Activity 1:

Consider a toy problem that can be represented as a following graph. How would you represent this graph in python?

In [66]:
```
class node:
  def __init__(self,state,parent,actions,totalcost):
    self.state = state
    self.parent = parent
    self.actions = actions
    self.totalcost = totalcost

graph = {'A': node('A',None,['B','C','E'],None),
           'B': node('B',None,['A','D','E'],None),
           'C': node('C',None,['A','F','G'],None),
           'D': node('D',None,['B','E'],None),
           'E': node('E',None,['A','B','D'],None),
           'F': node('F',None,['C'],None),
           'G': node('G',None,['C'],None)
         }
```

Activity 2:

For the graph in previous activity, imagine node A as starting node and your goal is to reach F. Keeping depth first search in mind, describe a sequence of actions that you must take to reach that goal state.

In [61]:
```
class Node:
    def __init__(self, state, parent, adjacentNodes, cost):
        self.state = state
        self.parent = parent
        self.adjacentNodes = adjacentNodes
        self.totalCost = cost

def actionSequence(graph, initialNode, goalNode):
    solution = [goalNode]
    currentParent = graph[goalNode].parent
    while currentParent is not None:
        solution.append(currentParent)
        currentParent = graph[currentParent].parent
    solution.reverse()
    return solution
```

```
def DFS():
    initialState = 'A'
    goalState = 'D'
    graph = {
        'A': Node('A', None, ['B', 'C', 'E'], None),
        'B': Node('B', None, ['A', 'D', 'E'], None),
        'C': Node('C', None, ['A', 'F', 'G'], None),
        'D': Node('D', None, ['B', 'E'], None),
        'E': Node('E', None, ['A', 'B', 'D'], None),
        'F': Node('F', None, ['C'], None),
        'G': Node('G', None, ['C'], None)
    }
    stack = [initialState]
    visited = []
    while len(stack) != 0:
        currentNode = stack.pop(len(stack)-1)
        visited.append(currentNode)
        for child in graph[currentNode].adjacentNodes:
            if child not in stack and child not in visited:
                graph[child].parent = currentNode
                if graph[child].state == goalState:
                    return actionSequence(graph, initialState, goalState)
                stack.append(child)

print(DFS())
```

```
['A', 'E', 'D']
```

Activity 3:

Change initial state to D and set goal state as C. What will be resulting path of BFS search? What will be the sequence of nodes explored?

In [62]:

```
class Node:
    def __init__(self, state, parent, adjacentNodes, cost):
        self.state = state
        self.parent = parent
        self.adjacentNodes = adjacentNodes
        self.totalCost = cost

def actionSequence(graph, initialNode, goalNode):
    solution = [goalNode]
    currentParent = graph[goalNode].parent
    while currentParent is not None:
        solution.append(currentParent)
        currentParent = graph[currentParent].parent
```

```
    solution.reverse()
    return solution

def DFS():
    initialState = 'D'
    goalState = 'C'
    graph = {
        'A': Node('A', None, ['B', 'C', 'E'], None),
        'B': Node('B', None, ['A', 'D', 'E'], None),
        'C': Node('C', None, ['A', 'F', 'G'], None),
        'D': Node('D', None, ['B', 'E'], None),
        'E': Node('E', None, ['A', 'B', 'D'], None),
        'F': Node('F', None, ['C'], None),
        'G': Node('G', None, ['C'], None)
    }
    stack = [initialState]
    visited = []
    while len(stack) != 0:
        currentNode = stack.pop(len(stack)-1)
        visited.append(currentNode)
        for child in graph[currentNode].adjacentNodes:
            if child not in stack and child not in visited:
                graph[child].parent = currentNode
                if graph[child].state == goalState:
                    return actionSequence(graph, initialState, goalState)
                stack.append(child)

print(DFS())
```

```
['D', 'E', 'A', 'C']
```

Activity 4:

Change initial state to D and set goal state as C. What will be resulting path of BFS search? What will be the sequence of nodes explored?

