CSCI 232:

Data Structures and Algorithms

Minimum Spanning Tree (MST) Part 1

Reese Pearsall

Spring 2025

Announcements

Lab 8 due on Friday

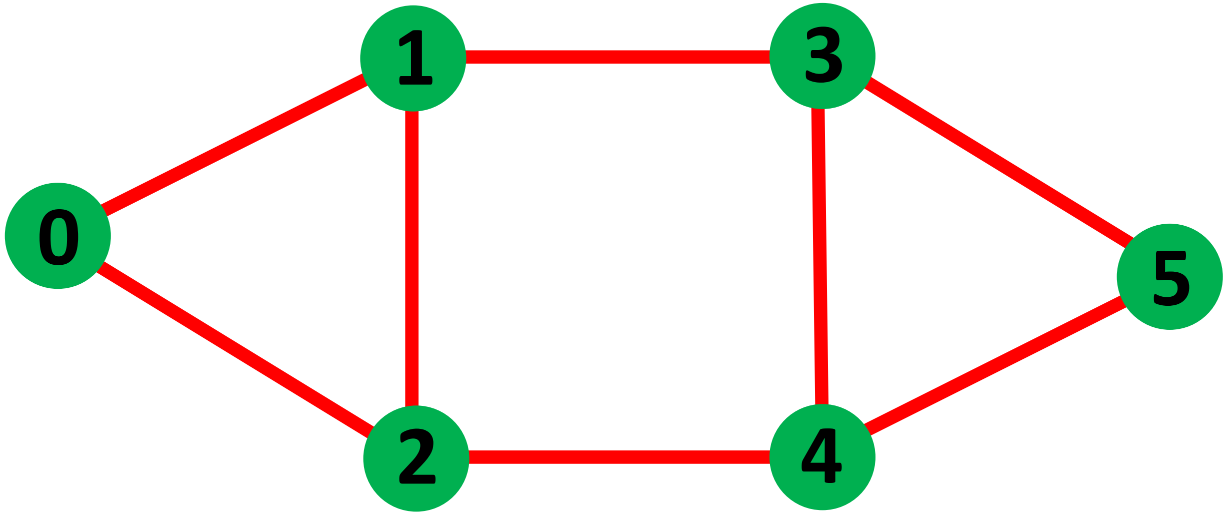
Quiz next week Friday

Program 3 will hopefully be posted soon

Very relevant meme for this week's lab

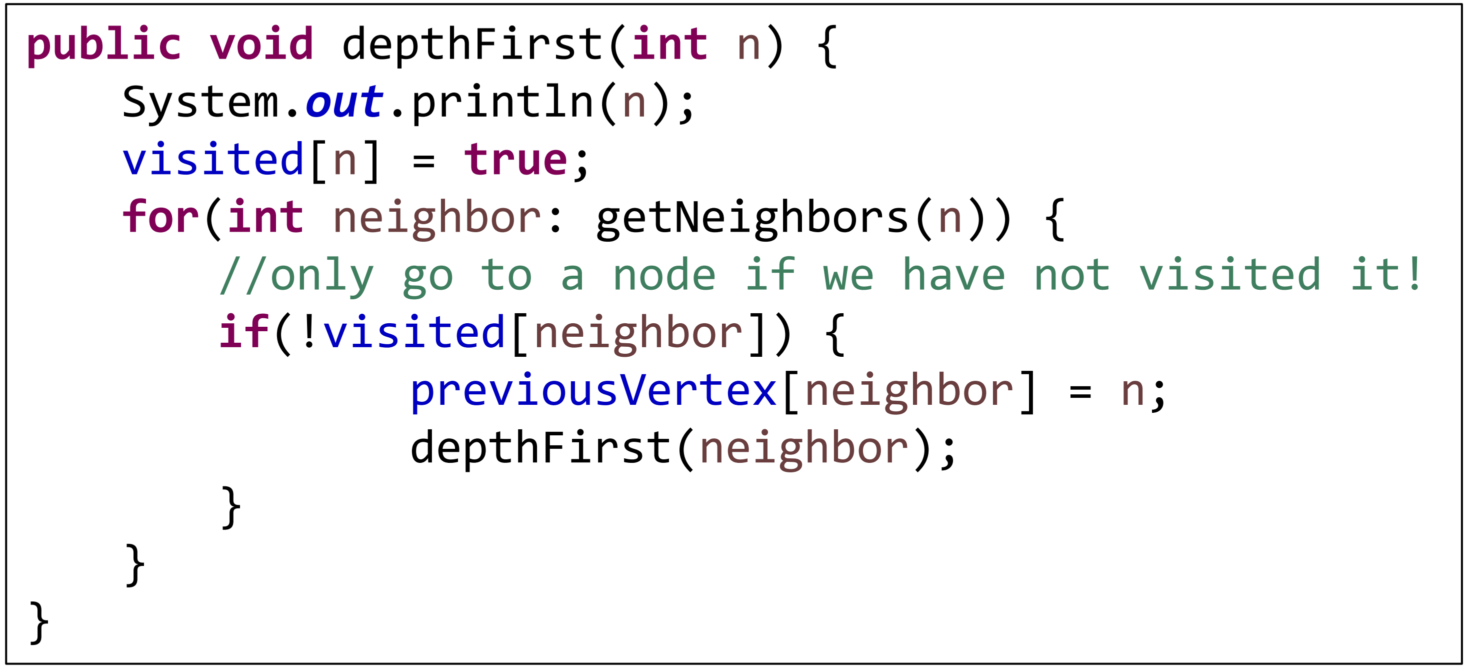
Graphs

G = (V, E)

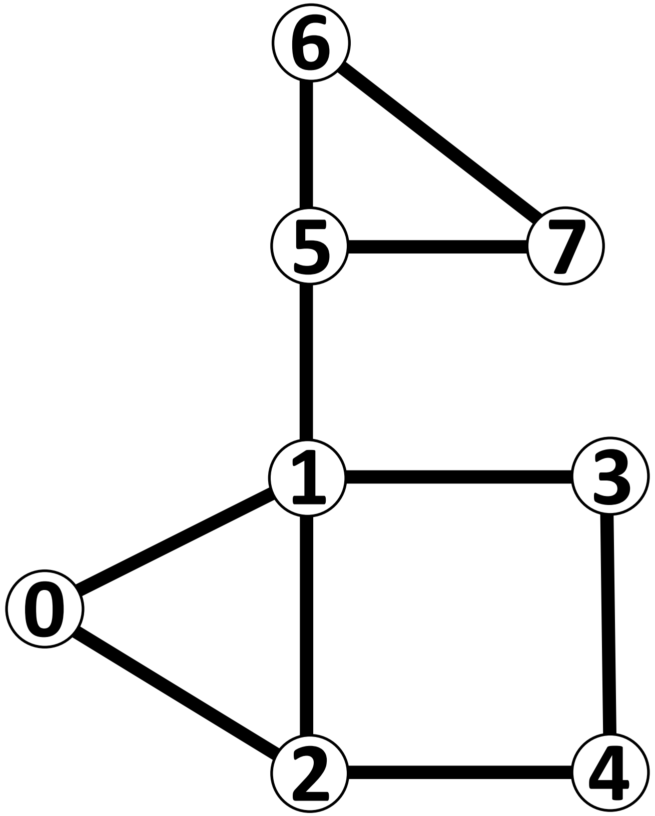


Adjacency List

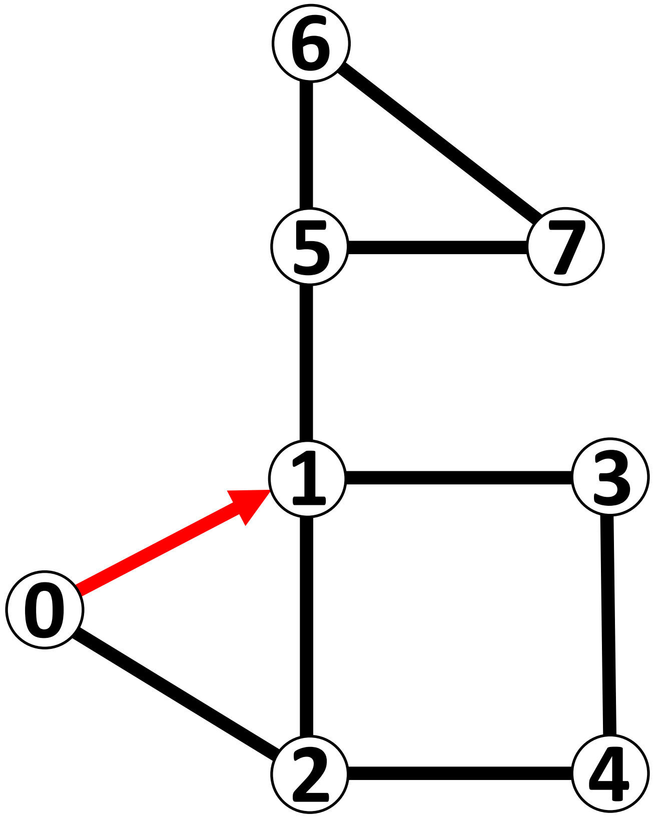
0	→	{1,2}
1	→	{0,2,3}
2	→	{0,1,4}
3	→	{1,4,5}
4	→	{2,3,5}
5	→	{3,4}



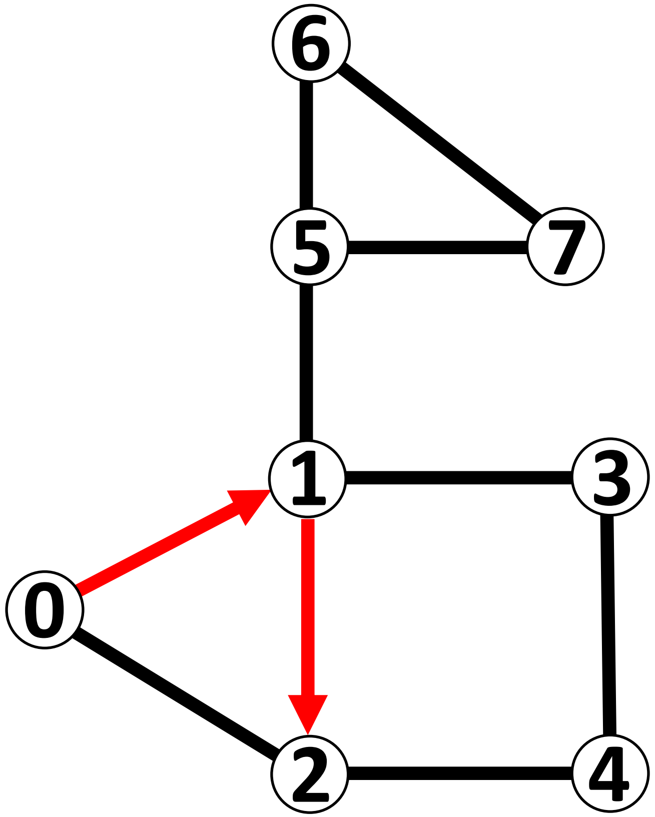
```
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for(int neighbor: getNeighbors(n)) {
        //only go to a node if we have not visited it!
        if(!visited[neighbor]) {
                previousVertex[neighbor] = n;
                depthFirst(neighbor);
        }
    }
}
```



