CSCI 232:
Data Structures and Algorithms

Graphs (Traversal and Searching)

Reese Pearsall

Spring 2025

Announcements

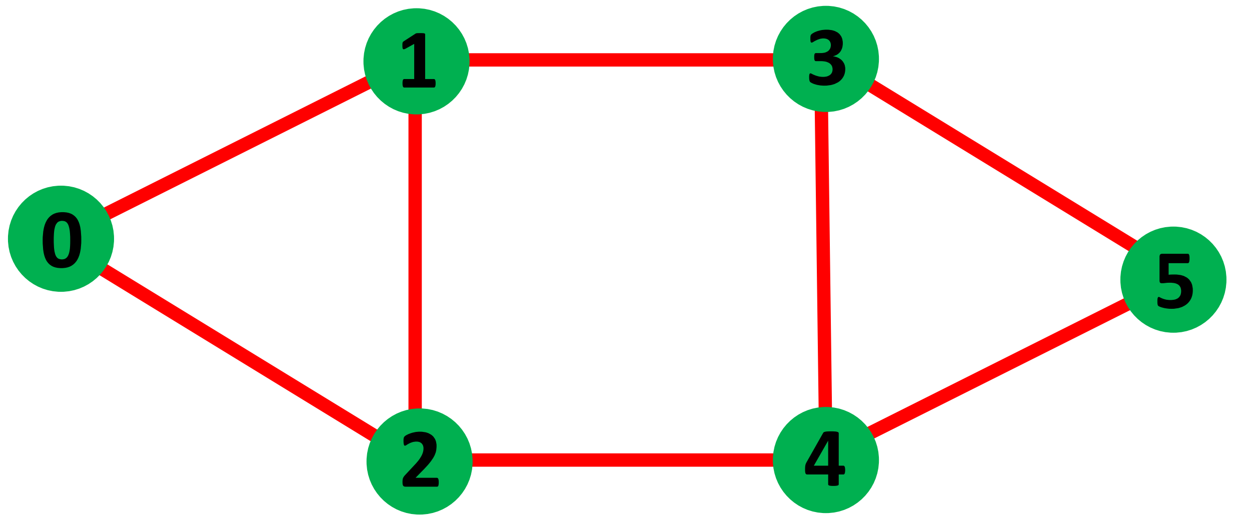
Lab 7 due tomorrow

No class next week (spring break)

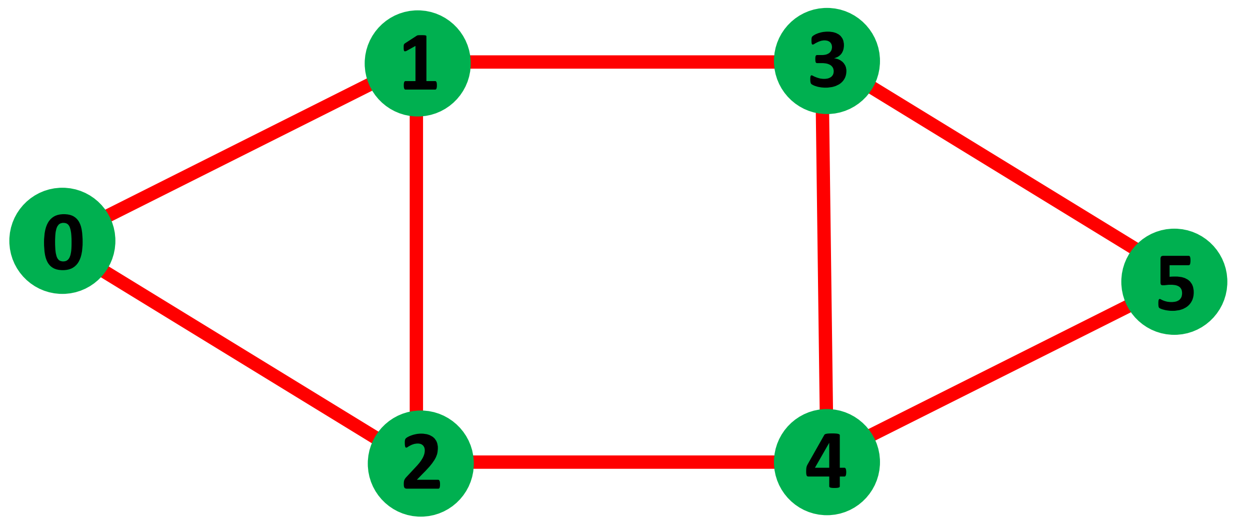
Graphs

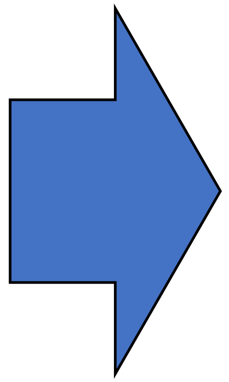
$$G = (V, E)$$

TREE	GRAPH
There exists a hierarchical structure, and the top node is called the root node.	The concept of hierarchy leading to a unique root node does not apply here.
Is an acyclic graph.	Cycles can exist.
Must be a connected graph.	Isn't necessarily a connected graph.
Data representation is similar to a tree, with concepts like branches, roots, and leaf nodes.	Data representation is similar to a network
Applications: decision trees, implementing heaps to find max/min numbers, sorting.	Applications: Finding shortest path, navigation, route optimization.



Graphs

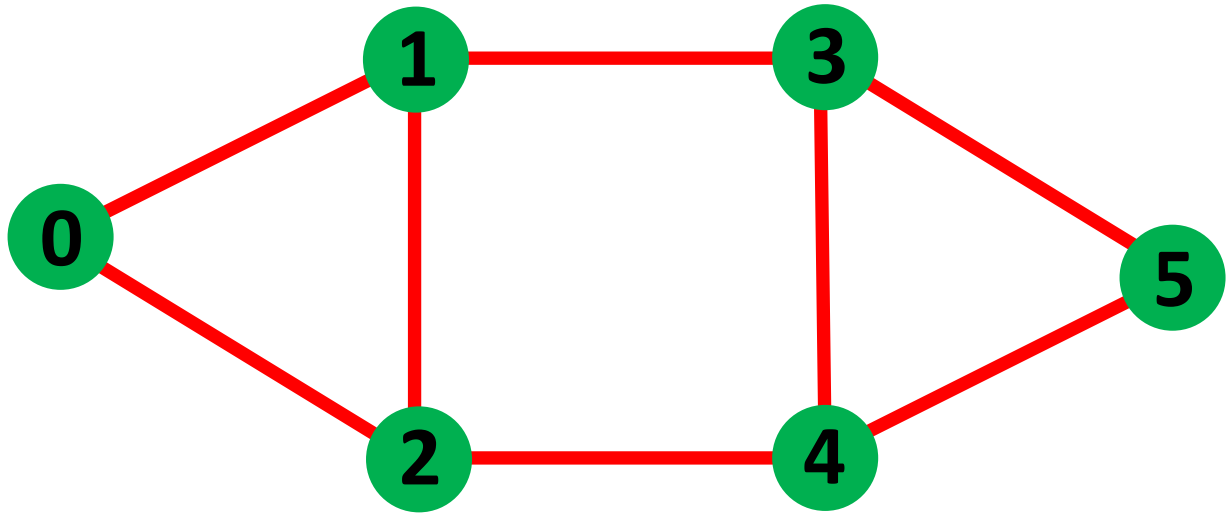




Adjacency List

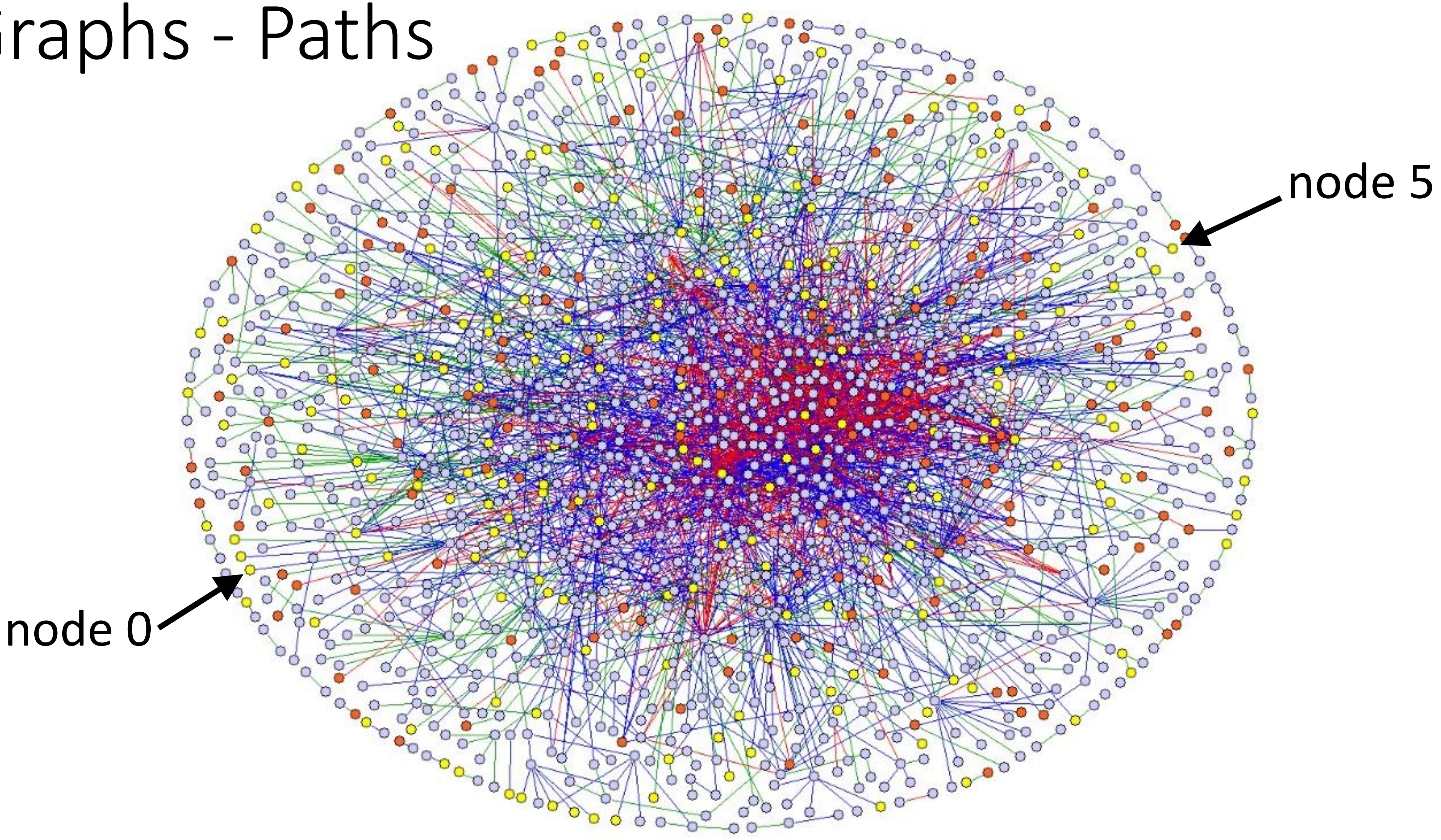
0	→	{1,2}
1	→	{0,2,3}
2	→	{0,1,4}
3	→	{1,4,5}
4	→	{2,3,5}
5	→	{3,4}

Graphs - Paths



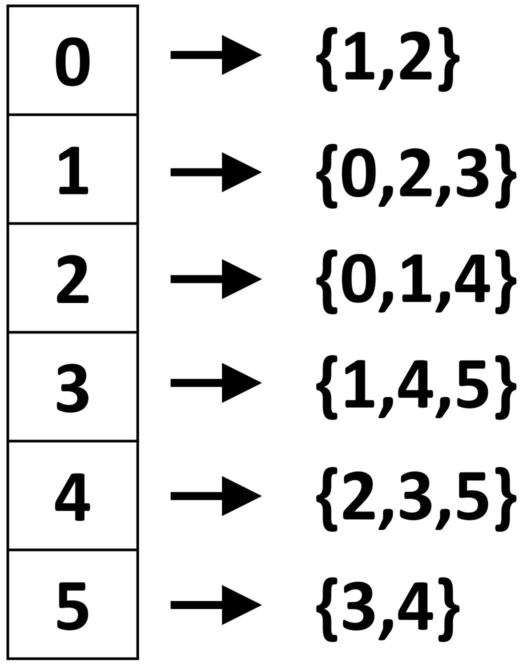
Is there a path from node 0 to node 5?

Graphs - Paths



Is there a path from node 0 to node 5?

Graphs - Paths



Is there a path from node 0 to node 5?

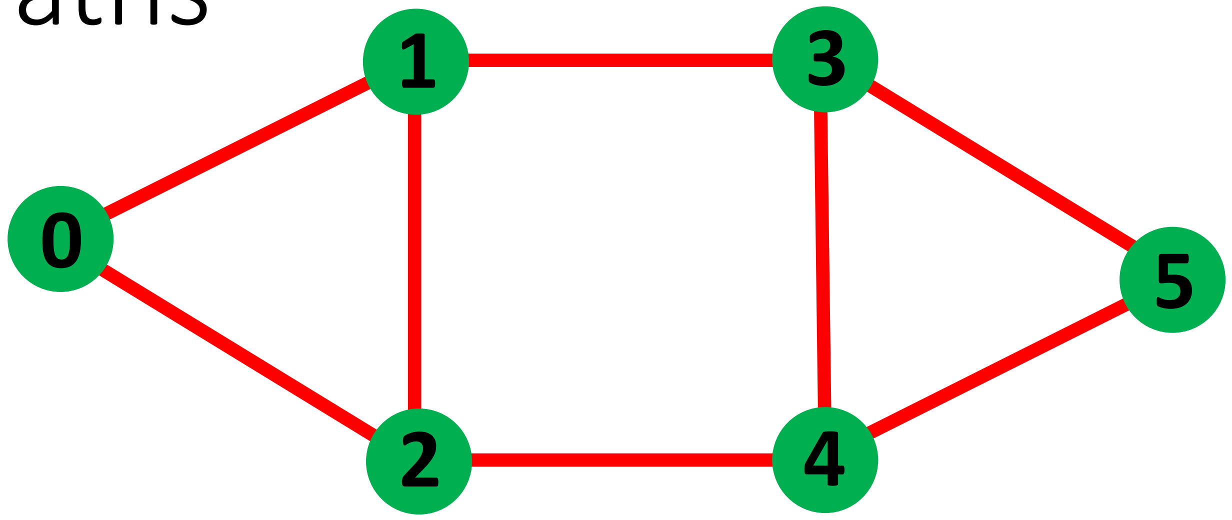
Graphs - Paths

$$\begin{tikzcd}
\boxed{0} \arrow[r] & \{1,2\} \\
\boxed{1} \arrow[r] & \{0,2,3\} \\
\boxed{2} \arrow[r] & \{0,1,4\} \\
\boxed{3} \arrow[r] & \{1,4,5\} \\
\boxed{4} \arrow[r] & \{2,3,5\} \\
\boxed{5} \arrow[r] & \{3,4\}
\end{tikzcd}$$

We need a better process than “eyeballing it” for finding paths.

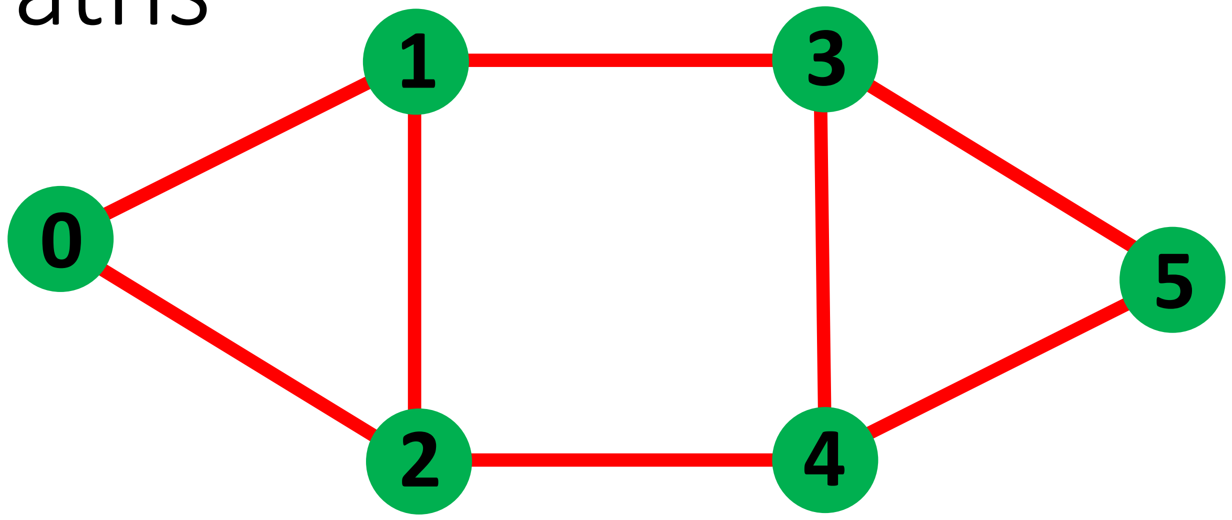
Is there a path from node 0 to node 5?

Graphs - Paths



What is a **generalizable** process to see if there is a path from node 0 to node 5?

Graphs - Paths



What is a <u>**generalizable**</u> process to see if there is a path from node 0 to node 5?

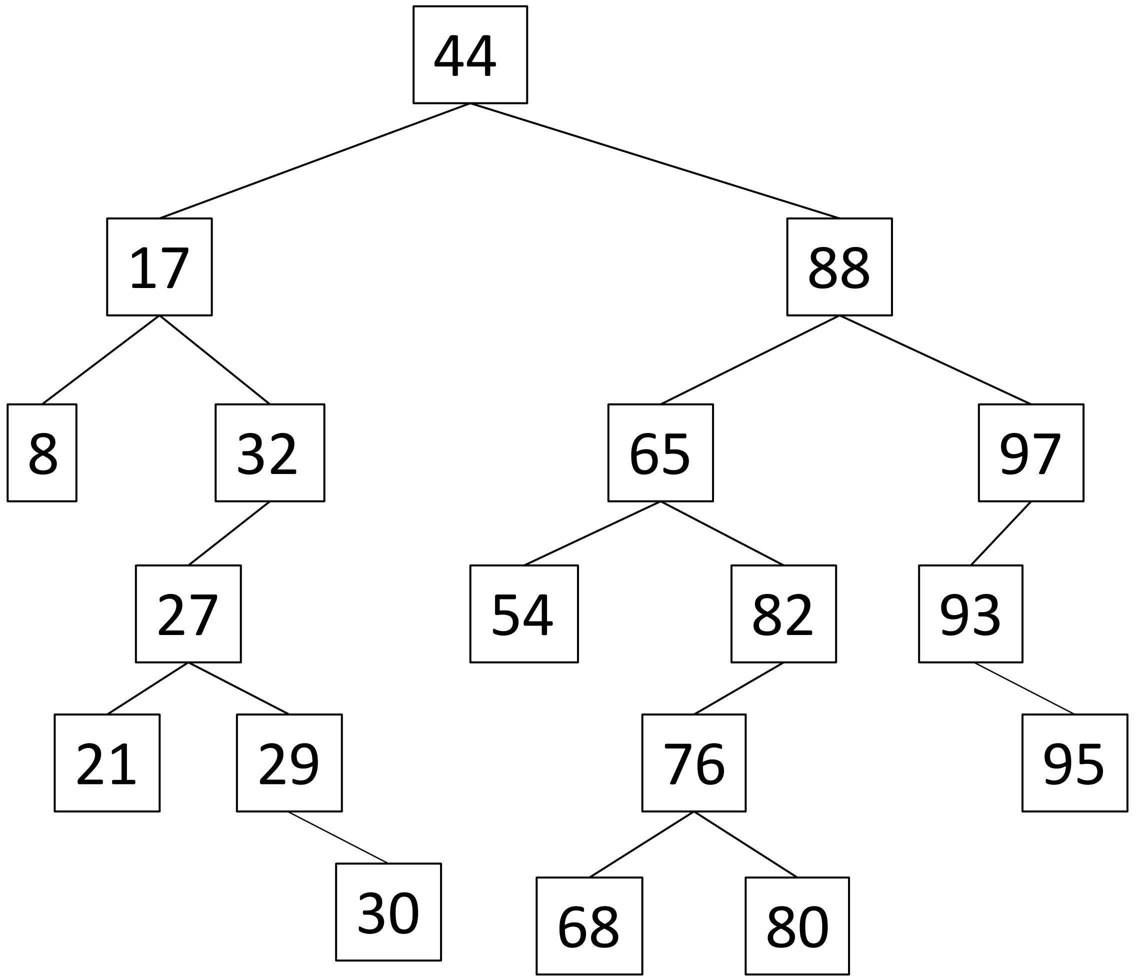
Start at node 0.
Go to each neighbor.
Check each neighbor’s neighbor.
Check each neighbor’s neighbor’s neighbor….

Binary Search Tree - Traversal

```
public void depthFirst(Node n) {
    if (n != null) {
        System.out.println(n.getValue());
        depthFirst(n.getLeft());
        depthFirst(n.getRight());
    }
}
```

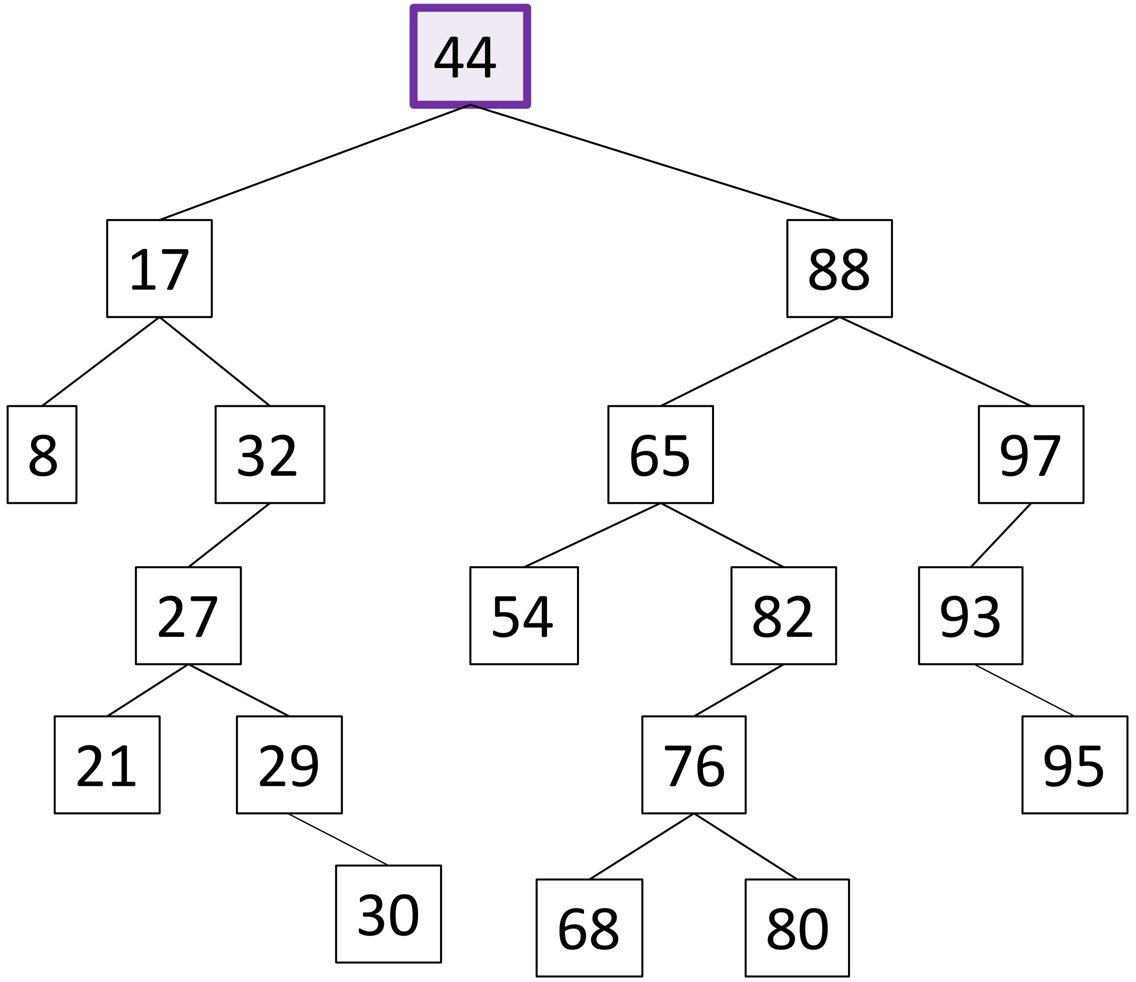
Recursion:

- **Calling a method from inside itself.**
- **Solve the problem by solving identical smaller problems.**
- **What is the "smaller problem"?**
  - **Process the left side, then process the right side.**



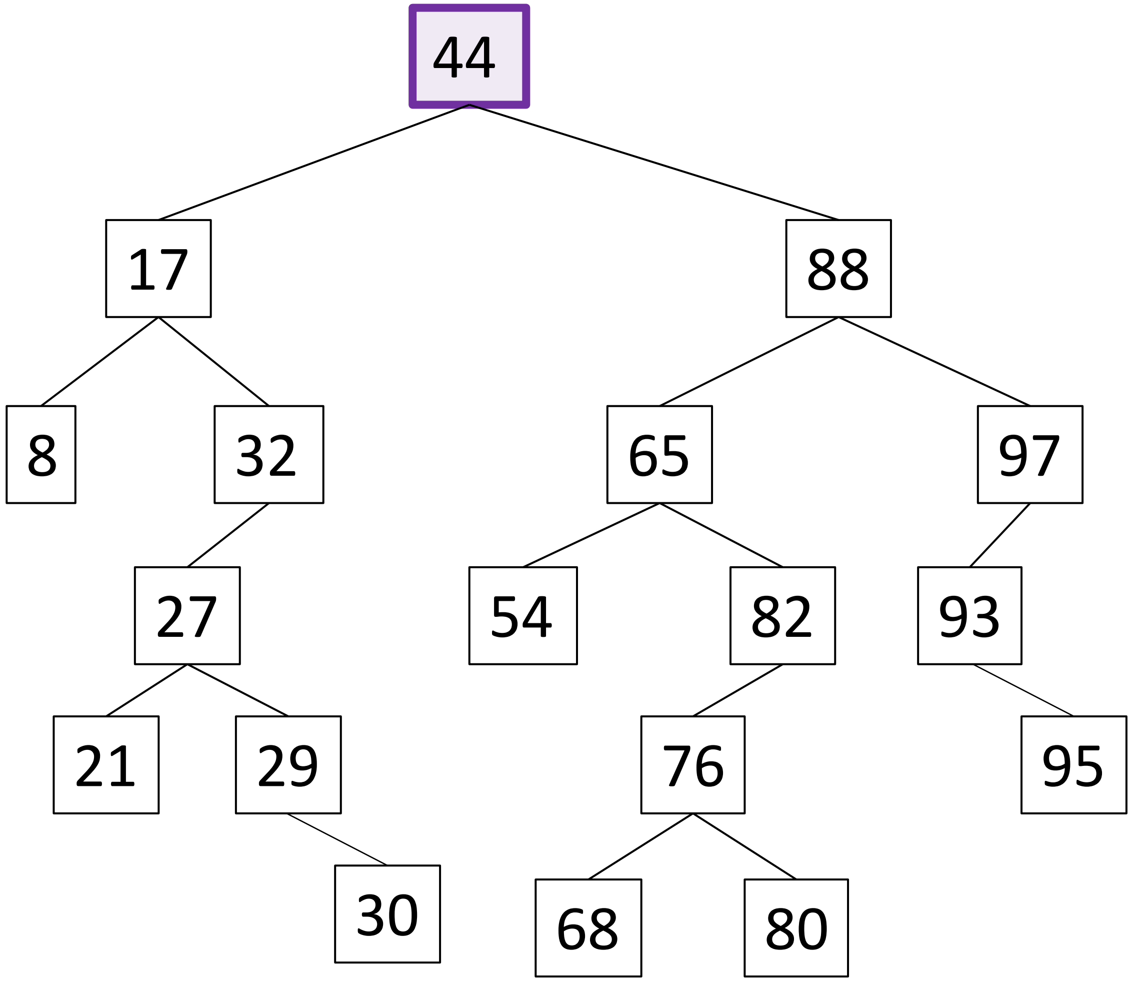
Binary Search Tree - Traversal

```
public void depthFirst(44) {
    if (n != null) {
        System.out.println(n.getValue());
        depthFirst(n.getLeft());
        depthFirst(n.getRight());
    }
}
```



Binary Search Tree - Traversal

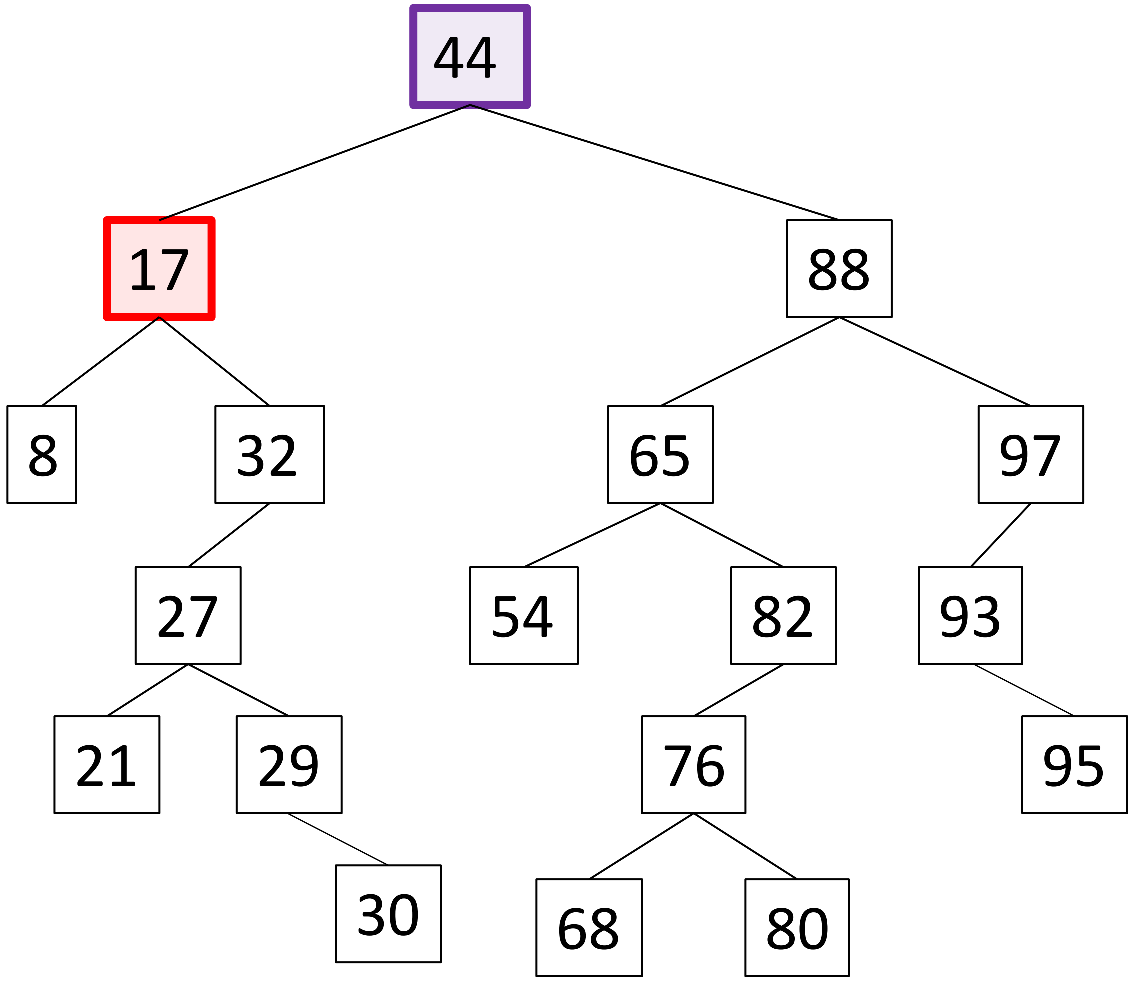
```
public void depthFirst(44) {
    if (n != null) {
        System.out.println(n.getValue());
        depthFirst(n.getLeft());
        depthFirst(n.getRight());
    }
}
```



Binary Search Tree - Traversal

```
public void depthFirst(44) {
    if (n != null) {
        System.out.println(n.getValue());
        depthFirst(n.getLeft());
        depthFirst(n.getRight());
    }
}
```

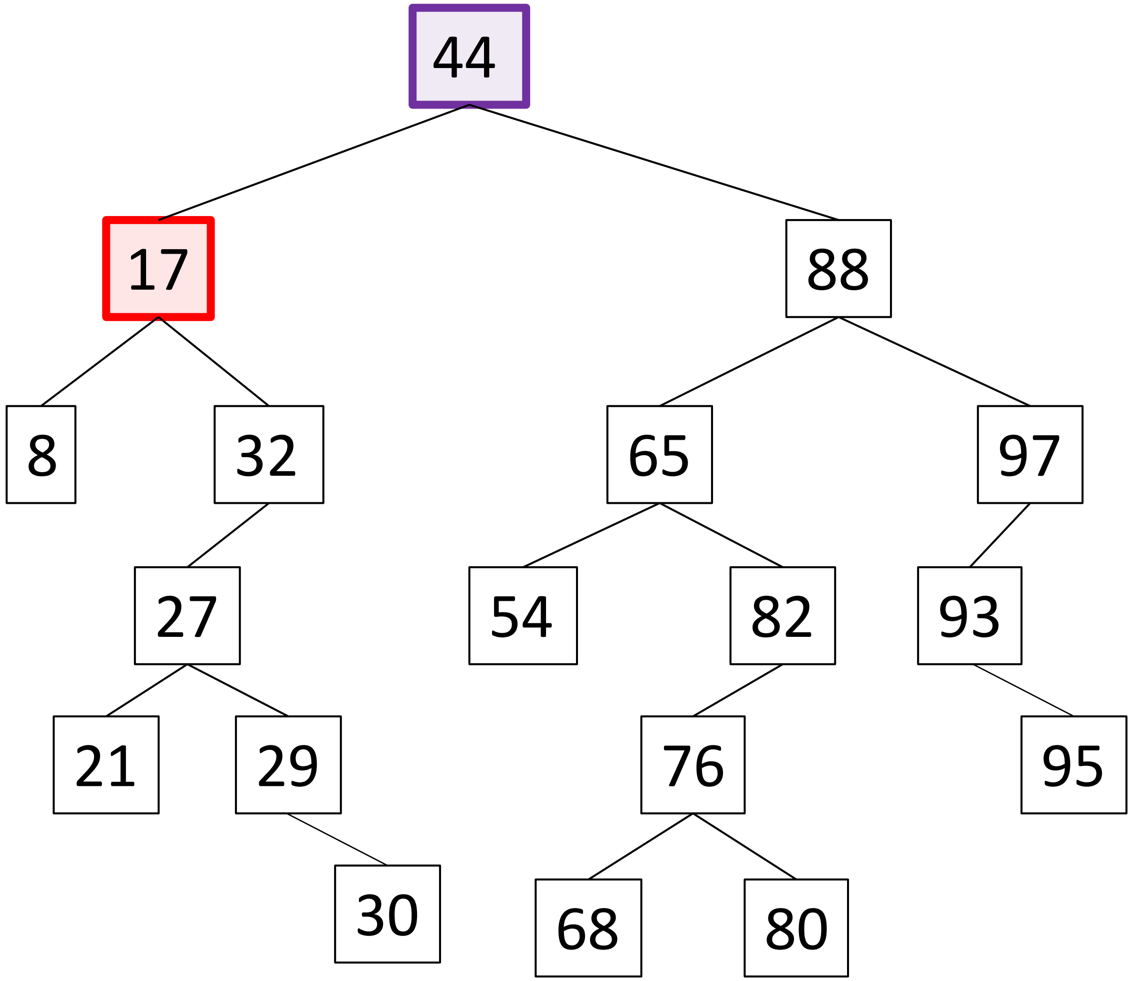
```
public void depthFirst(17) {
    if (n != null) {
        System.out.println(n.getValue());
        depthFirst(n.getLeft());
        depthFirst(n.getRight());
    }
}
```



Binary Search Tree - Traversal

```
public void depthFirst(44) {
    if (n != null) {
        System.out.println(n.getValue());
        depthFirst(n.getLeft());
        depthFirst(n.getRight());
    }
}
```

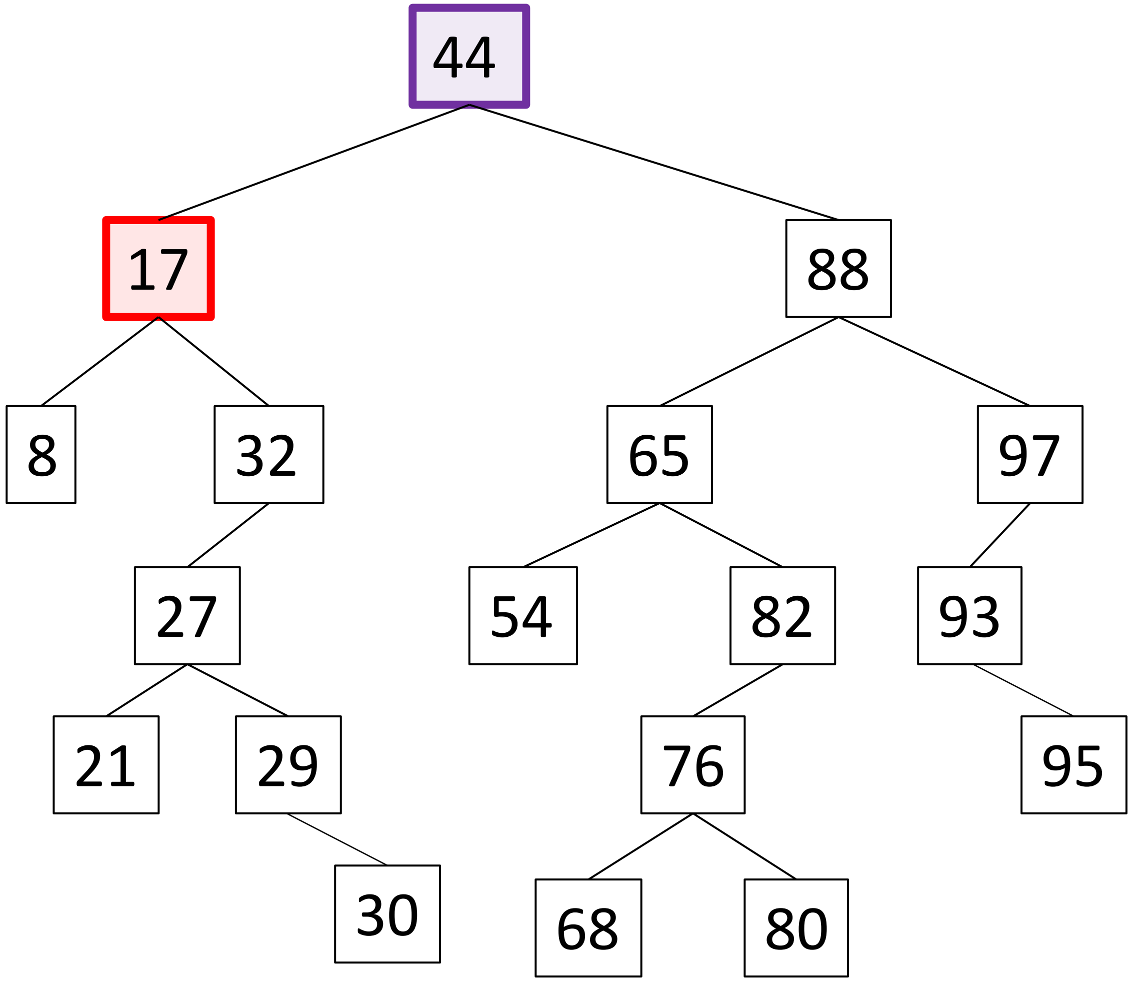
```
public void depthFirst(17) {
    if (n != null) {
        System.out.println(n.getValue());
        depthFirst(n.getLeft());
        depthFirst(n.getRight());
    }
}
```



Binary Search Tree - Traversal

```
public void depthFirst(44) {
    if (n != null) {
        System.out.println(n.getValue());
        depthFirst(n.getLeft());
        depthFirst(n.getRight());
    }
}
```

```
public void depthFirst(17) {
    if (n != null) {
        System.out.println(n.getValue());
        depthFirst(n.getLeft());
        depthFirst(n.getRight());
    }
}
```

