Breadth-First Search

Breadth-First Search (BFS) is a graph traversal algorithm used in computer science to explore or search through the vertices of a graph in a breadthward motion. The algorithm starts at a designated source vertex and explores its neighbors before moving on to their neighbors, and so on. BFS is often used to find the shortest path between two vertices in an unweighted graph.

Key characteristics of Breadth-First Search:

Queue-based

BFS uses a **queue data structure to keep track of the vertices that need to be explored**. The algorithm starts by enqueueing the source vertex and then iteratively dequeues vertices, exploring their neighbors and enqueueing any unvisited neighbors.

Level-order traversal

BFS explores vertices level by level, visiting all neighbors at the current level before moving on to the next level. This ensures that the shortest path to each reachable vertex is discovered before longer paths.

Visited nodes

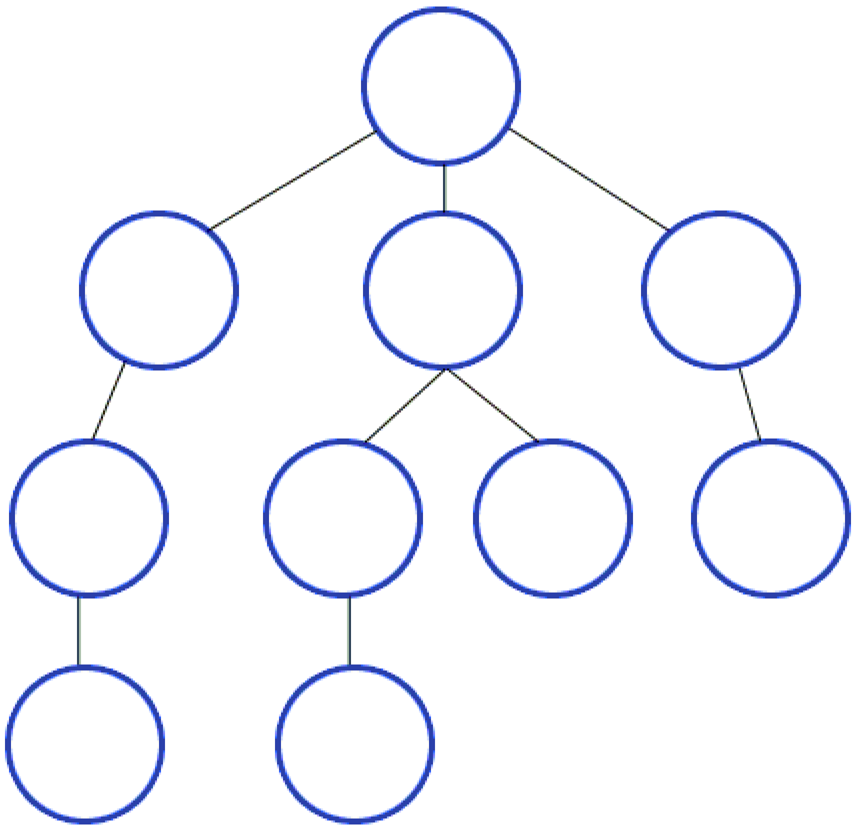
To avoid revisiting already-explored vertices and to prevent infinite loops in the case of cyclic graphs, BFS maintains a record of visited vertices.

Optimal for unweighted graphs

BFS is particularly suitable for unweighted graphs where all edges have the same weight. In such cases, the first time a vertex is reached in the search, it is guaranteed to be via the shortest path.