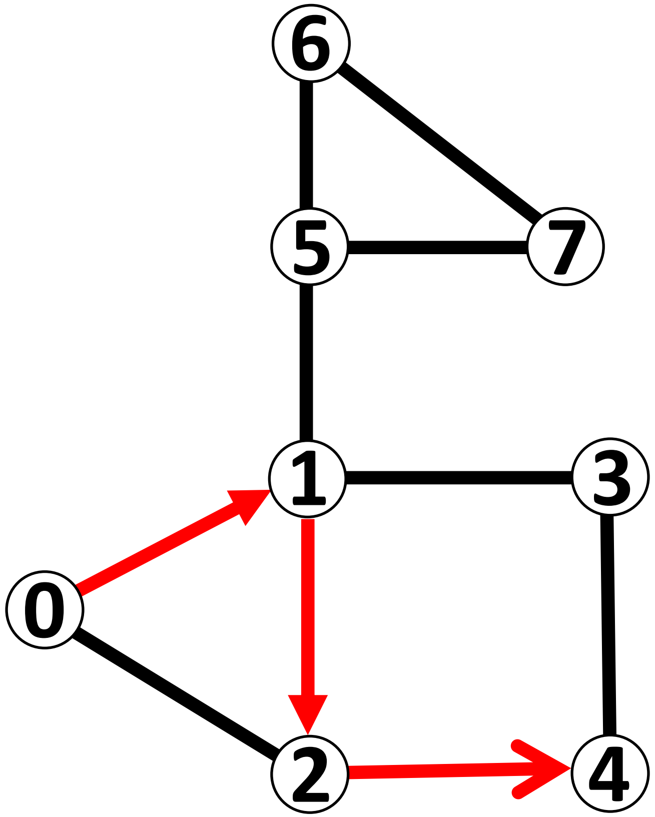
```java
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for(int neighbor: getNeighbors(n)) {
        //only go to a node if we have not visited it!
        if(!visited[neighbor]) {
                previousVertex[neighbor] = n;
                depthFirst(neighbor);
        }
    }
}
```



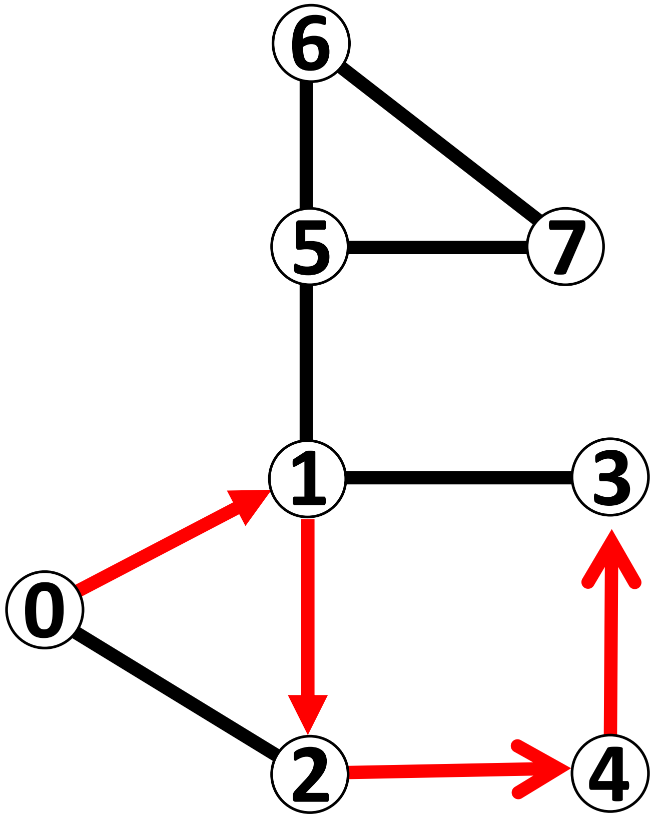
```java
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for(int neighbor: getNeighbors(n)) {
        //only go to a node if we have not visited it!
        if(!visited[neighbor]) {
                previousVertex[neighbor] = n;
                depthFirst(neighbor);
        }
    }
}
```



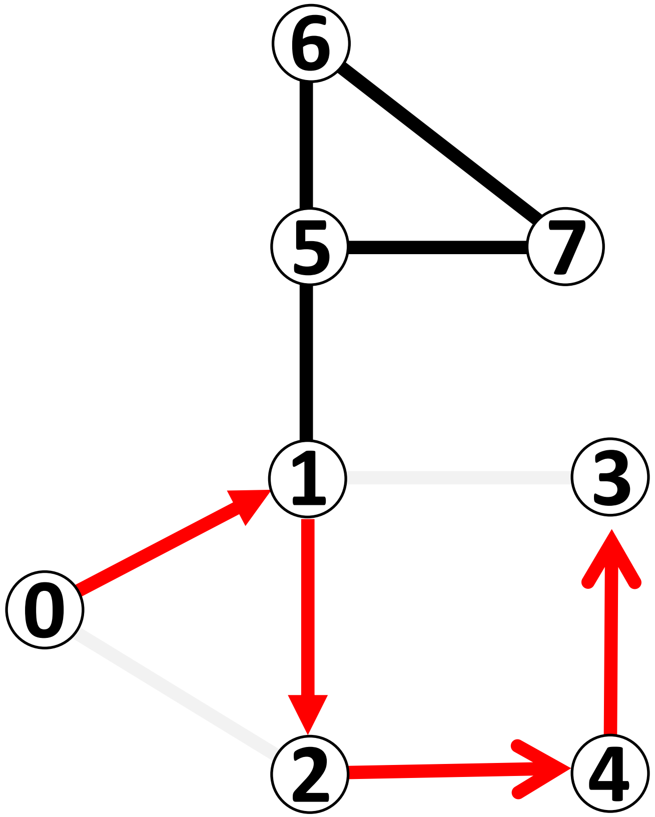
```java
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for(int neighbor: getNeighbors(n)) {
        //only go to a node if we have not visited it!
        if(!visited[neighbor]) {
                previousVertex[neighbor] = n;
                depthFirst(neighbor);
        }
    }
}
```



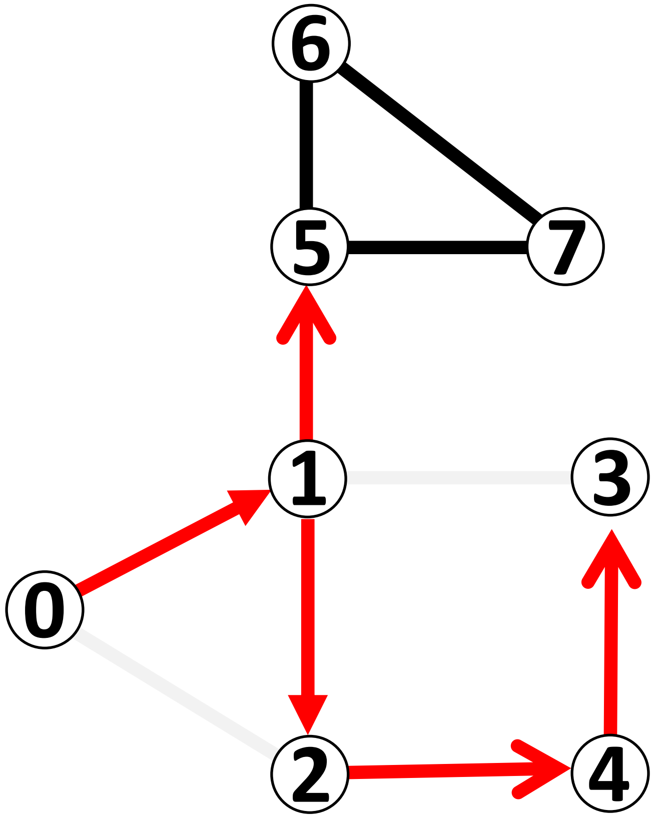
```
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for(int neighbor: getNeighbors(n)) {
        //only go to a node if we have not visited it!
        if(!visited[neighbor]) {
                previousVertex[neighbor] = n;
                depthFirst(neighbor);
        }
    }
}
```



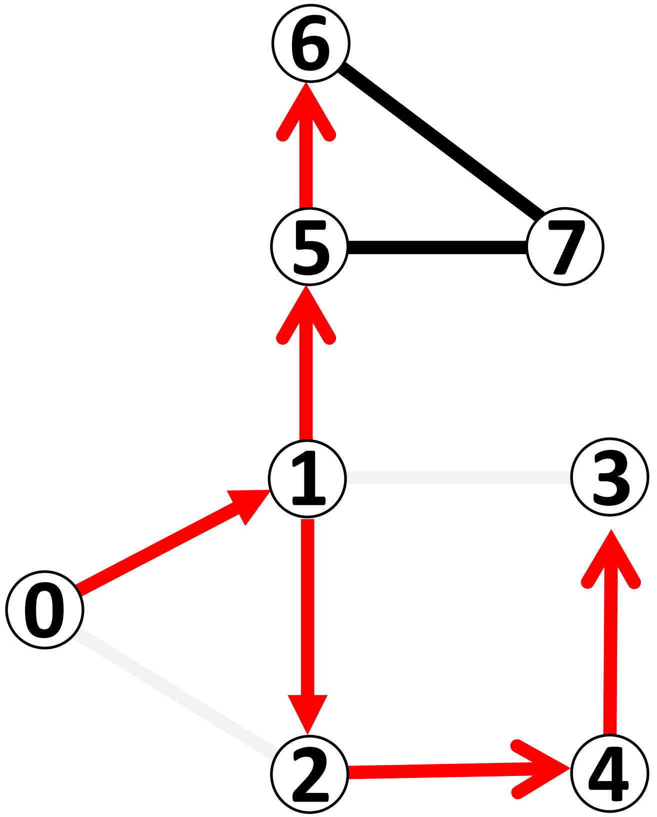
```java
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for(int neighbor: getNeighbors(n)) {
        //only go to a node if we have not visited it!
        if(!visited[neighbor]) {
                previousVertex[neighbor] = n;
                depthFirst(neighbor);
        }
    }
}
```



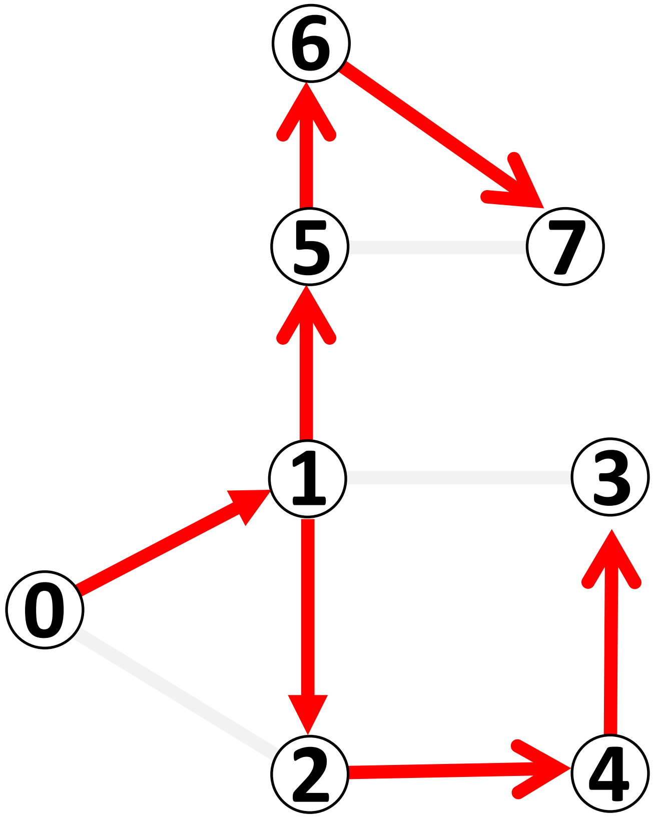
```java
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for(int neighbor: getNeighbors(n)) {
        //only go to a node if we have not visited it!
        if(!visited[neighbor]) {
                previousVertex[neighbor] = n;
                depthFirst(neighbor);
        }
    }
}
```



```java
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for(int neighbor: getNeighbors(n)) {
        //only go to a node if we have not visited it!
        if(!visited[neighbor]) {
                previousVertex[neighbor] = n;
                depthFirst(neighbor);
        }
    }
}
```



```java
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for(int neighbor: getNeighbors(n)) {
        //only go to a node if we have not visited it!
        if(!visited[neighbor]) {
                previousVertex[neighbor] = n;
                depthFirst(neighbor);
        }
    }
}
```