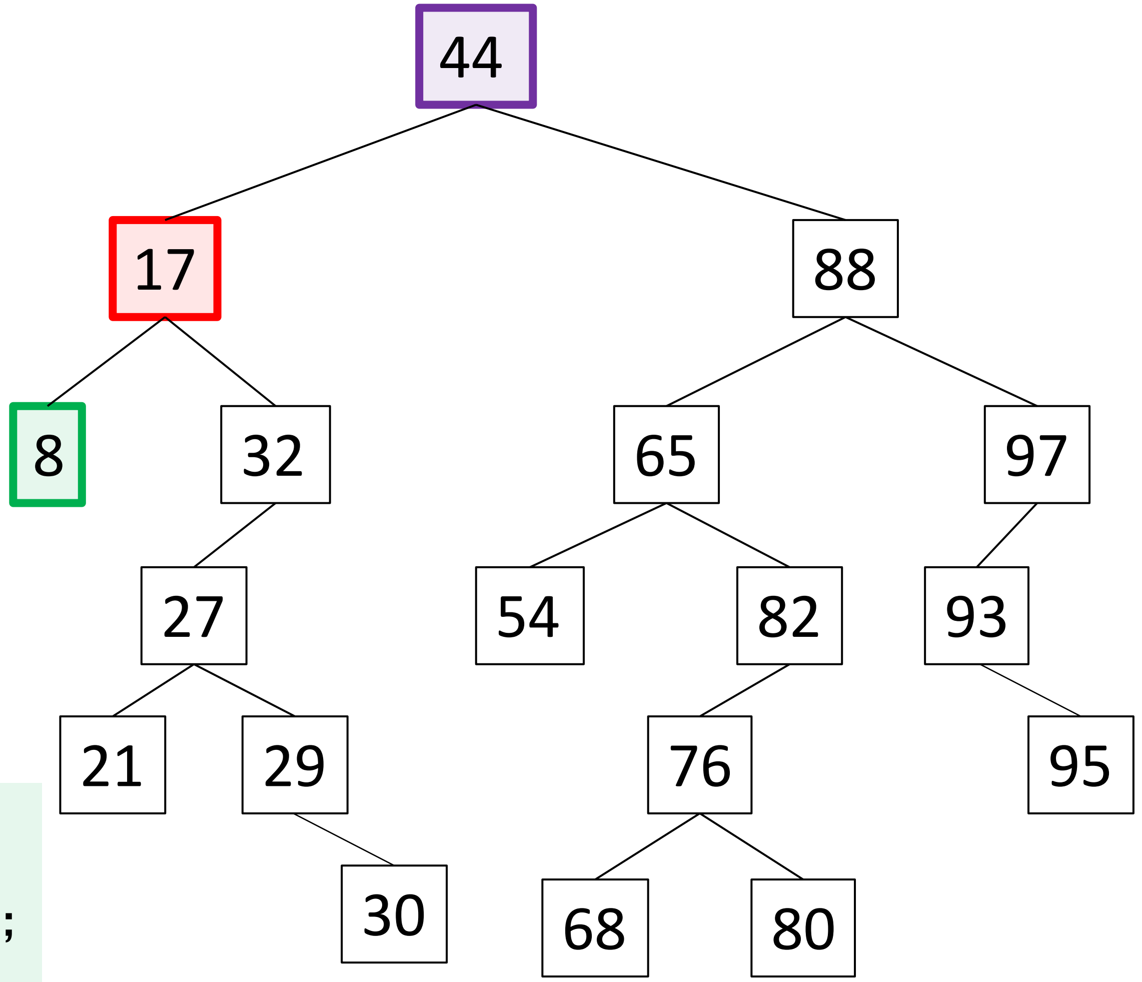


Binary Search Tree - Traversal

```
public void depthFirst(44) {
    if (n != null) {
        System.out.println(n.getValue());
        depthFirst(n.getLeft());
        depthFirst(n.getRight());
    }
}
```

```
public void depthFirst(17) {
    if (n != null) {
        System.out.println(n.getValue());
        depthFirst(n.getLeft());
        depthFirst(n.getRight());
    }
}
```

```
public void depthFirst(8) {
    if (n != null) {
        System.out.println(n.getValue());
        depthFirst(n.getLeft());
        depthFirst(n.getRight());
    }
}
```

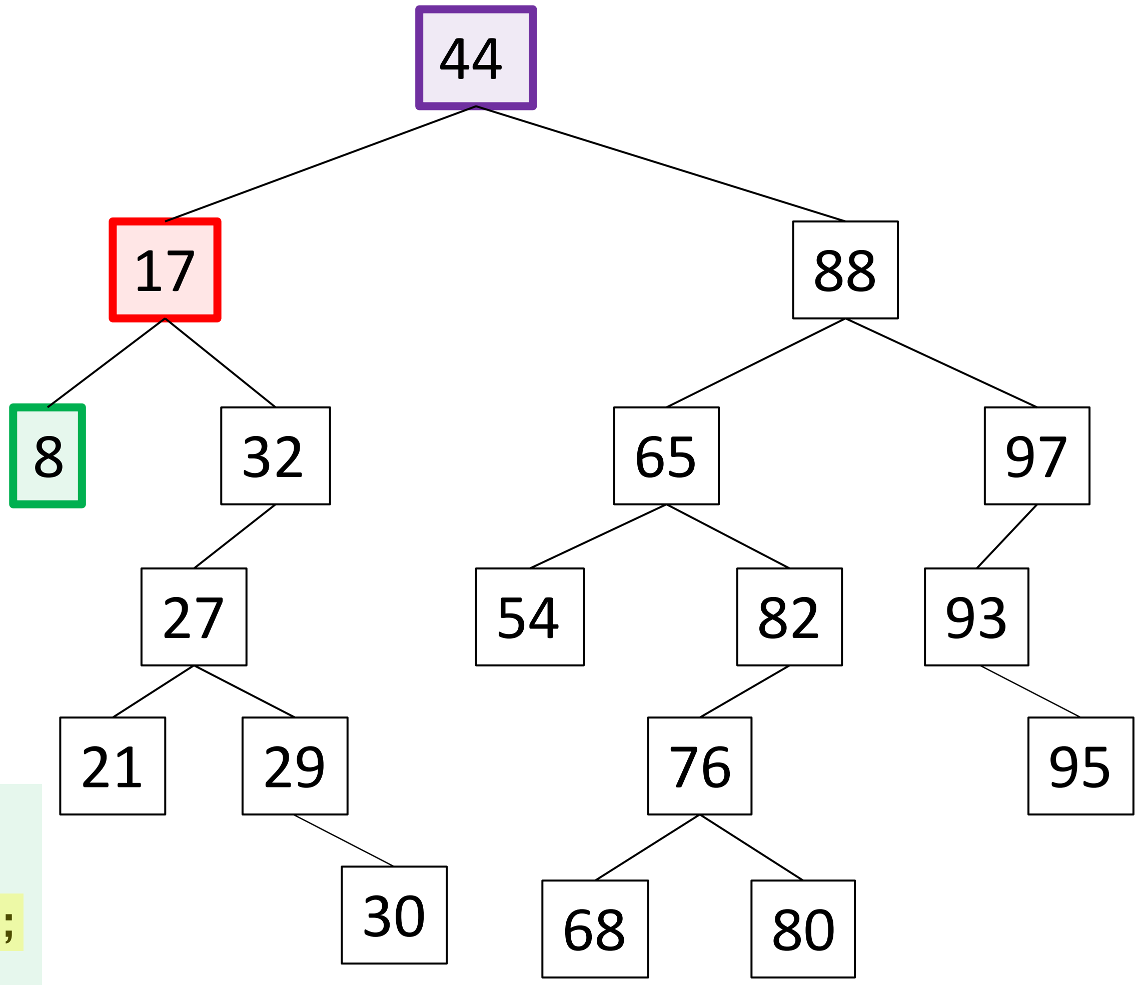


Binary Search Tree - Traversal

```
public void depthFirst(44) {
    if (n != null) {
        System.out.println(n.getValue());
        depthFirst(n.getLeft());
        depthFirst(n.getRight());
    }
}
```

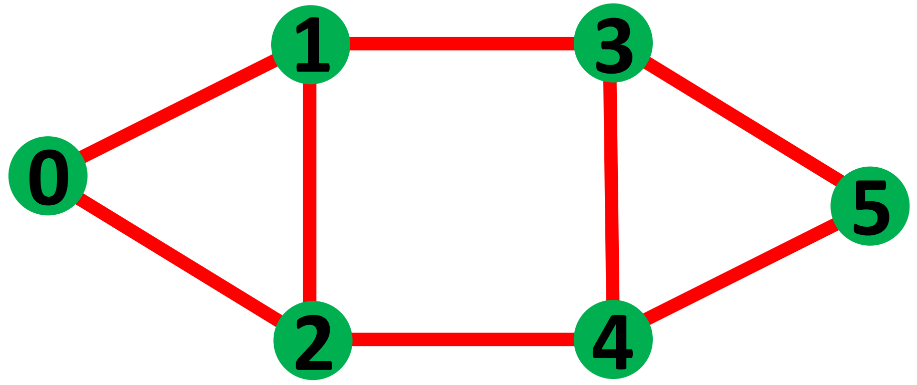
```
public void depthFirst(17) {
    if (n != null) {
        System.out.println(n.getValue());
        depthFirst(n.getLeft());
        depthFirst(n.getRight());
    }
}
```

```
public void depthFirst(8) {
    if (n != null) {
        System.out.println(n.getValue());
        depthFirst(n.getLeft());
        depthFirst(n.getRight());
    }
}
```



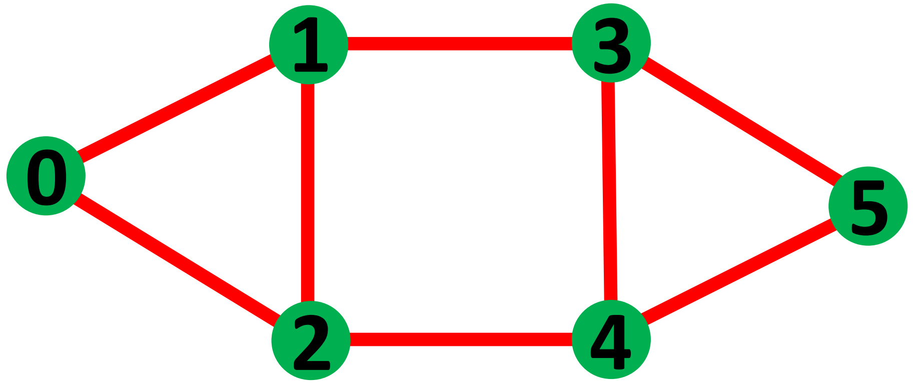
Graphs - Traversal

```
public void depthFirst(Node n) {
    if (n != null) {
        System.out.println(n.getValue());
        depthFirst(n.getLeft());
        depthFirst(n.getRight());
    }
}
```



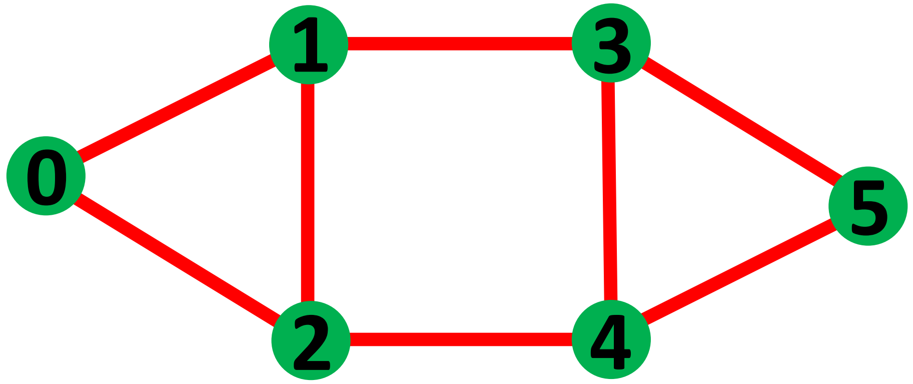
Graphs - Traversal

```
public void depthFirst(~~Node~~ n) {
    if (n != null) {
        System.out.println(n.getValue());
        depthFirst(n.getLeft());
        depthFirst(n.getRight());
    }
}
```



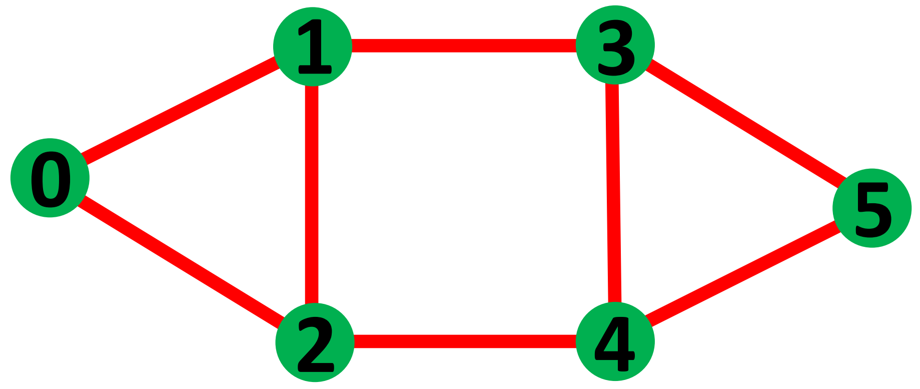
Graphs - Traversal

```
                             int
public void depthFirst(~~Node~~ n) {
    if (n != null) {
        System.out.println(n.getValue());
        depthFirst(n.getLeft());
        depthFirst(n.getRight());
    }
}
```



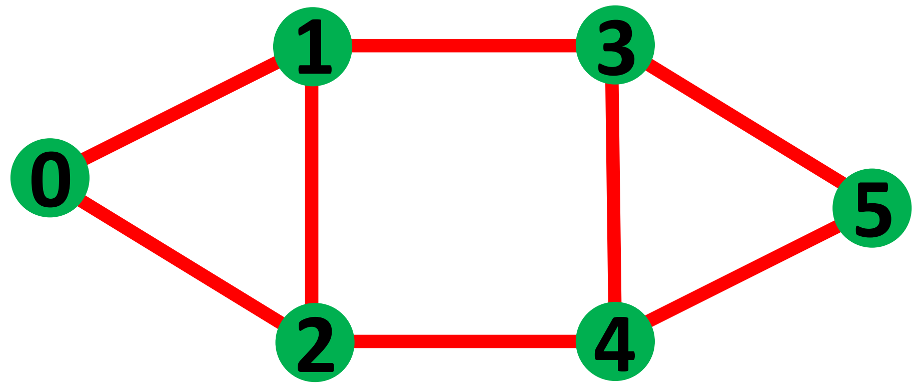
Graphs - Traversal

```
                                int
public void depthFirst(~~Node~~ n) {
    ~~if (n != null) {~~
        System.out.println(n.getValue());
        depthFirst(n.getLeft());
        depthFirst(n.getRight());
    ~~}~~
}
```



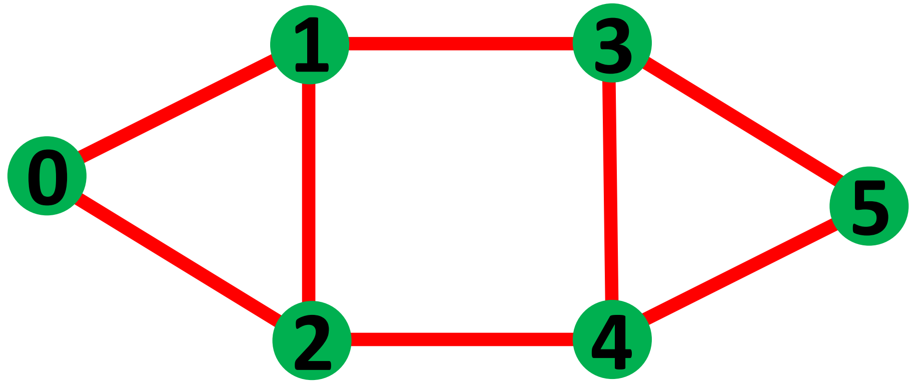
Graphs - Traversal

```
                          int
public void depthFirst(~~Node~~ n) {
    ~~if (n != null) {~~
        System.out.println(n~~.getValue()~~);
        depthFirst(n.getLeft());
        depthFirst(n.getRight());
    ~~}~~
}
```



Graphs - Traversal

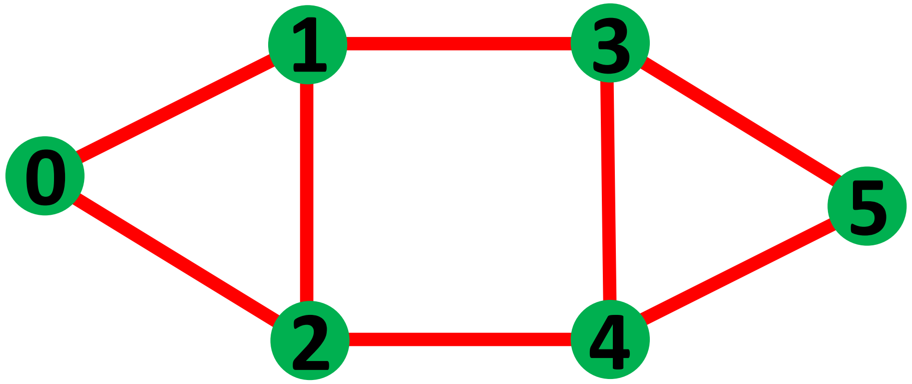
```
                             int
public void depthFirst(~~Node~~ n) {
    ~~if (n != null) {~~
        System.out.println(n~~.getValue()~~);
        ~~depthFirst(n.getLeft());~~
        ~~depthFirst(n.getRight());~~
    ~~}~~
}
```



Graphs - Traversal

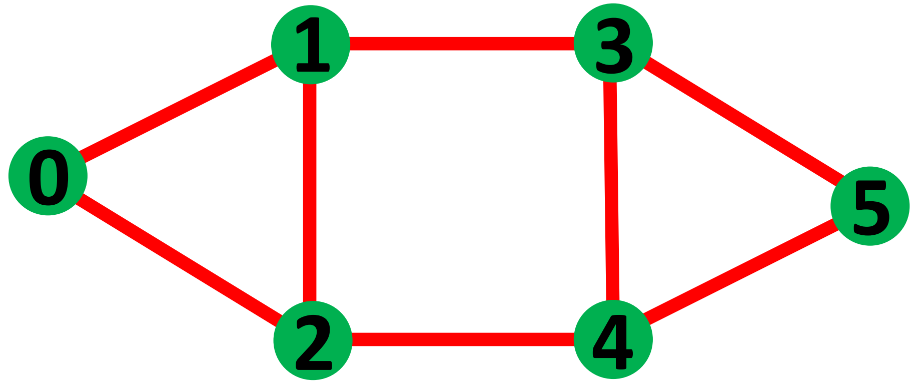
```
                               int
public void depthFirst(~~Node~~ n) {
    ~~if (n != null) {~~
        System.out.println(n~~.getValue()~~);
        ~~depthFirst(n.getLeft());~~
        ~~depthFirst(n.getRight());~~
    ~~}~~
}
```

```
  for (int neighbor : getNeighbors(n)) {
     depthFirst(neighbor);
  }
```



Graphs - Traversal

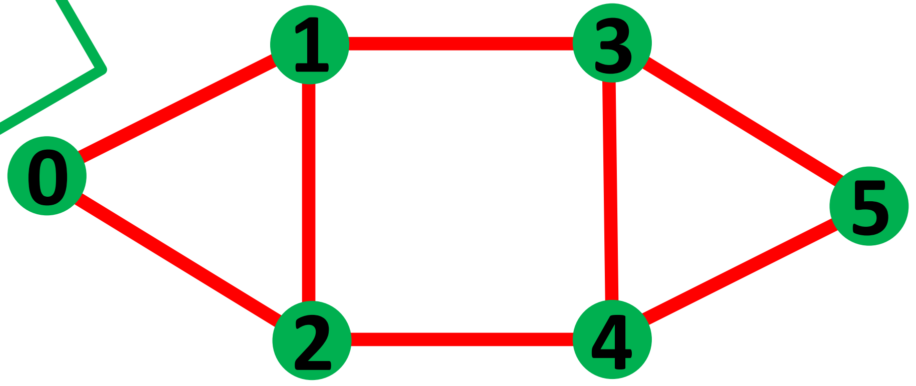
```
public void depthFirst(int n) {
    System.out.println(n);
    for (int neighbor : getNeighbors(n)) {
        depthFirst(neighbor);
    }
}
```



Graphs - Traversal

```
public void depthFirst(int n) {
    System.out.println(n);
    for (int neighbor : getNeighbors(n)
        depthFirst(neighbor);
    }
}
```

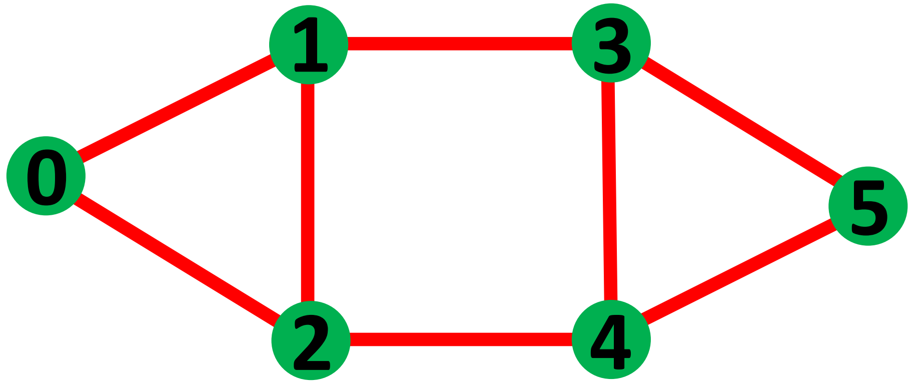
CODE



Graphs - Traversal

```
public void depthFirst(int n) {
    System.out.println(n);
    for (int neighbor : getNeighbors(n)) {
        depthFirst(neighbor);
    }
}
```

Output

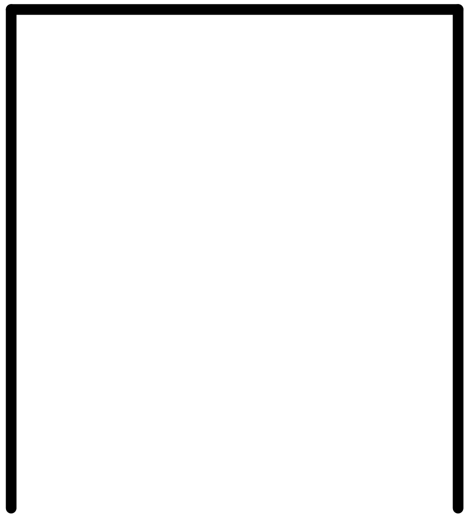


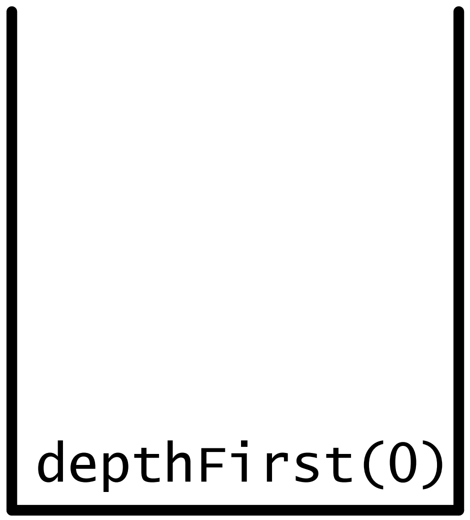
Run-time Stack

Graphs - Traversal

```
public void depthFirst(int n) {
    System.out.println(n);
    for (int neighbor : getNeighbors(n)) {
        depthFirst(neighbor);
    }
}
```

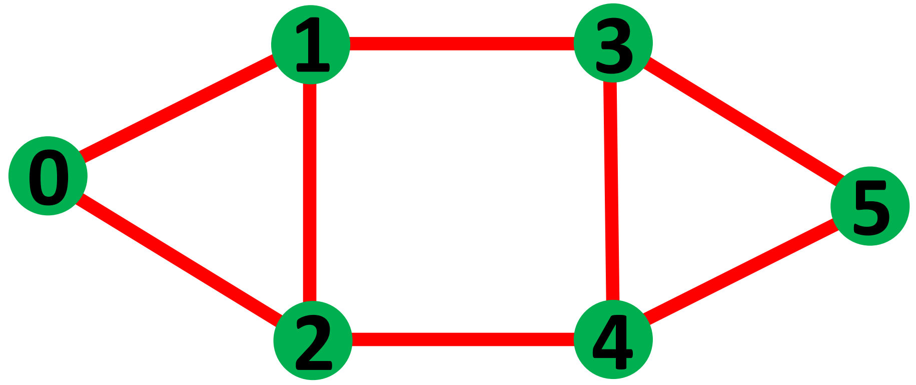
Output





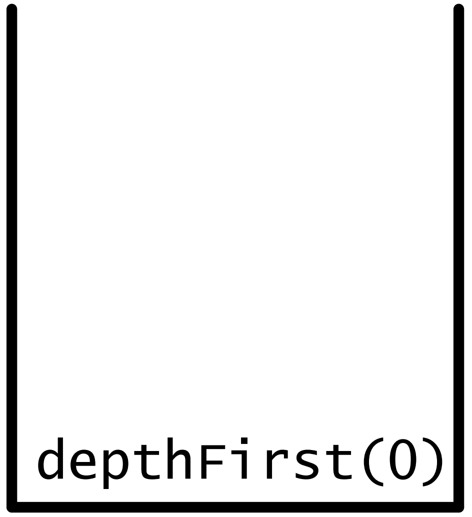
```
depthFirst(0)
```

Run-time Stack



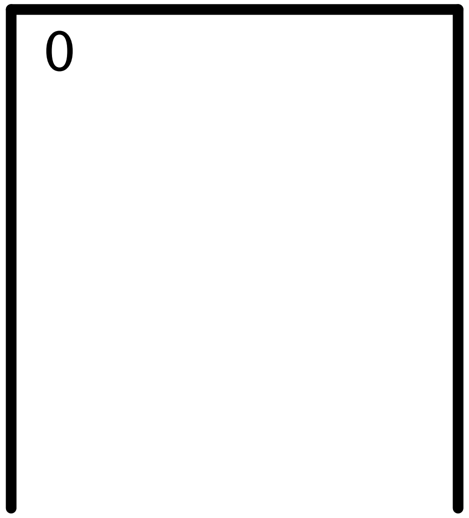
Graphs - Traversal

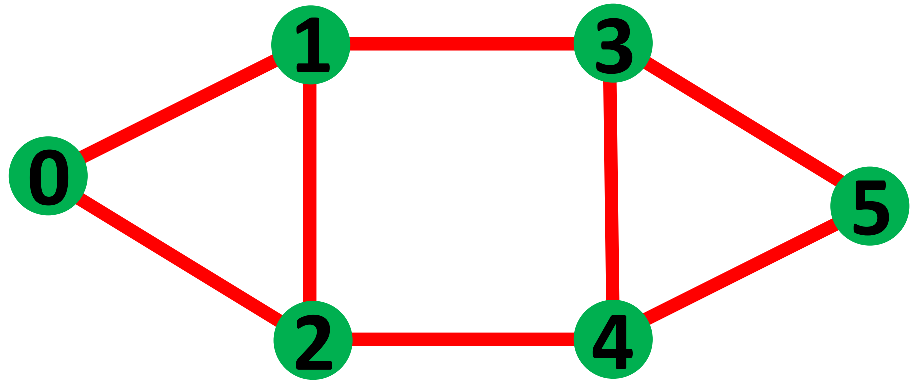
```
public void depthFirst(int n) {
    System.out.println(n);
    for (int neighbor : getNeighbors(n)) {
        depthFirst(neighbor);
    }
}
```



Run-time Stack

Output





Graphs - Traversal

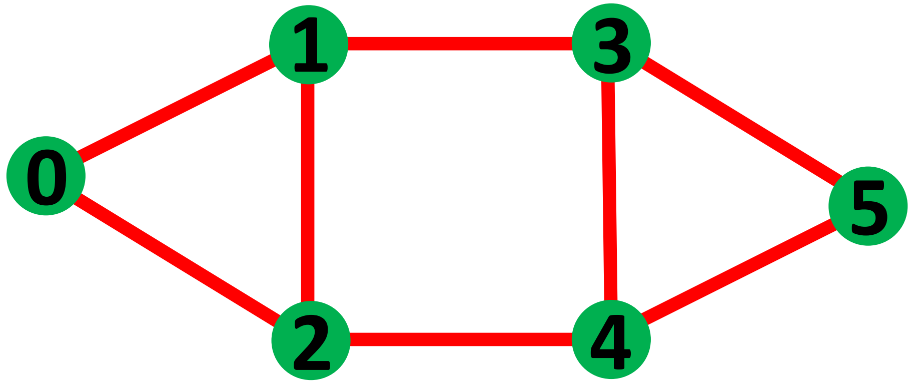
```
public void depthFirst(int n) {
    System.out.println(n);
    for (int neighbor : getNeighbors(n)) {
        depthFirst(neighbor);
    }
}
```

Output

```
0
```

```
depthFirst(1)
depthFirst(0)
```

Run-time Stack



Graphs - Traversal

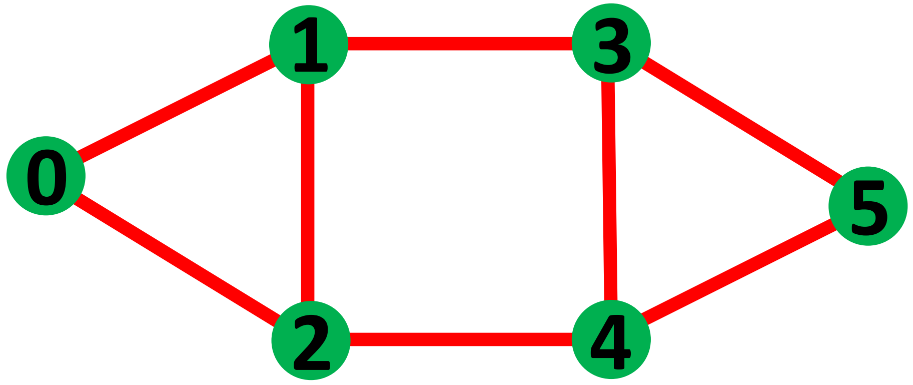
```
public void depthFirst(int n) {
    System.out.println(n);
    for (int neighbor : getNeighbors(n)) {
        depthFirst(neighbor);
    }
}
```

Output

```
0
1
```

```
depthFirst(1)
depthFirst(0)
```

Run-time Stack



Graphs - Traversal

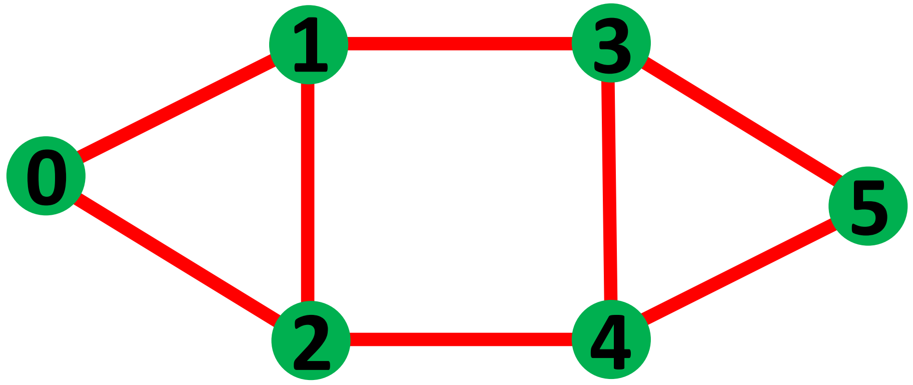
```
public void depthFirst(int n) {
    System.out.println(n);
    for (int neighbor : getNeighbors(n)) {
        depthFirst(neighbor);
    }
}
```

Output

```
0
1
```

```
depthFirst(0)
depthFirst(1)
depthFirst(0)
```

Run-time Stack



Graphs - Traversal

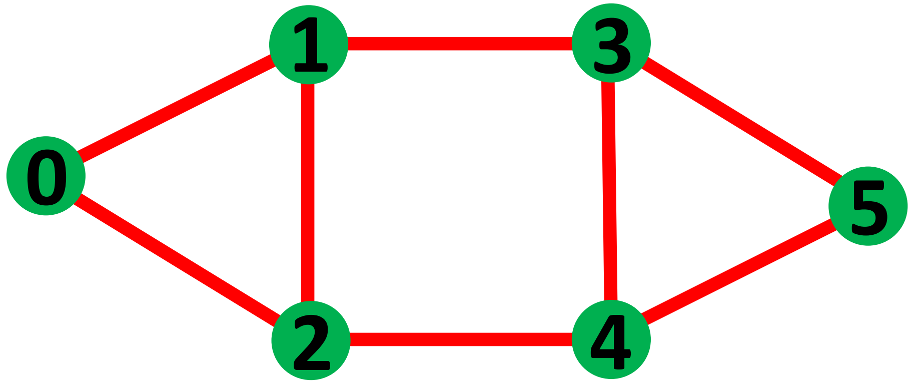
```
public void depthFirst(int n) {
    System.out.println(n);
    for (int neighbor : getNeighbors(n)) {
        depthFirst(neighbor);
    }
}
```

Output

```
0
1
0
```

```
depthFirst(0)
depthFirst(1)
depthFirst(0)
```

Run-time Stack



Graphs - Traversal

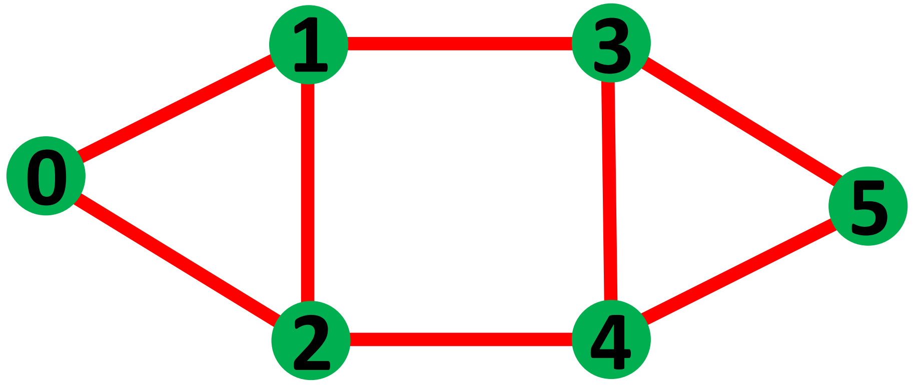
```
public void depthFirst(int n) {
    System.out.println(n);
    for (int neighbor : getNeighbors(n)) {
        depthFirst(neighbor);
    }
}
```

Output

```
0
1
0
```

```
depthFirst(1)
depthFirst(0)
depthFirst(1)
depthFirst(0)
```

Run-time Stack



Graphs - Traversal

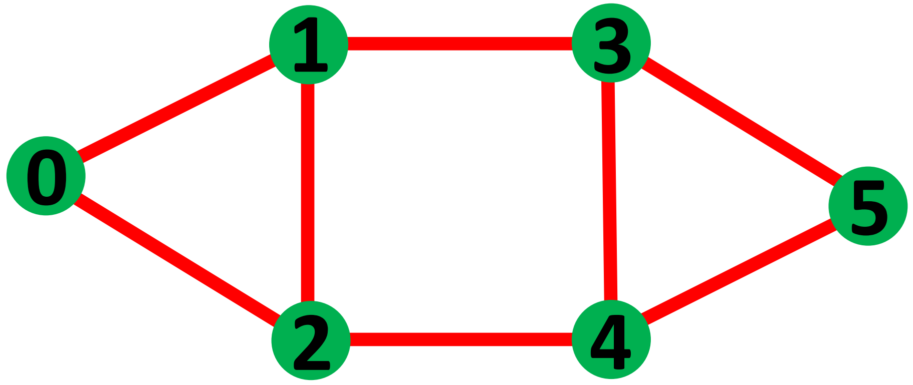
```
public void depthFirst(int n) {
    System.out.println(n);
    for (int neighbor : getNeighbors(n)) {
        depthFirst(neighbor);
    }
}
```

Output

```
0
1
0
1
```

```
depthFirst(1)
depthFirst(0)
depthFirst(1)
depthFirst(0)
```

Run-time Stack



Graphs - Traversal

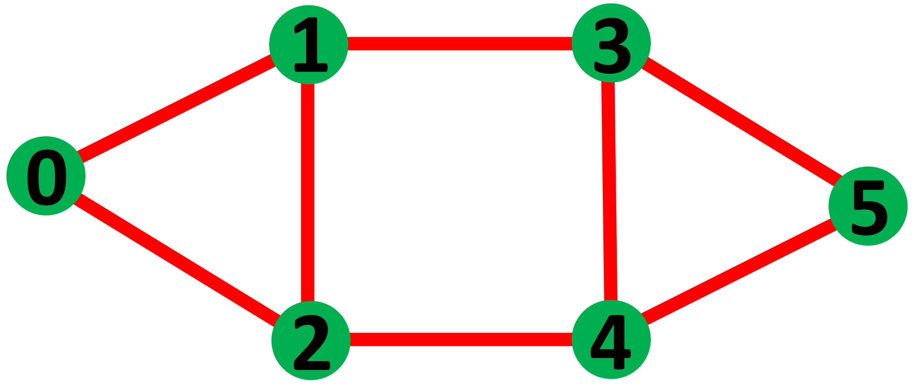
```
public void depthFirst(int n) {
    System.out.println(n);
    for (int neighbor : getNeighbors(n)) {
        depthFirst(neighbor);
    }
}
```

Output

```
0
1
0
1
0
```

```
depthFirst(0)
depthFirst(1)
depthFirst(0)
depthFirst(1)
depthFirst(0)
```

Run-time Stack



Graphs - Traversal

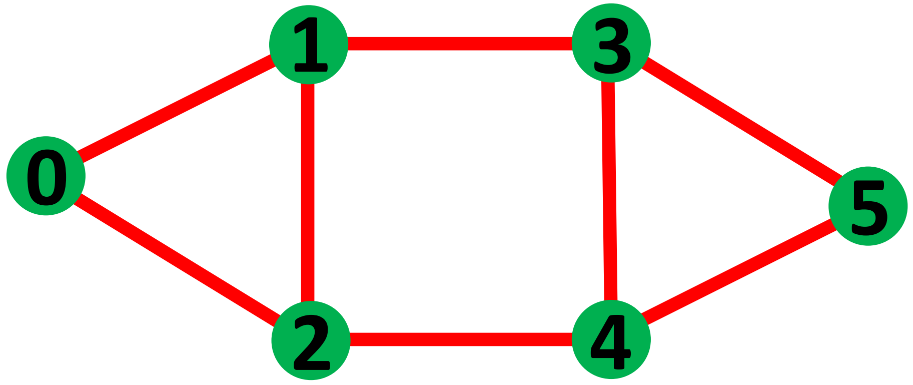
```
public void depthFirst(int n) {
    System.out.println(n);
    for (int neighbor : getNeighbors(n)) {
        depthFirst(neighbor);
    }
}
```

Output

```
0
1
0
1
0
1
```

```
depthFirst(1)
depthFirst(0)
depthFirst(1)
depthFirst(0)
depthFirst(1)
depthFirst(0)
```

Run-time Stack



Graphs - Traversal

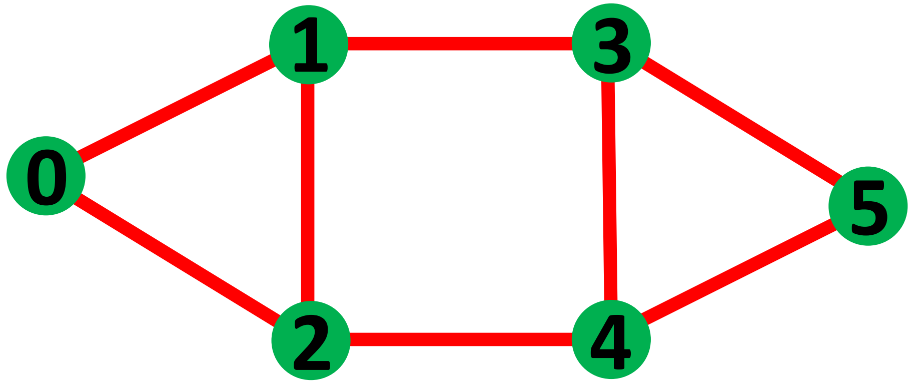
```
public void depthFirst(int n) {
    System.out.println(n);
    for (int neighbor : getNeighbors(n)) {
        depthFirst(neighbor);
    }
}
```

Output

```
0
1
0
1
0
1
0
```

```
depthFirst(0)
depthFirst(1)
depthFirst(0)
depthFirst(1)
depthFirst(0)
depthFirst(1)
depthFirst(0)
```

Run-time Stack



Graphs - Traversal

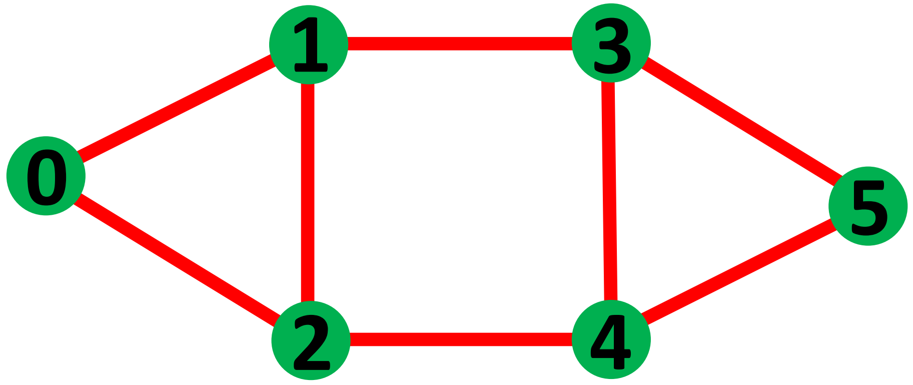
```
public void depthFirst(int n) {
    System.out.println(n);
    for (int neighbor : getNeighbors(n)) {
        depthFirst(neighbor);
    }
}
```

```
depthFirst(1)
depthFirst(0)
depthFirst(1)
depthFirst(0)
depthFirst(1)
depthFirst(0)
depthFirst(1)
depthFirst(0)
```

Run-time Stack

Output

```
0
1
0
1
0
1
0
1
```



Graphs - Traversal

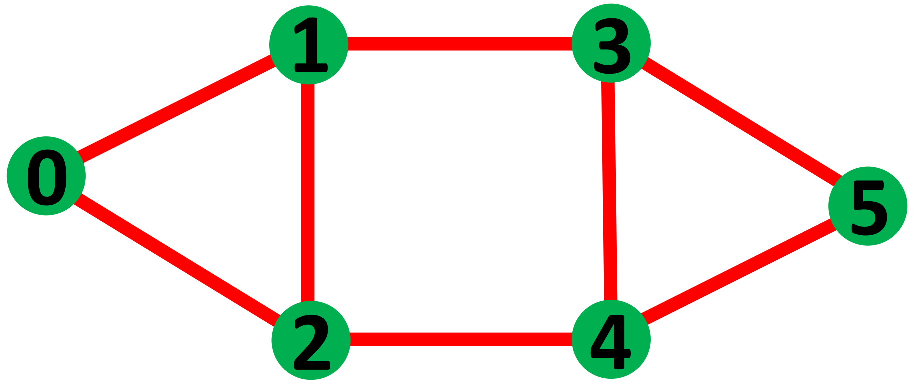
```
public void depthFirst(int n) {
    System.out.println(n);
    for (int neighbor : getNeighbors(n)) {
        depthFirst(neighbor);
    }
}
```

```
depthFirst(0)
depthFirst(1)
depthFirst(0)
depthFirst(1)
depthFirst(0)
depthFirst(1)
depthFirst(0)
depthFirst(1)
depthFirst(0)
```

Run-time Stack

Output

```
0
1
0
1
0
1
0
1
0
```



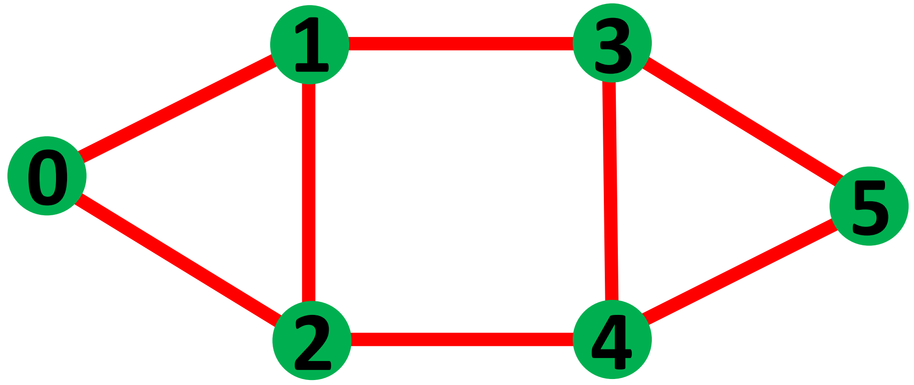
Graphs - Traversal

```
public void depthFirst(int n) {
    System.out.println(n);
    for (int neighbor : getNeighbors(n)) {
        depthFirst(neighbor);
    }
}
```

```
depthFirst(0)
depthFirst(1)
depthFirst(0)
depthFirst(1)
depthFirst(0)
depthFirst(1)
depthFirst(0)
depthFirst(1)
depthFirst(0)
```

Run-time Stack

STACK OVERFLOW ERROR



Graphs - Traversal

Where is the problem?

