# Practice 9

[Lecture 12] Graphs and Traversals

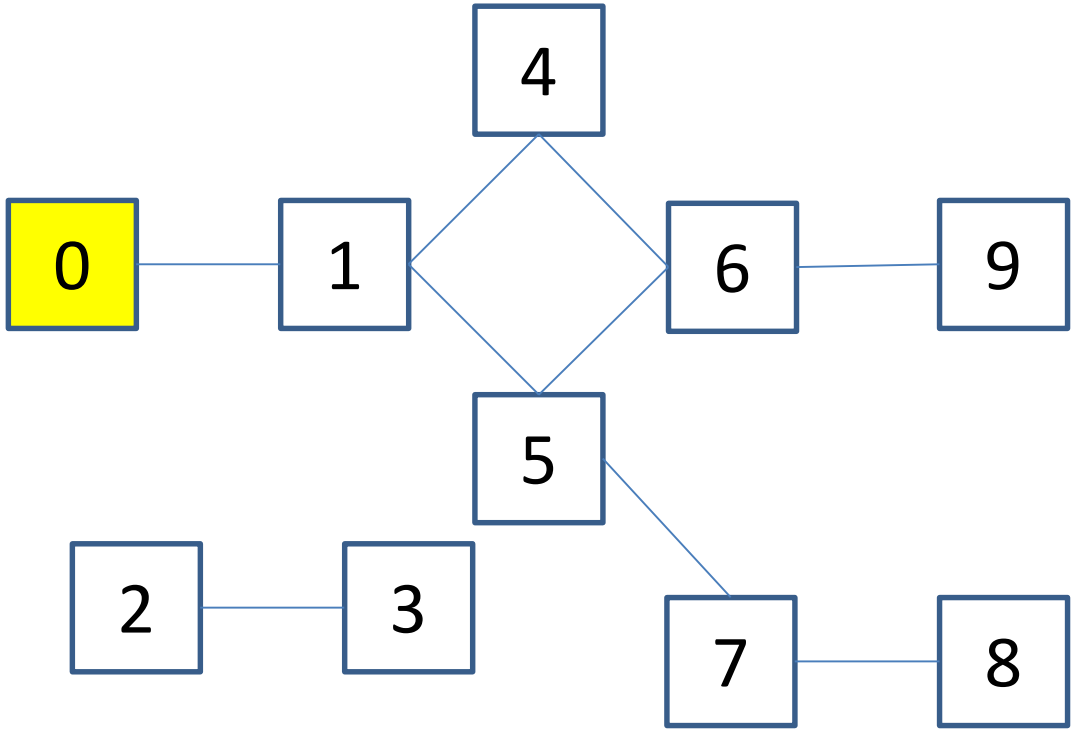

Seoul National University
Graduate School of Data Science

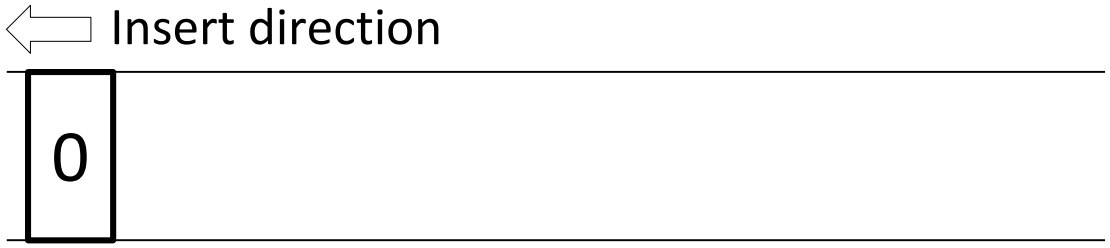
# 01. Exercise

# 1. Implement BFT

**Implement a BFT method for a given graph**

- BFT: Visit all connected nodes level by level
- Use a queue (in Python, use `deque()`)
- Use `visited` as a dictionary that marks visited nodes as `True`

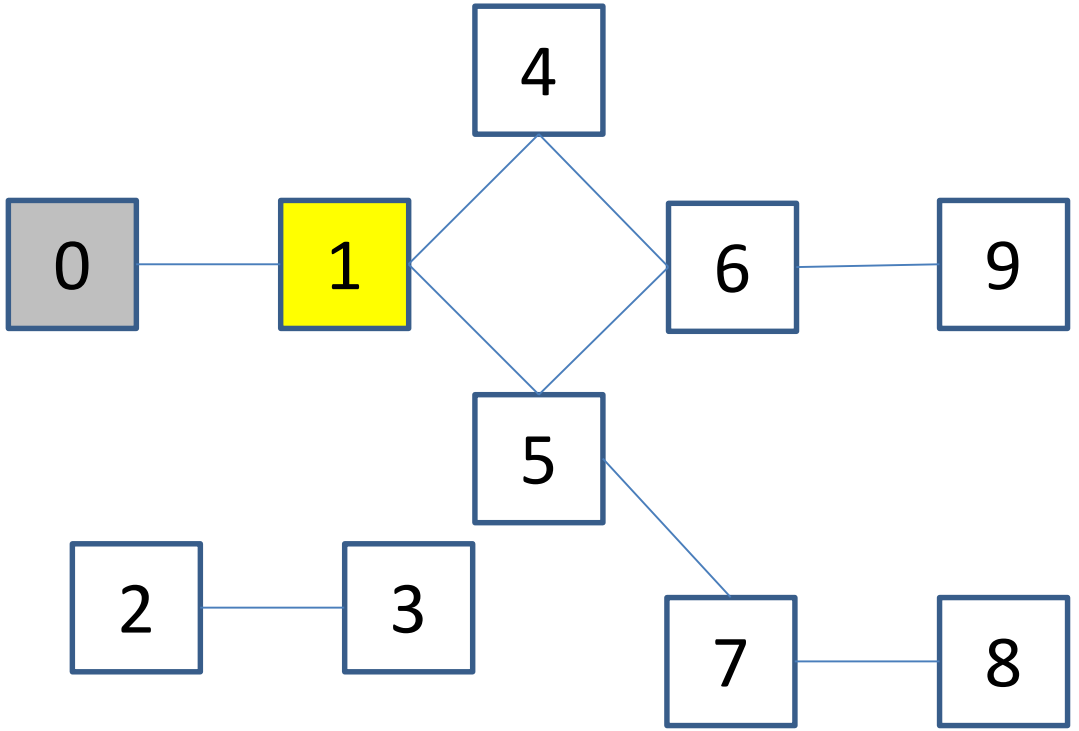


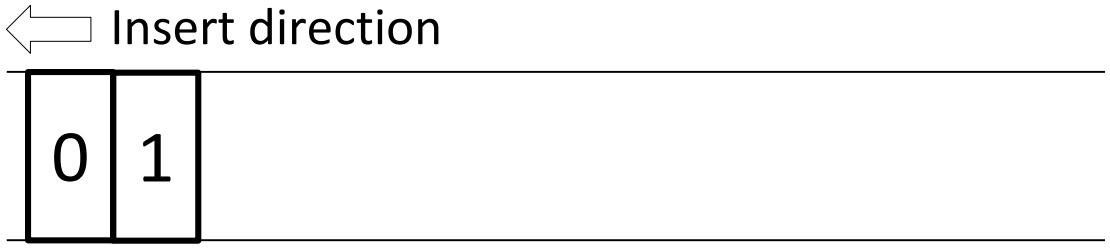



# 1. Implement BFT

**Implement a BFT method for a given graph**

- BFT: Visit all connected nodes level by level
- Use a queue (in Python, use `deque()`)
- Use `visited` as a dictionary that marks visited nodes as `True`

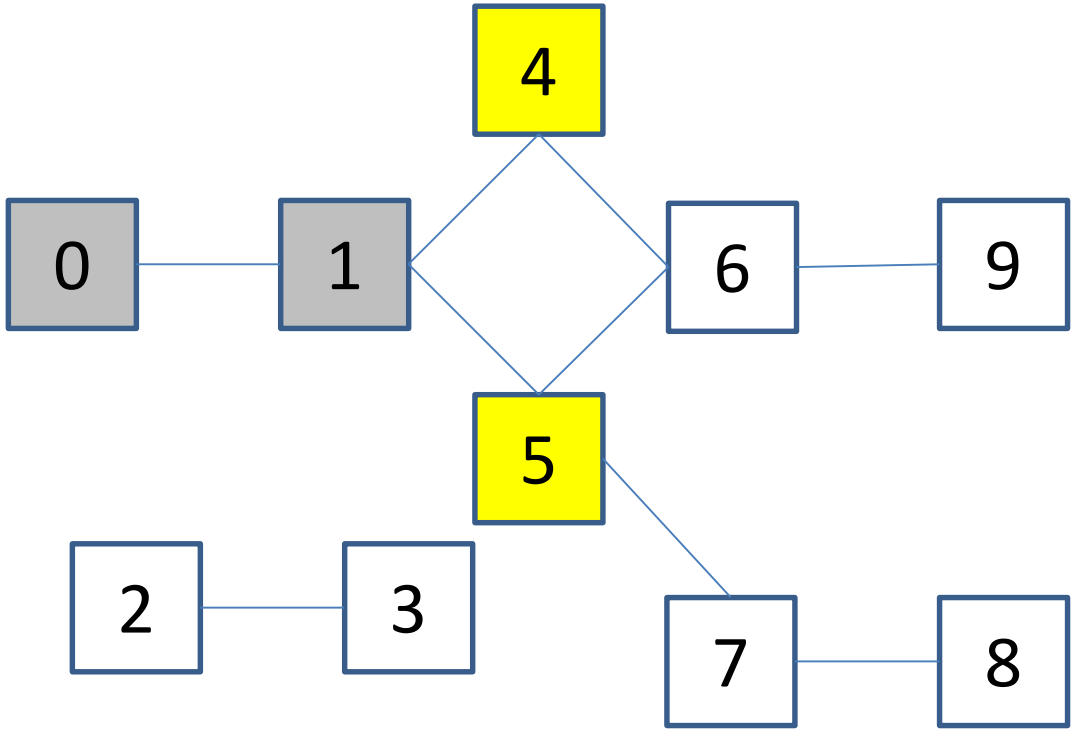


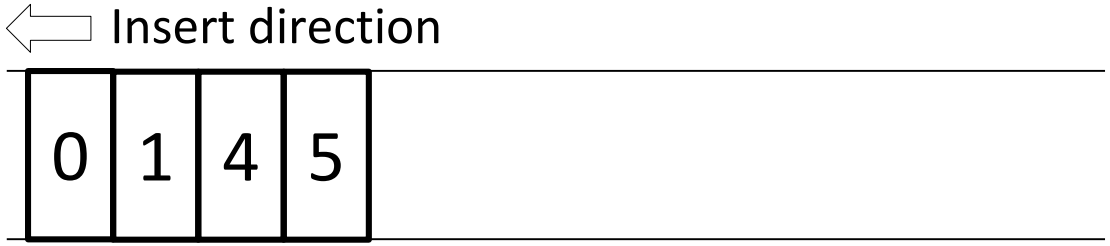



# 1. Implement BFT

**Implement a BFT method for a given graph**

- BFT: Visit all connected nodes level by level
- Use a queue (in Python, use `deque()`)
- Use `visited` as a dictionary that marks visited nodes as `True`

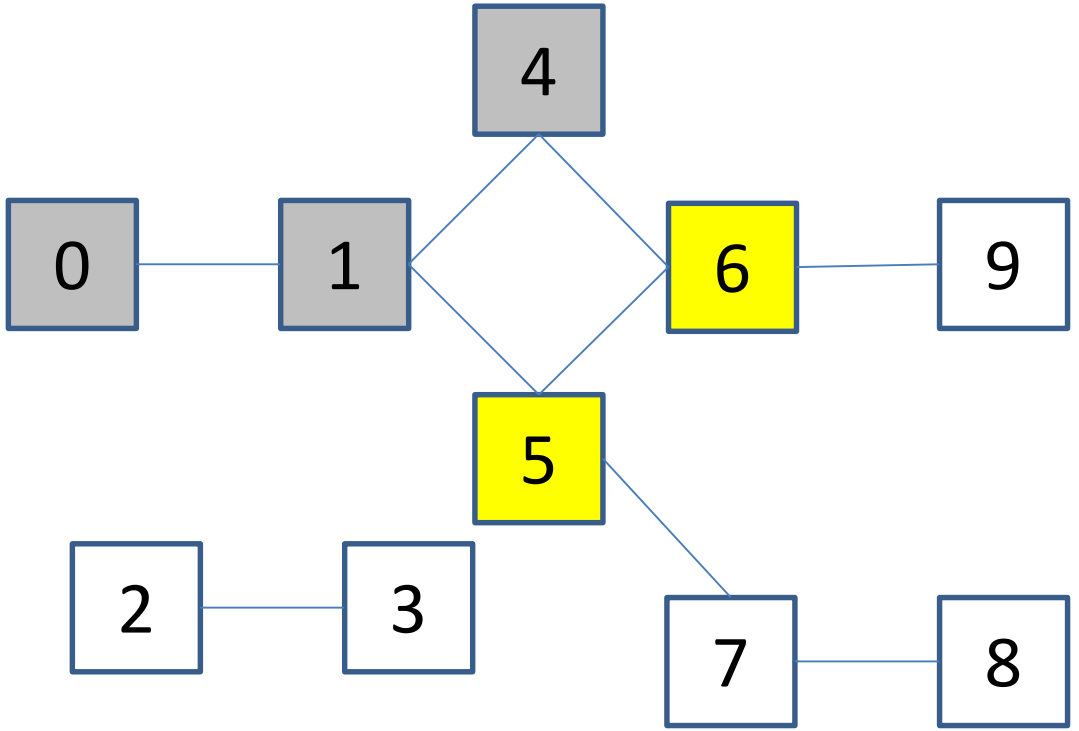


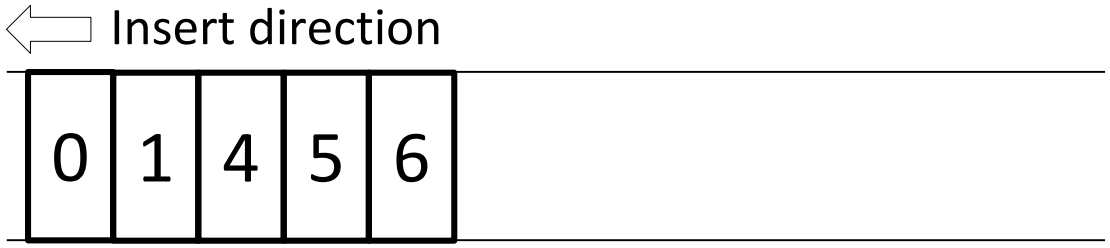



# 1. Implement BFT

**Implement a BFT method for a given graph**

- BFT: Visit all connected nodes level by level
- Use a queue (in Python, use `deque()`)
- Use `visited` as a dictionary that marks visited nodes as `True`

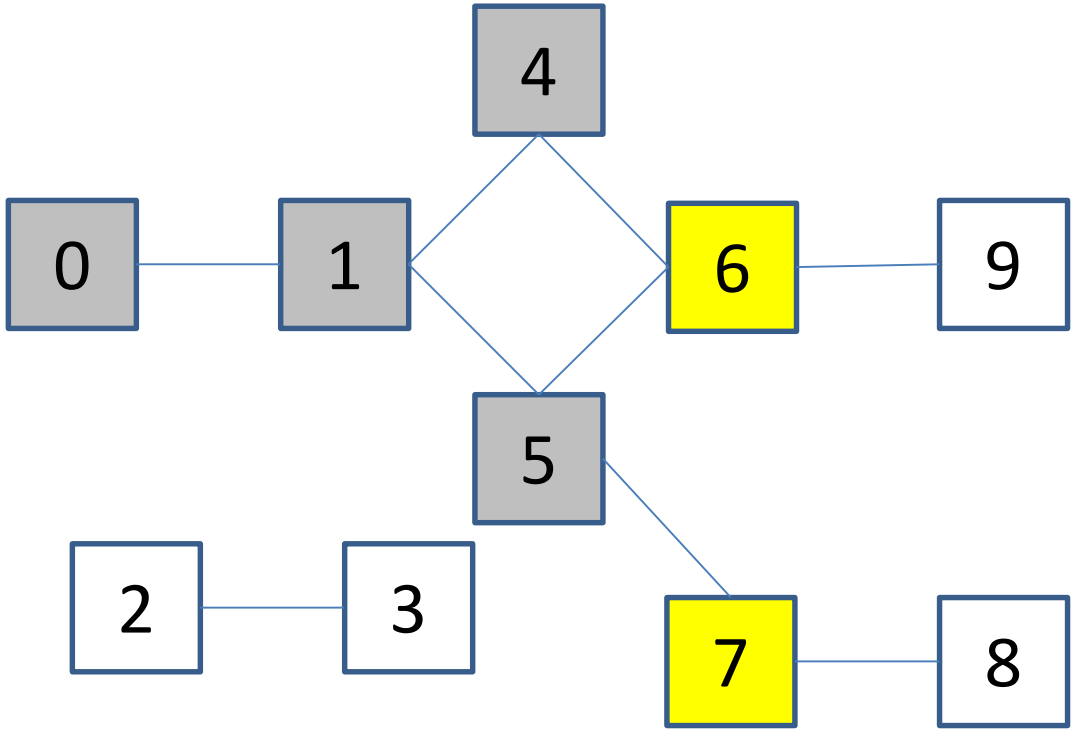


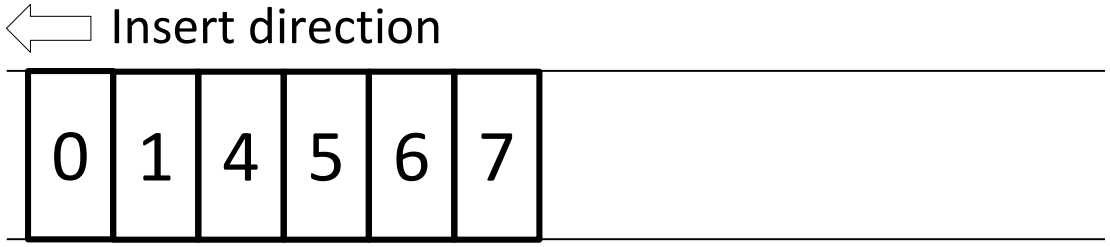



# 1. Implement BFT

**Implement a BFT method for a given graph**

- BFT: Visit all connected nodes level by level
- Use a queue (in Python, use `deque()`)
- Use `visited` as a dictionary that marks visited nodes as `True`

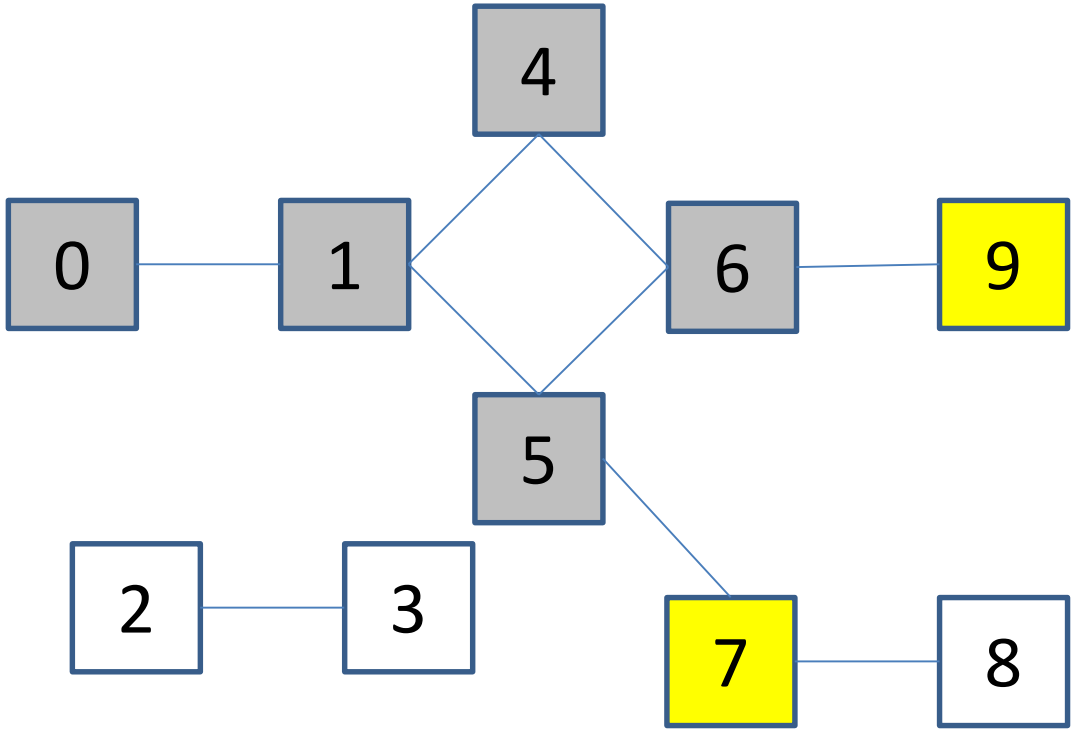


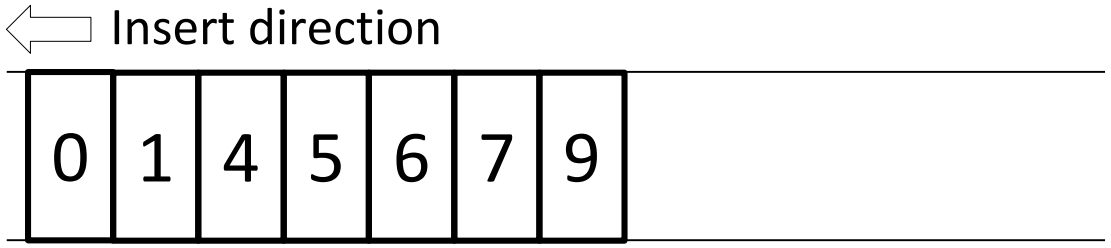



# 1. Implement BFT

**Implement a BFT method for a given graph**

- BFT: Visit all connected nodes level by level
- Use a queue (in Python, use `deque()`)
- Use `visited` as a dictionary that marks visited nodes as `True`

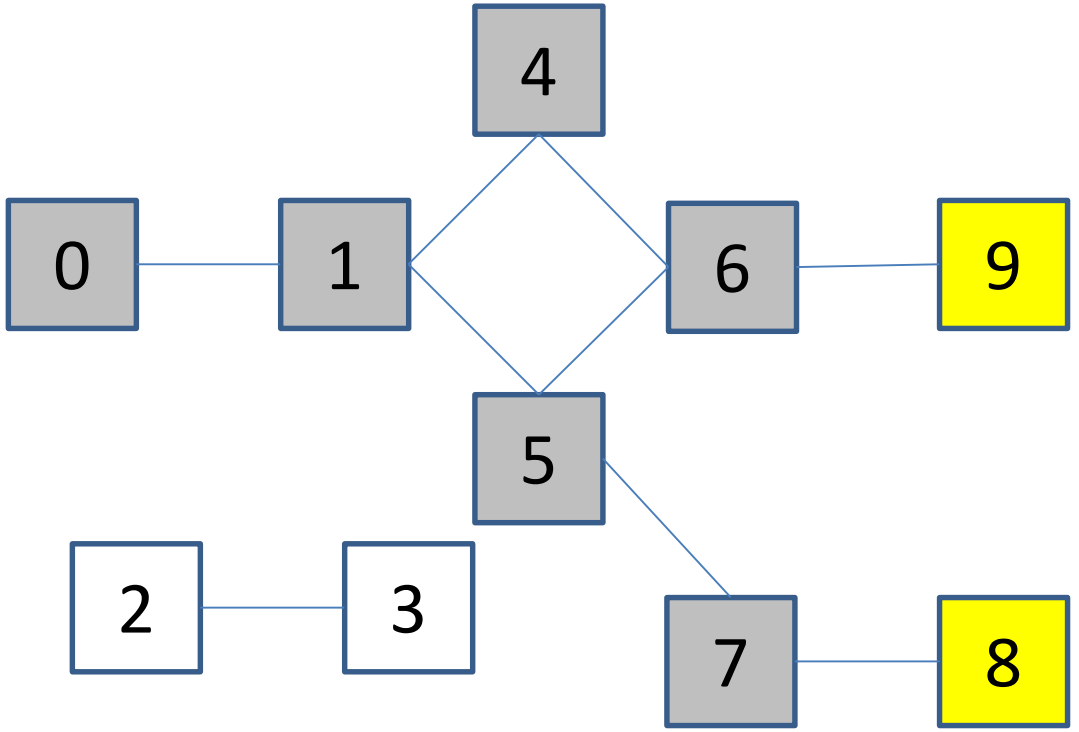


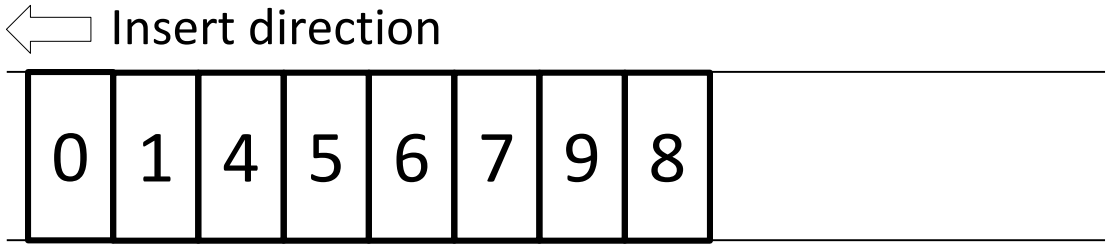



# 1. Implement BFT

**Implement a BFT method for a given graph**

- BFT: Visit all connected nodes level by level
- Use a queue (in Python, use `deque()`)
- Use `visited` as a dictionary that marks visited nodes as `True`

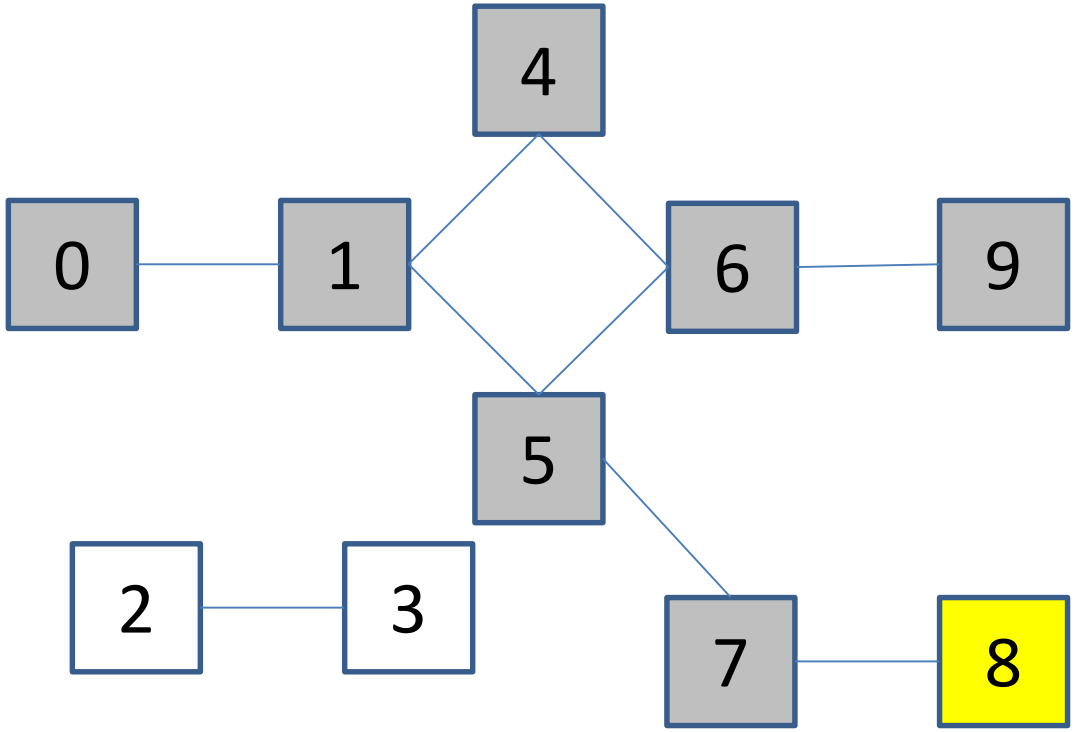


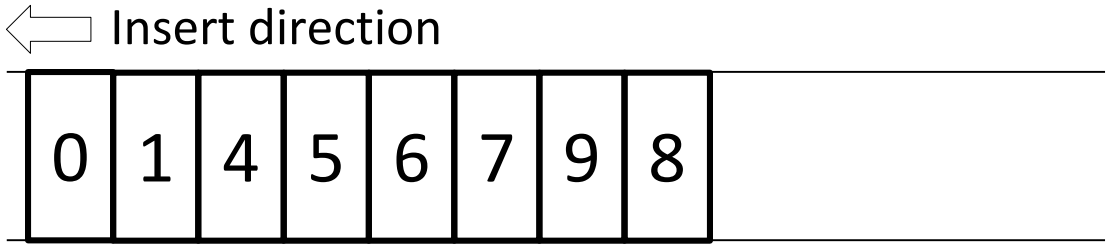



# 1. Implement BFT

**Implement a BFT method for a given graph**

- BFT: Visit all connected nodes level by level
- Use a queue (in Python, use `deque()`)
- Use `visited` as a dictionary that marks visited nodes as `True`

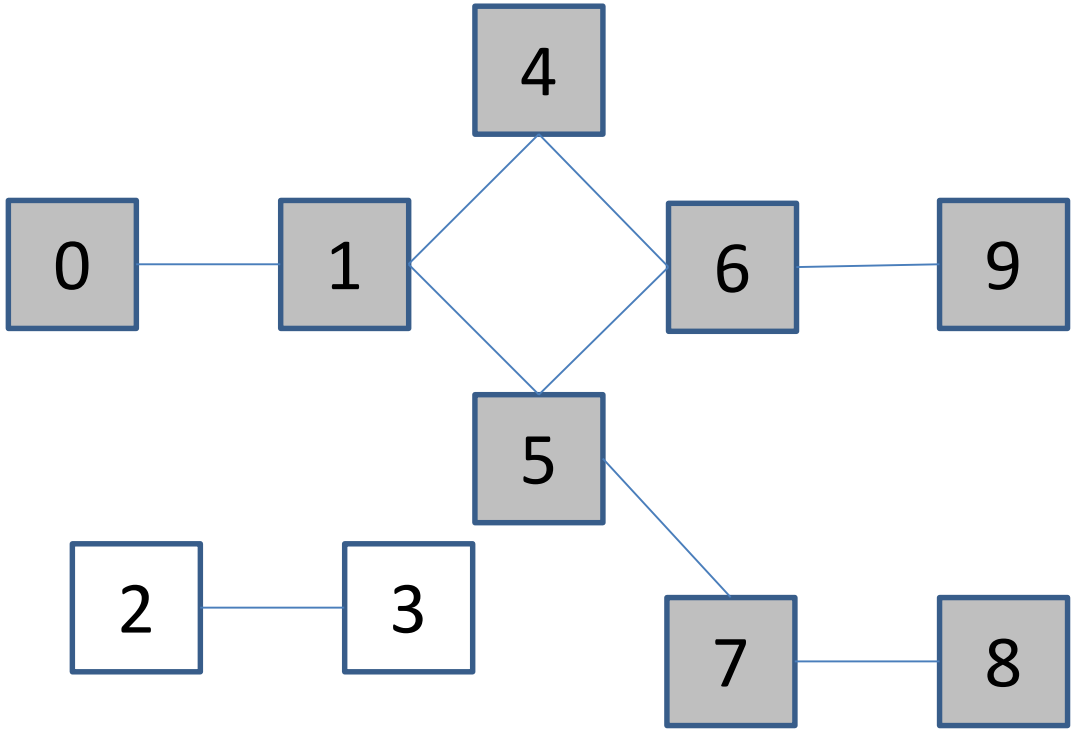


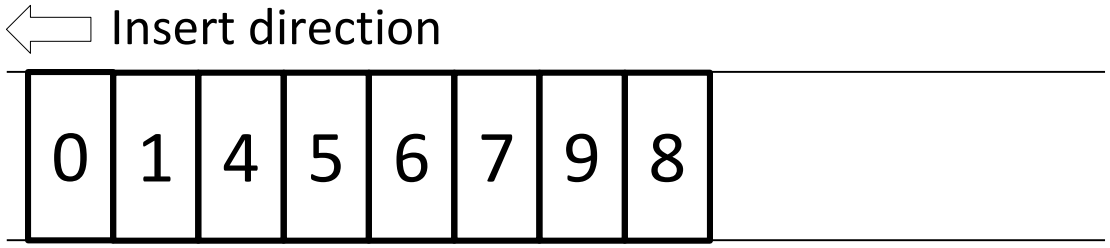



# 1. Implement BFT

**Implement a BFT method for a given graph**

- BFT: Visit all connected nodes level by level
- Use a queue (in Python, use `deque()`)
- Use `visited` as a dictionary that marks visited nodes as `True`

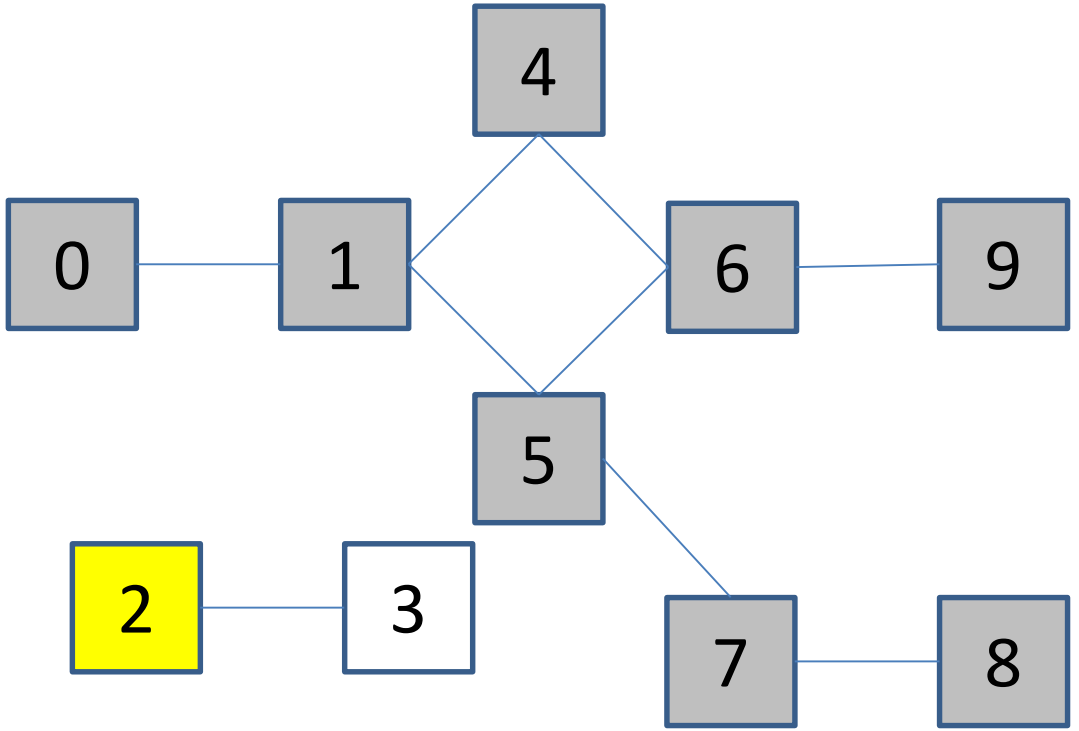


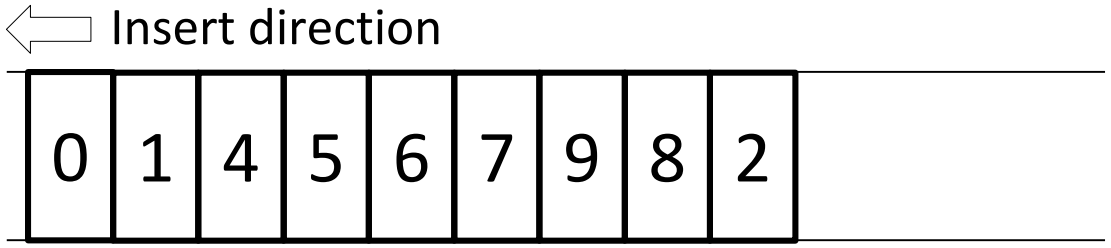



# 1. Implement BFT

**Implement a BFT method for a given graph**

- BFT: Visit all connected nodes level by level
- Use a queue (in Python, use `deque()`)
- Use `visited` as a dictionary that marks visited nodes as `True`

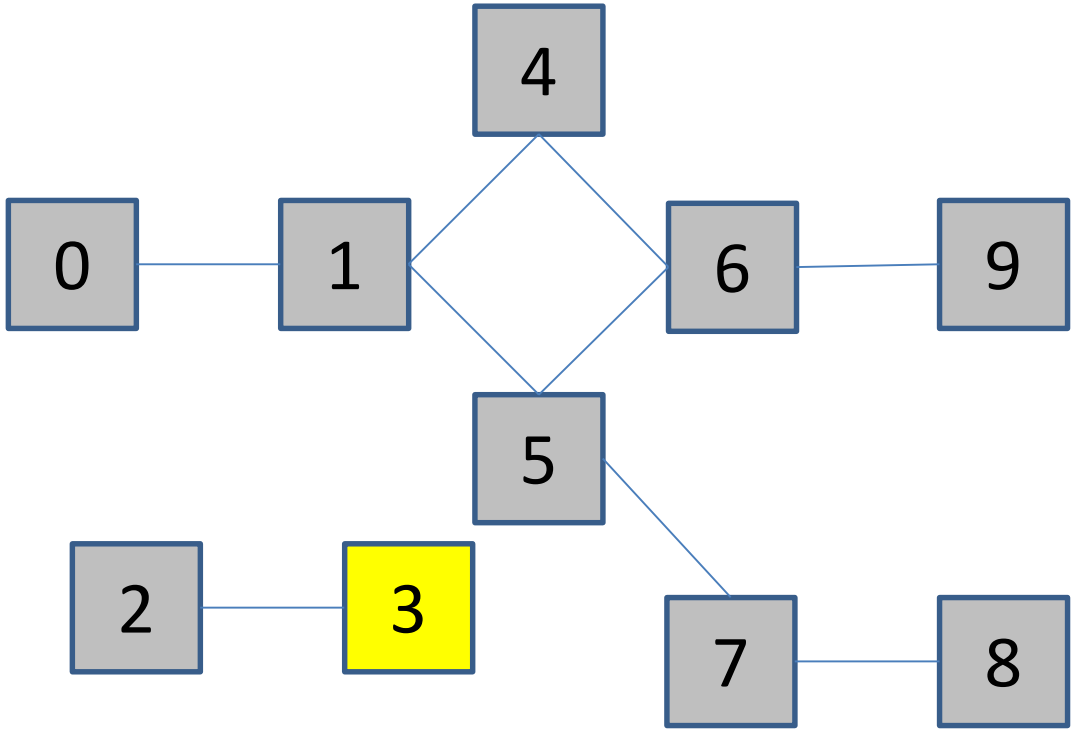


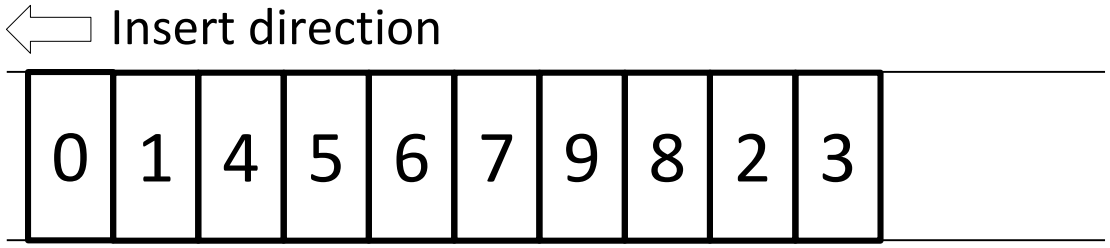



# 1. Implement BFT

**Implement a BFT method for a given graph**

- BFT: Visit all connected nodes level by level
- Use a queue (in Python, use `deque()`)
- Use `visited` as a dictionary that marks visited nodes as `True`

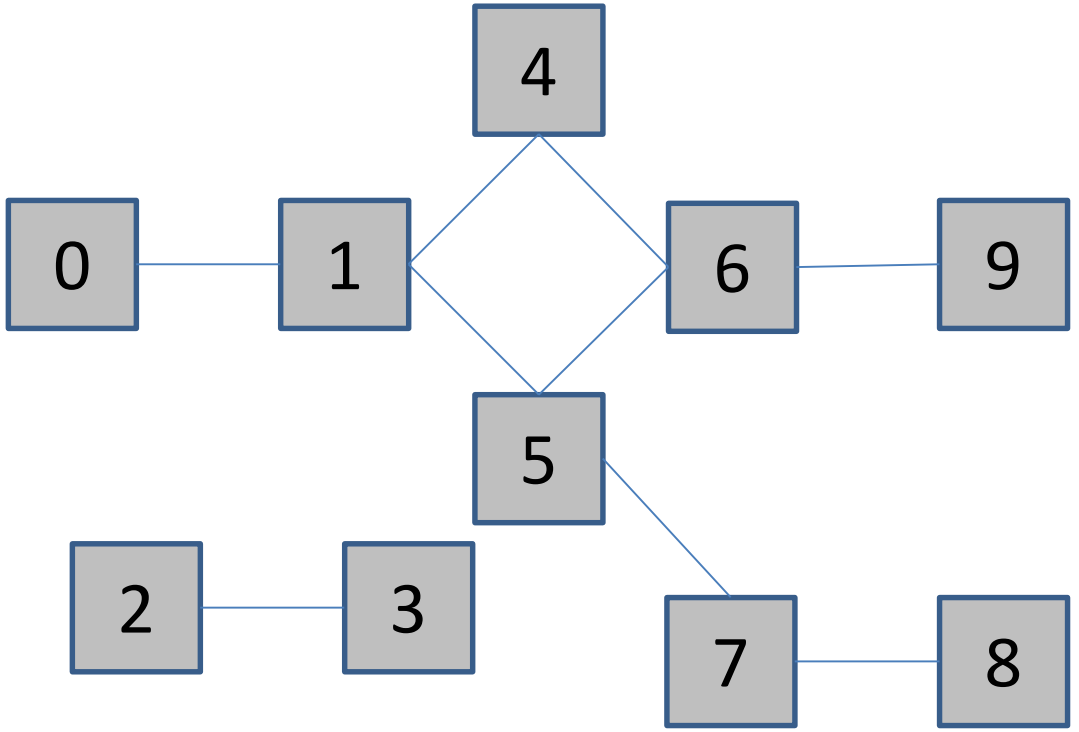


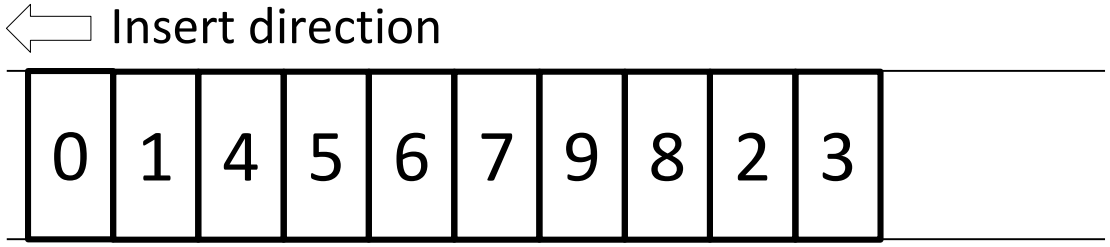



# 1. Implement BFT

**Implement a BFT method for a given graph**

- BFT: Visit all connected nodes level by level
- Use a queue (in Python, use `deque()`)
- Use `visited` as a dictionary that marks visited nodes as `True`

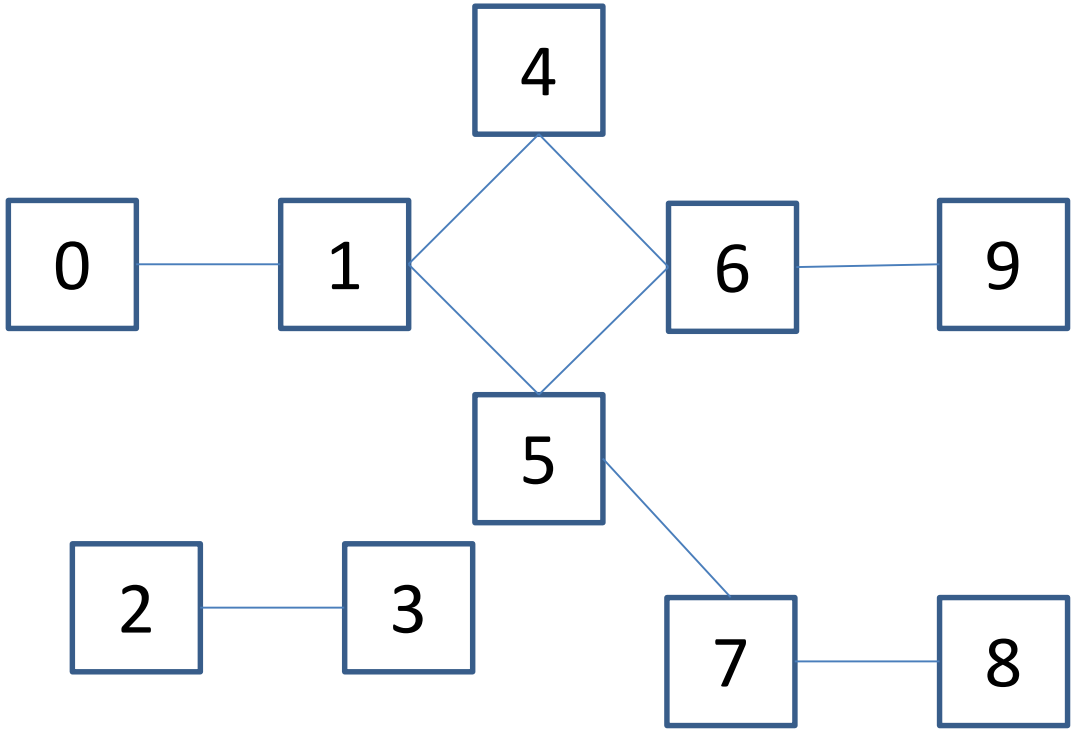


# 2. Implement DFT Post-order

**Implement a post-order DFT function for a given graph**

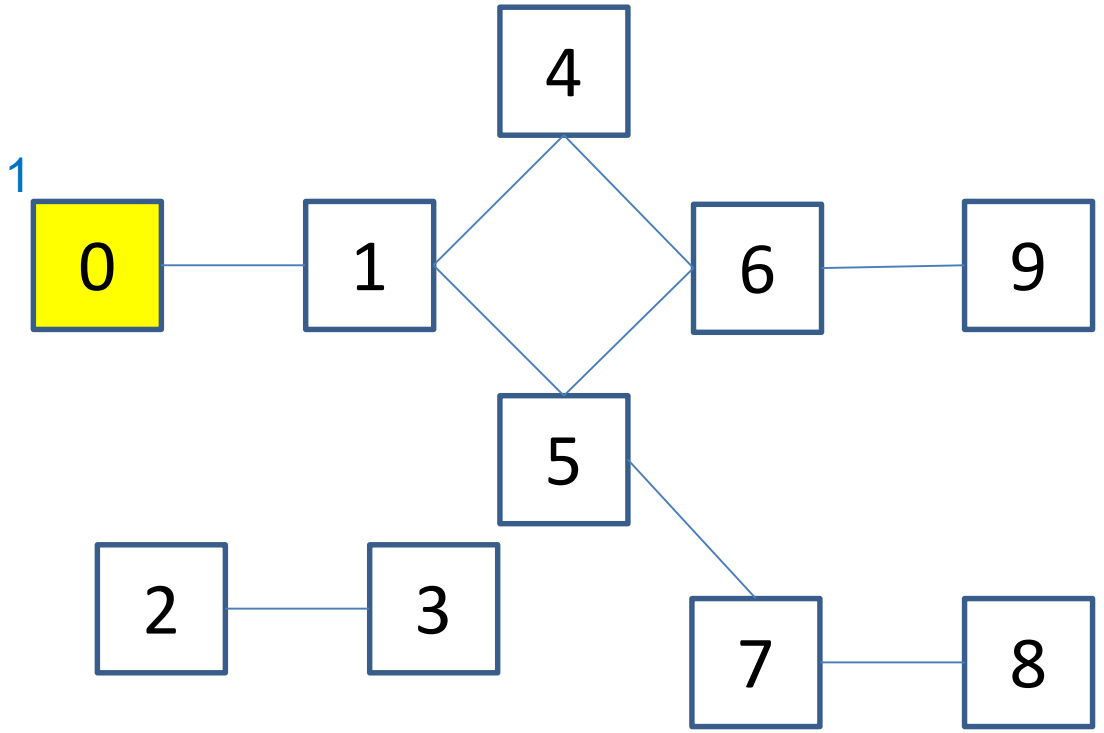
- Use recursion for this task
- Use `visited` as a dictionary that marks visited nodes as `True`
- Code for `DFT()` is given, complete the recursive `__DFTHelp()` method.