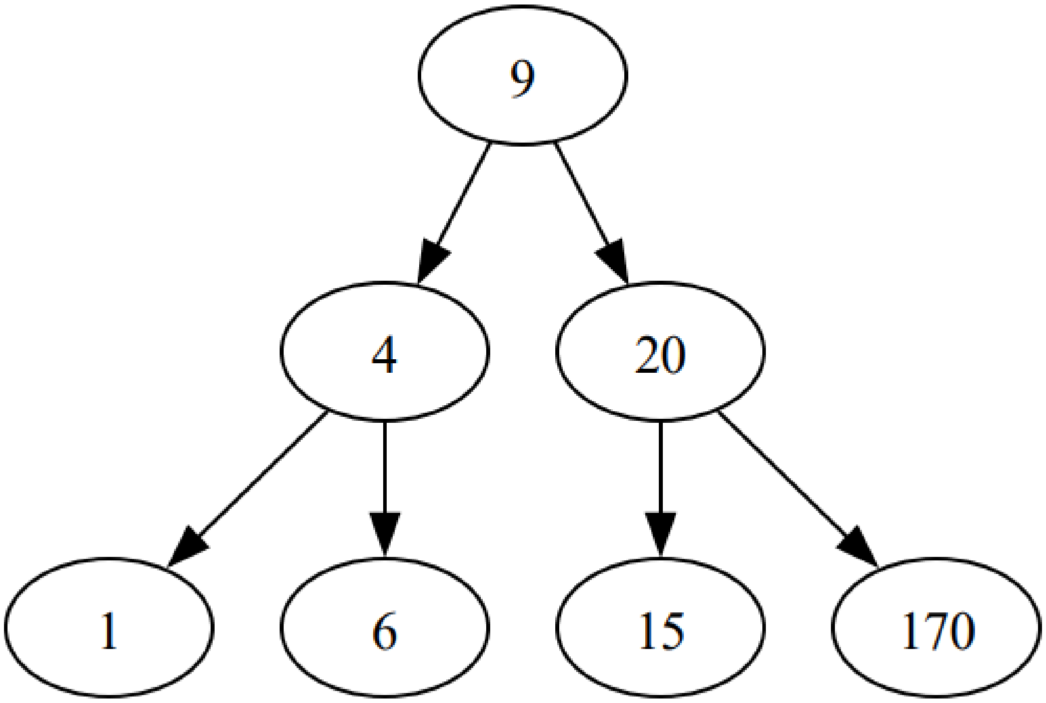
BFS is widely used in various applications, including network routing, social network analysis, and puzzle solving, among others. Its time complexity is O(V + E), where V is the number of vertices and E is the number of edges in the graph.

Trees



In [43]:
```python
from collections import deque # deque is a double-ended queue

class Node:
    def __init__(self, value):
        self.left = None
        self.right = None
        self.value = value

class BinarySearchTree:
    def __init__(self) -> None:
        self.root = None

    def insert(self, value):
        new_node = Node(value)

        if self.root is None:
            self.root = new_node
        else:
            current_node = self.root

            while True:
                if value < current_node.value:
                    if current_node.left is None:
                        current_node.left = new_node
                        return self
                    current_node = current_node.left
                elif value > current_node.value:
                    if current_node.right is None:
                        current_node.right = new_node
                        return self
                    current_node = current_node.right
    def lookup(self, value):
        if self.root is None:
            return False
        current_node = self.root
        while current_node:
            if value < current_node.value:
                current_node = current_node.left
            elif value > current_node.value:
                current_node = current_node.right
            elif current_node.value == value:
                return current_node
        return False

    def traverse(self):
        """Breadth First Search"""
        if not self.root:
            return []

        result = []
        queue = deque([self.root])

        while queue:
            current_node = queue.popleft()
            result.append(current_node.value)

            if current_node.left:
                queue.append(current_node.left)
            if current_node.right:
                queue.append(current_node.right)

        return result

    def traverse_R(self, queue, array):
        """Recursive Breadth First Search"""
        if not queue:
            return array

        current_node = queue.popleft()
        array.append(current_node.value)

        if current_node.left:
            queue.append(current_node.left)
        if current_node.right:
            queue.append(current_node.right)

        return self.traverse_R(queue, array)
```

In [44]:
```
bst = BinarySearchTree()
values = [9, 4, 6, 20, 170, 15, 1]

for value in values:
    bst.insert(value)
```



In [45]:
```
print(bst.traverse())
```

```
[9, 4, 20, 1, 6, 15, 170]
```

In [46]:
```
result = bst.traverse_R(deque([bst.root]), [])
print(result)
```

```
[9, 4, 20, 1, 6, 15, 170]
```