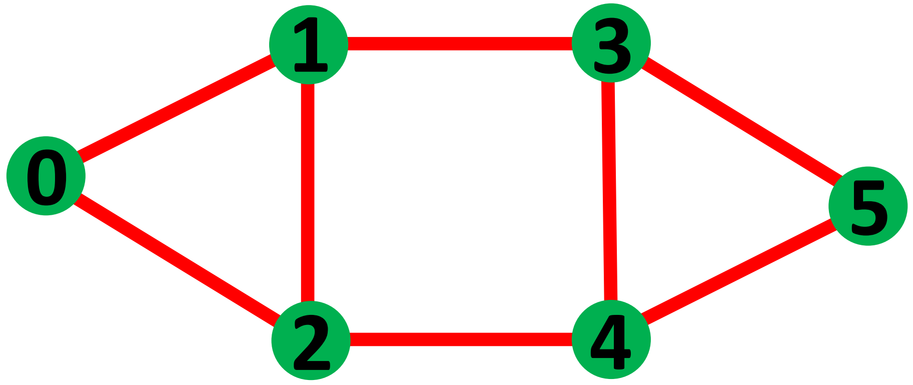
```
public void depthFirst(int n) {
    System.out.println(n);
    for (int neighbor : getNeighbors(n)) {
        depthFirst(neighbor);
    }
}
```

Output

```
0
1
```

```
depthFirst(1)
depthFirst(0)
```

Run-time Stack



Graphs - Traversal

What neighbors should we call `depthFirst()` on?

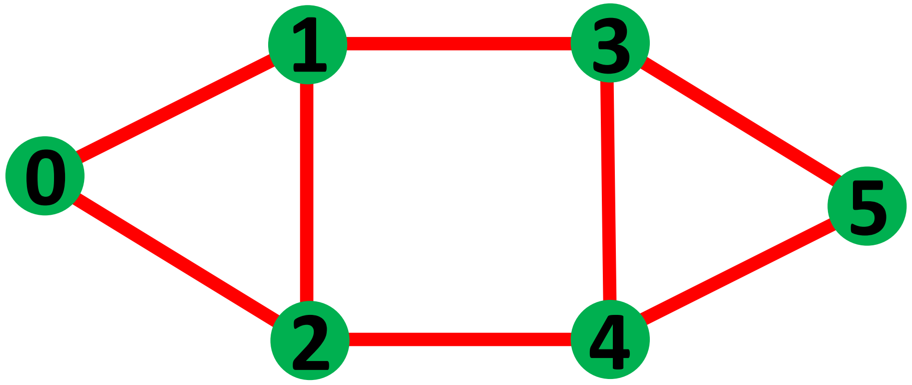
```
public void depthFirst(int n) {
    System.out.println(n);
    for (int neighbor : getNeighbors(n)) {
        depthFirst(neighbor);
    }
}
```

Output

```
0
1
```

```
depthFirst(1)
depthFirst(0)
```

Run-time Stack



Graphs - Traversal

What neighbors should we call `depthFirst()` on?

Neighbors that have not already been visited.

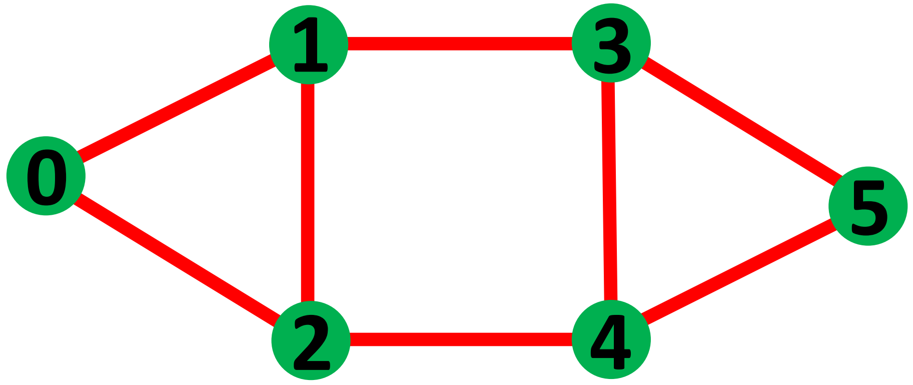
```
public void depthFirst(int n) {
    System.out.println(n);
    for (int neighbor : getNeighbors(n)) {
        depthFirst(neighbor);
    }
}
```

Output

```
0
1
```

```
depthFirst(1)
depthFirst(0)
```

Run-time Stack



Graphs - Traversal

```
public void depthFirst(int n) {
    System.out.println(n);
    for (int neighbor : getNeighbors(n)) {
        depthFirst(neighbor);
    }
}
```

What neighbors should we call `depthFirst()` on?

Neighbors that have not already been visited.

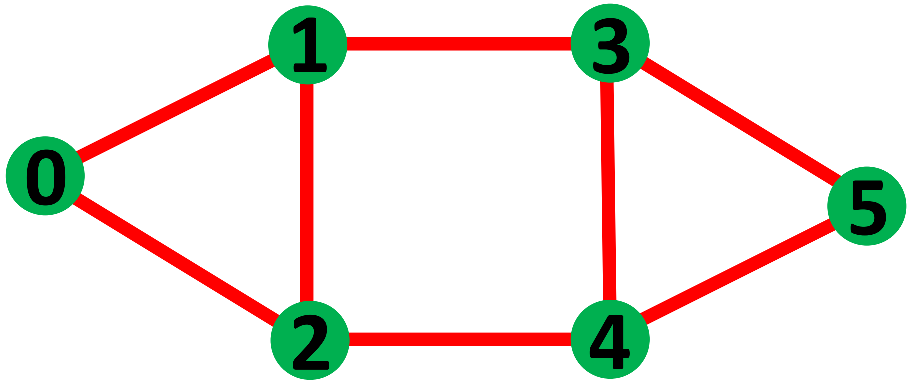
How can we do that?

Output

```
0
1
```

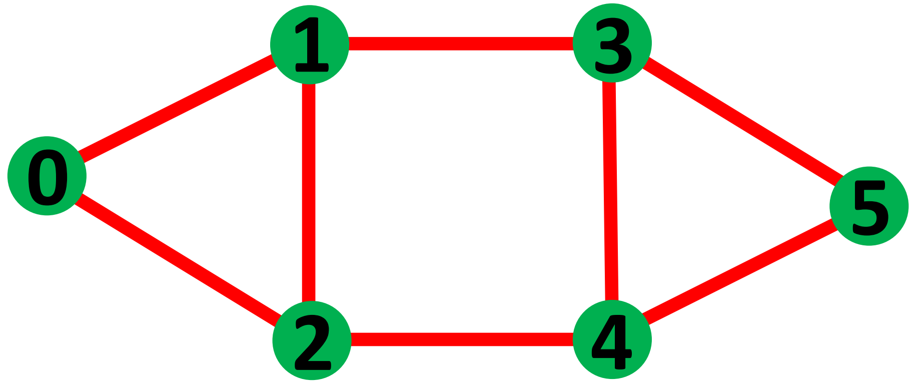
```
depthFirst(1)
depthFirst(0)
```

Run-time Stack



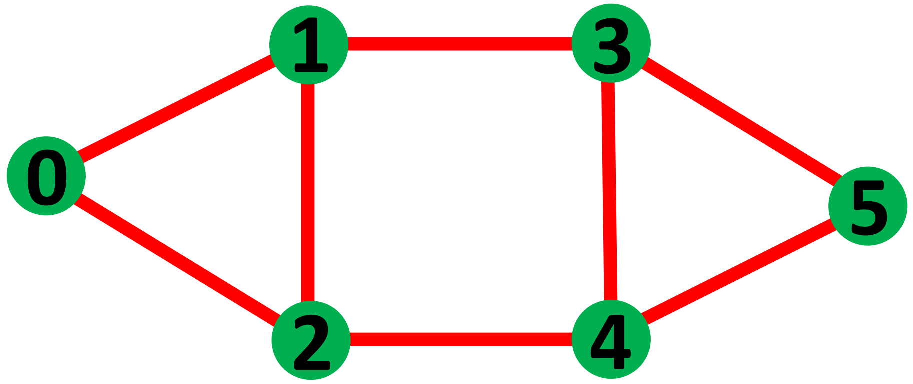
Graphs - Traversal

```
public void depthFirst(int n) {
    System.out.println(n);
    for (int neighbor : getNeighbors(n)) {
        depthFirst(neighbor);
    }
}
```



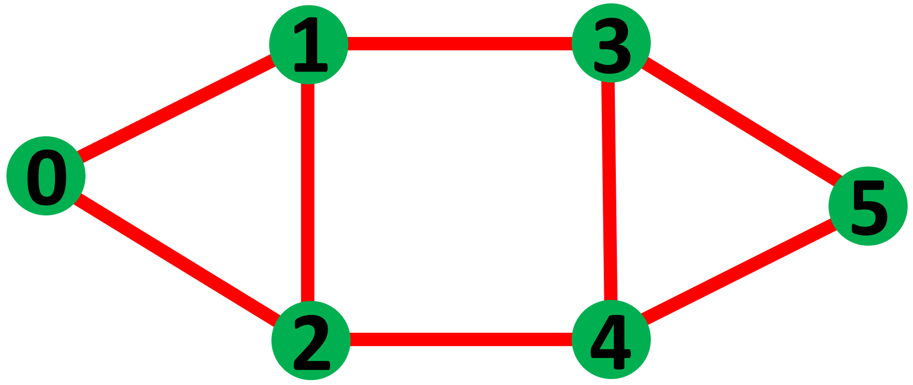
Graphs - Traversal

```
private boolean visited[] = new boolean[getNumVertices()];
public void depthFirst(int n) {
    System.out.println(n);
    for (int neighbor : getNeighbors(n)) {
        depthFirst(neighbor);
    }
}
```



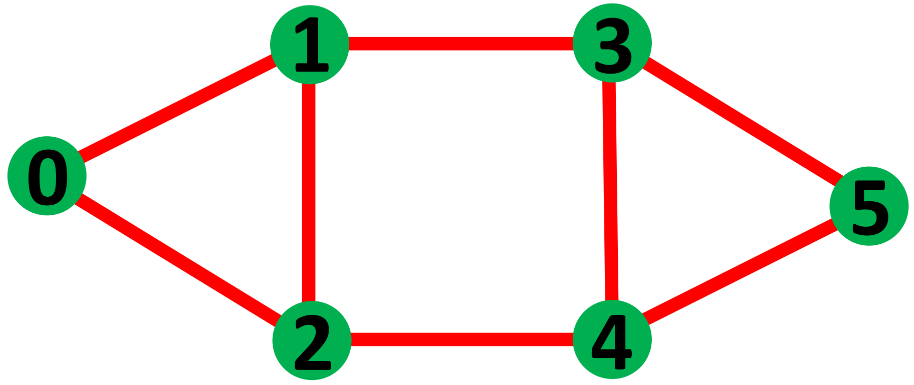
Graphs - Traversal

```
private boolean visited[] = new boolean[getNumVertices()];
public void depthFirst(int n) {
    System.out.println(n);
    for (int neighbor : getNeighbors(n)) {
        if (!visited[neighbor]) {
            depthFirst(neighbor);
        }
    }
}
```



Graphs - Traversal

```
private boolean visited[] = new boolean[getNumVertices()];
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for (int neighbor : getNeighbors(n)) {
        if (!visited[neighbor]) {
            depthFirst(neighbor);
        }
    }
}
```



Graphs - Traversal

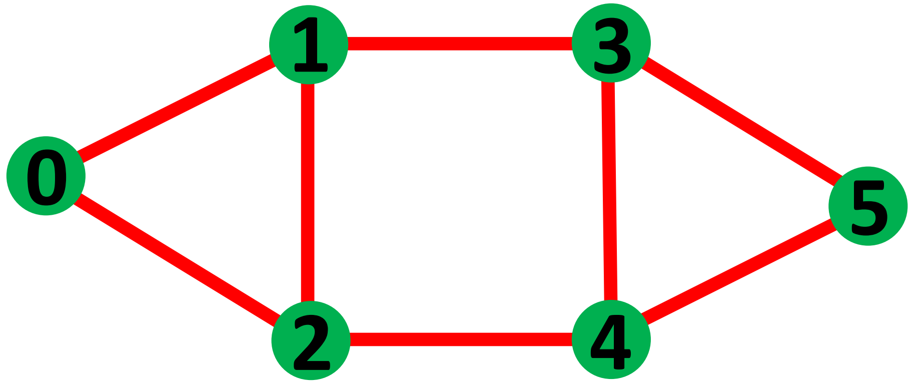
```
private boolean visited[] = new boolean[getNumVertices()];
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for (int neighbor : getNeighbors(n)) {
        if (!visited[neighbor]) {
            depthFirst(neighbor);
        }
    }
}
```

```
depthFirst(1)
depthFirst(0)
```

Run-time Stack

Output

```
0
1
```



Graphs - Traversal

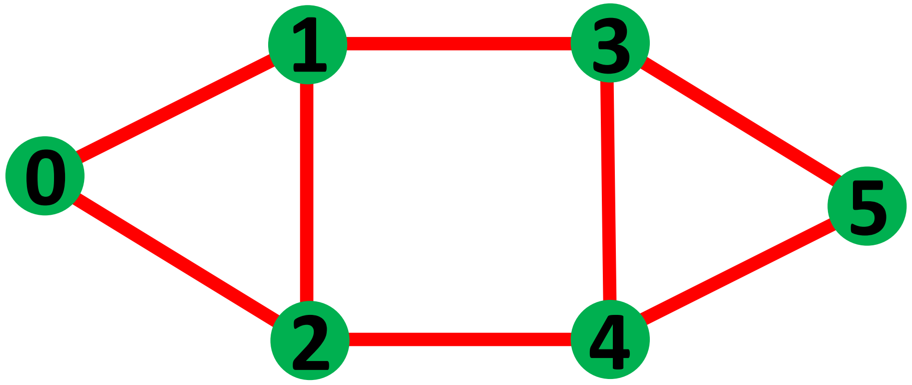
```
private boolean visited[] = new boolean[getNumVertices()];
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for (int neighbor : getNeighbors(n)) {
        if (!visited[neighbor]) {
            depthFirst(neighbor);
        }
    }
}
```

```
depthFirst(2)
depthFirst(1)
depthFirst(0)
```

Run-time Stack

Output

```
0
1
```



Graphs - Traversal

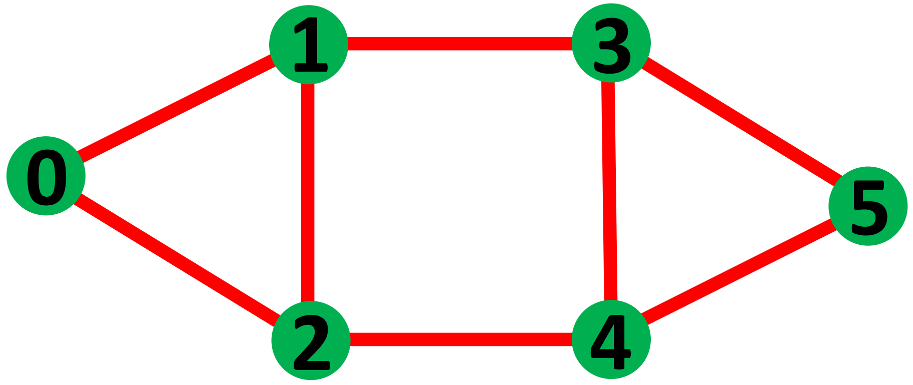
```
private boolean visited[] = new boolean[getNumVertices()];
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for (int neighbor : getNeighbors(n)) {
        if (!visited[neighbor]) {
            depthFirst(neighbor);
        }
    }
}
```

```
depthFirst(2)
depthFirst(1)
depthFirst(0)
```

Run-time Stack

Output

```
0
1
2
```



Graphs - Traversal

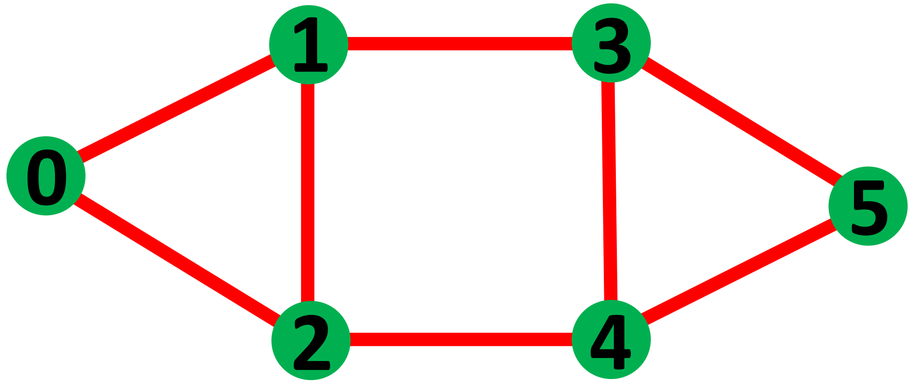
```
private boolean visited[] = new boolean[getNumVertices()];
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for (int neighbor : getNeighbors(n)) {
        if (!visited[neighbor]) {
            depthFirst(neighbor);
        }
    }
}
```

```
depthFirst(4)
depthFirst(2)
depthFirst(1)
depthFirst(0)
```

Run-time Stack

Output

```
0
1
2
```



Graphs - Traversal

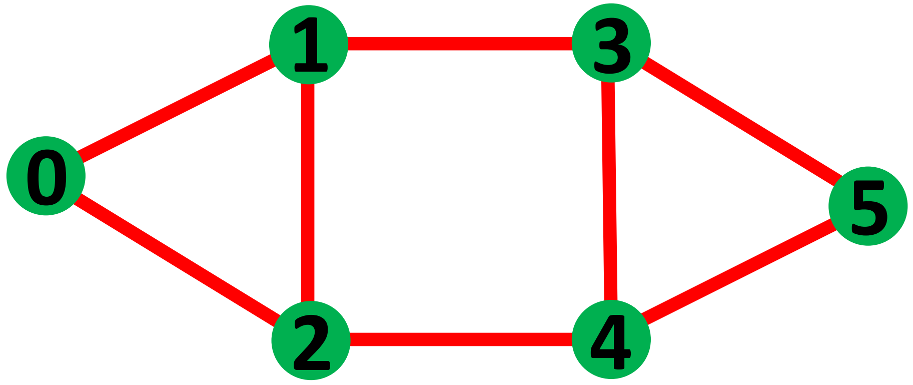
```
private boolean visited[] = new boolean[getNumVertices()];
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for (int neighbor : getNeighbors(n)) {
        if (!visited[neighbor]) {
            depthFirst(neighbor);
        }
    }
}
```

```
depthFirst(4)
depthFirst(2)
depthFirst(1)
depthFirst(0)
```

Run-time Stack

Output

```
0
1
2
4
```



Graphs - Traversal

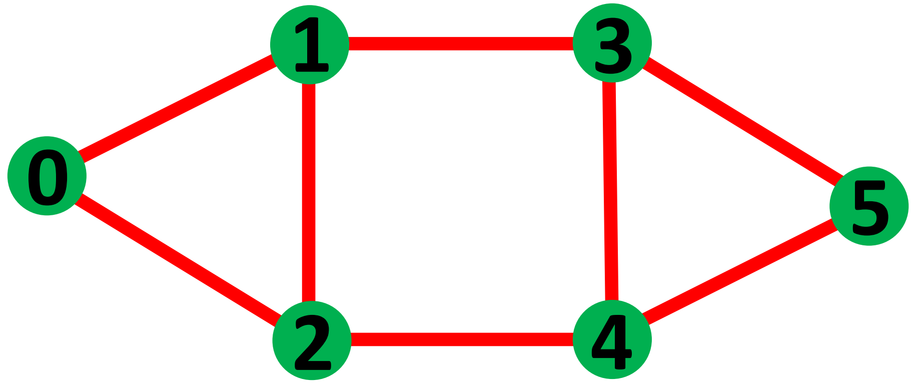
```
private boolean visited[] = new boolean[getNumVertices()];
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for (int neighbor : getNeighbors(n)) {
        if (!visited[neighbor]) {
            depthFirst(neighbor);
        }
    }
}
```

```
depthFirst(3)
depthFirst(4)
depthFirst(2)
depthFirst(1)
depthFirst(0)
```

Run-time Stack

Output

```
0
1
2
4
```



Graphs - Traversal

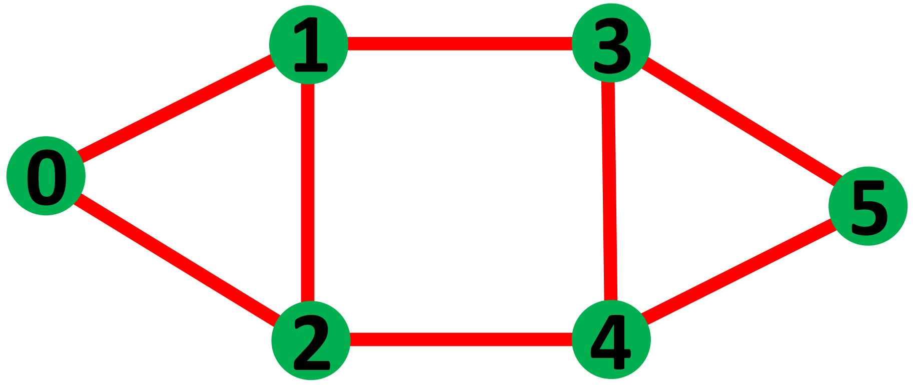
```
private boolean visited[] = new boolean[getNumVertices()];
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for (int neighbor : getNeighbors(n)) {
        if (!visited[neighbor]) {
            depthFirst(neighbor);
        }
    }
}
```

```
depthFirst(3)
depthFirst(4)
depthFirst(2)
depthFirst(1)
depthFirst(0)
```

Run-time Stack

Output

```
0
1
2
4
3
```



Graphs - Traversal

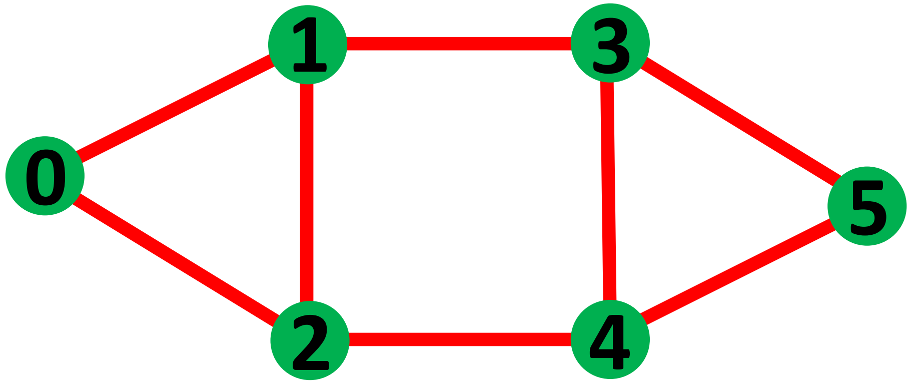
```
private boolean visited[] = new boolean[getNumVertices()];
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for (int neighbor : getNeighbors(n)) {
        if (!visited[neighbor]) {
            depthFirst(neighbor);
        }
    }
}
```

```
depthFirst(5)
depthFirst(3)
depthFirst(4)
depthFirst(2)
depthFirst(1)
depthFirst(0)
```

Run-time Stack

Output

```
0
1
2
4
3
```



Graphs - Traversal

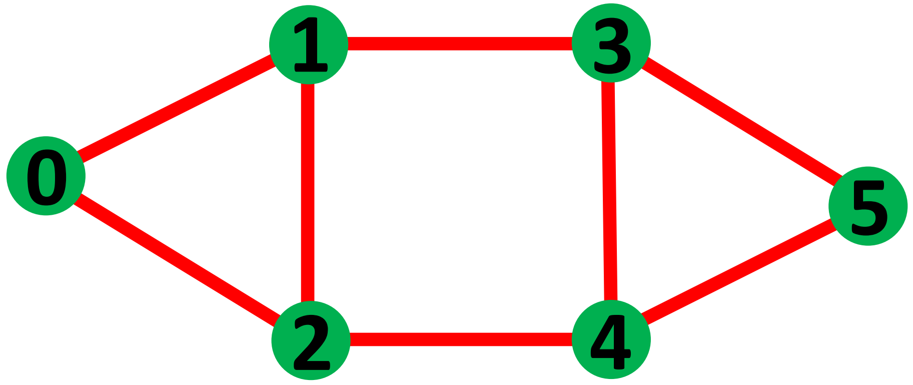
```
private boolean visited[] = new boolean[getNumVertices()];
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for (int neighbor : getNeighbors(n)) {
        if (!visited[neighbor]) {
            depthFirst(neighbor);
        }
    }
}
```

```
depthFirst(5)
depthFirst(3)
depthFirst(4)
depthFirst(2)
depthFirst(1)
depthFirst(0)
```

Run-time Stack

Output

```
0
1
2
4
3
5
```



Graphs - Traversal

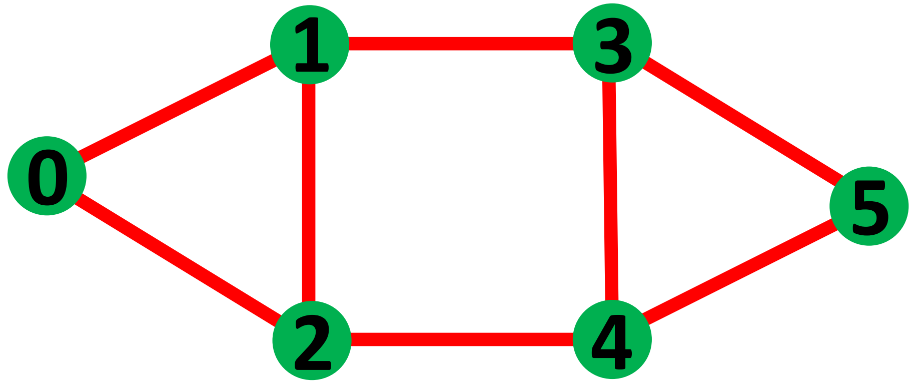
```
private boolean visited[] = new boolean[getNumVertices()];
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for (int neighbor : getNeighbors(n)) {
        if (!visited[neighbor]) {
            depthFirst(neighbor);
        }
    }
}
```

```
depthFirst(3)
depthFirst(4)
depthFirst(2)
depthFirst(1)
depthFirst(0)
```

Run-time Stack

Output

```
0
1
2
4
3
5
```



Graphs - Traversal

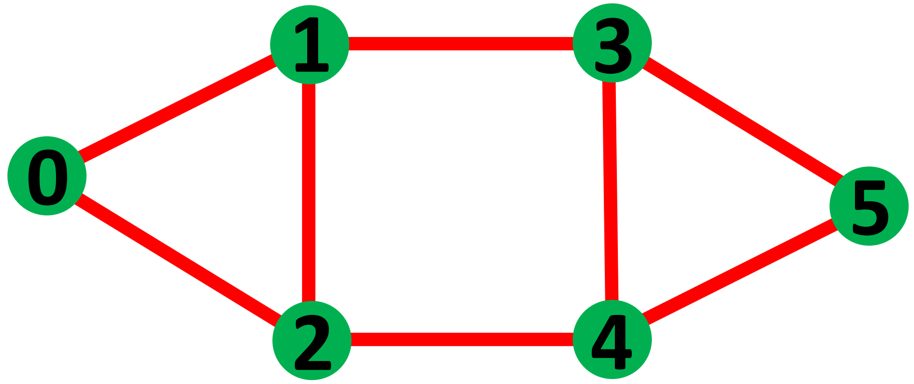
```
private boolean visited[] = new boolean[getNumVertices()];
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for (int neighbor : getNeighbors(n)) {
        if (!visited[neighbor]) {
            depthFirst(neighbor);
        }
    }
}
```

```
depthFirst(4)
depthFirst(2)
depthFirst(1)
depthFirst(0)
```

Run-time Stack

Output

```
0
1
2
4
3
5
```



Graphs - Traversal

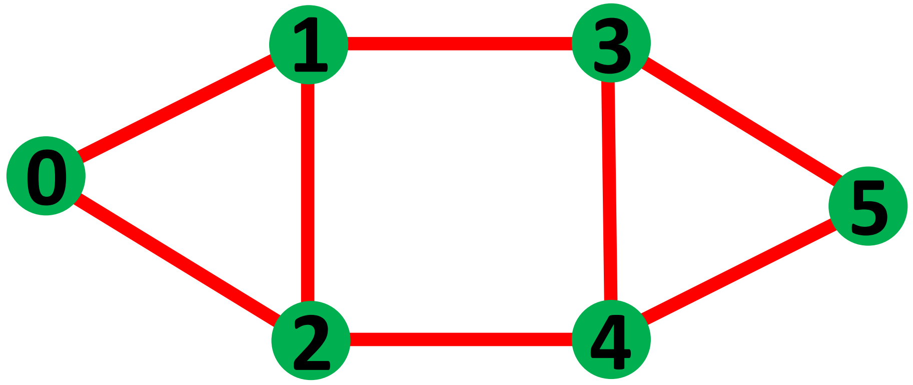
```
private boolean visited[] = new boolean[getNumVertices()];
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for (int neighbor : getNeighbors(n)) {
        if (!visited[neighbor]) {
            depthFirst(neighbor);
        }
    }
}
```

```
depthFirst(2)
depthFirst(1)
depthFirst(0)
```

Run-time Stack

Output

```
0
1
2
4
3
5
```



Graphs - Traversal

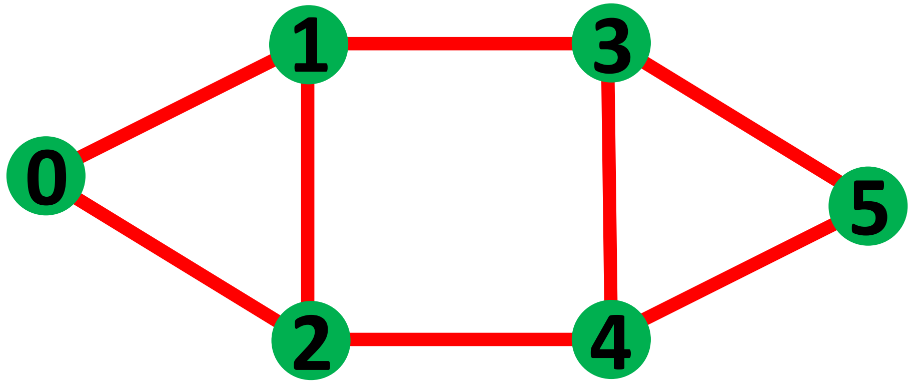
```
private boolean visited[] = new boolean[getNumVertices()];
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for (int neighbor : getNeighbors(n)) {
        if (!visited[neighbor]) {
            depthFirst(neighbor);
        }
    }
}
```

```
depthFirst(1)
depthFirst(0)
```

Run-time Stack

Output

```
0
1
2
4
3
5
```



Graphs - Traversal

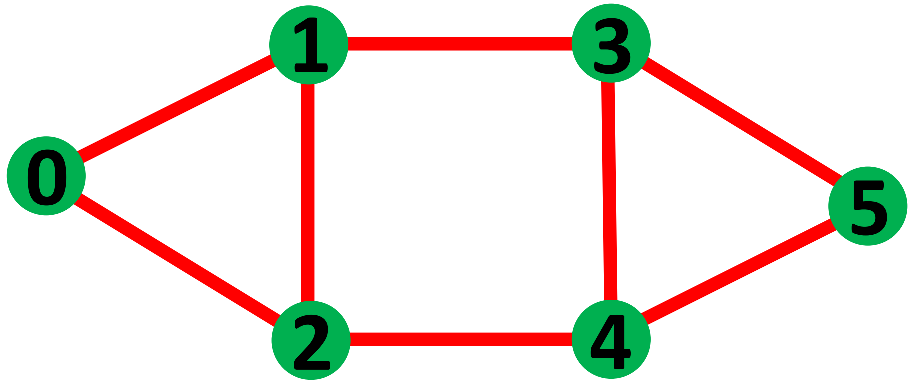
```
private boolean visited[] = new boolean[getNumVertices()];
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for (int neighbor : getNeighbors(n)) {
        if (!visited[neighbor]) {
            depthFirst(neighbor);
        }
    }
}
```

Run-time Stack

```
depthFirst(0)
```

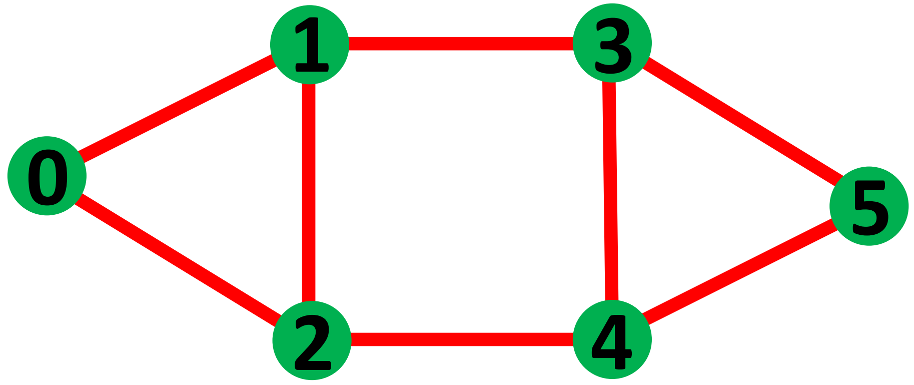
Output

```
0
1
2
4
3
5
```



Graphs - Traversal

```
private boolean visited[] = new boolean[getNumVertices()];
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for (int neighbor : getNeighbors(n)) {
        if (!visited[neighbor]) {
            depthFirst(neighbor);
        }
    }
}
```



Run-time Stack

Output

```
0
1
2
4
3
5
```

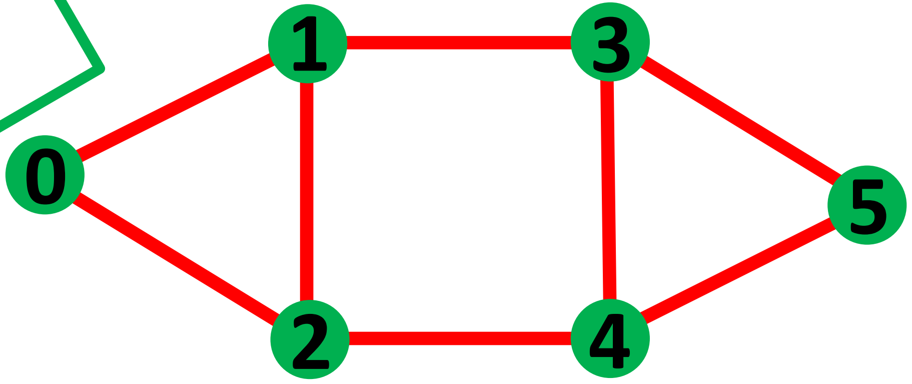
Graphs - Traversal

```
private boolean visited[] = new boolean[getNumVertices()];
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for (int neighbor : getNeighbor
        if (!visited[neighbor])
            depthFirst(neigh
        }
    }
}
```

CODE

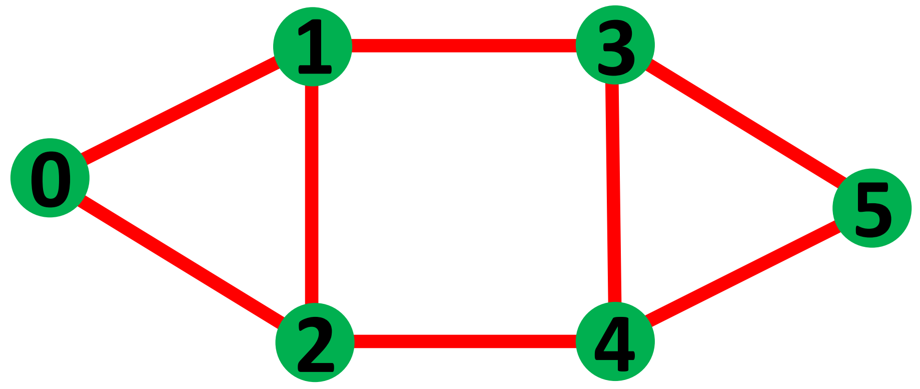
Run-time Stack

```
2
4
3
5
```



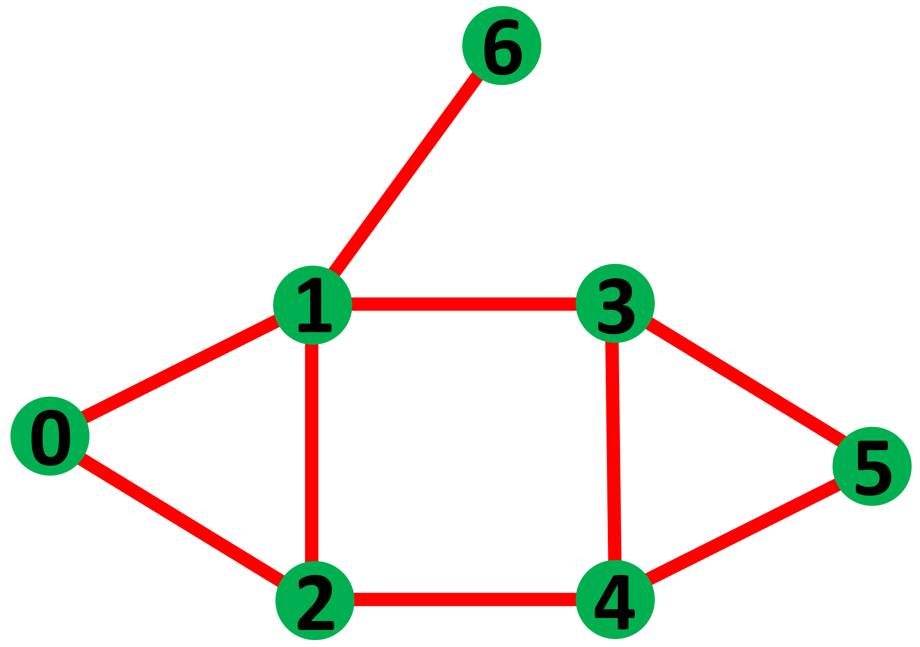
Graphs - Traversal

```
private boolean visited[] = new boolean[getNumVertices()];
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for (int neighbor : getNeighbors(n)) {
        if (!visited[neighbor]) {
            depthFirst(neighbor);
        }
    }
}
```