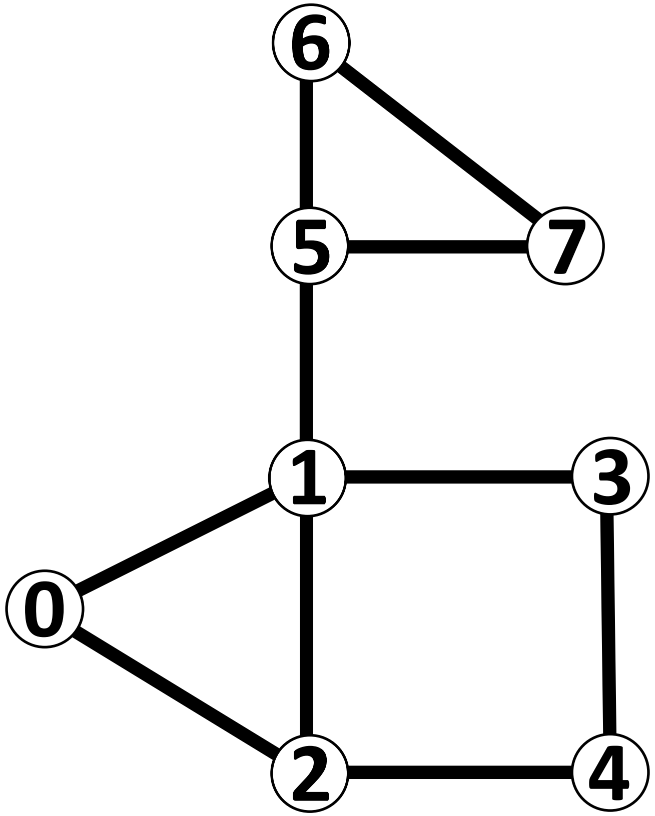


Depth First Order*:

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

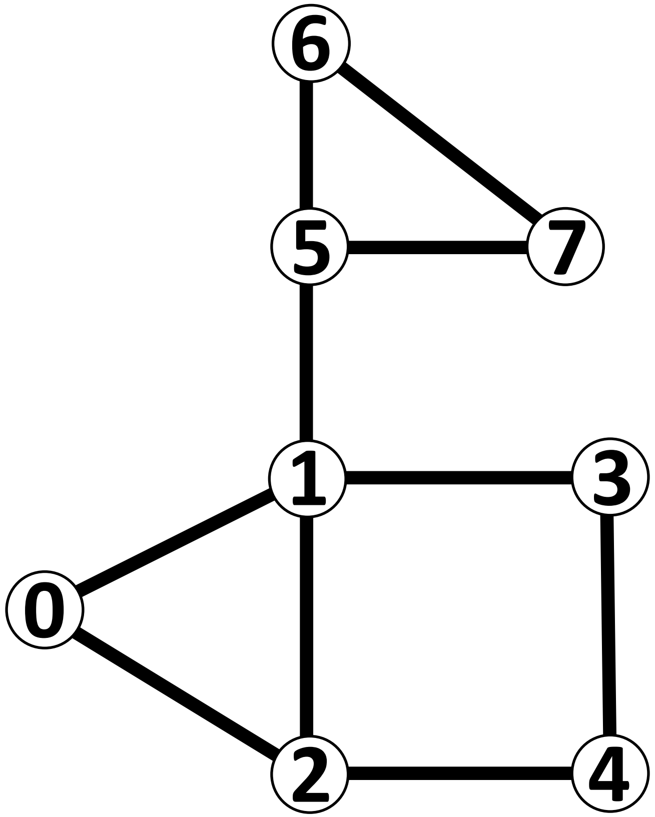
*not the only DFS order

Breadth First?



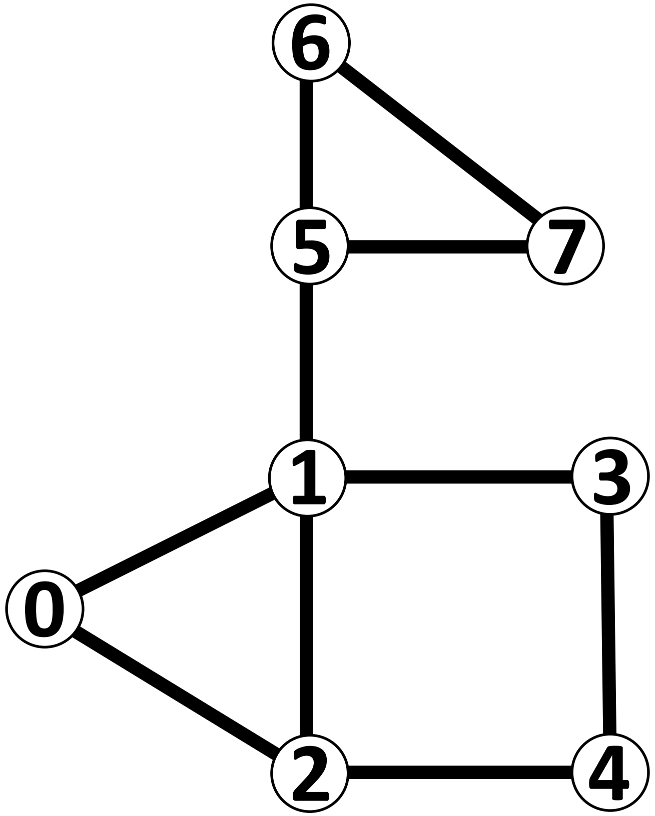
```java
public void breadthFirst(int n) {
    Queue<Integer> queue = new LinkedList<>();
    visited[n] = true;
    queue.add(n);









}
```

queue 0



Output: 0,

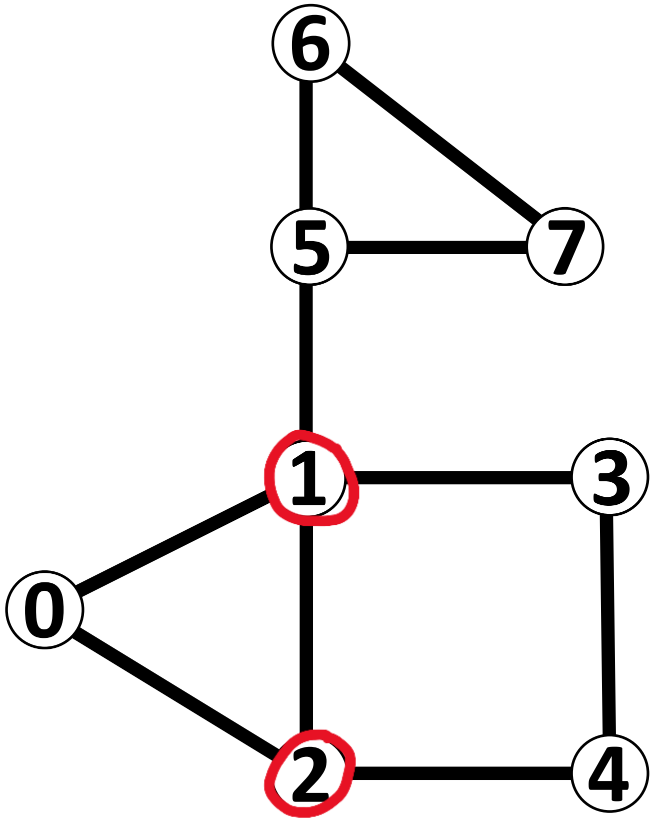
```
public void breadthFirst(int n) {
    Queue<Integer> queue = new LinkedList<>();
    visited[n] = true;
    queue.add(n);
    while(!queue.isEmpty()) {
        int vertex = queue.poll();
        System.out.println(vertex);









    }
}
```



queue

Output: 0,

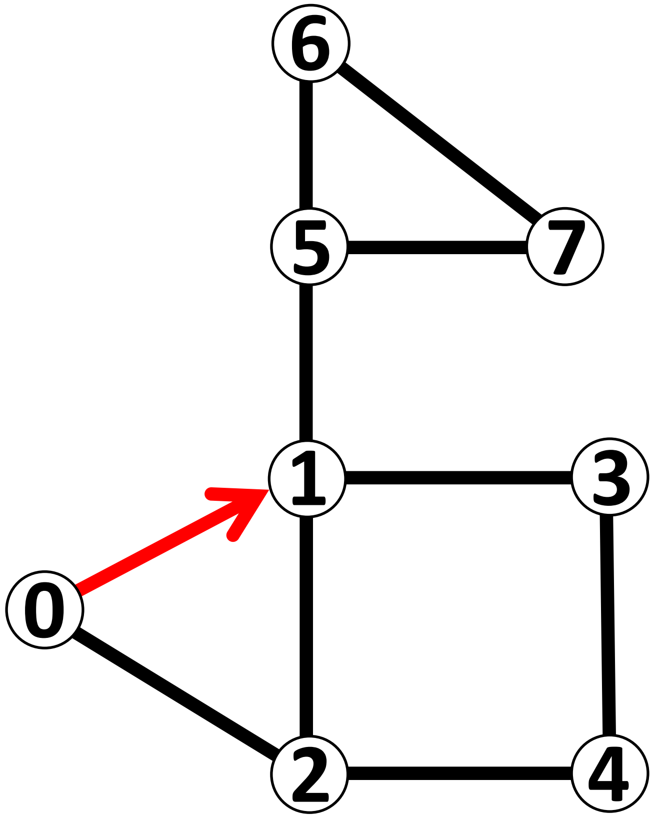
```
public void breadthFirst(int n) {
    Queue<Integer> queue = new LinkedList<>();
    visited[n] = true;
    queue.add(n);
    while(!queue.isEmpty()) {
        int vertex = queue.poll();
        System.out.println(vertex);
        for(int neighbor: getNeighbors(vertex)) {
            if(!visited[neighbor]) {
                previousVertex[neighbor] = vertex;
                visited[neighbor] = true;
                queue.add(neighbor);
            }
        }
    }
}
```



