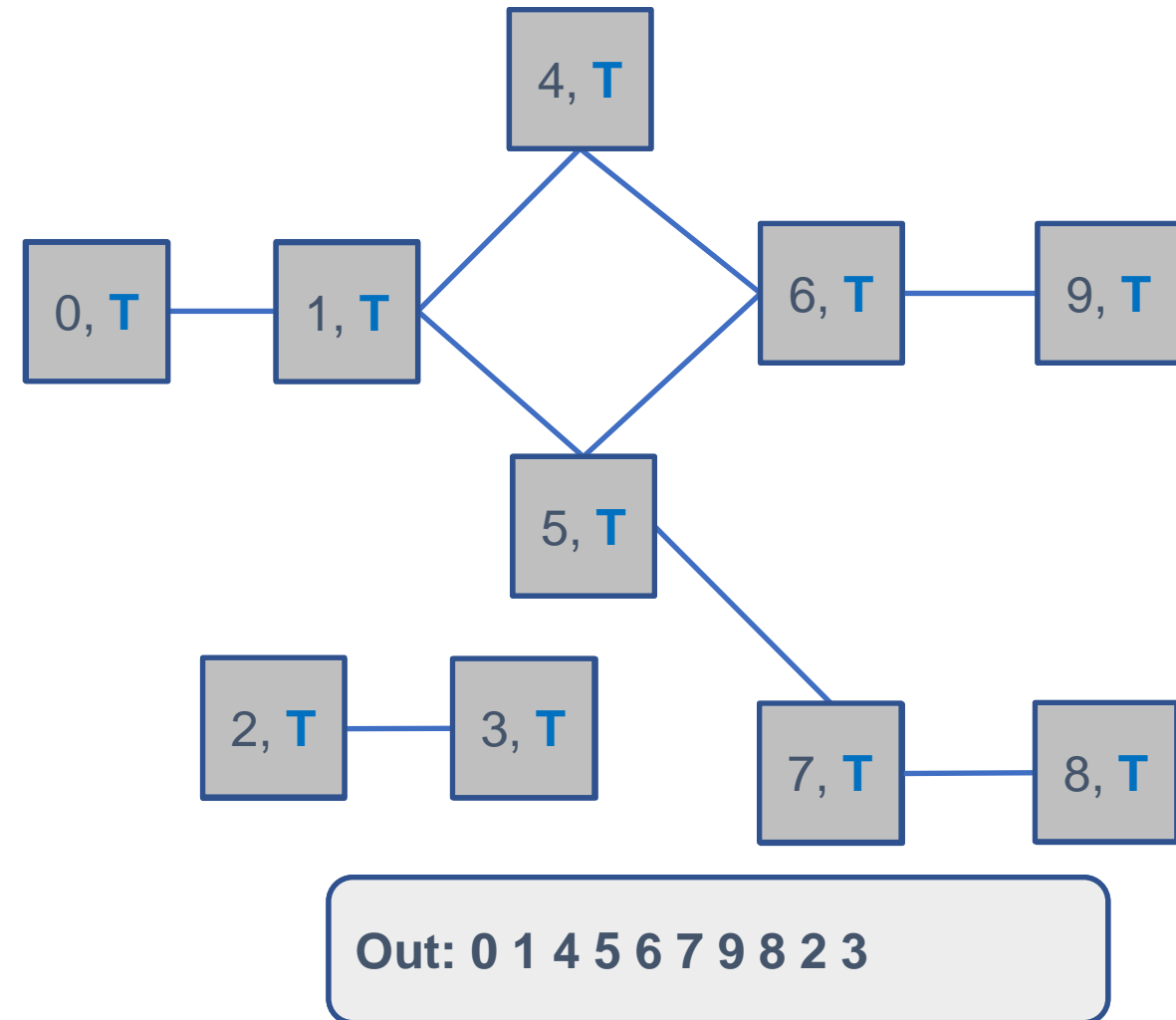
```
class undi_graph():  
    def __DFTHelp(self, visited: list, v: int) -> None:  
        if not visited[v]:  
            visited[v] = True  
            for w in self.neighbor[v]:  
                self.__DFTHelp(visited, w)  
            print(v)  
  
    def DFT(self) -> None:  
        if self.V:  
            visited = {}  
            for v in self.V:  
                visited[v] = False  
            for v in self.V:  
                self.__DFTHelp(visited, v)
```