Graphs - Traversal

```
private boolean visited[] = new boolean[getNumVertices()];
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for (int neighbor : getNeighbors(n)) {
        if (!visited[neighbor]) {
            depthFirst(neighbor);
        }
    }
}
```



Graphs - Traversal

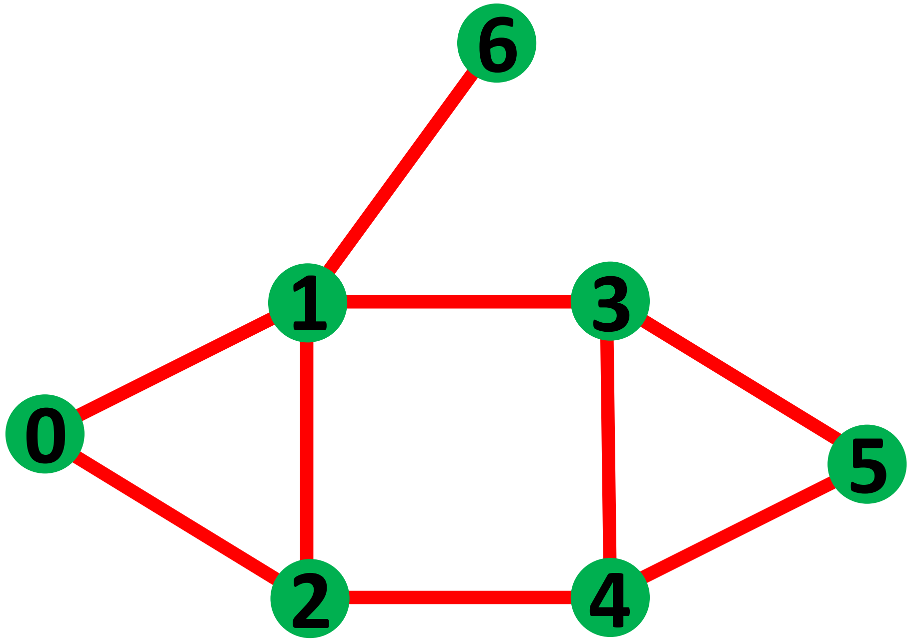
```
private boolean visited[] = new boolean[getNumVertices()];
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for (int neighbor : getNeighbors(n)) {
        if (!visited[neighbor]) {
            depthFirst(neighbor);
        }
    }
}
```

```
depthFirst(5)
depthFirst(3)
depthFirst(4)
depthFirst(2)
depthFirst(1)
depthFirst(0)
```

Run-time Stack

Output

```
0
1
2
4
3
5
```



Graphs - Traversal

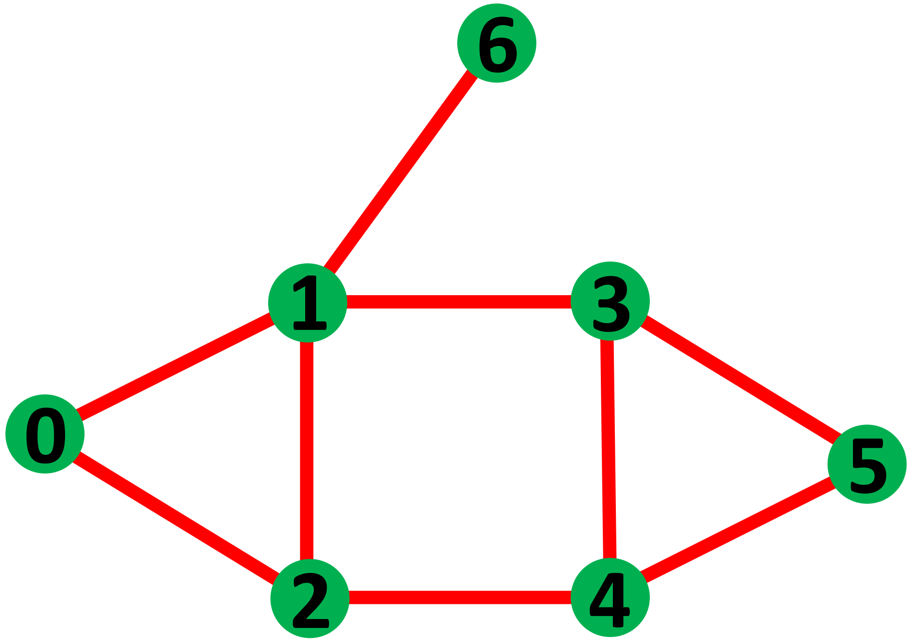
```
private boolean visited[] = new boolean[getNumVertices()];
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for (int neighbor : getNeighbors(n)) {
        if (!visited[neighbor]) {
            depthFirst(neighbor);
        }
    }
}
```

```
depthFirst(3)
depthFirst(4)
depthFirst(2)
depthFirst(1)
depthFirst(0)
```

Run-time Stack

Output

```
0
1
2
4
3
5
```



Graphs - Traversal

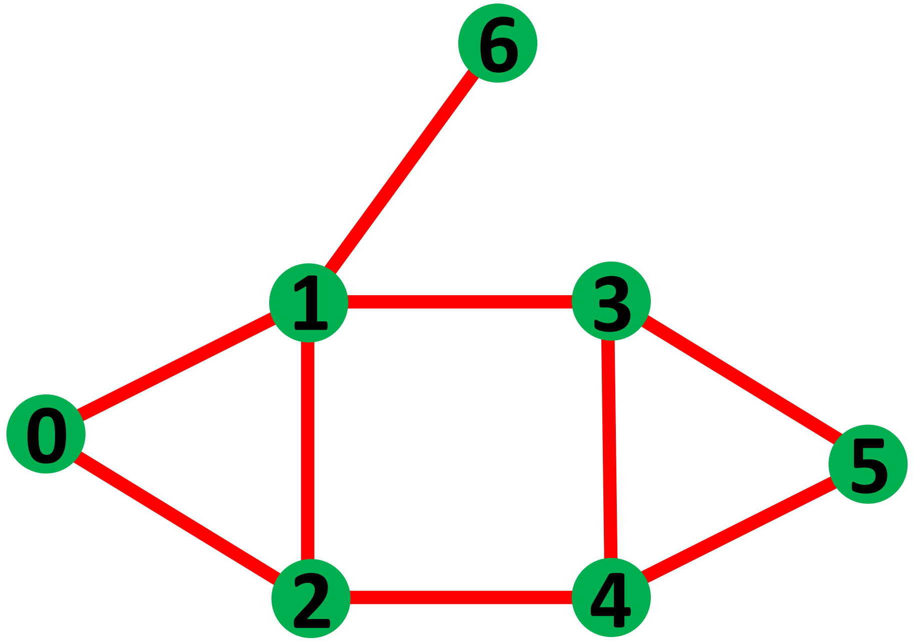
```
private boolean visited[] = new boolean[getNumVertices()];
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for (int neighbor : getNeighbors(n)) {
        if (!visited[neighbor]) {
            depthFirst(neighbor);
        }
    }
}
```

```
depthFirst(4)
depthFirst(2)
depthFirst(1)
depthFirst(0)
```

Run-time Stack

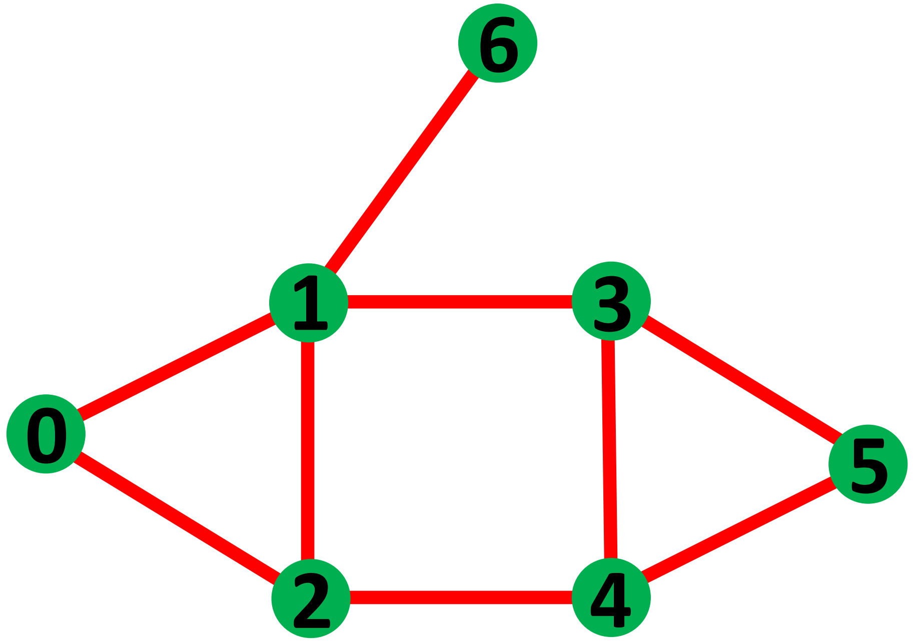
Output

```
0
1
2
4
3
5
```



Graphs - Traversal

```
private boolean visited[] = new boolean[getNumVertices()];
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for (int neighbor : getNeighbors(n)) {
        if (!visited[neighbor]) {
            depthFirst(neighbor);
        }
    }
}
```



```
depthFirst(2)
depthFirst(1)
depthFirst(0)
```

Run-time Stack

Output

```
0
1
2
4
3
5
```

Graphs - Traversal

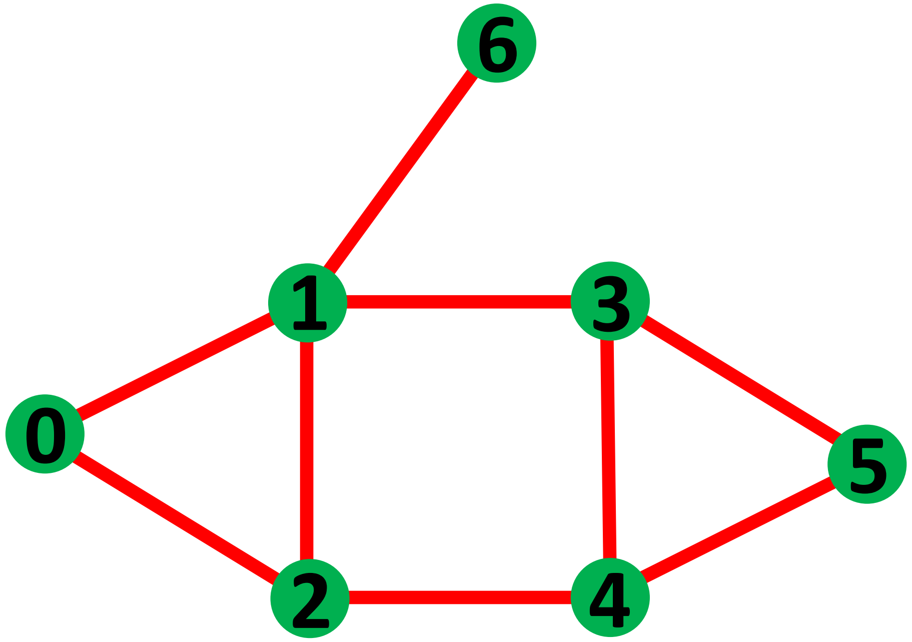
```
private boolean visited[] = new boolean[getNumVertices()];
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for (int neighbor : getNeighbors(n)) {
        if (!visited[neighbor]) {
            depthFirst(neighbor);
        }
    }
}
```

Run-time Stack

```
depthFirst(1)
depthFirst(0)
```

Output

```
0
1
2
4
3
5
```



Graphs - Traversal

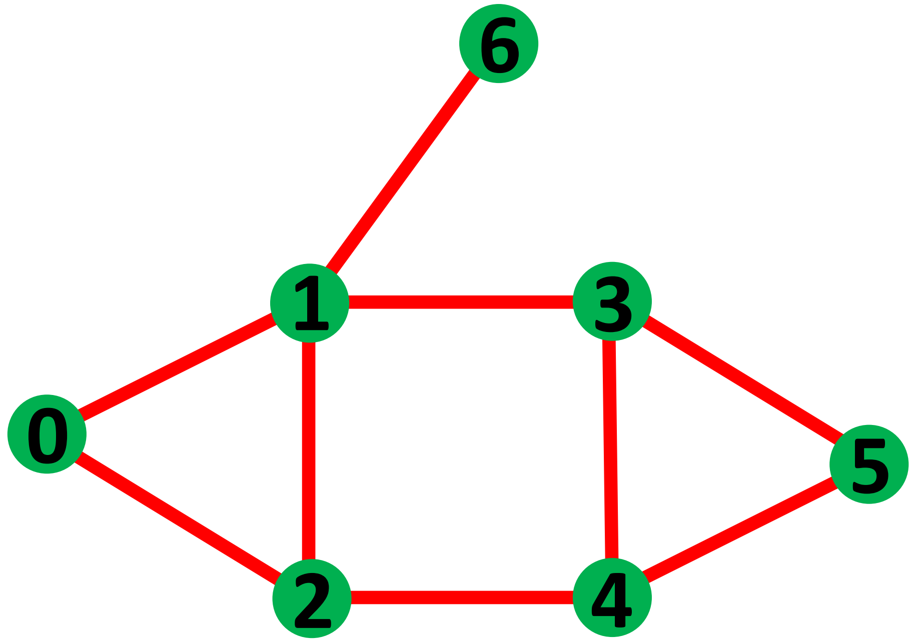
```
private boolean visited[] = new boolean[getNumVertices()];
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for (int neighbor : getNeighbors(n)) {
        if (!visited[neighbor]) {
            depthFirst(neighbor);
        }
    }
}
```

```
depthFirst(6)
depthFirst(1)
depthFirst(0)
```

Run-time Stack

Output

```
0
1
2
4
3
5
```



Graphs - Traversal

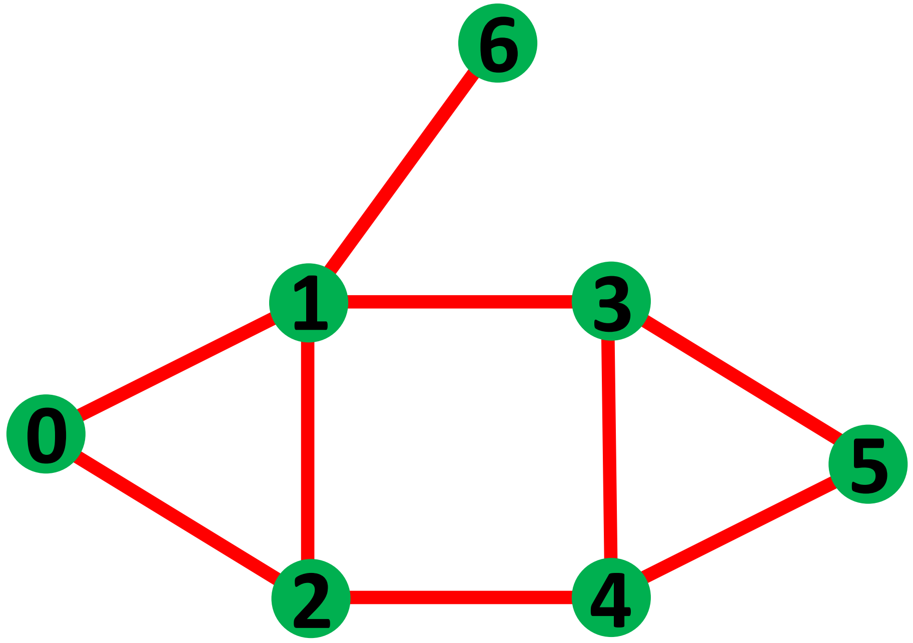
```
private boolean visited[] = new boolean[getNumVertices()];
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for (int neighbor : getNeighbors(n)) {
        if (!visited[neighbor]) {
            depthFirst(neighbor);
        }
    }
}
```

```
depthFirst(6)
depthFirst(1)
depthFirst(0)
```

Run-time Stack

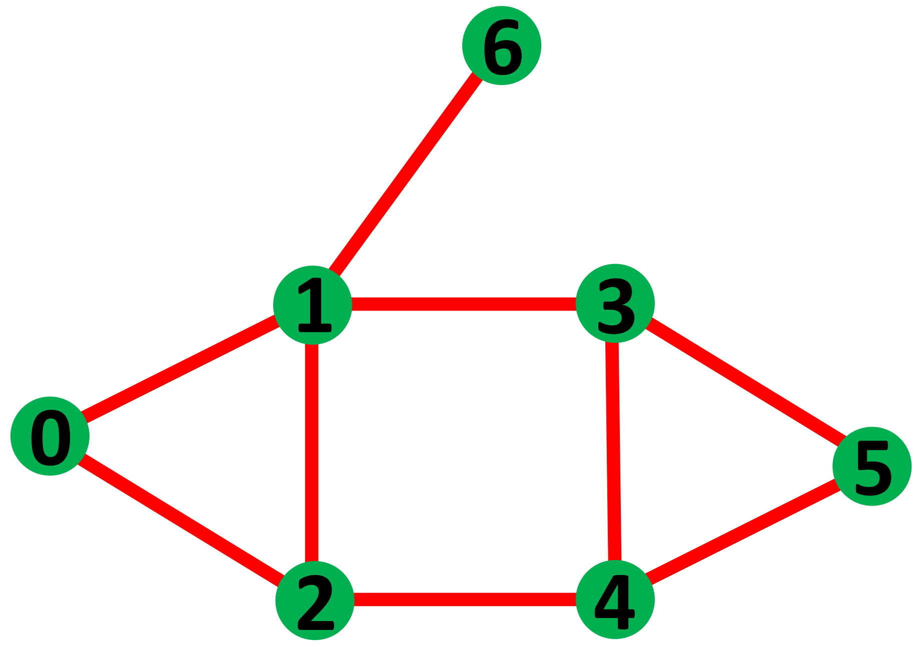
Output

```
0
1
2
4
3
5
6
```



Graphs - Traversal

```
private boolean visited[] = new boolean[getNumVertices()];
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for (int neighbor : getNeighbors(n)) {
        if (!visited[neighbor]) {
            depthFirst(neighbor);
        }
    }
}
```



```
depthFirst(1)
depthFirst(0)
```

Run-time Stack

Output

```
0
1
2
4
3
5
6
```

Graphs - Traversal

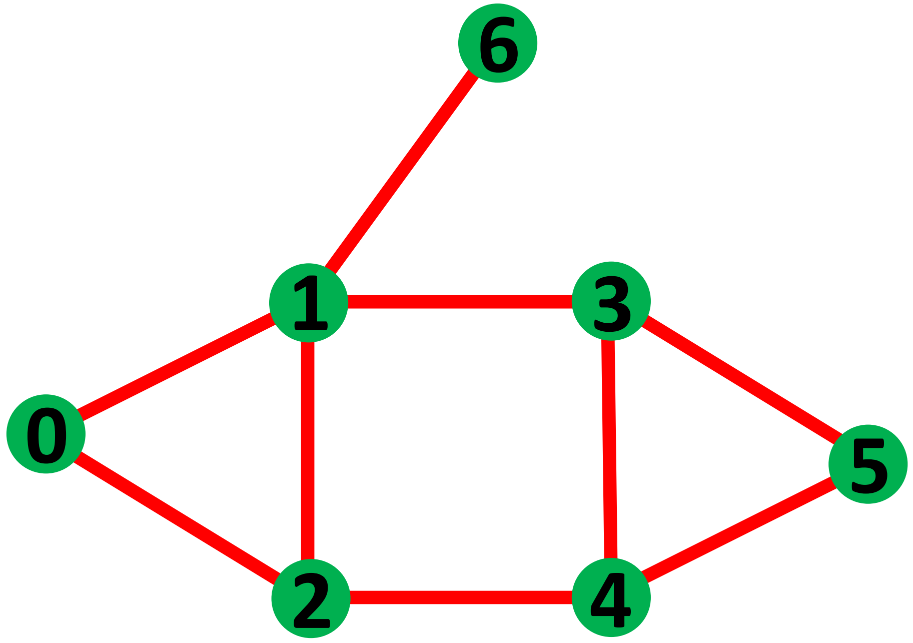
```
private boolean visited[] = new boolean[getNumVertices()];
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for (int neighbor : getNeighbors(n)) {
        if (!visited[neighbor]) {
            depthFirst(neighbor);
        }
    }
}
```

```
depthFirst(0)
```

Run-time Stack

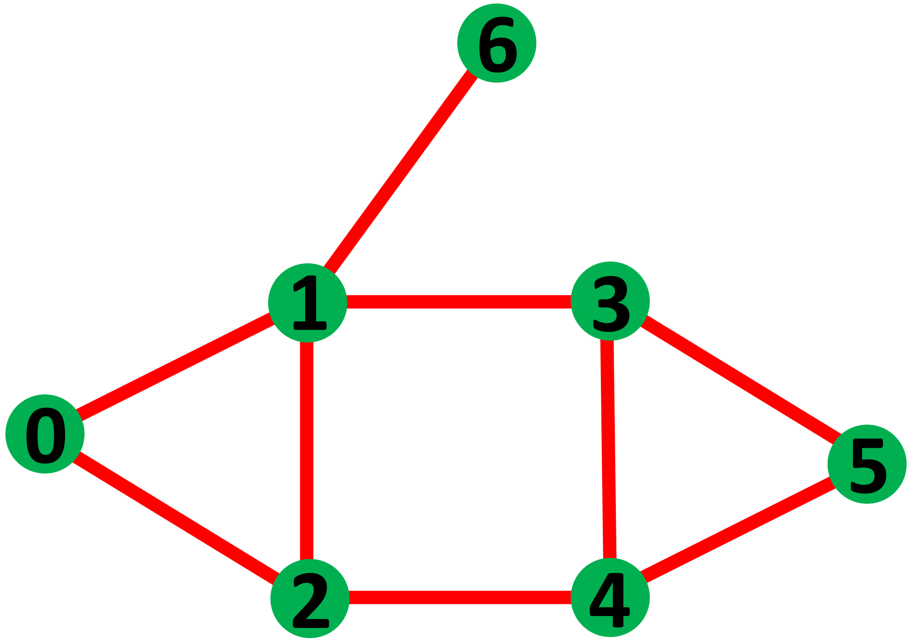
Output

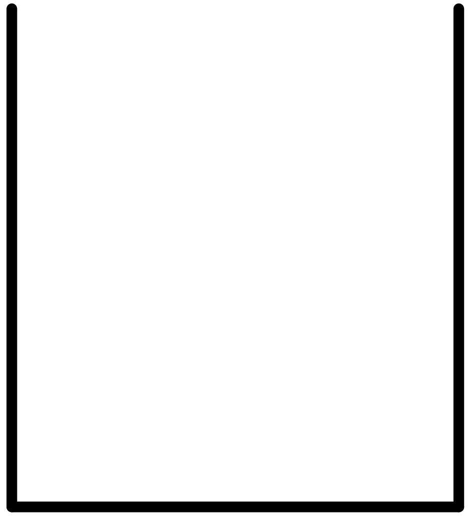
```
0
1
2
4
3
5
6
```



Graphs - Traversal

```
private boolean visited[] = new boolean[getNumVertices()];
public void depthFirst(int n) {
    System.out.println(n);
    visited[n] = true;
    for (int neighbor : getNeighbors(n)) {
        if (!visited[neighbor]) {
            depthFirst(neighbor);
        }
    }
}
```



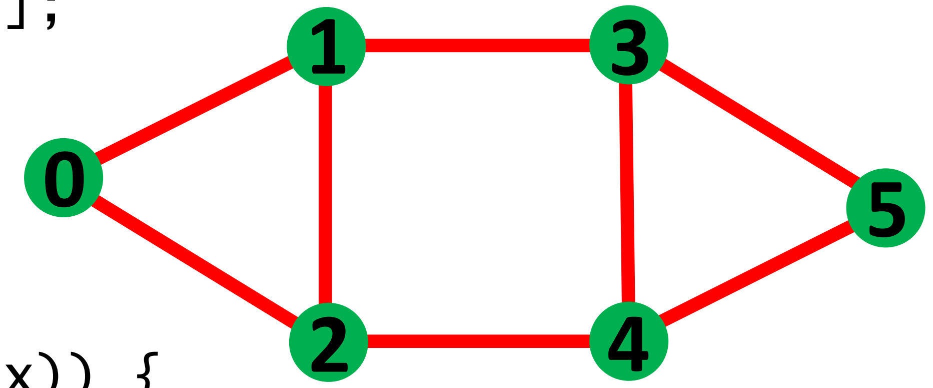


Output

```
0
1
2
4
3
5
6
```

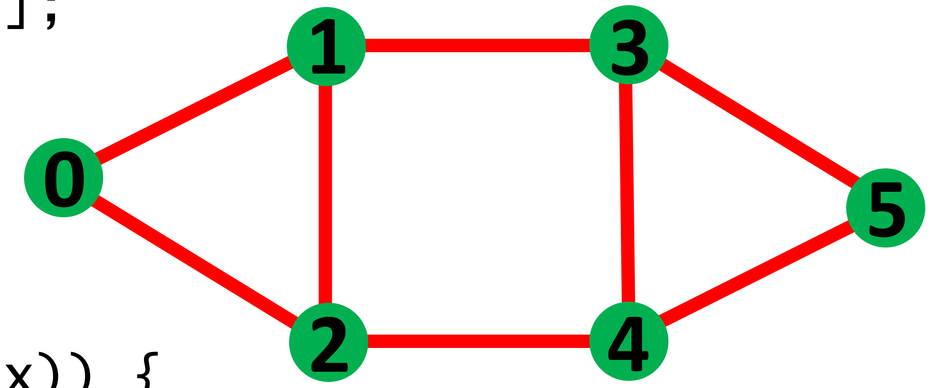
Graphs - Paths

```
private boolean[] visited;

public DepthFirstSearch(Graph graph, int startVertex) {
   visited = new boolean[graph.getNumVertices()];
   dfs(graph, startVertex);
}

private void dfs(Graph graph, int vertex) {
   visited[vertex] = true;

   for (int neighbor : graph.getNeighbors(vertex)) {
      if (!visited[neighbor]) {
         dfs(graph, neighbor);
      }
   }
}

public boolean reachable(int endVertex) {
   return visited[endVertex];
}
```



Graphs - Paths

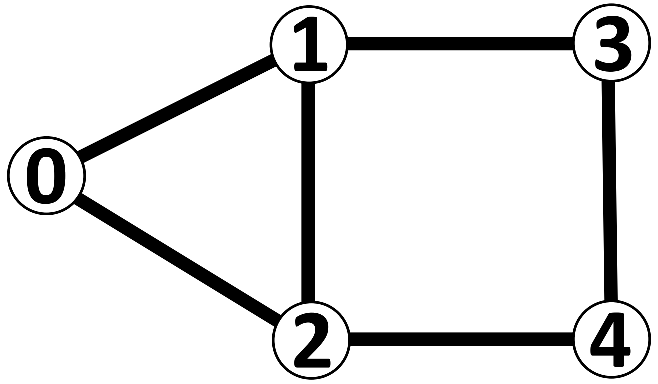
```
private boolean[] visited;

public DepthFirstSearch(Graph graph, int startVertex) {
   visited = new boolean[graph.getNumVertices()];
   dfs(graph, startVertex);
}

private void dfs(Graph graph, int vertex) {
   visited[vertex] = true;

   for (int neighbor : graph.getNeighbors(vertex)) {
      if (!visited[neighbor]) {
         dfs(graph, neighbor);
      }
   }
}

public boolean reachable(int endVertex) {
   return visited[endVertex];
}
```



How do we get actual paths between vertices?

Graphs - Paths

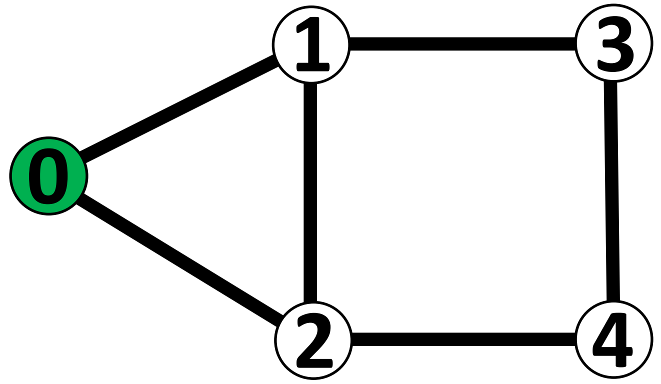
```
private boolean[] visited;

public DepthFirstSearch(Graph graph, int startVertex) {
   visited = new boolean[graph.getNumVertices()];
   dfs(graph, startVertex);
}

private void dfs(Graph graph, int vertex) {
   visited[vertex] = true;

   for (int neighbor : graph.getNeighbors(vertex)) {
      if (!visited[neighbor]) {
         dfs(graph, neighbor);
      }
   }
}
```

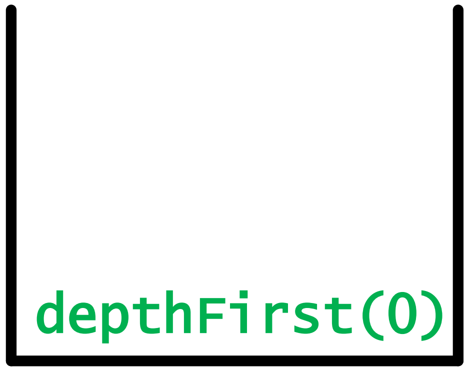


Run-time Stack

Graphs - Paths

```
private boolean[] visited;

public DepthFirstSearch(Graph graph, int startVertex) {
   visited = new boolean[graph.getNumVertices()];
   dfs(graph, startVertex);
}

private void dfs(Graph graph, int vertex) {
   visited[vertex] = true;

   for (int neighbor : graph.getNeighbors(vertex)) {
      if (!visited[neighbor]) {
         dfs(graph, neighbor);
      }
   }
}
```

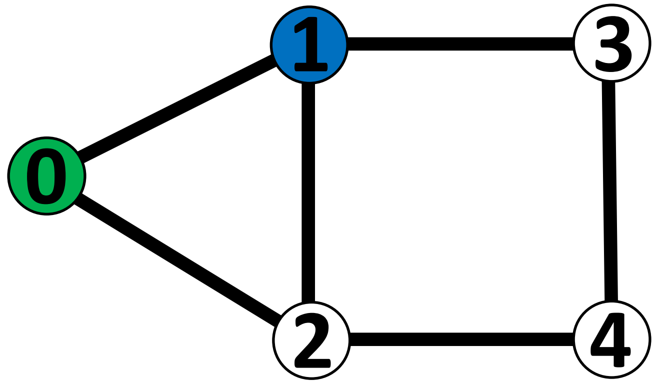


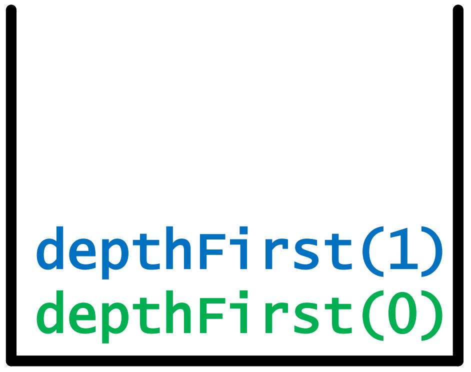


Run-time Stack

Graphs - Paths

```
private boolean[] visited;

public DepthFirstSearch(Graph graph, int startVertex) {
   visited = new boolean[graph.getNumVertices()];
   dfs(graph, startVertex);
}

private void dfs(Graph graph, int vertex) {
   visited[vertex] = true;

   for (int neighbor : graph.getNeighbors(vertex)) {
      if (!visited[neighbor]) {
         dfs(graph, neighbor);
      }
   }
}
```

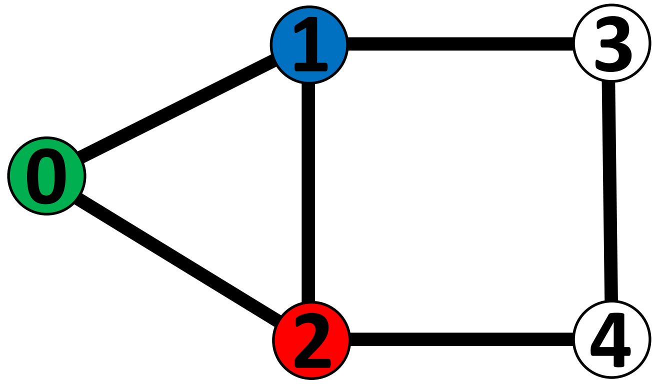


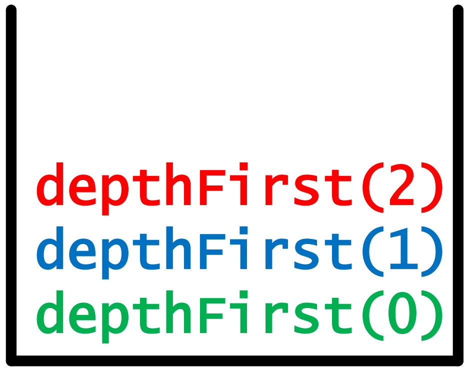


Run-time Stack

Graphs - Paths

```
private boolean[] visited;

public DepthFirstSearch(Graph graph, int startVertex) {
   visited = new boolean[graph.getNumVertices()];
   dfs(graph, startVertex);
}

private void dfs(Graph graph, int vertex) {
   visited[vertex] = true;

   for (int neighbor : graph.getNeighbors(vertex)) {
      if (!visited[neighbor]) {
         dfs(graph, neighbor);
      }
   }
}
```

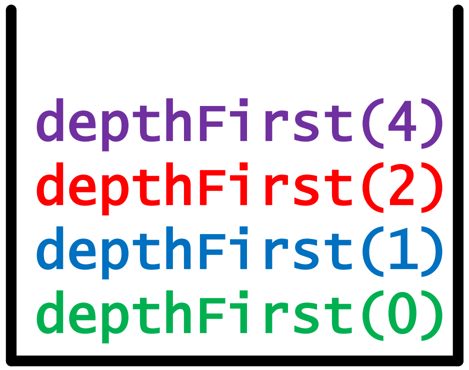




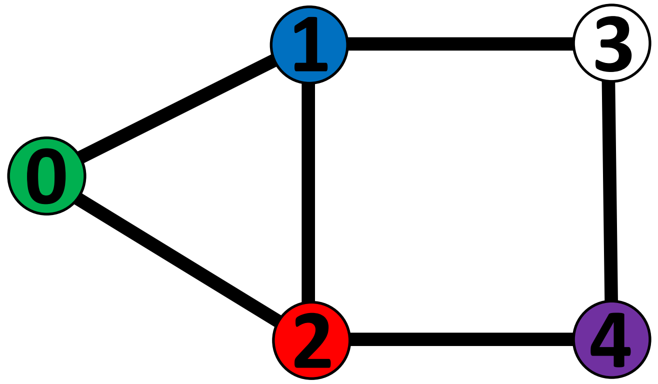
Run-time Stack

Graphs - Paths

```
private boolean[] visited;

public DepthFirstSearch(Graph graph, int startVertex) {
   visited = new boolean[graph.getNumVertices()];
   dfs(graph, startVertex);
}

private void dfs(Graph graph, int vertex) {
   visited[vertex] = true;

   for (int neighbor : graph.getNeighbors(vertex)) {
      if (!visited[neighbor]) {
         dfs(graph, neighbor);
      }
   }
}
```

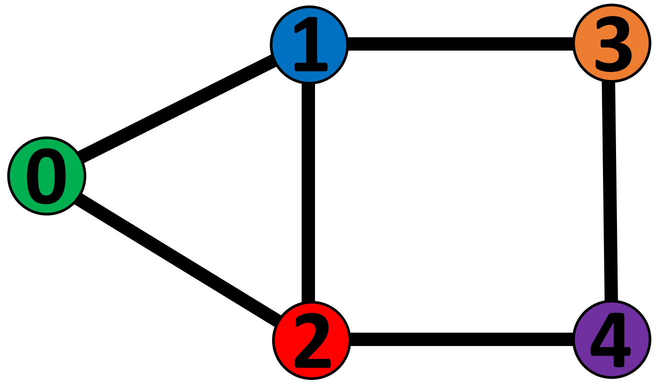


Run-time Stack



Graphs - Paths

```
private boolean[] visited;

public DepthFirstSearch(Graph graph, int startVertex) {
   visited = new boolean[graph.getNumVertices()];
   dfs(graph, startVertex);
}

private void dfs(Graph graph, int vertex) {
   visited[vertex] = true;

   for (int neighbor : graph.getNeighbors(vertex)) {
      if (!visited[neighbor]) {
         dfs(graph, neighbor);
      }
   }
}
```



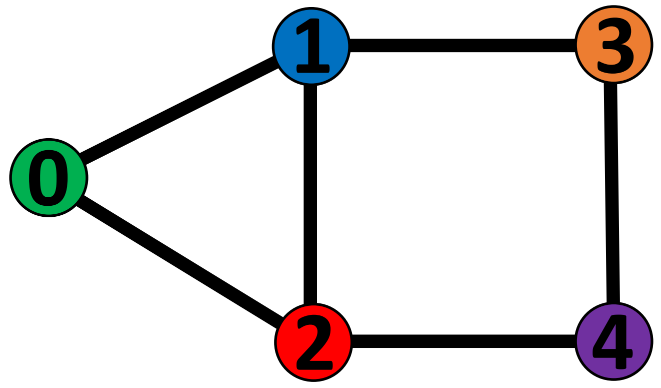
```
depthFirst(3)
depthFirst(4)
depthFirst(2)
depthFirst(1)
depthFirst(0)
```

Run-time Stack

Was a path identified when determining that 0 and 3 are connected?

Graphs - Paths

```
private boolean[] visited;

public DepthFirstSearch(Graph graph, int startVertex) {
   visited = new boolean[graph.getNumVertices()];
   dfs(graph, startVertex);
}

private void dfs(Graph graph, int vertex) {
   visited[vertex] = true;

   for (int neighbor : graph.getNeighbors(vertex)) {
      if (!visited[neighbor]) {
         dfs(graph, neighbor);
      }
   }
}
```



```
depthFirst(3)
depthFirst(4)
depthFirst(2)
depthFirst(1)
depthFirst(0)
```

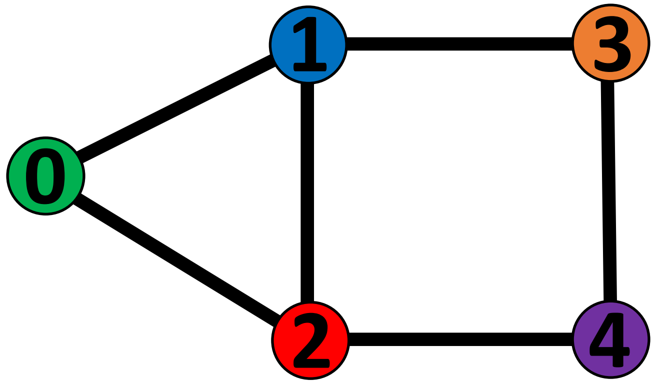
Run-time Stack

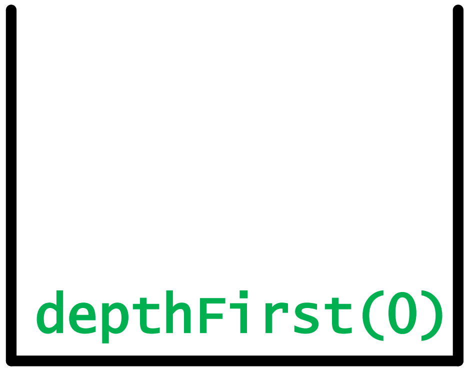
Was a path identified when determining that 0 and 3 are connected?

YES!

Graphs - Paths

```
private boolean[] visited;

public DepthFirstSearch(Graph graph, int startVertex) {
   visited = new boolean[graph.getNumVertices()];
   dfs(graph, startVertex);
}

private void dfs(Graph graph, int vertex) {
   visited[vertex] = true;

   for (int neighbor : graph.getNeighbors(vertex)) {
      if (!visited[neighbor]) {
         dfs(graph, neighbor);
      }
   }
}
```





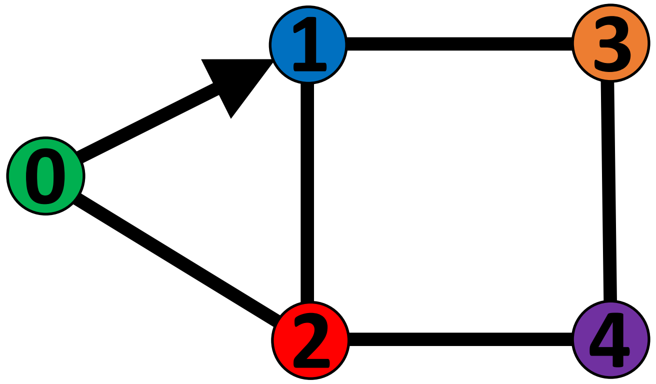
Run-time Stack

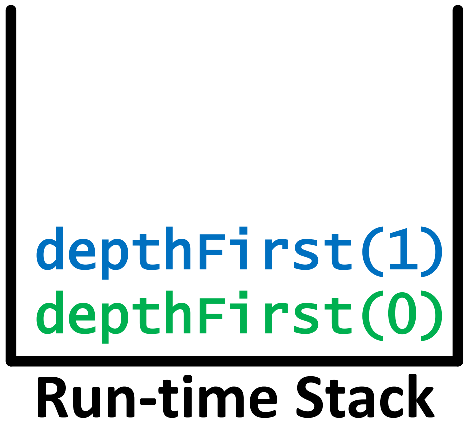
Was a path identified when determining that 0 and 3 are connected?