Depth First Search

Depth-First Search (DFS) is a graph traversal algorithm that explores as far as possible along each branch before backtracking. It's a fundamental algorithm used in graph theory and is employed to visit and process all the vertices and edges of a graph. The main idea behind DFS is to start from an initial vertex, visit one of its neighbors, then move to that neighbor's unvisited neighbor, and so on, until there are no more unvisited neighbors. When there are no more neighbors to visit at the current level, the algorithm backtracks to the previous level and explores any remaining unvisited neighbors from there.

Key points about Depth-First Search:

Stack or Recursion

DFS can be implemented using a stack data structure or through recursion. The stack keeps track of the vertices to be explored, and the recursion uses the call stack to manage the traversal.

Visited Marking

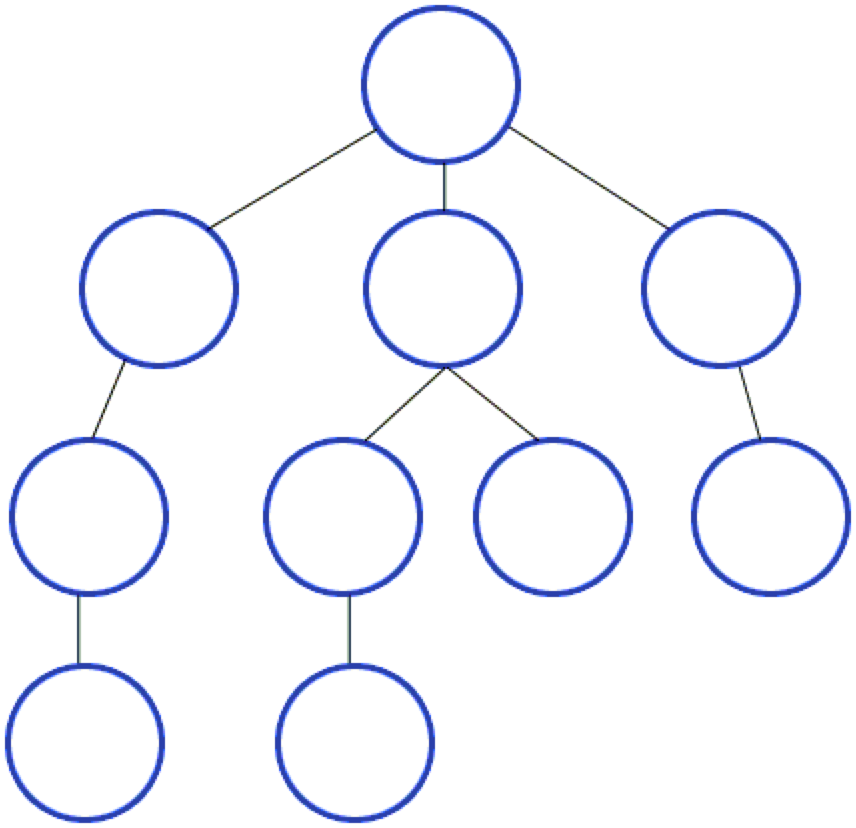
To avoid visiting the same vertex multiple times and to prevent infinite loops in cyclic graphs, a mechanism is used to mark vertices as "visited" once they have been explored.

Non-Optimal for Shortest Paths

While DFS is excellent for exploring all nodes in a graph, it may not be the best choice for finding the shortest path between two nodes in terms of the number of edges, as it does not necessarily prioritize the shortest paths.

Applications

DFS is widely used in various applications, such as topological sorting, detecting cycles in a graph, solving puzzles, and maze-solving algorithms.