# 2. Implement DFT Post-order

**Implement a post-order DFT function for a given graph**

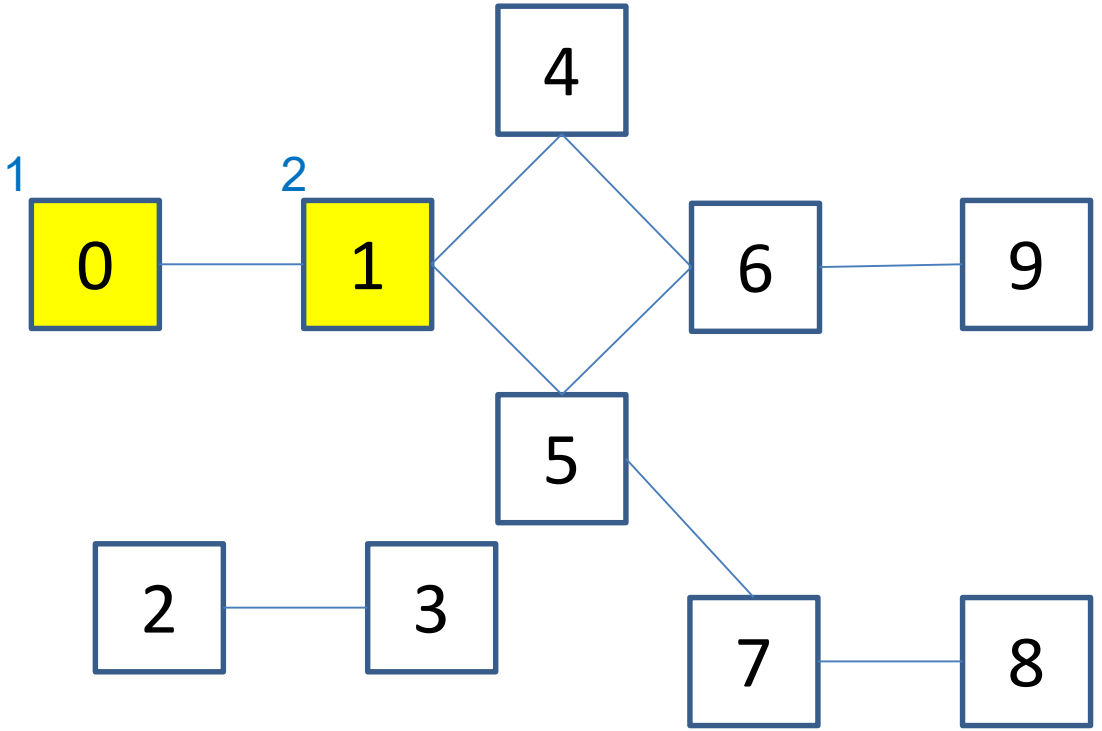
- Use recursion for this task
- Use `visited` as a dictionary that marks visited nodes as `True`
- Code for `DFT()` is given, complete the recursive `__DFTHelp()` method.



# 2. Implement DFT Post-order

**Implement a post-order DFT function for a given graph**

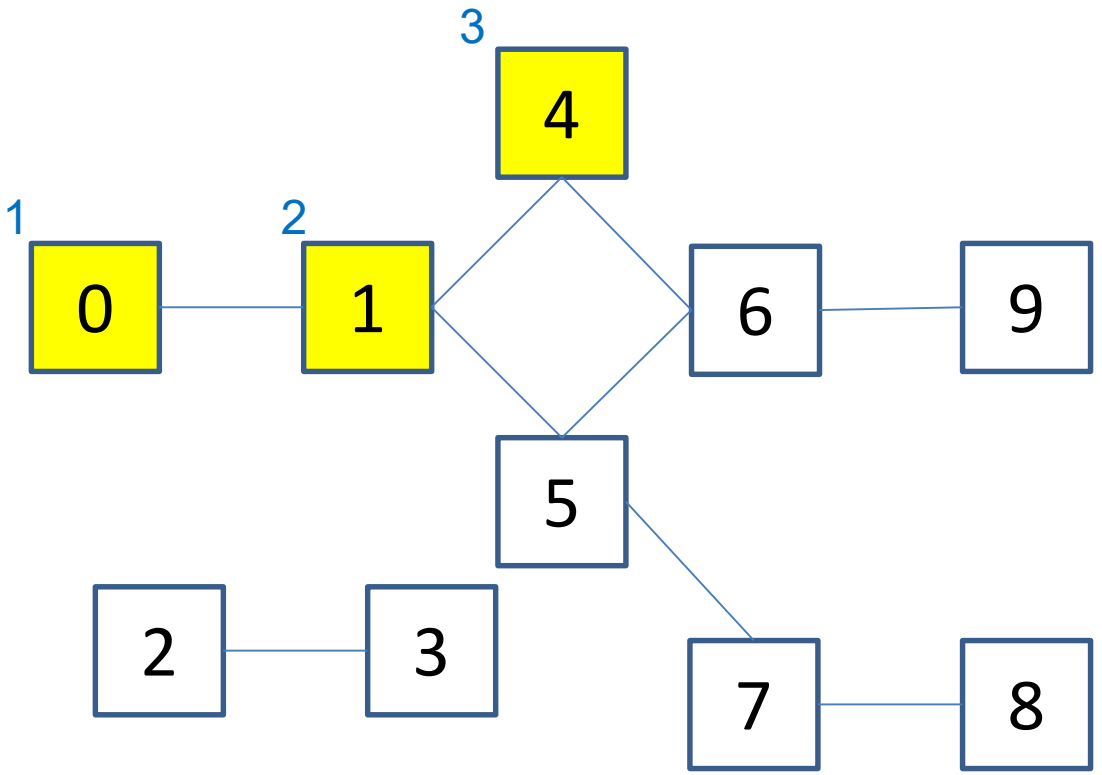
- Use recursion for this task
- Use `visited` as a dictionary that marks visited nodes as `True`
- Code for `DFT()` is given, complete the recursive `__DFTHelp()` method.



# 2. Implement DFT Post-order

**Implement a post-order DFT function for a given graph**

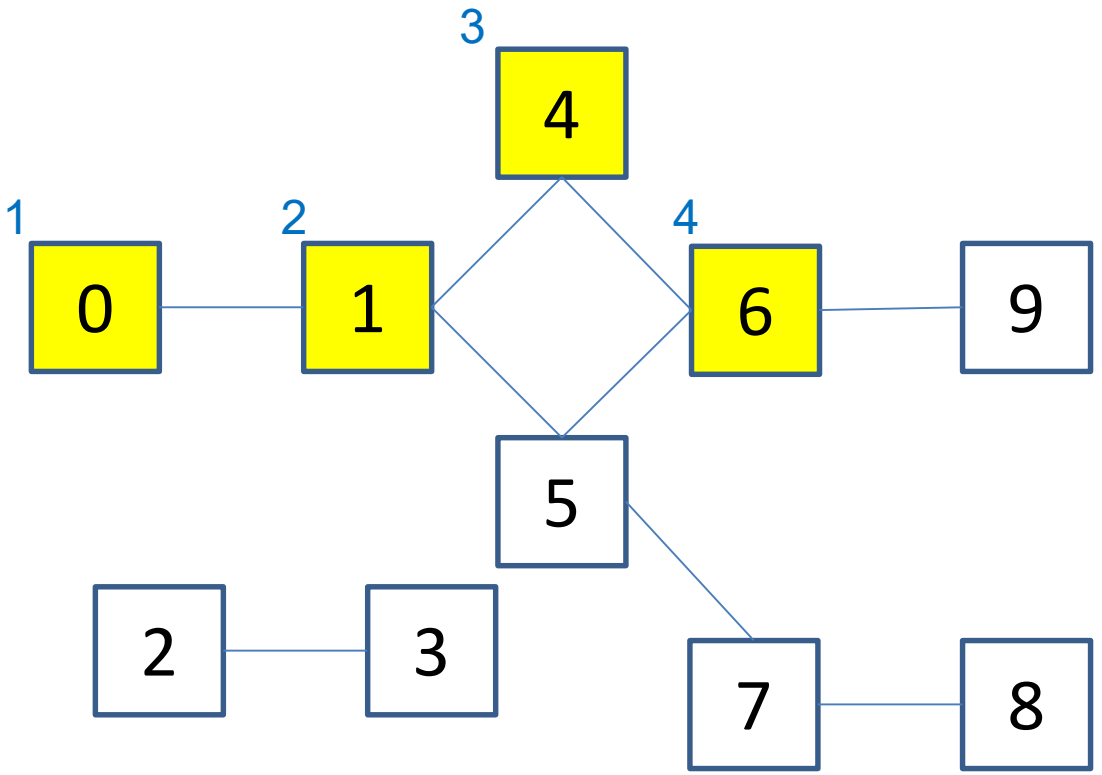
- Use recursion for this task
- Use `visited` as a dictionary that marks visited nodes as `True`
- Code for `DFT()` is given, complete the recursive `__DFTHelp()` method.



# 2. Implement DFT Post-order

**Implement a post-order DFT function for a given graph**

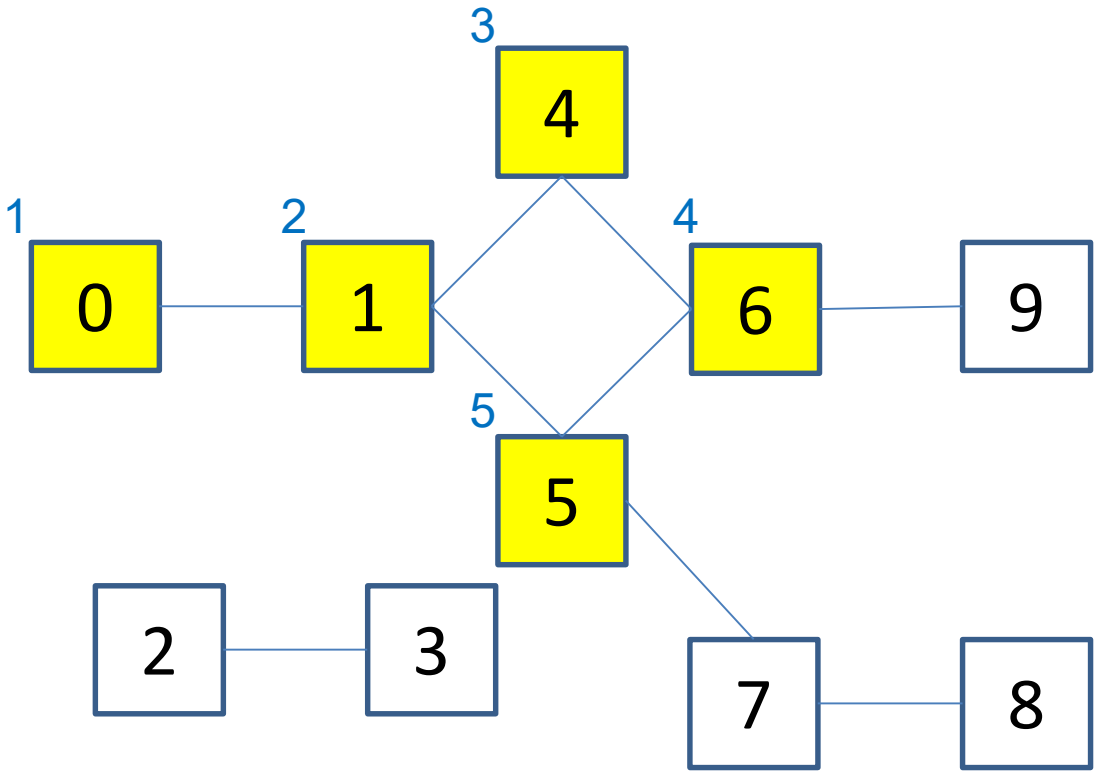
- Use recursion for this task
- Use `visited` as a dictionary that marks visited nodes as `True`
- Code for `DFT()` is given, complete the recursive `__DFTHelp()` method.



# 2. Implement DFT Post-order

**Implement a post-order DFT function for a given graph**

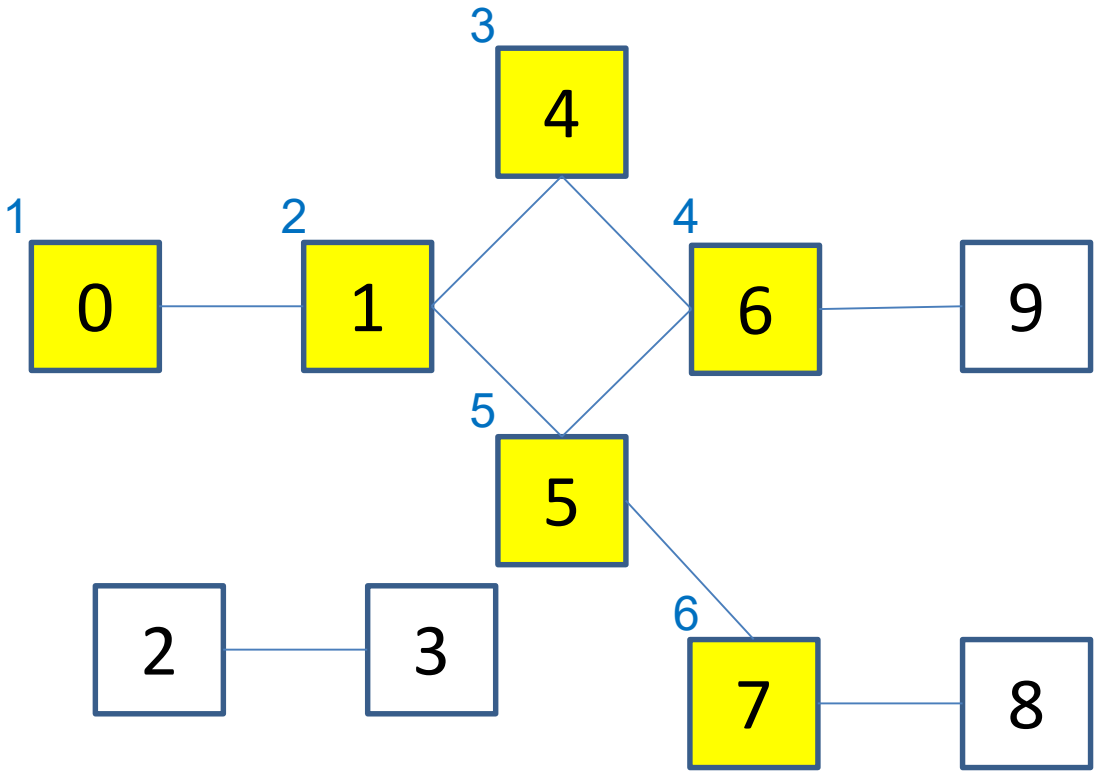
- Use recursion for this task
- Use `visited` as a dictionary that marks visited nodes as `True`
- Code for `DFT()` is given, complete the recursive `__DFTHelp()` method.



# 2. Implement DFT Post-order

**Implement a post-order DFT function for a given graph**

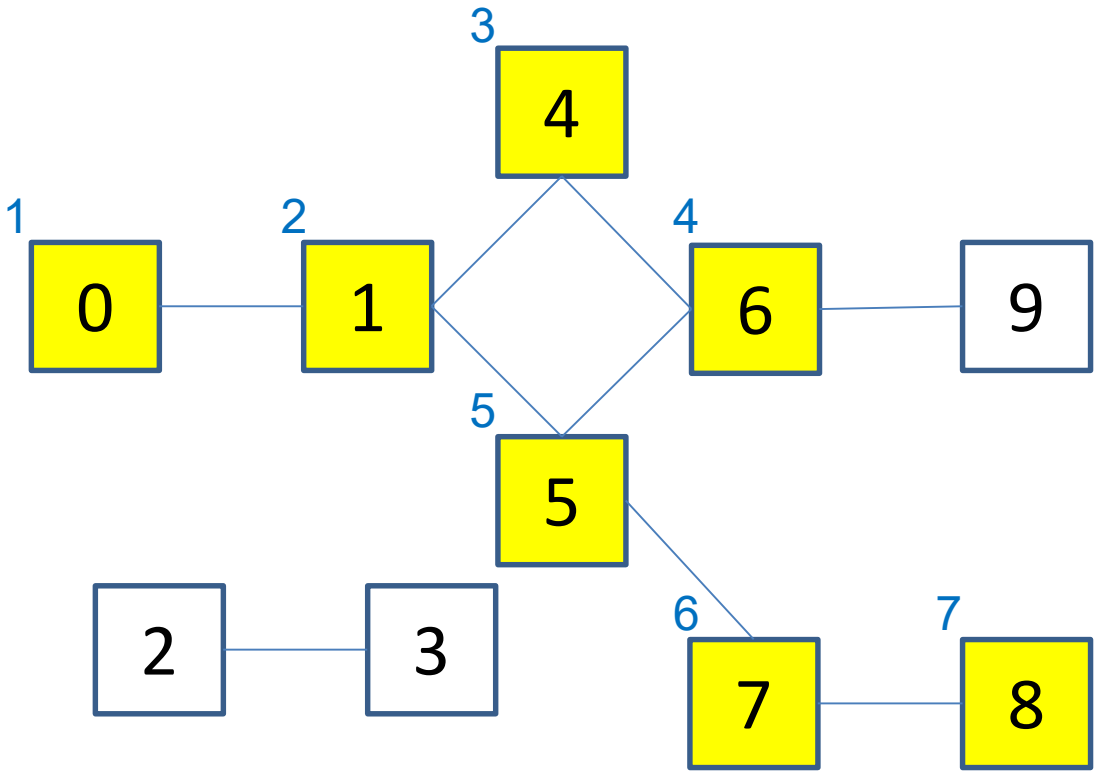
- Use recursion for this task
- Use `visited` as a dictionary that marks visited nodes as `True`
- Code for `DFT()` is given, complete the recursive `__DFTHelp()` method.



# 2. Implement DFT Post-order

**Implement a post-order DFT function for a given graph**

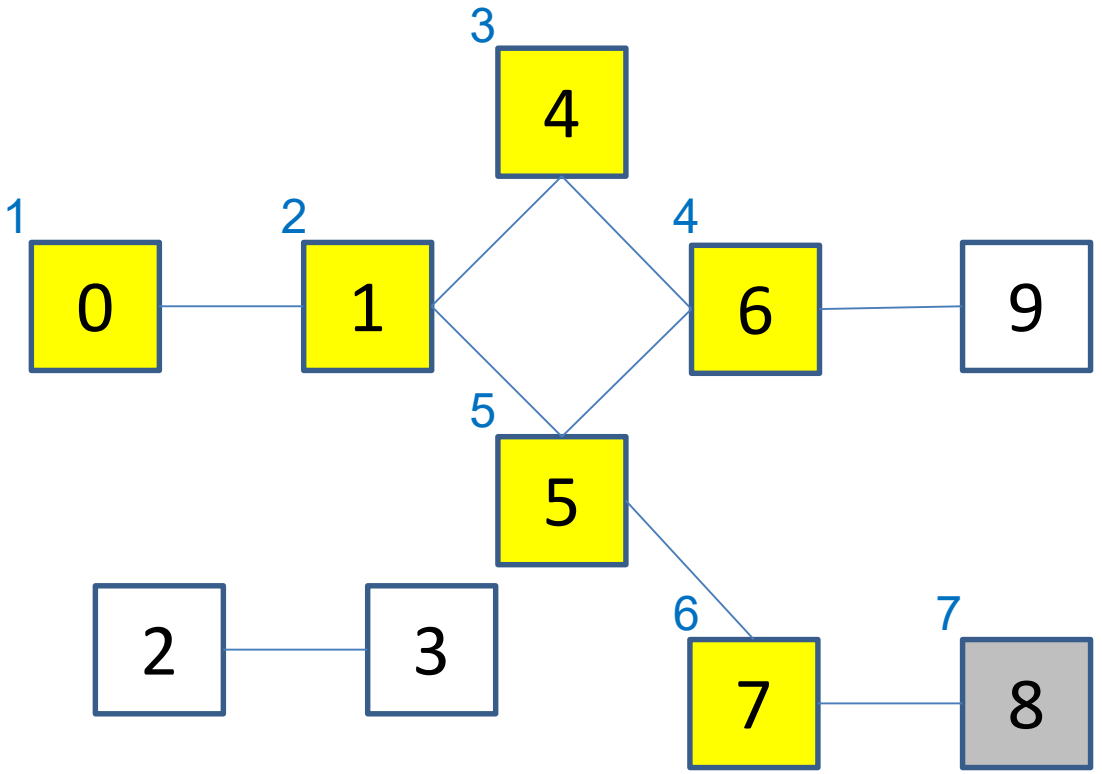
- Use recursion for this task
- Use `visited` as a dictionary that marks visited nodes as `True`
- Code for `DFT()` is given, complete the recursive `__DFTHelp()` method.



# 2. Implement DFT Post-order

**Implement a post-order DFT function for a given graph**

- Use recursion for this task
- Use `visited` as a dictionary that marks visited nodes as `True`
- Code for `DFT()` is given, complete the recursive `__DFTHelp()` method.

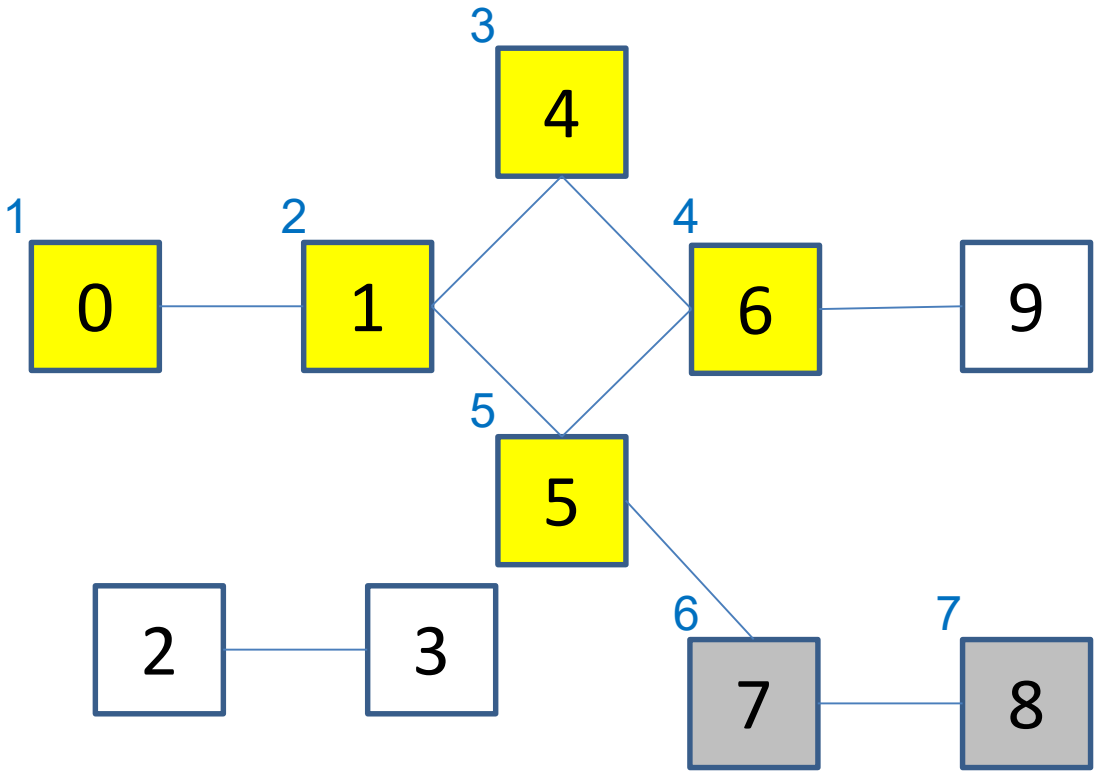



Ans: `8`

# 2. Implement DFT Post-order

**Implement a post-order DFT function for a given graph**

- Use recursion for this task
- Use `visited` as a dictionary that marks visited nodes as `True`
- Code for `DFT()` is given, complete the recursive `__DFTHelp()` method.

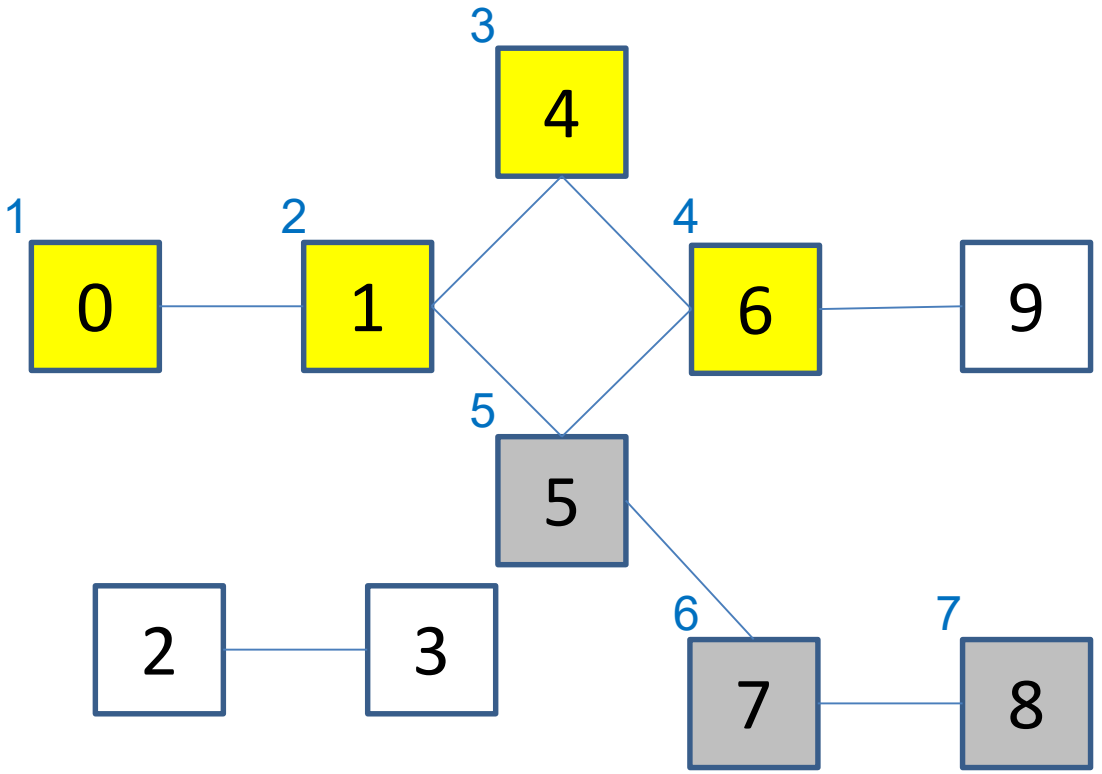



Ans: `8 7`

# 2. Implement DFT Post-order

**Implement a post-order DFT function for a given graph**

- Use recursion for this task
- Use `visited` as a dictionary that marks visited nodes as `True`
- Code for `DFT()` is given, complete the recursive `__DFTHelp()` method.

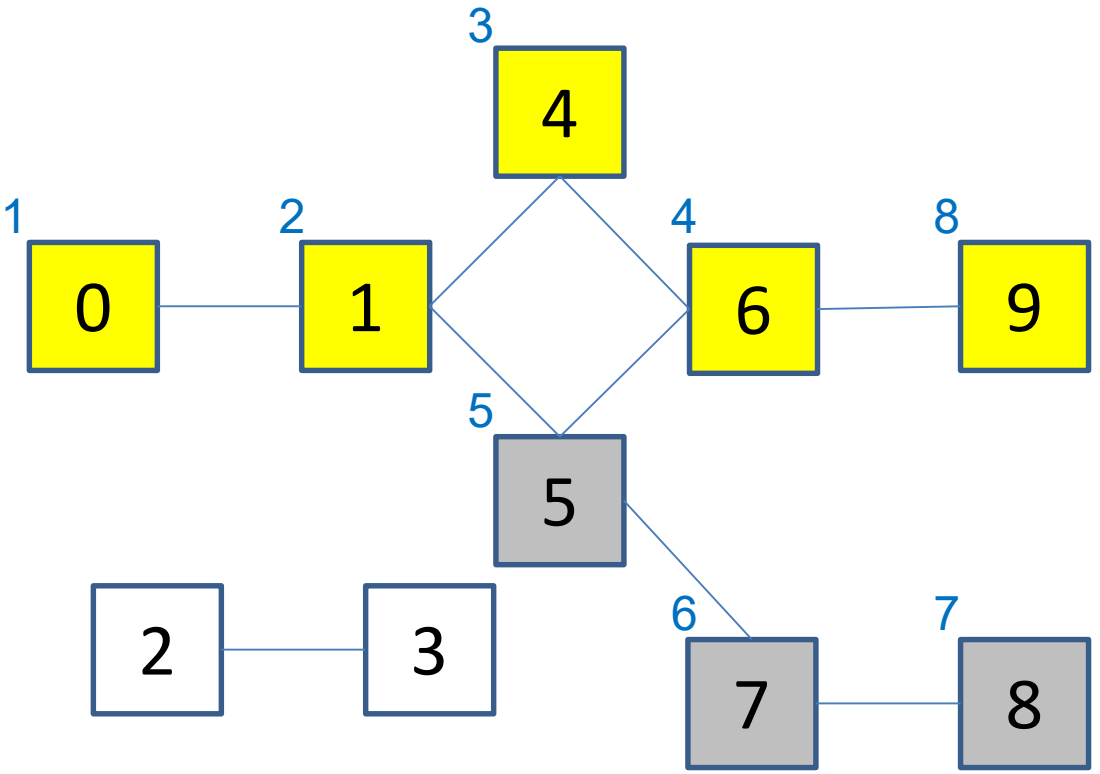



Ans: `8 7 5`

# 2. Implement DFT Post-order

**Implement a post-order DFT function for a given graph**

- Use recursion for this task
- Use `visited` as a dictionary that marks visited nodes as `True`
- Code for `DFT()` is given, complete the recursive `__DFTHelp()` method.

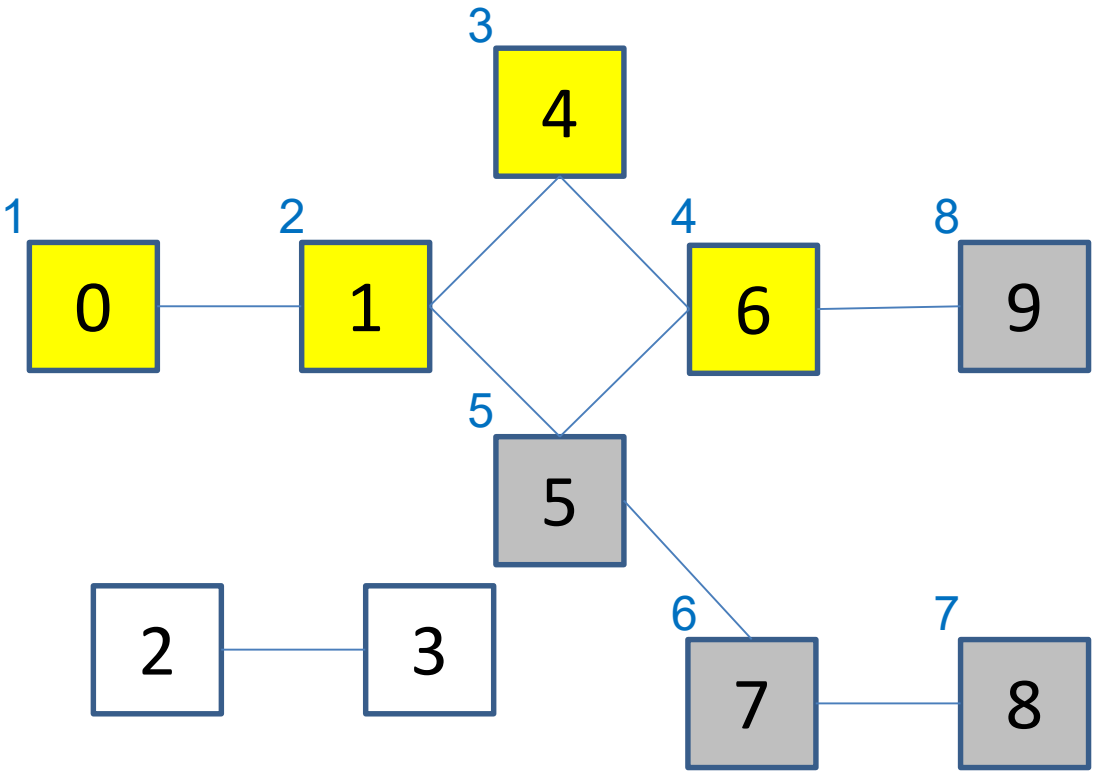



Ans: `8 7 5`

# 2. Implement DFT Post-order

**Implement a post-order DFT function for a given graph**

- Use recursion for this task
- Use `visited` as a dictionary that marks visited nodes as `True`
- Code for `DFT()` is given, complete the recursive `__DFTHelp()` method.

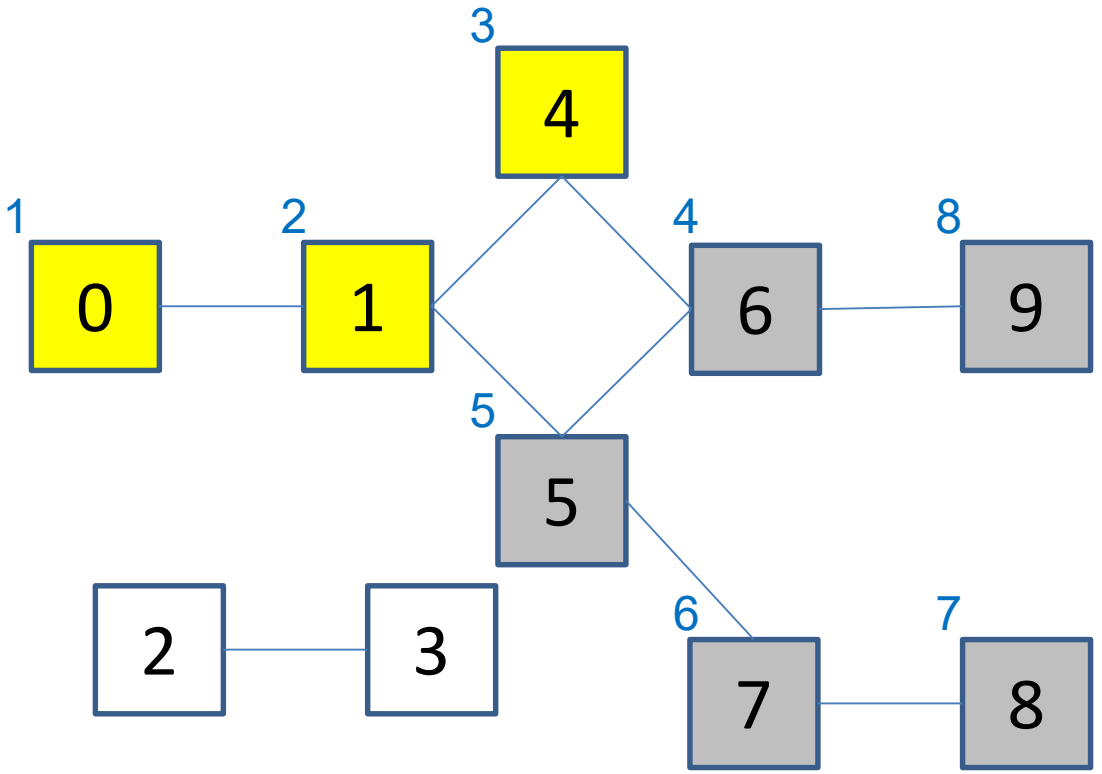



Ans: `8 7 5 9`

# 2. Implement DFT Post-order

**Implement a post-order DFT function for a given graph**

- Use recursion for this task
- Use `visited` as a dictionary that marks visited nodes as `True`
- Code for `DFT()` is given, complete the recursive `__DFTHelp()` method.

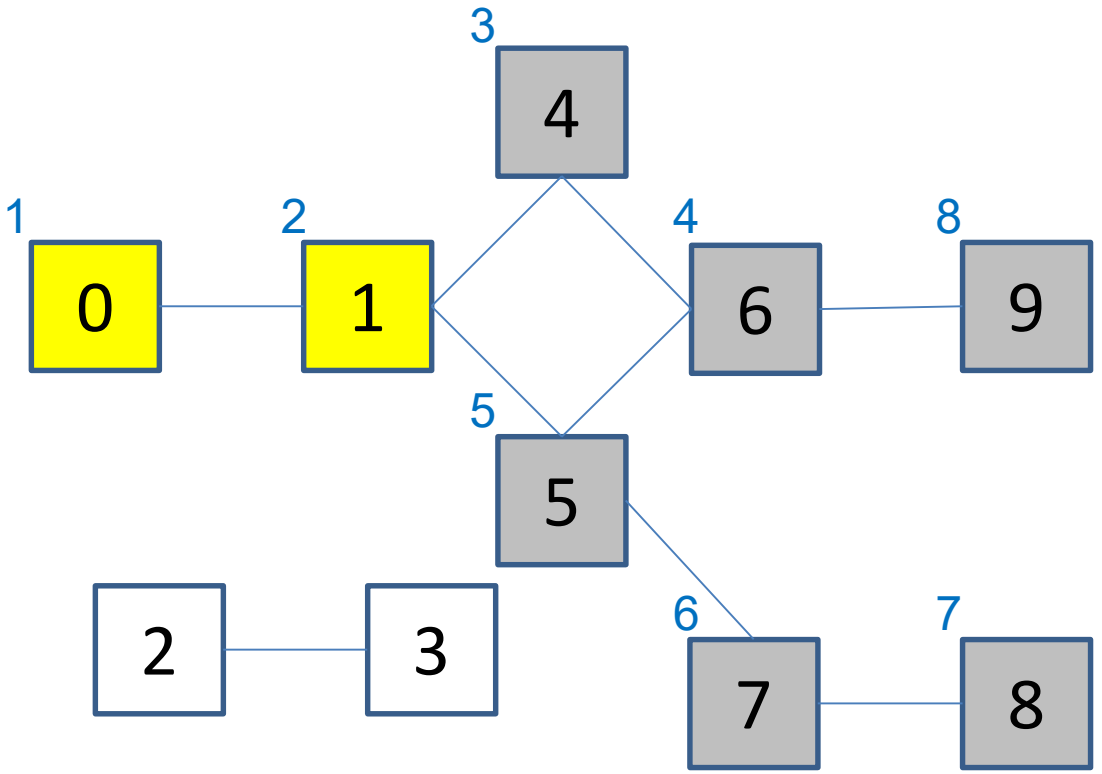



Ans: `8 7 5 9 6`

# 2. Implement DFT Post-order

**Implement a post-order DFT function for a given graph**

- Use recursion for this task
- Use `visited` as a dictionary that marks visited nodes as `True`
- Code for `DFT()` is given, complete the recursive `__DFTHelp()` method.

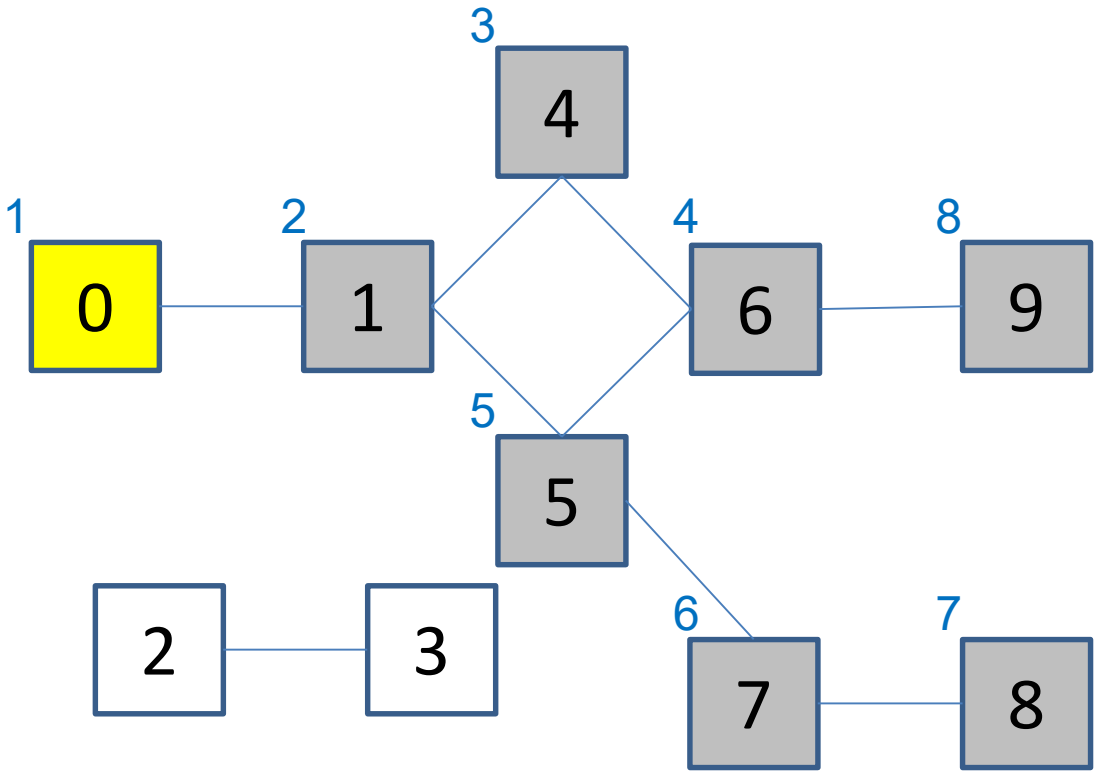



Ans: `8 7 5 9 6 4`

# 2. Implement DFT Post-order

**Implement a post-order DFT function for a given graph**

- Use recursion for this task
- Use `visited` as a dictionary that marks visited nodes as `True`
- Code for `DFT()` is given, complete the recursive `__DFTHelp()` method.