YES!

Graphs - Paths

```
private boolean[] visited;

public DepthFirstSearch(Graph graph, int startVertex) {
   visited = new boolean[graph.getNumVertices()];
   dfs(graph, startVertex);
}

private void dfs(Graph graph, int vertex) {
   visited[vertex] = true;

   for (int neighbor : graph.getNeighbors(vertex)) {
      if (!visited[neighbor]) {
         dfs(graph, neighbor);
      }
   }
}
```



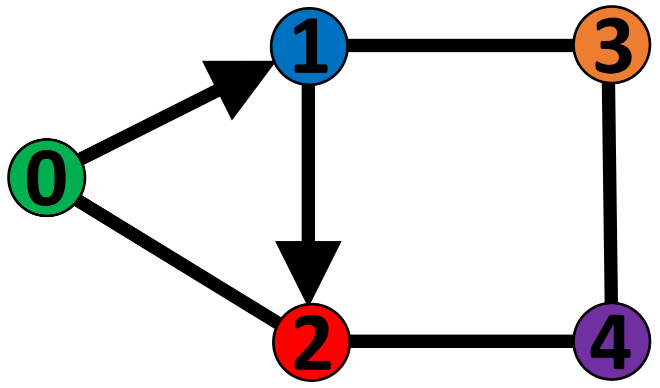


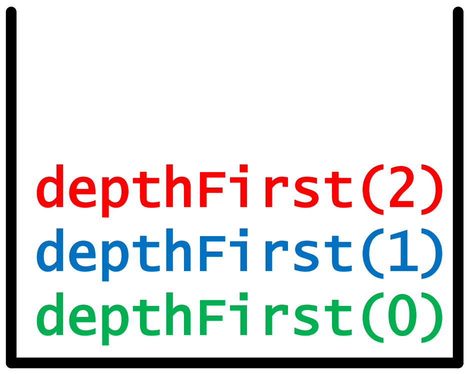
Was a path identified when determining that 0 and 3 are connected?

YES!

Graphs - Paths

```
private boolean[] visited;

public DepthFirstSearch(Graph graph, int startVertex) {
   visited = new boolean[graph.getNumVertices()];
   dfs(graph, startVertex);
}

private void dfs(Graph graph, int vertex) {
   visited[vertex] = true;

   for (int neighbor : graph.getNeighbors(vertex)) {
      if (!visited[neighbor]) {
         dfs(graph, neighbor);
      }
   }
}
```





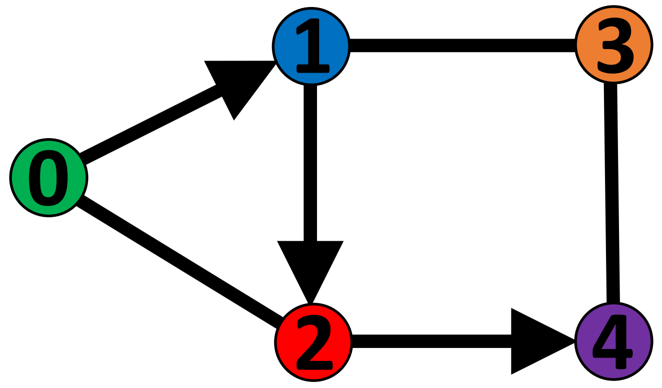
Run-time Stack

Was a path identified when determining that 0 and 3 are connected?

YES!

Graphs - Paths

```
private boolean[] visited;

public DepthFirstSearch(Graph graph, int startVertex) {
   visited = new boolean[graph.getNumVertices()];
   dfs(graph, startVertex);
}

private void dfs(Graph graph, int vertex) {
   visited[vertex] = true;

   for (int neighbor : graph.getNeighbors(vertex)) {
      if (!visited[neighbor]) {
         dfs(graph, neighbor);
      }
   }
}
```



```
depthFirst(4)
depthFirst(2)
depthFirst(1)
depthFirst(0)
```

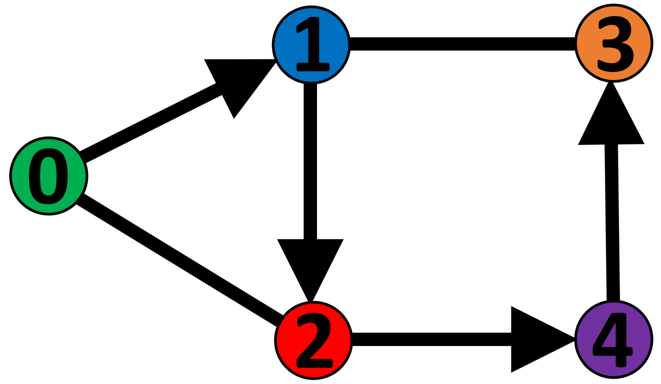
Run-time Stack

Was a path identified when determining that 0 and 3 are connected?

YES!

Graphs - Paths

```
private boolean[] visited;

public DepthFirstSearch(Graph graph, int startVertex) {
   visited = new boolean[graph.getNumVertices()];
   dfs(graph, startVertex);
}

private void dfs(Graph graph, int vertex) {
   visited[vertex] = true;

   for (int neighbor : graph.getNeighbors(vertex)) {
      if (!visited[neighbor]) {
         dfs(graph, neighbor);
      }
   }
}
```



```
depthFirst(3)
depthFirst(4)
depthFirst(2)
depthFirst(1)
depthFirst(0)
```

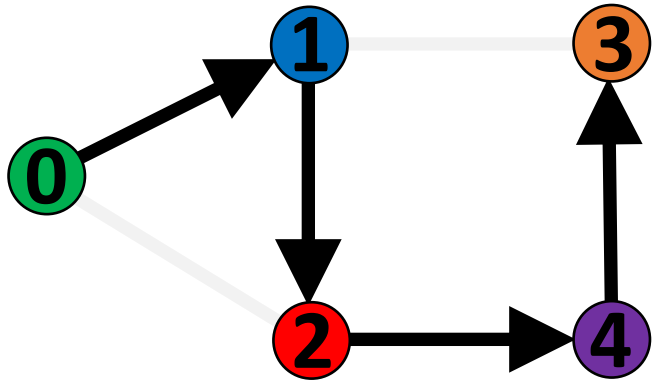
Run-time Stack

Was a path identified when determining that 0 and 3 are connected?

YES!

Graphs - Paths

```
private boolean[] visited;

public DepthFirstSearch(Graph graph, int startVertex) {
   visited = new boolean[graph.getNumVertices()];
   dfs(graph, startVertex);
}

private void dfs(Graph graph, int vertex) {
   visited[vertex] = true;

   for (int neighbor : graph.getNeighbors(vertex)) {
      if (!visited[neighbor]) {
         dfs(graph, neighbor);
      }
   }
}
```



```
depthFirst(3)
depthFirst(4)
depthFirst(2)
depthFirst(1)
depthFirst(0)
```

Run-time Stack

Was a path identified when determining that 0 and 3 are connected?

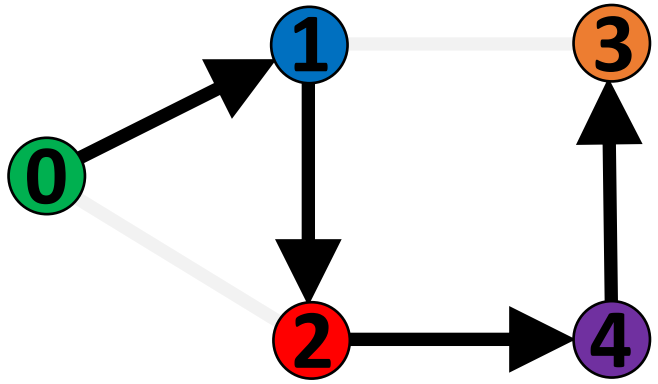
YES!

Graphs - Paths

```
private boolean[] visited;

public DepthFirstSearch(Graph graph, int startVertex) {
   visited = new boolean[graph.getNumVertices()];
   dfs(graph, startVertex);
}

private void dfs(Graph graph, int vertex) {
   visited[vertex] = true;

   for (int neighbor : graph.getNeighbors(vertex)) {
      if (!visited[neighbor]) {
         dfs(graph, neighbor);
      }
   }
}
```

```
depthFirst(3)
depthFirst(4)
depthFirst(2)
depthFirst(1)
depthFirst(0)
```

Run-time Stack



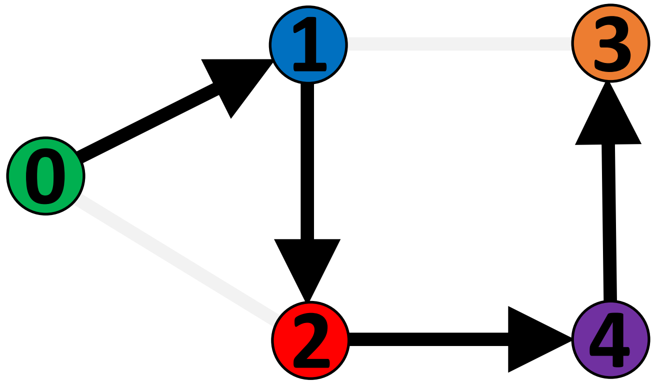
What else was identified?

Graphs - Paths

```
private boolean[] visited;

public DepthFirstSearch(Graph graph, int startVertex) {
   visited = new boolean[graph.getNumVertices()];
   dfs(graph, startVertex);
}

private void dfs(Graph graph, int vertex) {
   visited[vertex] = true;

   for (int neighbor : graph.getNeighbors(vertex)) {
      if (!visited[neighbor]) {
         dfs(graph, neighbor);
      }
   }
}
```

```
depthFirst(3)
depthFirst(4)
depthFirst(2)
depthFirst(1)
depthFirst(0)
```

Run-time Stack

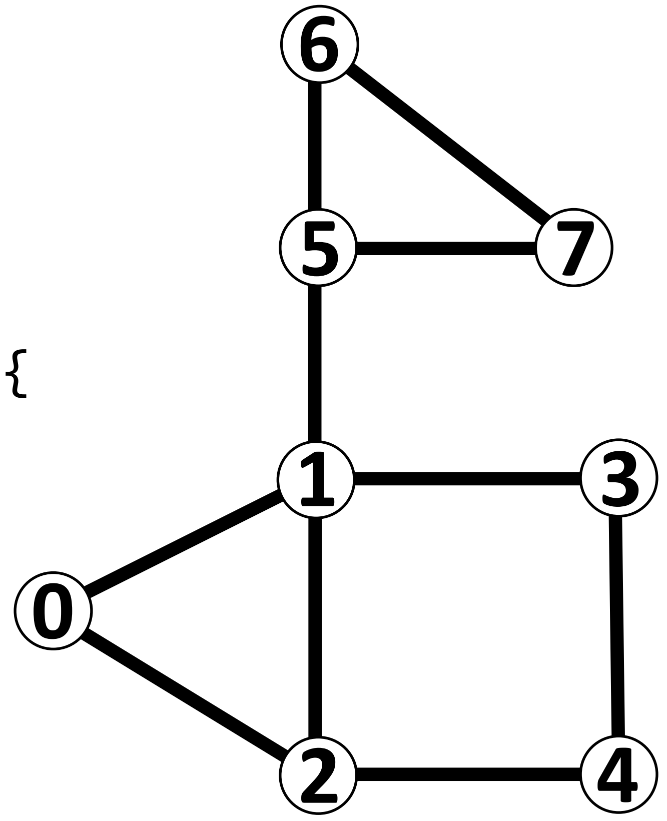


What else was identified?

A path from 0 to <u>everything</u> connected to 0!

Graphs - Paths

```
private boolean[] visited;

public DepthFirstSearch(Graph graph, int startVertex) {
   visited = new boolean[graph.getNumVertices()];
   dfs(graph, startVertex);
}

private void dfs(Graph graph, int vertex) {
   visited[vertex] = true;

   for (int neighbor : graph.getNeighbors(vertex)) {
      if (!visited[neighbor]) {
         dfs(graph, neighbor);
      }
   }
}
```

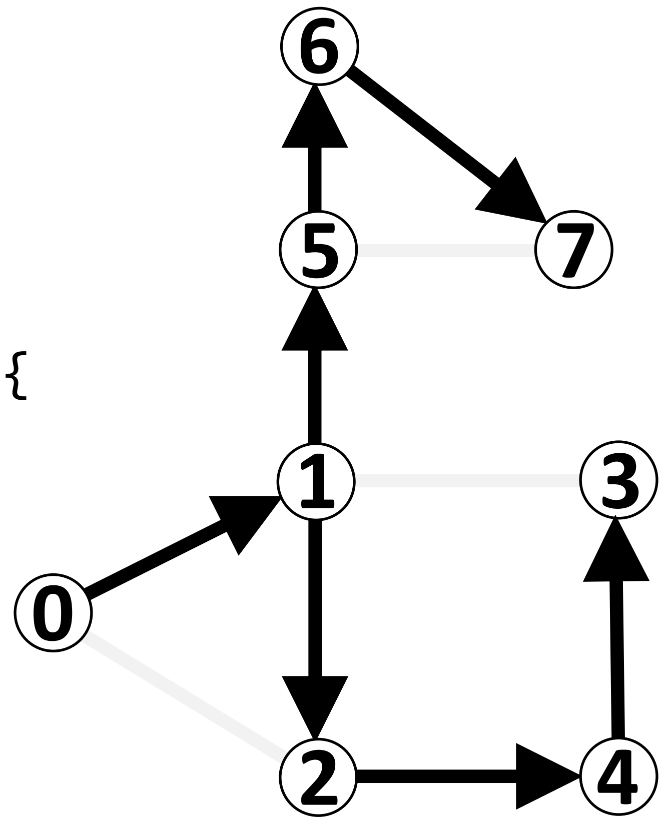


What else was identified?

A path from 0 to <u>everything</u> connected to 0!

Graphs - Paths

```
private boolean[] visited;

public DepthFirstSearch(Graph graph, int startVertex) {
   visited = new boolean[graph.getNumVertices()];
   dfs(graph, startVertex);
}

private void dfs(Graph graph, int vertex) {
   visited[vertex] = true;

   for (int neighbor : graph.getNeighbors(vertex)) {
      if (!visited[neighbor]) {
         dfs(graph, neighbor);
      }
   }
}
```

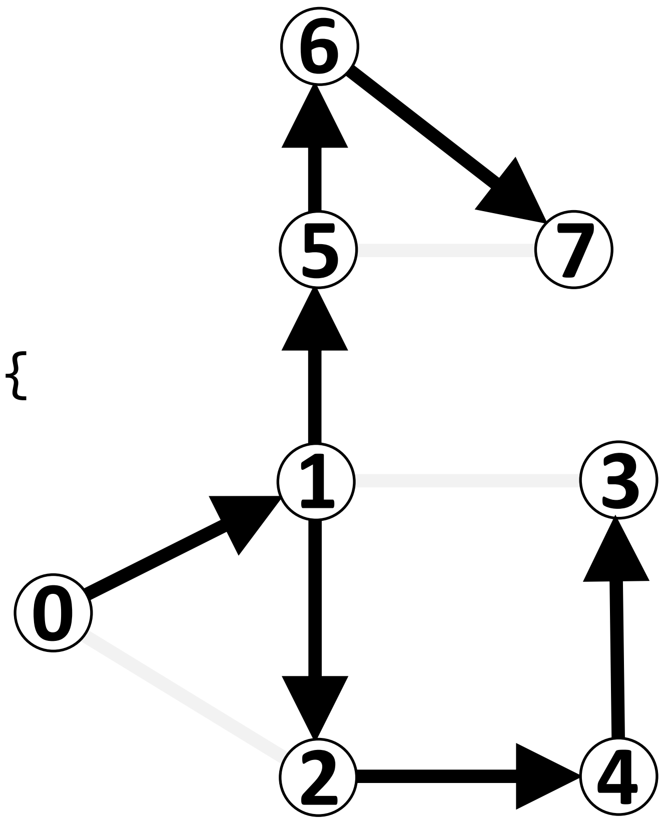


What else was identified?

A path from 0 to <u>everything</u> connected to 0!

Graphs - Paths

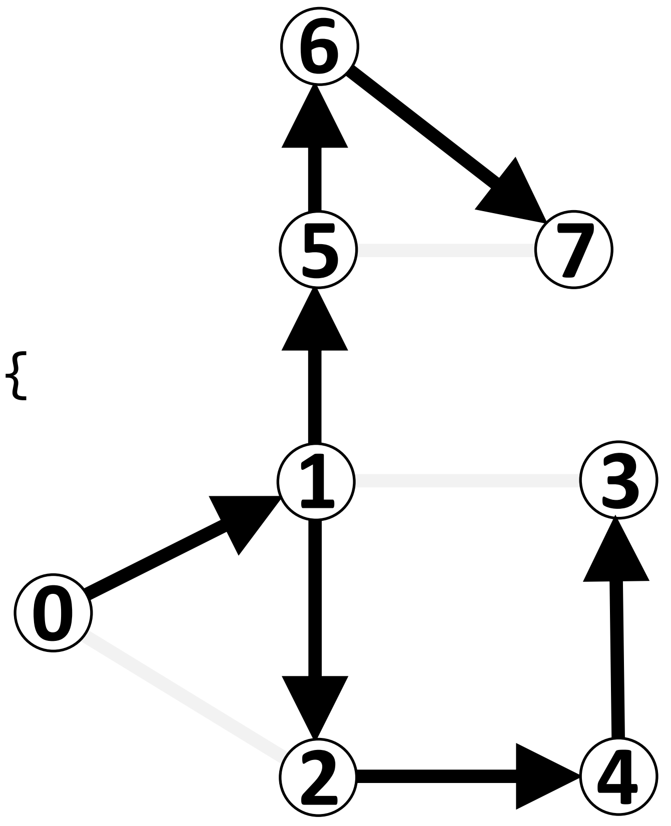
```
private boolean[] visited;

public DepthFirstSearch(Graph graph, int startVertex) {
   visited = new boolean[graph.getNumVertices()];
   dfs(graph, startVertex);
}

private void dfs(Graph graph, int vertex) {
   visited[vertex] = true;

   for (int neighbor : graph.getNeighbors(vertex)) {
      if (!visited[neighbor]) {
         dfs(graph, neighbor);
      }
   }
}
```



How can we store/compute these paths?

Graphs - Paths

```
private boolean[] visited;

public DepthFirstSearch(Graph graph, int startVertex) {
   visited = new boolean[graph.getNumVertices()];
   dfs(graph, startVertex);
}

private void dfs(Graph graph, int vertex) {
   visited[vertex] = true;

   for (int neighbor : graph.getNeighbors(vertex)) {
      if (!visited[neighbor]) {
         dfs(graph, neighbor);
      }
   }
}
```

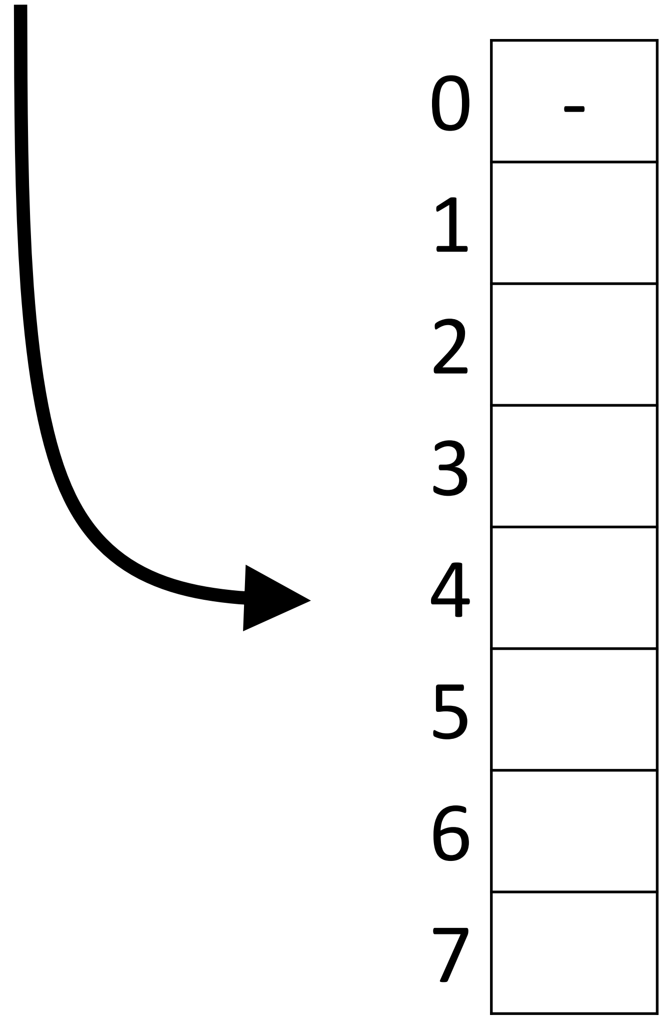


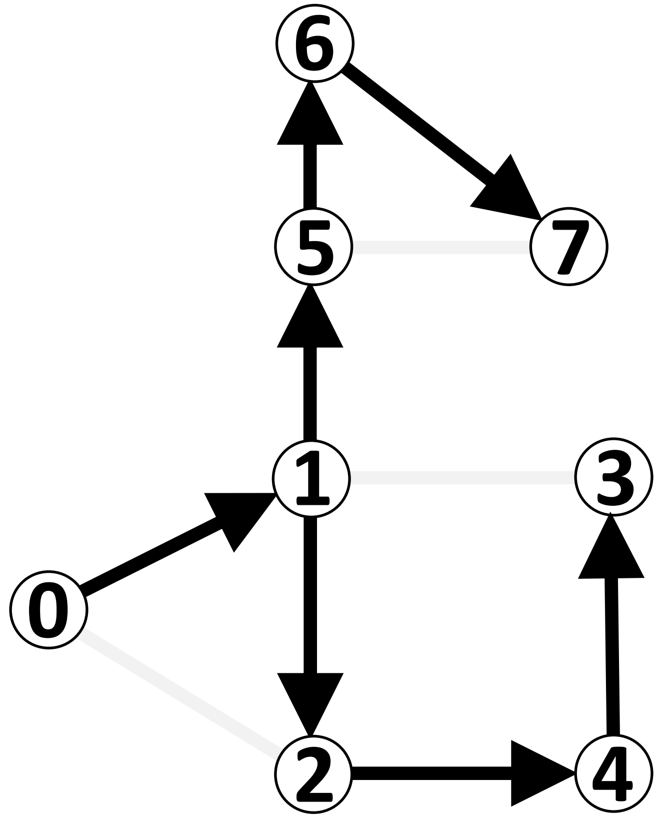
How can we store/compute these paths?

What if, for each vertex, we stored the previous vertex?

Graphs - Paths

`int[] previousVertex`





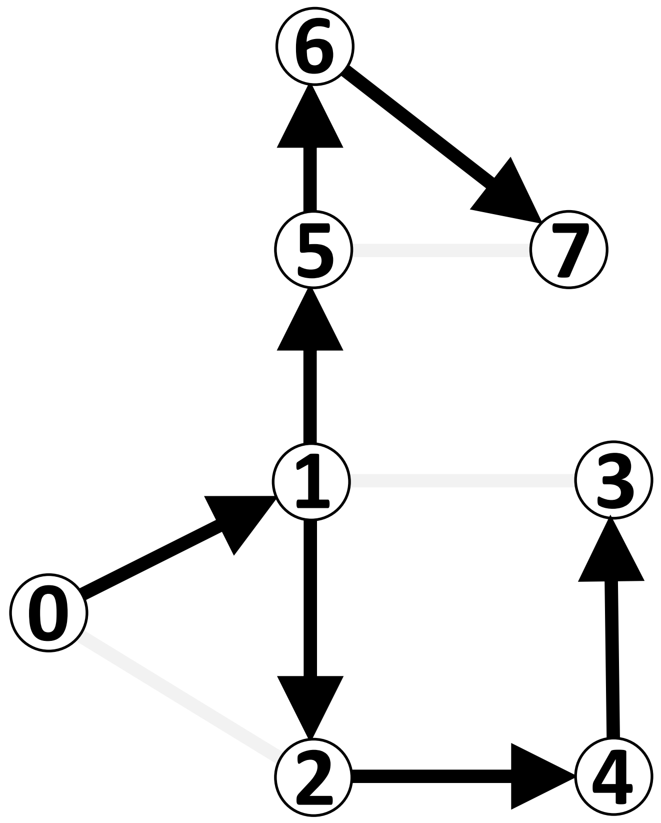
How can we store/compute these paths?

What if, for each vertex, we stored the previous vertex?

Graphs - Paths

```
int[] previousVertex
```

0	-
1	0
2	
3	
4	
5	
6	
7	



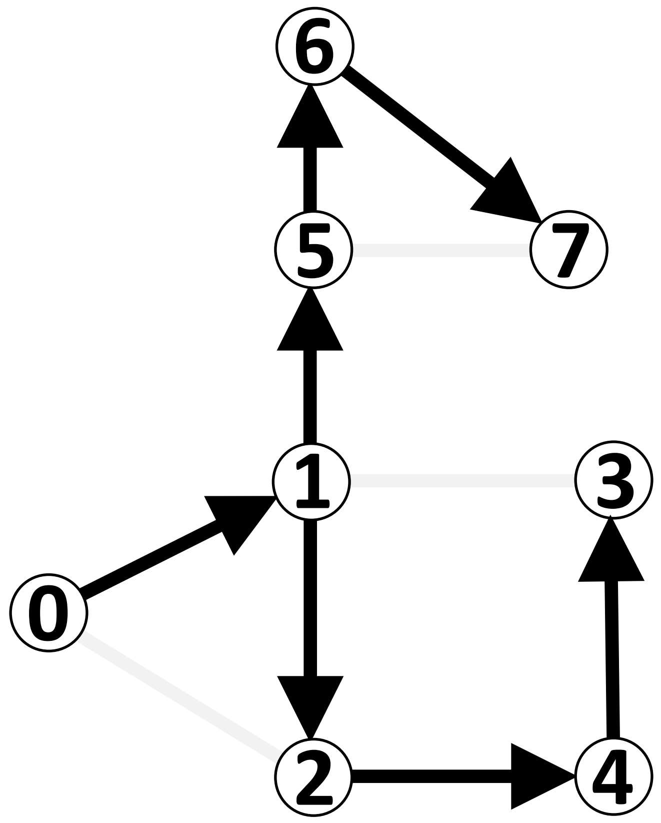
How can we store/compute these paths?

What if, for each vertex, we stored the previous vertex?

Graphs - Paths

```
int[] previousVertex
```

0	-
1	0
2	1
3	
4	
5	
6	
7	



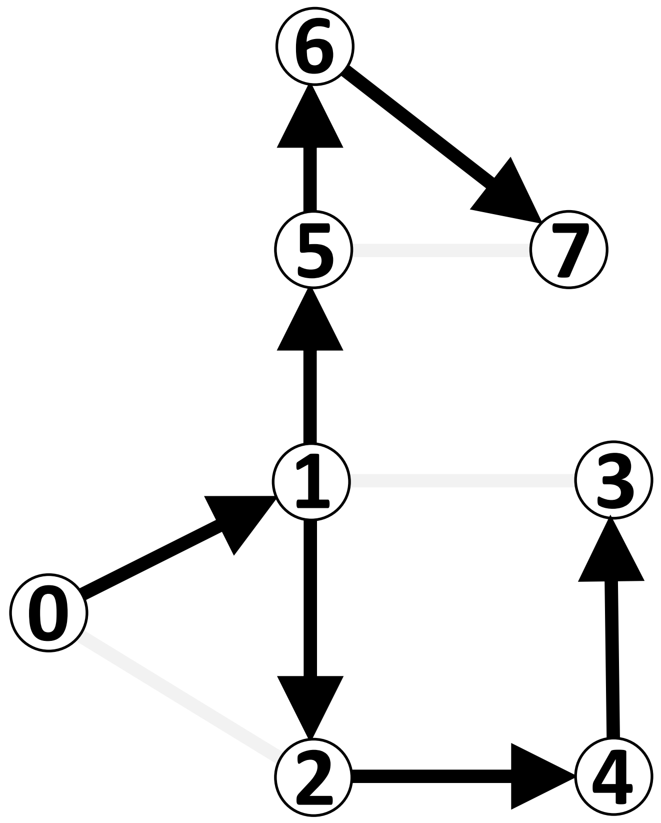
How can we store/compute these paths?

What if, for each vertex, we stored the previous vertex?

Graphs - Paths

```
int[] previousVertex
```

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



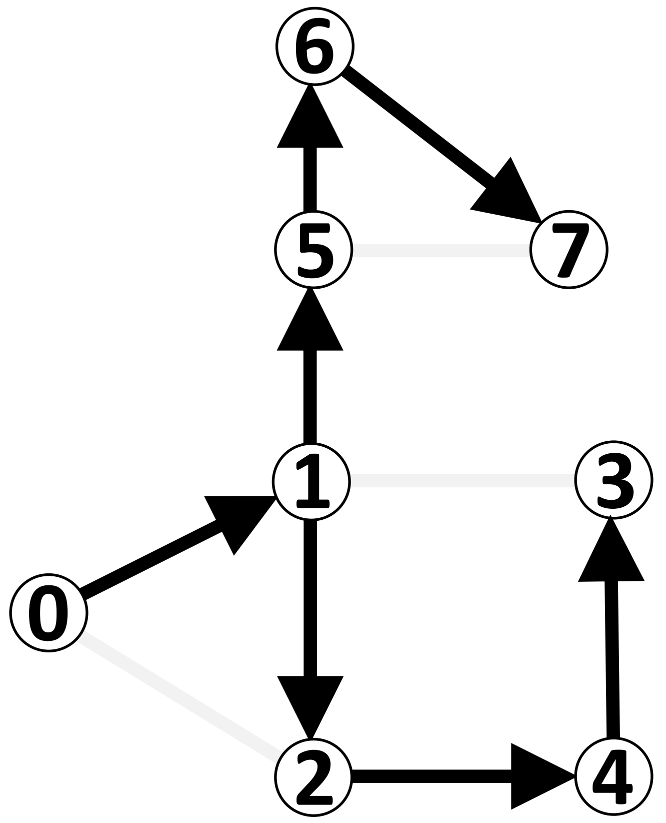
How can we store/compute these paths?

What if, for each vertex, we stored the previous vertex?

Graphs - Paths

```
int[] previousVertex
```

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



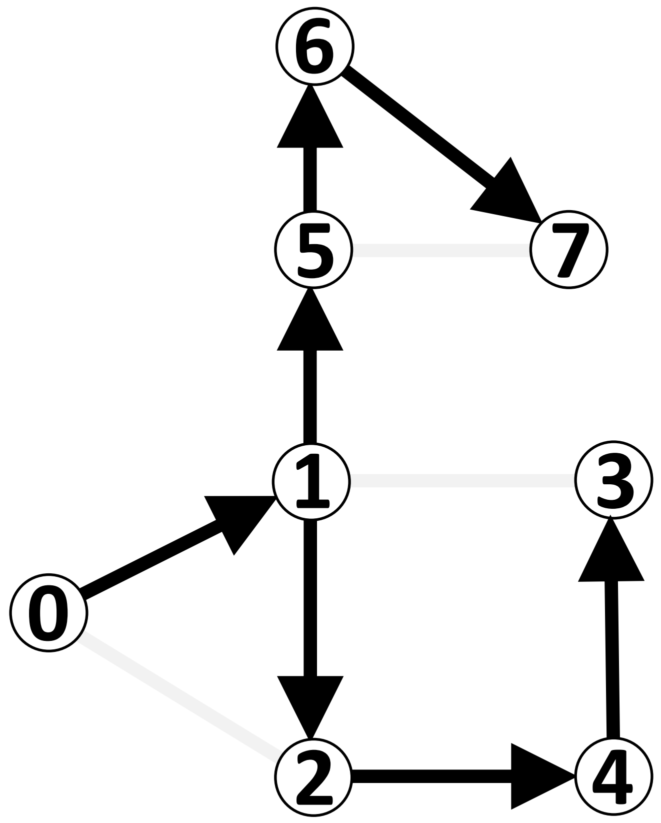
How can we store/compute these paths?

What if, for each vertex, we stored the previous vertex?

Graphs - Paths

```
int[] previousVertex
```

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



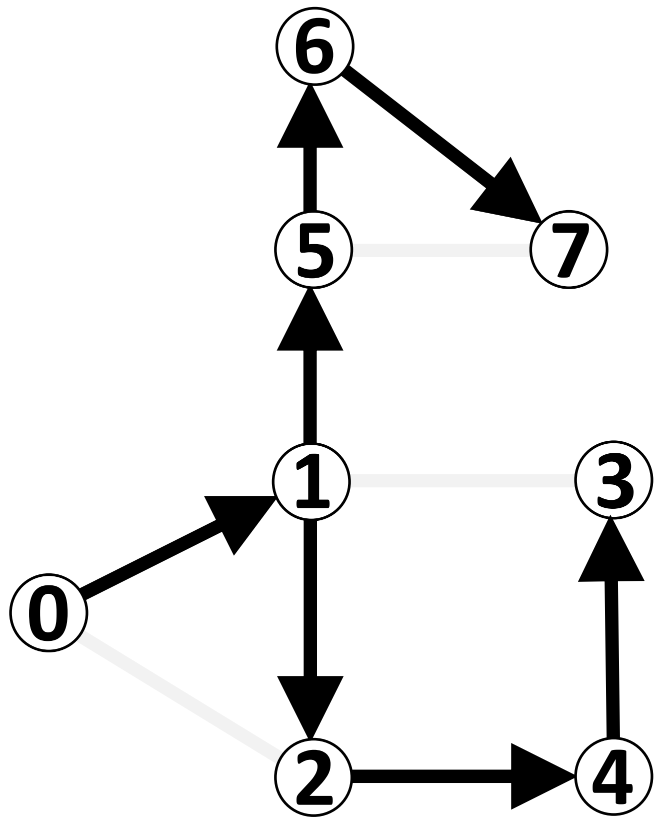
How can we store/compute these paths?

What if, for each vertex, we stored the previous vertex?

Graphs - Paths

```
int[] previousVertex
```

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



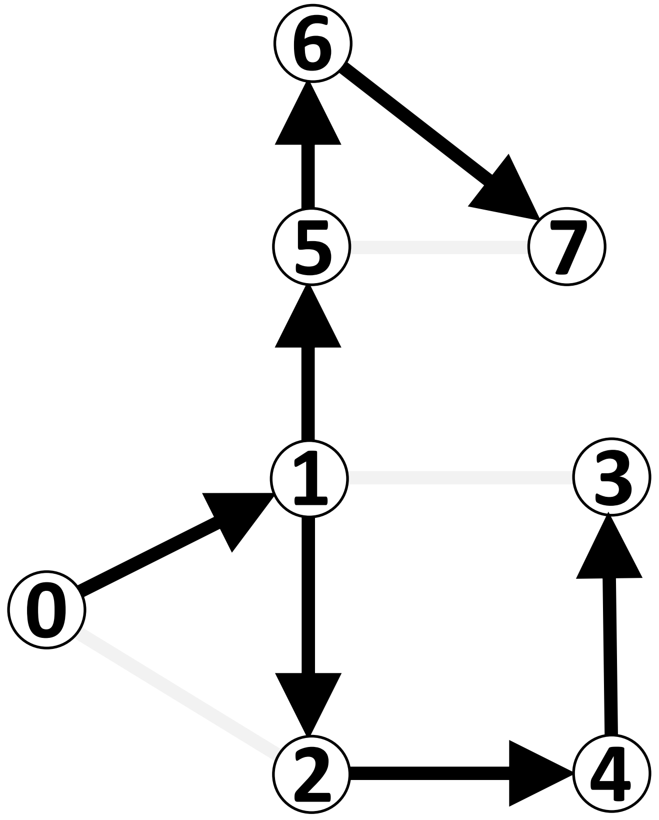
How can we store/compute these paths?

What if, for each vertex, we stored the previous vertex?

Graphs - Paths

`int[] previousVertex`

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



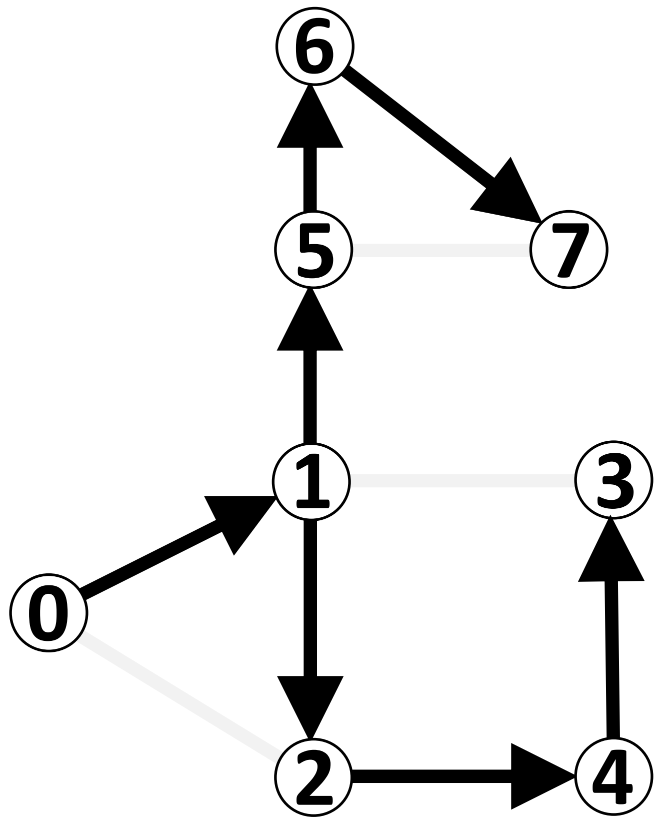
How can we store/compute these paths?

What if, for each vertex, we stored the previous vertex?

Graphs - Paths

```
int[] previousVertex
```

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

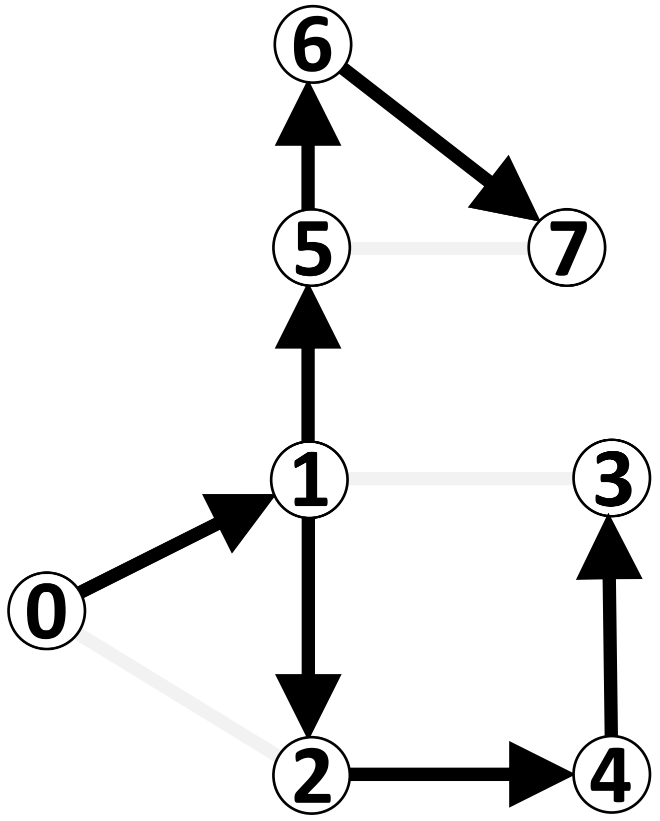


How do we determine the path from 0 to 6?

Graphs - Paths

`int[] previousVertex`

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



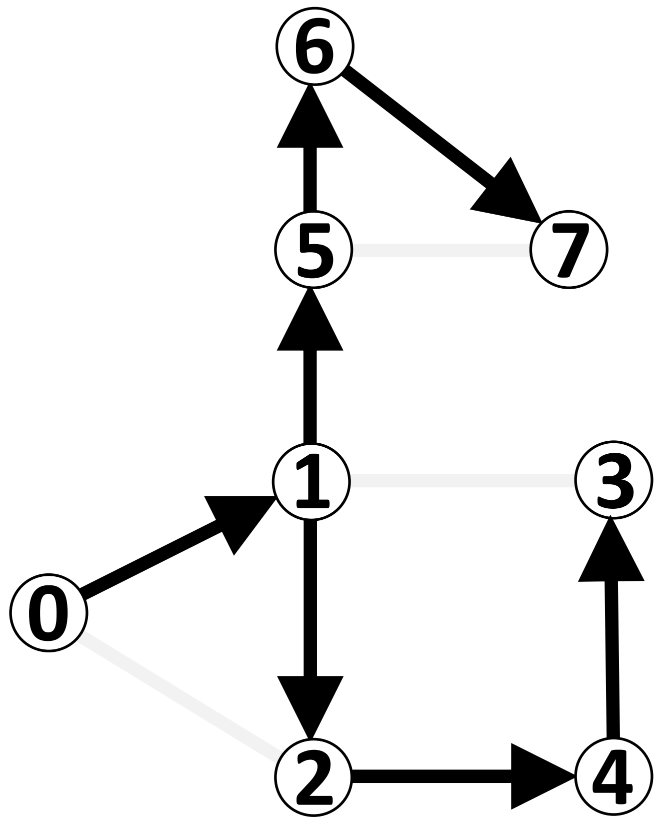
How do we determine the path from 0 to 6?