queue 1 2

Output: 0,

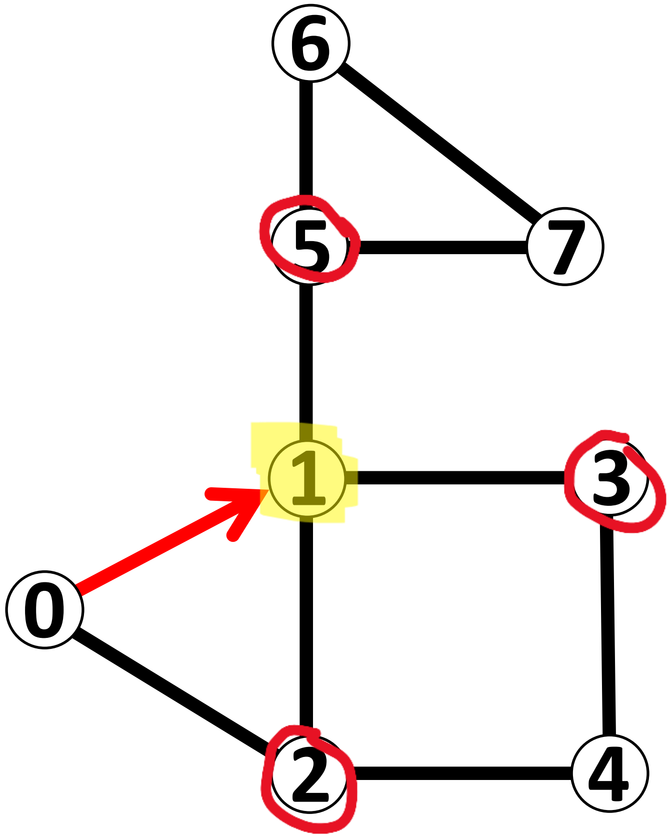
```java
public void breadthFirst(int n) {
    Queue<Integer> queue = new LinkedList<>();
    visited[n] = true;
    queue.add(n);
    while(!queue.isEmpty()) {
        int vertex = queue.poll();
        System.out.println(vertex);
        for(int neighbor: getNeighbors(vertex)) {
            if(!visited[neighbor]) {
                previousVertex[neighbor] = vertex;
                visited[neighbor] = true;
                queue.add(neighbor);
            }
        }
    }
}
```



queue 2

Output: 0, 1

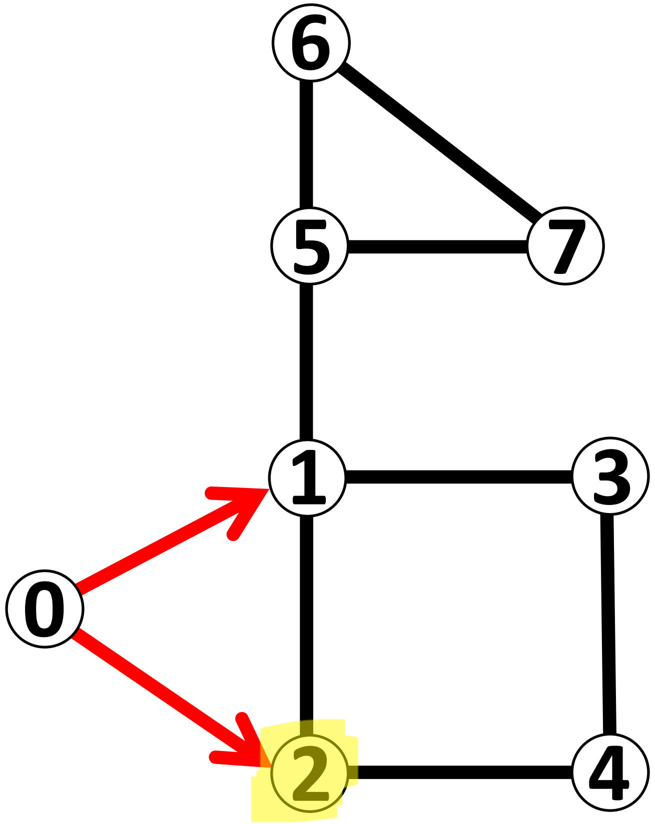
```
public void breadthFirst(int n) {
    Queue<Integer> queue = new LinkedList<>();
    visited[n] = true;
    queue.add(n);
    while(!queue.isEmpty()) {
        int vertex = queue.poll();
        System.out.println(vertex);
        for(int neighbor: getNeighbors(vertex)) {
            if(!visited[neighbor]) {
                previousVertex[neighbor] = vertex;
                visited[neighbor] = true;
                queue.add(neighbor);
            }
        }
    }
}
```



queue 2 3 5

Output: 0, 1

```java
public void breadthFirst(int n) {
    Queue<Integer> queue = new LinkedList<>();
    visited[n] = true;
    queue.add(n);
    while(!queue.isEmpty()) {
        int vertex = queue.poll();
        System.out.println(vertex);
        for(int neighbor: getNeighbors(vertex)) {
            if(!visited[neighbor]) {
                previousVertex[neighbor] = vertex;
                visited[neighbor] = true;
                queue.add(neighbor);
            }
        }
    }
}
```

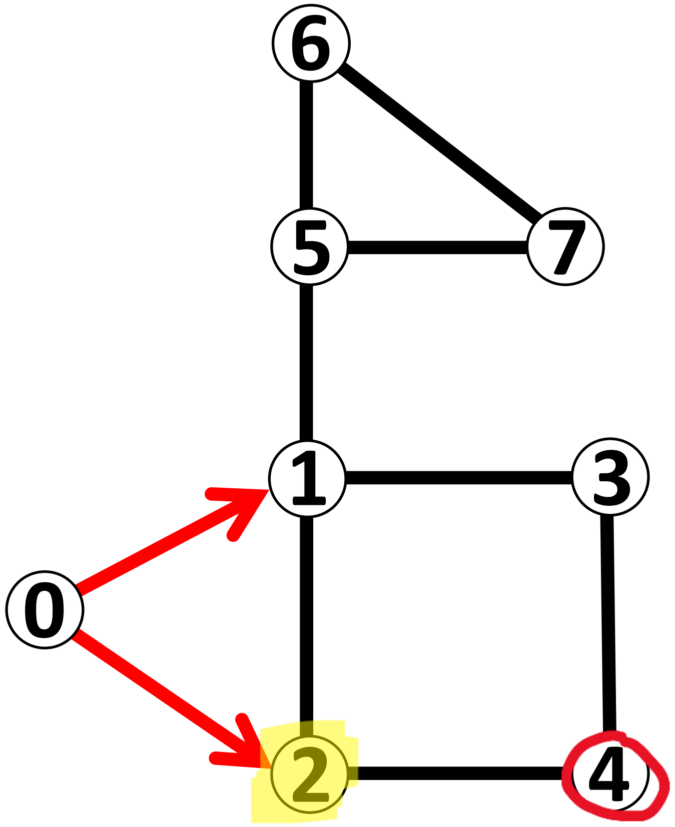


queue 3 5

Output: 0, 1, 2

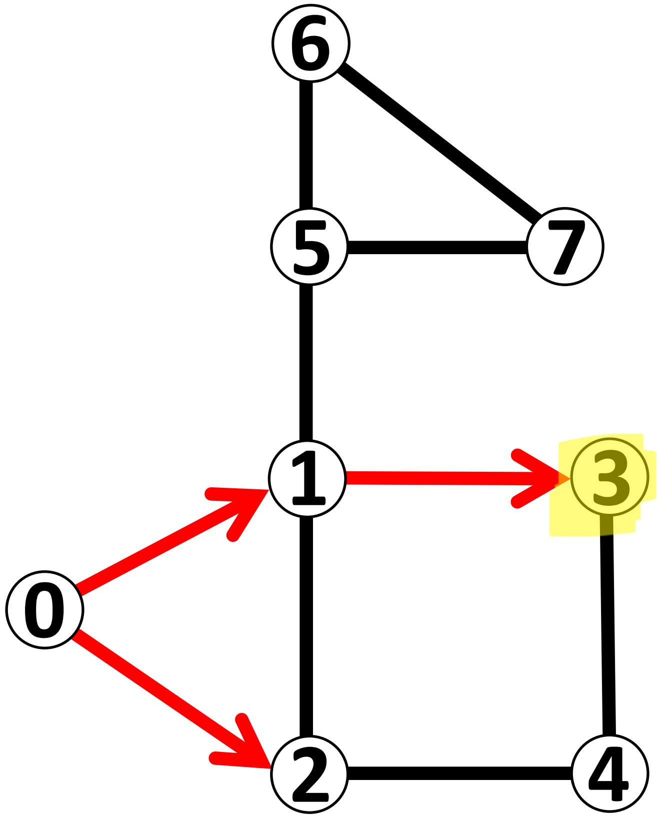
```java
public void breadthFirst(int n) {
    Queue<Integer> queue = new LinkedList<>();
    visited[n] = true;
    queue.add(n);
    while(!queue.isEmpty()) {
        int vertex = queue.poll();
        System.out.println(vertex);
        for(int neighbor: getNeighbors(vertex)) {
            if(!visited[neighbor]) {
                previousVertex[neighbor] = vertex;
                visited[neighbor] = true;
                queue.add(neighbor);
            }
        }
    }
}
```

queue 3 5 4



Output: 0, 1, 2

```java
public void breadthFirst(int n) {
    Queue<Integer> queue = new LinkedList<>();
    visited[n] = true;
    queue.add(n);
    while(!queue.isEmpty()) {
        int vertex = queue.poll();
        System.out.println(vertex);
        for(int neighbor: getNeighbors(vertex)) {
            if(!visited[neighbor]) {
                previousVertex[neighbor] = vertex;
                visited[neighbor] = true;
                queue.add(neighbor);
            }
        }
    }
}
```

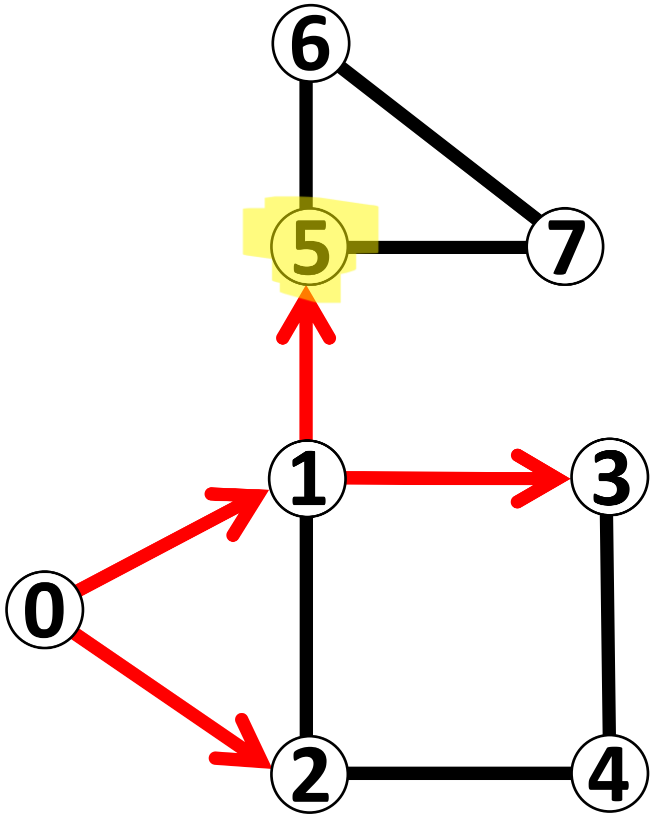


queue 5 4

Output: 0, 1, 2, 3

```
public void breadthFirst(int n) {
    Queue<Integer> queue = new LinkedList<>();
    visited[n] = true;
    queue.add(n);
    while(!queue.isEmpty()) {
        int vertex = queue.poll();
        System.out.println(vertex);
        for(int neighbor: getNeighbors(vertex)) {
            if(!visited[neighbor]) {
                previousVertex[neighbor] = vertex;
                visited[neighbor] = true;
                queue.add(neighbor);
            }
        }
    }
}
```