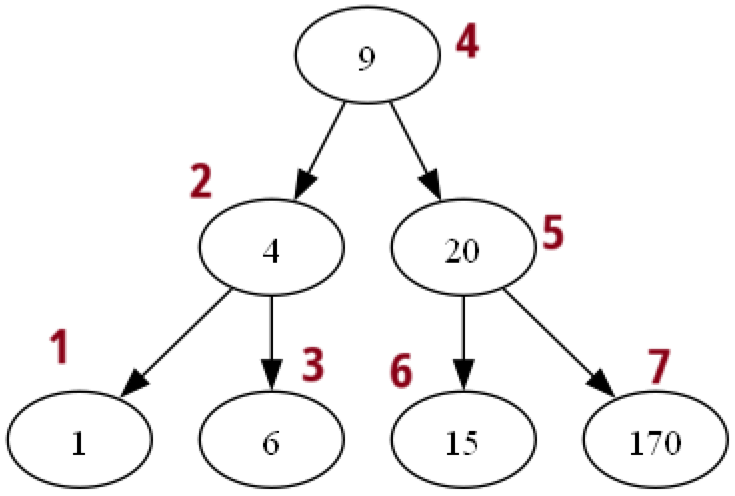


Methods:

In-Order Traversal:

1. Visit Left
2. Visit Current
3. Visit Right

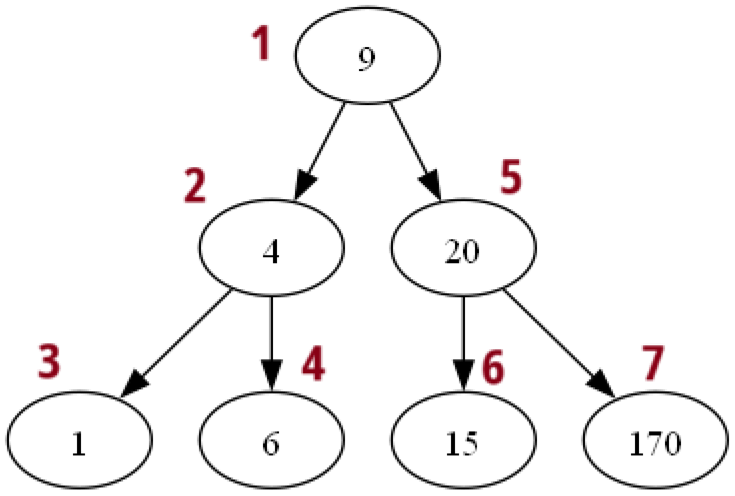
Order:



Pre-Order Traversal:

1. Visit Current
2. Visit Left
3. Visit Right

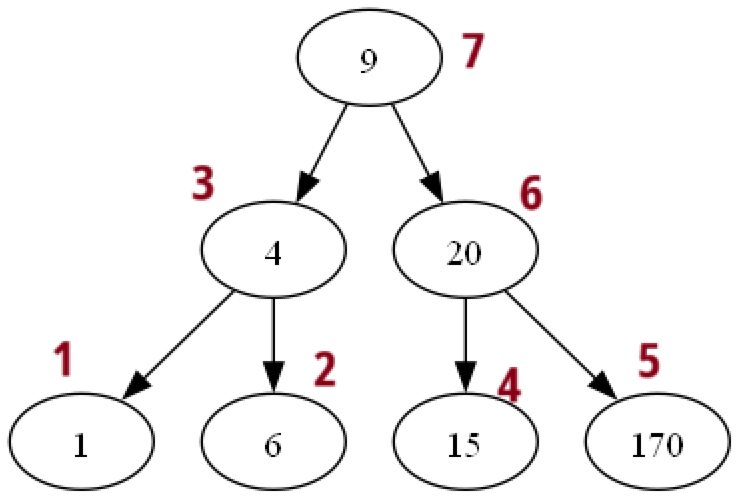
Order:



Post-Order Traversal:

1. Visit Left
2. Visit Right
3. Visit Current

Order:



These traversals are methods for systematically visiting and processing each node in a binary tree. The order in which you visit nodes determines the sequence in which you process or print them.