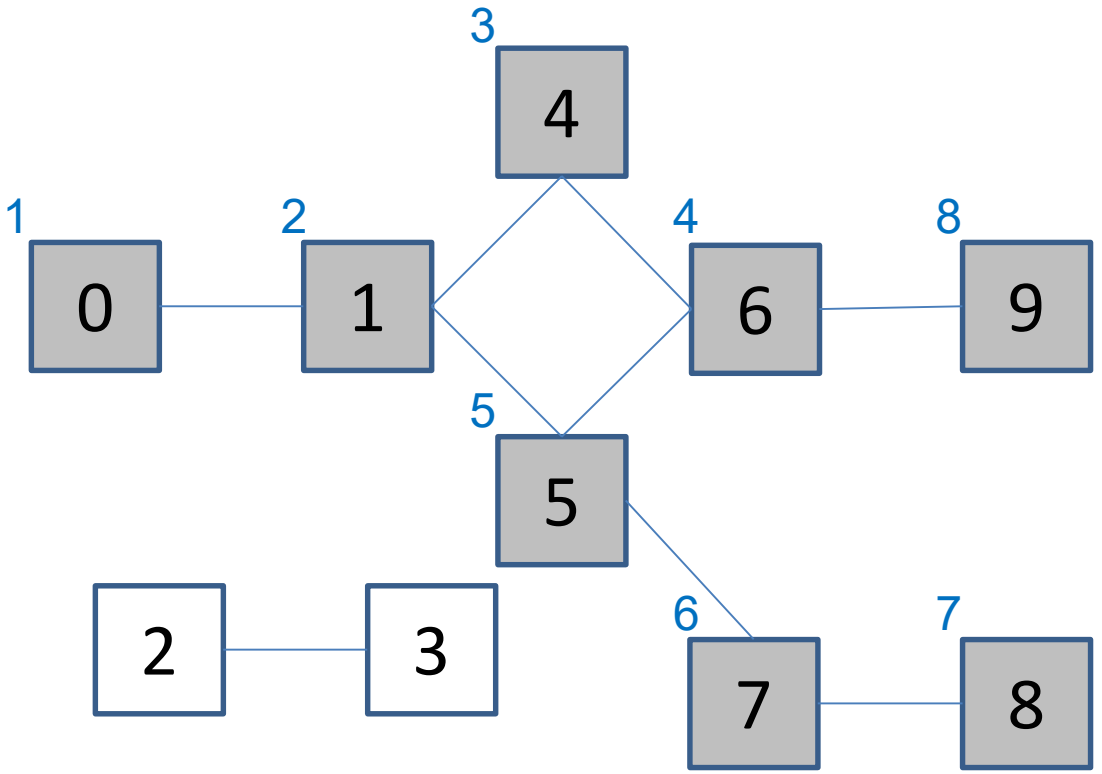



Ans: `8 7 5 9 6 4 1`

# 2. Implement DFT Post-order

**Implement a post-order DFT function for a given graph**

- Use recursion for this task
- Use `visited` as a dictionary that marks visited nodes as `True`
- Code for `DFT()` is given, complete the recursive `__DFTHelp()` method.

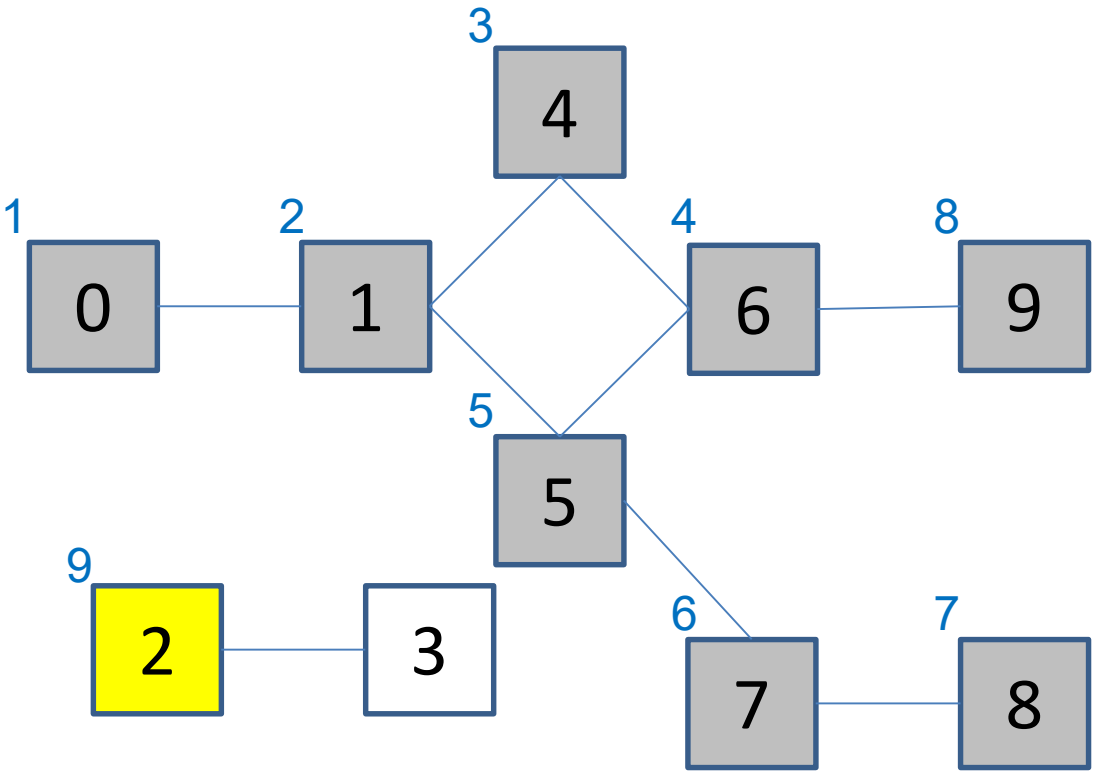



Ans: `8 7 5 9 6 4 1 0`

# 2. Implement DFT Post-order

**Implement a post-order DFT function for a given graph**

- Use recursion for this task
- Use `visited` as a dictionary that marks visited nodes as `True`
- Code for `DFT()` is given, complete the recursive `__DFTHelp()` method.

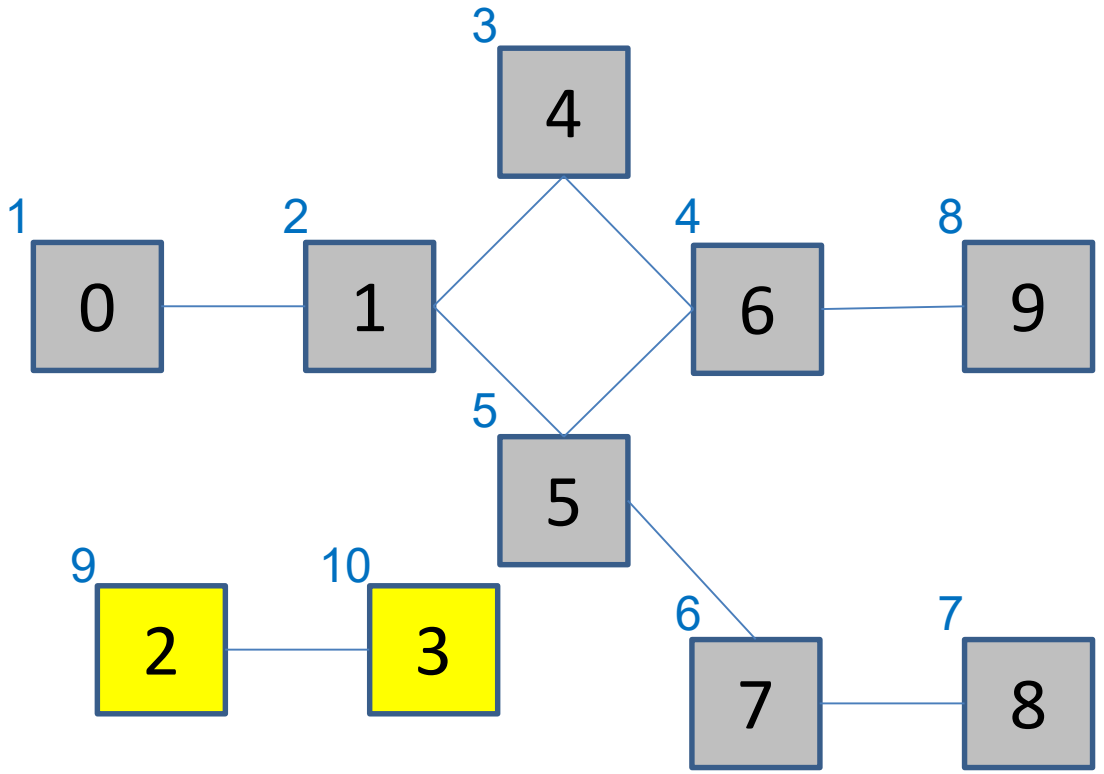



Ans: `8 7 5 9 6 4 1 0`

# 2. Implement DFT Post-order

**Implement a post-order DFT function for a given graph**

- Use recursion for this task
- Use `visited` as a dictionary that marks visited nodes as `True`
- Code for `DFT()` is given, complete the recursive `__DFTHelp()` method.

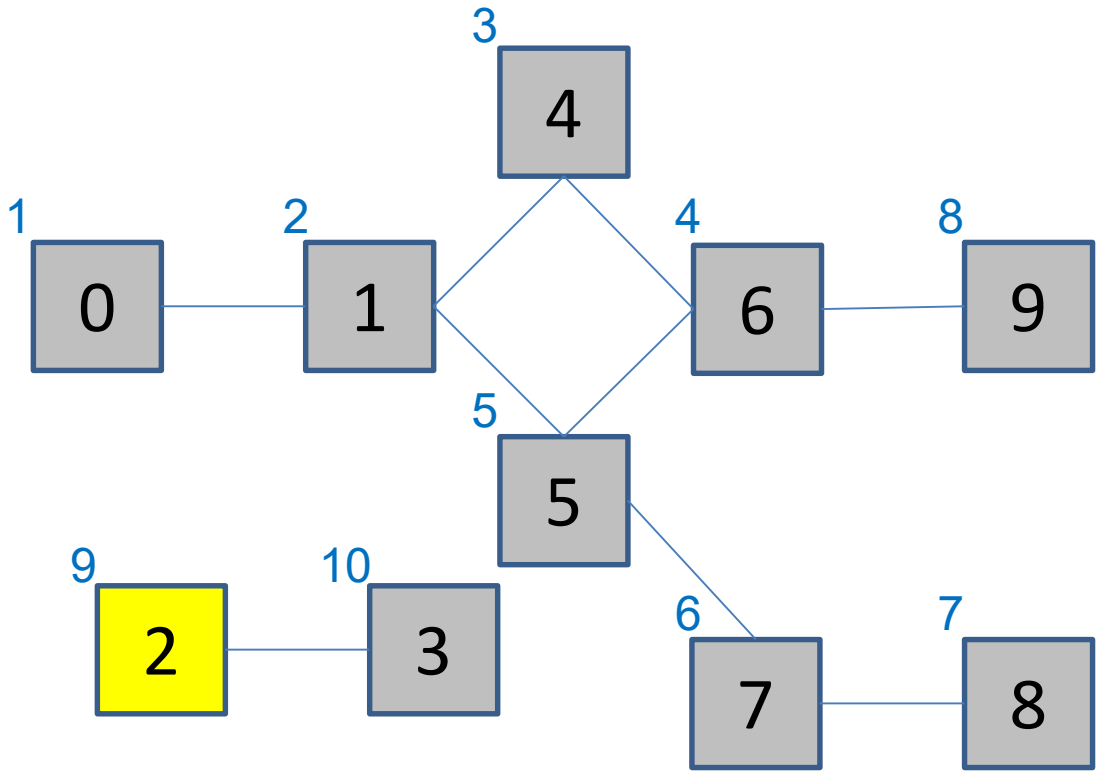



Ans: `8 7 5 9 6 4 1 0`

# 2. Implement DFT Post-order

**Implement a post-order DFT function for a given graph**

- Use recursion for this task
- Use `visited` as a dictionary that marks visited nodes as `True`
- Code for `DFT()` is given, complete the recursive `__DFTHelp()` method.

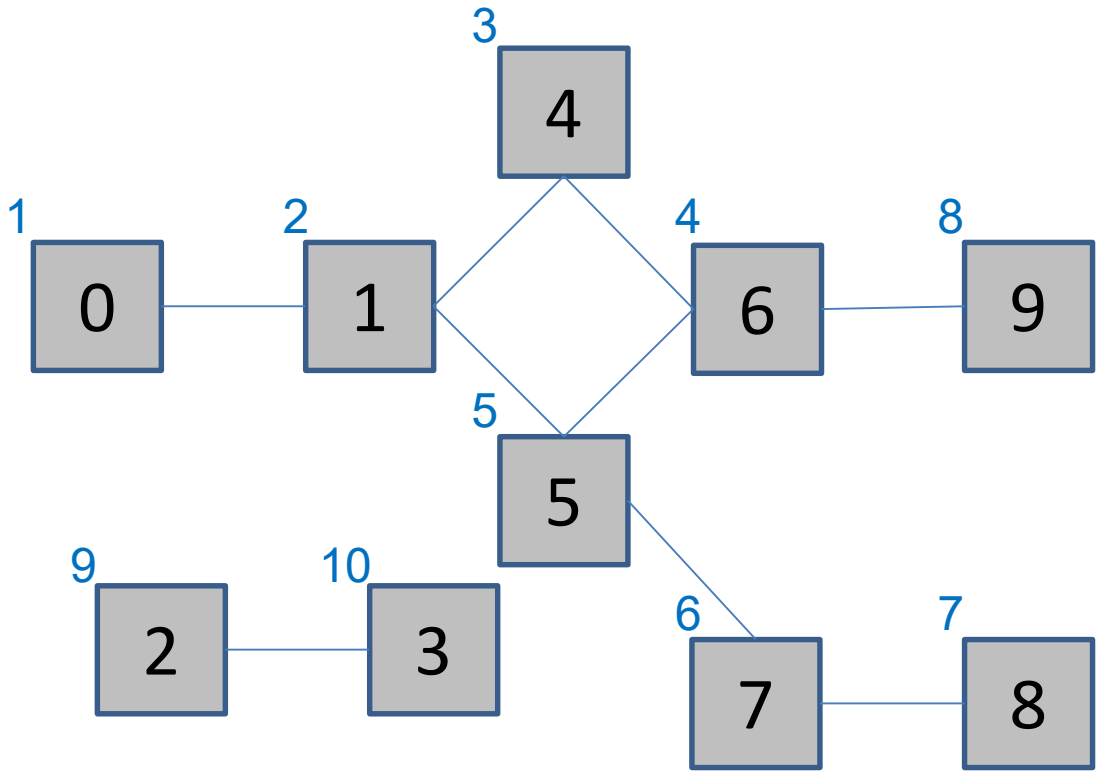



Ans: `8 7 5 9 6 4 1 0 3`

# 2. Implement DFT Post-order

**Implement a post-order DFT function for a given graph**

- Use recursion for this task
- Use `visited` as a dictionary that marks visited nodes as `True`
- Code for `DFT()` is given, complete the recursive `__DFTHelp()` method.



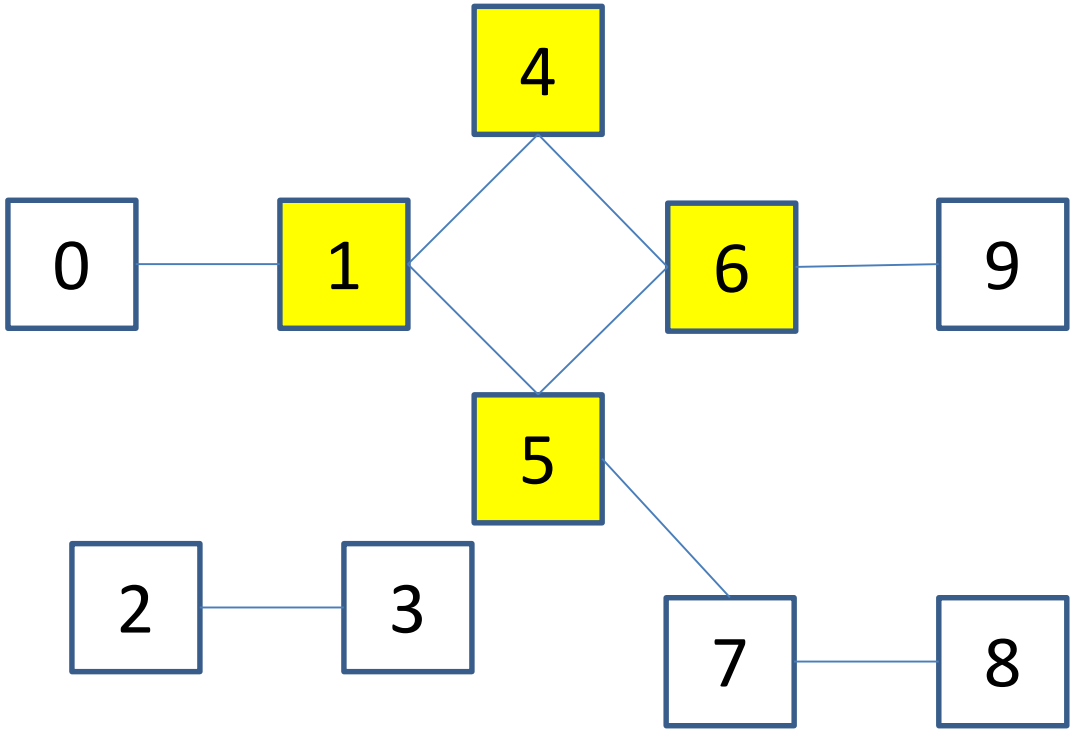

Ans: `8 7 5 9 6 4 1 0 3 2`

# 3. Cycle Detection

**Implement a method to detect if cycles exists in an undirected graph**

- Given a node, use recursion to see if a cycle exists (hint: DFT!)
- If at least one cycle is found, immediately return `True`

`True` (There is a cycle 1 – 4 – 6 – 5 – 1)



# Breakout room guidelines

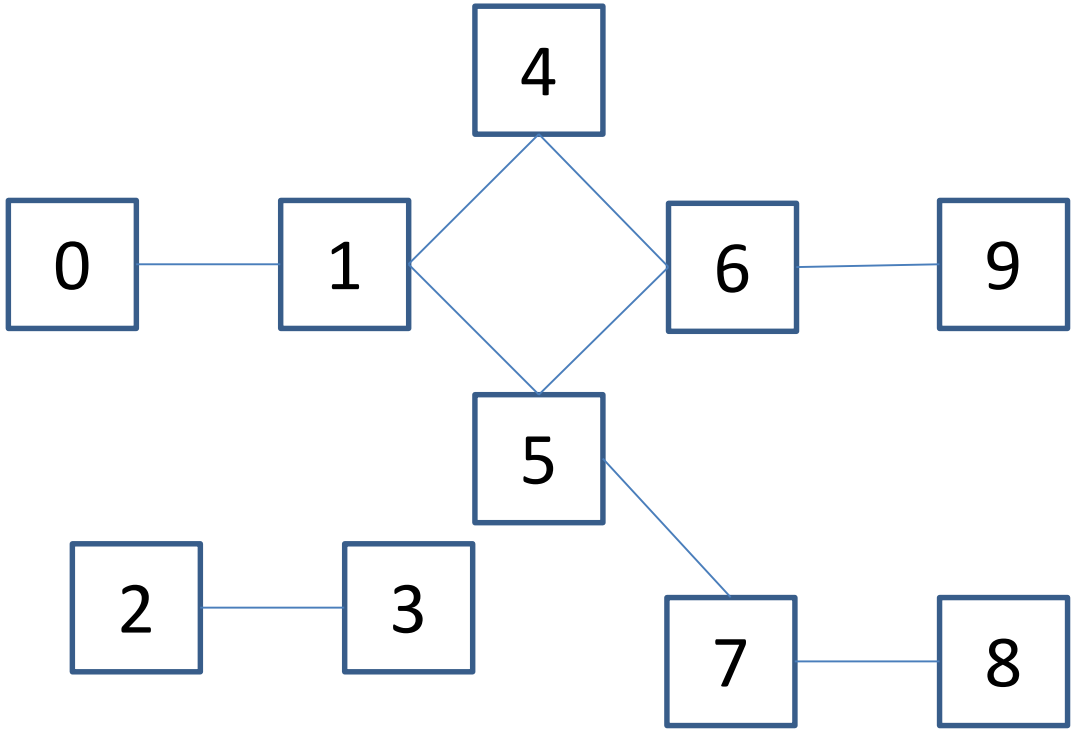
- 조를 짜신 분들은 빈 소회의실에 들어가서 자유롭게 실습하셔도 좋습니다.
- 실습 중에 질문이 있다면 본 줌 미팅에서 채팅 혹은 손들기 후 질문해도 괜찮습니다.
- 조를 아직 안 편성하셨거나 다른 분들과 토의하시고 싶은 분들 또한 소회의실에 접속하셔도 좋습니다.

# 02. Solution

# 1. Implement BFT

**Implement a BFT method for a given graph**

- BFT: Visit all connected nodes level by level
- Use a queue (in Python, use `deque()`)
- Use `visited` as a dictionary that marks visited nodes as `True`




Ans: `0 1 4 5 6 7 9 8 2 3`

# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Initialize all vertices as unvisited

# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Initialize queue

# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Iterate through each vertex

# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Add current vertex to queue

# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

While queue is not empty (and there exists vertices to visit)

# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Retrieve earliest vertex inserted into the queue (remember queue is FIFO!)

# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

If the vertex has not been visited yet, we need to add its neighbors for future visits

# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Add unvisited neighbors of the vertex `v` for future visits