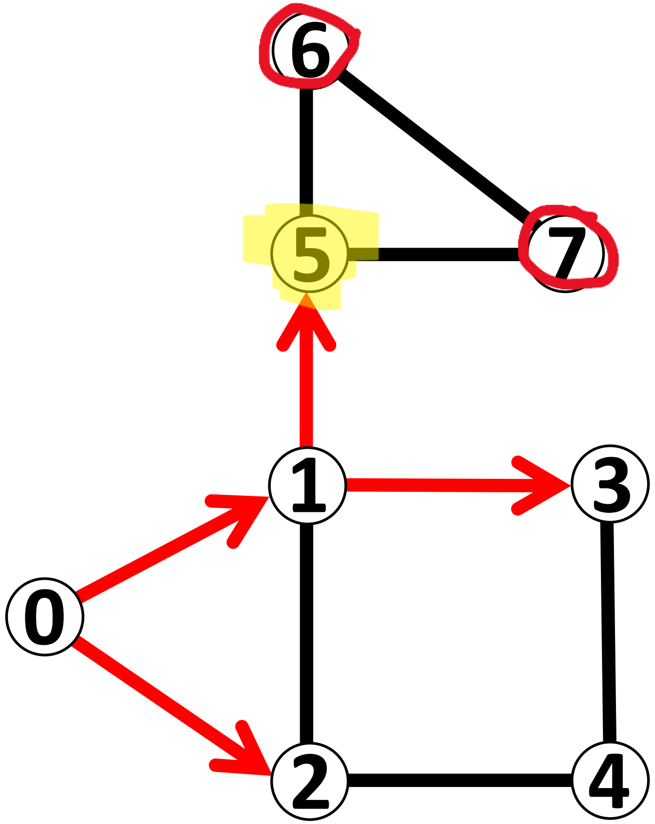
queue 4



Output: 0, 1, 2, 3, 5

```java
public void breadthFirst(int n) {
    Queue<Integer> queue = new LinkedList<>();
    visited[n] = true;
    queue.add(n);
    while(!queue.isEmpty()) {
        int vertex = queue.poll();
        System.out.println(vertex);
        for(int neighbor: getNeighbors(vertex)) {
            if(!visited[neighbor]) {
                previousVertex[neighbor] = vertex;
                visited[neighbor] = true;
                queue.add(neighbor);
            }
        }
    }
}
```

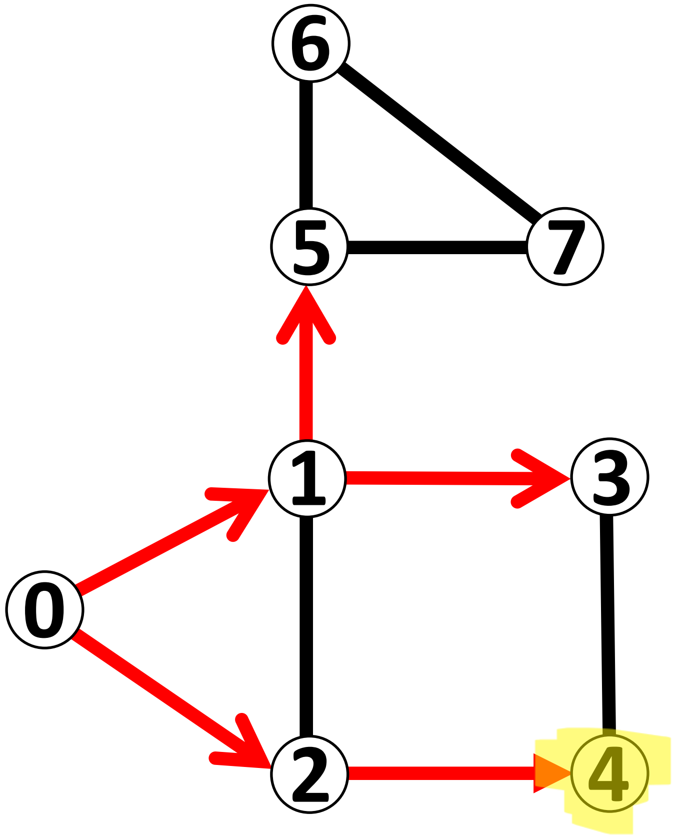
queue 4 6 7



Output: 0, 1, 2, 3, 5

```
public void breadthFirst(int n) {
    Queue<Integer> queue = new LinkedList<>();
    visited[n] = true;
    queue.add(n);
    while(!queue.isEmpty()) {
        int vertex = queue.poll();
        System.out.println(vertex);
        for(int neighbor: getNeighbors(vertex)) {
            if(!visited[neighbor]) {
                previousVertex[neighbor] = vertex;
                visited[neighbor] = true;
                queue.add(neighbor);
            }
        }
    }
}
```

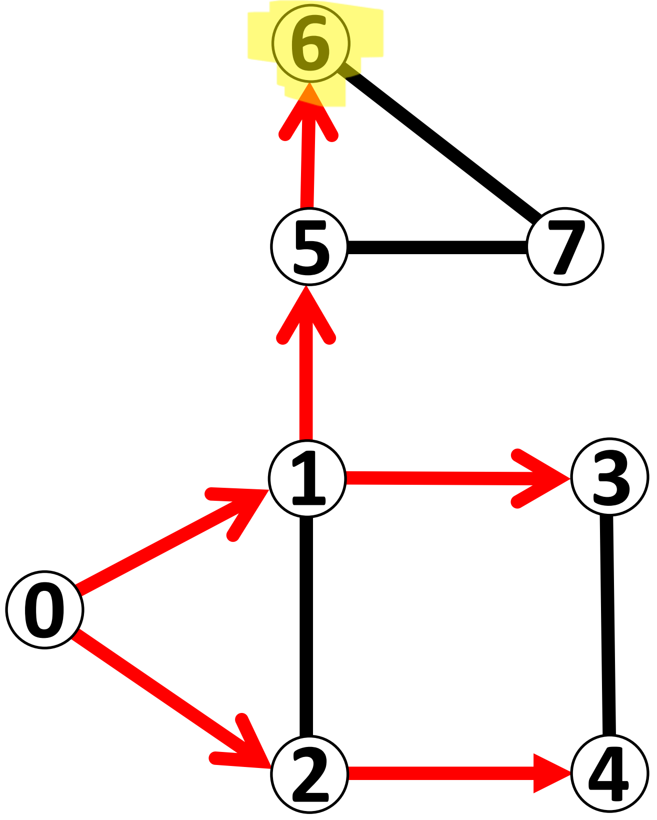
queue 6 7



Output: 0, 1, 2, 3, 5, 4

```
public void breadthFirst(int n) {
    Queue<Integer> queue = new LinkedList<>();
    visited[n] = true;
    queue.add(n);
    while(!queue.isEmpty()) {
        int vertex = queue.poll();
        System.out.println(vertex);
        for(int neighbor: getNeighbors(vertex)) {
            if(!visited[neighbor]) {
                previousVertex[neighbor] = vertex;
                visited[neighbor] = true;
                queue.add(neighbor);
            }
        }
    }
}
```

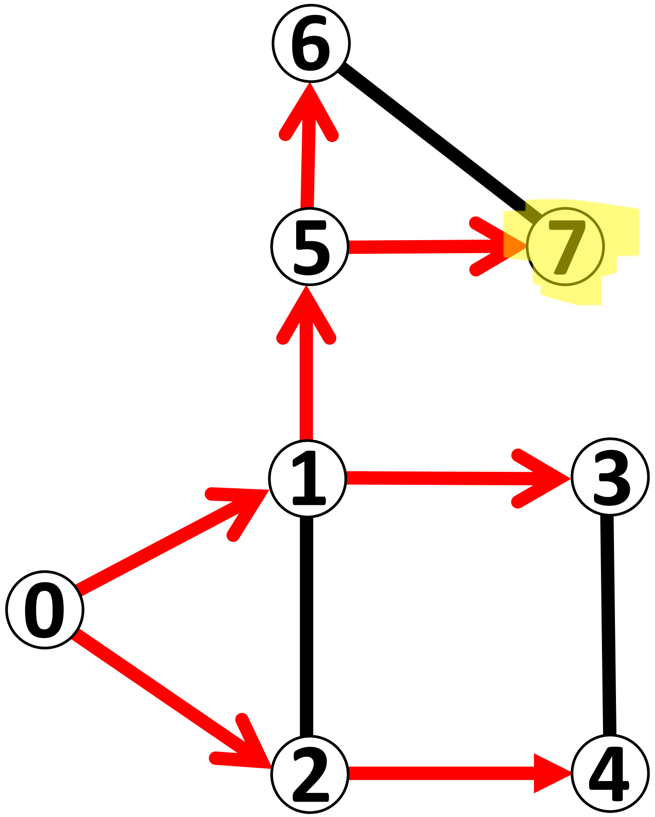
queue 7



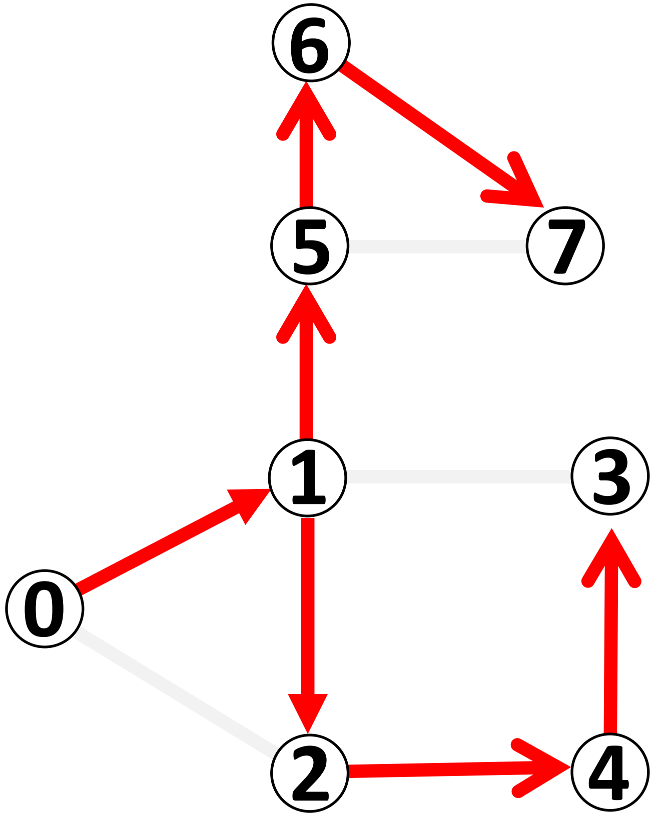
Output: 0, 1, 2, 3, 5, 4, 6

```
public void breadthFirst(int n) {
    Queue<Integer> queue = new LinkedList<>();
    visited[n] = true;
    queue.add(n);
    while(!queue.isEmpty()) {
        int vertex = queue.poll();
        System.out.println(vertex);
        for(int neighbor: getNeighbors(vertex)) {
            if(!visited[neighbor]) {
                previousVertex[neighbor] = vertex;
                visited[neighbor] = true;
                queue.add(neighbor);
            }
        }
    }
}
```

