S Abhishek AM.EN.U4CSE19147

# Data Structures

## Graph

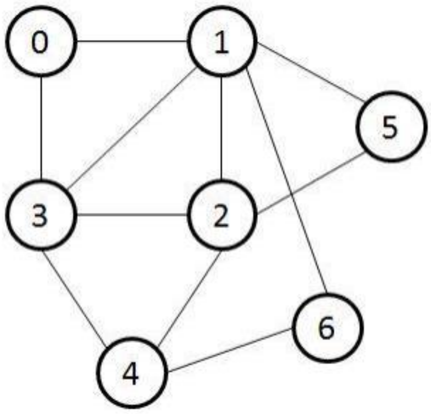


Figure 1

1. Implement the adjacency matrix representation of the graph shown in Figure 1.

```
class Graph(object):

    def __init__(self, row_col):
        self.arr = []
        for i in range(row_col):
            self.arr.append([0 for i in range(row_col)])
        self.row_col = row_col

    def add(self, v1, v2):
        if v1 == v2:
            print("Same vertex %d and %d" % (v1, v2))
        self.arr[v1][v2] = 1
        self.arr[v2][v1] = 1

    def display(self):
        for row in self.arr:
            for val in row:
                print('{}'.format(val),end="  ")
```

```python
            print()


if __name__ == '__main__':
    graph = Graph(7)
    graph.add(0, 1)
    graph.add(0, 3)
    graph.add(1, 2)
    graph.add(1, 3)
    graph.add(1, 5)
    graph.add(1, 6)
    graph.add(2, 3)
    graph.add(2, 4)
    graph.add(2, 5)
    graph.add(3, 4)
    graph.add(4, 6)

    graph.display()
```

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

2. Implement the adjacency list representation of the graph shown in Figure 1.

```
class Node:
    def __init__(self, value):
        self.vertex = value
        self.next = None


class Graph:
    def __init__(self, data):
        self.V = data
        self.graph = [None] * self.V

    def add(self, x, y):
        node = Node(y)
        node.next = self.graph[x]
        self.graph[x] = node

        node = Node(x)
        node.next = self.graph[y]
        self.graph[y] = node

    def display(self):
        for i in range(self.V):
            print("Vertex " + str(i) + " :", end="")
            ptr = self.graph[i]
            while ptr:
                print(" -> {}".format(ptr.vertex), end="")
                ptr = ptr.next
            print(" \n")


if __name__ == "__main__":
    graph = Graph(7)
    graph.add(0, 1)
```

```
graph.add(0, 3)
graph.add(1, 2)
graph.add(1, 3)
graph.add(1, 5)
graph.add(1, 6)
graph.add(2, 3)
graph.add(2, 4)
graph.add(2, 5)
graph.add(3, 4)
graph.add(4, 6)

graph.display()
```

```
Vertex 0 : -> 3 -> 1

Vertex 1 : -> 6 -> 5 -> 3 -> 2 -> 0

Vertex 2 : -> 5 -> 4 -> 3 -> 1

Vertex 3 : -> 4 -> 2 -> 1 -> 0

Vertex 4 : -> 6 -> 3 -> 2

Vertex 5 : -> 2 -> 1

Vertex 6 : -> 4 -> 1
```

3. Implement the functions for BFS traversal and DFS traversal on the graph shown in Figure 1.

```python
from collections import defaultdict


class Node:
    def __init__(self):
        self.graph = defaultdict(list)

    def add(self, u: int, v: int):
        self.graph[u].append(v)
        self.graph[v].append(u)

    def bfs(self, source: int):

        bfs_traverse = []

        is_visited = [False] * len(self.graph)
        queue = [source]

        is_visited[source] = True

        while len(queue) > 0:

            curr_node = queue.pop(0)
            bfs_traverse.append(curr_node)

            for neighbour_node in self.graph[curr_node]:
                if not is_visited[neighbour_node]:
                    queue.append(neighbour_node)
                    is_visited[neighbour_node] = True

        return bfs_traverse


def run_bfs(node: Node, source: int):
    return node.bfs(source)
```

```python
if __name__ == "__main__":
    graph = Node()
    graph.add(0, 1)
    graph.add(0, 3)
    graph.add(1, 2)
    graph.add(1, 3)
    graph.add(1, 5)
    graph.add(1, 6)
    graph.add(2, 3)
    graph.add(2, 4)
    graph.add(2, 5)
    graph.add(3, 4)
    graph.add(4, 6)

    bfs_traverse = run_bfs(graph, 0)

    print("\nBreadth First Search Traversal : ", end="")
    print(' '.join(str(ele) for ele in bfs_traverse))
```

```
Breadth First Search Traversal : 0 1 3 2 5 6 4
```

```python
from collections import defaultdict


class Node:
    def __init__(self):

        self.graph = defaultdict(list)

    def add(self, u: int, v: int):
        self.graph[u].append(v)
        self.graph[v].append(u)
```

```python
    def dfs(self, source: int):

        dfs_traverse = []

        is_visited = [False] * len(self.graph)
        stack = [source]

        is_visited[source] = True

        curr_node = source
        while len(stack) > 0:
            dfs_traverse.append(curr_node)

            flag_found_next = False

            while not flag_found_next and len(stack) > 0:

                for neighbour_node in self.graph[curr_node]:
                    if not is_visited[neighbour_node]:
                        # make visited True as they join queue
                        is_visited[neighbour_node] = True
                        stack.append(neighbour_node)
                        curr_node = neighbour_node
                        flag_found_next = True
                        break

                if not flag_found_next and len(stack):
                    curr_node = stack.pop()

        return dfs_traverse


def run_dfs(node: Node, source: int):
    return node.dfs(source)



if __name__ == "__main__":
```

```python
graph = Node()
graph.add(0, 1)
graph.add(0, 3)
graph.add(1, 2)
graph.add(1, 3)
graph.add(1, 5)
graph.add(1, 6)
graph.add(2, 3)
graph.add(2, 4)
graph.add(2, 5)
graph.add(3, 4)
graph.add(4, 6)

dfs_traverse = run_dfs(graph, 0)

print("Depth First Search Traversal : ", end="")
print(' '.join(str(ele) for ele in dfs_traverse))
```

```
Depth First Search Traversal : 0 1 2 3 4 6 5
```

**One Drive : [Click Me!!]**

**Thankyou!**