# 1. Implement BFT

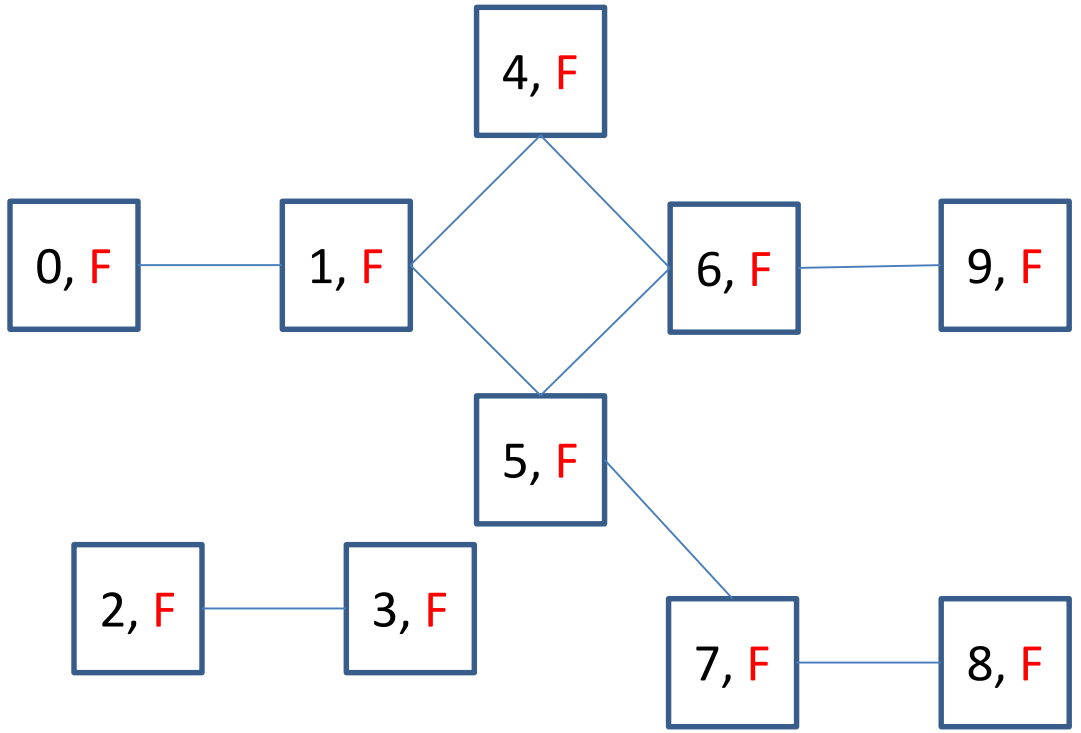
```
def BFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        q = deque()
        for v in self.V:
            q.append(v)
            while q:
                v = q.popleft()
                if not visited[v]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Mark current vertex v as visited and print

# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Output:

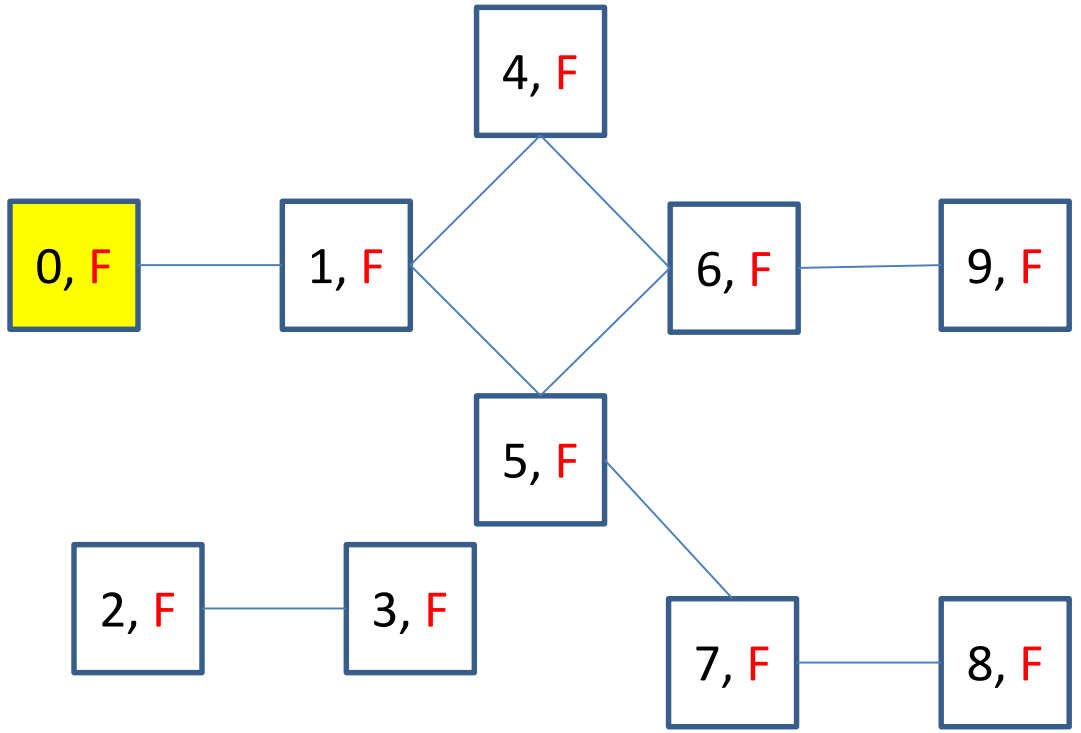


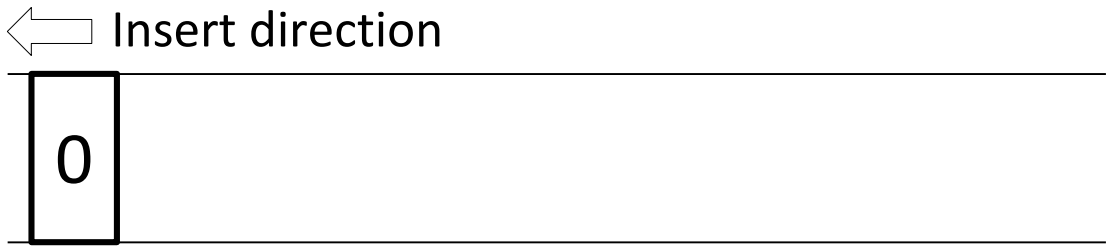

⇦ Insert direction

# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

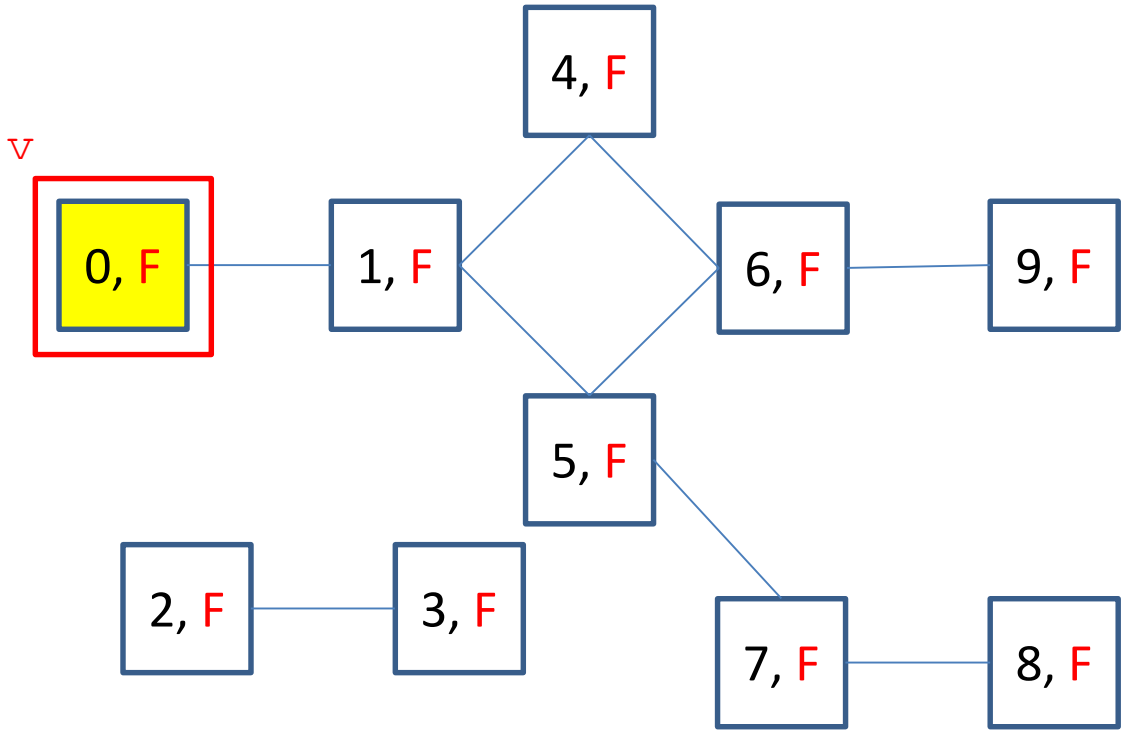
Output:



# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```



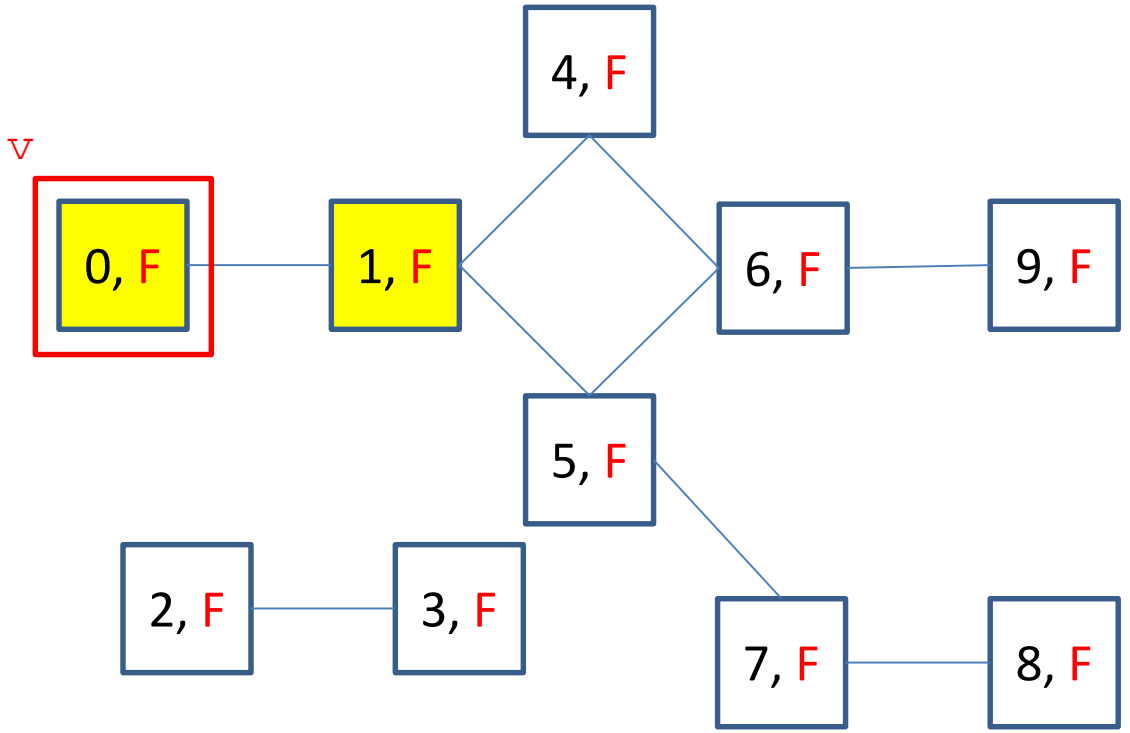

⇦ Insert direction

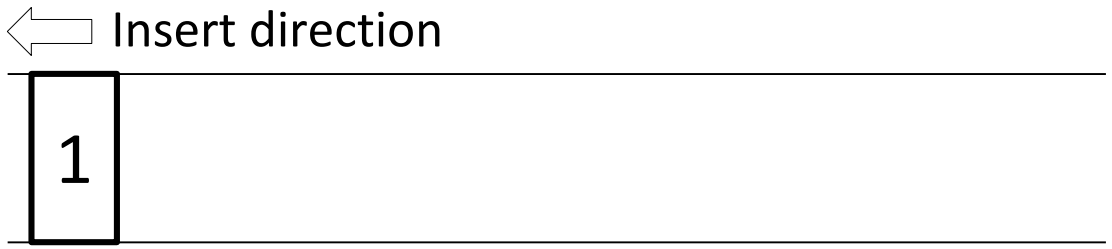
Output:

# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Output:

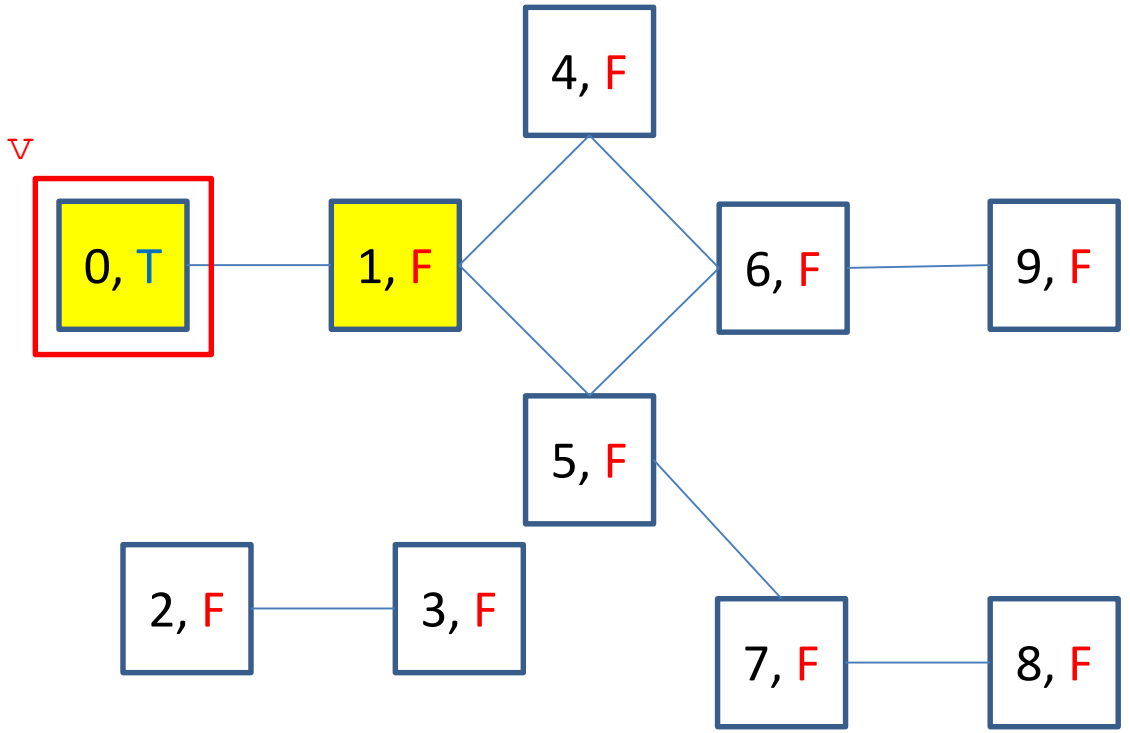


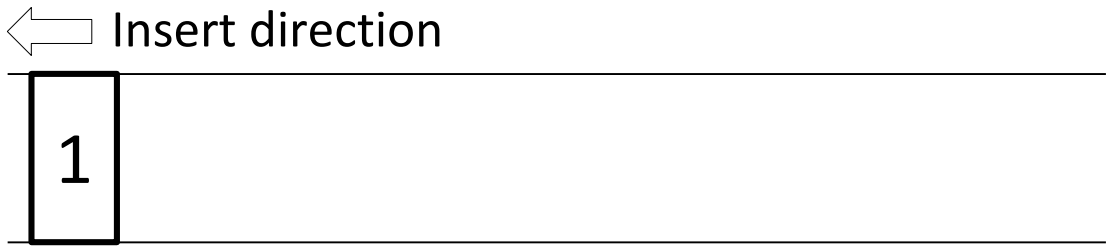



# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Output: 0

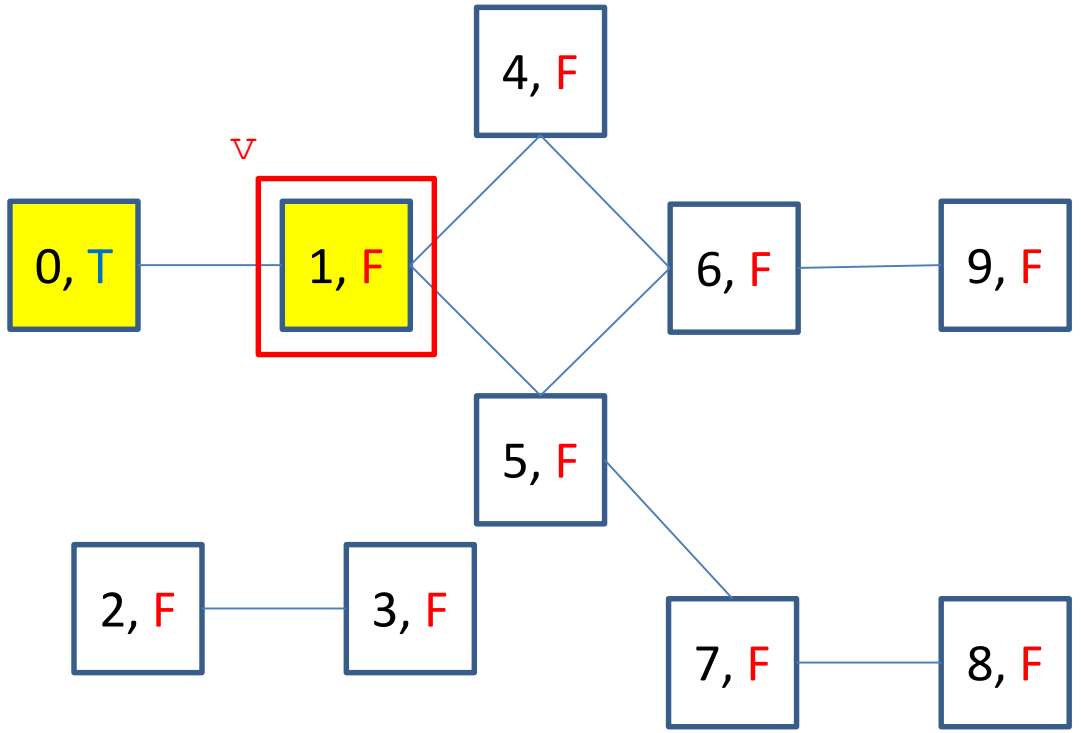


# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Output: 0

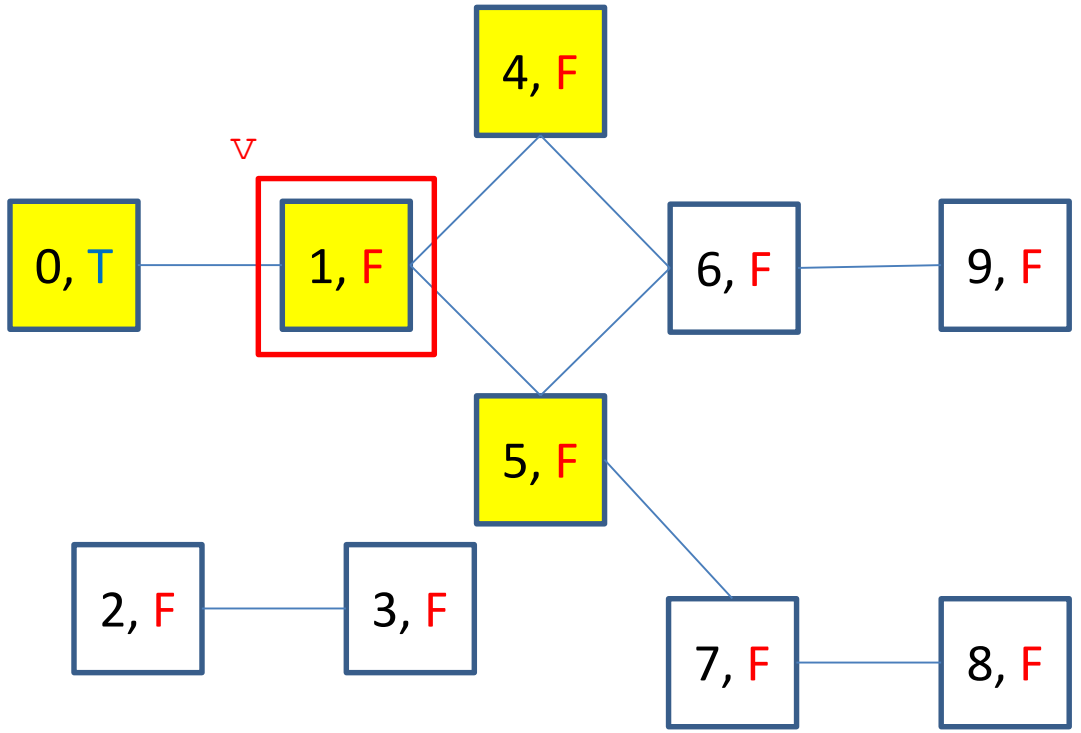


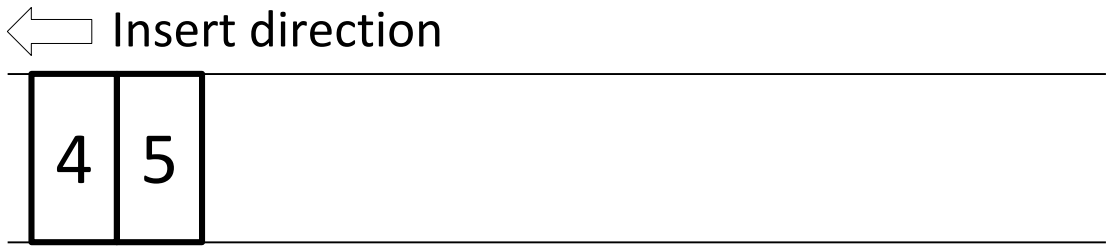

⇦ Insert direction

# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Output: 0

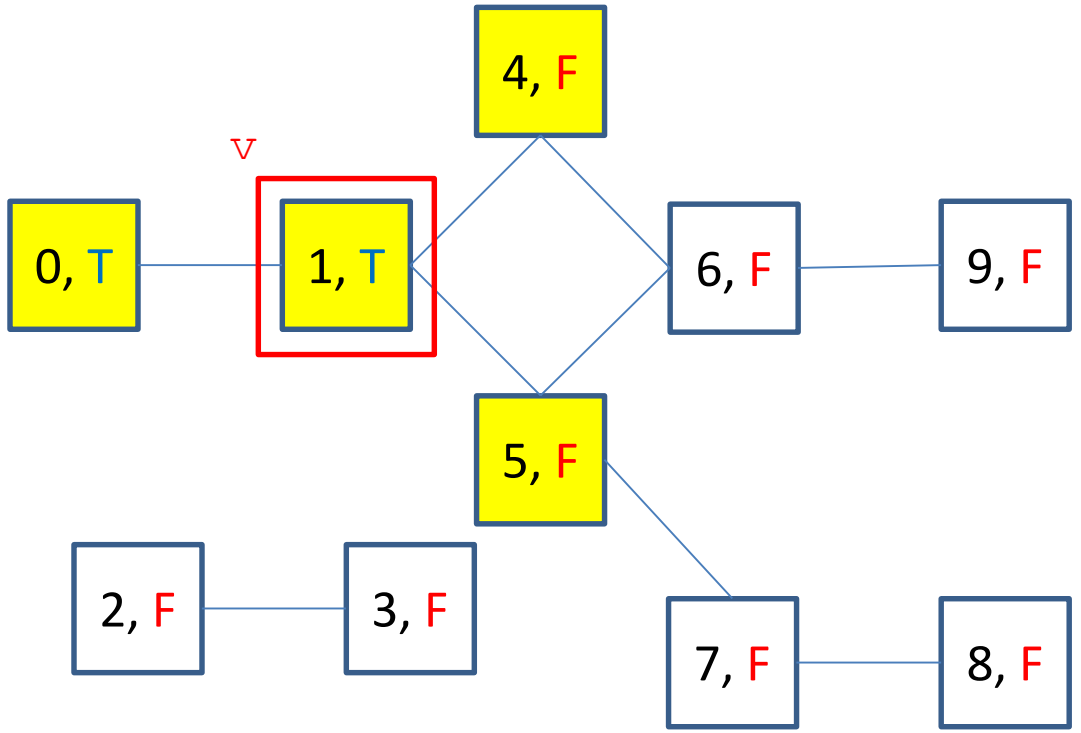


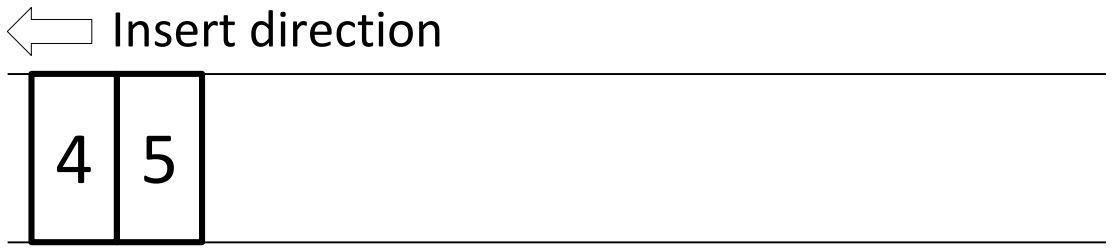



# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Output: 0 1

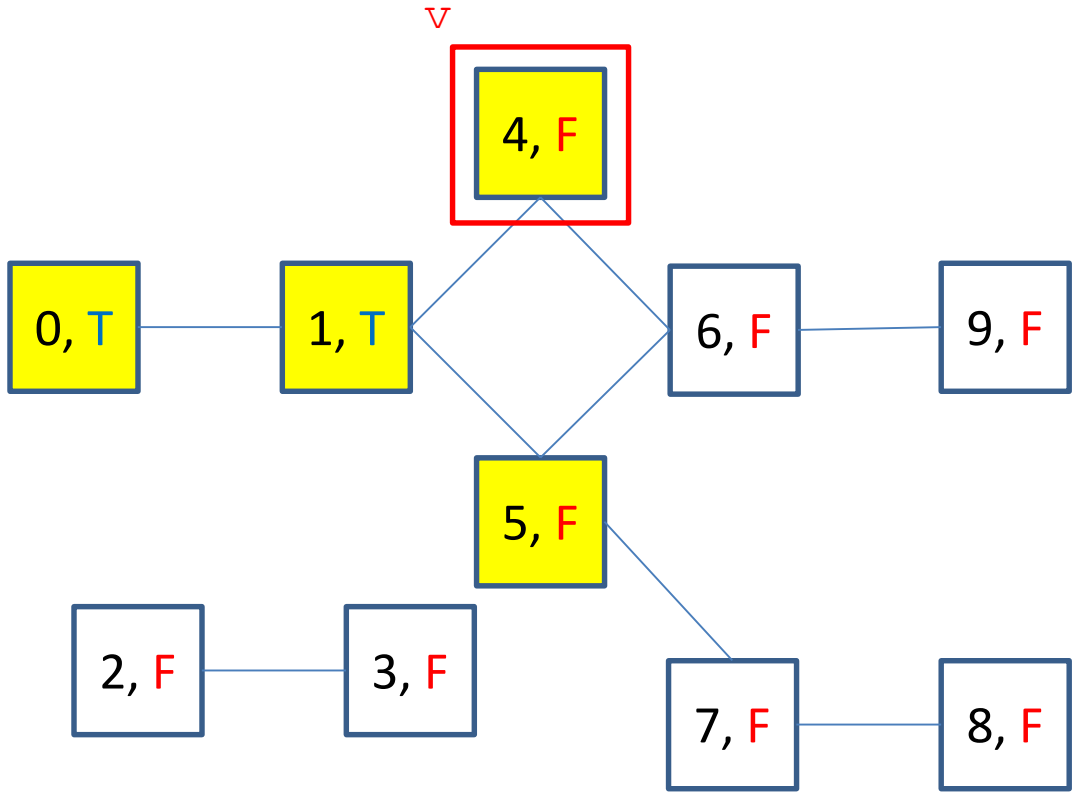


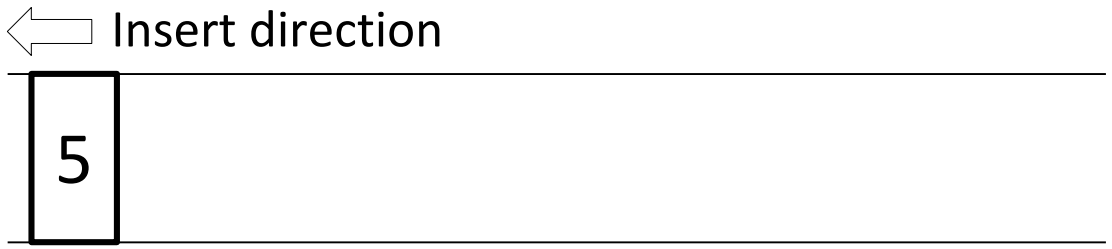



# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Output: 0 1

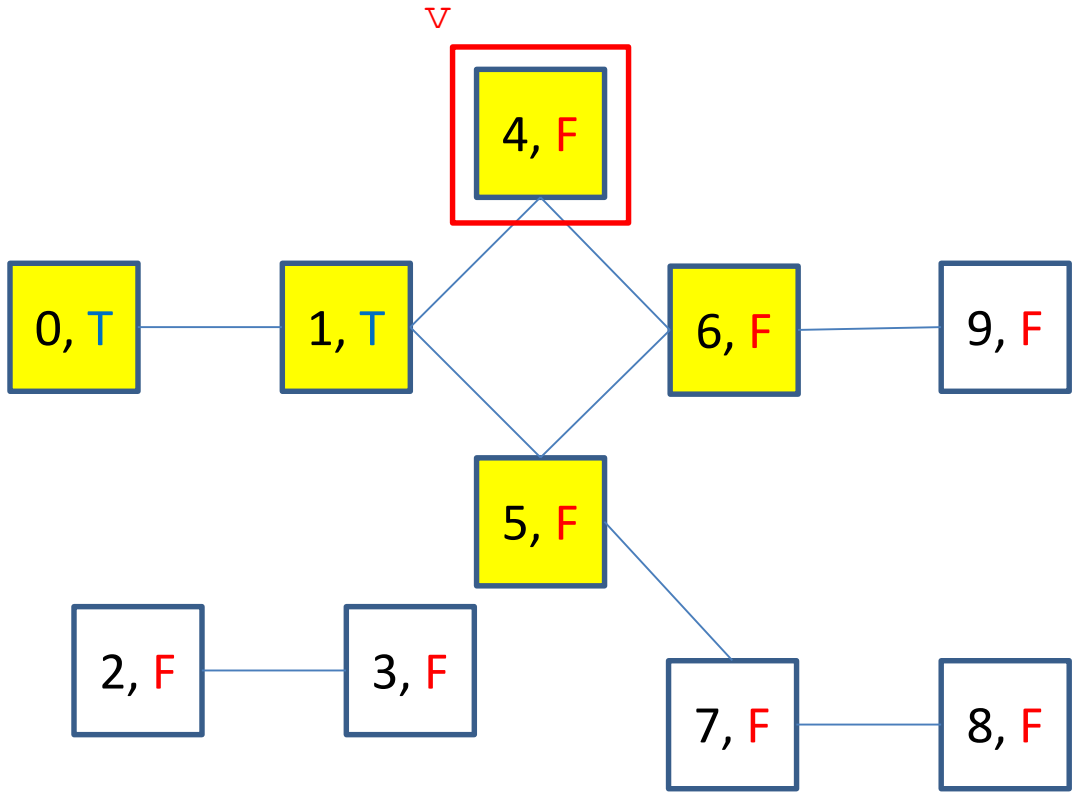


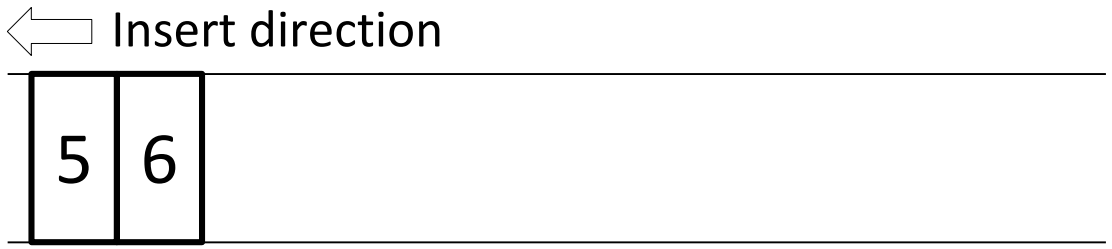



# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Output: `0 1`

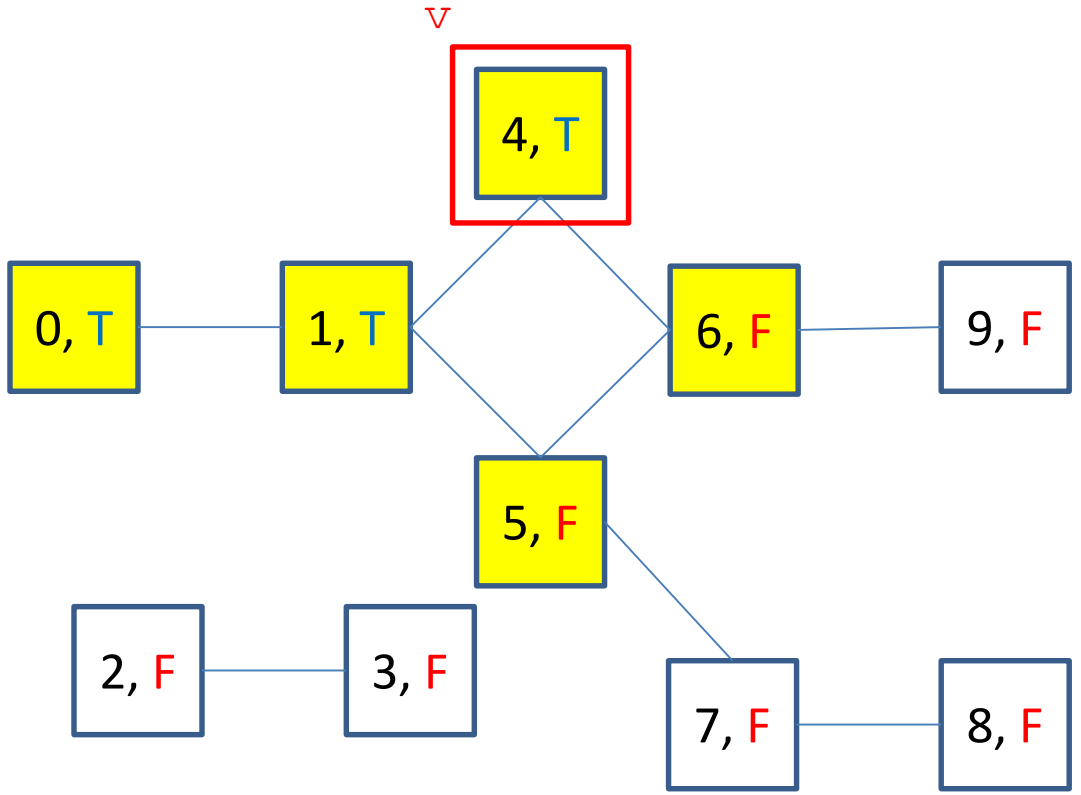


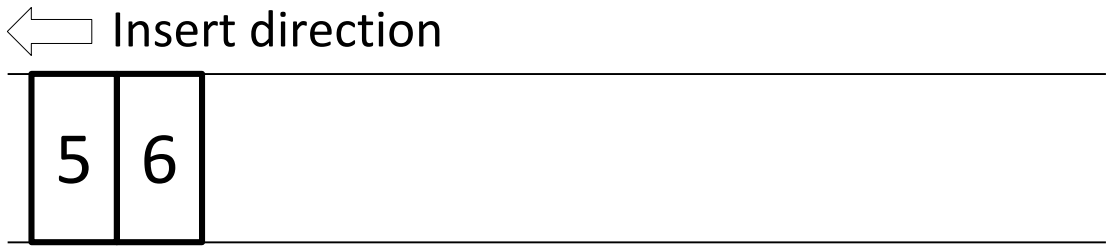



# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Output: 0 1 4

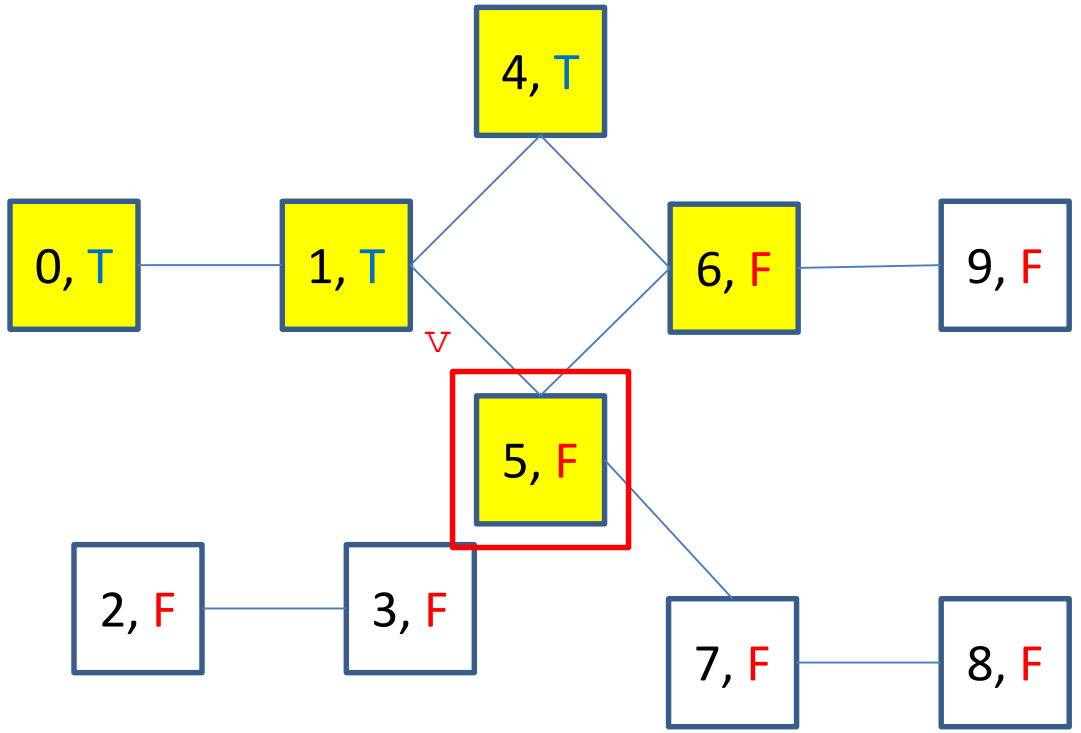


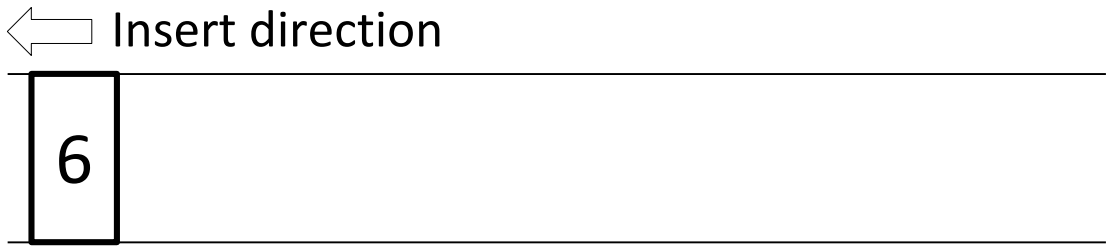



# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Output: 0 1 4

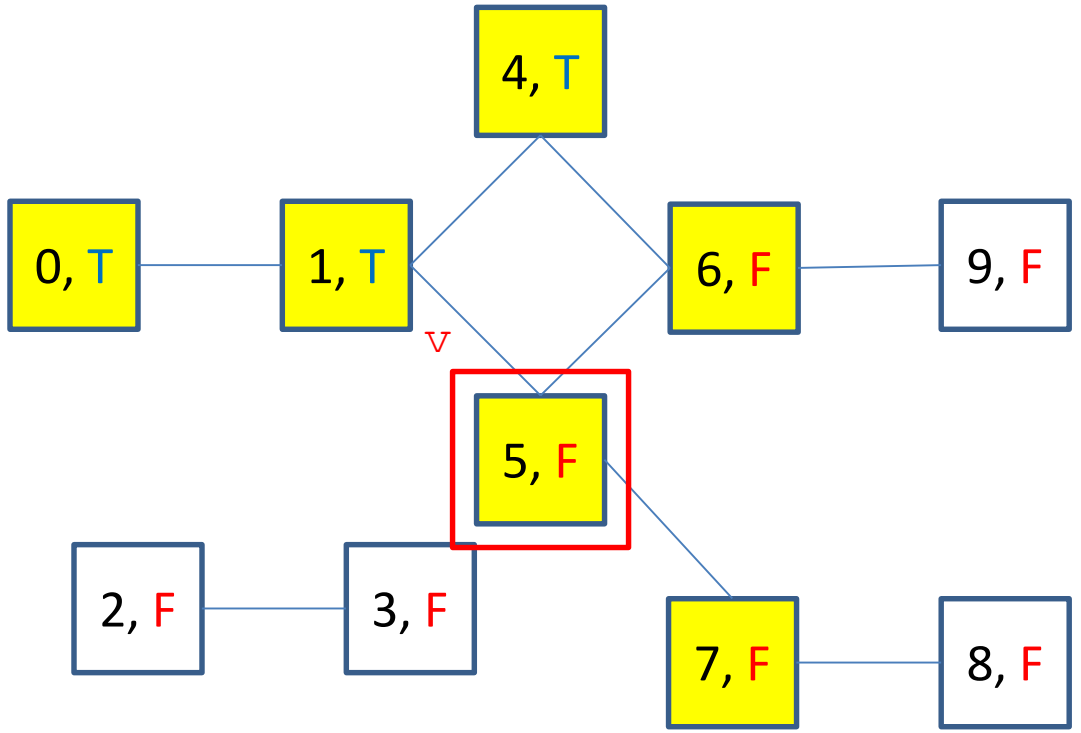


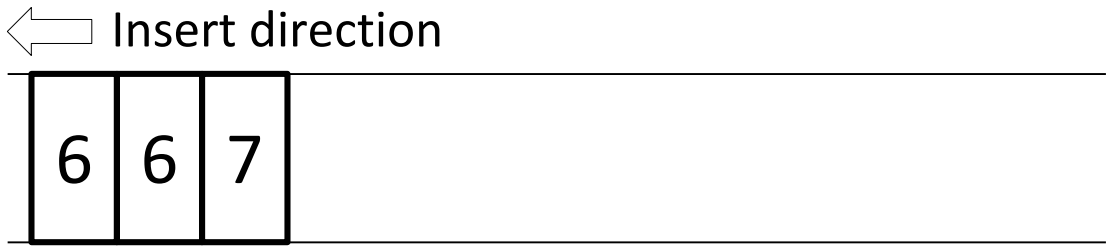



# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Output: 0 1 4

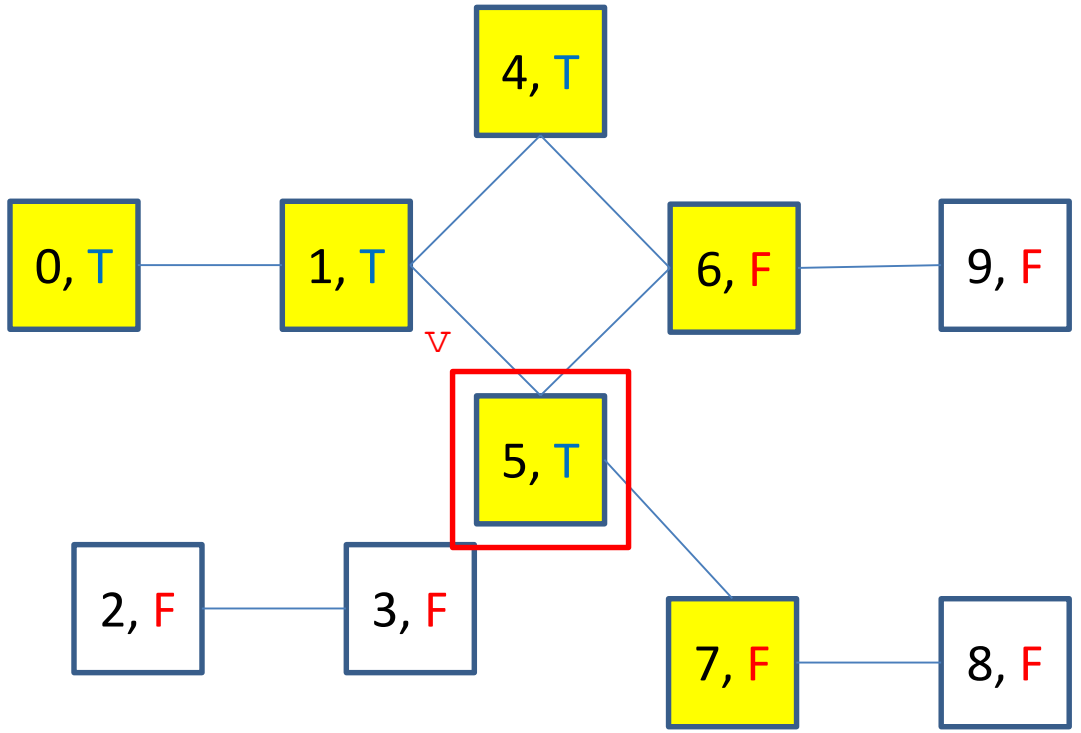


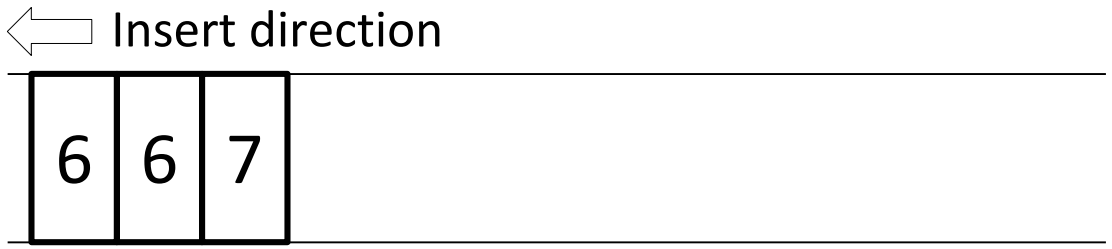



# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Output: 0 1 4 5

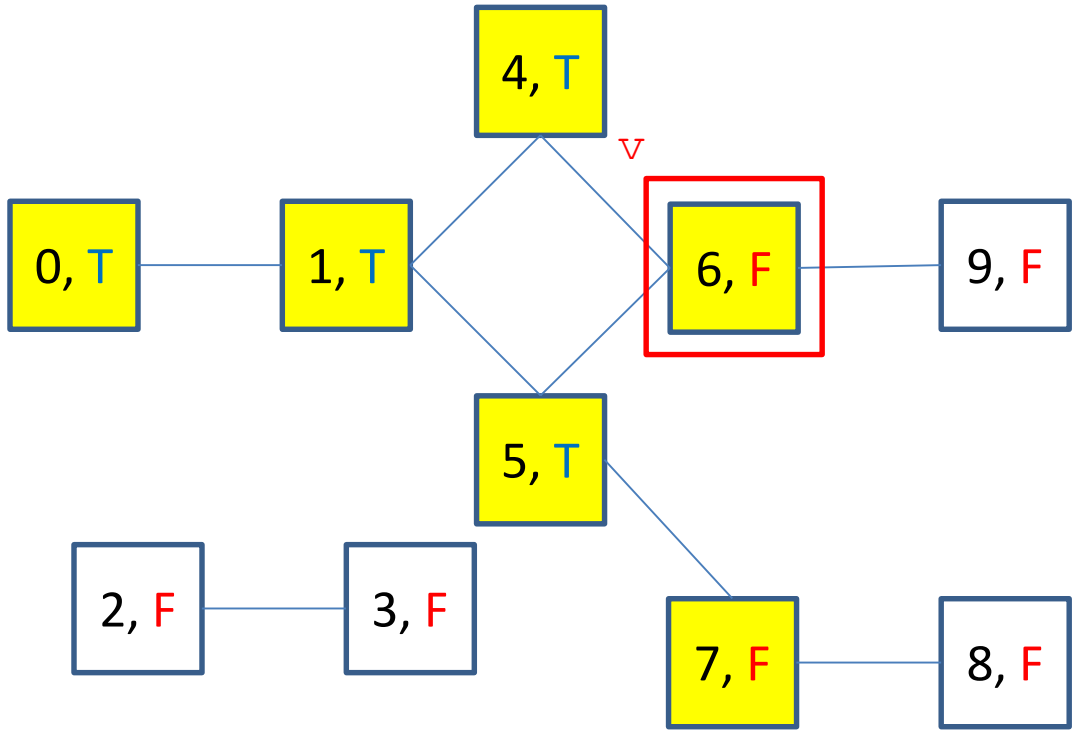


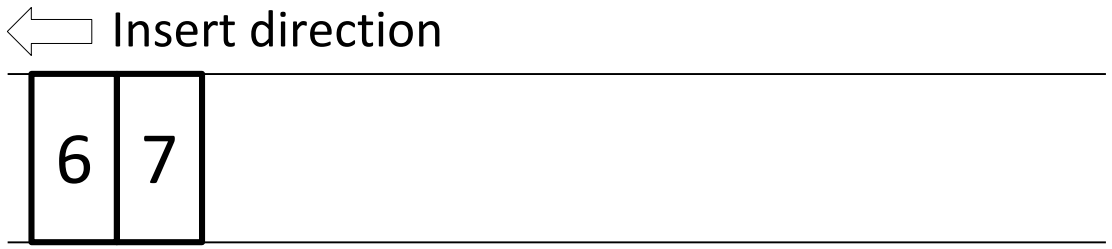



# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Output: 0 1 4 5

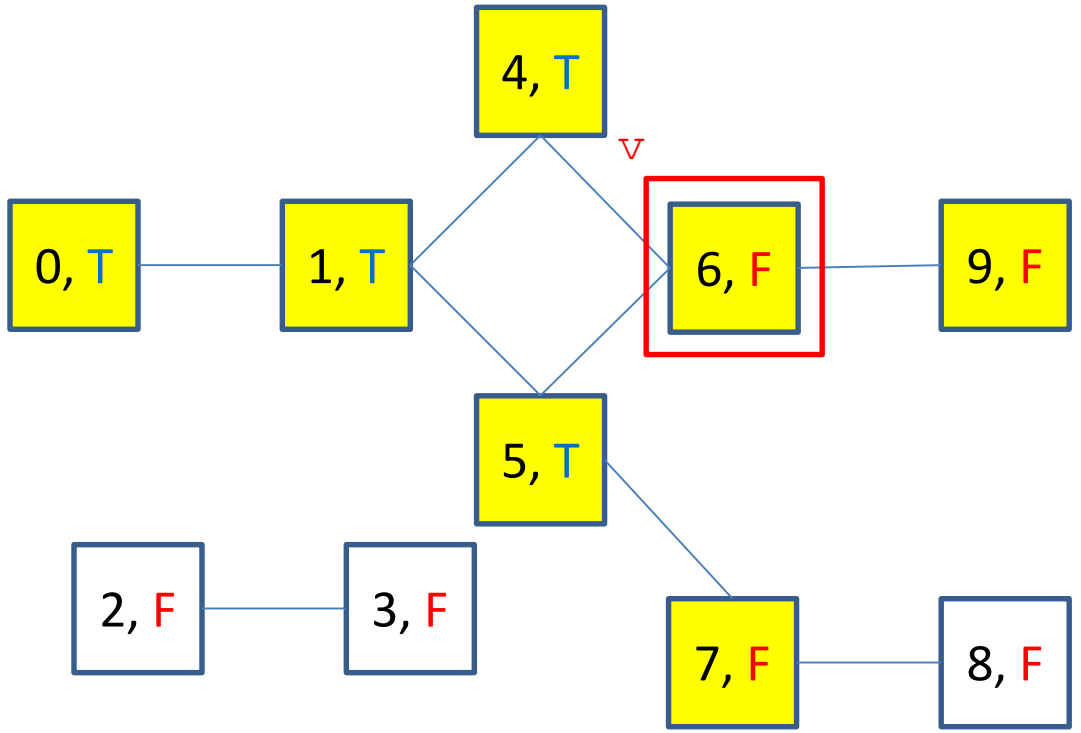


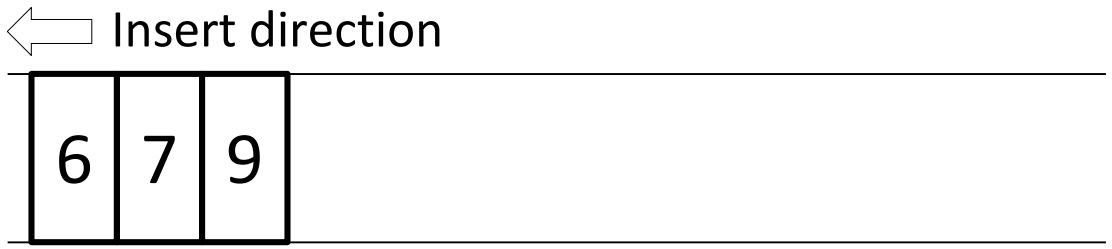



# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Output: 0 1 4 5

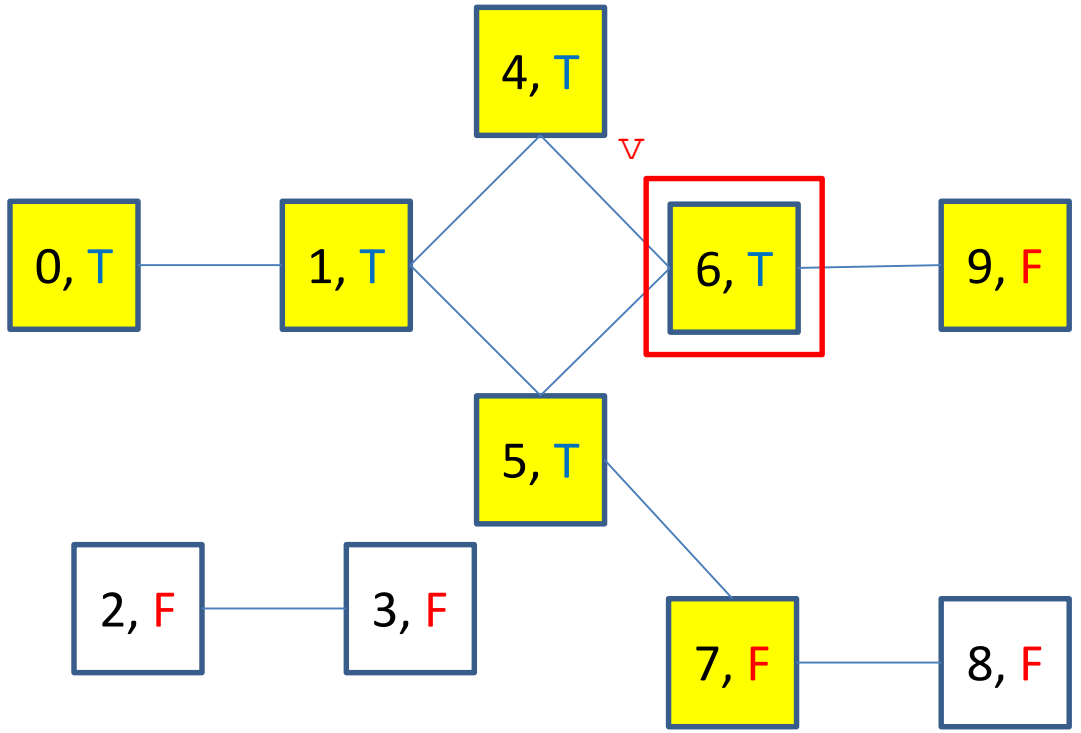


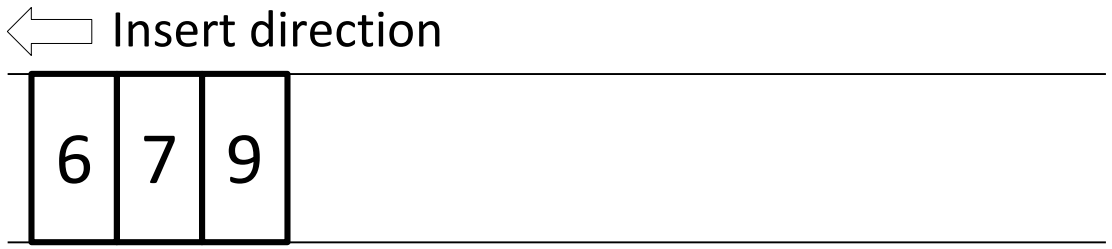



# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Output: 0 1 4 5 6

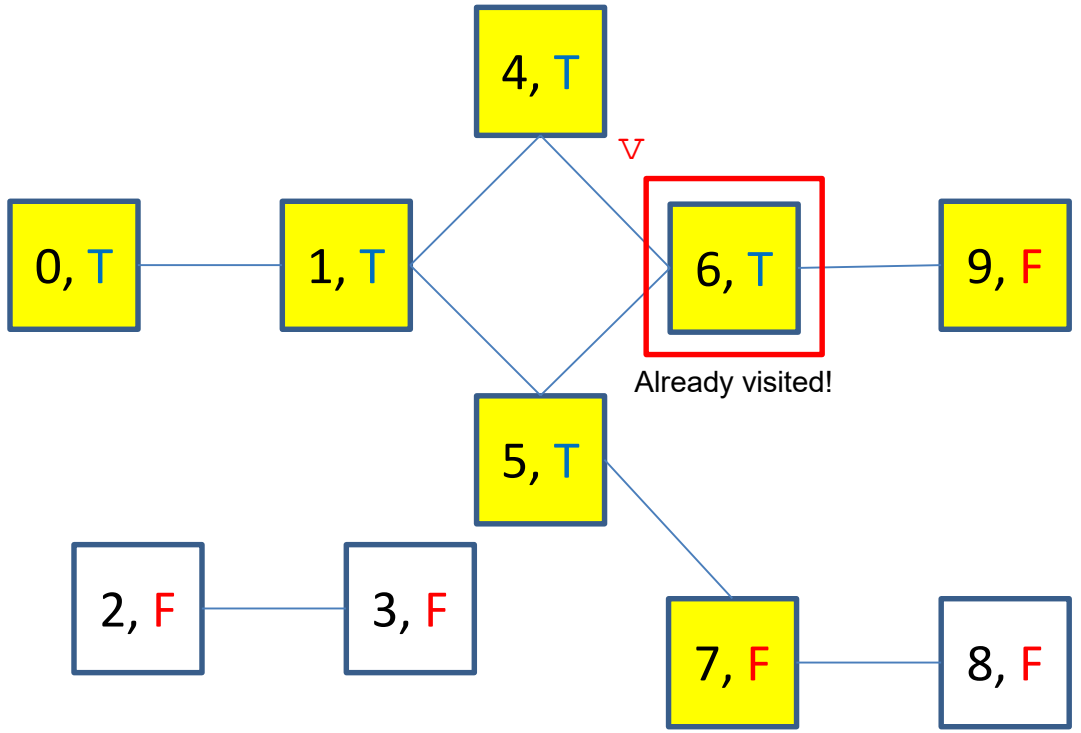


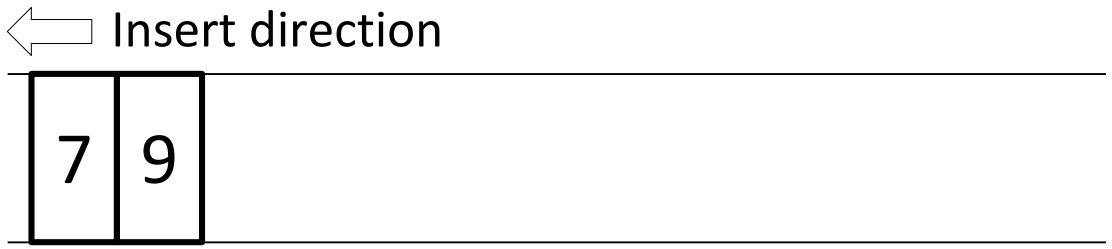



# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Output: 0 1 4 5 6

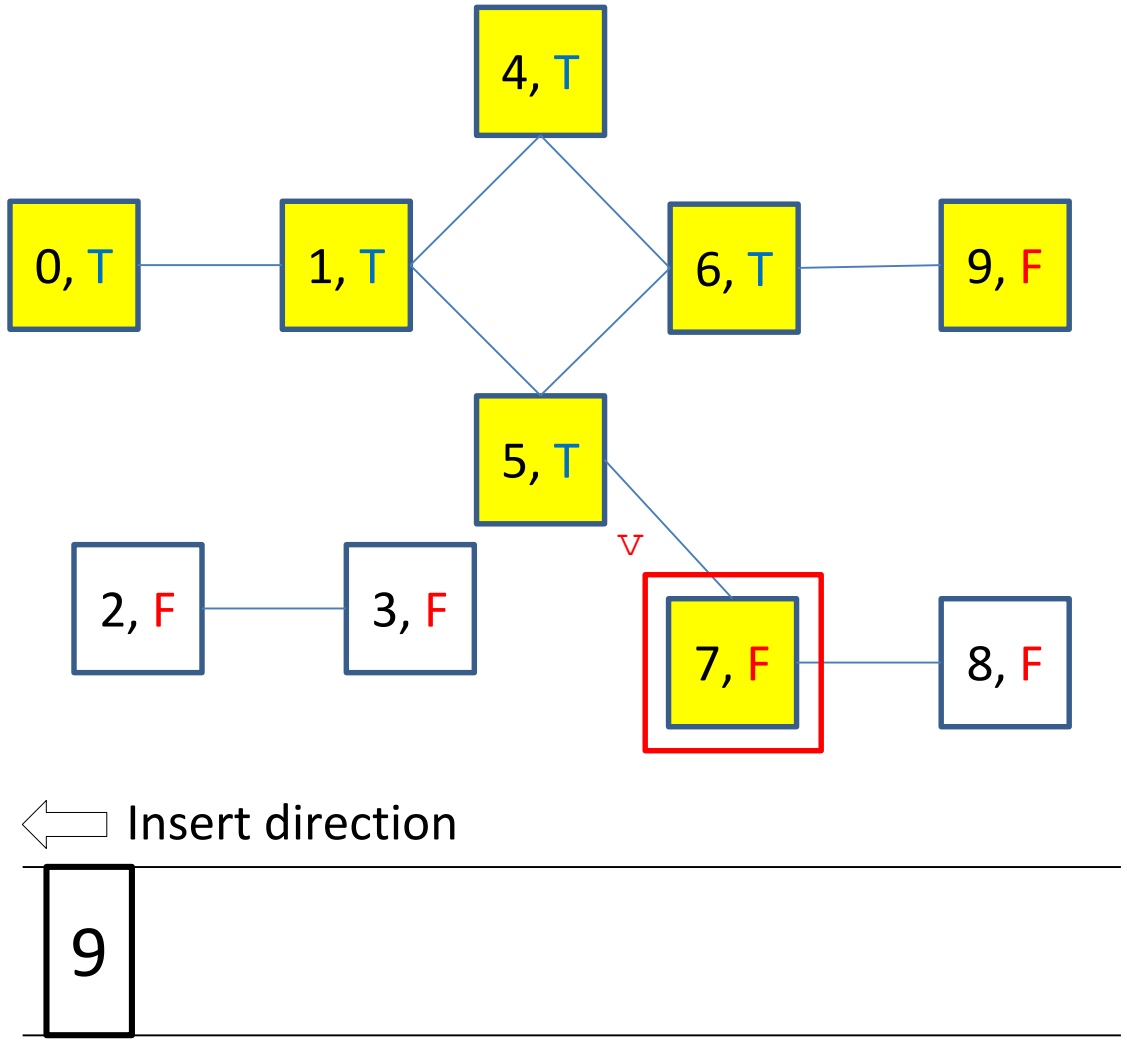


# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

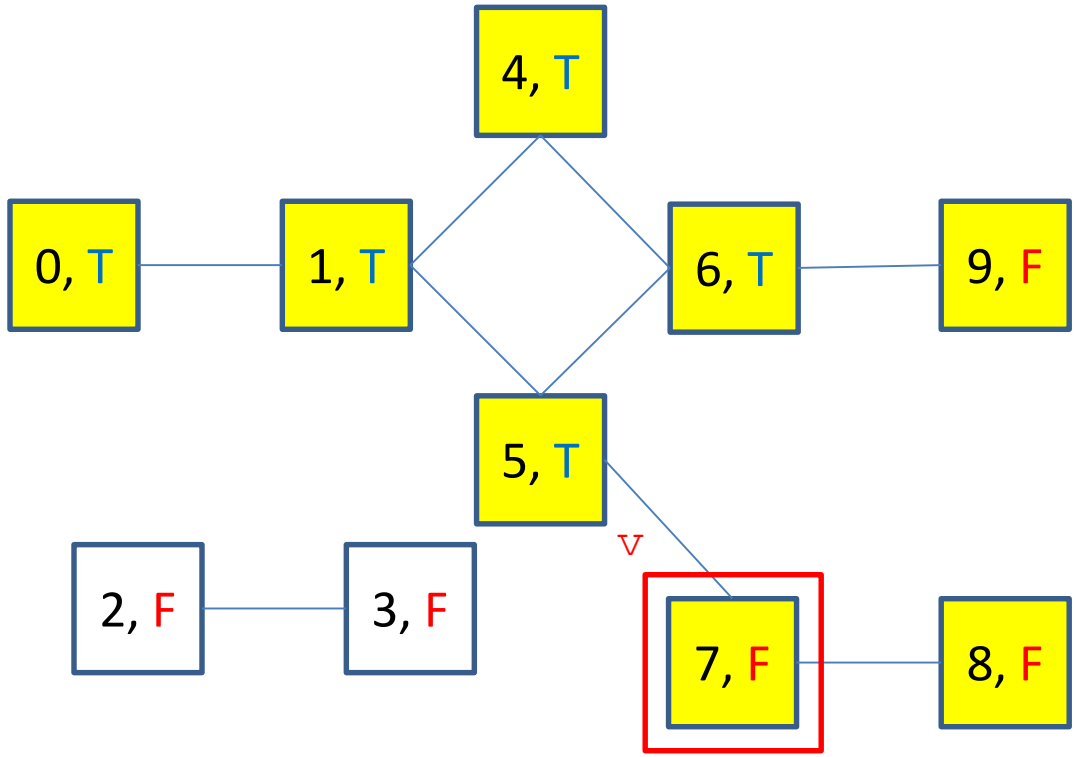
Output: 0 1 4 5 6

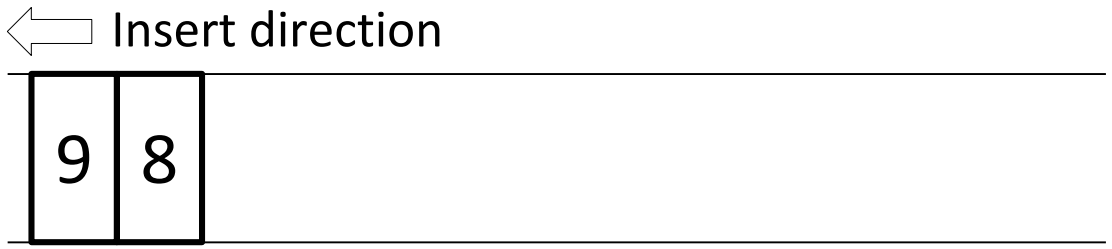


# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Output: 0 1 4 5 6

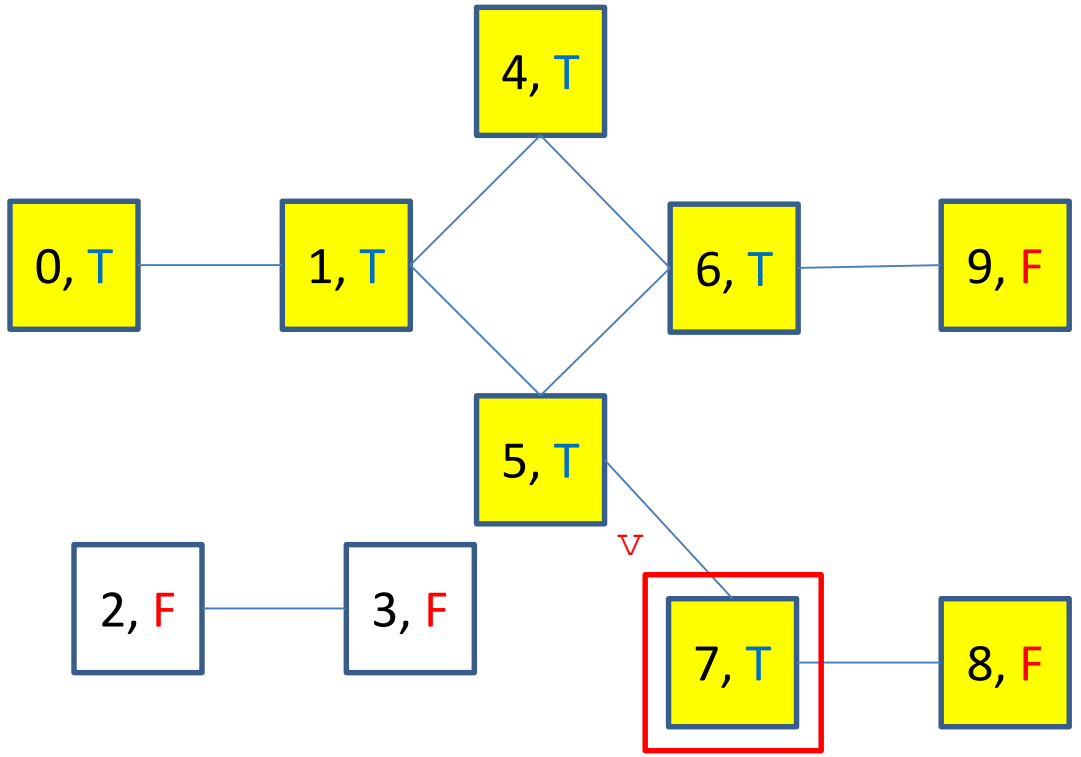


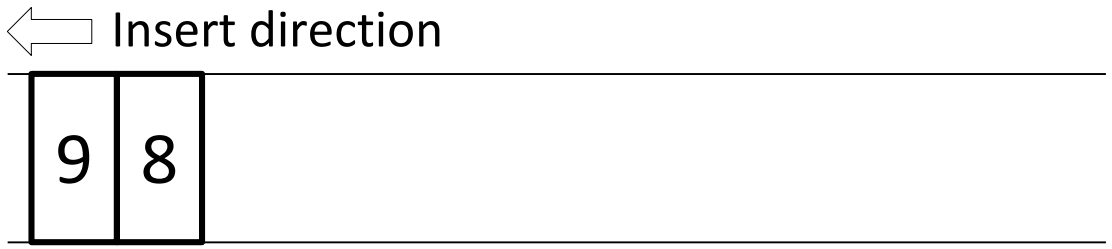



# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Output: 0 1 4 5 6 7

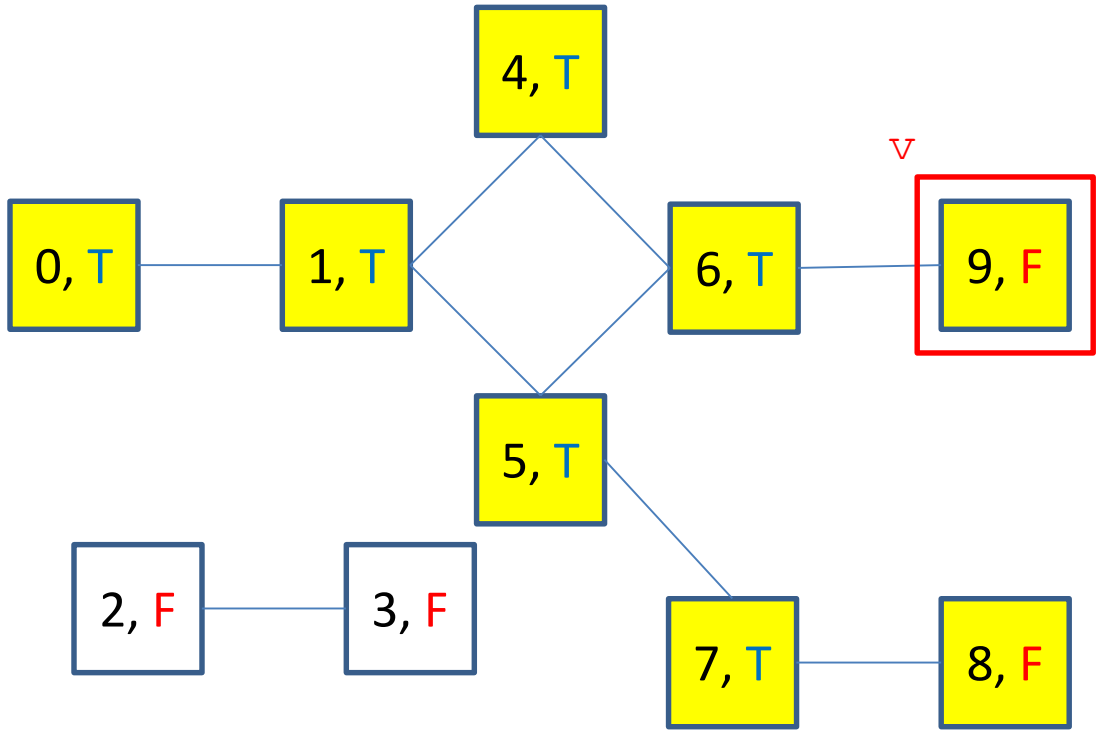


# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Output: 0 1 4 5 6 7

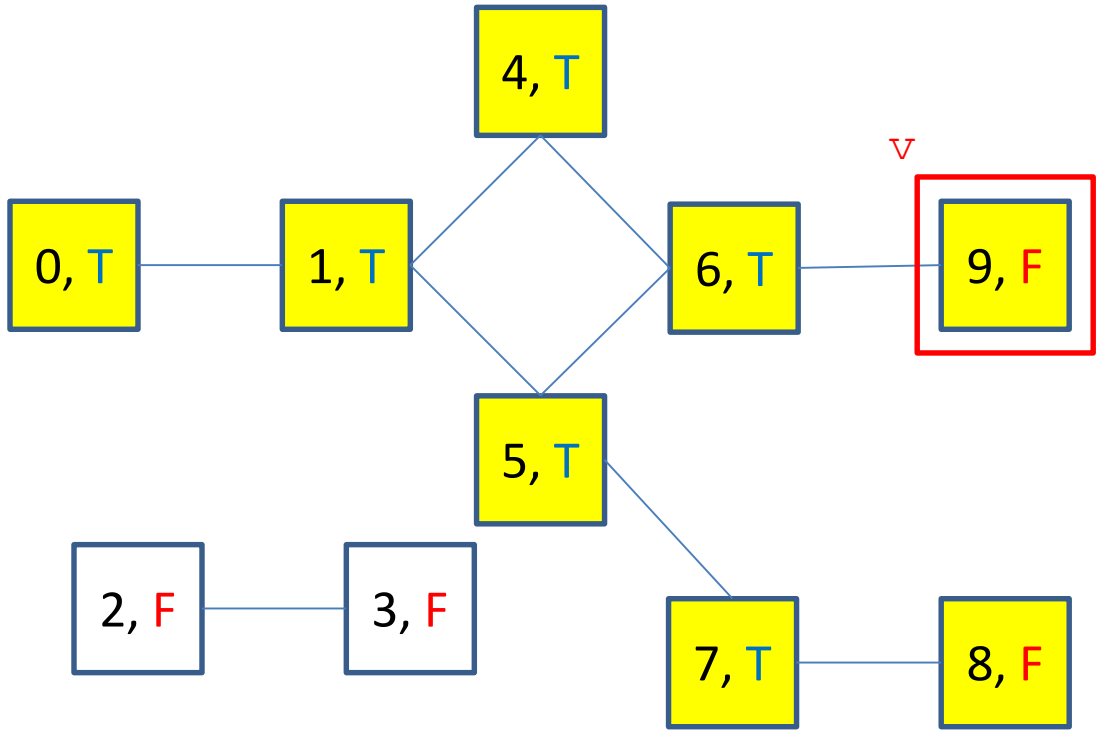



⇦ Insert direction

# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Output: 0 1 4 5 6 7

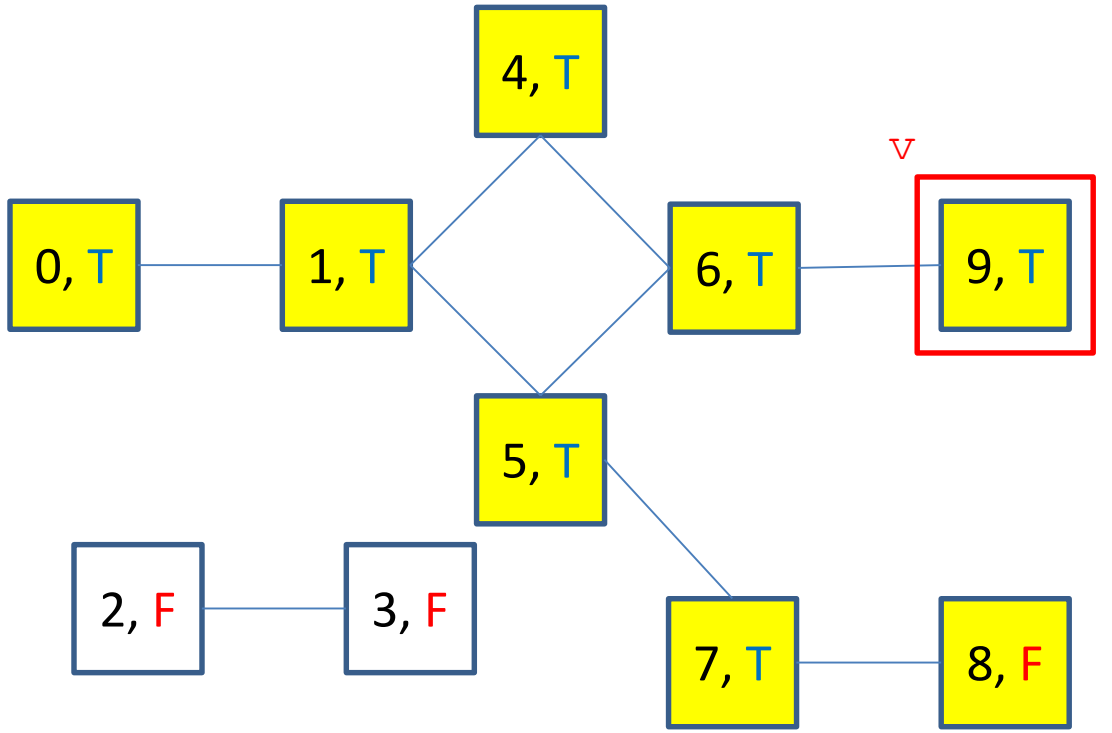


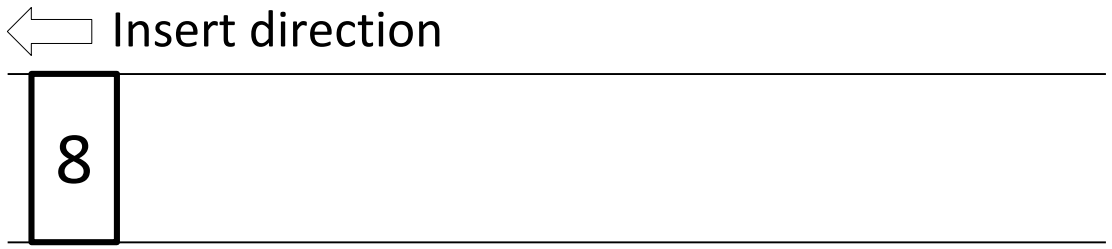

⇦ Insert direction

# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Output: 0 1 4 5 6 7 9

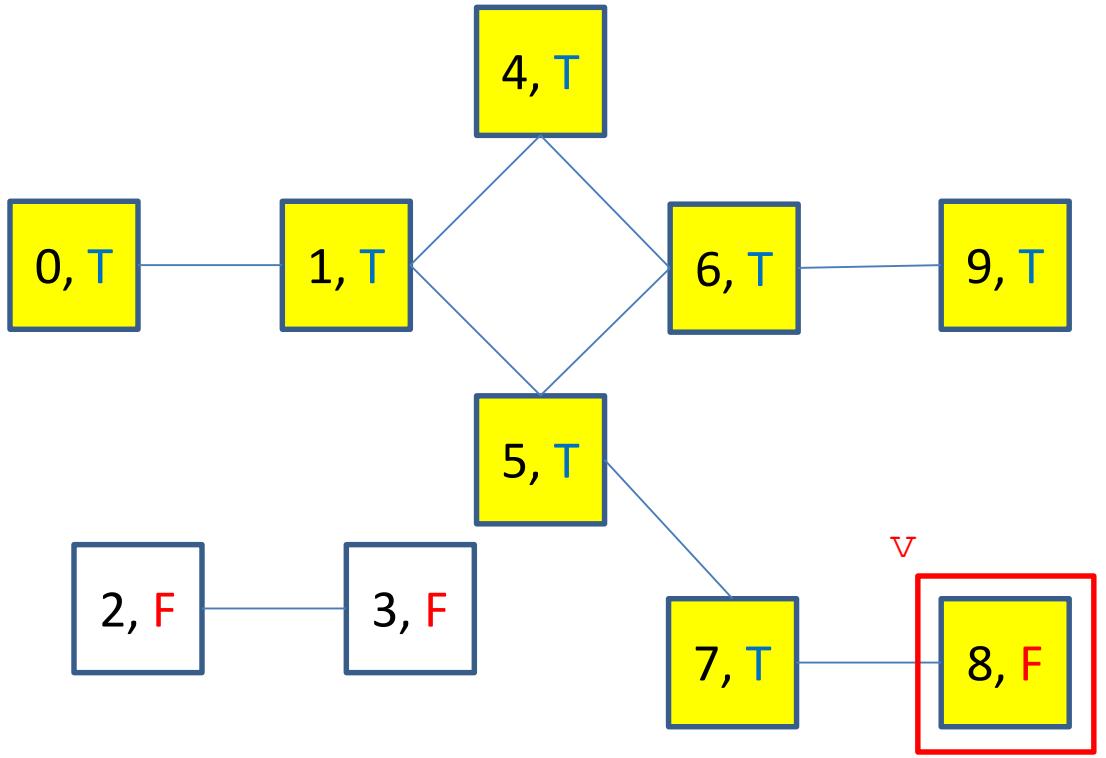


# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Output: 0 1 4 5 6 7 9

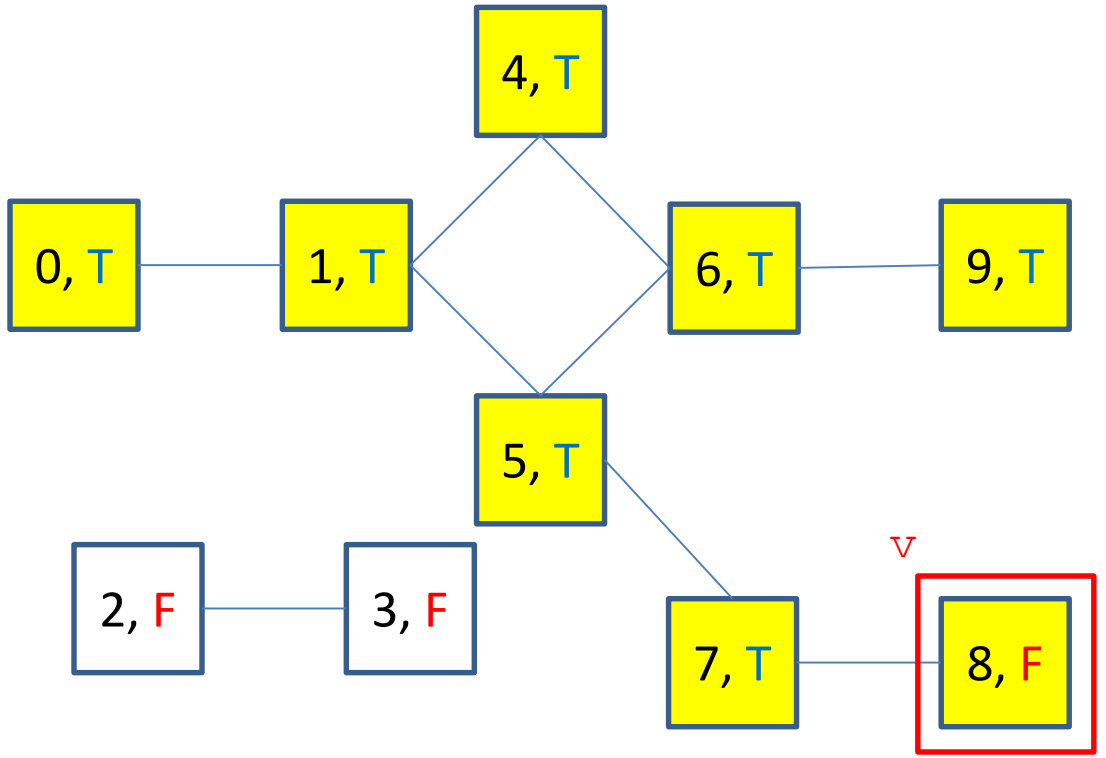



⇦ Insert direction

# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Output: 0 1 4 5 6 7 9

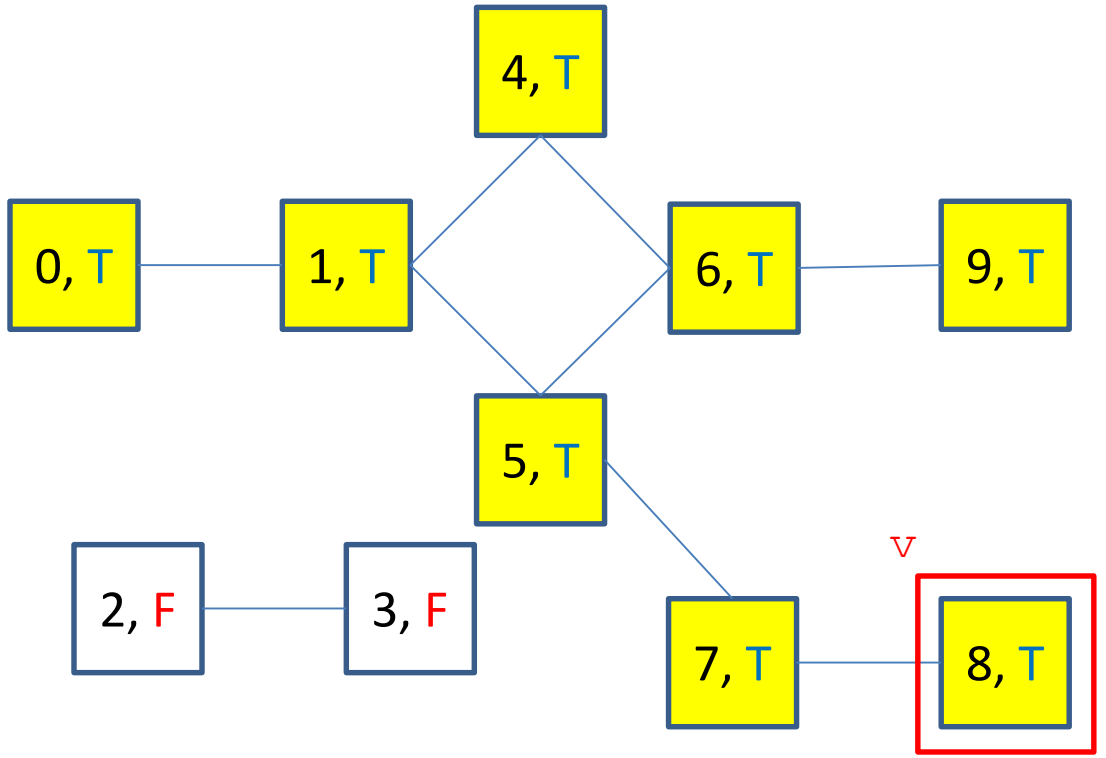



⇦ Insert direction

# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Output: 0 1 4 5 6 7 9 8

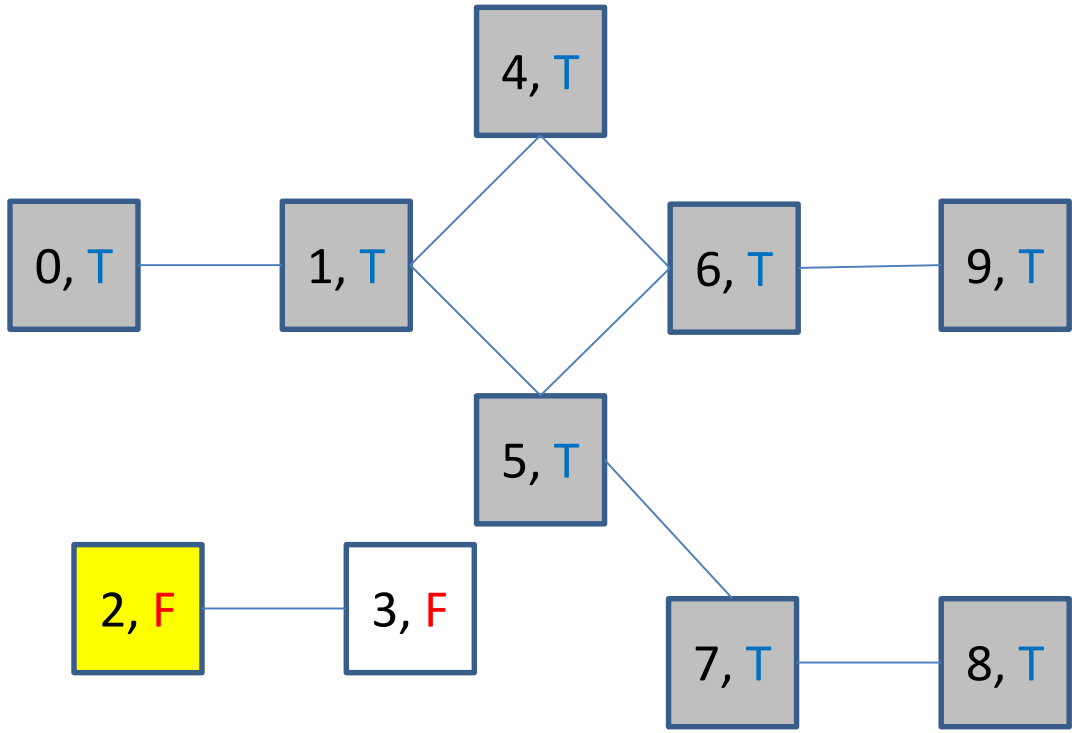


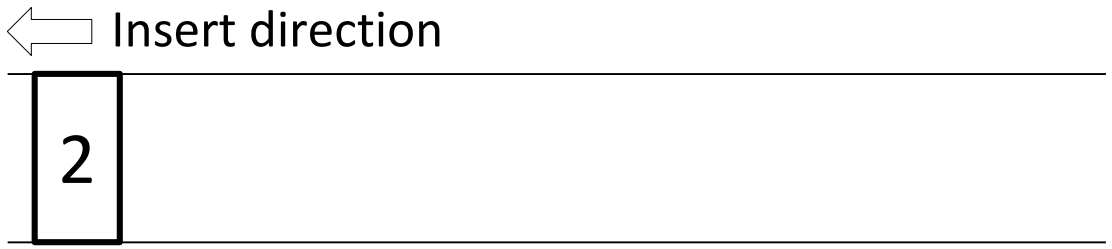

⇦ Insert direction

# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Output: 0 1 4 5 6 7 9 8

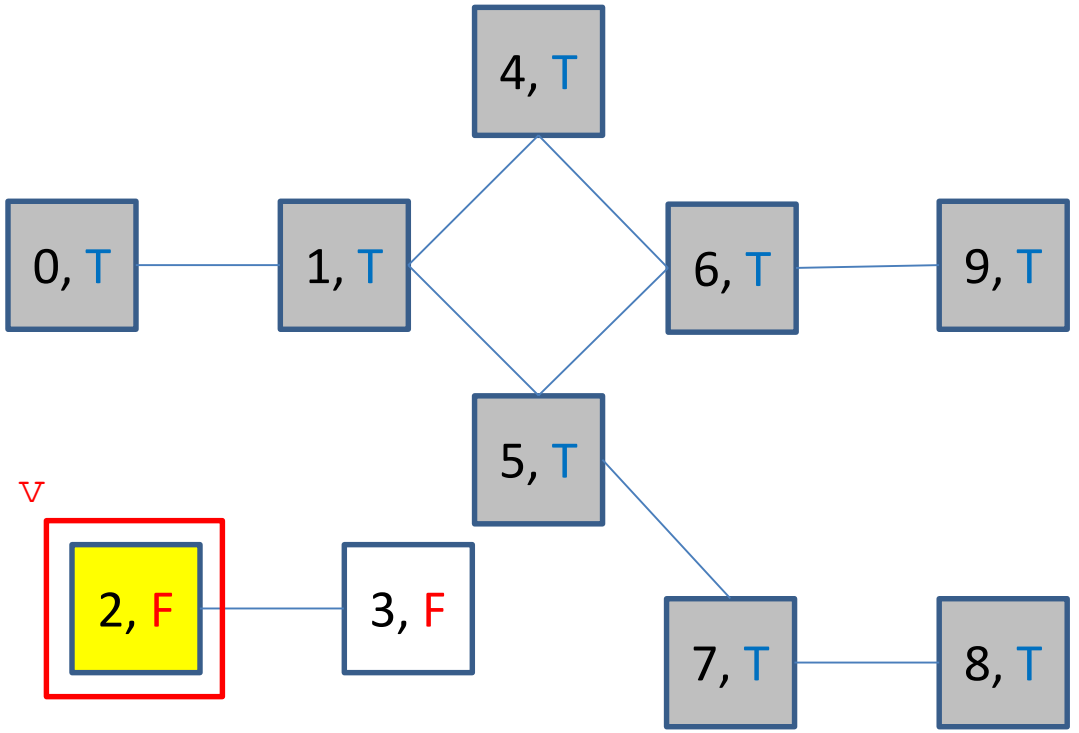


# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Output: 0 1 4 5 6 7 9 8

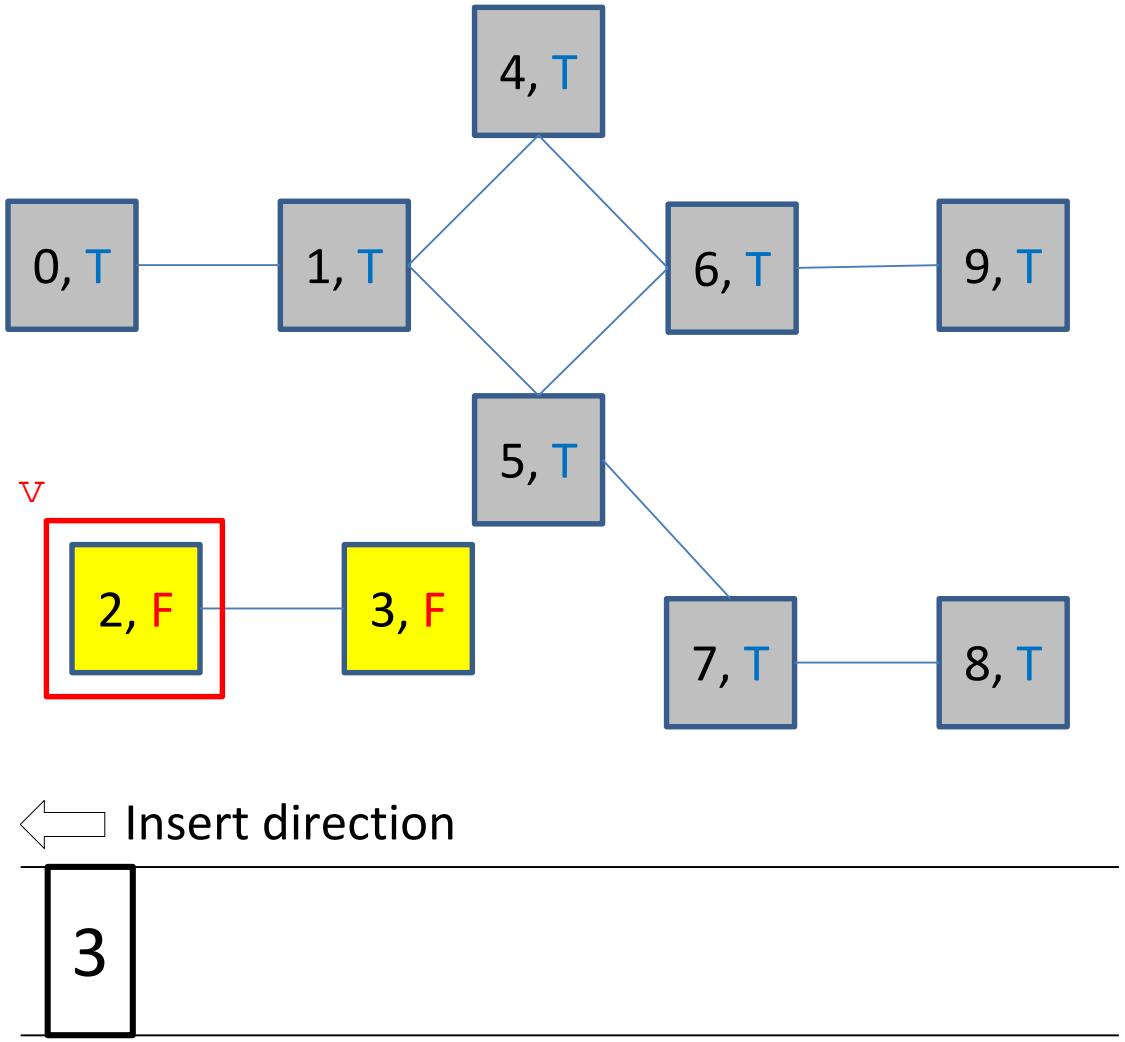



⇦ Insert direction

# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

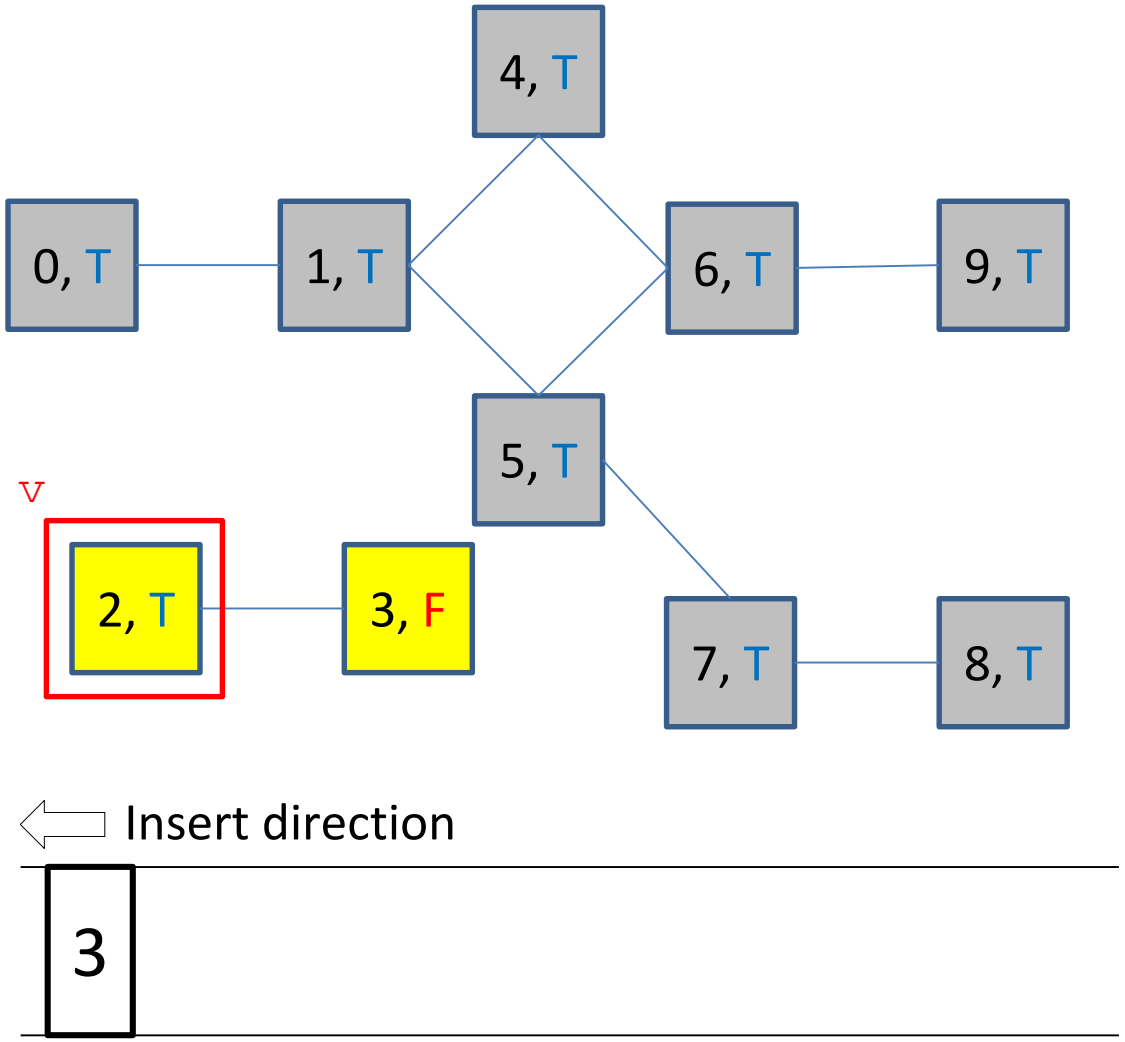
Output: 0 1 4 5 6 7 9 8



# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

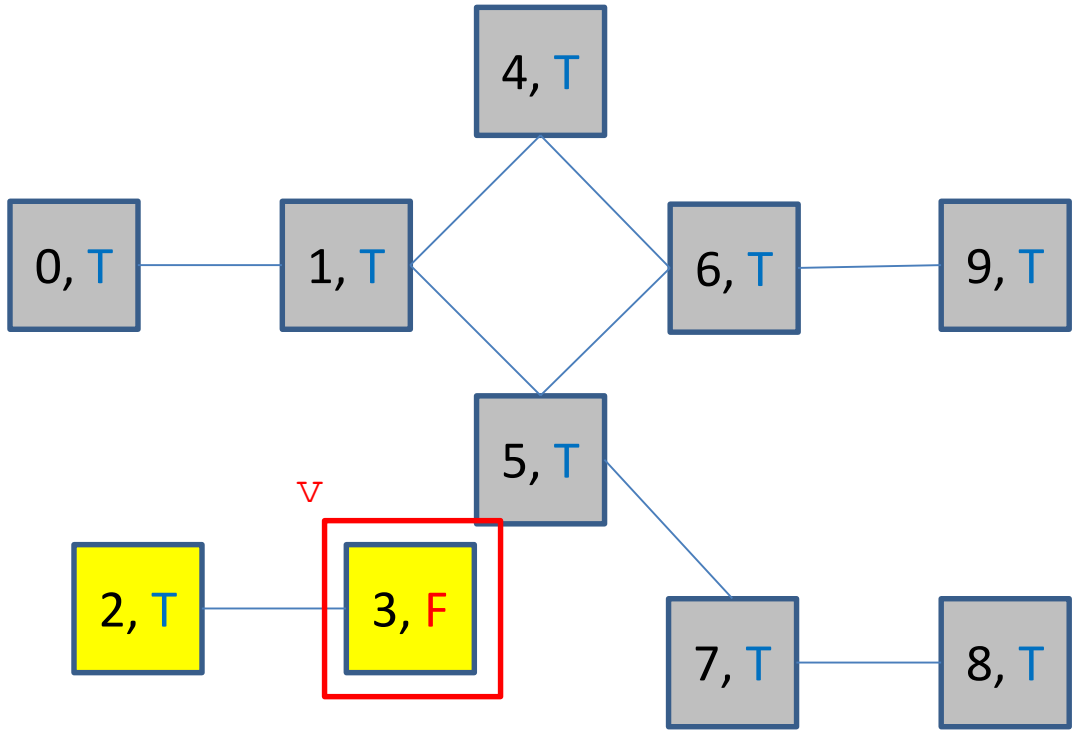
Output: 0 1 4 5 6 7 9 8 2



# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Output: 0 1 4 5 6 7 9 8 2

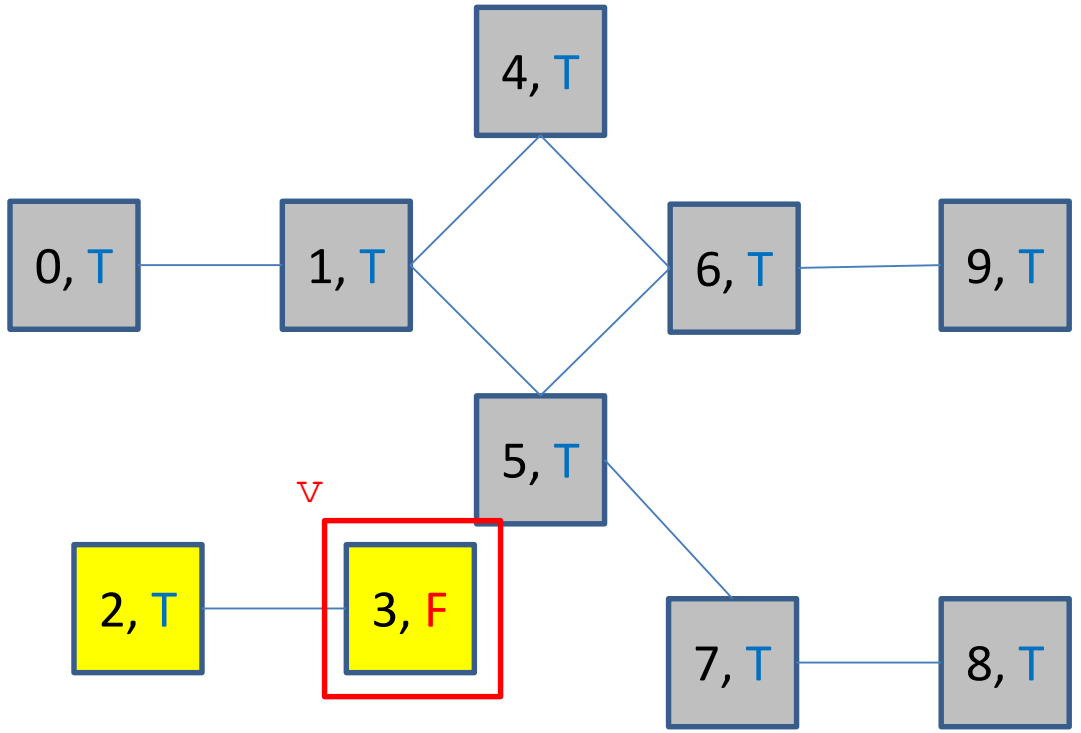



⇦ Insert direction

# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Output: 0 1 4 5 6 7 9 8 2

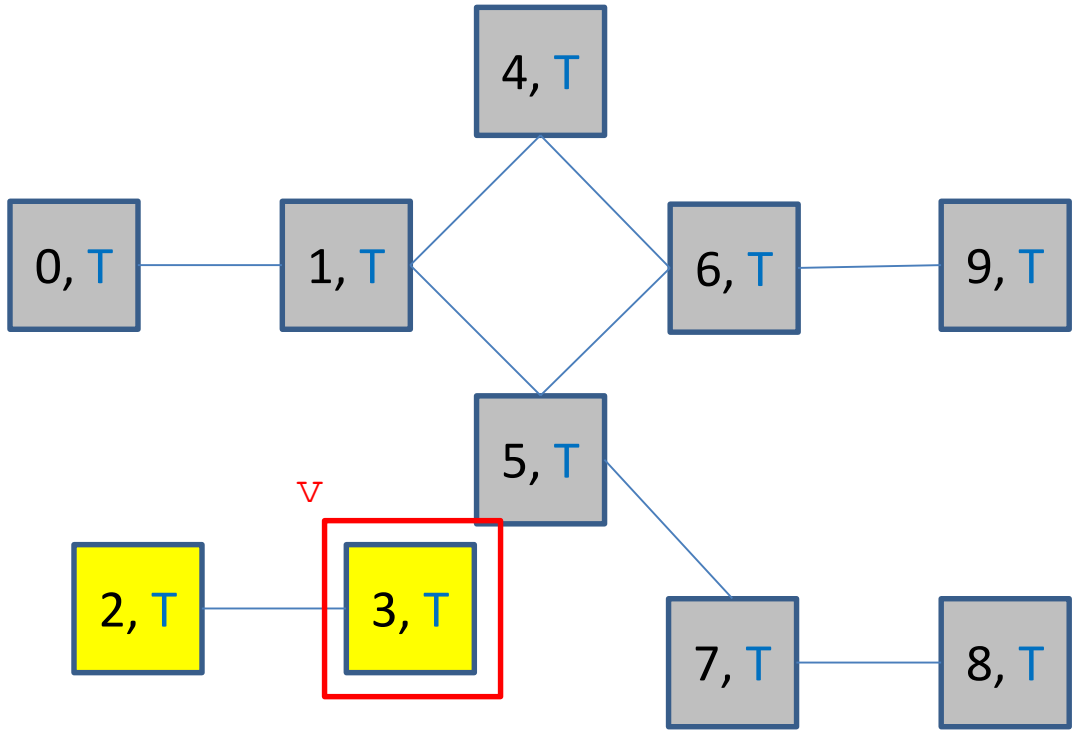



⇦ Insert direction

# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        if not visited[neighbor]:
                            q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Output: 0 1 4 5 6 7 9 8 2 3

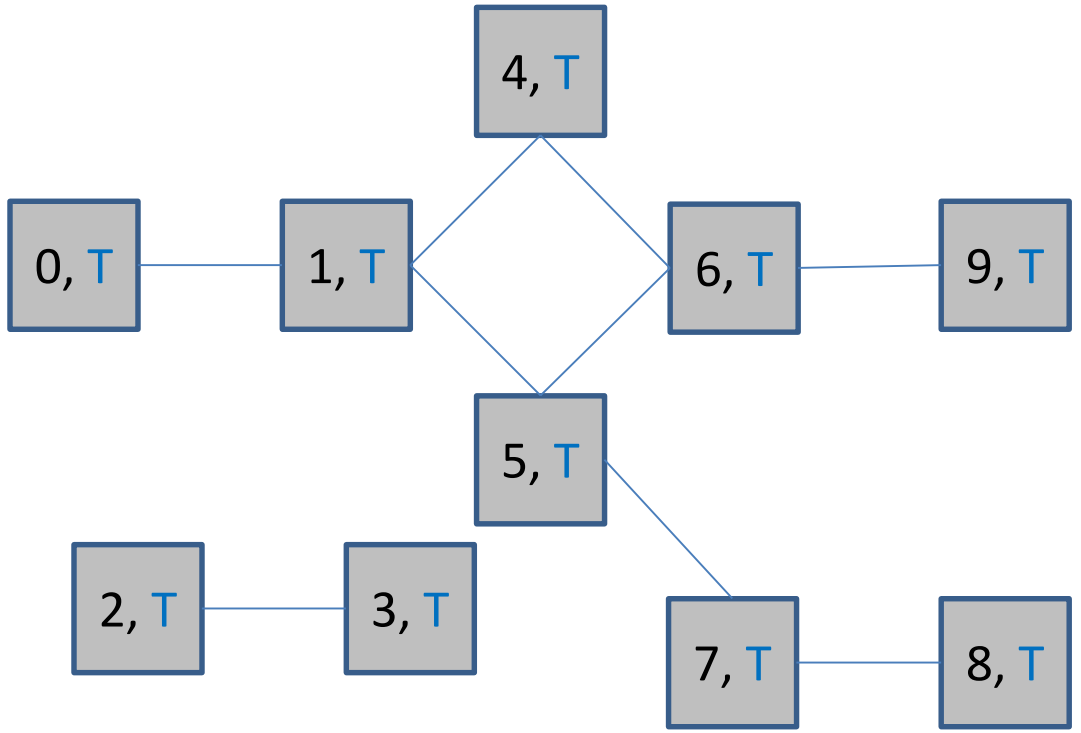



⇦ Insert direction

# 1. Implement BFT