In [35]:
```
from collections import deque # deque is a double-ended queue

class Node:
    def __init__(self, value):
        self.left = None
        self.right = None
        self.value = value

class BinarySearchTree:
    def __init__(self) -> None:
        self.root = None

    def insert(self, value):
        new_node = Node(value)

        if self.root is None:
            self.root = new_node
        else:
            current_node = self.root

            while True:
                if value < current_node.value:
                    if current_node.left is None:
                        current_node.left = new_node
                        return self
                    current_node = current_node.left
                elif value > current_node.value:
                    if current_node.right is None:
                        current_node.right = new_node
                        return self
                    current_node = current_node.right

    def lookup(self, value):
        if self.root is None:
            return False
        current_node = self.root
        while current_node:
            if value < current_node.value:
                current_node = current_node.left
            elif value > current_node.value:
                current_node = current_node.right
            elif current_node.value == value:
                return current_node
        return False

    def traverse_in_order(self):
        """Depth First Search"""
        return _traverse_in_order(self.root, [])

    def traverse_post_order(self):
        """Depth First Search"""
        return _traverse_post_order(self.root, [])

    def traverse_pre_order(self):
        """Depth First Search"""
        return _traverse_pre_order(self.root, [])

"""Recursive Auxiliary Functions"""

def _traverse_in_order(node, array):
    if node.left:
        _traverse_in_order(node.left, array)
    array.append(node.value)
    if node.right:
        _traverse_in_order(node.right, array)
    return array

def _traverse_pre_order(node, array):
    array.append(node.value)
    if node.left:
        _traverse_pre_order(node.left, array)
    if node.right:
        _traverse_pre_order(node.right, array)
    return array

def _traverse_post_order(node, array):
    if node.left:
        _traverse_post_order(node.left, array)
    if node.right:
        _traverse_post_order(node.right, array)
    array.append(node.value)
    return array
```

In [36]:
```
bst2 = BinarySearchTree()
values = [9, 4, 6, 20, 170, 15, 1]

for value in values:
    bst2.insert(value)
```

In [37]:
```
in_order_result = bst2.traverse_in_order()
print(in_order_result)
```

```
[1, 4, 6, 9, 15, 20, 170]
```

In [38]:
```
pre_order_result = bst2.traverse_pre_order()
print(pre_order_result)
```

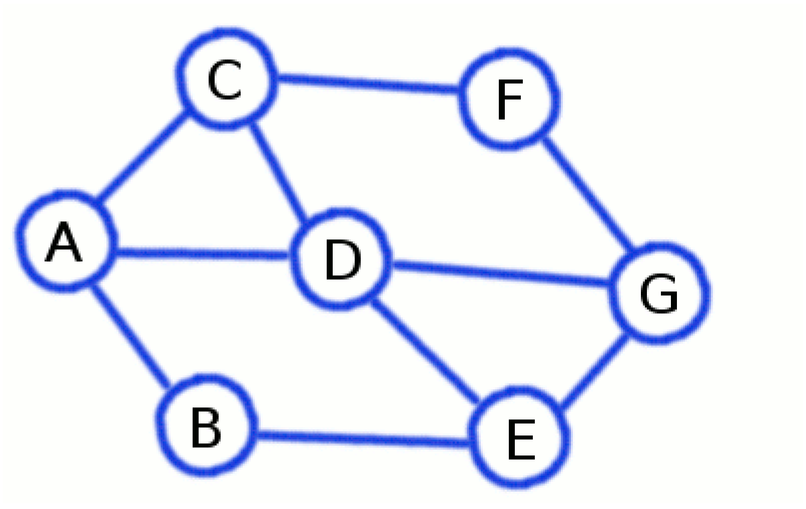
```
[9, 4, 1, 6, 20, 15, 170]
```

In [39]:
```
post_order_result = bst2.traverse_post_order()
print(post_order_result)
```

