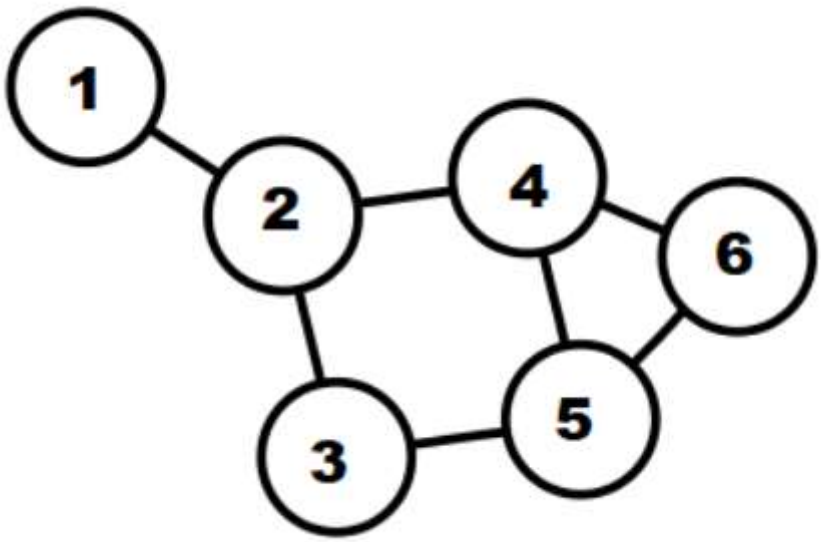



**Vertex**: Every individual data element is called a vertex or a node. In the above image 1,2,3,4,5 & 6 are the vertices.

**Edge**: It is a connecting link between two nodes or vertices. Each edge has two ends and is represented as (startingVertex,
endingVertex).

**Undirected Edge**: It is a bidirectional edge.

**Directed Edge**: It is a unidirectional edge.
m **Weighted Edge**: An edge with value (cost) on it.

**Degree**: The total number of edges connected to a vertex in a graph.

**Indegree**: The total number of incoming edges connected to a vertex.

**Outdegree**: The total number of outgoing edges connected to a vertex.

**Self-loop**: An edge is called a self-loop if its two endpoints coincide with each other.

**Adjacency**: Vertices are said to be adjacent to one another if there is an edge connecting them.

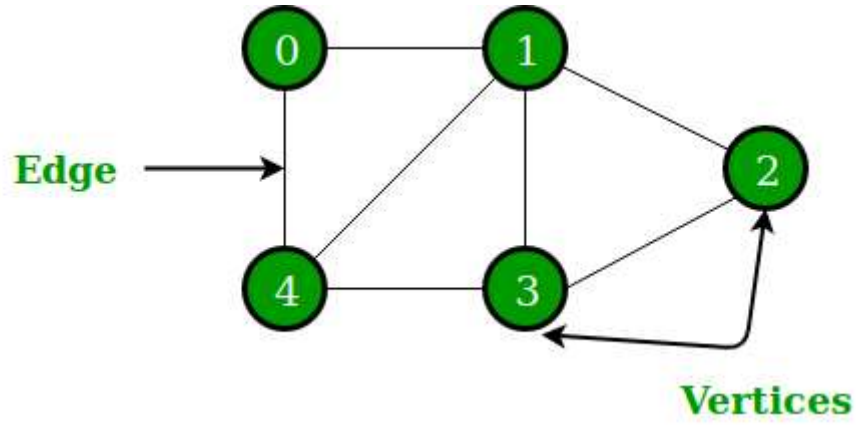
**

In [ ]:
```
struct TreeNode {
    int Data;
    TreeNode *left;
    TreeNode *right;
};
```

Graph Representation:

1. Adjacency list
2. Matrix representationm

## Represensation of Unweighted Undirected Graph



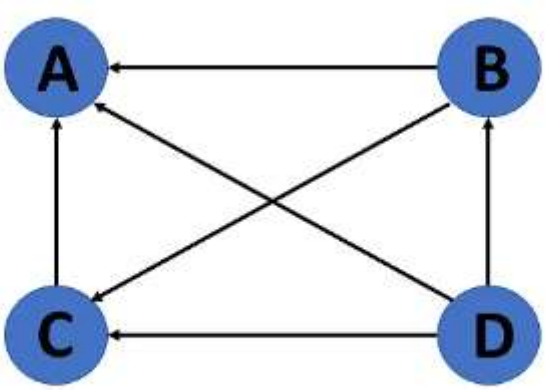

```
v: vertices
e: edges

0: [1,4]
1: [0,4,3,2]
4: [0,1,3]
3: [4,1,2]
2: [1,3]

SC: (v+2e)
```

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 |
| 2 | 0 | 1 | 0 | 1 | 0 |
| 3 | 0 | 1 | 1 | 0 | 1 |
| 4 | 1 | 1 | 0 | 1 | 0 |

```
SC: (v^2)
```

In [ ]:

## Represensation of Unweighted Directed Graph




```
A: []
B: [A,C]
C: [A]
D: [A,B,C]

SC: (v+2e)

  A  B  C  D
A 0  0  0  0
B 1  0  1  0
C 1  0  0  0
D 1  1  1  0

SC: (v^2)
```

In [ ]:

In [4]:
```python
# Adjancey List Repr
# dict<int, List>
# map<int, vector<int>>
class UndirectedGraph:

    def __init__(self):
        self.__data = {} # empty dict/map

    def add(self, v1, v2):

        if v1 not in self.__data:
            self.__data[v1] = []
        self.__data[v1].append(v2)

        if v2 not in self.__data:
            self.__data[v2] = []
        self.__data[v2].append(v1)

    def print(self):
        print(self.__data)


g = UndirectedGraph()
g.add(0,1)
g.add(0,4)
g.add(1,4)
g.add(1,3)
g.add(1,2)
g.add(4,3)
g.add(3,2)
g.print()

# 0: [1,4]
# 1: [0,4,3,2]
# 4: [0,1,3]
# 3: [4,1,2]
# 2: [1,3]
```

```
{0: [1, 4], 1: [0, 4, 3, 2], 4: [0, 1, 3], 3: [1, 4, 2], 2: [1, 3]}
```

In [8]:
```
# Adjancey List Repr
# dict<int, List>
# map<int, vector<int>>
class DirectedGraph:

    def __init__(self):
        self.__data = {} # empty dict/map

    def add(self, v1, v2):
        """
        Create an edge from v1->v2
        """

        if v1 not in self.__data:
            self.__data[v1] = []
        self.__data[v1].append(v2)

        if v2 not in self.__data:
            self.__data[v2] = []

    def print(self):
        print(self.__data)


g = DirectedGraph()
g.add('B','A')
g.add('B','C')
g.add('C','A')
g.add('D','A')
g.add('D','B')
g.add('D','C')
# A: []
# B: [A,C]
# C: [A]
# D: [A,B,C]
g.print()
```

```
{'B': ['A', 'C'], 'A': [], 'C': ['A'], 'D': ['A', 'B', 'C']}
```

In [ ]:
```

```

## Matrix Repr

In [27]:
```
# dict<int, List>
# map<int, vector<int>>
class UndirectedGraph:

    def __init__(self, num_vertices):
        self.__data = []
        for _ in range(num_vertices):
            self.__data.append( [0 for _ in range(num_vertices) ] )

    def add(self, v1, v2):
        self.__data[v1][v2] = 1
        self.__data[v2][v1] = 1

    def print(self):
        for row in self.__data:
            print(row)


g = UndirectedGraph(5)
g.add(0,1)
g.add(0,4)
g.add(1,4)
g.add(1,3)
g.add(1,2)
g.add(4,3)
g.add(3,2)
g.print()

# 0: [1,4]
# 1: [0,4,3,2]
# 4: [0,1,3]
# 3: [4,1,2]
# 2: [1,3]
```

```
[0, 1, 0, 0, 1]
[1, 0, 1, 1, 1]
[0, 1, 0, 1, 0]
[0, 1, 1, 0, 1]
[1, 1, 0, 1, 0]
```