Start at vertex 6.

Graphs - Paths

```
int[] previousVertex
```

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



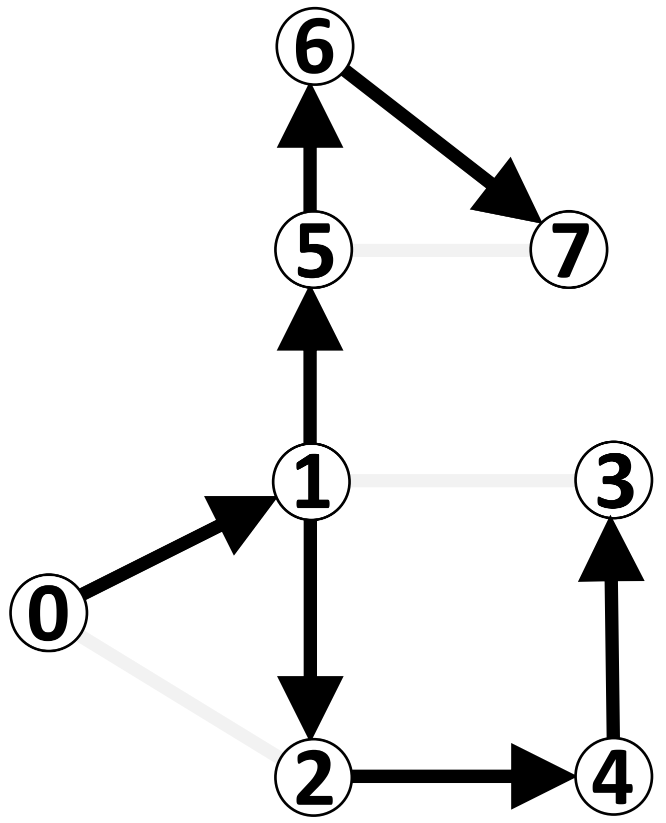
How do we determine the path from 0 to 6?

Start at vertex 6. Find its previous vertex.

Graphs - Paths

```
int[] previousVertex
```

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



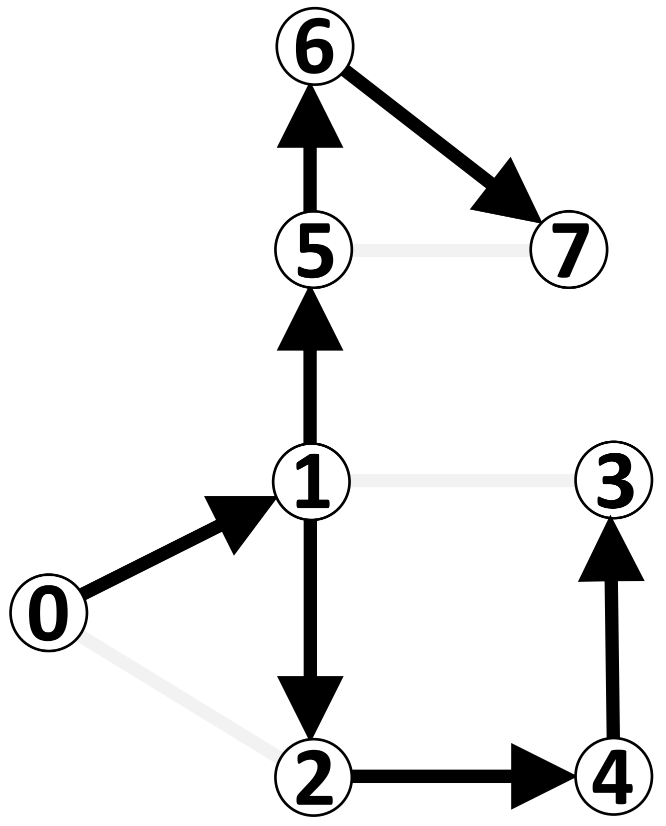
How do we determine the path from 0 to 6?

Start at vertex 6. Find its previous vertex. Find its previous vertex

Graphs - Paths

```
int[] previousVertex
```

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

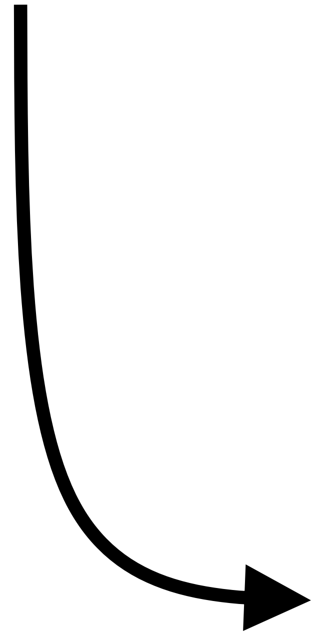


How do we determine the path from 0 to 6?

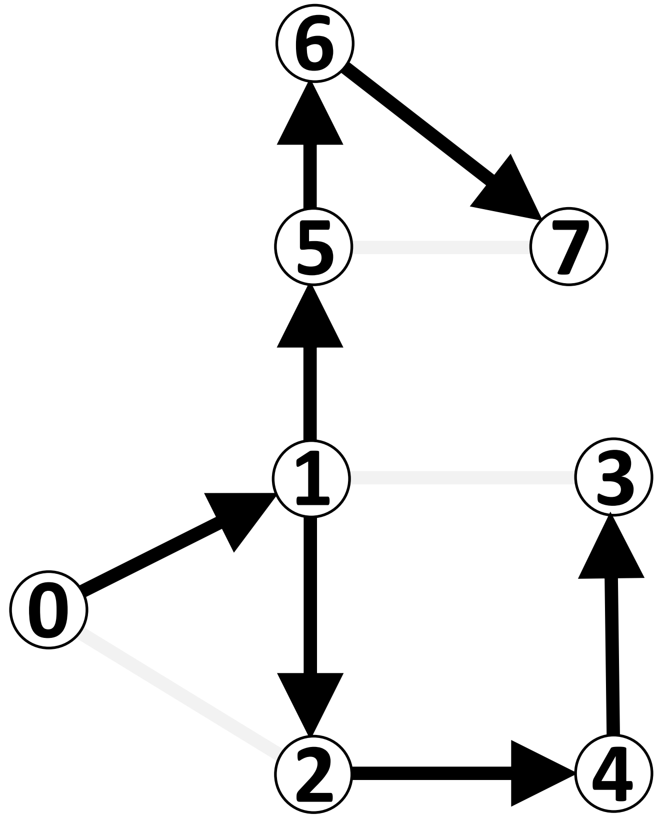
Start at vertex 6. Find its previous vertex. Find its previous vertex… until we get back to the start (0).

Graphs - Paths

`int[] previousVertex`



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

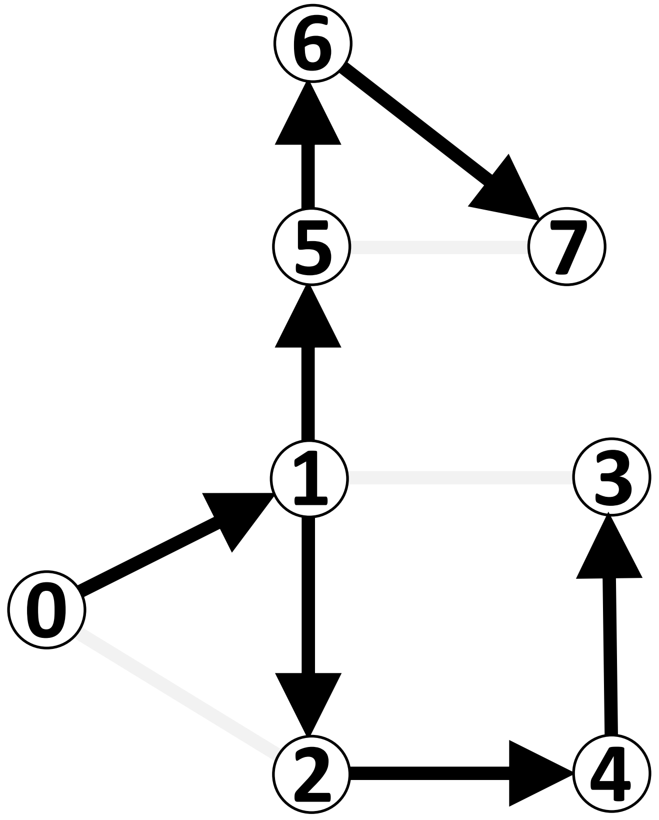


How do we determine the path from 0 to 3?

Graphs - Paths

`int[] previousVertex`

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

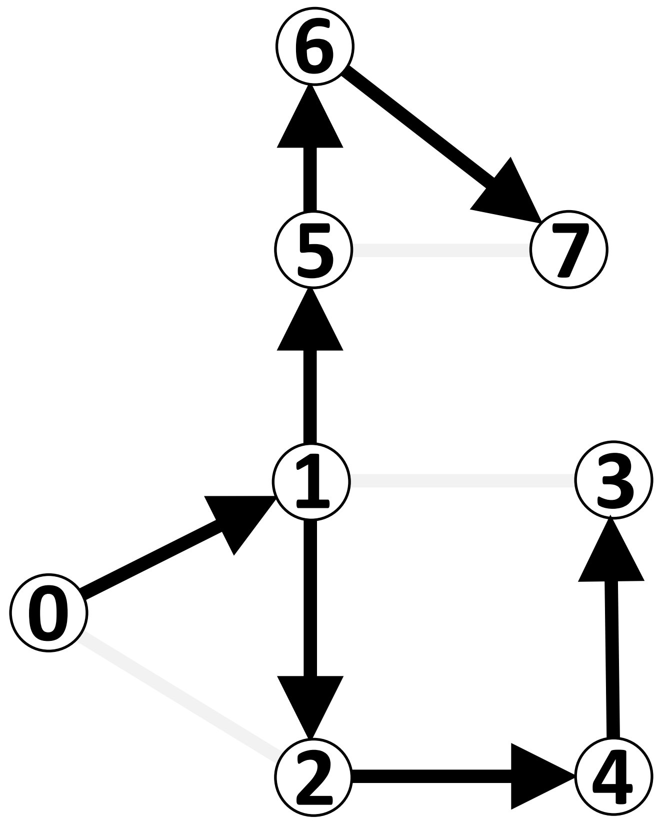


How do we determine the path from 0 to 3?

Graphs - Paths

```
int[] previousVertex
```

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

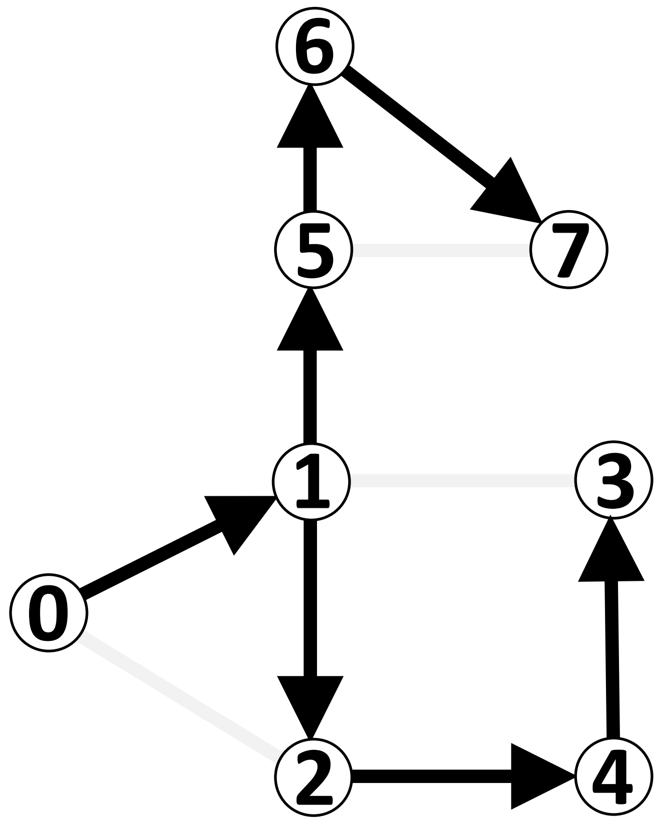


How do we determine the path from 0 to 3?

Graphs - Paths

```
int[] previousVertex
```

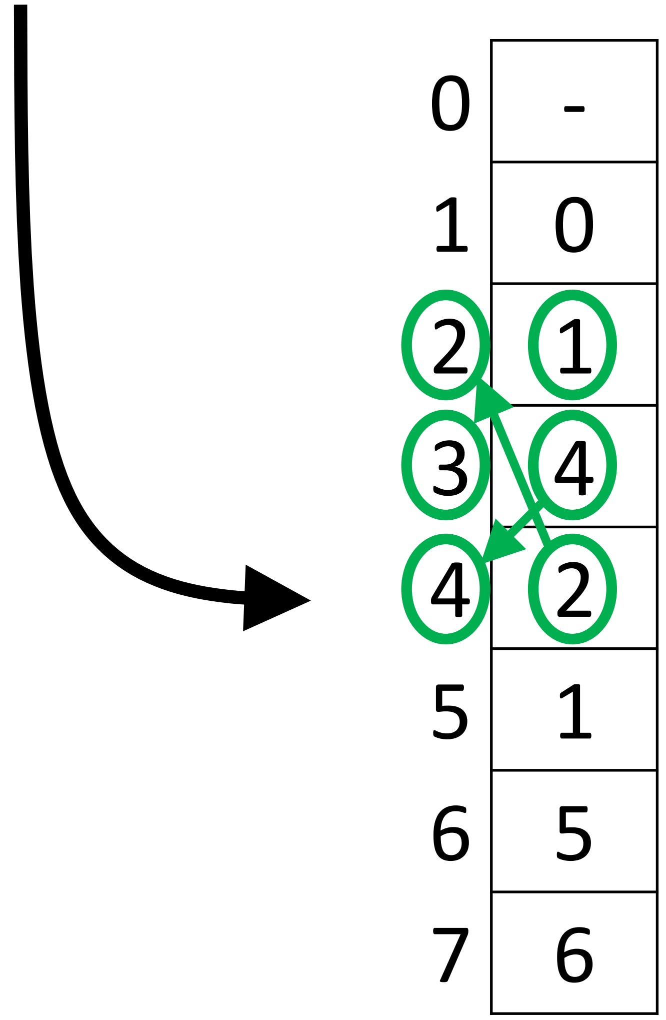
0	-
1	0
2	1
3	4
4	2
5	1
6	5
7	6



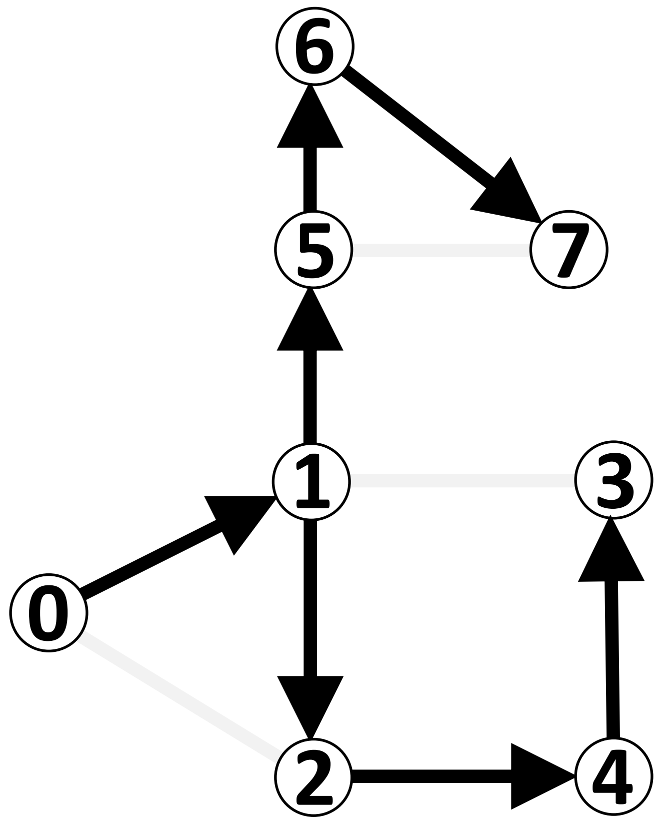
How do we determine the path from 0 to 3?

Graphs - Paths

```
int[] previousVertex
```



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

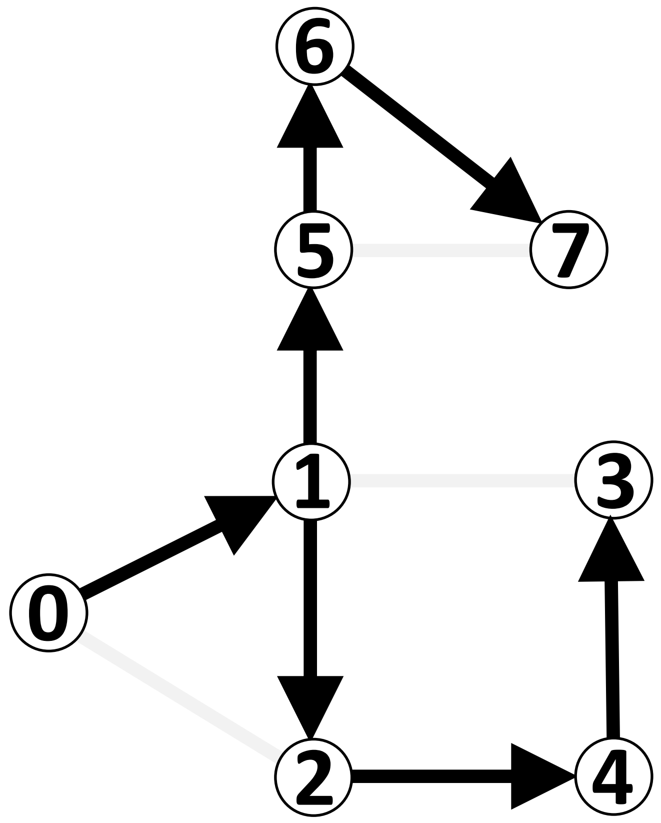


How do we determine the path from 0 to 3?

Graphs - Paths

```
int[] previousVertex
```

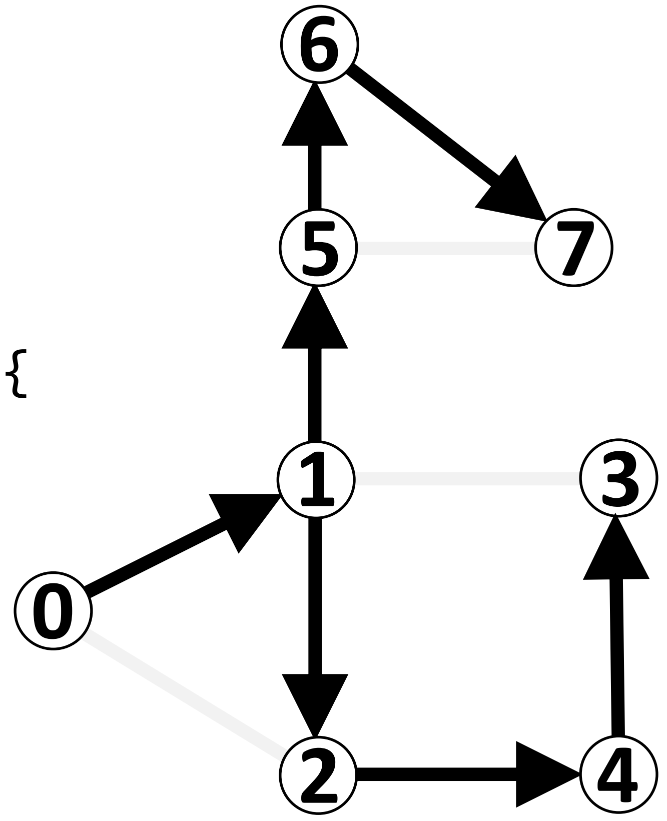
0	-
1	0
2	1
3	4
4	2
5	1
6	5
7	6



How do we determine the path from 0 to 3?

Graphs - Paths

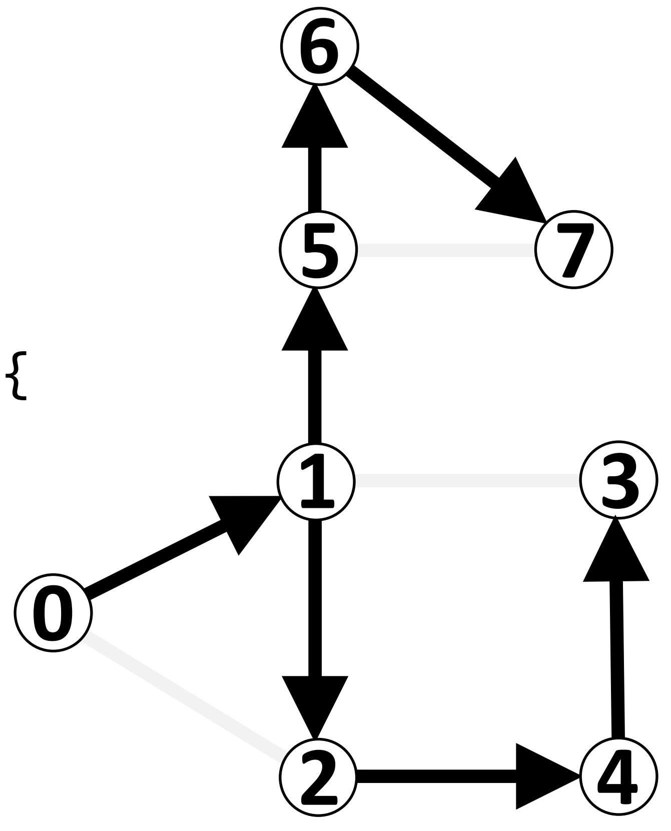
```
private boolean[] visited;

public DepthFirstSearch(Graph graph, int startVertex) {
   visited = new boolean[graph.getNumVertices()];
   dfs(graph, startVertex);
}

private void dfs(Graph graph, int vertex) {
   visited[vertex] = true;

   for (int neighbor : graph.getNeighbors(vertex)) {
      if (!visited[neighbor]) {
         dfs(graph, neighbor);
      }
   }
}
```



What do we need to do in the code?

Graphs - Paths

```
private boolean[] visited;

public DepthFirstSearch(Graph graph, int startVertex) {
   visited = new boolean[graph.getNumVertices()];
   dfs(graph, startVertex);
}

private void dfs(Graph graph, int vertex) {
   visited[vertex] = true;

   for (int neighbor : graph.getNeighbors(vertex)) {
      if (!visited[neighbor]) {
         dfs(graph, neighbor);
      }
   }
}
```



What do we need to do in the code?

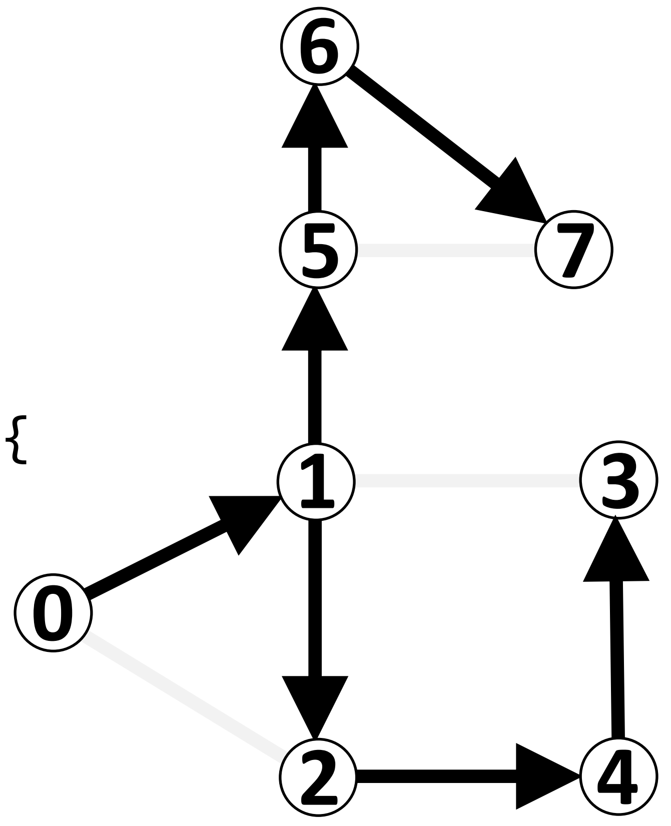
1 – Create/Initialize previousVertex

2 – Populate previousVertex

3 – getPathTo(int endVertex)

Graphs - Paths

```
private boolean[] visited;
private int[] previousVertex;

public DepthFirstSearch(Graph graph, int startVertex) {
   visited = new boolean[graph.getNumVertices()];
   previousVertex = new int[graph.getNumVertices()];
   dfs(graph, startVertex);
}

private void dfs(Graph graph, int vertex) {
   visited[vertex] = true;

   for (int neighbor : graph.getNeighbors(vertex)) {
      if (!visited[neighbor]) {
         dfs(graph, neighbor);
      }
   }
}
```



What do we need to do in the code?

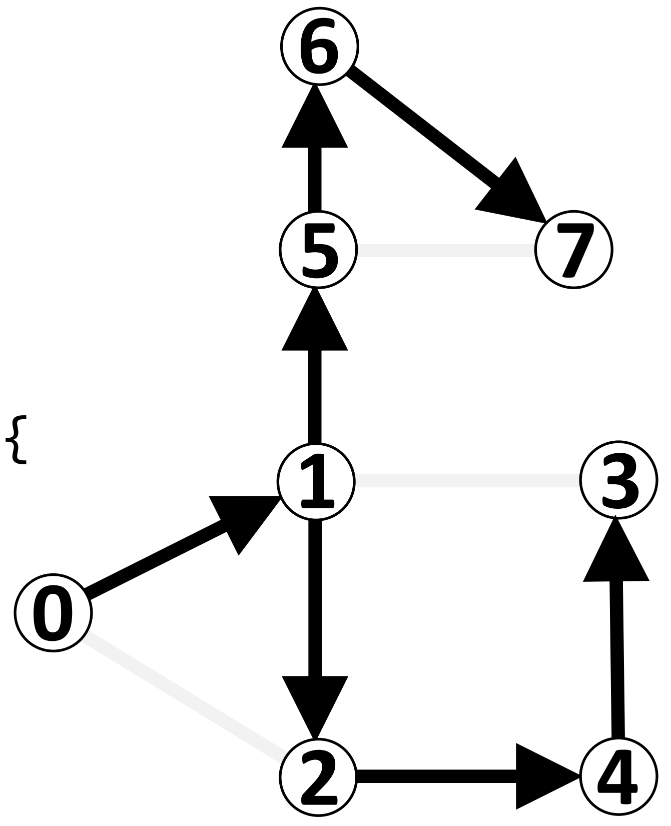
1 – Create/Initialize previousVertex

2 – Populate previousVertex

3 – getPathTo(int endVertex)

Graphs - Paths

```
private boolean[] visited;
private int[] previousVertex;

public DepthFirstSearch(Graph graph, int startVertex) {
   visited = new boolean[graph.getNumVertices()];
   previousVertex = new int[graph.getNumVertices()];
   dfs(graph, startVertex);
}

private void dfs(Graph graph, int vertex) {
   visited[vertex] = true;

   for (int neighbor : graph.getNeighbors(vertex)) {
      if (!visited[neighbor]) {
         previousVertex[?????] = ?????;
         dfs(graph, neighbor);
      }
   }
}
```



What do we need to do in the code?

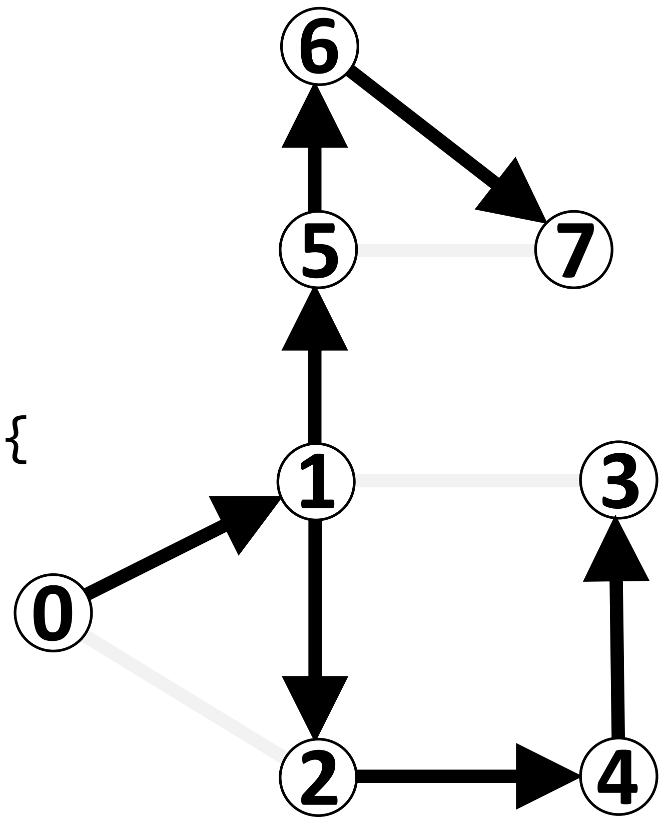
1 – Create/Initialize previousVertex

2 – Populate previousVertex

3 – getPathTo(int endVertex)

Graphs - Paths

```
private boolean[] visited;
private int[] previousVertex;

public DepthFirstSearch(Graph graph, int startVertex) {
   visited = new boolean[graph.getNumVertices()];
   previousVertex = new int[graph.getNumVertices()];
   dfs(graph, startVertex);
}

private void dfs(Graph graph, int vertex) {
   visited[vertex] = true;

   for (int neighbor : graph.getNeighbors(vertex)) {
      if (!visited[neighbor]) {
         previousVertex[vertex] = neighbor;
         dfs(graph, neighbor);
      }
   }
}
```



What do we need to do in the code?

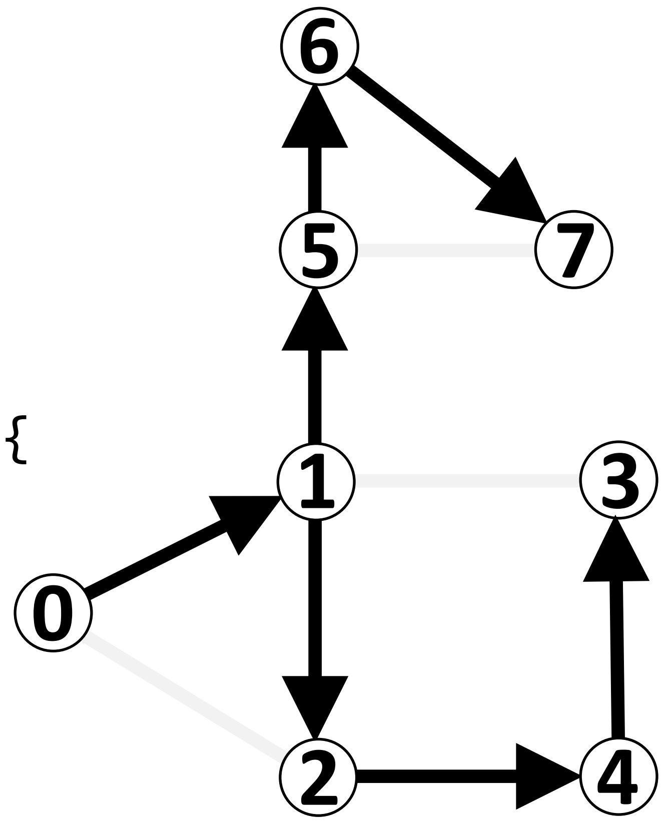
1 – Create/Initialize previousVertex

2 – Populate previousVertex

3 – getPathTo(int endVertex)

Graphs - Paths

```
private boolean[] visited;
private int[] previousVertex;

public DepthFirstSearch(Graph graph, int startVertex) {
   visited = new boolean[graph.getNumVertices()];
   previousVertex = new int[graph.getNumVertices()];
   dfs(graph, startVertex);
}

private void dfs(Graph graph, int vertex) {
   visited[vertex] = true;

   for (int neighbor : graph.getNeighbors(vertex)) {
      if (!visited[neighbor]) {
         previousVertex[neighbor] = vertex;
         dfs(graph, neighbor);
      }
   }
}
```



What do we need to do in the code?

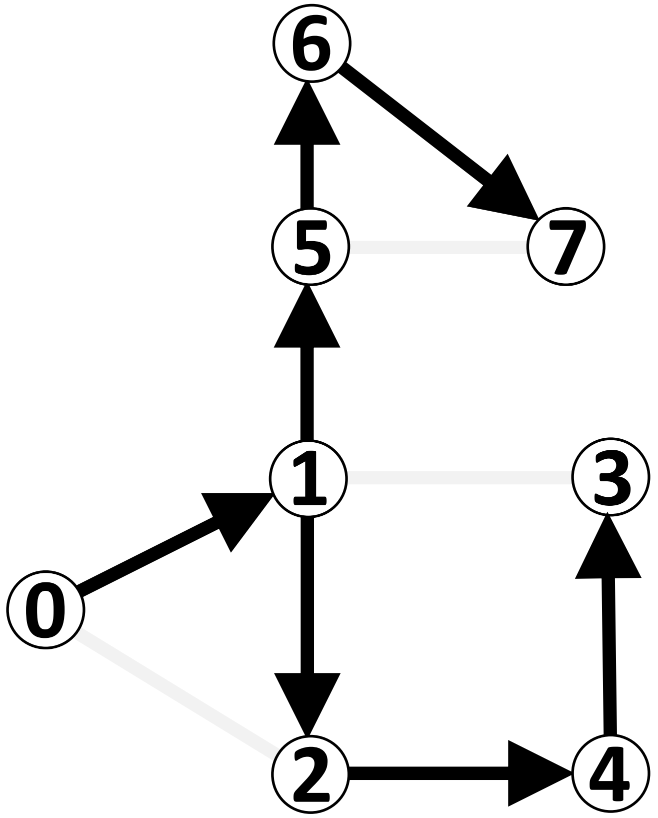
1 – Create/Initialize previousVertex

2 – Populate previousVertex

3 – getPathTo(int endVertex)

Graphs - Paths

```
public         ???          getPathTo(int endVertex) {



}
```



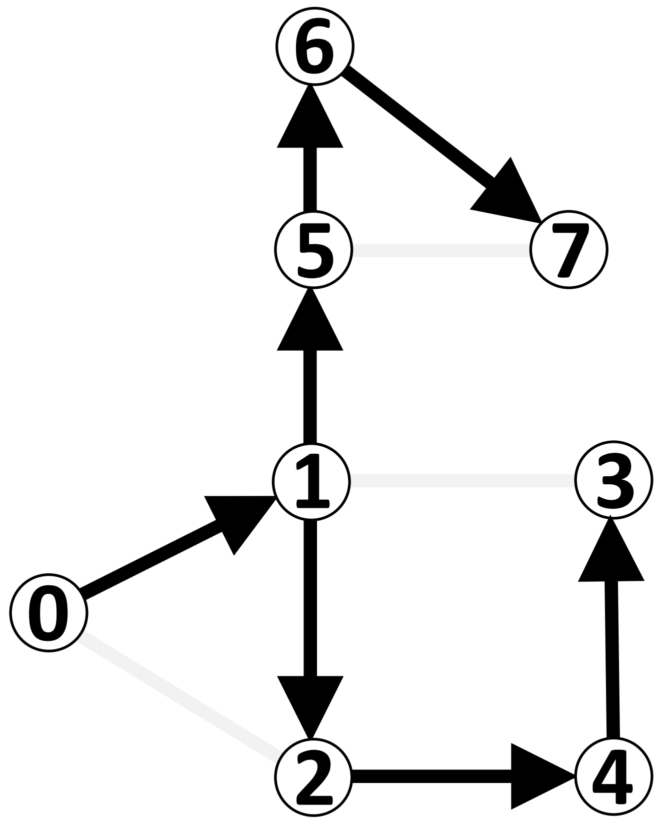
What do we need to do in the code?

1 – Create/Initialize previousVertex

2 – Populate previousVertex

3 – getPathTo(int endVertex)

Graphs - Paths



```
public         ???          getPathTo(int endVertex) {
   if (!reachable(endVertex)) {

   } else {







   }
}
```

What do we need to do in the code?

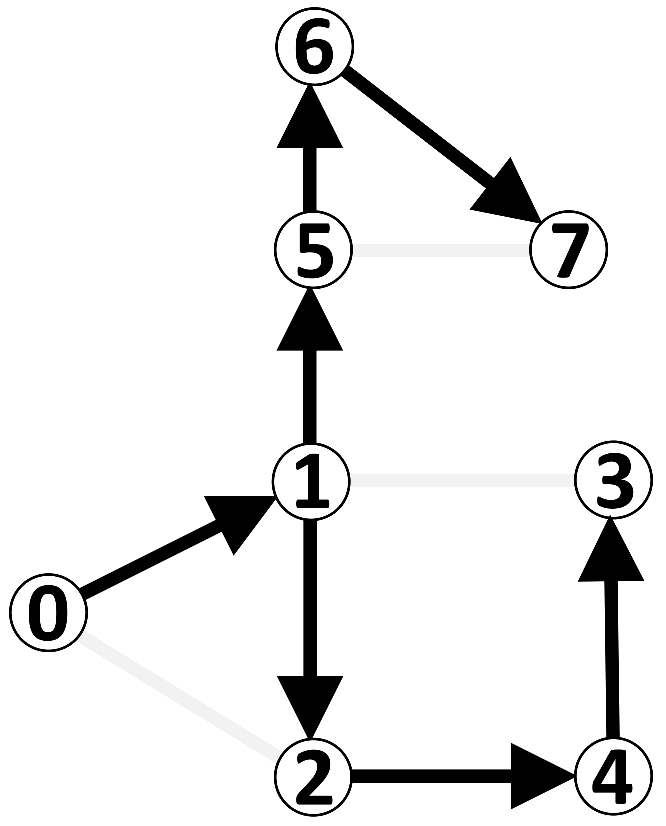
1 – Create/Initialize previousVertex

2 – Populate previousVertex

3 – getPathTo(int endVertex)

Graphs - Paths

```
public         ???          getPathTo(int endVertex) {
   if (!reachable(endVertex)) {
      return null;
   } else {







   }
}
```



What do we need to do in the code?

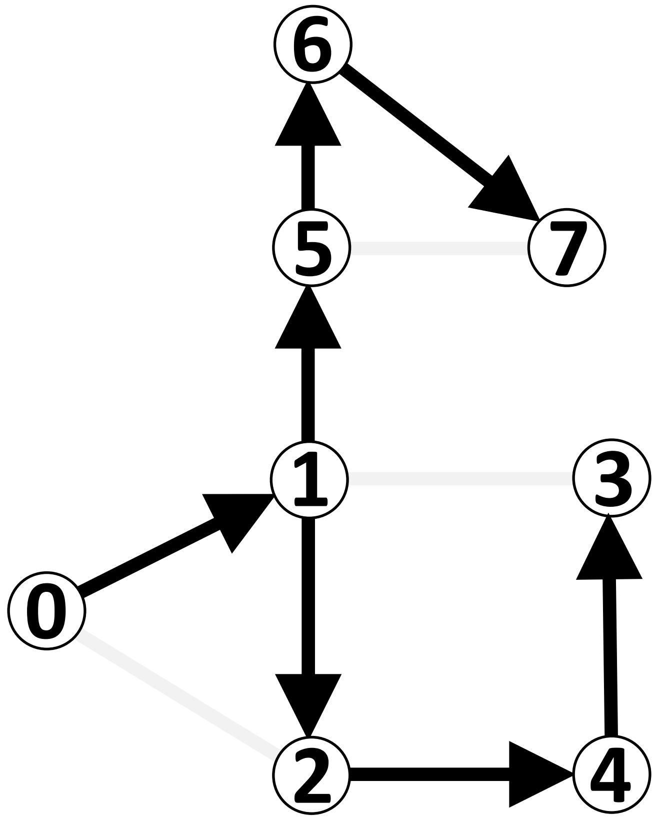
1 – Create/Initialize previousVertex

2 – Populate previousVertex

3 – getPathTo(int endVertex)

Graphs - Paths

```
public         ???          getPathTo(int endVertex) {
   if (!reachable(endVertex)) {
      return null;
   } else {
                     ??    path = ??




   }
}
```



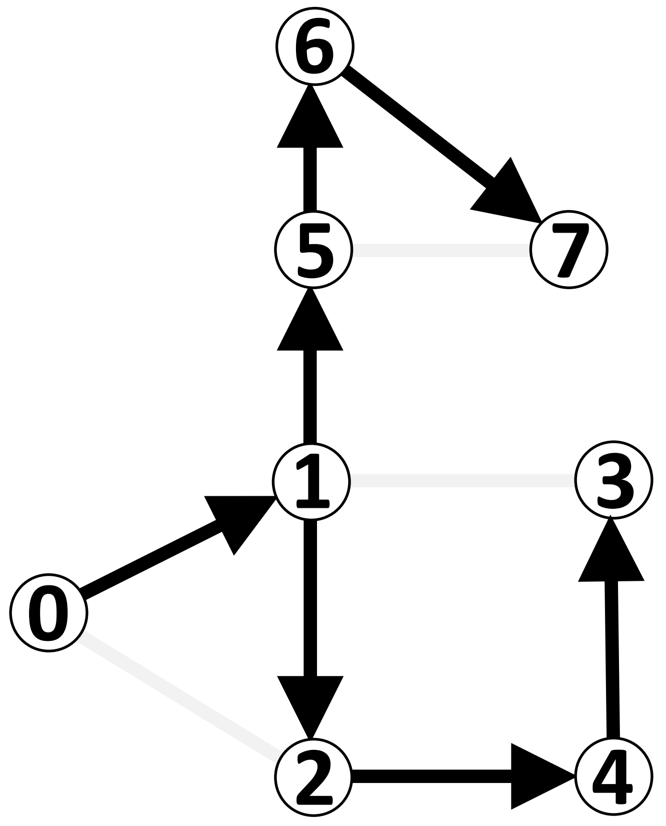
What do we need to do in the code?