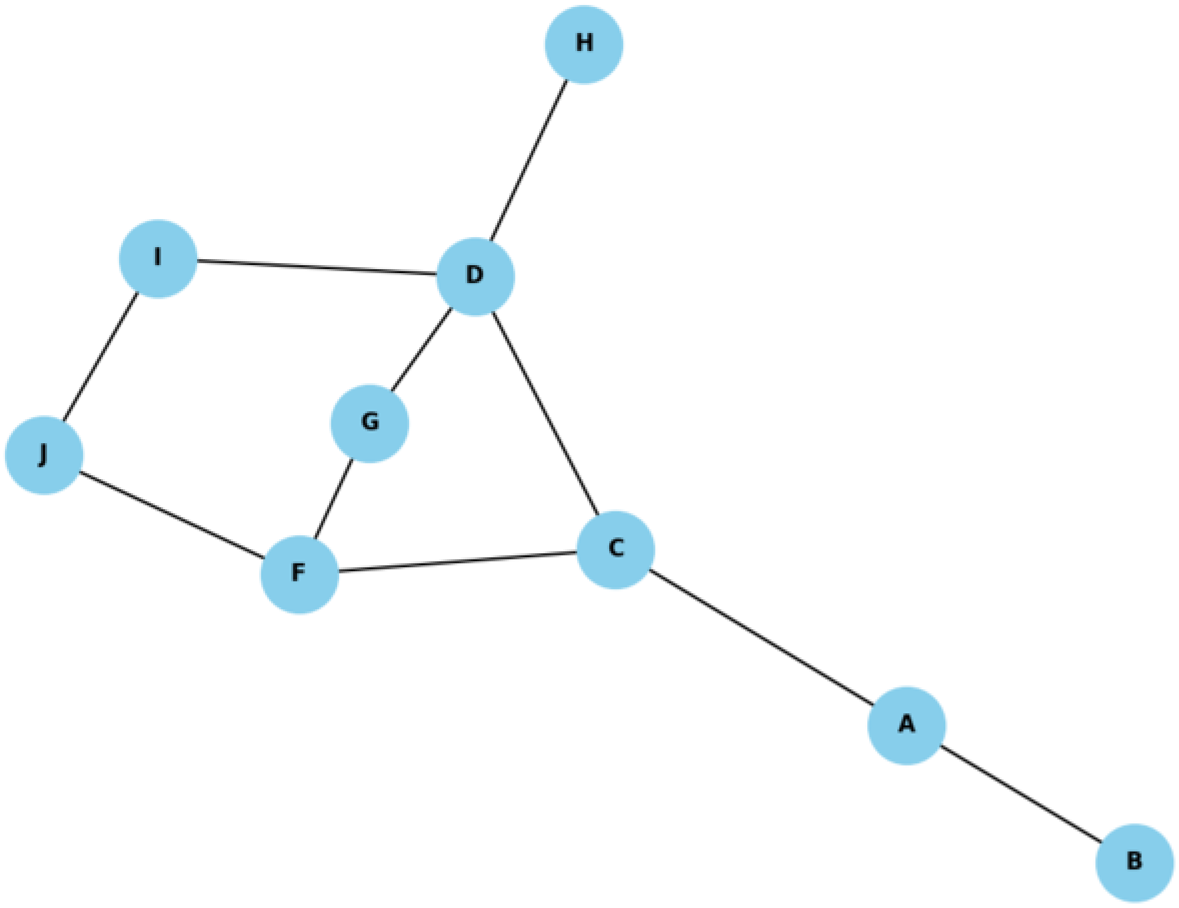
```
[1, 6, 4, 15, 170, 20, 9]
```

How it works visually with general graphs:

Breadth First Search



Given a undirected and unweighted graph A:



Traversing all the graph:

In [20]:
```
from collections import deque

def bfs(A, start):
    visited = set()
    queue = deque([start])
    while queue:
        node = queue.popleft()
        if node not in visited:
            print(node, end=' -> ')
            visited.add(node)

            for neighbor in A[node]:
                if neighbor not in visited:
                    queue.append(neighbor)

A = {
    'A': ['B', 'C'],
    'B': ['A'],
    'C': ['A', 'D', 'F'],
    'D': ['H', 'G', 'I', 'C'],
    'F': ['J', 'C', 'G'],
    'G': ['F', 'D'],
    'H': ['D'],
    'I': ['J', 'D'],
    'J': ['F', 'I'],
}

start_node = 'D'
print("BFS starting from node", start_node)
bfs(A, start_node)
```