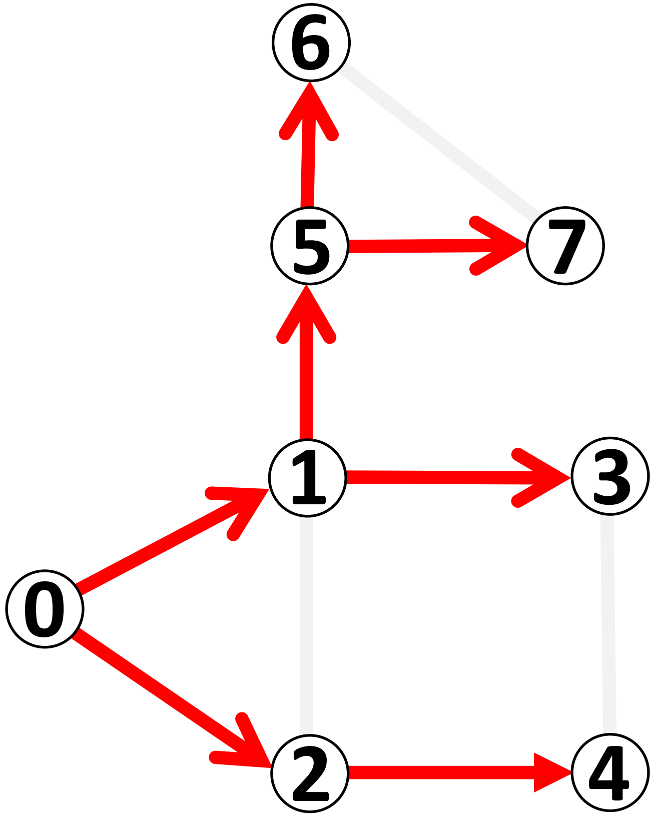
queue



Output: 0, 1, 2, 3, 5, 4, 6, 7



Depth First Order*:
0, 1, 2, 4, 3, 5, 6, 7



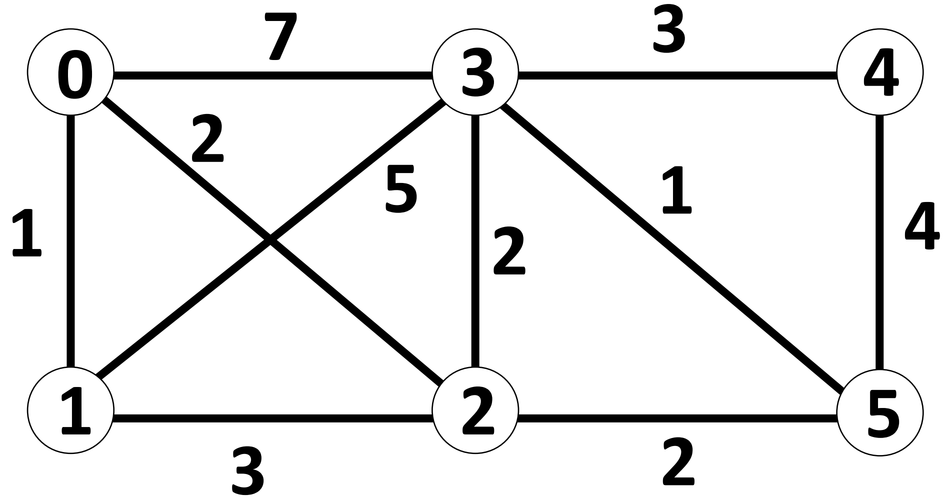
Breadth First Order*
0, 1, 2, 3, 5, 4, 6, 7

Given a starting point, DFS and BFS will visit every vertex in a graph it is a **connected** graph

*not the only DFS and BFS order

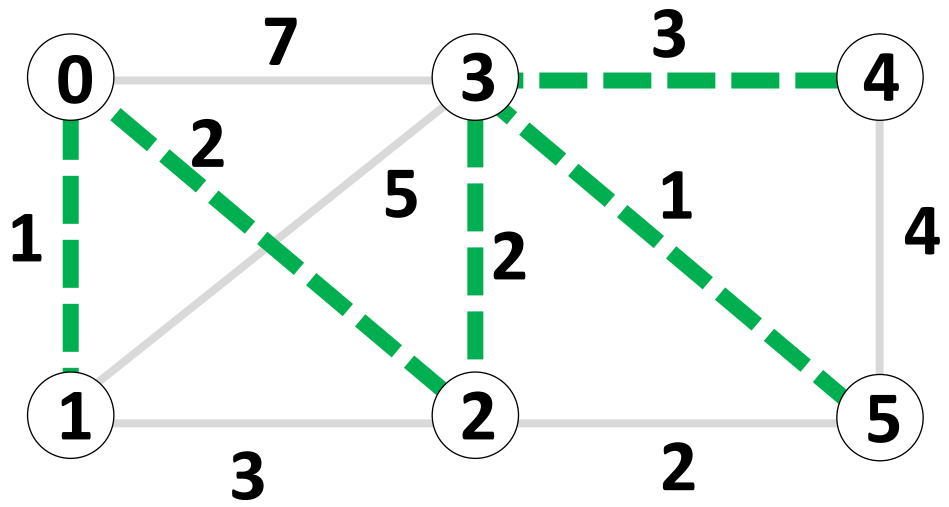
Lab 8

Minimum Spanning Tree



Edge-weighted graph: A graph where each edge has a weight (cost).

Minimum Spanning Tree

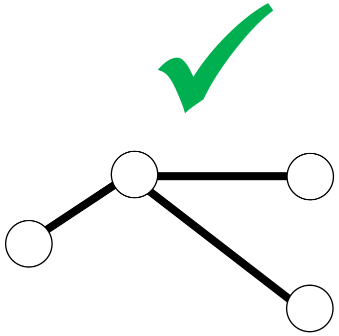


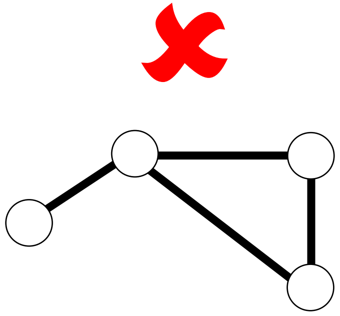
Edge-weighted graph: A graph where each edge has a weight (cost).

MST Goal: Connect all vertices to each other with a minimum weight subset of edges.

Minimum Spanning Tree

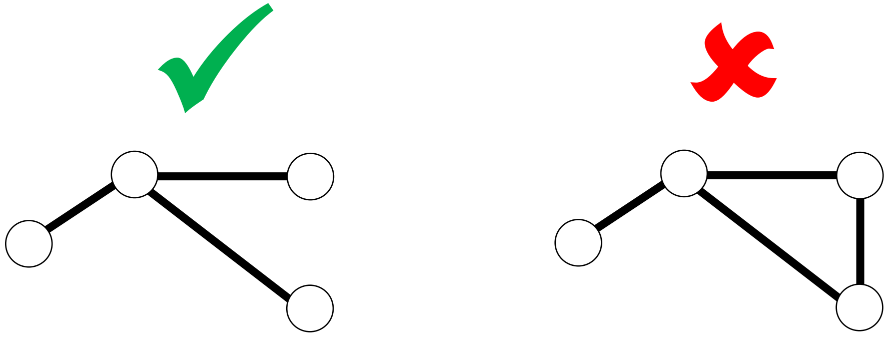
Tree – connected graph with no loops.



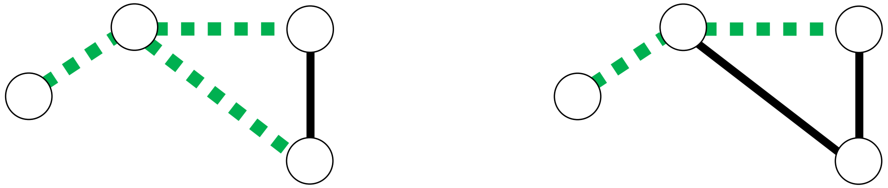


Minimum Spanning Tree

Tree – connected graph with no loops.

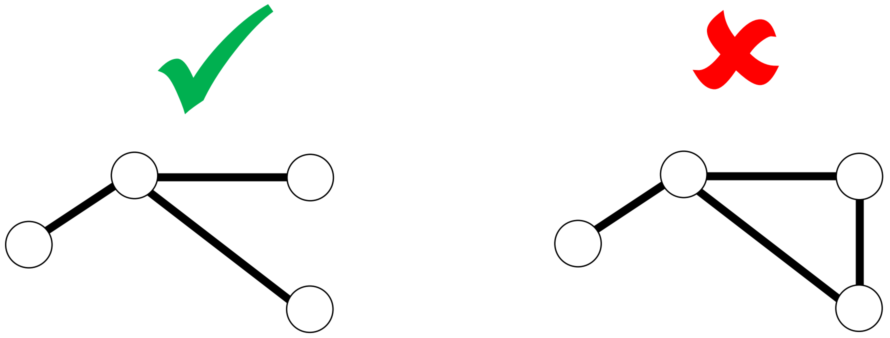


Spanning tree – tree that includes all vertices in a graph.

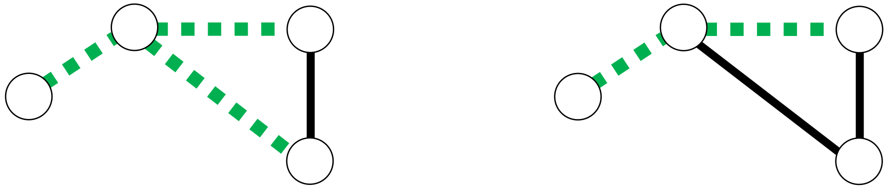


Minimum Spanning Tree

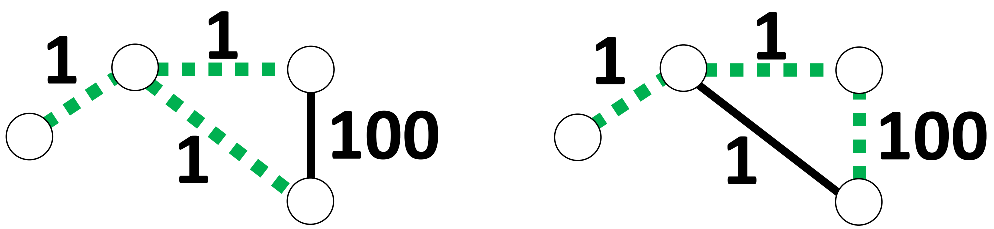
Tree – connected graph with no loops.