```
def BFT(self):
    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]:
                    for neighbor in self.neighbors[v]:
                        q.append(neighbor)
                    visited[v] = True
                    print(v, end = ' ')
```

Output: 0 1 4 5 6 7 9 8 2 3

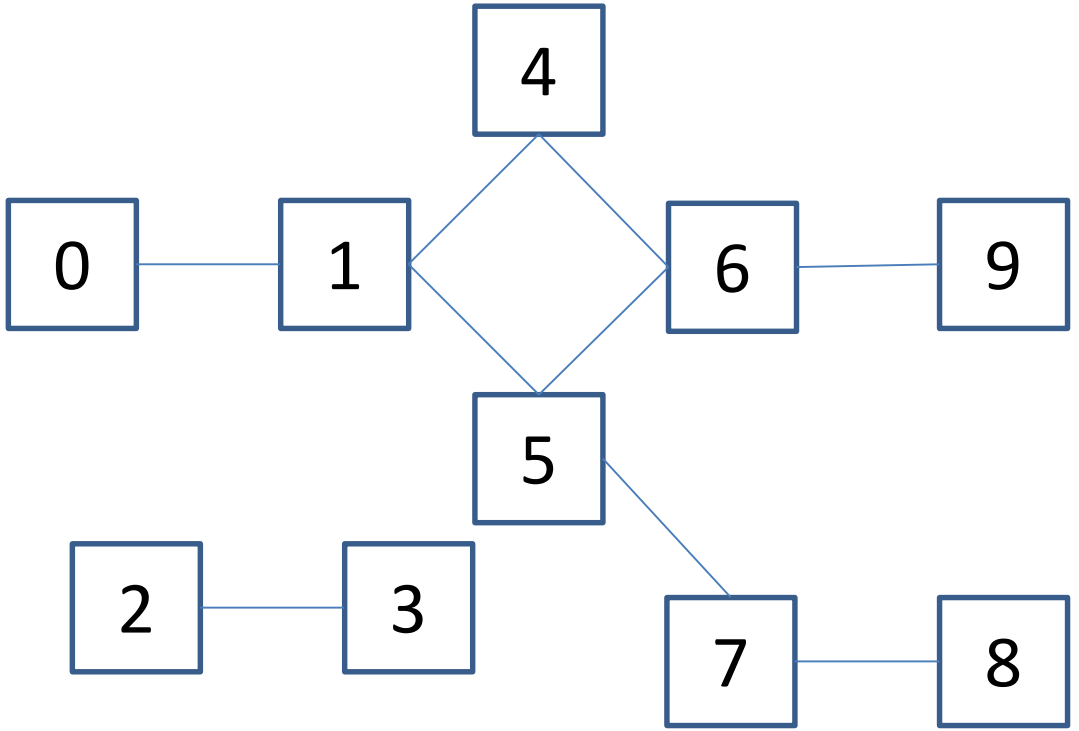



⇦ Insert direction

# 2. Implement DFT Post-order

**Implement a post-order DFT function for a given graph**

- Use recursion for this task
- Use `visited` as a dictionary that marks visited nodes as `True`
- Code for `DFT()` is given, complete the recursive `__DFTHelp()` method.




Ans: `8 7 5 9 6 4 1 0 3 2`

# 2. Implement DFT Post-order

```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```

# 2. Implement DFT Post-order

```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')
```

```
def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```

Same as in lecture notes

# 2. Implement DFT Post-order

```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```

Print at the end of iteration

Post-order DFT implies printing child first, then parent

- Mark vertex as visited first to avoid duplicates
- **Traverse to children nodes first**
- Then print the node