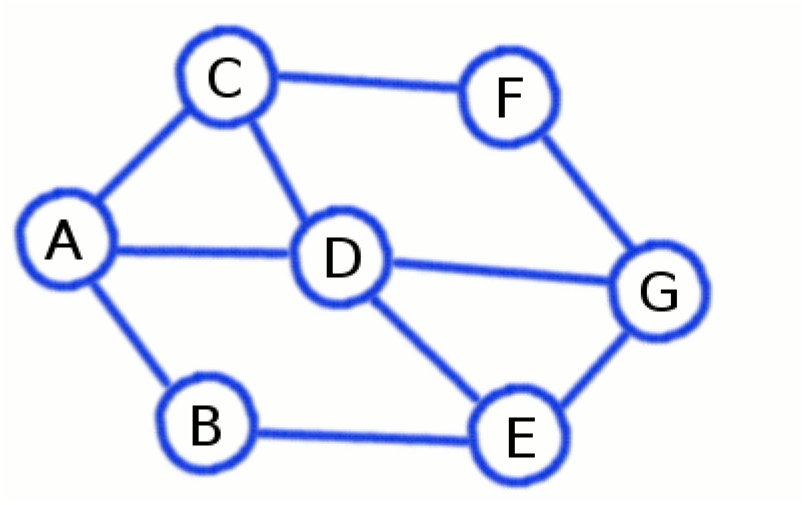
```
BFS starting from node D
D -> H -> G -> I -> C -> F -> J -> A -> B ->
```

Finding the shortest path between two nodes:

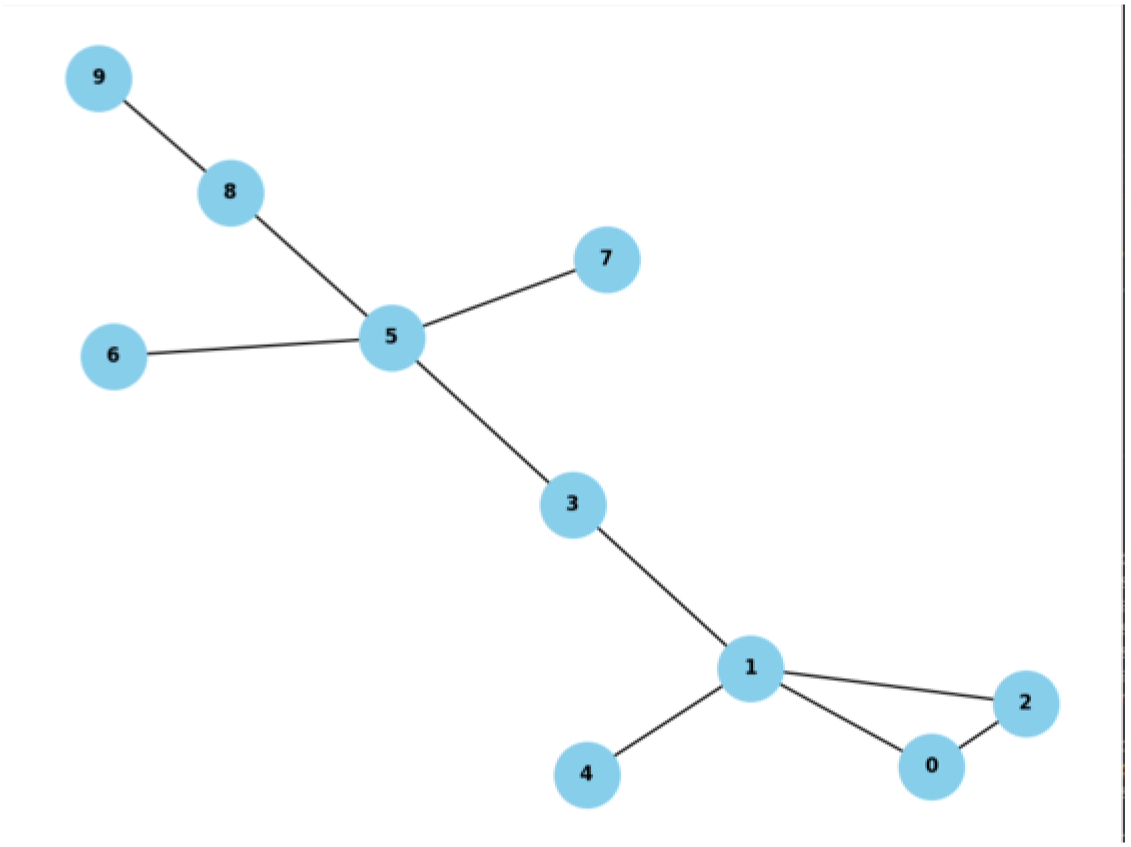
In [47]:
```
from collections import deque

def bfs_shortest_path(graph, start, end):
    visited = set()
    queue = deque([(start, 0)])

    while queue:
        node, distance = queue.popleft()

        if node == end:
            return distance

        if node not in visited:
            visited.add(node)

            for neighbor in graph[node]:
                if neighbor not in visited:
                    queue.append((neighbor, distance + 1))

    return -1

A = {
    'A': ['B', 'C'],
    'B': ['A'],
    'C': ['A', 'D', 'F'],
    'D': ['H', 'G', 'I', 'C'],
    'F': ['J', 'C', 'G'],
    'G': ['F', 'D'],
    'H': ['D'],
    'I': ['J', 'D'],
    'J': ['F', 'I'],
}

start_node = 'I'
end_node = 'B'

print(f"The shortest path between {start_node} and {end_node} is:", bfs_shortest_path(A, start_n
```

```
The shortest path between I and B is: 4
```