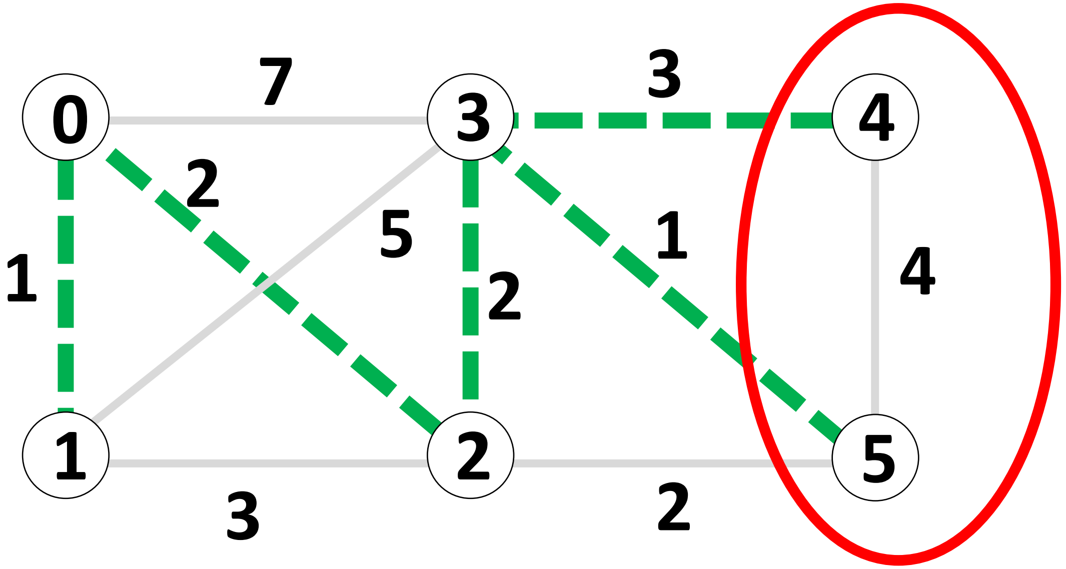
Spanning tree – tree that includes all vertices in a graph.



Minimum spanning tree – spanning tree whose sum of edge costs is the minimum possible value.



Minimum Spanning Tree



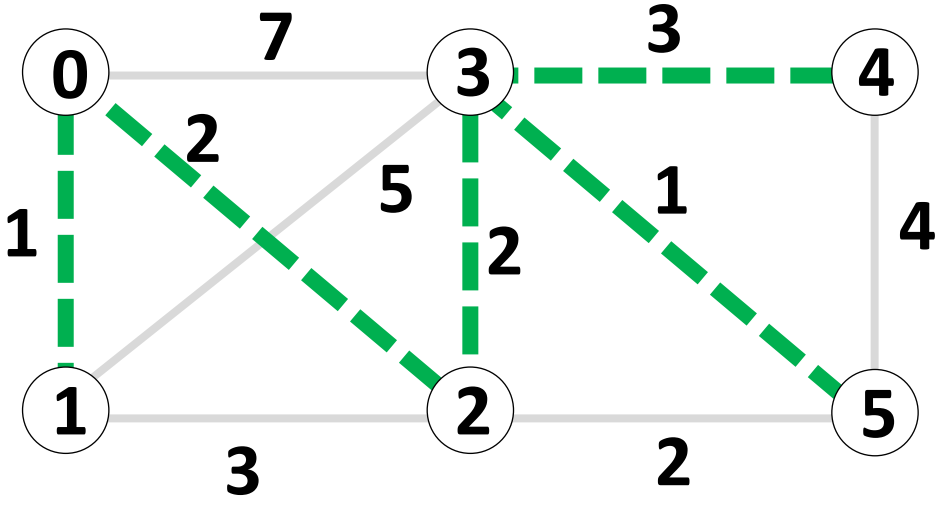
Does it ever make sense to have a cycle?

No!

Must be a tree!

MST Goal: Connect all vertices to each other with a minimum weight subset of edges.

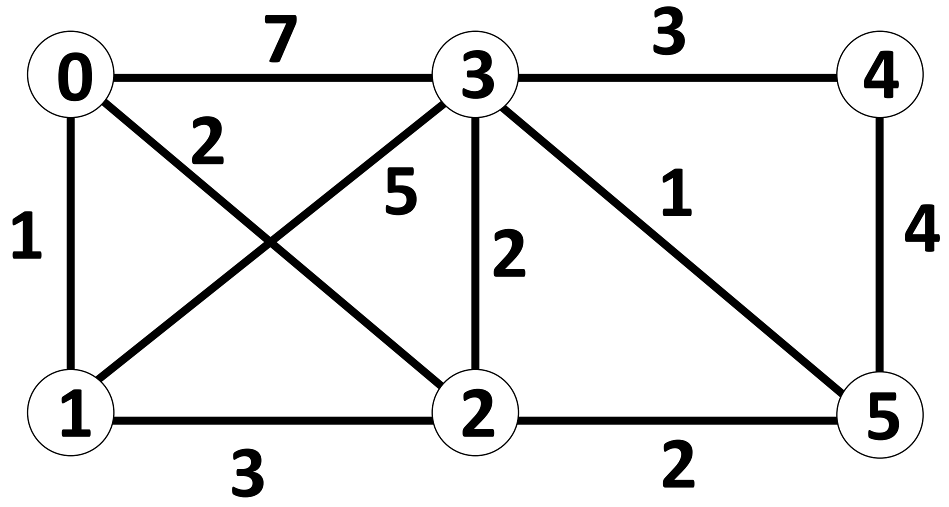
Minimum Spanning Tree



How to find MSTs?

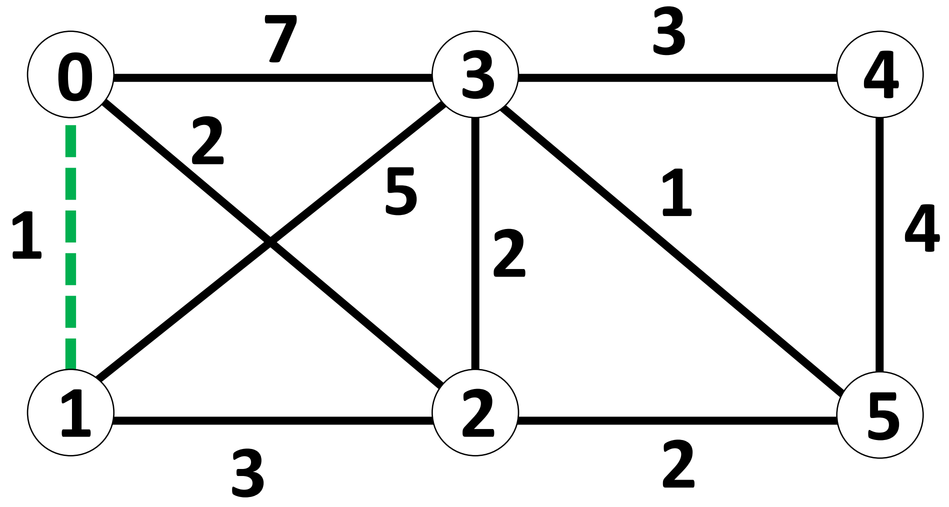
Kruskal’s MST Algorithm

At each iteration, add the edge with smallest weight, that does not create a cycle.



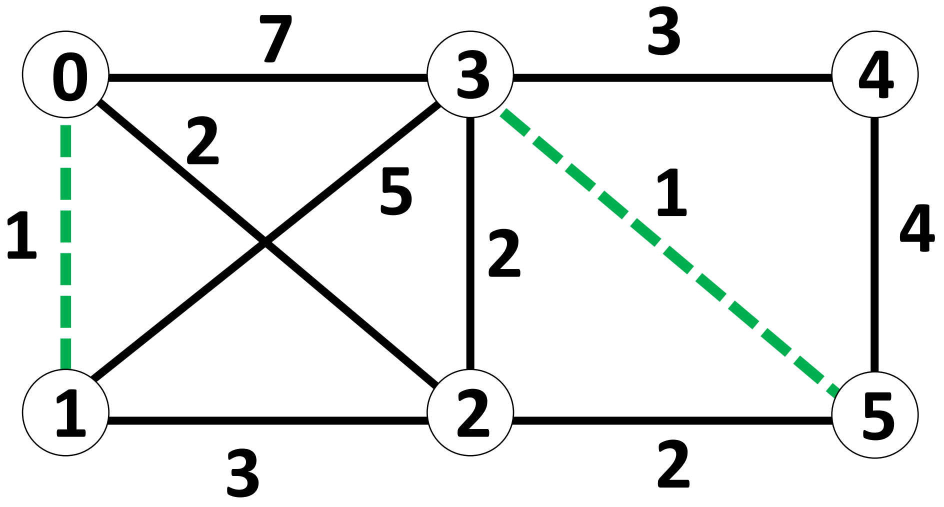
Kruskal’s MST Algorithm

At each iteration, add the edge with smallest weight, that does not create a cycle.



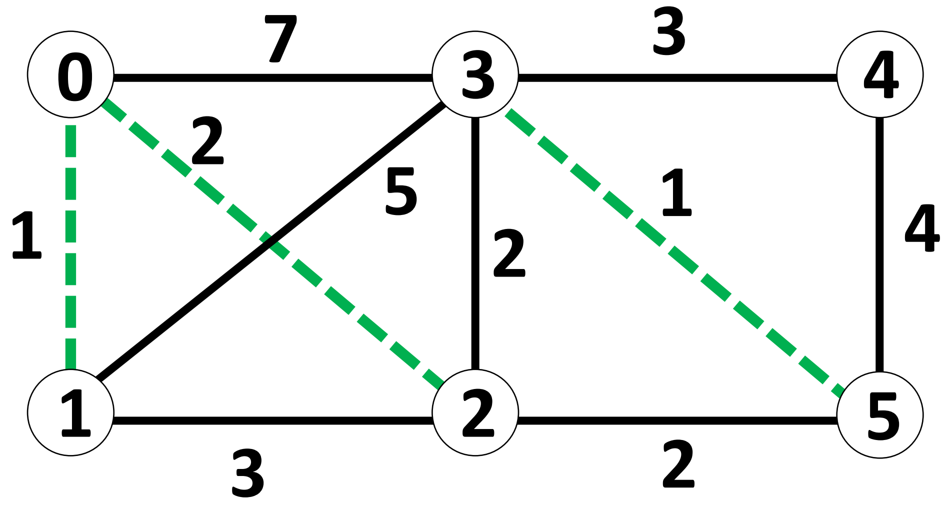
Kruskal’s MST Algorithm

At each iteration, add the edge with smallest weight, that does not create a cycle.



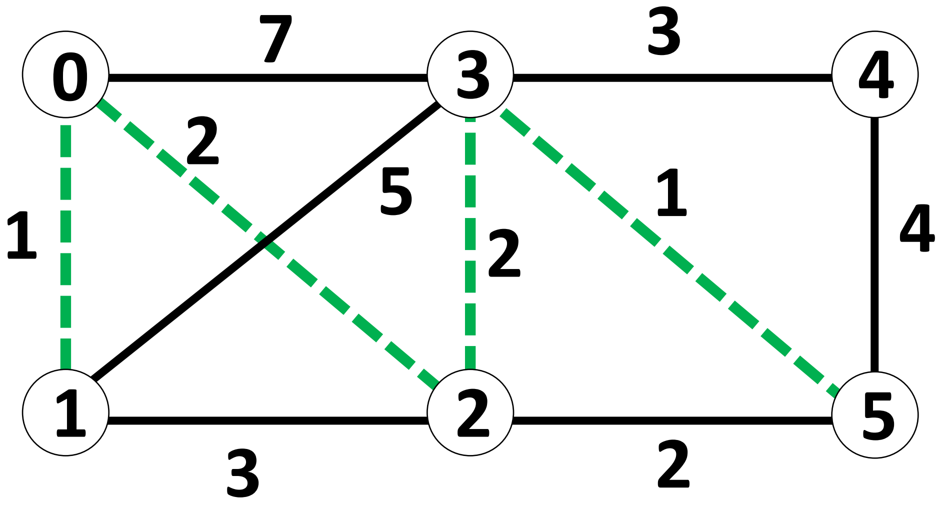
Kruskal’s MST Algorithm

At each iteration, add the edge with smallest weight, that does not create a cycle.