# 2. Implement DFT Pre- vs. Post-order

```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        print(v, end = ' ')
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```

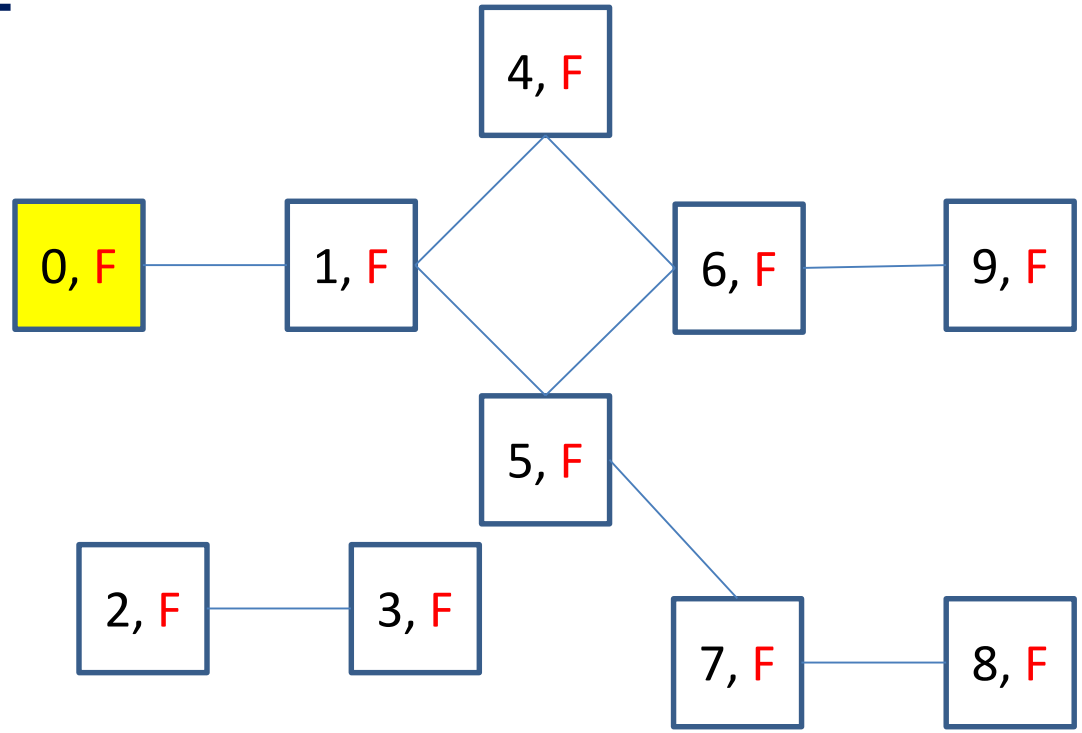
Pre-order

```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```

Post-order

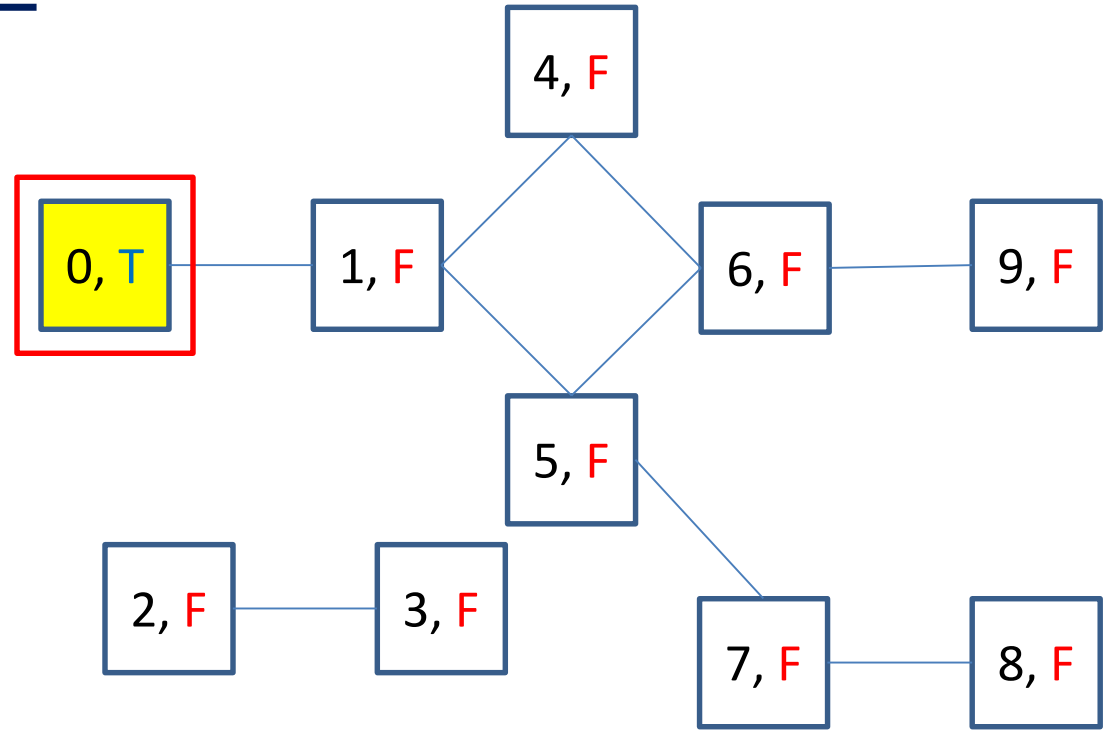
# 2. Implement DFT Post-order

```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```



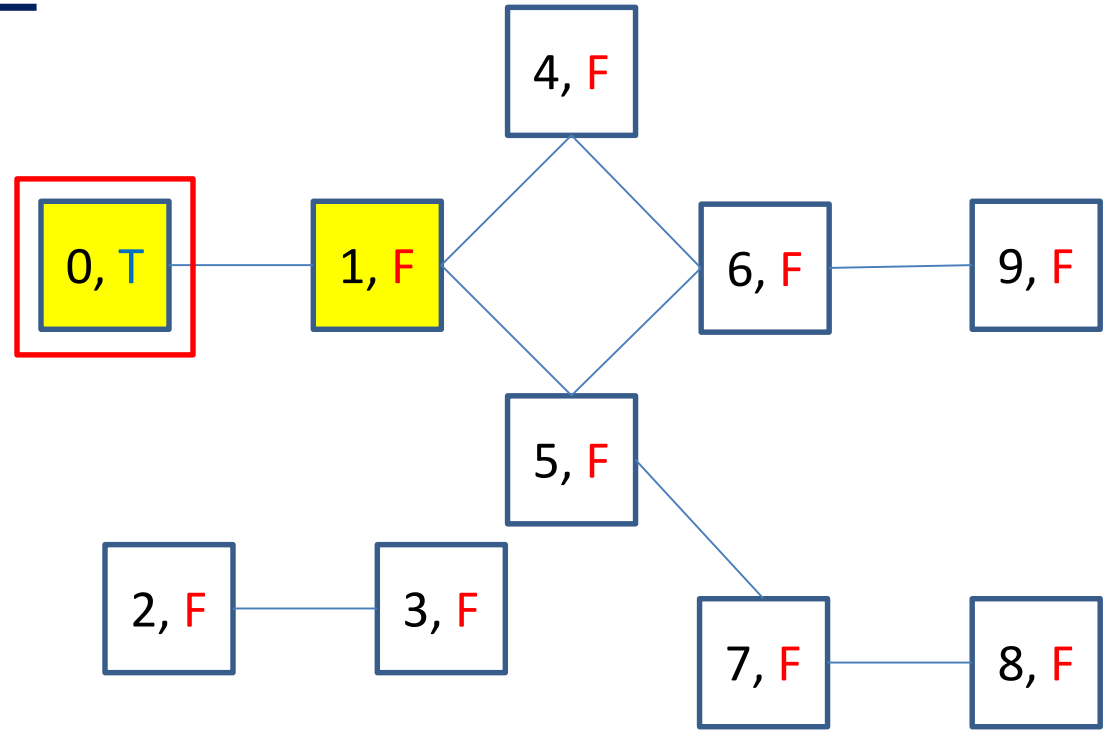
# 2. Implement DFT Post-order

```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```



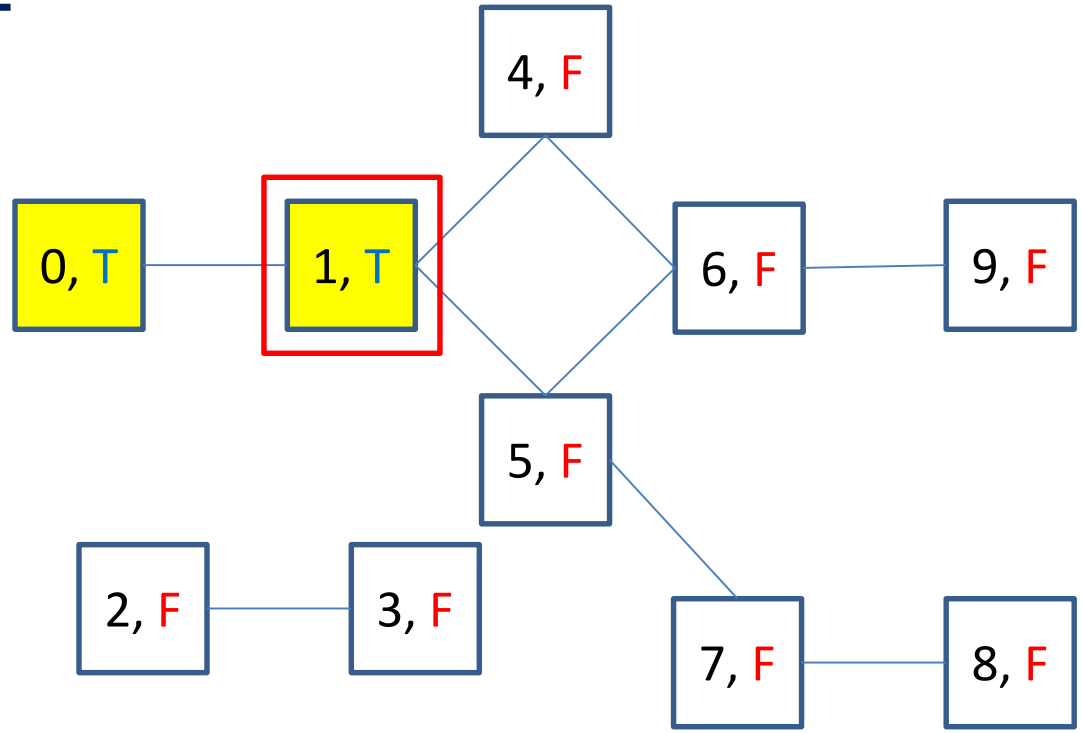
# 2. Implement DFT Post-order

```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```



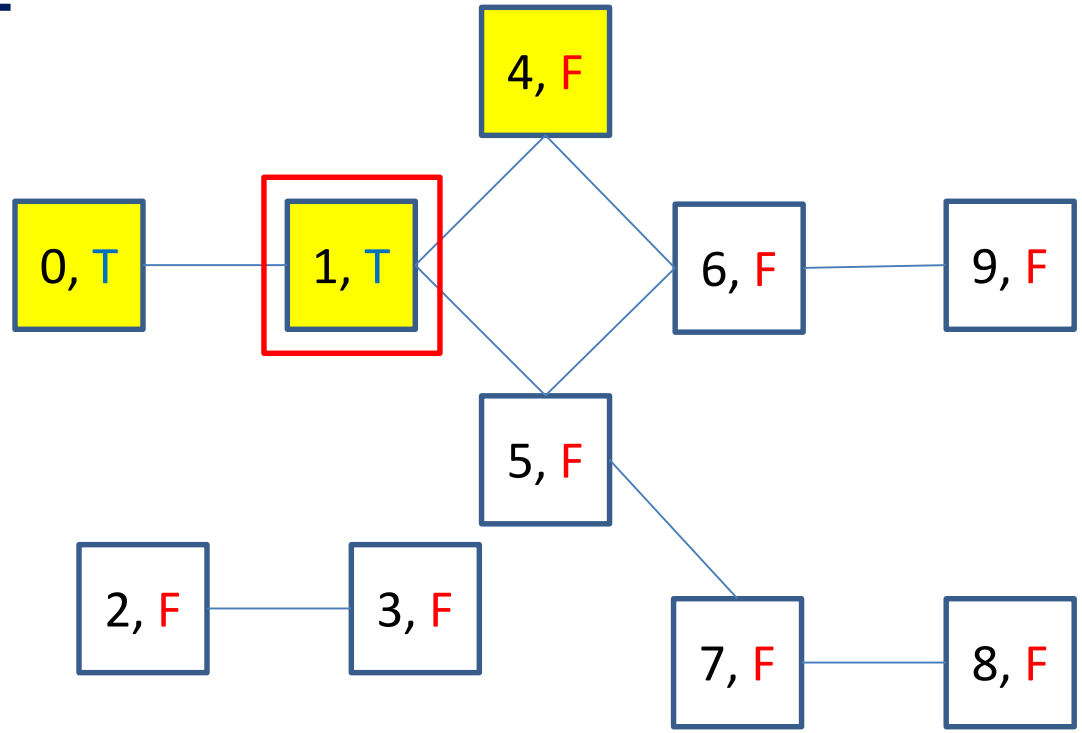
# 2. Implement DFT Post-order

```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```



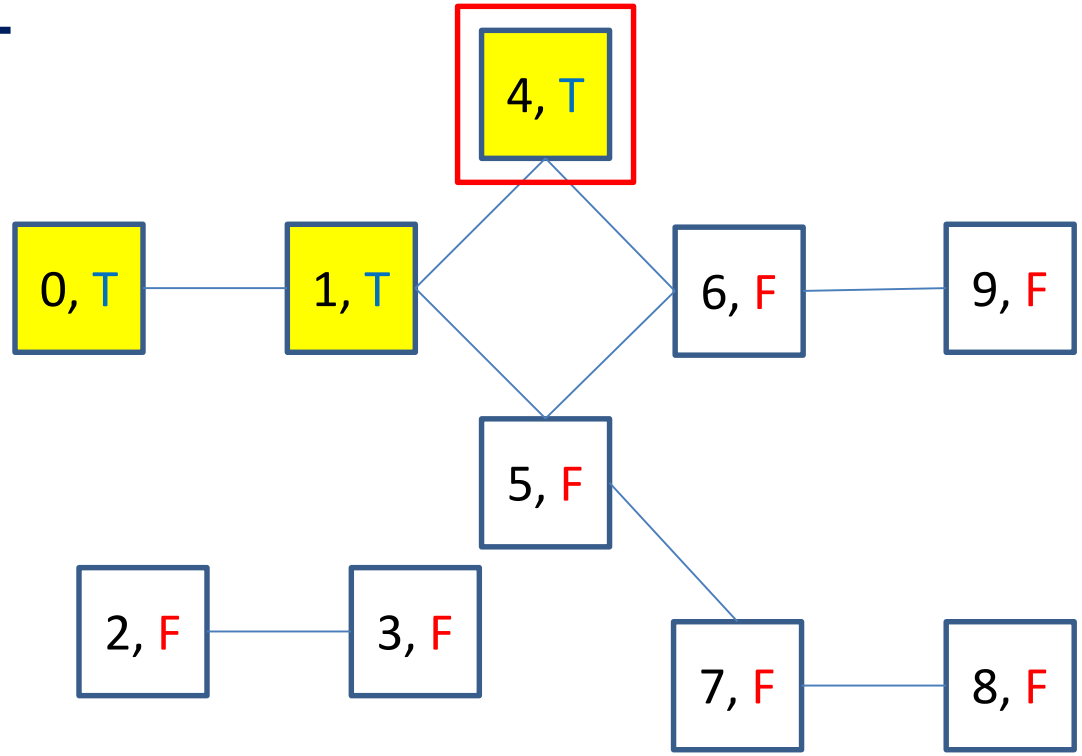
# 2. Implement DFT Post-order

```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```



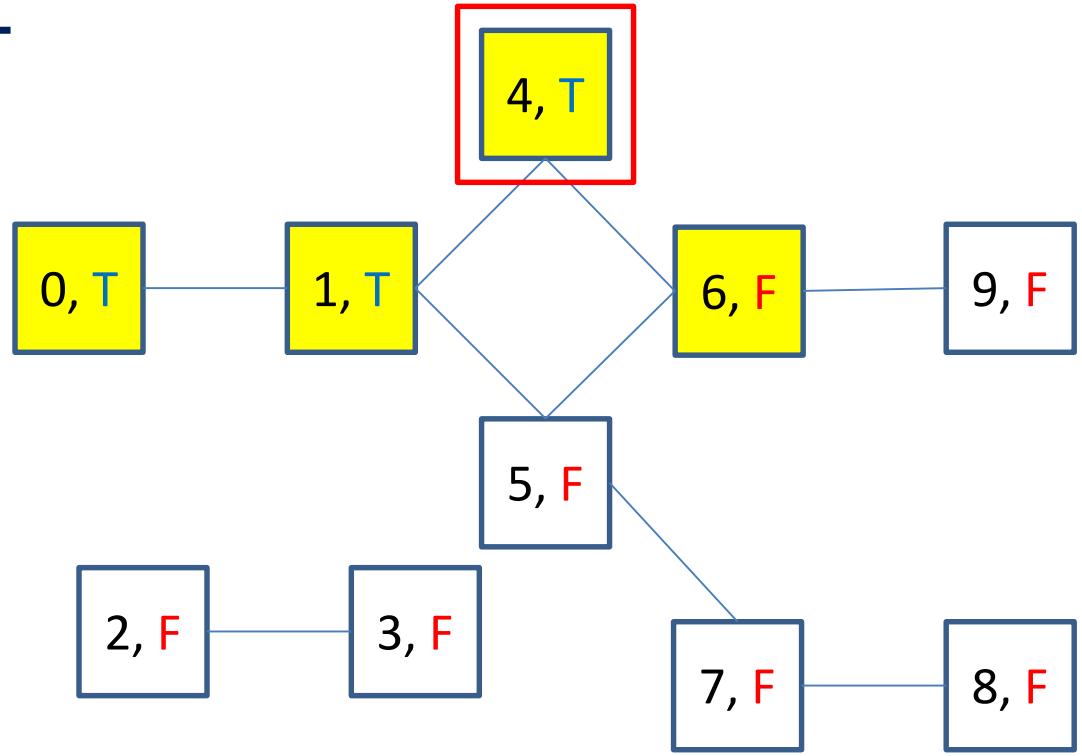
# 2. Implement DFT Post-order

```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```



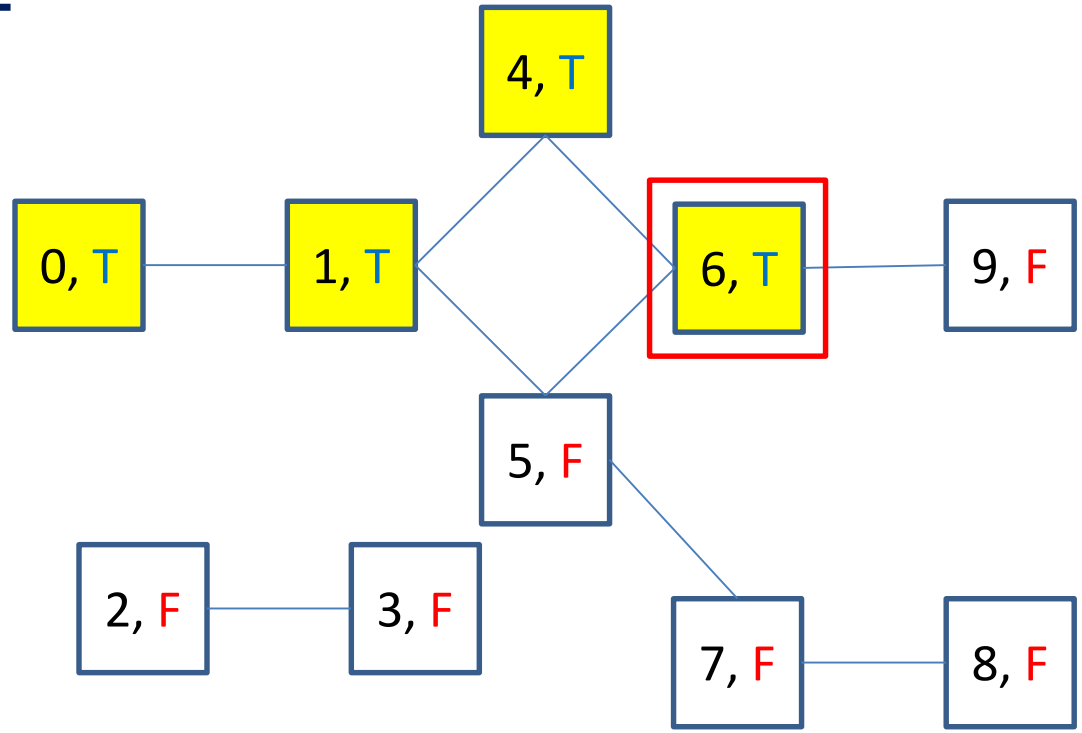
# 2. Implement DFT Post-order

```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```



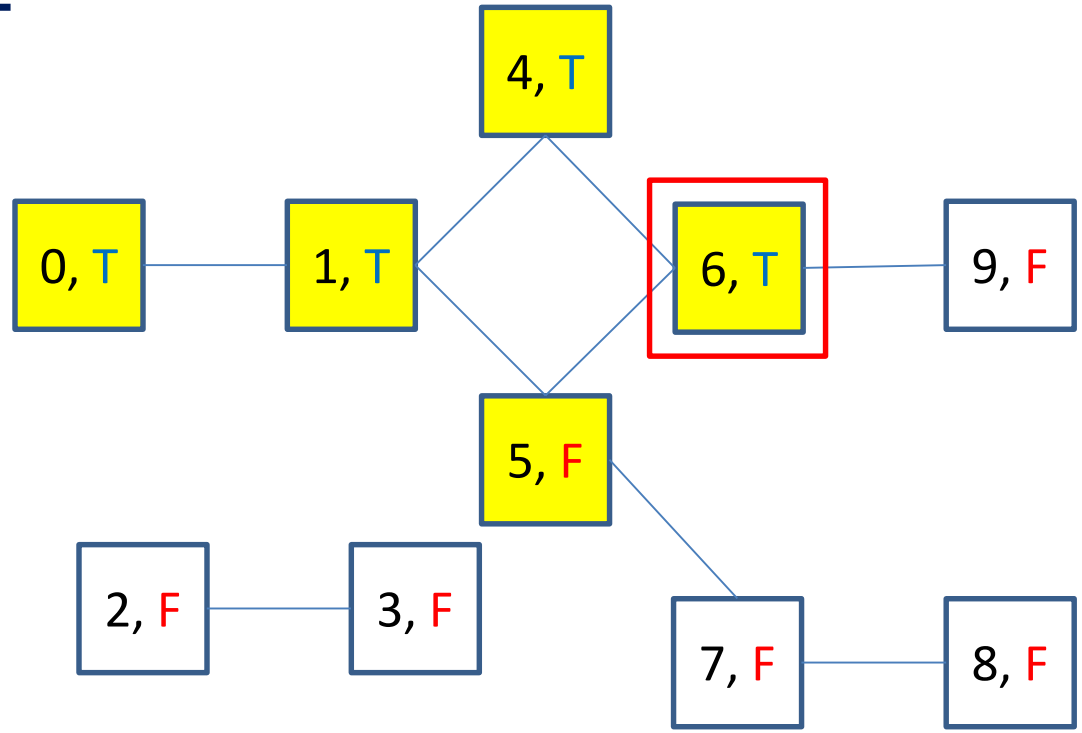
# 2. Implement DFT Post-order

```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```



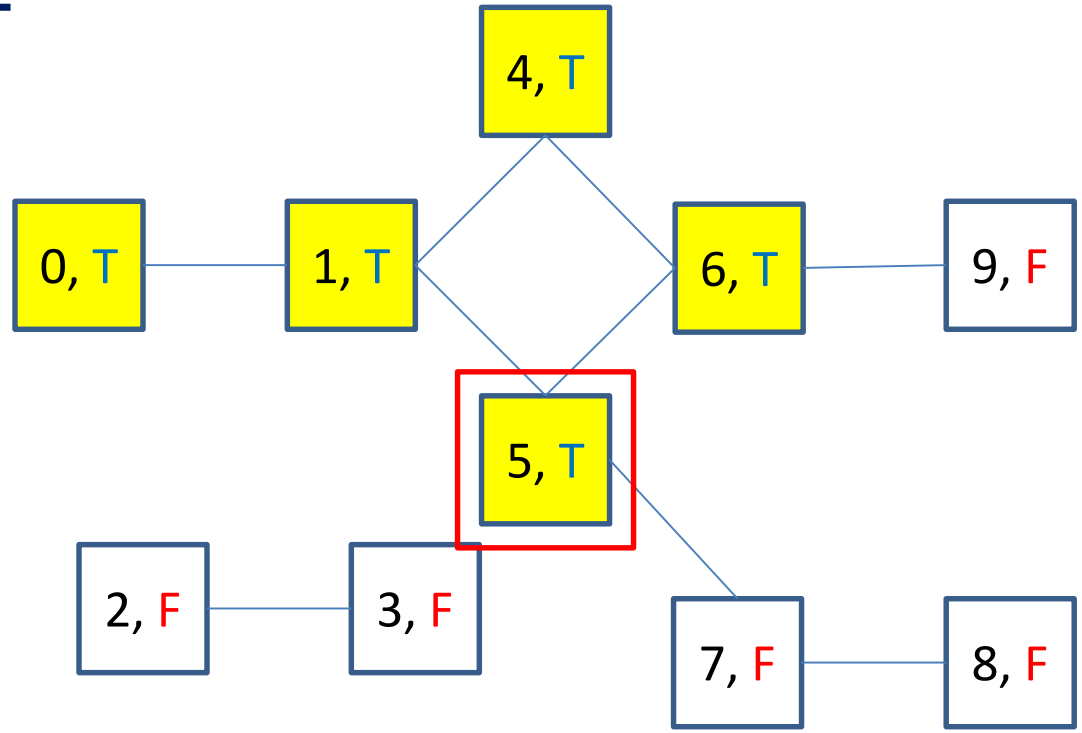
# 2. Implement DFT Post-order

```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```



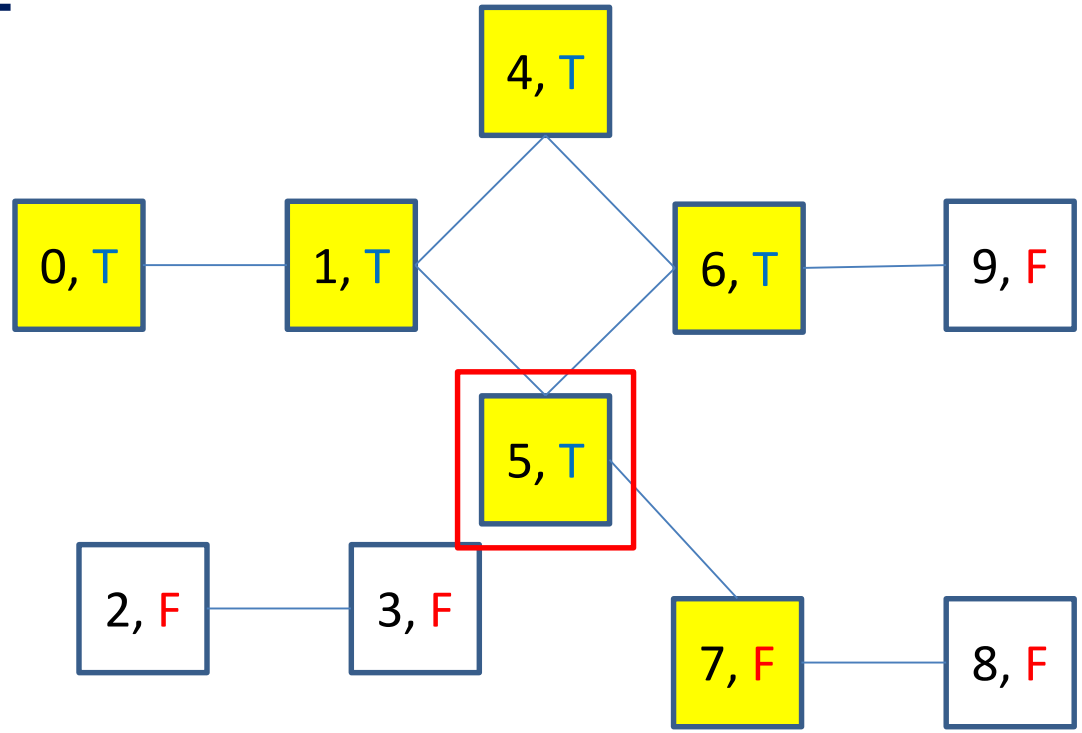
# 2. Implement DFT Post-order

```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```



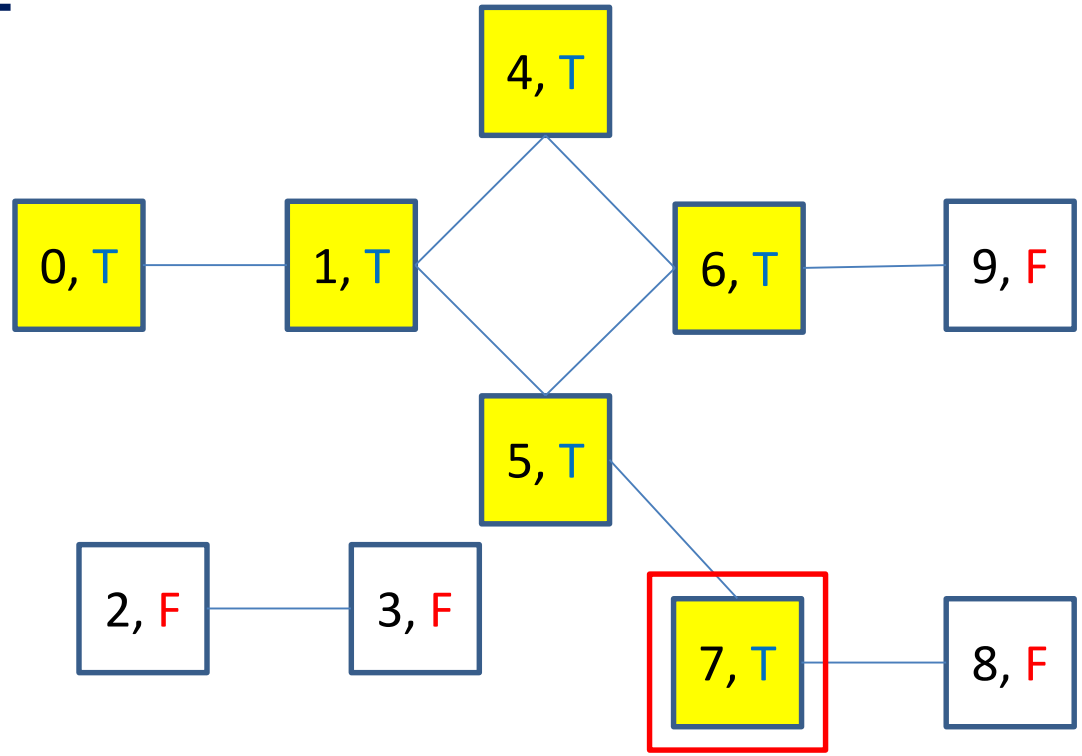
# 2. Implement DFT Post-order

```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```



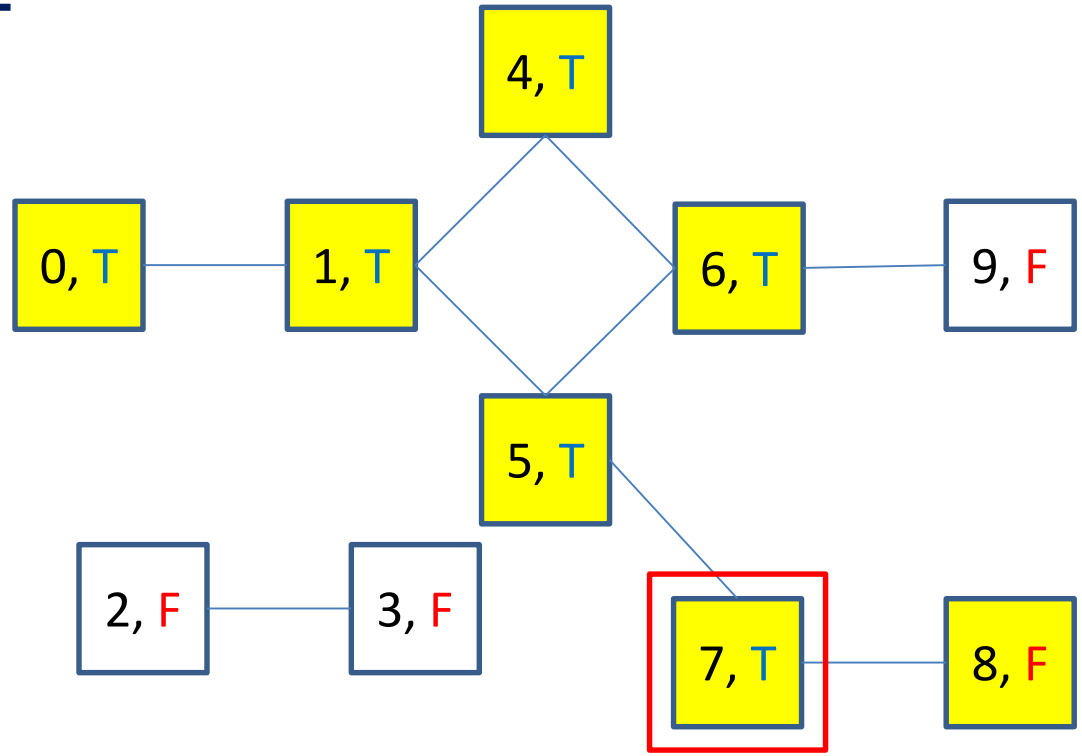
# 2. Implement DFT Post-order

```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```



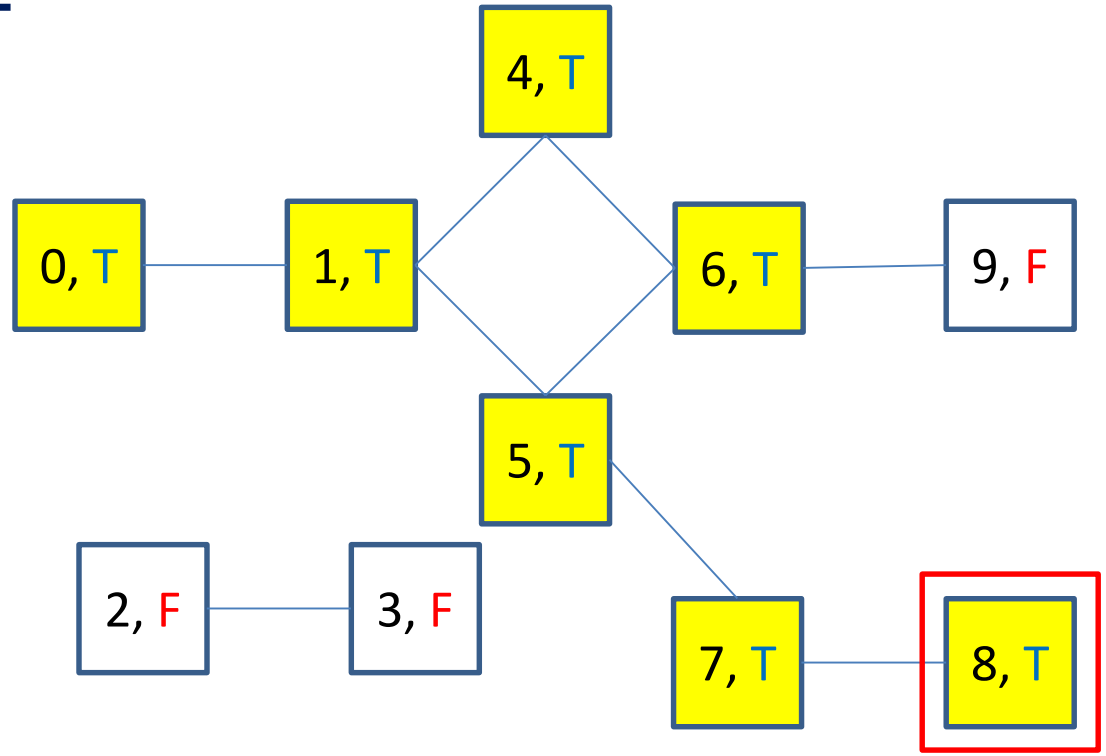
# 2. Implement DFT Post-order

```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```



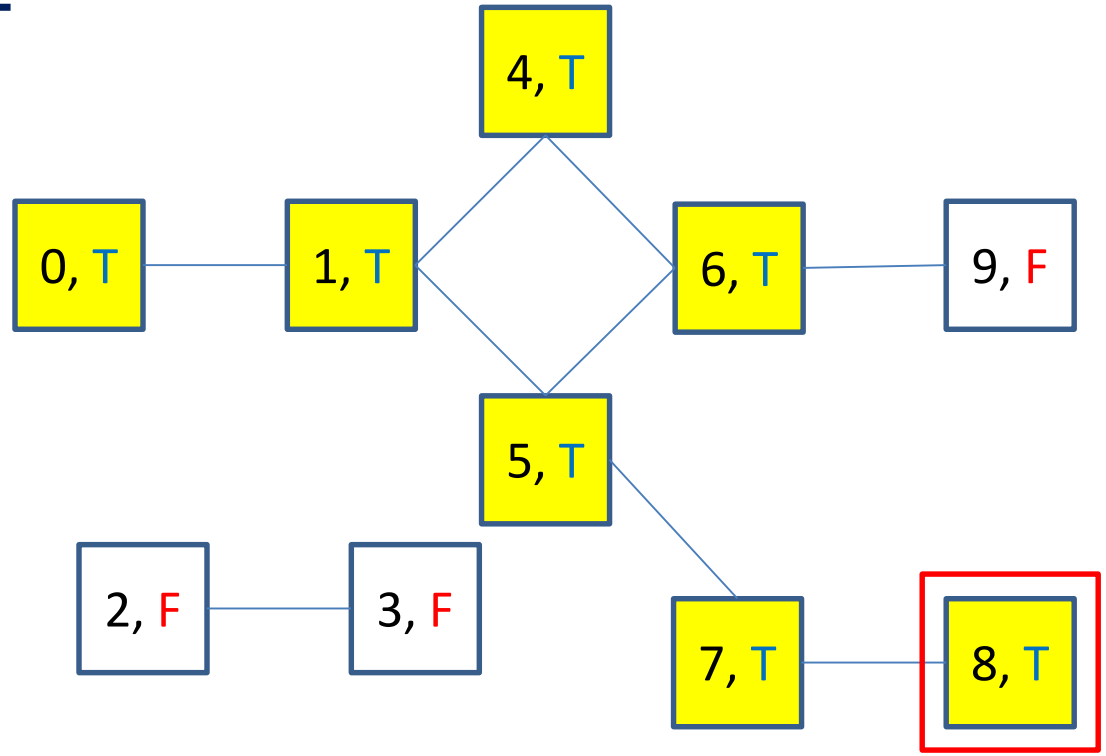
# 2. Implement DFT Post-order

```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```



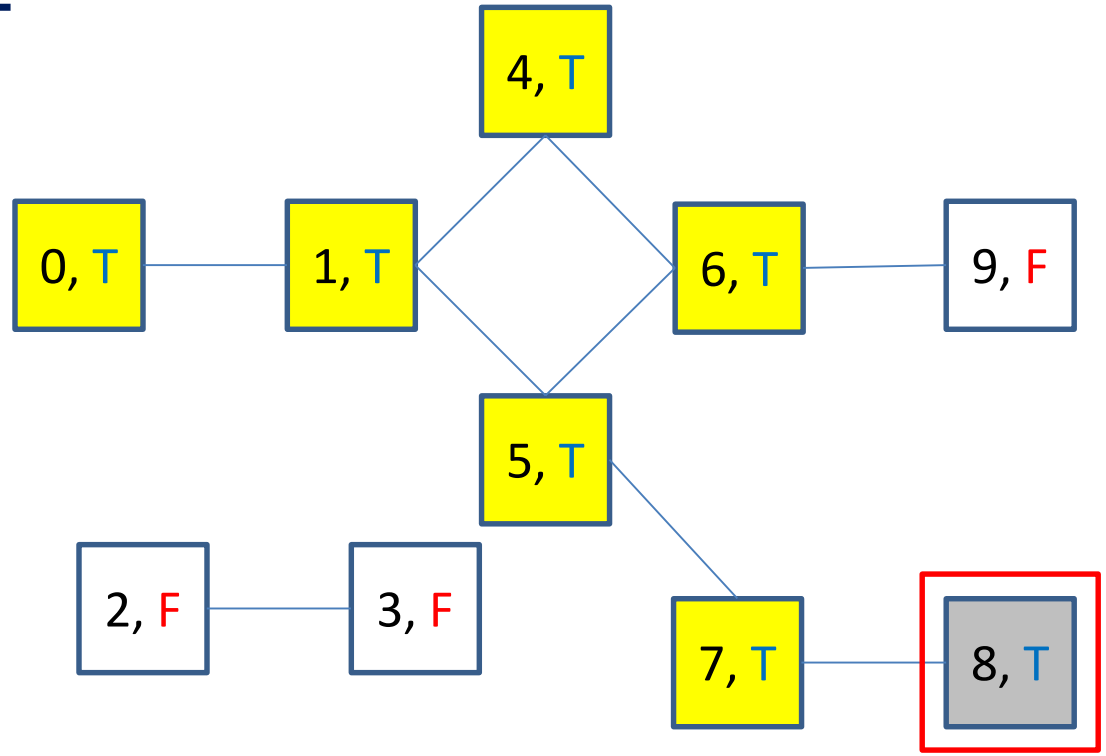
# 2. Implement DFT Post-order

```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```



# 2. Implement DFT Post-order

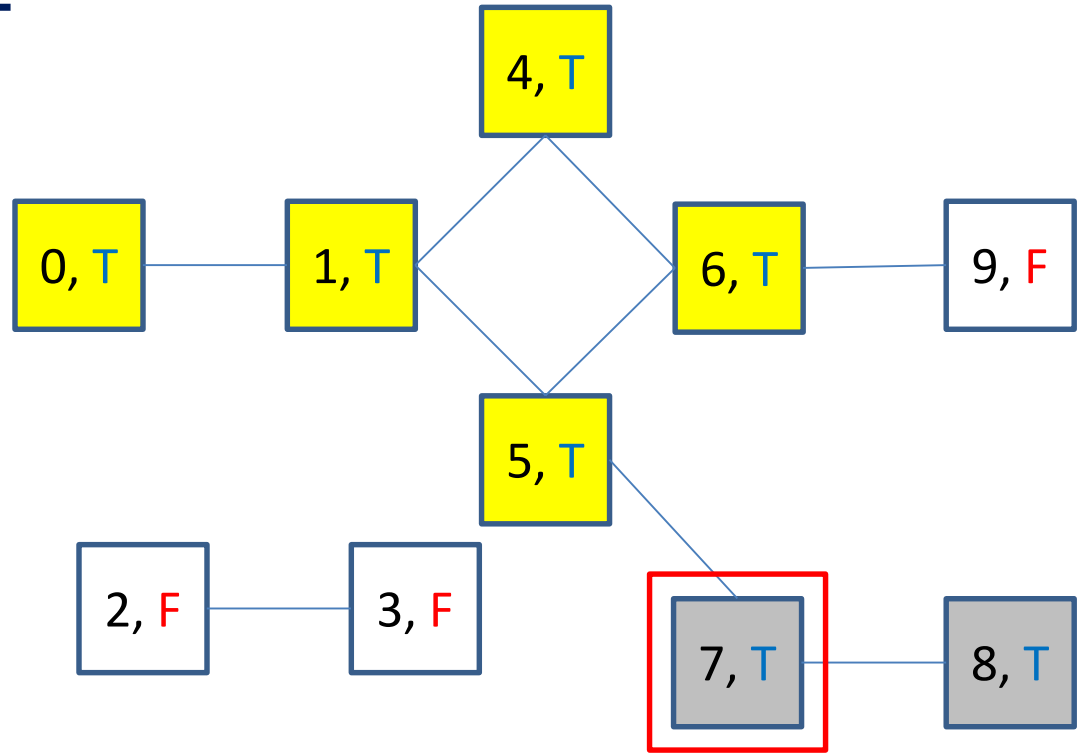
```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```