1 – Create/Initialize previousVertex

2 – Populate previousVertex

3 – getPathTo(int endVertex)

Graphs - Paths



```
public          ???          getPathTo(int endVertex) {
   if (!reachable(endVertex)) {
      return null;
   } else {
      LinkedList<Integer> path = new LinkedList<>();





   }
}
```

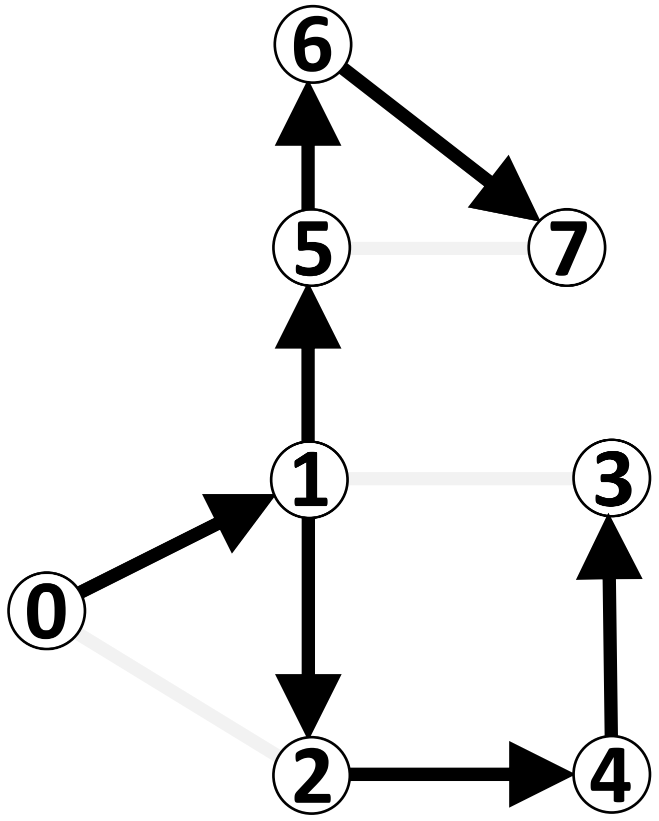
What do we need to do in the code?

1 – Create/Initialize previousVertex

2 – Populate previousVertex

3 – getPathTo(int endVertex)

Graphs - Paths



```
public LinkedList<Integer> getPathTo(int endVertex) {
   if (!reachable(endVertex)) {
      return null;
   } else {
      LinkedList<Integer> path = new LinkedList<>();






   }
}
```

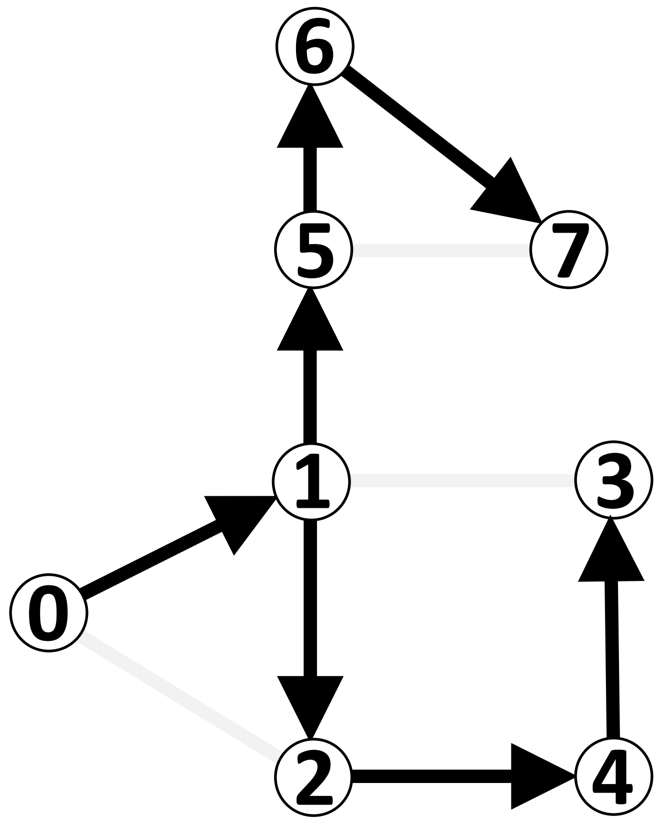
What do we need to do in the code?

1 – Create/Initialize previousVertex

2 – Populate previousVertex

3 – getPathTo(int endVertex)

Graphs - Paths



```
public LinkedList<Integer> getPathTo(int endVertex) {
   if (!reachable(endVertex)) {
      return null;
   } else {
      LinkedList<Integer> path = new LinkedList<>();
      for (int v = ??       ;                 ;                      ) {

      }


   }
}
```

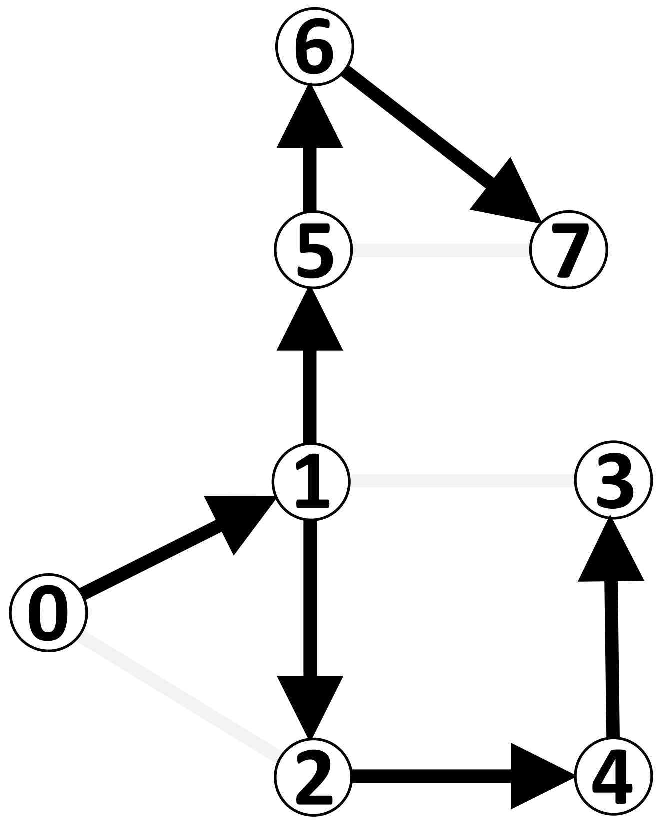
What do we need to do in the code?

1 – Create/Initialize previousVertex

2 – Populate previousVertex

3 – getPathTo(int endVertex)

Graphs - Paths



```
public LinkedList<Integer> getPathTo(int endVertex) {
   if (!reachable(endVertex)) {
      return null;
   } else {
      LinkedList<Integer> path = new LinkedList<>();
      for (int v = endVertex; v ??            ;                      ) {

      }


   }
}
```

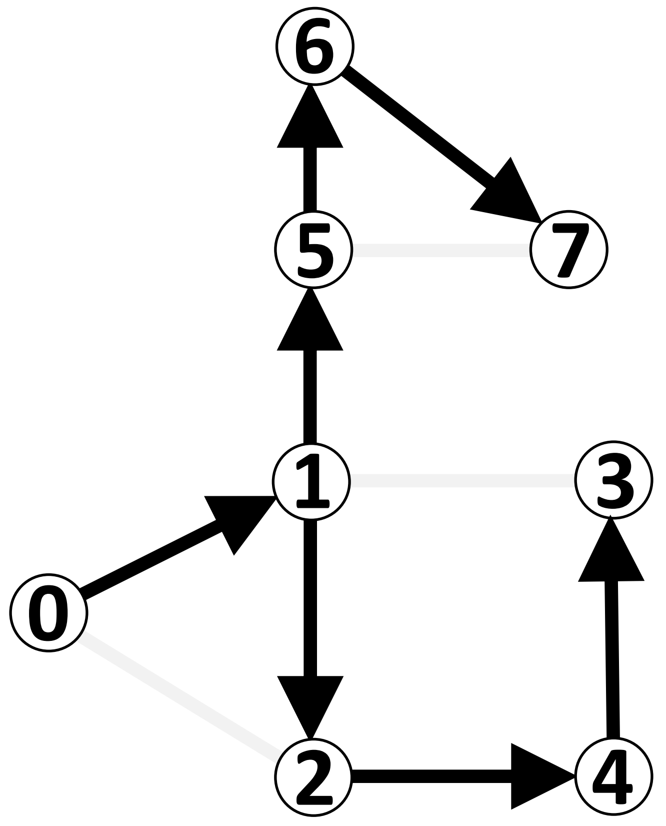
What do we need to do in the code?

1 – Create/Initialize previousVertex

2 – Populate previousVertex

3 – getPathTo(int endVertex)

Graphs - Paths



```
public LinkedList<Integer> getPathTo(int endVertex) {
   if (!reachable(endVertex)) {
      return null;
   } else {
      LinkedList<Integer> path = new LinkedList<>();
      for (int v = endVertex; v != startVertex;                   ) {

      }


   }
}
```

What do we need to do in the code?

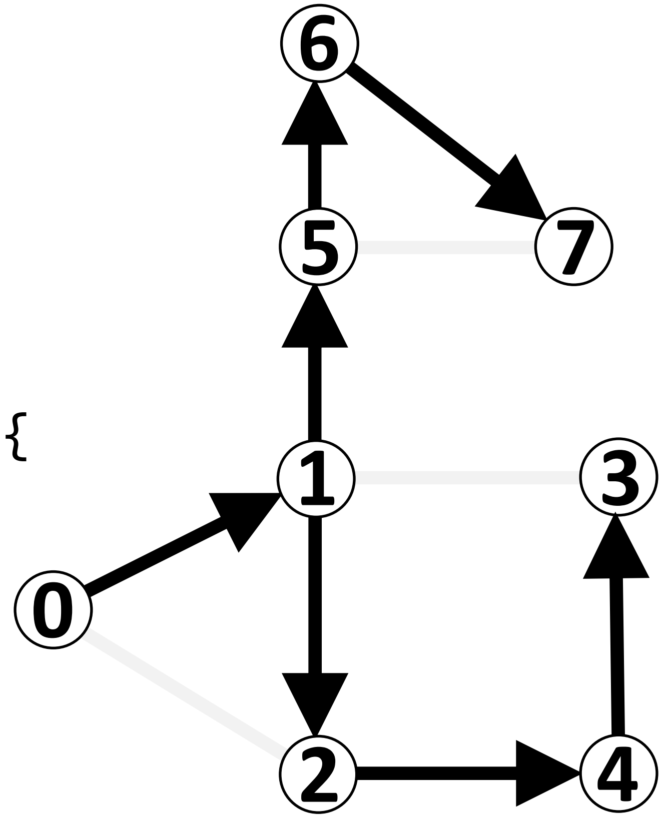
1 – Create/Initialize previousVertex

2 – Populate previousVertex

3 – getPathTo(int endVertex)

Graphs - Paths

```
private boolean[] visited;
private int[] previousVertex;

public DepthFirstSearch(Graph graph, int startVertex) {
   visited = new boolean[graph.getNumVertices()];
   previousVertex = new int[graph.getNumVertices()];
   dfs(graph, startVertex);
}

private void dfs(Graph graph, int vertex) {
   visited[vertex] = true;

   for (int neighbor : graph.getNeighbors(vertex)) {
      if (!visited[neighbor]) {
         previousVertex[neighbor] = vertex;
         dfs(graph, neighbor);
      }
   }
}
```

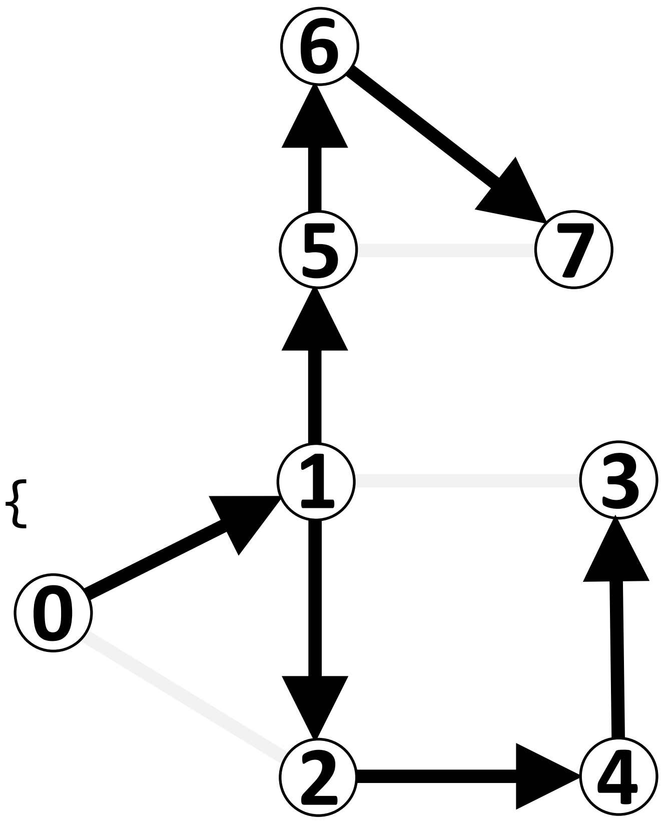


What do we need to do in the code?

- **1 – Create/Initialize previousVertex**
- **2 – Populate previousVertex**
- **3 – getPathTo(int endVertex)**

Graphs - Paths

```
private boolean[] visited;
private int[] previousVertex;
private int startVertex;

public DepthFirstSearch(Graph graph, int startVertex) {
   visited = new boolean[graph.getNumVertices()];
   previousVertex = new int[graph.getNumVertices()];
   this.startVertex = startVertex;
   dfs(graph, startVertex);
}

private void dfs(Graph graph, int vertex) {
   visited[vertex] = true;

   for (int neighbor : graph.getNeighbors(vertex)) {
      if (!visited[neighbor]) {
         previousVertex[neighbor] = vertex;
         dfs(graph, neighbor);
      }
}  }
```



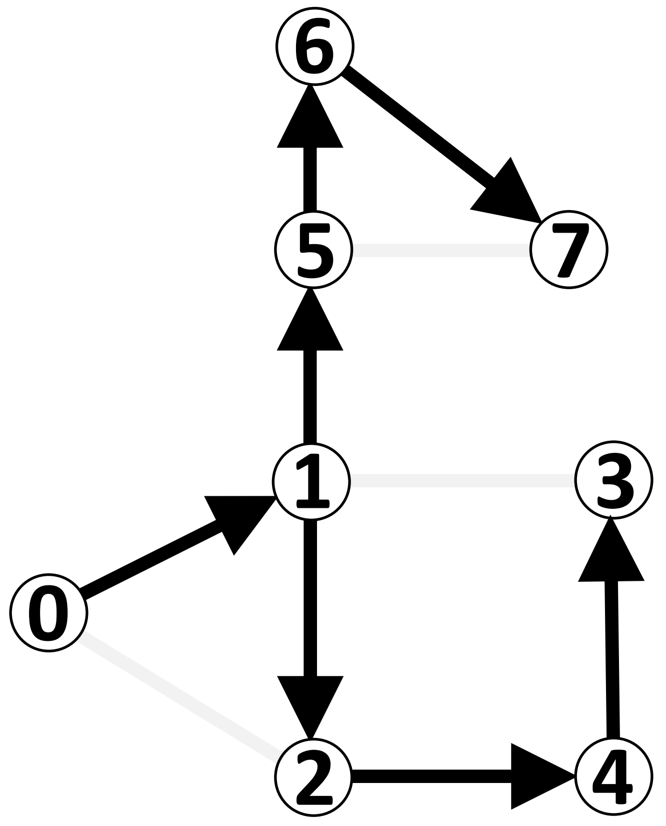
What do we need to do in the code?

1 – Create/Initialize previousVertex

2 – Populate previousVertex

3 – getPathTo(int endVertex)

Graphs - Paths



```
public LinkedList<Integer> getPathTo(int endVertex) {
   if (!reachable(endVertex)) {
      return null;
   } else {
      LinkedList<Integer> path = new LinkedList<>();
      for (int v = endVertex; v != startVertex;                    ) {

      }


   }
}
```

What do we need to do in the code?

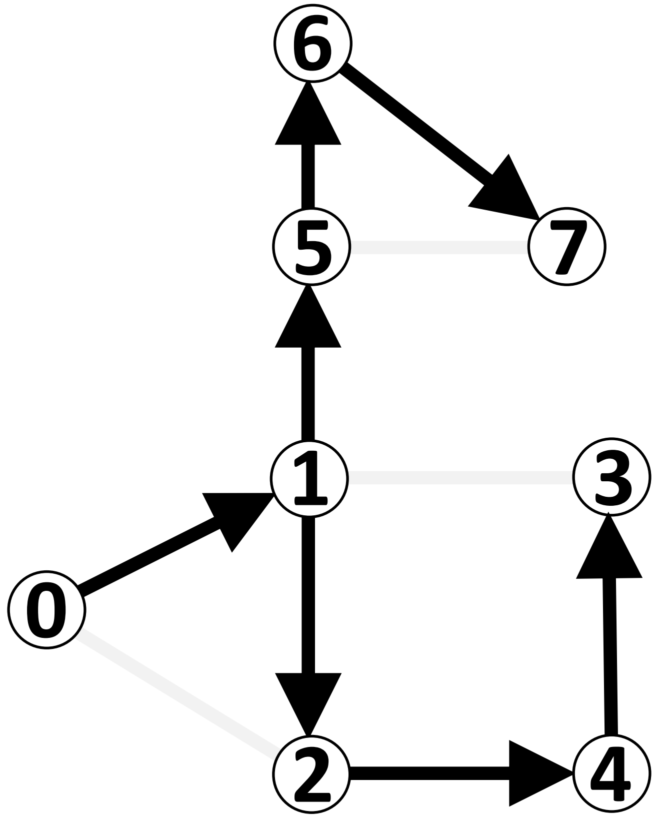
1 – Create/Initialize previousVertex

2 – Populate previousVertex

3 – getPathTo(int endVertex)

Graphs - Paths

```
private int[] previousVertex;



public LinkedList<Integer> getPathTo(int endVertex) {
   if (!reachable(endVertex)) {
      return null;
   } else {
      LinkedList<Integer> path = new LinkedList<>();
      for (int v = endVertex; v != startVertex; v ??                ) {

      }


   }
}
```



What do we need to do in the code?

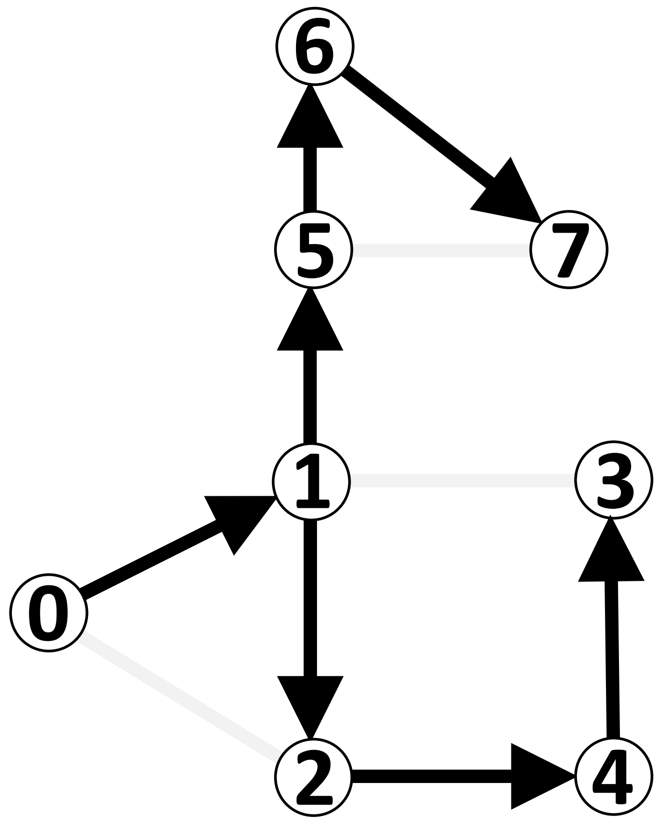
1 – Create/Initialize previousVertex

2 – Populate previousVertex

3 – getPathTo(int endVertex)

Graphs - Paths

```
private int[] previousVertex;




public LinkedList<Integer> getPathTo(int endVertex) {
   if (!reachable(endVertex)) {
      return null;
   } else {
      LinkedList<Integer> path = new LinkedList<>();
      for (int v = endVertex; v != startVertex; v = previousVertex[v]) {

      }


   }
}
```



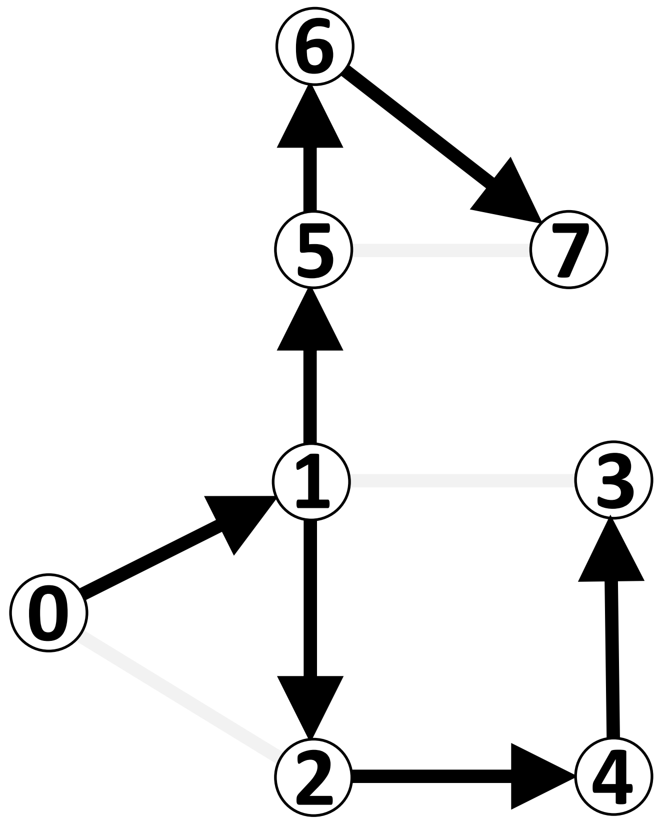
What do we need to do in the code?

1 – Create/Initialize previousVertex

2 – Populate previousVertex

3 – getPathTo(int endVertex)

Graphs - Paths



```
public LinkedList<Integer> getPathTo(int endVertex) {
   if (!reachable(endVertex)) {
      return null;
   } else {
      LinkedList<Integer> path = new LinkedList<>();
      for (int v = endVertex; v != startVertex; v = previousVertex[v]) {
         path.add(v);
      }


   }
}
```

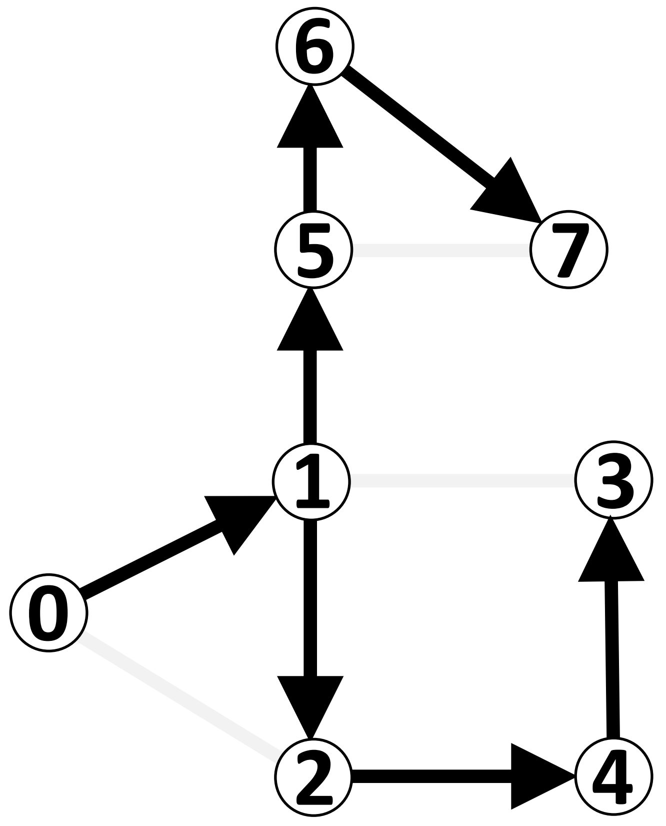
What do we need to do in the code?

1 – Create/Initialize previousVertex

2 – Populate previousVertex

3 – getPathTo(int endVertex)

Graphs - Paths



```
public LinkedList<Integer> getPathTo(int endVertex) {
   if (!reachable(endVertex)) {
      return null;
   } else {
      LinkedList<Integer> path = new LinkedList<>();
      for (int v = endVertex; v != startVertex; v = previousVertex[v]) {
         path.addFirst(v);
      }


   }
}
```

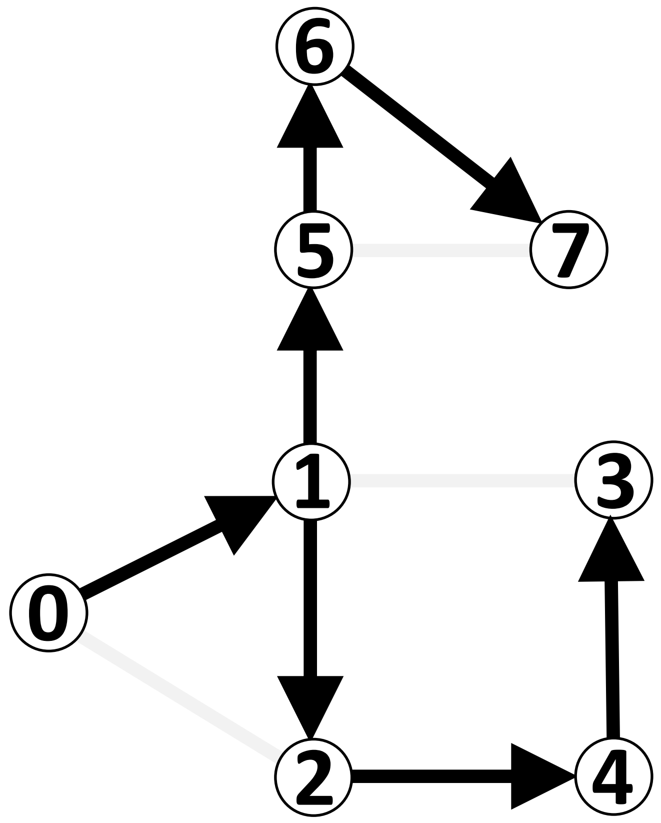
What do we need to do in the code?

1 – Create/Initialize previousVertex

2 – Populate previousVertex

3 – getPathTo(int endVertex)

Graphs - Paths



```
public LinkedList<Integer> getPathTo(int endVertex) {
    if (!reachable(endVertex)) {
        return null;
    } else {
        LinkedList<Integer> path = new LinkedList<>();
        for (int v = endVertex; v != startVertex; v = previousVertex[v]) {
            path.addFirst(v);
        }
        path.addFirst(startVertex);
        return path;
    }
}
```

What do we need to do in the code?

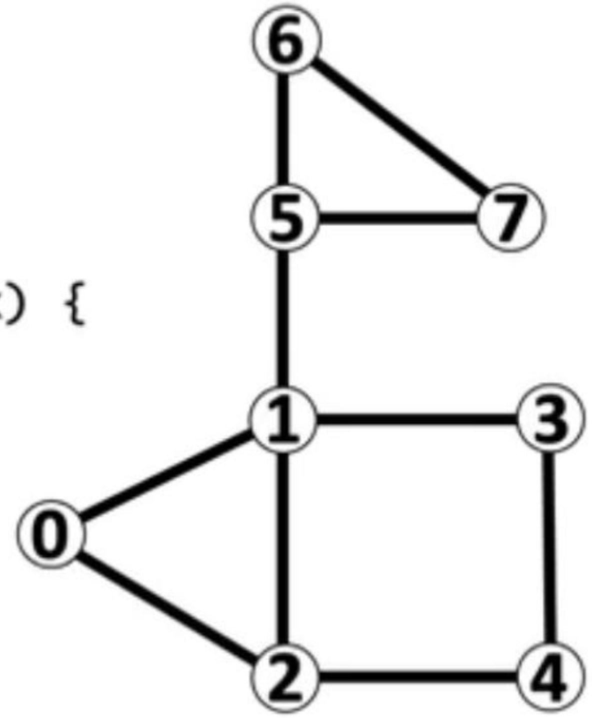
1 – Create/Initialize previousVertex

2 – Populate previousVertex

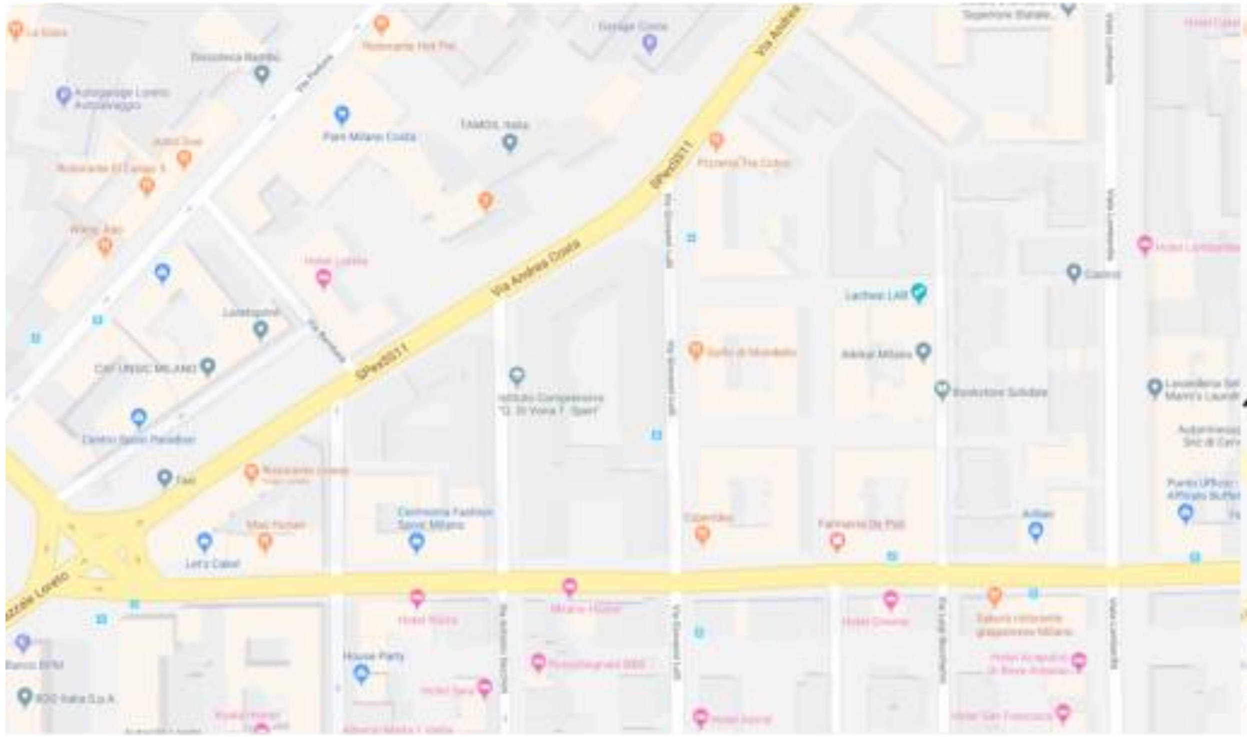
3 – getPathTo(int endVertex)

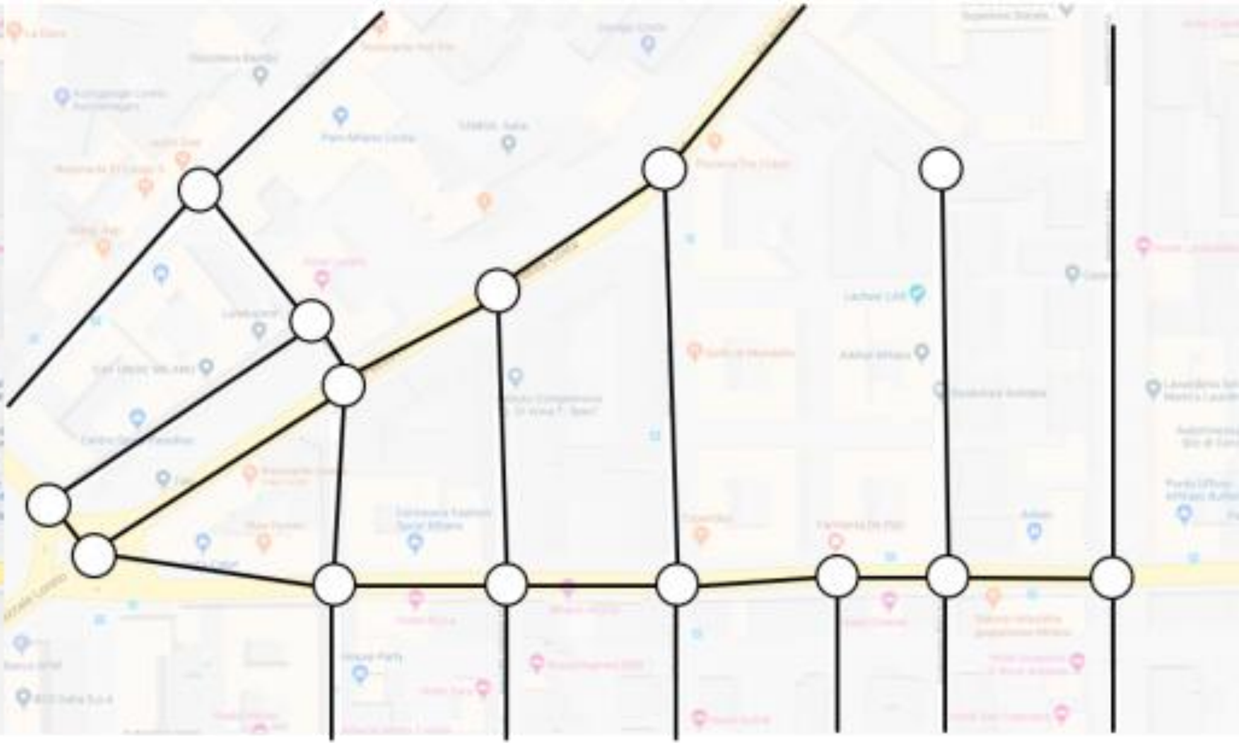
Graphs - Breadth First Search

```
private boolean[] visited;

public BreadthFirstSearch(Graph graph, int startVertex) {
   visited = new boolean[graph.getNumVertices()];
   bfs(graph, startVertex);
}

private void bfs(Graph graph, int startVertex) {
   Queue<Integer> queue = new Queue<>();
   visited[startVertex] = true;
   queue.enqueue(startVertex);

   while (!queue.isEmpty()) {
      int vertex = queue.dequeue();
      for (int neighbor : graph.getNeighbors(vertex)) {
         if (!visited[neighbor]) {
            visited[neighbor] = true;
            queue.enqueue(neighbor);
         }
}  }  }
```

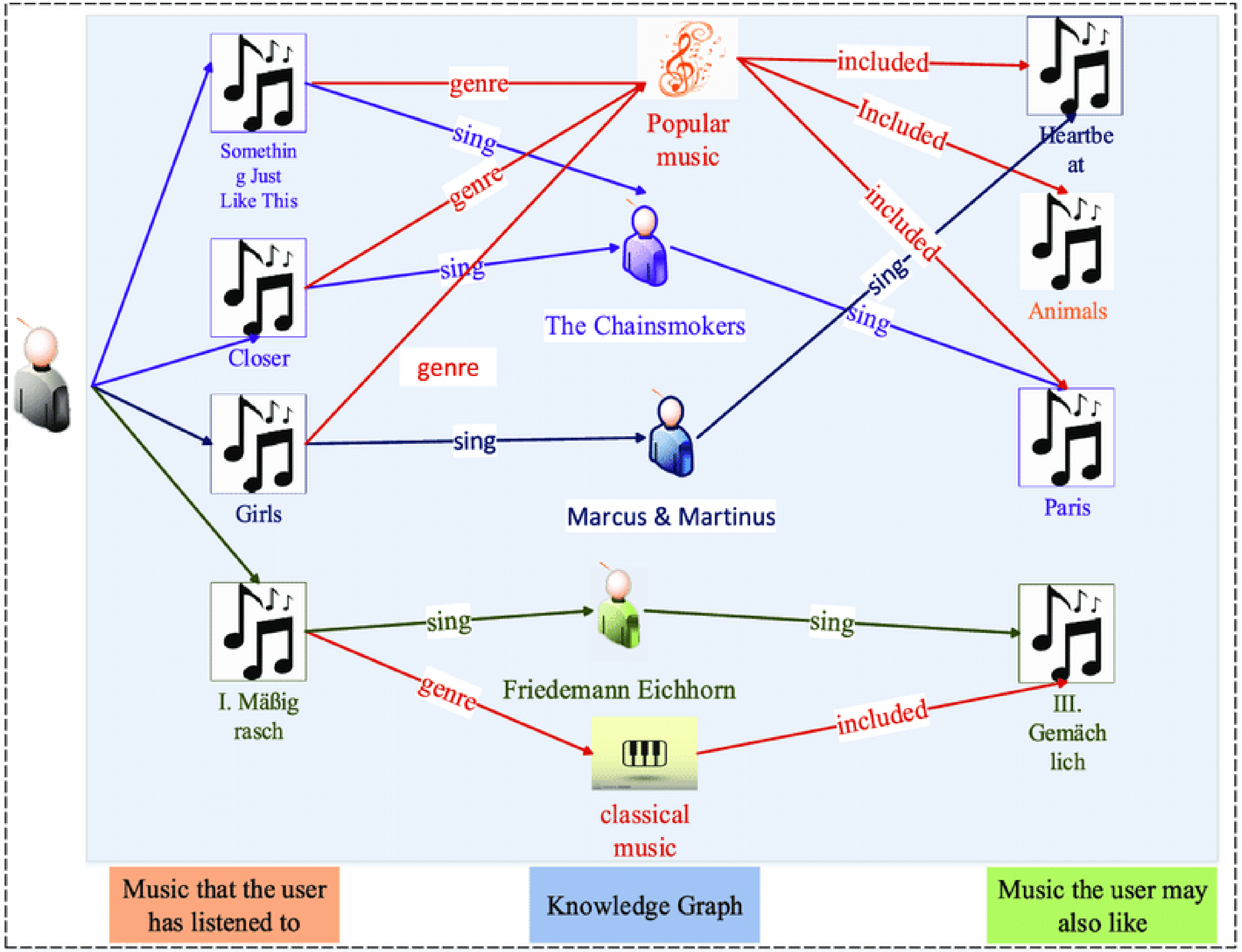


Graphs – More Applications

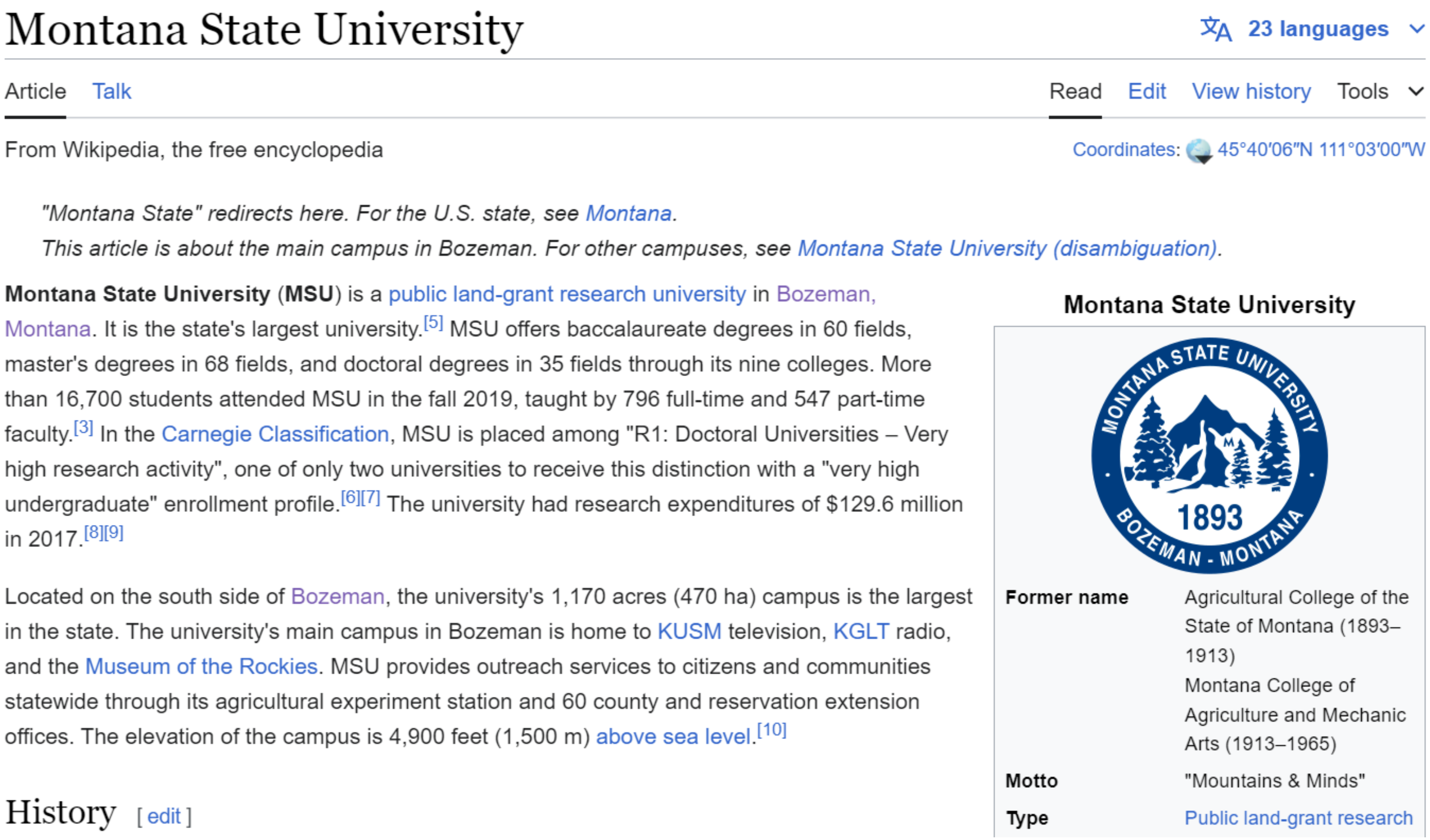




Graphs – More Applications



Graphs – More Applications



Montana State University

23 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

Coordinates: 45°40′06″N 111°03′00″W

"Montana State" redirects here. For the U.S. state, see Montana.