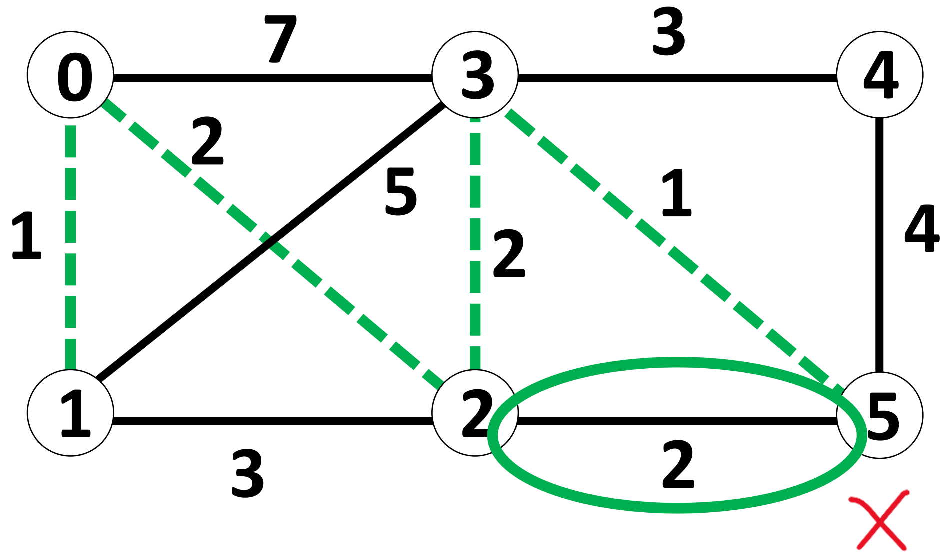
Kruskal’s MST Algorithm

At each iteration, add the edge with smallest weight, that does not create a cycle.



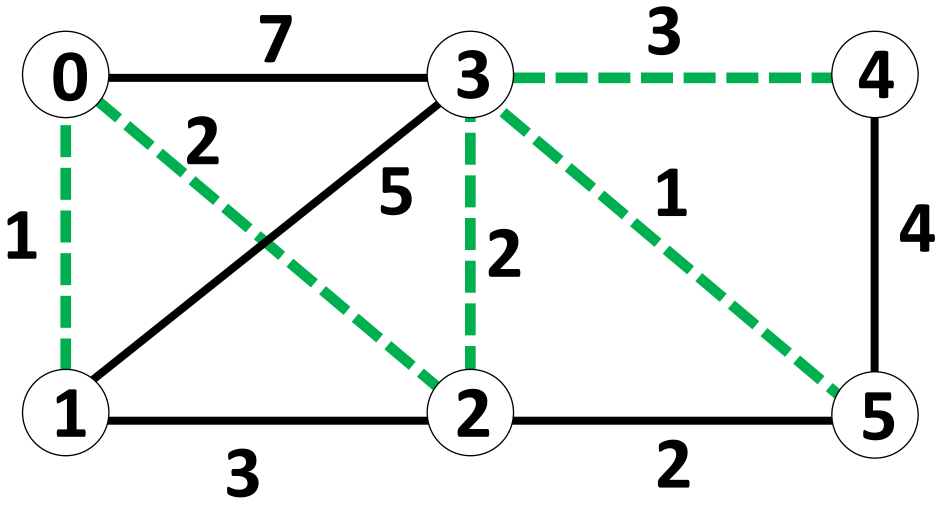
Kruskal’s MST Algorithm

At each iteration, add the edge with smallest weight, that does not create a cycle.



Kruskal’s MST Algorithm

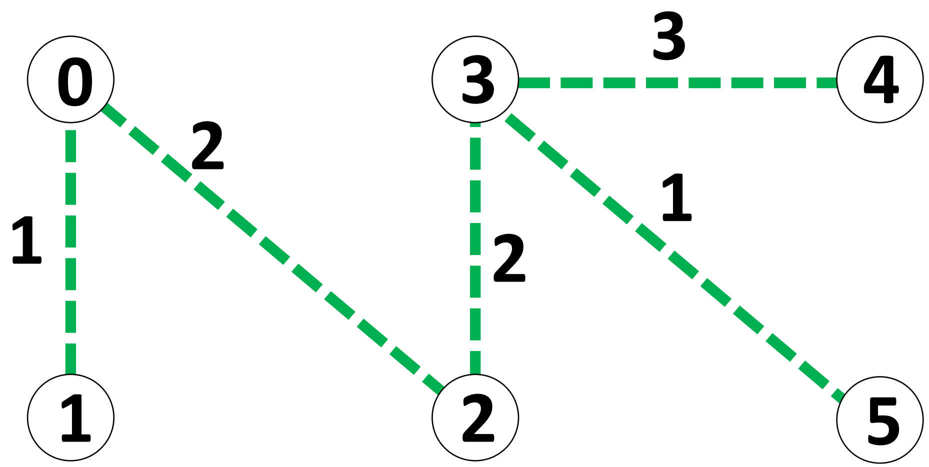
At each iteration, add the edge with smallest weight, that does not create a cycle.



Kruskal’s MST Algorithm

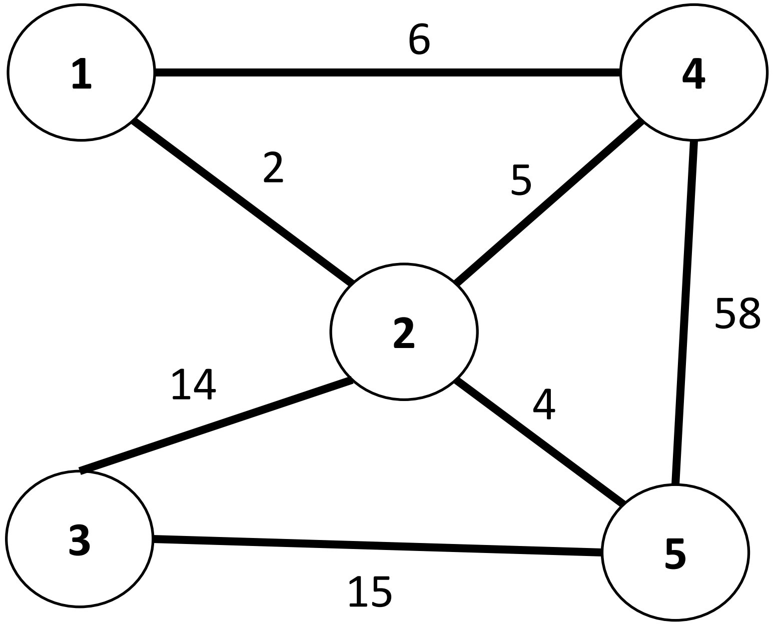
At each iteration, add the edge with smallest weight, that does not create a cycle.

MST = [0, 1], [0, 2], [2,3], [3,5], [3,4] **Total Cost = 9**



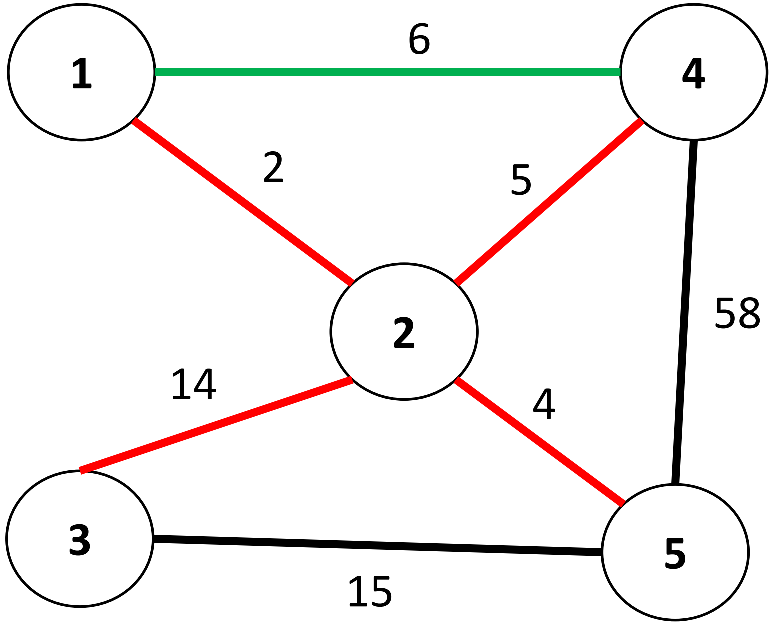
MST vs Shortest Path

MST and shortest path are two different problems, and sometimes that shortest path will not be part of the MST



MST vs Shortest Path

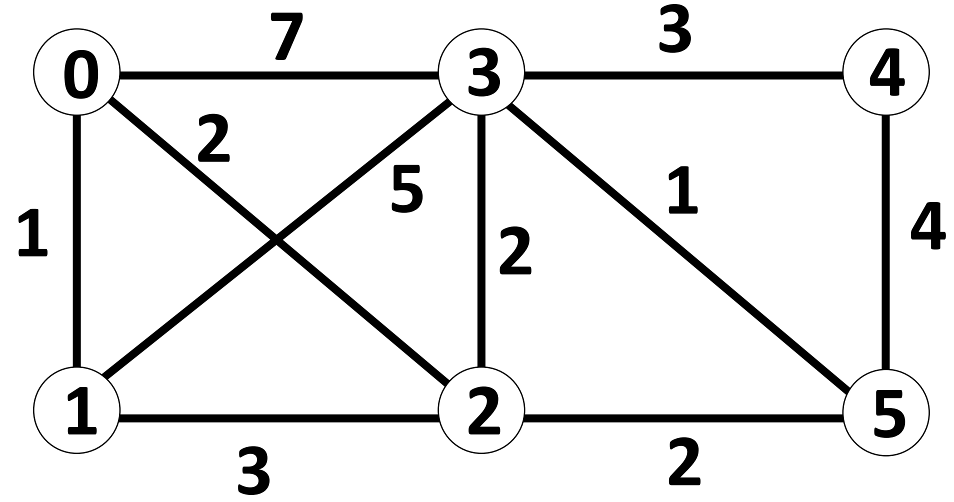
MST and shortest path are two different problems, and sometimes that shortest path will not be part of the MST



MST Cost = 25

Shortest Path from 1 to 4 = 6

Weighted Graph



```
public class Edge {

        private int vertex1;
        private int vertex2;

        private int weight;

        public int[] getVertices()

        public int getWeight()

        public String toString()

        public boolean equals()
```