Ans: 8

# 2. Implement DFT Post-order

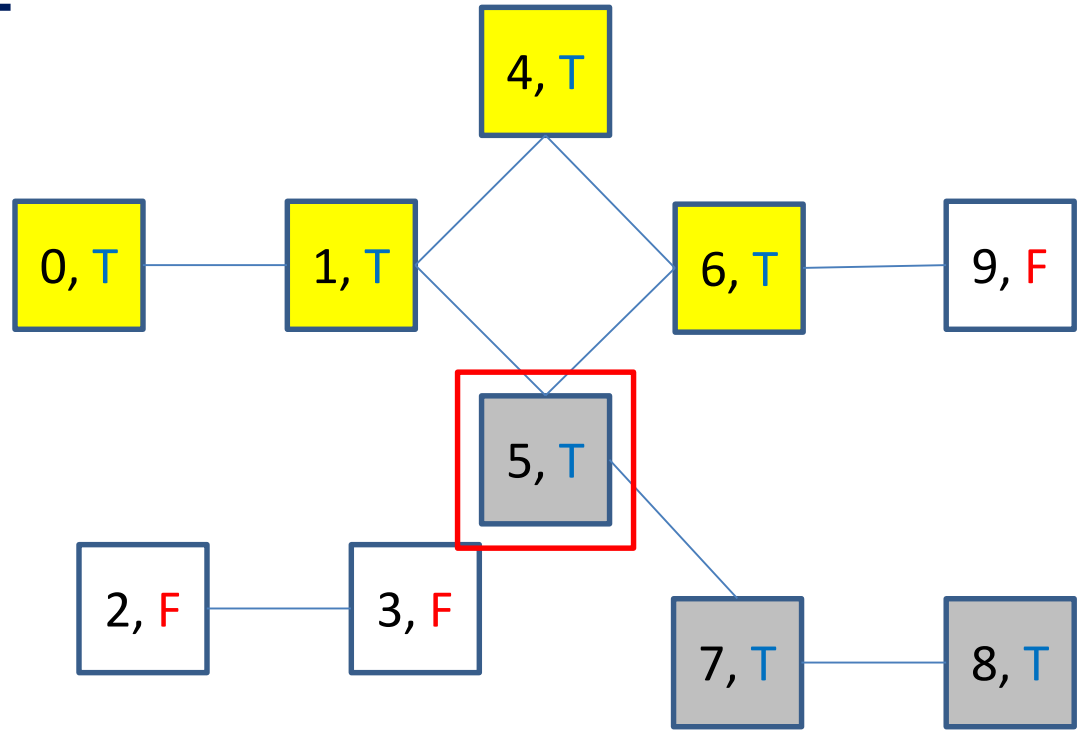
```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```




Ans: 8 7

# 2. Implement DFT Post-order

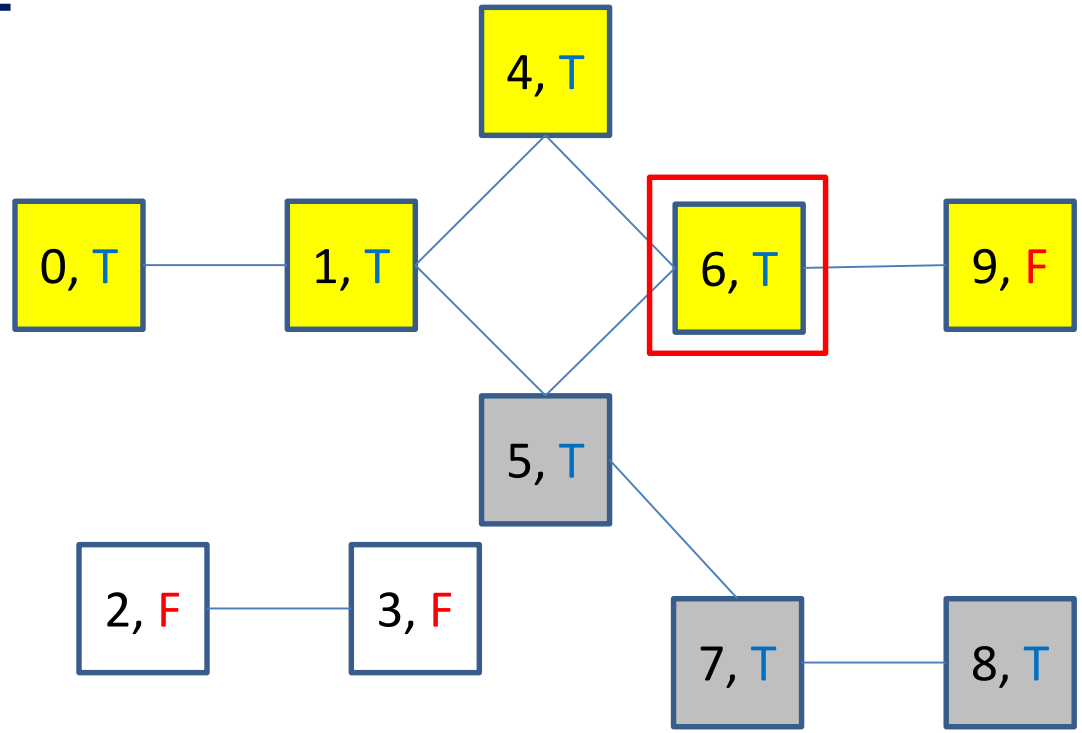
```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```




Ans: 8 7 5

# 2. Implement DFT Post-order

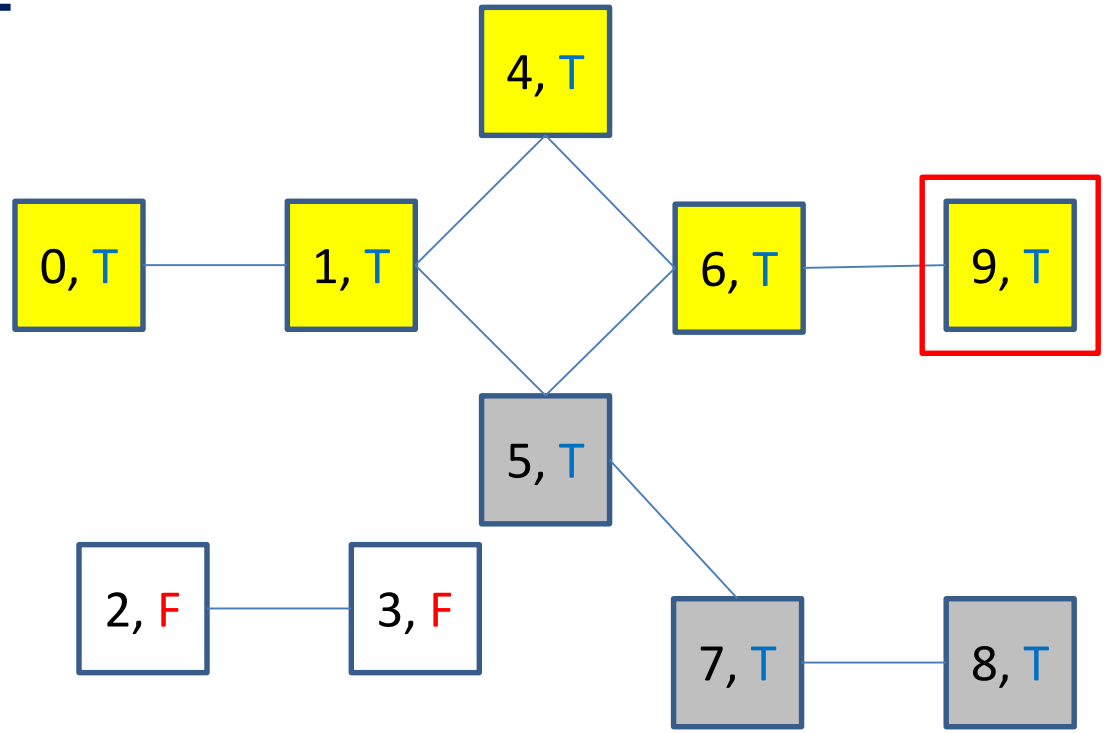
```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```




Ans: 8 7 5

# 2. Implement DFT Post-order

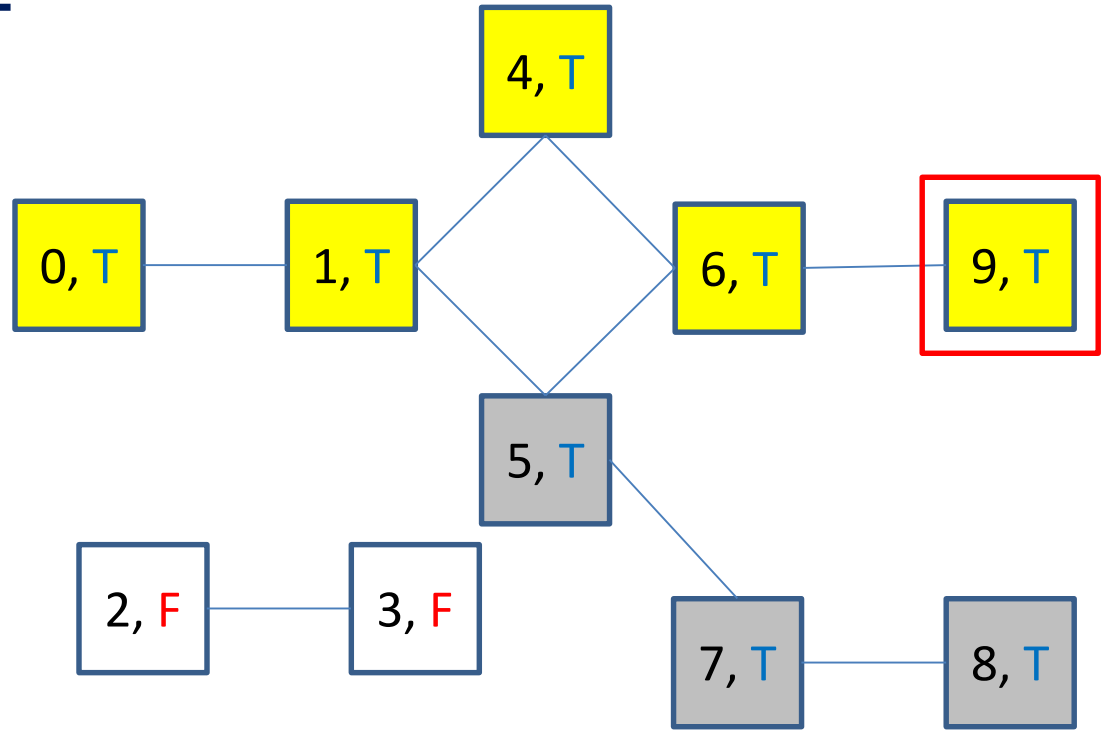
```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```




Ans: 8 7 5

# 2. Implement DFT Post-order

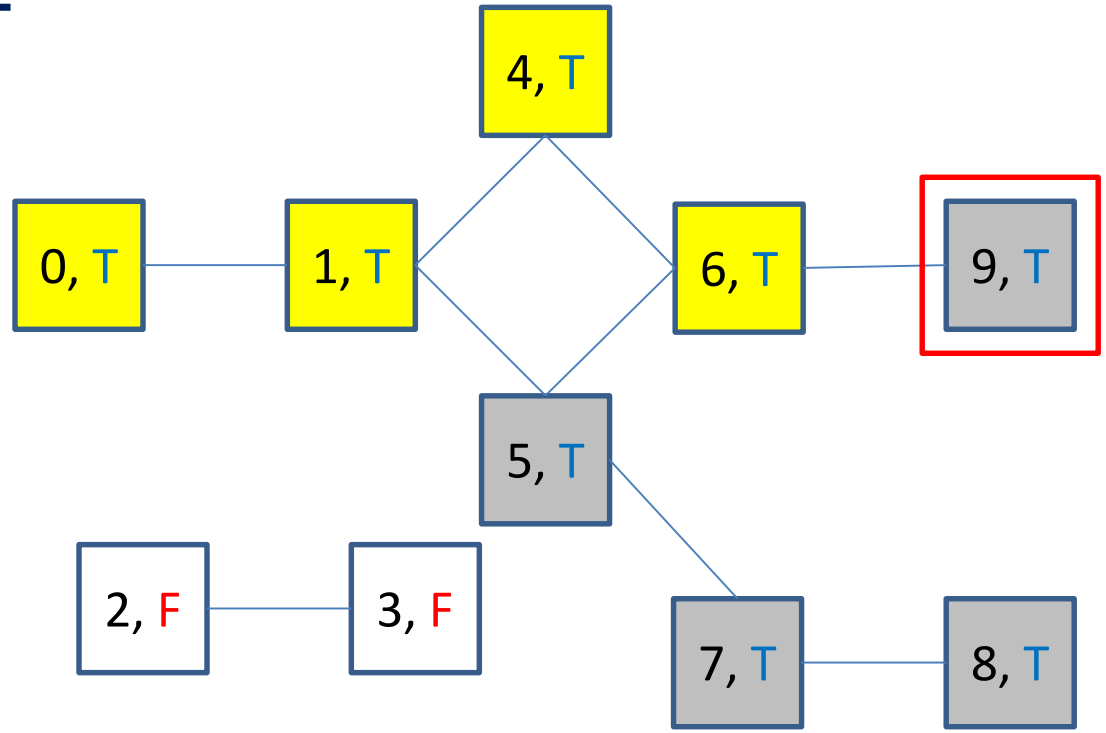
```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```




Ans: 8 7 5

# 2. Implement DFT Post-order

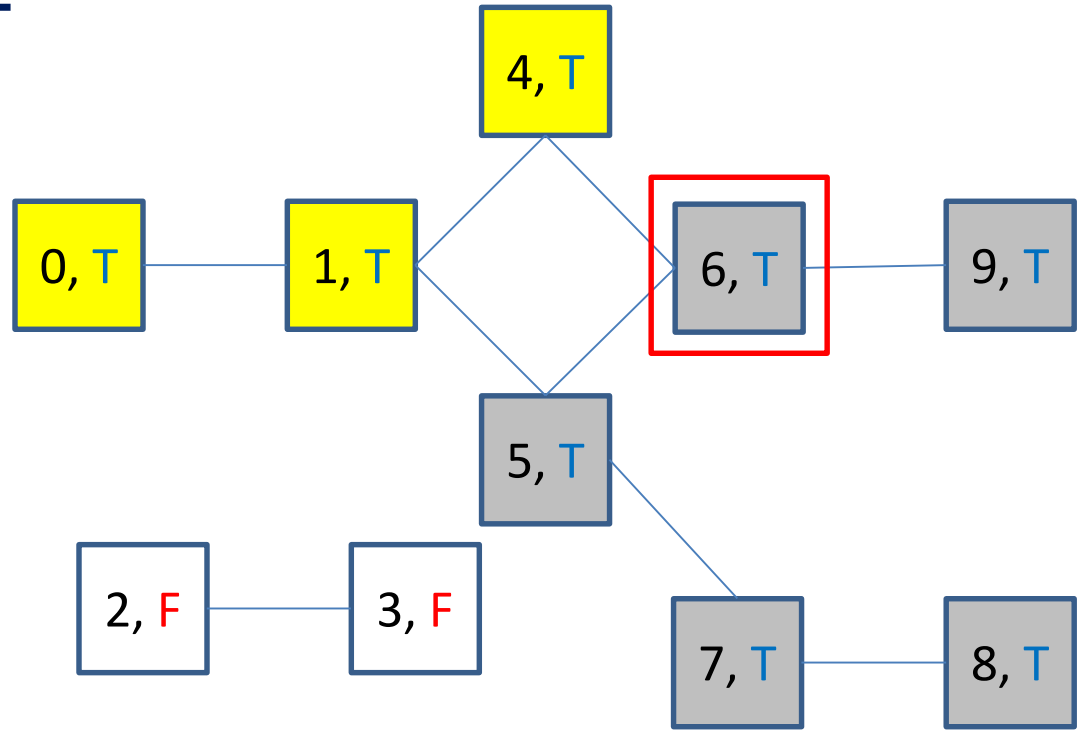
```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```




Ans: 8 7 5 9

# 2. Implement DFT Post-order

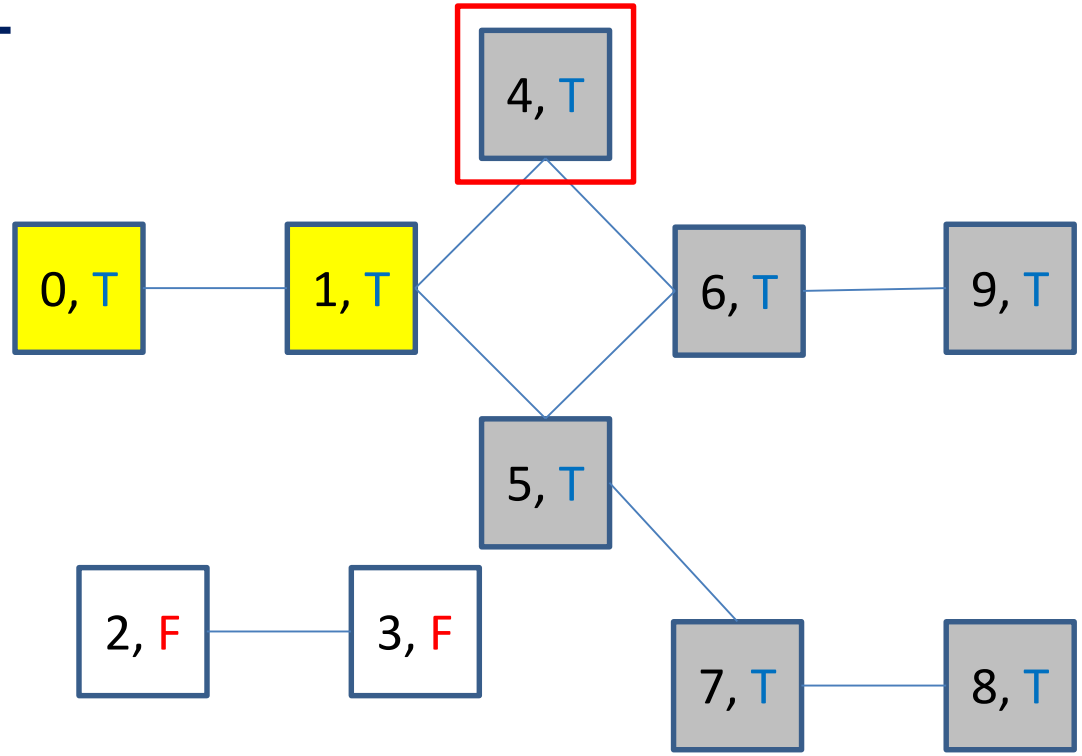
```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```




Ans: 8 7 5 9 6

# 2. Implement DFT Post-order

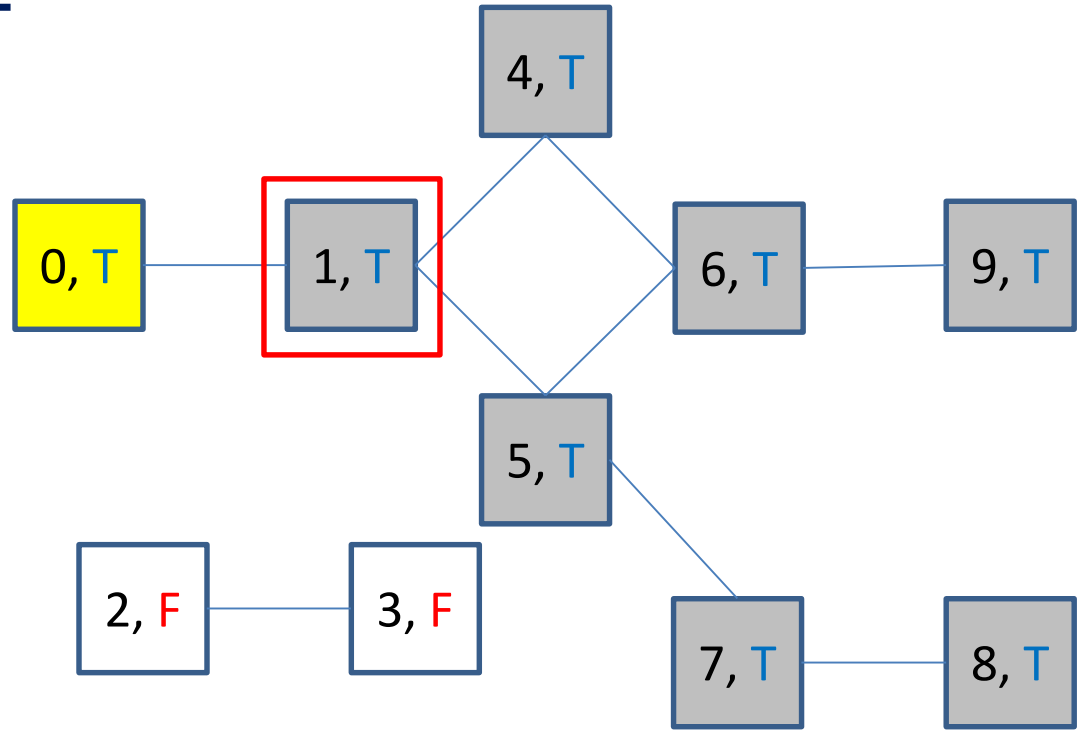
```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```




Ans: 8 7 5 9 6 4

# 2. Implement DFT Post-order

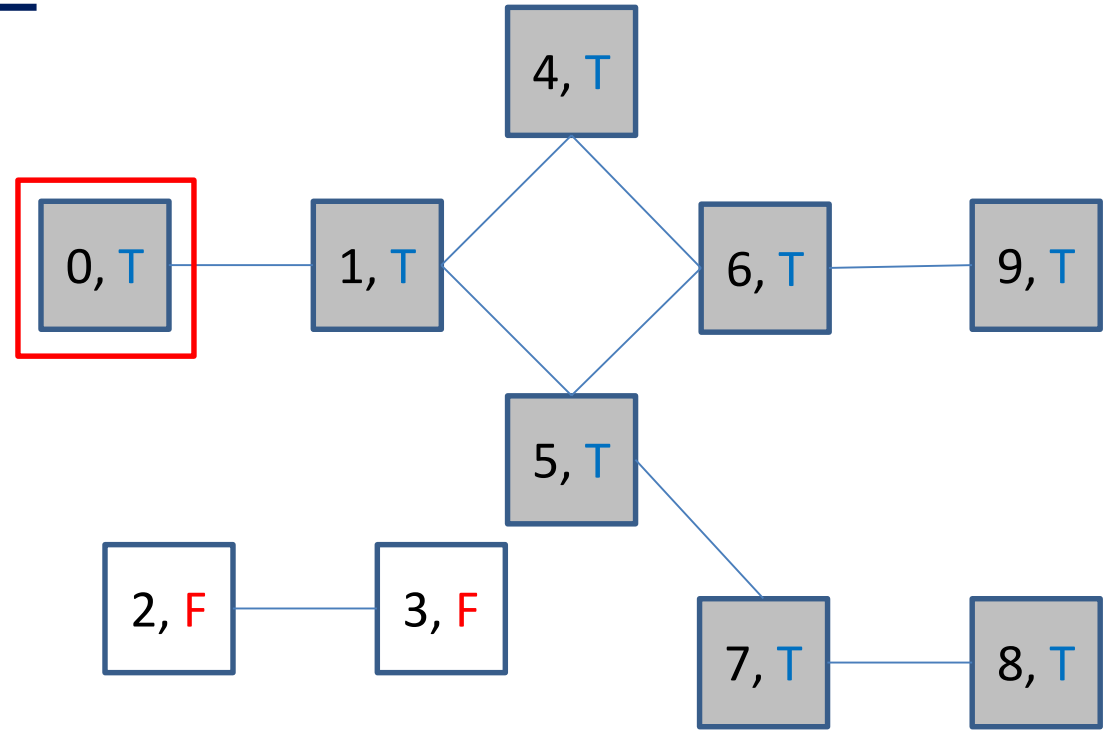
```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```




Ans: 8 7 5 9 6 4 1

# 2. Implement DFT Post-order

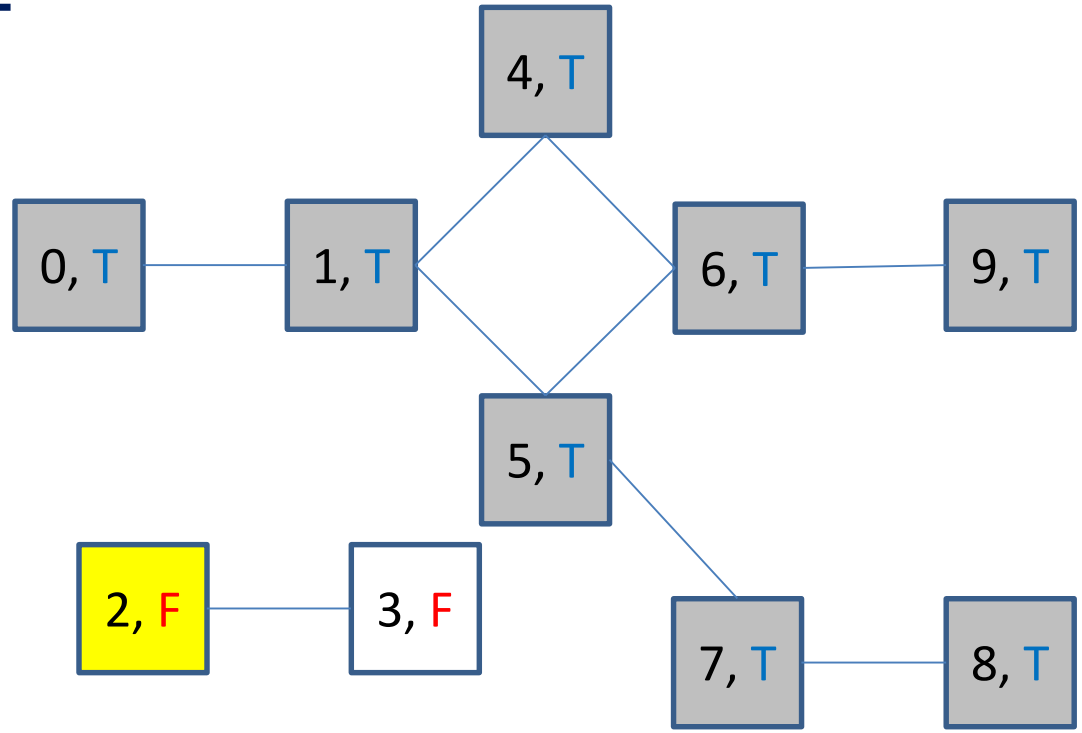
```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```




Ans: 8 7 5 9 6 4 1 0

# 2. Implement DFT Post-order

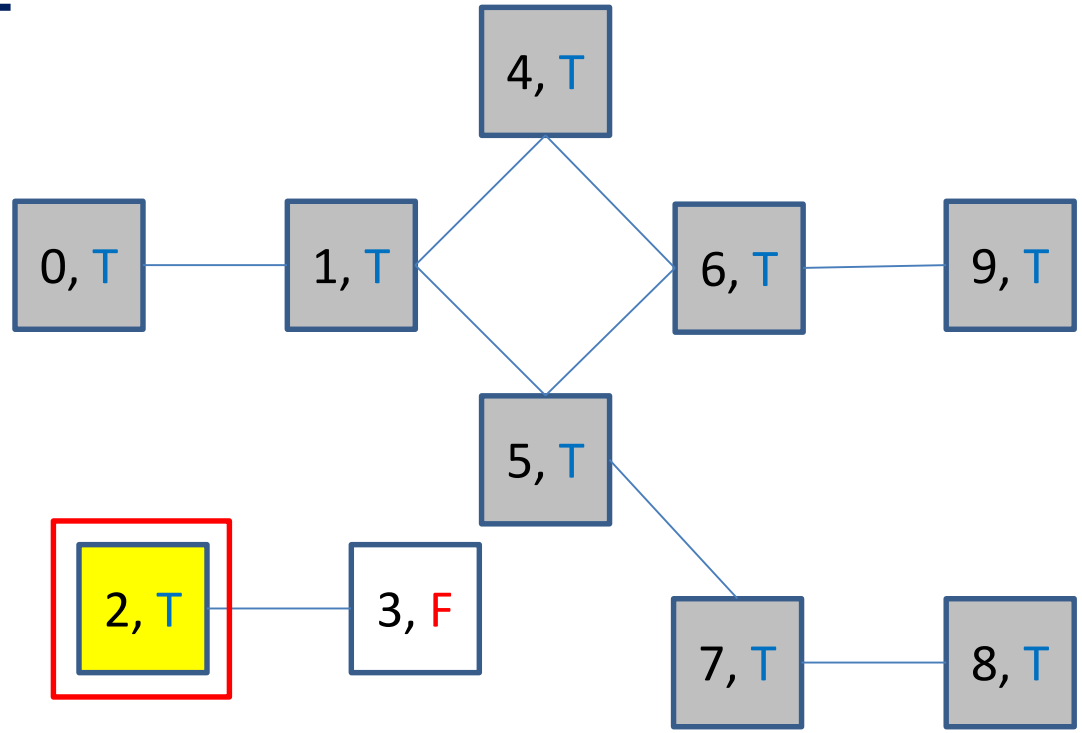
```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```




Ans: 8 7 5 9 6 4 1 0

# 2. Implement DFT Post-order

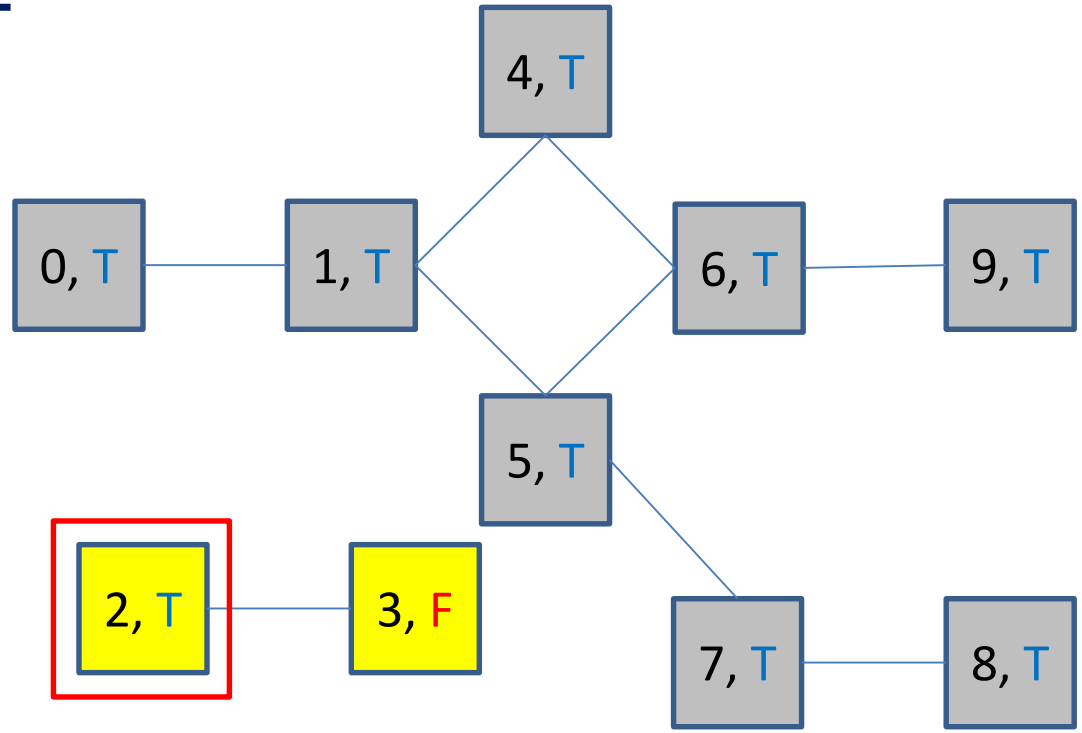
```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```




Ans: 8 7 5 9 6 4 1 0

# 2. Implement DFT Post-order

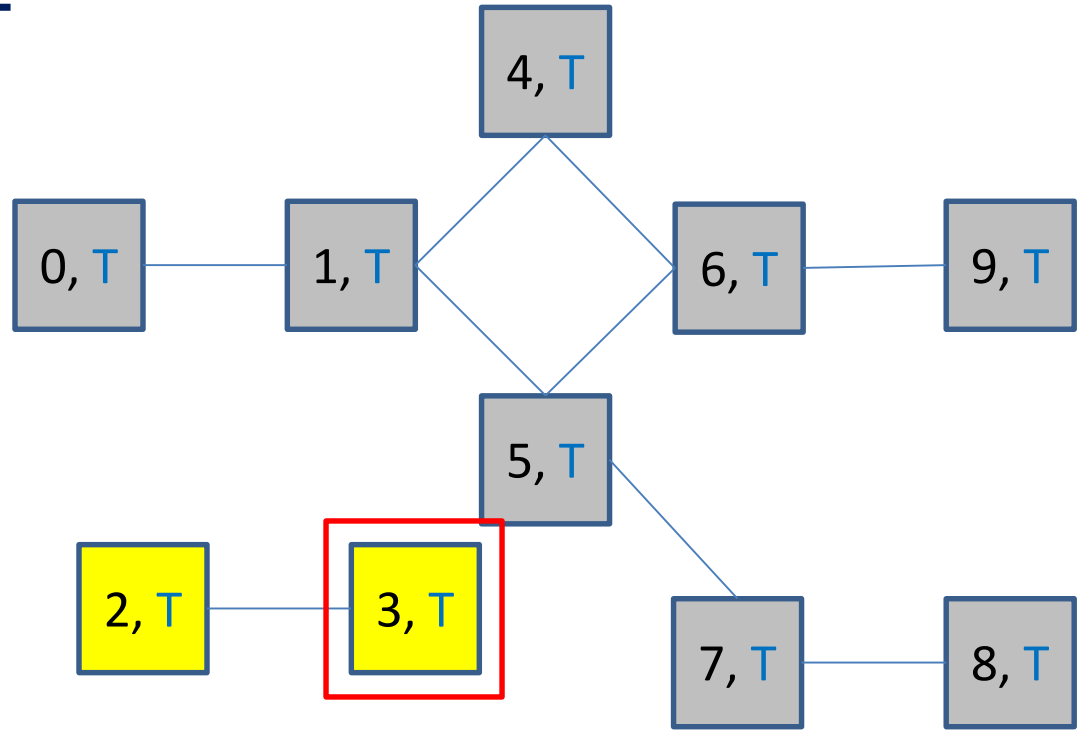
```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```




Ans: 8 7 5 9 6 4 1 0

# 2. Implement DFT Post-order

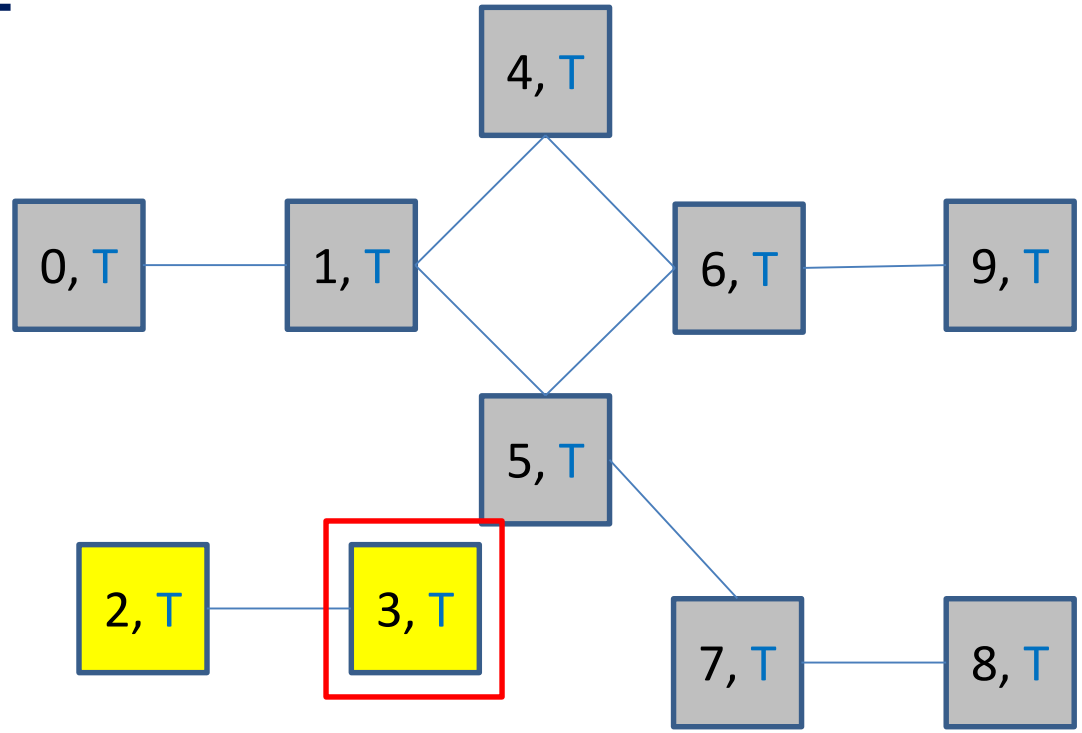
```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```




Ans: 8 7 5 9 6 4 1 0

# 2. Implement DFT Post-order

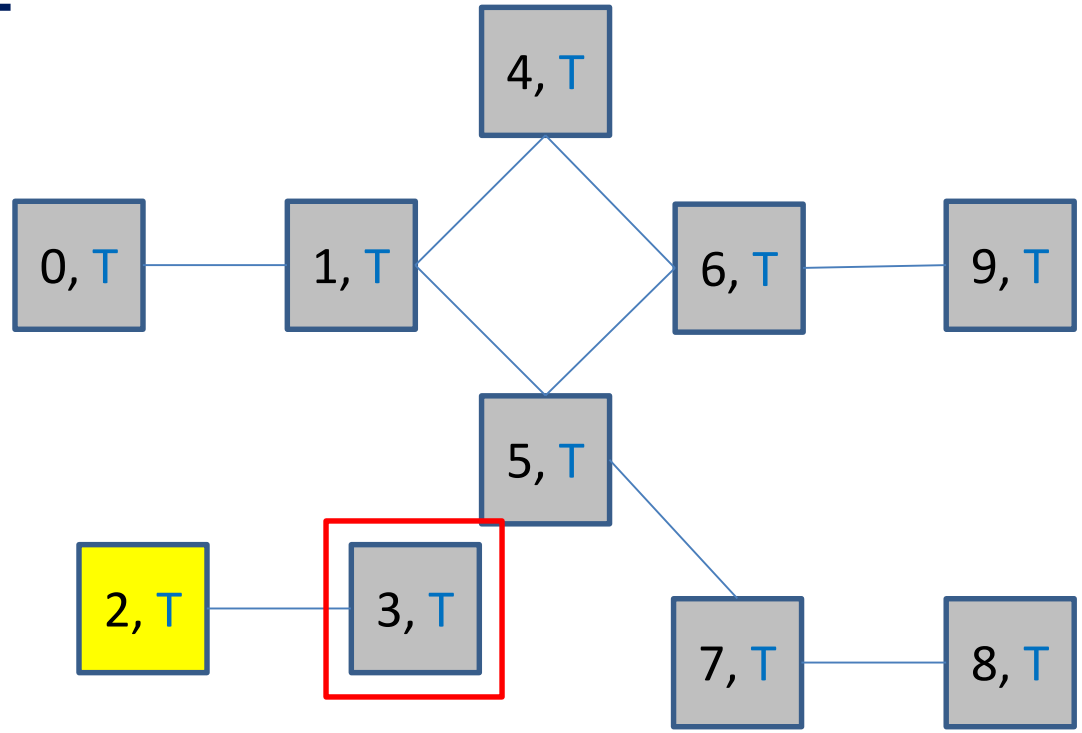
```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```




Ans: 8 7 5 9 6 4 1 0

# 2. Implement DFT Post-order

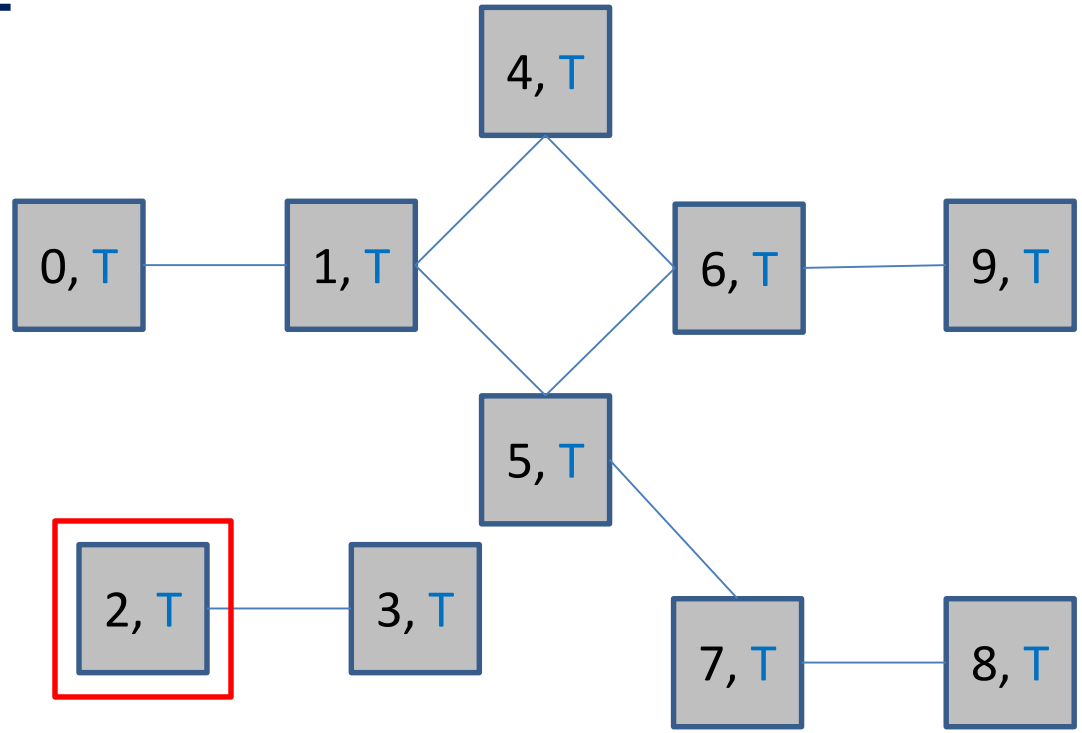
```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```




Ans: 8 7 5 9 6 4 1 0 3

# 2. Implement DFT Post-order

```
def __DFTHelp(self, visited, v):
    if not visited[v]:
        visited[v] = True
        for w in self.neighbors[v]:
            self.__DFTHelp(visited, w)
        print(v, end = ' ')

def DFT(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False
        for v in self.V:
            self.__DFTHelp(visited, v)
```



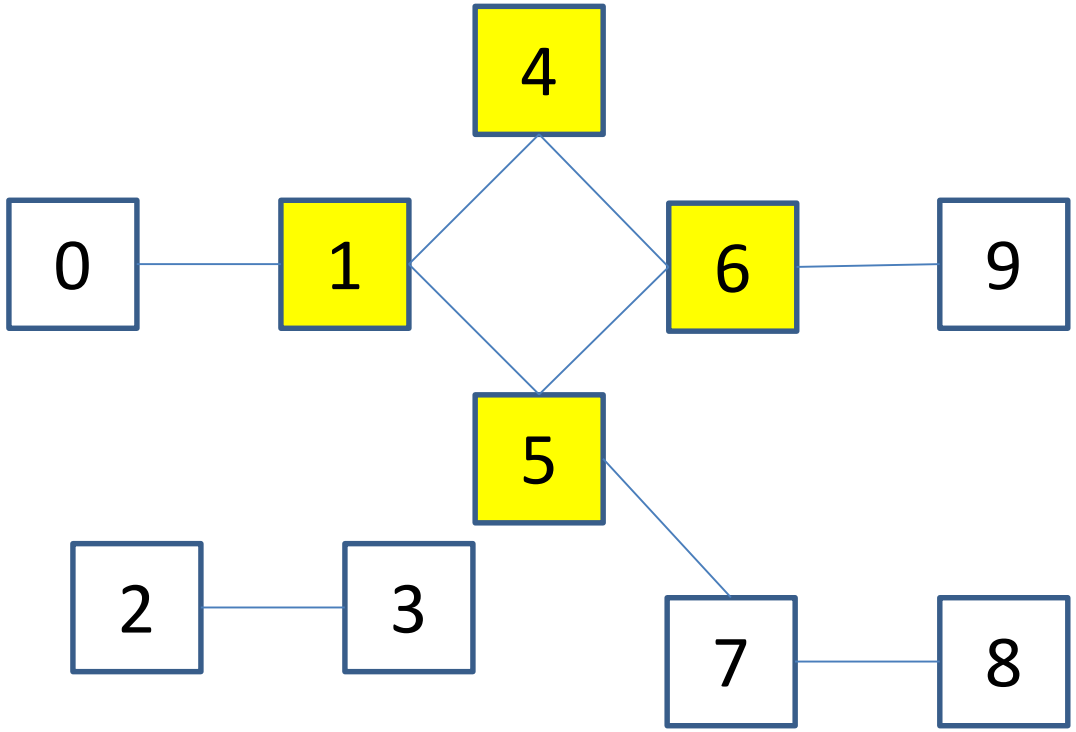

Ans: 8 7 5 9 6 4 1 0 3 2

# 3. Cycle Detection

**Implement a method to detect if cycles exists in an undirected graph**

- Given a node, use recursion to see if a cycle exists (hint: DFT!)
- If at least one cycle is found, immediately return `True`

`True` (There is a cycle 1 – 4 – 6 – 5 – 1)



# 3. Cycle Detection

```
def has_cycles(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False

        for vertex in self.V:
            if not visited[vertex]:
                if self.has_cycle_help(vertex, visited, -1):
                    return True
    return False

def has_cycle_help(self, v, visited, parent):
    visited[v] = True

    for neighbor in self.neighbors[v]:
        if not visited[neighbor]:
            if self.has_cycle_help(neighbor, visited, v):
                return True
        elif neighbor != parent:
            return True

    return False
```

# 3. Cycle Detection

```
def has_cycles(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False

        for vertex in self.V:
            if not visited[vertex]:
                if self.has_cycle_help(vertex, visited, -1):
                    return True
    return False

def has_cycle_help(self, v, visited, parent):
    visited[v] = True

    for neighbor in self.neighbors[v]:
        if not visited[neighbor]:
            if self.has_cycle_help(neighbor, visited, v):
                return True
        elif neighbor != parent:
            return True

    return False
```

Same as in DFT: initialized all vertices as unvisited

# 3. Cycle Detection

```
def has_cycles(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False

        for vertex in self.V:
            if not visited[vertex]:
                if self.has_cycle_help(vertex, visited, -1):
                    return True
    return False

def has_cycle_help(self, v, visited, parent):
    visited[v] = True

    for neighbor in self.neighbors[v]:
        if not visited[neighbor]:
            if self.has_cycle_help(neighbor, visited, v):
                return True
        elif neighbor != parent:
            return True

    return False
```

Iterate through each vertex to check for a cycle, so that disconnected islands are also accounted for.
Call the helper function `has_cycle_help()`: if there exists a cycle, immediately return `True`.

# 3. Cycle Detection

```
def has_cycles(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False

        for vertex in self.V:
            if not visited[vertex]:
                if self.has_cycle_help(vertex, visited, -1):
                    return True
    return False

def has_cycle_help(self, v, visited, parent):
    visited[v] = True

    for neighbor in self.neighbors[v]:
        if not visited[neighbor]:
            if self.has_cycle_help(neighbor, visited, v):
                return True
        elif neighbor != parent:
            return True

    return False
```

Helper function to check if a cycle exists in the connected component with vertex `v`

# 3. Cycle Detection

```
def has_cycles(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False

        for vertex in self.V:
            if not visited[vertex]:
                if self.has_cycle_help(vertex, visited, -1):
                    return True
    return False

def has_cycle_help(self, v, visited, parent):
    visited[v] = True
    for neighbor in self.neighbors[v]:
        if not visited[neighbor]:
            if self.has_cycle_help(neighbor, visited, v):
                return True
        elif neighbor != parent:
            return True

    return False
```

Mark v as visited

# 3. Cycle Detection

```
def has_cycles(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False

        for vertex in self.V:
            if not visited[vertex]:
                if self.has_cycle_help(vertex, visited, -1):
                    return True
    return False

def has_cycle_help(self, v, visited, parent):
    visited[v] = True

    for neighbor in self.neighbors[v]:
        if not visited[neighbor]:
            if self.has_cycle_help(neighbor, visited, v):
                return True
        elif neighbor != parent:
            return True

    return False
```

For each neighbor of current vertex, if not visited, keep searching through its unvisited neighbors, just like DFT!