Depth First Search



Given a undirected and unweighted graph B:



In [51]:
```
def dfs(B, start, visited=None):
    """Graph with numeric values"""
    if visited is None:
        # OBS: In other languages, when set does not exist,
        # one alternative would be to use a boolean list:
```

```
        # marked = [False] * len(graph)
        visited = set()

    print(start, end=' ')
    visited.add(start)

    for neighbor in B[start]:
        if neighbor not in visited:
            dfs(B, neighbor, visited)

B = {
    '0': ['1', '2'],
    '1': ['0', '4', '3', '2'],
    '2': ['0', '1'],
    '3': ['5', '1'],
    '4': ['1'],
    '5': ['8', '6', '7', '3'],
    '6': ['5'],
    '7': ['5', '8'],
    '8': ['5', '9'],
    '9': ['8']
}

start_node = '3'
print("DFS starting from node", start_node)
dfs(B, start_node)
```

```
DFS starting from node 3
3 5 8 9 6 7 1 0 2 4
```

DFS Traversing based on minimum neighbor value:

In [50]:

```
def insertion_sort(arr):
    """If a node has a lot of neighbors, it is recommended to use a better sorting algorithm"""
    for i in range(1, len(arr)):
        for j in range(i - 1, -1, -1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
            else:
                break
    return arr

def dfs(graph, start, visited = None):
    if visited is None:
        visited = set()

    print(start, end=' ')
    visited.add(start)
    neighbors = insertion_sort([int(n) for n in graph[start]])

    for neighbor in neighbors:
        if str(neighbor) not in visited:
            dfs(graph, str(neighbor), visited)

"""
graph:

'node': [neighbors]
"""

graph = {
    '0': ['1', '2'],
    '1': ['0', '4', '3', '2'],
    '2': ['0', '1'],
    '3': ['5', '1'],
    '4': ['1'],
    '5': ['8', '6', '7', '3'],
    '6': ['5'],
    '7': ['5', '8'],
    '8': ['5', '9'],
    '9': ['8']
}

start_node = '3'
print("DFS starting from node", start_node)
dfs(graph, start_node)
```

```
DFS starting from node 3
3 1 0 2 4 5 6 7 8 9
```