In [ ]:

In [ ]:

## Traversal

- DFS: stack, recursion
- BFS: queue, iteration

In [ ]:

## BFS: Using queue

image.png

In [8]:
```
import queue
class UndirectedGraph:

    def __init__(self):
        self.__data = {} # empty dict/map

    def add(self, v1, v2):

        if v1 not in self.__data:
            self.__data[v1] = []
        self.__data[v1].append(v2)

        if v2 not in self.__data:
            self.__data[v2] = []
        self.__data[v2].append(v1)

    def print(self):
        print(self.__data)

    def level_order_traversal(self):
        start = list(self.__data.keys())[0] # get first key(node) from the has

        q = queue.Queue()
        visited = set()

        q.put(start)
        visited.add(start)

        while not q.empty():
            curr = q.get()

            adj = self.__data[curr]
            for v in adj:
                if v not in visited:
                    q.put(v)
                    visited.add(v)

            print(curr)

g = UndirectedGraph()
g.add(0,1)
g.add(0,4)
g.add(1,4)
g.add(1,3)
g.add(1,2)
g.add(4,3)
g.add(3,2)
g.print()

g.level_order_traversal()
```

```
{0: [1, 4], 1: [0, 4, 3, 2], 4: [0, 1, 3], 3: [1, 4, 2], 2: [1, 3]}
0
1
4
3
2
```

In [7]:
```
dir(dict)
```

Out[7]:
```
['__class__',
 '__class_getitem__',
 '__contains__',
 '__delattr__',
 '__delitem__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__ior__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__ne__',
 '__new__',
 '__or__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__reversed__',
 '__ror__',
 '__setattr__',
 '__setitem__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'clear',
 'copy',
 'fromkeys',
 'get',
 'items',
 'keys',
 'pop',
 'popitem',
 'setdefault',
 'update',
 'values']
```

# DFS

In [13]:
```
import queue
class UndirectedGraph:

    def __init__(self):
        self.__data = {} # empty dict/map

    def add(self, v1, v2):

        if v1 not in self.__data:
            self.__data[v1] = []
        self.__data[v1].append(v2)

        if v2 not in self.__data:
            self.__data[v2] = []
        self.__data[v2].append(v1)

    def print(self):
        print(self.__data)

    def __traverse(self, v, visited):
        if v in visited:
            return

        visited.add(v)
        print(v)

        adj = self.__data[v]
        for a in adj:
            if a not in visited:
                self.__traverse(a, visited)

    def dfs(self):

        visited =  set()

        for v in self.__data.keys():
            self.__traverse(v, visited)
# {0: [1, 4], 1: [0, 4, 3, 2], 4: [0, 1, 3], 3: [1, 4, 2], 2: [1, 3]}
# curr = 0
# visited     (0, 1, 4, 3, 2)
#             t(0)
#                 t(1)
#                     t(4)
#                         t(3)
#                             t(2)

g = UndirectedGraph()
g.add(0,1)
g.add(0,4)
g.add(1,4)
g.add(1,3)
g.add(1,2)
g.add(4,3)
g.add(3,2)
g.print()
```

```
g.dfs()
```

```
{0: [1, 4], 1: [0, 4, 3, 2], 4: [0, 1, 3], 3: [1, 4, 2], 2: [1, 3]}
0
1
4
3
2
```

In [ ]:

## Disconnected Graph

In [ ]:

In [ ]:

https://leetcode.com/problems/clone-graph/description/ (https://leetcode.com/problems/clone-graph/description/)

In [ ]:

In [ ]:

### DIY

DFS of a graph using Stack

https://leetcode.com/problems/keys-and-rooms/description/ (https://leetcode.com/problems/keys-and-rooms/description/)

How a hash map works internally. Try to implement a hash map on your own.