# 3. Cycle Detection

```
def has_cycles(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False

        for vertex in self.V:
            if not visited[vertex]:
                if self.has_cycle_help(vertex, visited, -1):
                    return True
    return False

def has_cycle_help(self, v, visited, parent):
    visited[v] = True

    for neighbor in self.neighbors[v]:
        if not visited[neighbor]:
            if self.has_cycle_help(neighbor, visited, v):
                return True
        elif neighbor != parent:
            return True

    return False
```

If we `neighbor` is already visited and is not the parent of `v` (since we just traversed to `v` through `parent`, it is obvious for `parent` to be a visited neighbor), then we must have a cycle

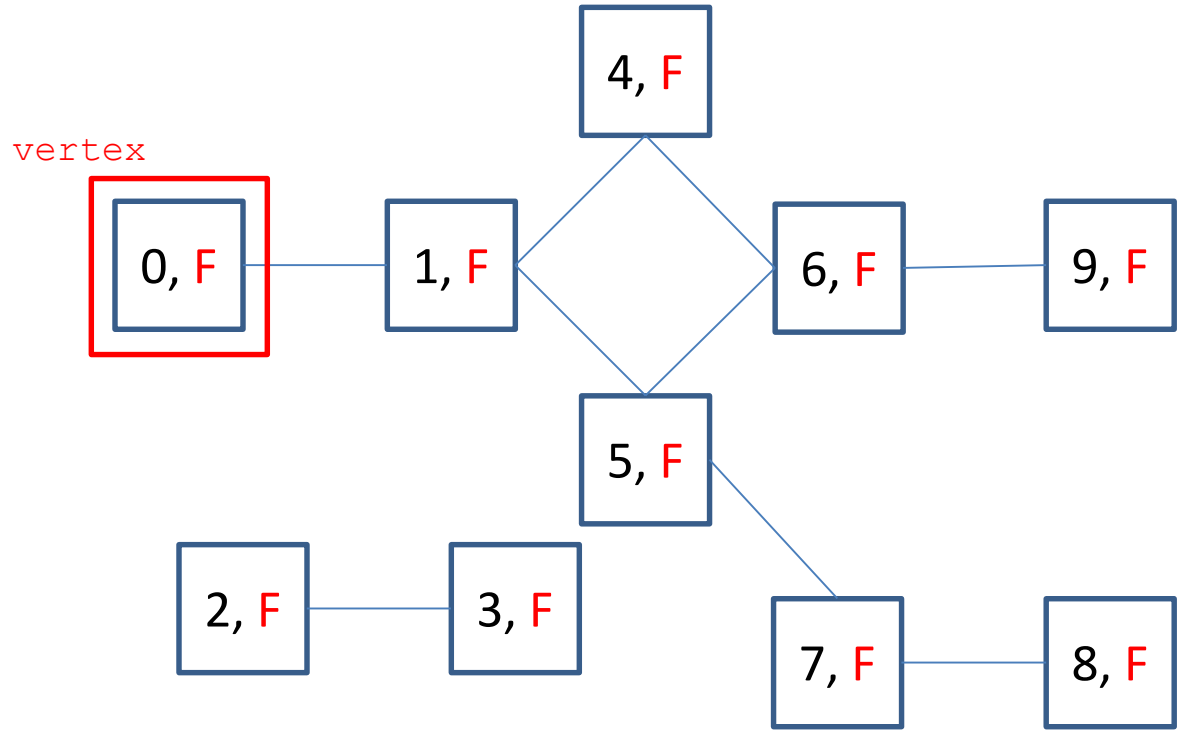
# 3. Cycle Detection

```
def has_cycles(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False

        for vertex in self.V:
            if not visited[vertex]:
                if self.has_cycle_help(vertex, visited, -1):
                    return True
    return False

def has_cycle_help(self, v, visited, parent):
    visited[v] = True

    for neighbor in self.neighbors[v]:
        if not visited[neighbor]:
            if self.has_cycle_help(neighbor, visited, v):
                return True
        elif neighbor != parent:
            return True

    return False
```

If for loop completes without exiting, implies no cycle exists for this connected component containing `v`.

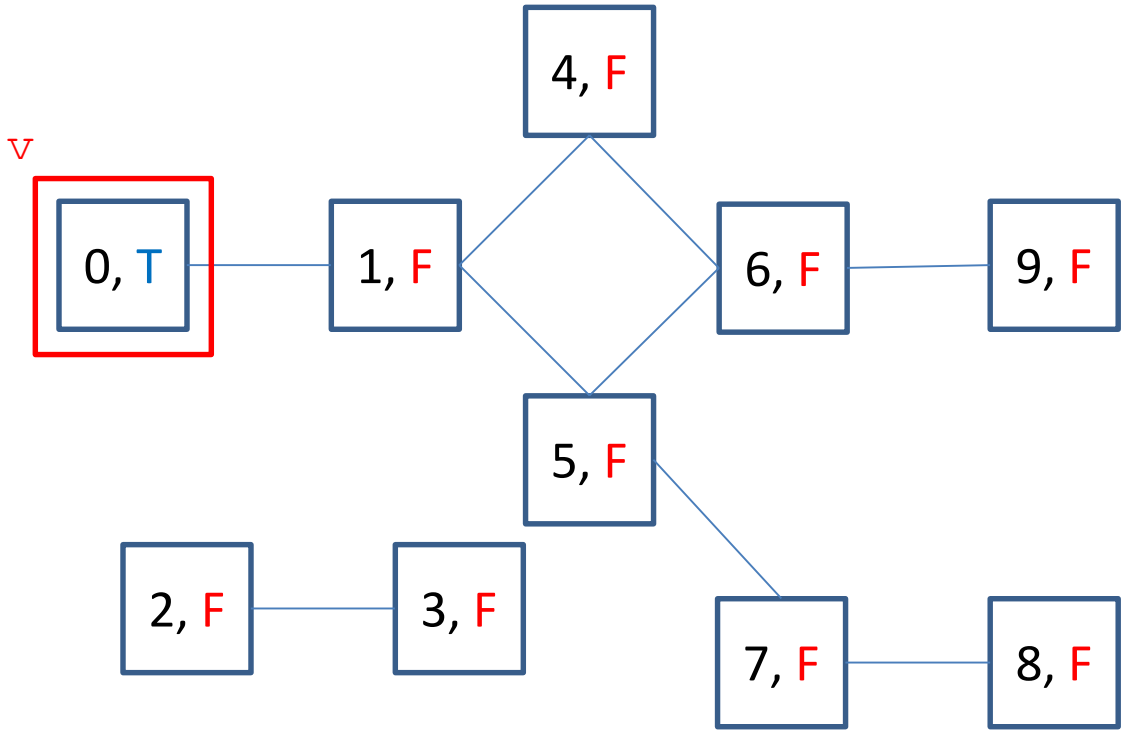
# 3. Cycle Detection

```
def has_cycles(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False

        for vertex in self.V:
            if not visited[vertex]:
                if self.has_cycle_help(vertex, visited, -1):
                    return True
    return False

def has_cycle_help(self, v, visited, parent):
    visited[v] = True

    for neighbor in self.neighbors[v]:
        if not visited[neighbor]:
            if self.has_cycle_help(neighbor, visited, v):
                return True
        elif neighbor != parent:
            return True

    return False
```



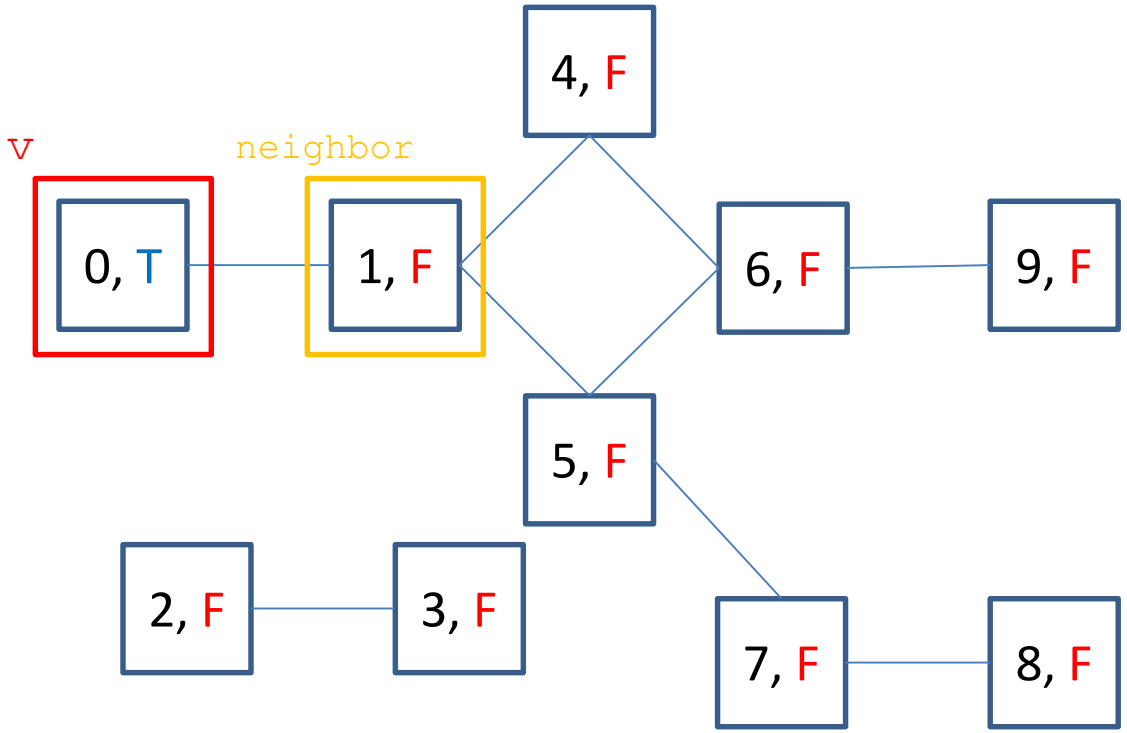
# 3. Cycle Detection

```
def has_cycles(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False

        for vertex in self.V:
            if not visited[vertex]:
                if self.has_cycle_help(vertex, visited, -1):
                    return True
    return False

def has_cycle_help(self, v, visited, parent):
    visited[v] = True

    for neighbor in self.neighbors[v]:
        if not visited[neighbor]:
            if self.has_cycle_help(neighbor, visited, v):
                return True
        elif neighbor != parent:
            return True

    return False
```



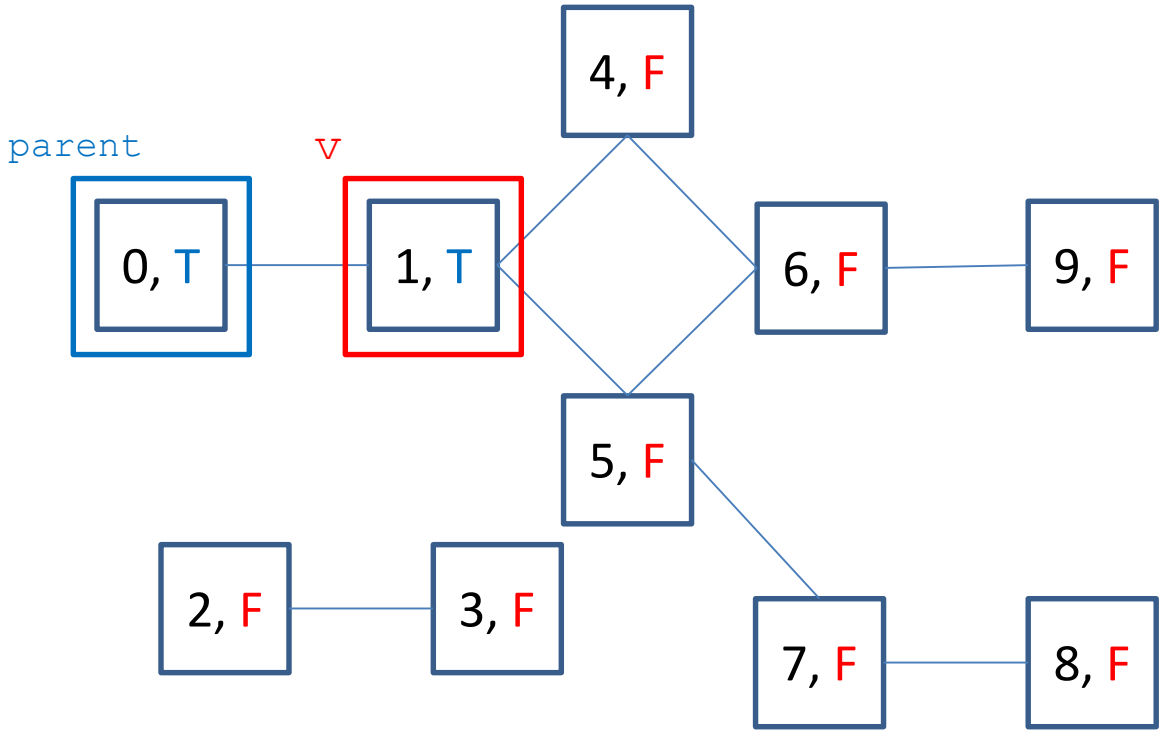
# 3. Cycle Detection

```
def has_cycles(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False

        for vertex in self.V:
            if not visited[vertex]:
                if self.has_cycle_help(vertex, visited, -1):
                    return True
    return False

def has_cycle_help(self, v, visited, parent):
    visited[v] = True

    for neighbor in self.neighbors[v]:
        if not visited[neighbor]:
            if self.has_cycle_help(neighbor, visited, v):
                return True
        elif neighbor != parent:
            return True

    return False
```



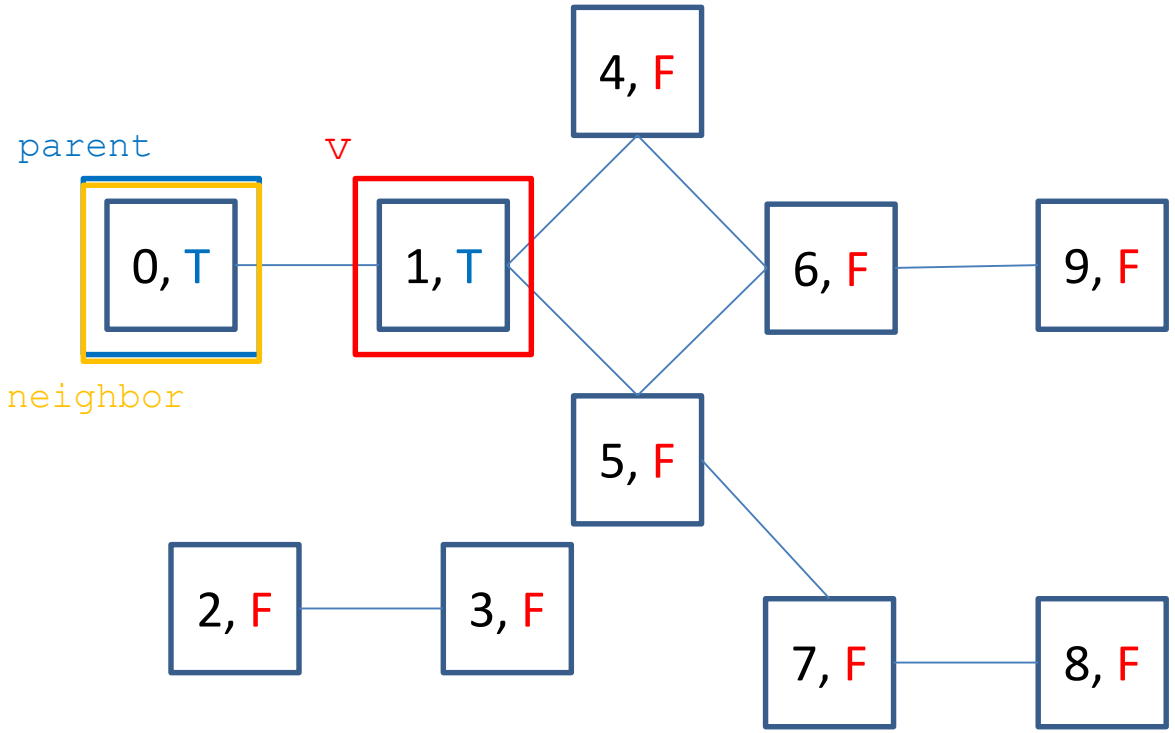
# 3. Cycle Detection

```
def has_cycles(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False

        for vertex in self.V:
            if not visited[vertex]:
                if self.has_cycle_help(vertex, visited, -1):
                    return True
    return False

def has_cycle_help(self, v, visited, parent):
    visited[v] = True

    for neighbor in self.neighbors[v]:
        if not visited[neighbor]:
            if self.has_cycle_help(neighbor, visited, v):
                return True
        elif neighbor != parent:
            return True

    return False
```



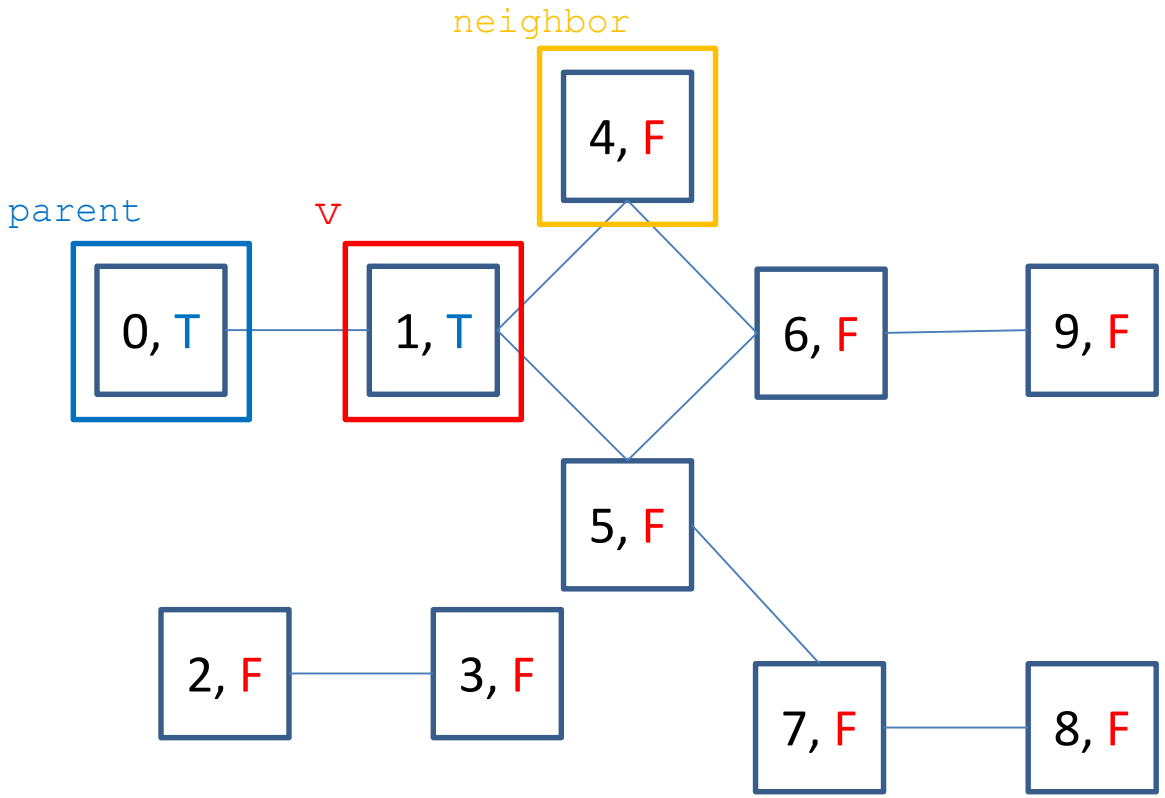
# 3. Cycle Detection

```
def has_cycles(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False

        for vertex in self.V:
            if not visited[vertex]:
                if self.has_cycle_help(vertex, visited, -1):
                    return True
    return False

def has_cycle_help(self, v, visited, parent):
    visited[v] = True

    for neighbor in self.neighbors[v]:
        if not visited[neighbor]:
            if self.has_cycle_help(neighbor, visited, v):
                return True
        elif neighbor != parent:
            return True

    return False
```



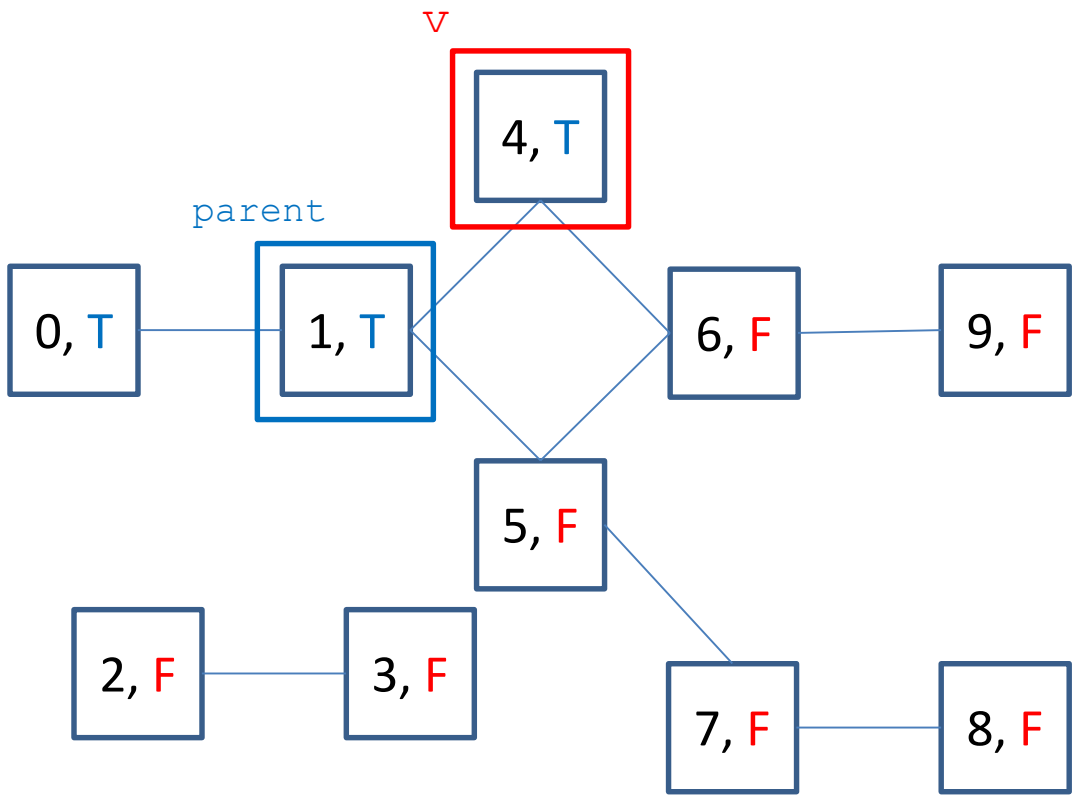
# 3. Cycle Detection

```
def has_cycles(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False

        for vertex in self.V:
            if not visited[vertex]:
                if self.has_cycle_help(vertex, visited, -1):
                    return True
    return False

def has_cycle_help(self, v, visited, parent):
    visited[v] = True

    for neighbor in self.neighbors[v]:
        if not visited[neighbor]:
            if self.has_cycle_help(neighbor, visited, v):
                return True
        elif neighbor != parent:
            return True

    return False
```



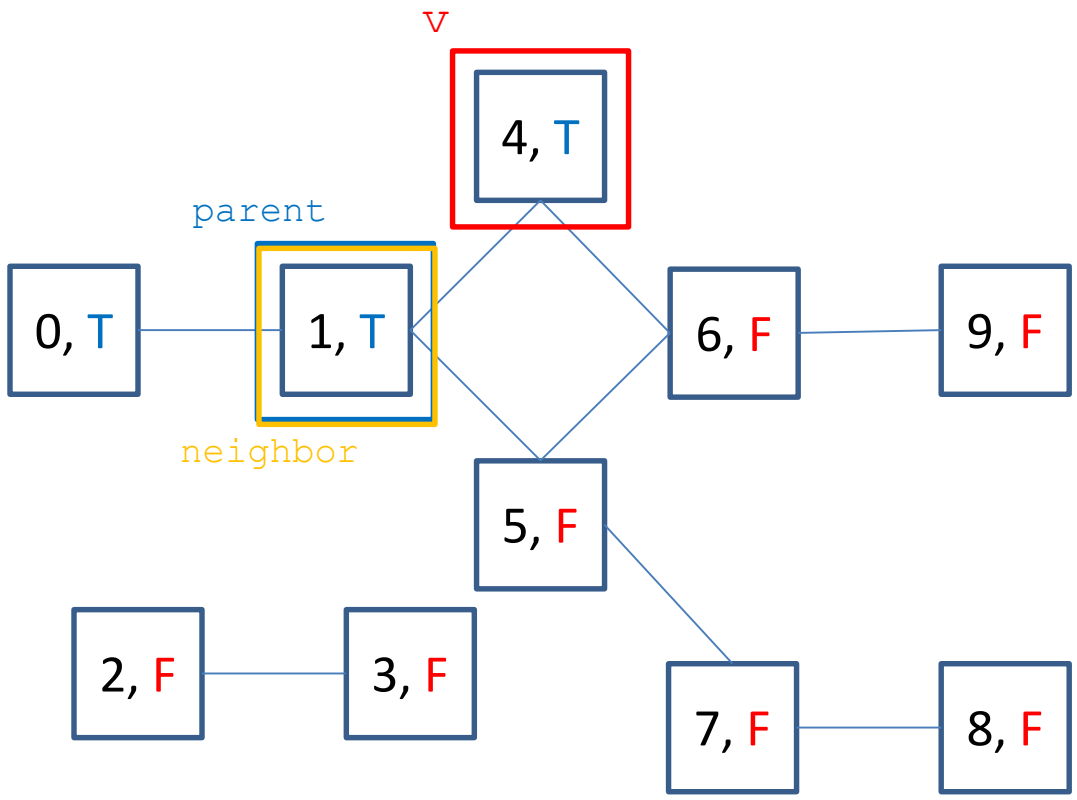
# 3. Cycle Detection

```
def has_cycles(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False

        for vertex in self.V:
            if not visited[vertex]:
                if self.has_cycle_help(vertex, visited, -1):
                    return True
    return False

def has_cycle_help(self, v, visited, parent):
    visited[v] = True

    for neighbor in self.neighbors[v]:
        if not visited[neighbor]:
            if self.has_cycle_help(neighbor, visited, v):
                return True
        elif neighbor != parent:
            return True

    return False
```



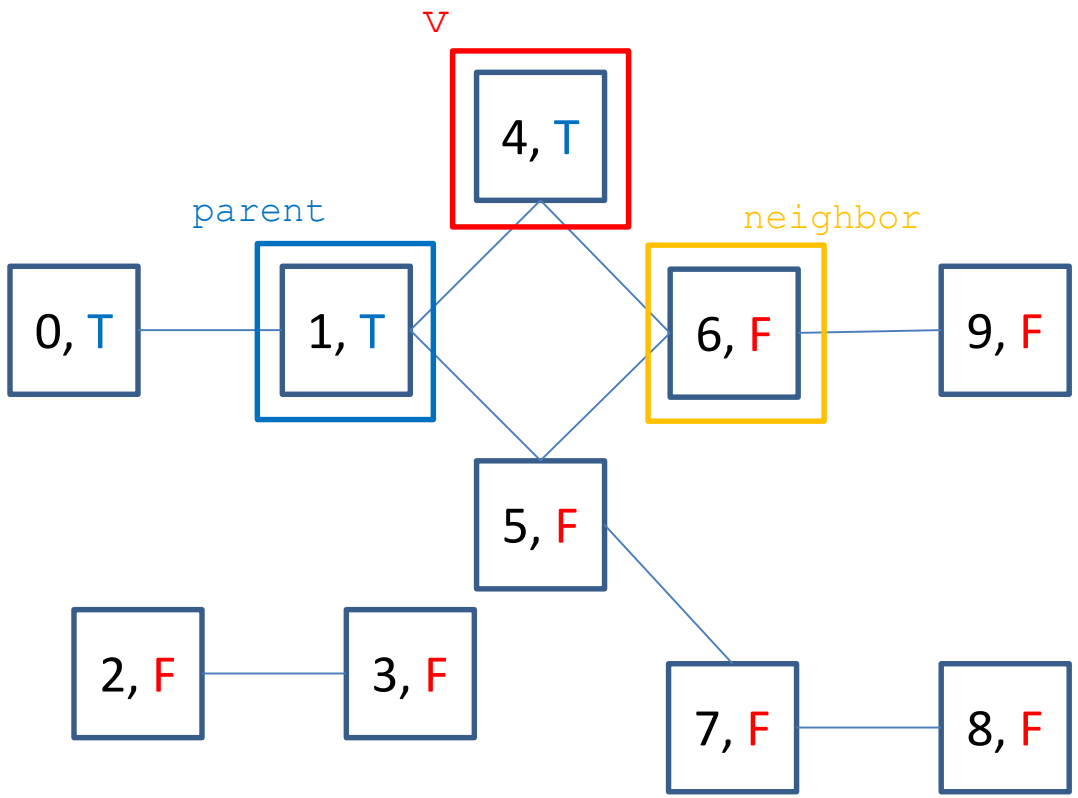
# 3. Cycle Detection

```
def has_cycles(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False

        for vertex in self.V:
            if not visited[vertex]:
                if self.has_cycle_help(vertex, visited, -1):
                    return True
    return False

def has_cycle_help(self, v, visited, parent):
    visited[v] = True

    for neighbor in self.neighbors[v]:
        if not visited[neighbor]:
            if self.has_cycle_help(neighbor, visited, v):
                return True
        elif neighbor != parent:
            return True

    return False
```



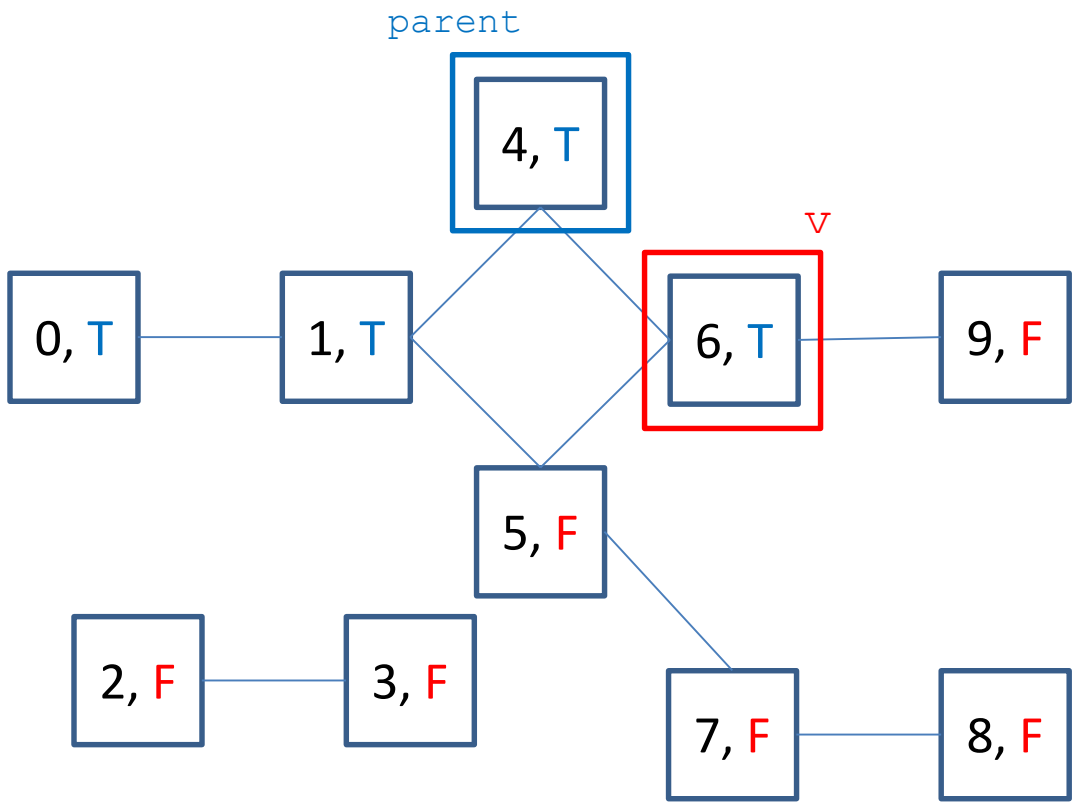
# 3. Cycle Detection

```
def has_cycles(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False

        for vertex in self.V:
            if not visited[vertex]:
                if self.has_cycle_help(vertex, visited, -1):
                    return True
    return False

def has_cycle_help(self, v, visited, parent):
    visited[v] = True

    for neighbor in self.neighbors[v]:
        if not visited[neighbor]:
            if self.has_cycle_help(neighbor, visited, v):
                return True
        elif neighbor != parent:
            return True

    return False
```



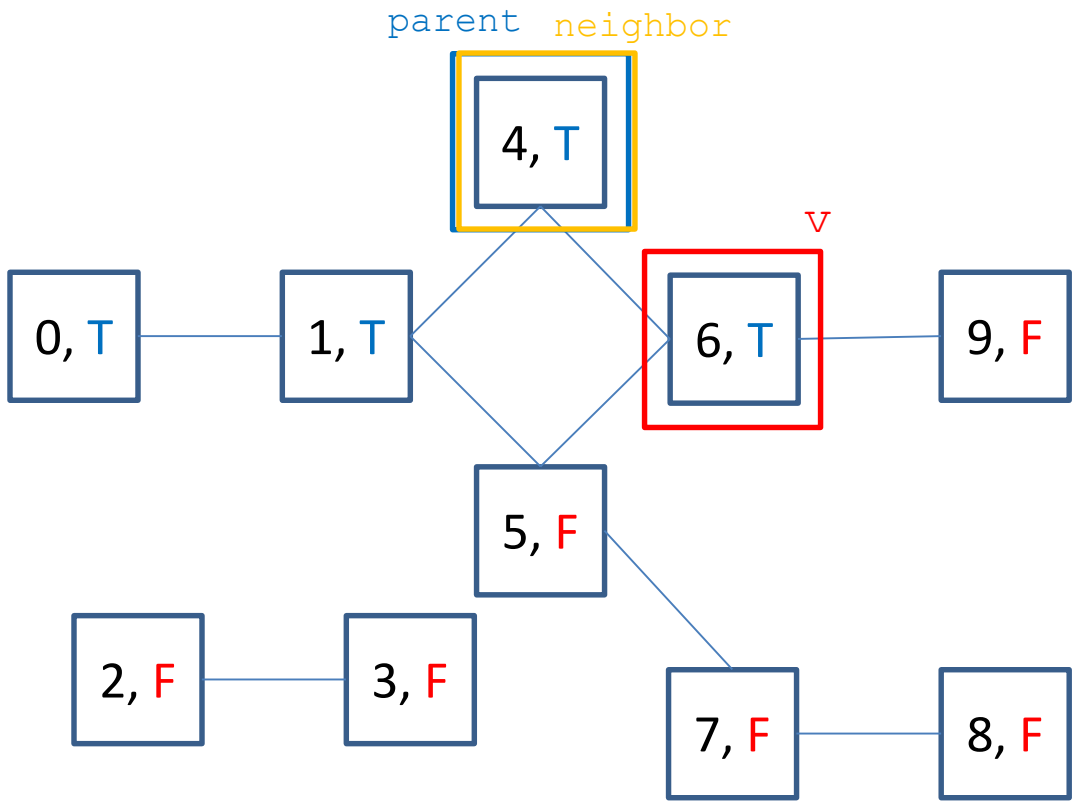
# 3. Cycle Detection

```
def has_cycles(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False

        for vertex in self.V:
            if not visited[vertex]:
                if self.has_cycle_help(vertex, visited, -1):
                    return True
    return False

def has_cycle_help(self, v, visited, parent):
    visited[v] = True

    for neighbor in self.neighbors[v]:
        if not visited[neighbor]:
            if self.has_cycle_help(neighbor, visited, v):
                return True
        elif neighbor != parent:
            return True

    return False
```



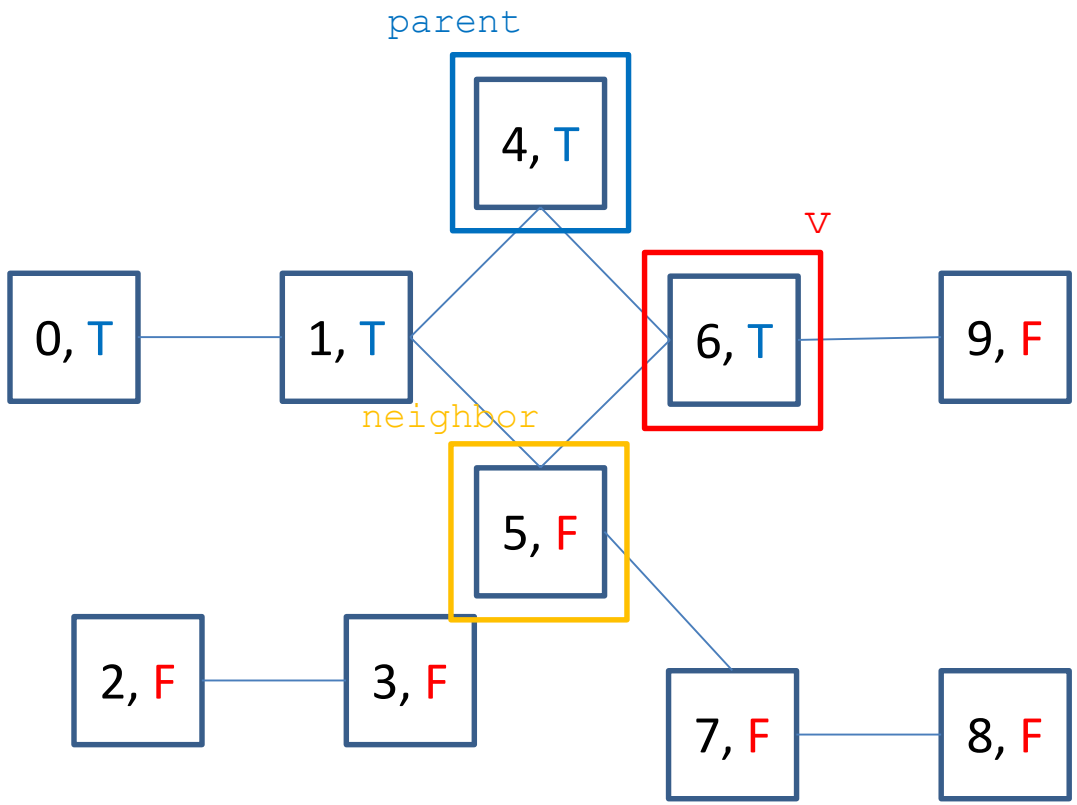
# 3. Cycle Detection

```
def has_cycles(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False

        for vertex in self.V:
            if not visited[vertex]:
                if self.has_cycle_help(vertex, visited, -1):
                    return True
    return False

def has_cycle_help(self, v, visited, parent):
    visited[v] = True

    for neighbor in self.neighbors[v]:
        if not visited[neighbor]:
            if self.has_cycle_help(neighbor, visited, v):
                return True
        elif neighbor != parent:
            return True

    return False
```



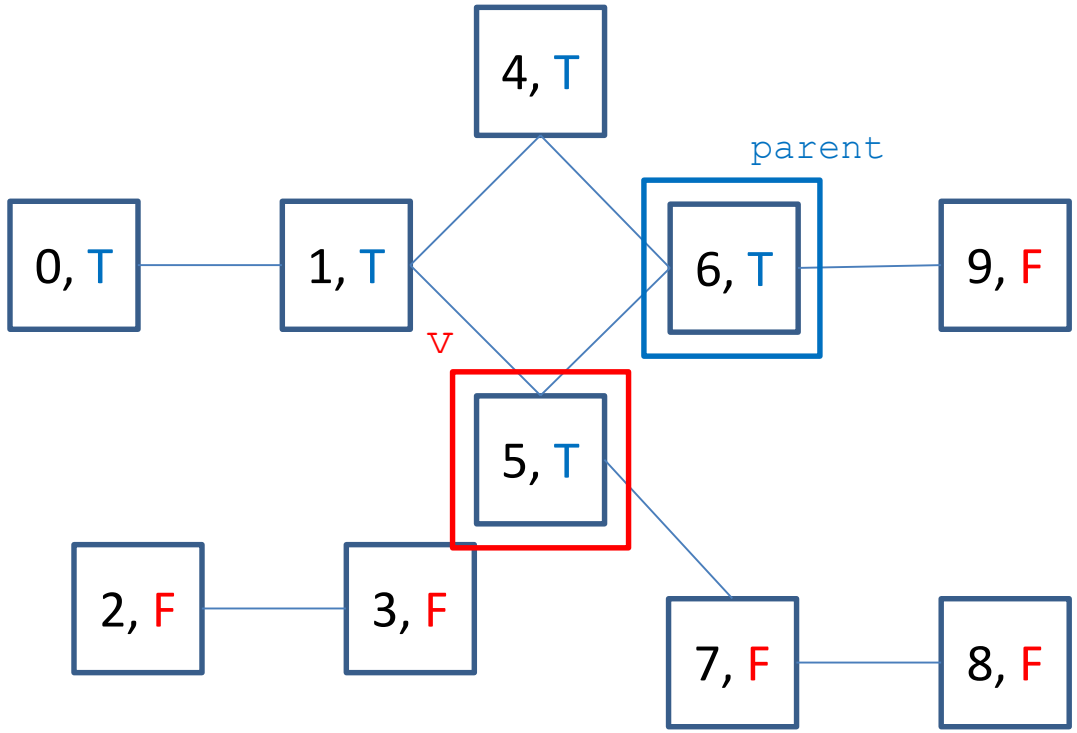
# 3. Cycle Detection

```
def has_cycles(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False

        for vertex in self.V:
            if not visited[vertex]:
                if self.has_cycle_help(vertex, visited, -1):
                    return True
    return False

def has_cycle_help(self, v, visited, parent):
    visited[v] = True

    for neighbor in self.neighbors[v]:
        if not visited[neighbor]:
            if self.has_cycle_help(neighbor, visited, v):
                return True
        elif neighbor != parent:
            return True

    return False
```



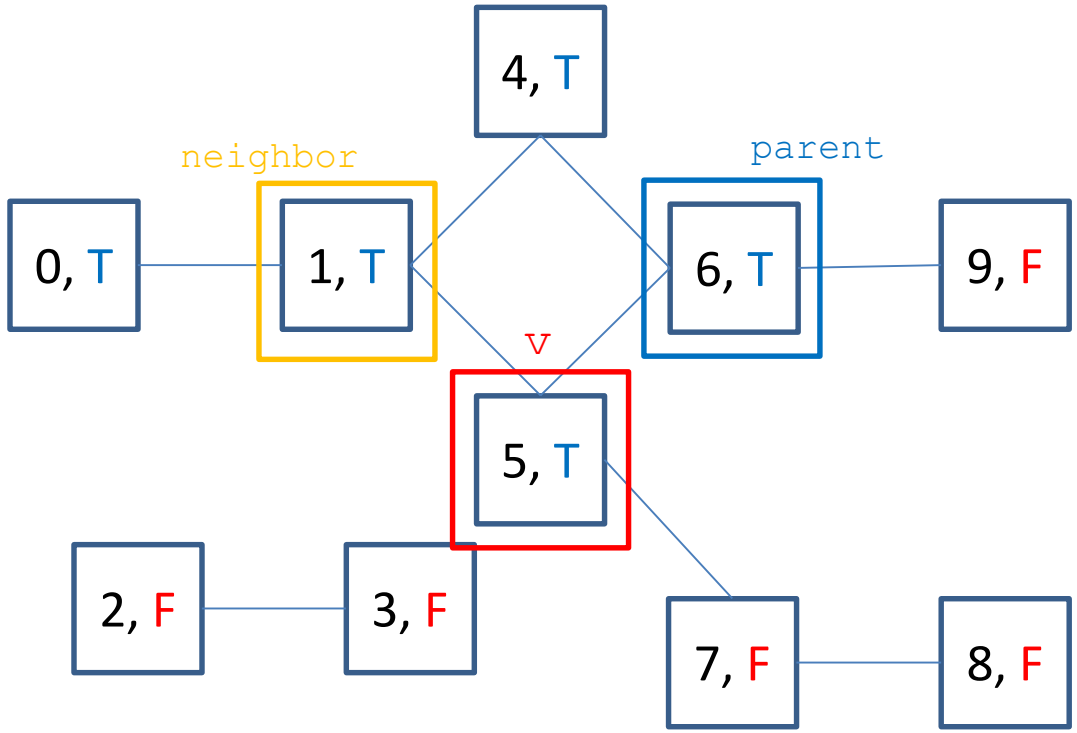
# 3. Cycle Detection

```
def has_cycles(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False

        for vertex in self.V:
            if not visited[vertex]:
                if self.has_cycle_help(vertex, visited, -1):
                    return True
    return False

def has_cycle_help(self, v, visited, parent):
    visited[v] = True

    for neighbor in self.neighbors[v]:
        if not visited[neighbor]:
            if self.has_cycle_help(neighbor, visited, v):
                return True
        elif neighbor != parent:
            return True

    return False
```



# 3. Cycle Detection

```
def has_cycles(self):
    if self.V:
        visited = {}
        for v in self.V:
            visited[v] = False

        for vertex in self.V:
            if not visited[vertex]:
                if self.has_cycle_help(vertex, visited, -1):
                    return True
    return False

def has_cycle_help(self, v, visited, parent):
    visited[v] = True

    for neighbor in self.neighbors[v]:
        if not visited[neighbor]:
            if self.has_cycle_help(neighbor, visited, v):
                return True
        elif neighbor != parent:
            return True

    return False
```



# Thanks