# Breadth-First Traversal

```
• class undi_graph():
•     def BFT(self) -> None:
•         if self.V:
•             visited = {}
•             for v in self.V:
•                 visited[v] = False
•             q = deque([])
•             for v in self.V:
•                 q.append(v)
•                 while q:
•                     v.popleft()
•                     if not visited[v]:
•                         visited[v] = True
•                         for w in self.neighbor[v]:
•                             q.append(w)
```



# Summary

- Graph is a more general concept than tree
  - Cycle and disconnection
- Graph traversals are a bit different from tree traversals but their core ideas are the same
  - Depth first traversals
  - Breadth first traversals
- Graph traversals are the most common tools for solving graph problems

# Practice for Graphs

Practice 10

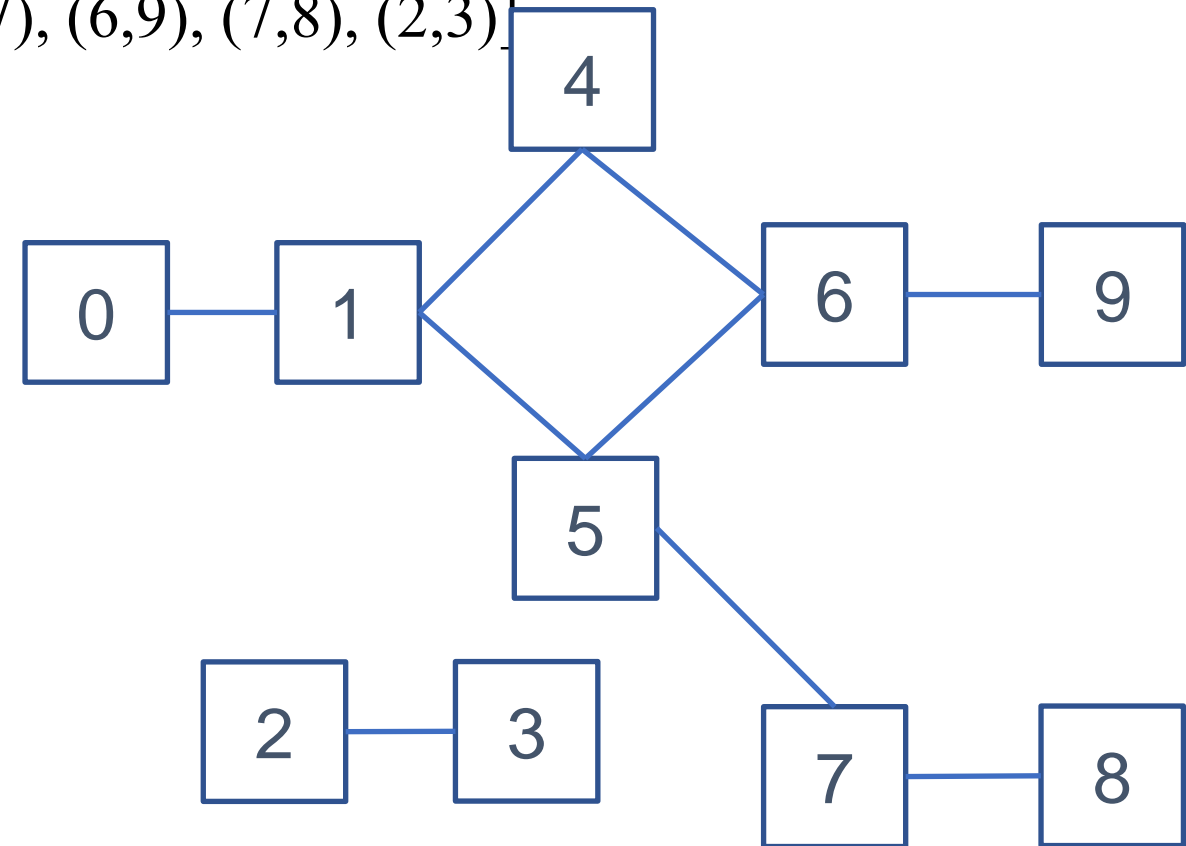
Hyung-Sin Kim



SNU Graduate School of Data Science

# Practice Problem

- Implement post-order DFT for the following undirected graph
  - $V = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$
  - $E = [(0,1), (1,4), (1,5), (4,6), (5,6), (5,7), (6,9), (7,8), (2,3)]$
- Your output when executing DFT
  - 8 7 5 9 6 4 1 0 3 2



Thanks!