This article is about the main campus in Bozeman. For other campuses, see Montana State University (disambiguation).

Montana State University (**MSU**) is a public land-grant research university in Bozeman, Montana. It is the state's largest university.[5] MSU offers baccalaureate degrees in 60 fields, master's degrees in 68 fields, and doctoral degrees in 35 fields through its nine colleges. More than 16,700 students attended MSU in the fall 2019, taught by 796 full-time and 547 part-time faculty.[3] In the Carnegie Classification, MSU is placed among "R1: Doctoral Universities – Very high research activity", one of only two universities to receive this distinction with a "very high undergraduate" enrollment profile.[6][7] The university had research expenditures of $129.6 million in 2017.[8][9]

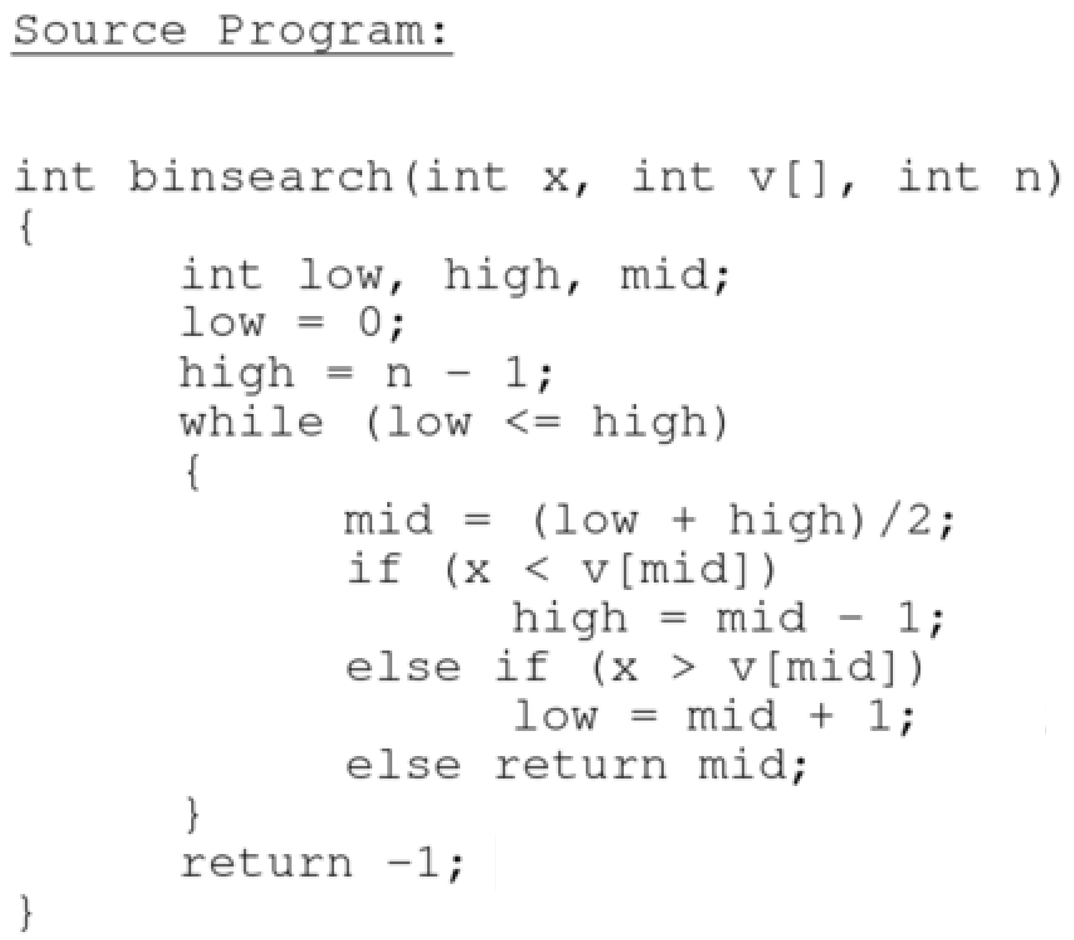
Located on the south side of Bozeman, the university's 1,170 acres (470 ha) campus is the largest in the state. The university's main campus in Bozeman is home to KUSM television, KGLT radio, and the Museum of the Rockies. MSU provides outreach services to citizens and communities statewide through its agricultural experiment station and 60 county and reservation extension offices. The elevation of the campus is 4,900 feet (1,500 m) above sea level.[10]

History [edit]

Montana State University

Former name	Agricultural College of the State of Montana (1893–1913) Montana College of Agriculture and Mechanic Arts (1913–1965)
Motto	"Mountains & Minds"
Type	Public land-grant research

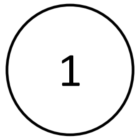
Graphs – More Applications

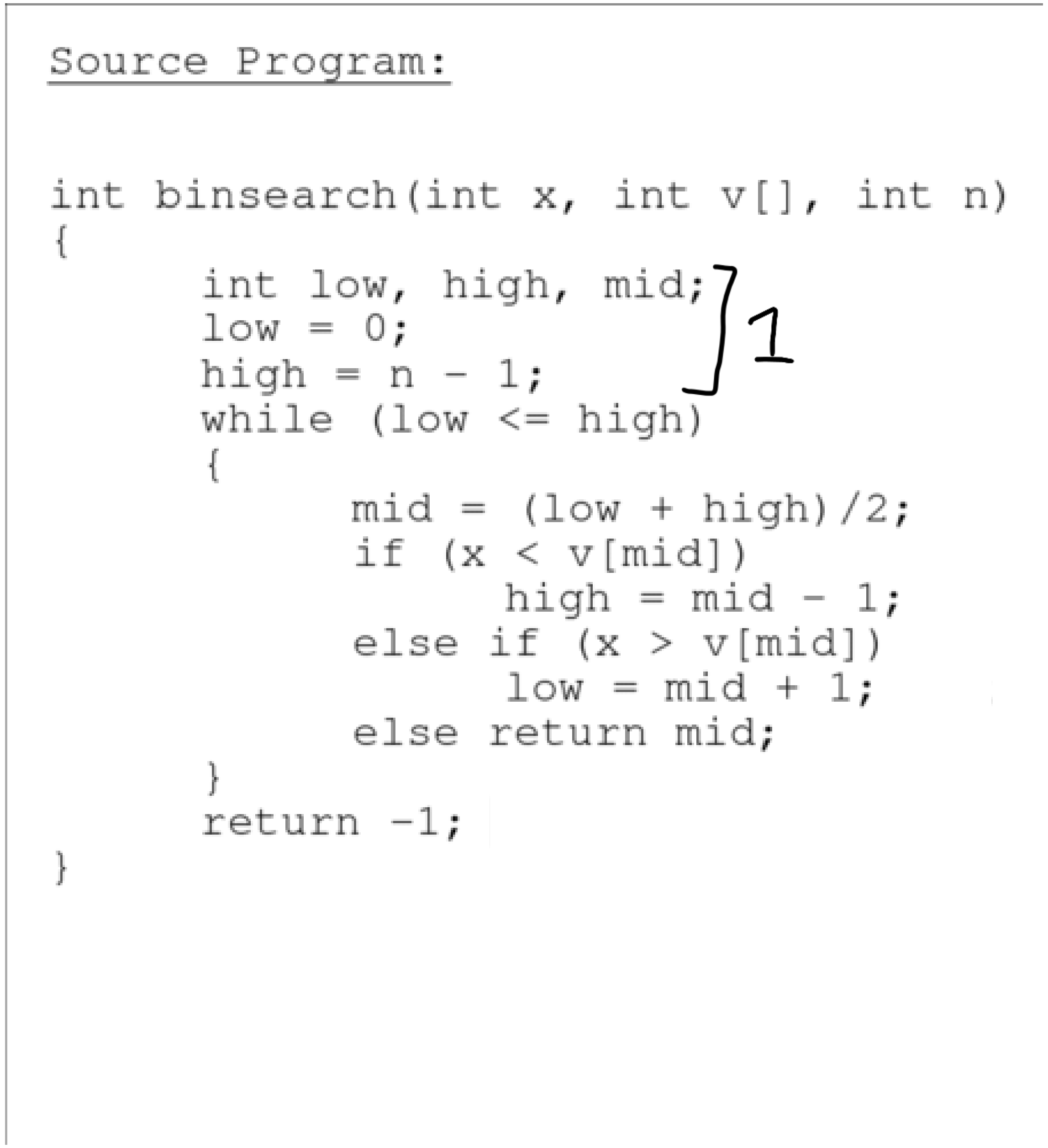


Source Program:

```
int binsearch(int x, int v[], int n)
{
       int low, high, mid;
       low = 0;
       high = n - 1;
       while (low <= high)
       {
              mid = (low + high)/2;
              if (x < v[mid])
                     high = mid - 1;
              else if (x > v[mid])
                     low = mid + 1;
              else return mid;
       }
       return -1;
}
```

Graphs – More Applications

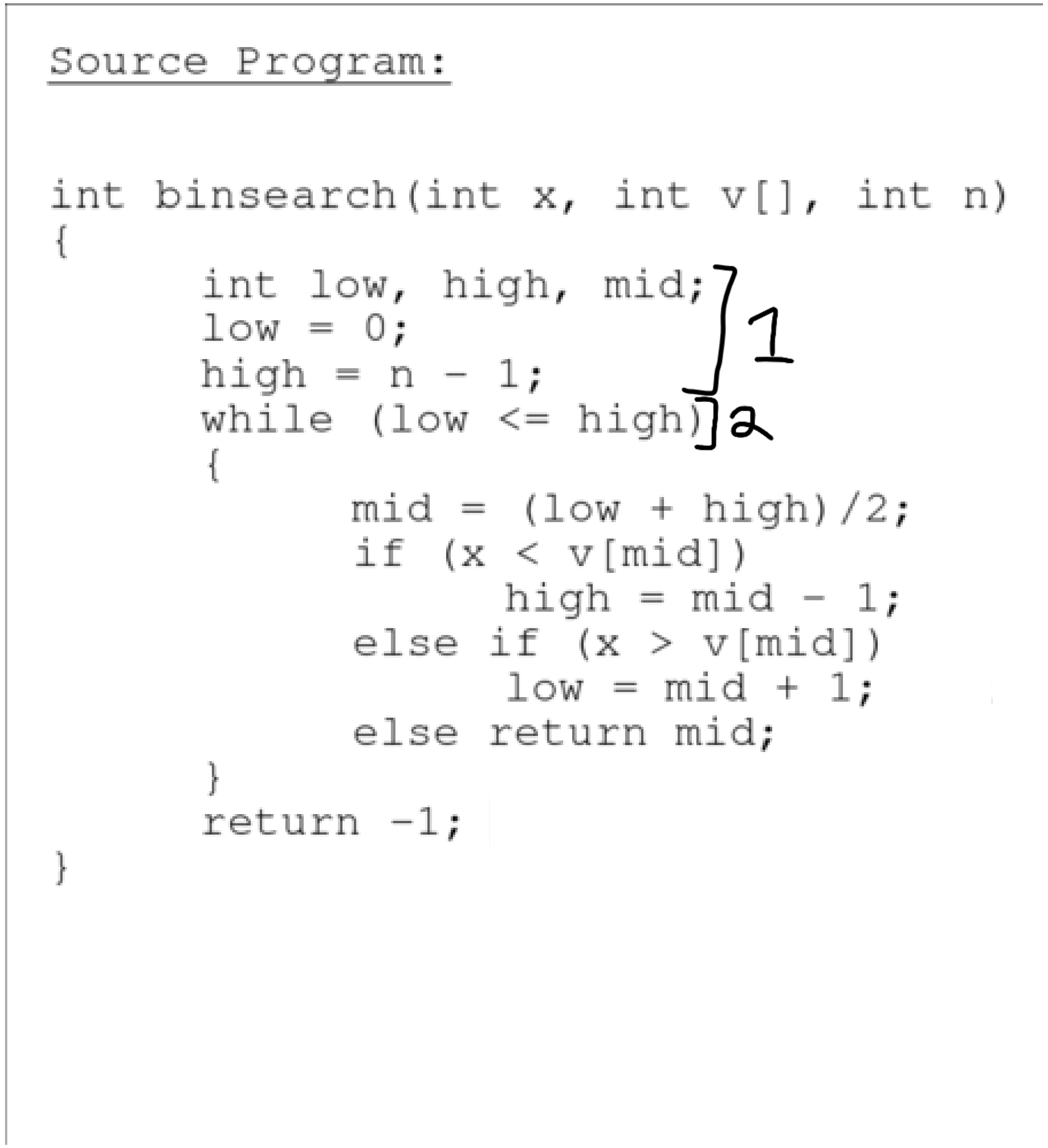




Source Program:

```
int binsearch(int x, int v[], int n)
{
      int low, high, mid;  ⎤
      low = 0;             ⎥ 1
      high = n - 1;        ⎦
      while (low <= high)
      {
            mid = (low + high)/2;
            if (x < v[mid])
                  high = mid - 1;
            else if (x > v[mid])
                  low = mid + 1;
            else return mid;
      }
      return -1;
}
```

Graphs – More Applications



Source Program:

```
int binsearch(int x, int v[], int n)
{
        int low, high, mid;
        low = 0;              1
        high = n - 1;
        while (low <= high)   2
        {
              mid = (low + high)/2;
              if (x < v[mid])
                    high = mid - 1;
              else if (x > v[mid])
                    low = mid + 1;
              else return mid;
        }
        return -1;
}
```

$$\begin{tikzcd}
1 \arrow[d] \\
2
\end{tikzcd}$$

Graphs – More Applications

Source Program:

```
int binsearch(int x, int v[], int n)
{
       int low, high, mid;     ]
       low = 0;                ] 1
       high = n - 1;           ]
       while (low <= high)     ] 2
       {
             mid = (low + high)/2;  ] 3
             if (x < v[mid])        ]
                   high = mid - 1;
             else if (x > v[mid])
                   low = mid + 1;
             else return mid;
       }
       return -1;
}
```

$$\begin{tikzcd}
1 \arrow[d] & \\
2 \arrow[dr] & \\
& 3
\end{tikzcd}$$

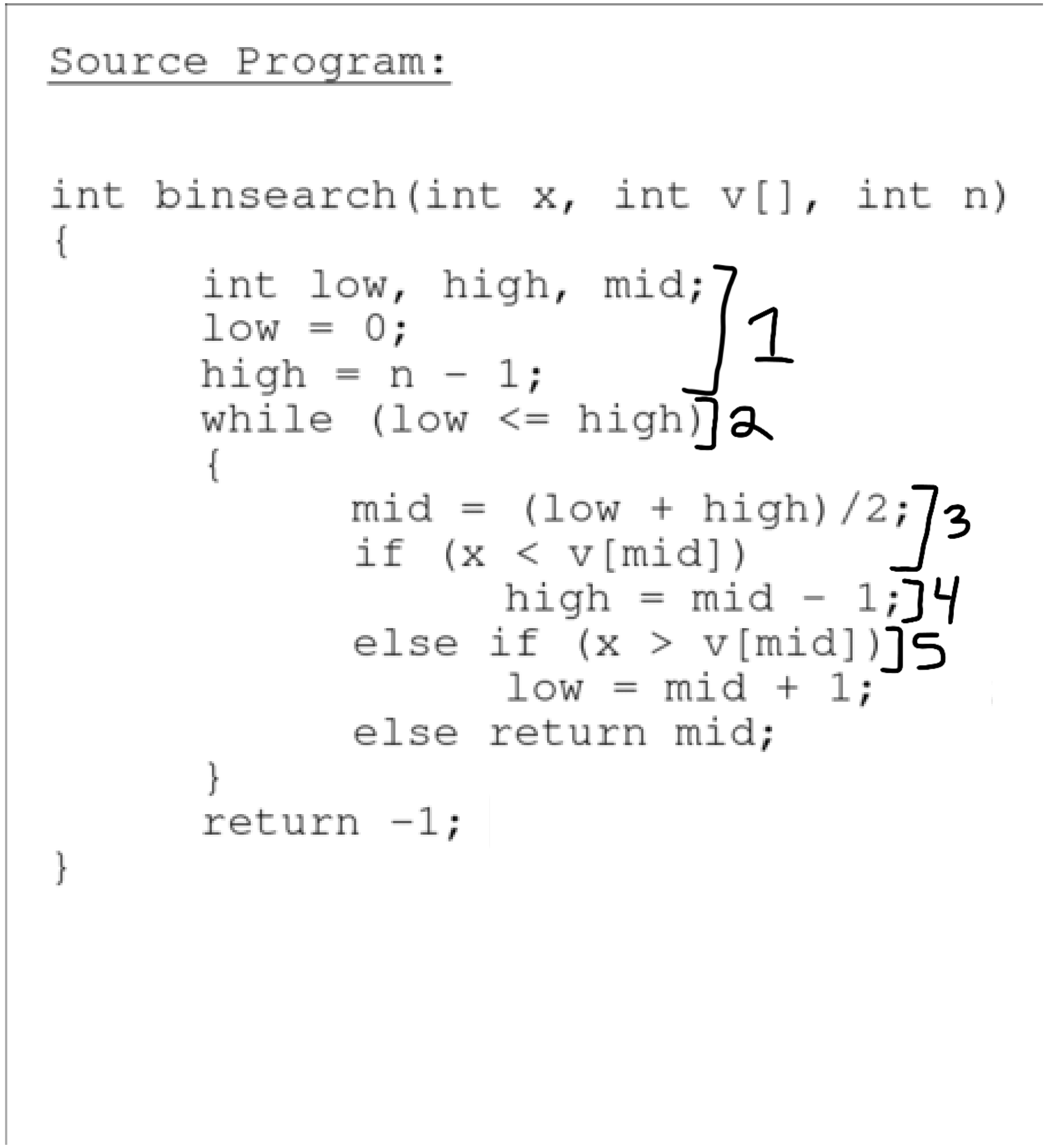
Graphs – More Applications

Source Program:

```
int binsearch(int x, int v[], int n)
{
        int low, high, mid;   ]
        low = 0;              ] 1
        high = n - 1;         ]
        while (low <= high)   ] 2
        {
              mid = (low + high)/2;  ] 3
              if (x < v[mid])        ]
                    high = mid - 1;  ] 4
              else if (x > v[mid])
                    low = mid + 1;
              else return mid;
        }
        return -1;
}
```

$$\begin{tikzcd}
1 \arrow[d] & \\
2 \arrow[dr] & \\
& 3 \arrow[dl] \\
4 \arrow[uu] &
\end{tikzcd}$$

Graphs – More Applications



Source Program:

```
int binsearch(int x, int v[], int n)
{
       int low, high, mid;
       low = 0;              1
       high = n - 1;
       while (low <= high)  2
       {
             mid = (low + high)/2;  3
             if (x < v[mid])
                   high = mid - 1;  4
             else if (x > v[mid])   5
                   low = mid + 1;
             else return mid;
       }
       return -1;
}
```

$$\begin{tikzcd}
1 \arrow[d] & & \\
2 \arrow[dr] & & \\
& 3 \arrow[dl] \arrow[dr] & \\
4 \arrow[uu] & & 5
\end{tikzcd}$$

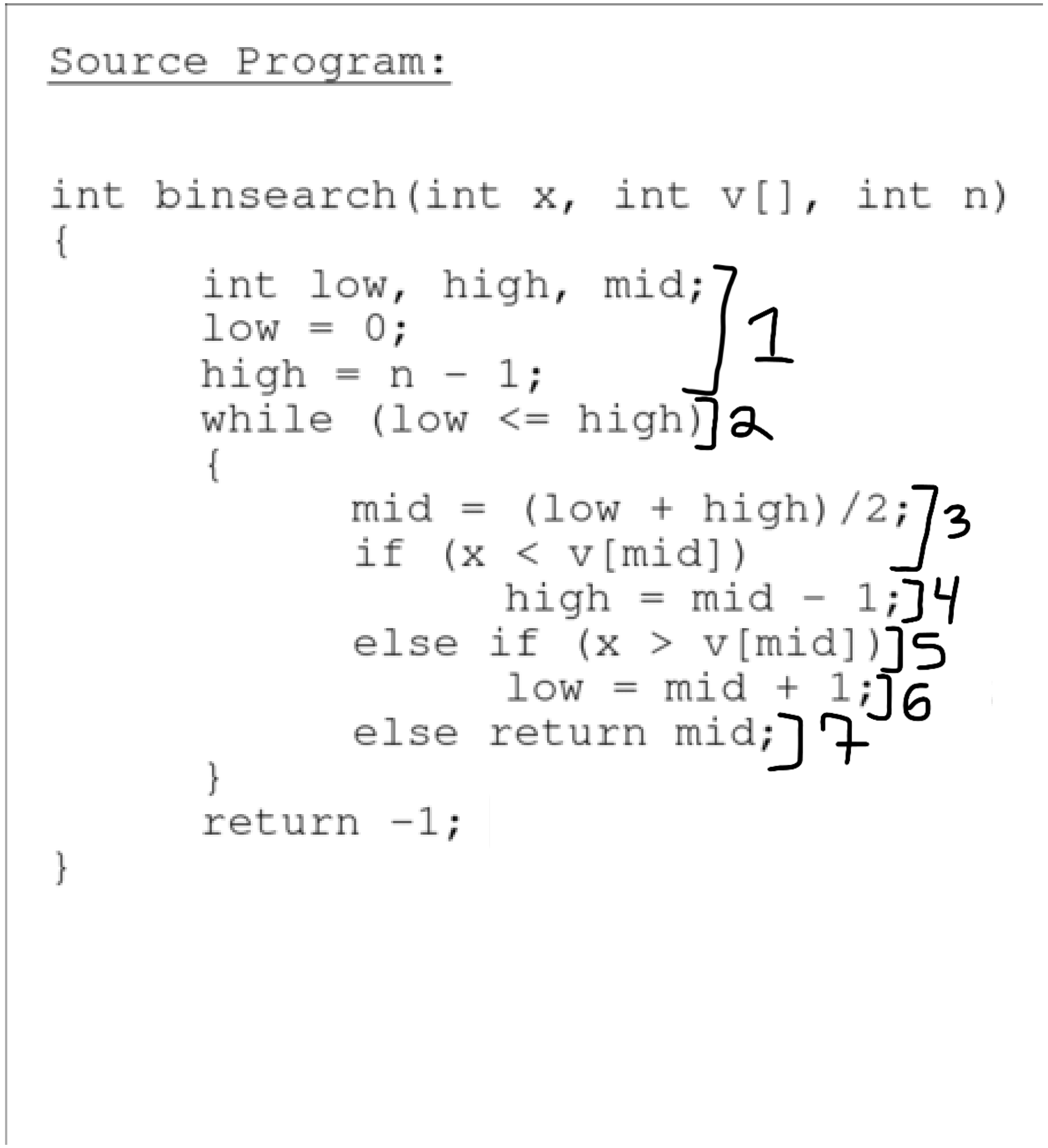
Graphs – More Applications

Source Program:

```
int binsearch(int x, int v[], int n)
{
       int low, high, mid;        ]
       low = 0;                   ]1
       high = n - 1;              ]
       while (low <= high)]2
       {
             mid = (low + high)/2;]3
             if (x < v[mid])      ]
                  high = mid - 1;]4
             else if (x > v[mid])]5
                  low = mid + 1;]6
             else return mid;
       }
       return -1;
}
```

$$\begin{tikzcd}
1 \arrow[d] & & & \\
2 \arrow[dr] & & & \\
& 3 \arrow[dl] \arrow[dr] & & \\
4 \arrow[uu] & & 5 \arrow[dr] & \\
& & & 6 \arrow[uuulll, bend right]
\end{tikzcd}$$

Graphs – More Applications

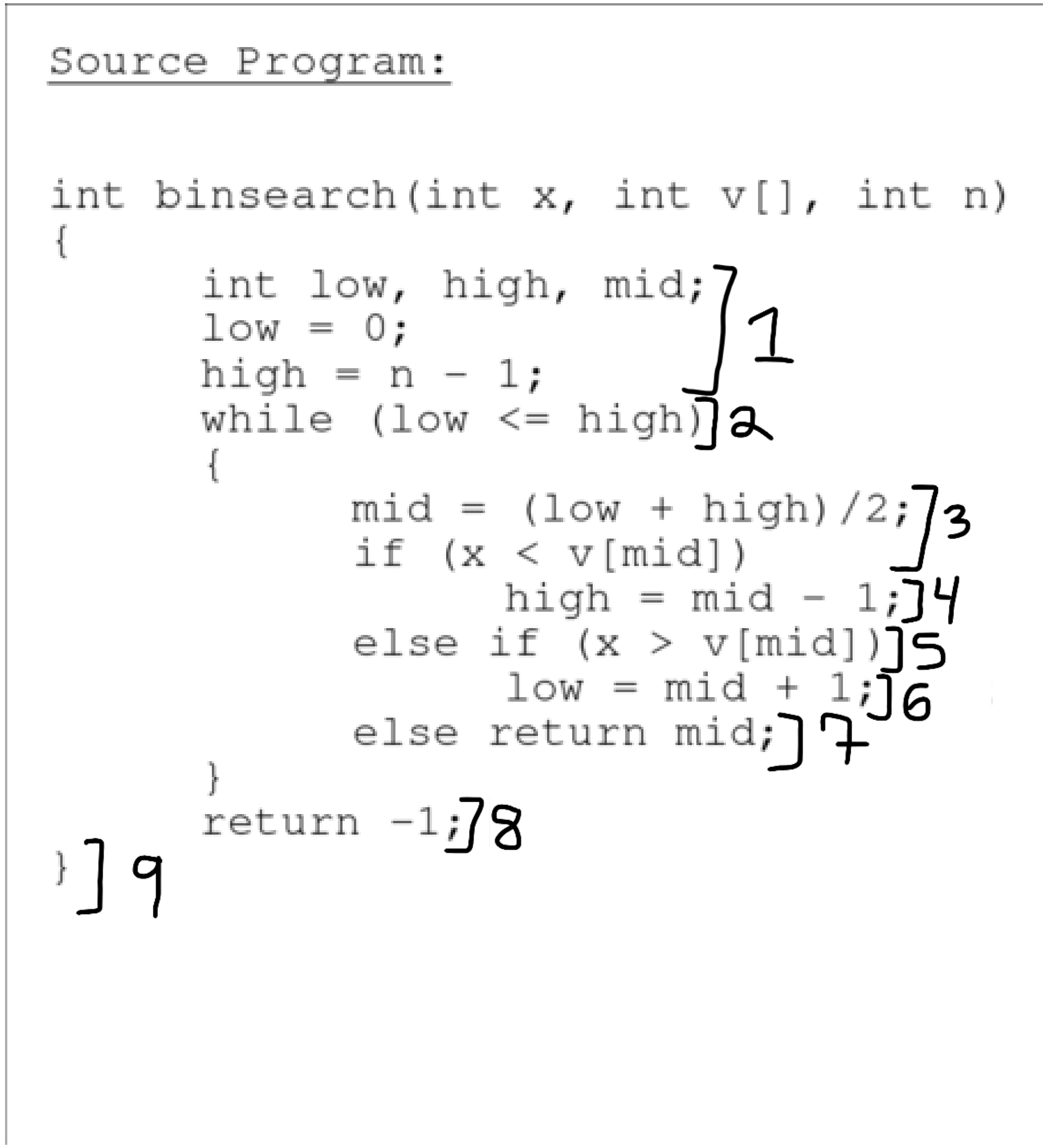


Source Program:

```
int binsearch(int x, int v[], int n)
{
        int low, high, mid;     ]
        low = 0;                ] 1
        high = n - 1;           ]
        while (low <= high)] 2
        {
                mid = (low + high)/2;] 3
                if (x < v[mid])      ]
                        high = mid - 1;] 4
                else if (x > v[mid])] 5
                        low = mid + 1;] 6
                else return mid;] 7
        }
        return -1;
}
```

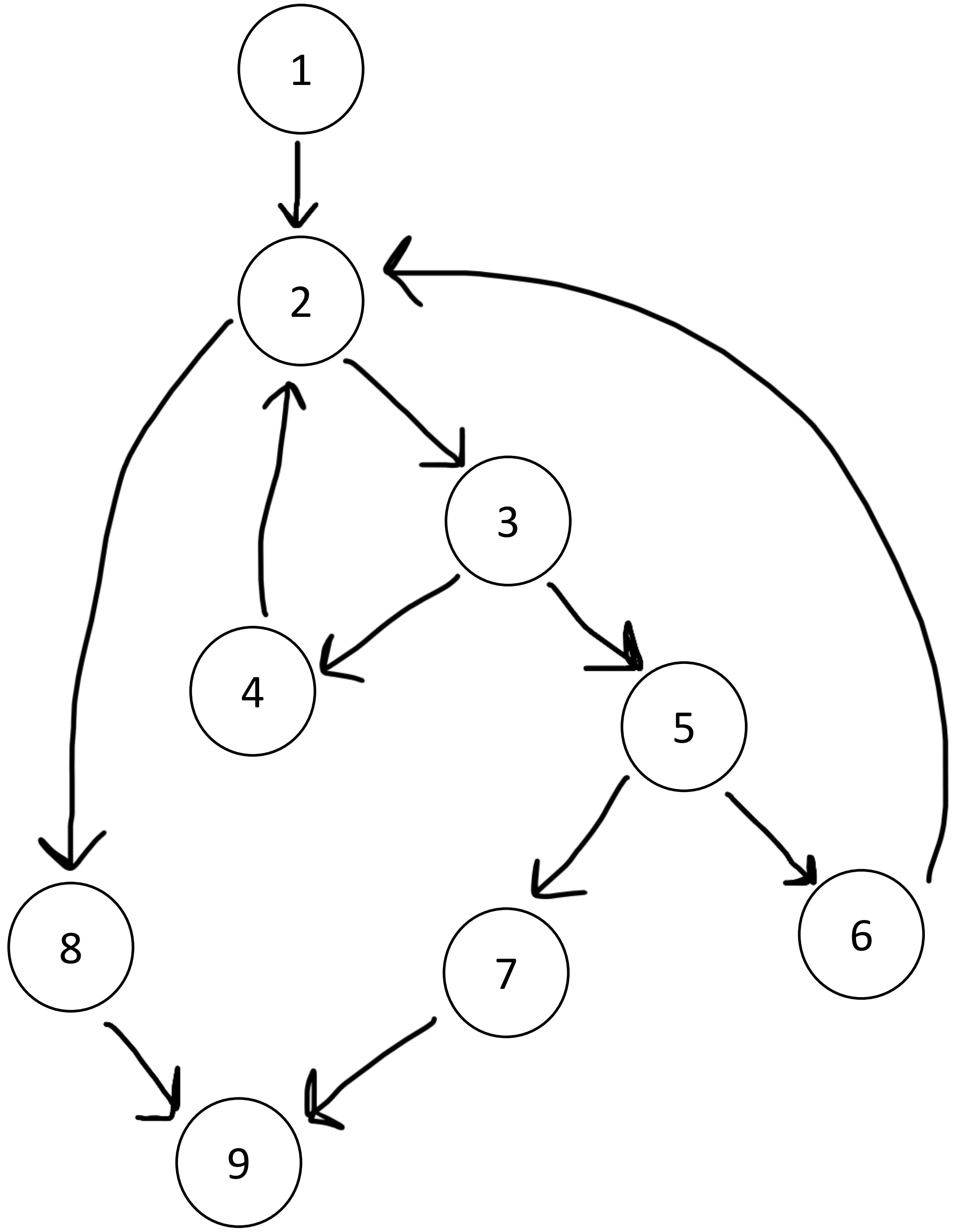
$$\begin{tikzcd}
& 1 \arrow[d] & & & \\
& 2 \arrow[dr] & & & \\
& & 3 \arrow[dll] \arrow[dr] & & \\
4 \arrow[uur] & & & 5 \arrow[dl] \arrow[dr] & \\
& & 7 & & 6 \arrow[uuulll, bend right]
\end{tikzcd}$$

Graphs – More Applications



Source Program:

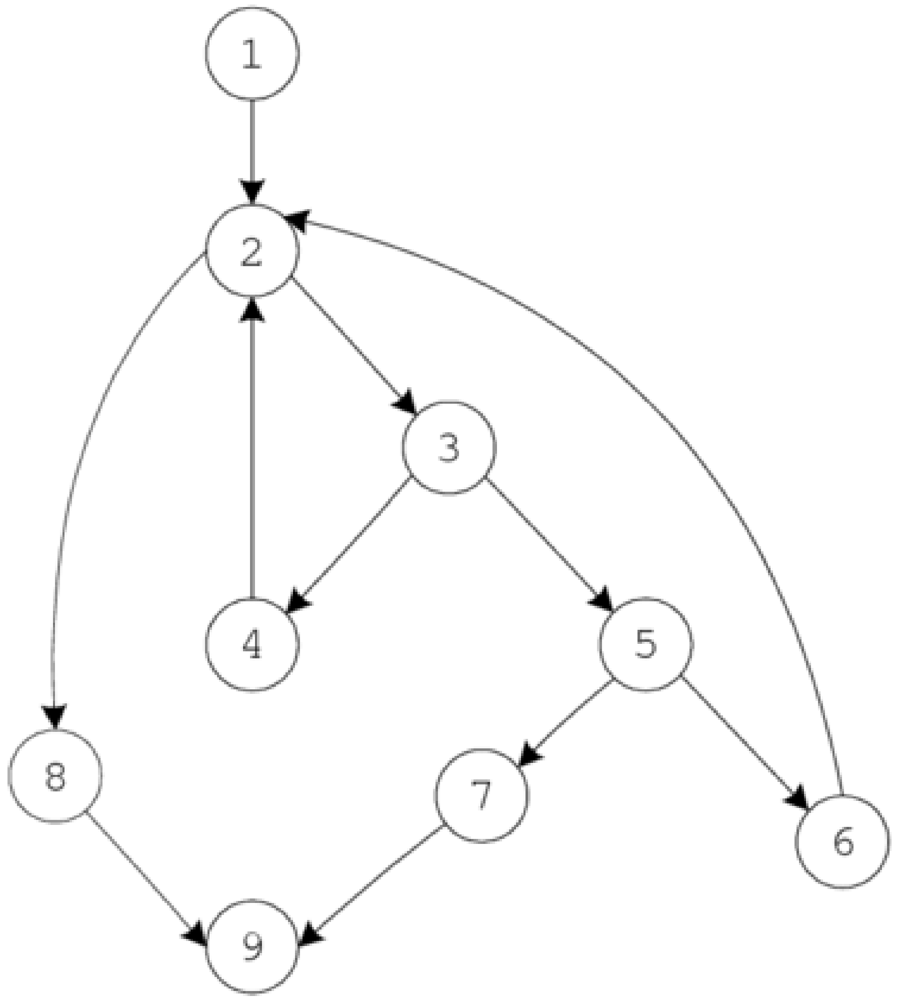
```
int binsearch(int x, int v[], int n)
{
        int low, high, mid;]
        low = 0;           ] 1
        high = n - 1;      ]
        while (low <= high)]2
        {
              mid = (low + high)/2;]3
              if (x < v[mid])      ]
                    high = mid - 1;]4
              else if (x > v[mid])]5
                    low = mid + 1;]6
              else return mid;]7
        }
        return -1;]8
}]9
```



Source Program:

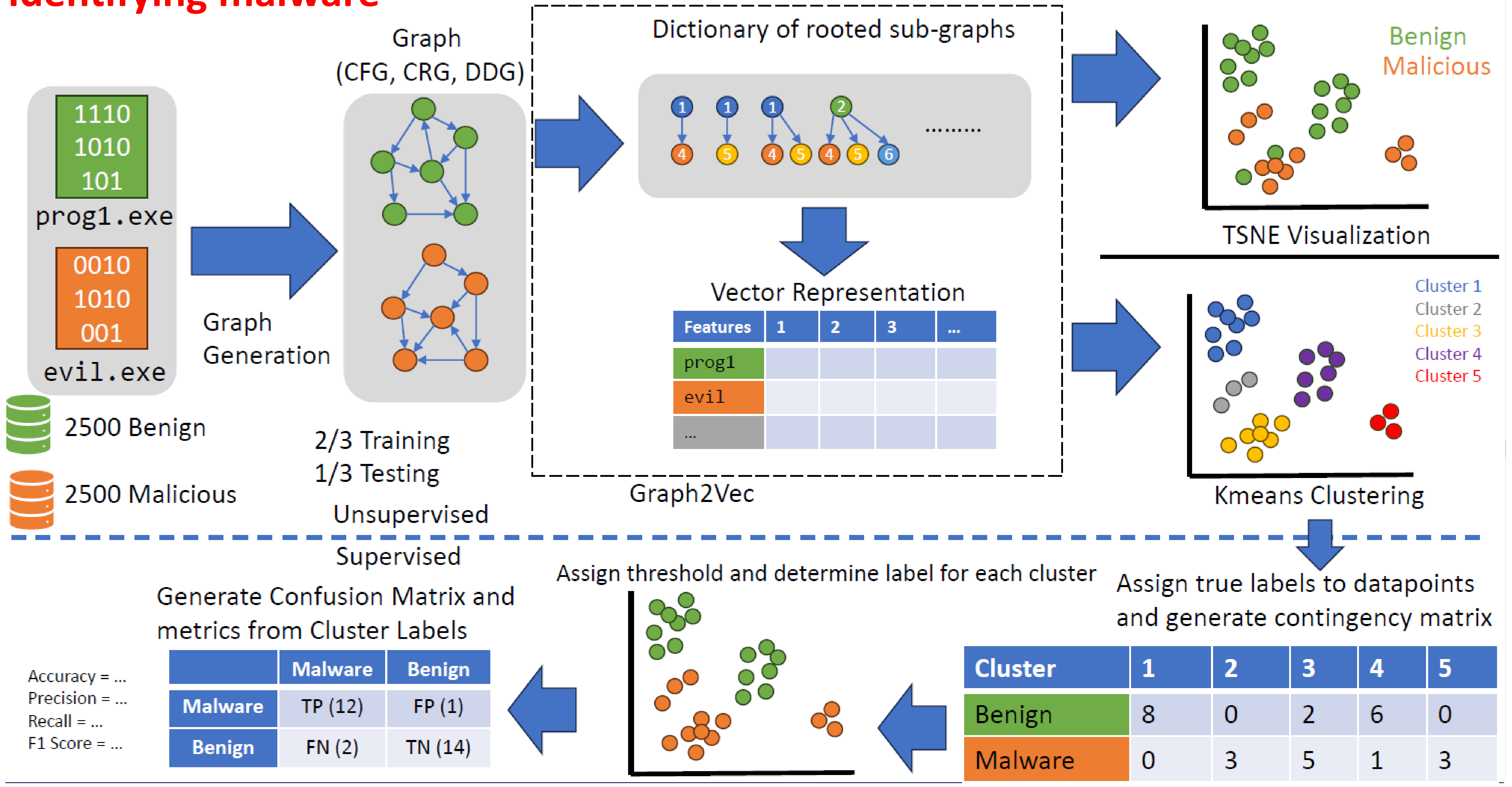
```
int binsearch(int x, int v[], int n)
{
        | int low, high, mid;
   1    | low = 0;
        | high = n - 1;
          while (low <= high) | 2
          {
               3 | mid = (low + high)/2;
                 | if (x < v[mid])
                          high = mid - 1;  | 4
               5 | else if (x > v[mid])
                          low = mid + 1;   | 6
               7 | else return mid;
          }
          return -1; | 8
} | 9
```

CFG:



Control Flow Graph (CFG)

Identifying malware



Features	1	2	3	...
prog1				
evil				
...				

Cluster	1	2	3	4	5
Benign	8	0	2	6	0
Malware	0	3	5	1	3

	Malware	Benign
Malware	TP (12)	FP (1)
Benign	FN (2)	TN (14)

Accuracy = ...
Precision = ...
Recall = ...
F1 Score = ...