In [63]:

```
import math

class Node:
    def __init__(self, state, parent, adjacentNodes, cost):
        self.state = state
        self.parent = parent
        self.adjacentNodes = adjacentNodes
        self.totalCost = cost
```

```
def findMin(frontier):
    minV = math.inf
    node = ''
    for i in frontier:
        if minV>frontier[i][1]:
            minV = frontier[i][1]
            node = i
    return node

def actionSequence(graph, initialNode, goalNode):
    solution = [goalNode]
    currentParent = graph[goalNode].parent
    while currentParent is not None:
        solution.append(currentParent)
        currentParent = graph[currentParent].parent
    solution.reverse()
    return solution




def UCS():
    initialNode = 'C'
    goalNode = 'B'

    graph = {
        'A': Node('A', None, [('B', 6), ('C', 9), ('E', 1)], 0),
        'B': Node('B', None, [('A', 6), ('D', 3), ('E', 4)], 0),
        'C': Node('C', None, [('A', 9), ('F', 2), ('G', 3)], 0),
        'D': Node('D', None, [('B', 3), ('E', 5), ('F', 7)], 0),
        'E': Node('E', None, [('A', 1), ('B', 4), ('D', 5), ('F', 6)], 0),
        'F': Node('F', None, [('C', 2), ('E', 6), ('D', 7)], 0),
        'G': Node('G', None, [('C', 3)], 0)
    }

    frontier = dict()
    frontier[initialNode] = (None, 0)
    explored = []

    while len(frontier)!=0:
        currentNode = findMin(frontier)
        del frontier[currentNode]
        if graph[currentNode].state == goalNode:
            return actionSequence(graph, initialNode, goalNode)
        explored.append(currentNode)
```

```
        for child in graph[currentNode].adjacentNodes:
            currentCost = child[1] + graph[currentNode].totalCost
            if child[0] not in frontier and child[0] not in explored:
                graph[child[0]].parent = currentNode
                graph[child[0]].totalCost = currentCost
                frontier[child[0]] = (graph[child[0]].parent, graph[child[0]].totalCost)
            elif child[0] in frontier:
                if frontier[child[0]][1] < currentCost:
                    graph[child[0]].parent=frontier[child[0]][0]
                    graph[child[0]].totalCost = frontier[child[0]][1]
                else:
                    frontier[child[0]] = (currentNode, currentCost)
                    graph[child[0]].parent = frontier[child[0]][0]
                    graph[child[0]].totalCost = frontier[child[0]][1]
print(UCS())
```

```
['C', 'F', 'D', 'B']
```

Activity 5:

Imagine going from Arad to Bucharest in the following map. Your goal is to minimize the distance mentioned in the map during your travel. Implement a uniform cost search to find the corresponding path.

In [74]:

```
class Node:
  def __init__(self,state,parent,actions,totalcost):
    self.state=state
    self.parent=parent
    self.actions=actions
    self.totalcost=totalcost

graph={'arad':Node('arad',None,['zernid','timisoara','sibiu'],None),
       'timisoara':Node('timisoara',None,['lugoj','arad'],None),
       'zernid':Node('zernid',None,['arad','oradea'],None),
       'sibiu':Node('sibiu',None,['arad','oradea','fagaras','rimnicu vilcea'],None),
       'lugoj':Node('lugoj',None,['mehadia','timisoara'],None),
       'oradea':Node('oradea',None,['zernid','sibiu'],None),
       'mehadia':Node('mehadia',None,['lugoj','drobeta'],None),
       'drobeta':Node('drobeta',None,['mehadia','craiova'],None),
       'craiova':Node('craiova',None,['drobeta','pitesti','rimnicu vilcea'],None),
       'rimnicu vilcea':Node('rimnicu vilcea',None,['craiova','pitesti','sibiu'],None),
       'pitesti':Node('pitesti',None,['craiova','rimnicu vilcea','bucharest'],None),
       'fagaras':Node('fagaras',None,['sibiu','bucharest'],None),
       'bucharest':Node('bucharest',None,['fagaras','pitesti','giurgiu','urziceni'],None),
       'giurgiu':Node('giurgiu',None,['bucharest'],None),
```

```
        'urziceni':Node('urziceni',None,['bucharest','hirsova','vaslui'],None),
        'hirsova':Node('hirsova',None,['urziceni','eforie'],None),
        'eforie':Node('eforie',None,['hirsova'],None),
        'vaslui':Node('vaslui',None,['urziceni','lasi'],None),
        'lasi':Node('lasi',None,['vaslui','neamt'],None),
        'neamt':Node('neamt',None,['lasi'],None),
}

def actionsequence(graph, initialstate,goalstate):
  solution=[goalstate]
  currentparent = graph[goalstate].parent
  while currentparent != None:
    solution.append(currentparent)
    currentparent = graph[currentparent].parent
  solution.reverse()
  print(solution)
  return solution

def BFS():
  initialstate = 'arad'
  goalstate = 'bucharest'
  frontier = [initialstate]
  explored = []

  while len(frontier)!=0:
    currentNode = frontier.pop(len(frontier)-1)
    explored.append(currentNode)
    for child in graph[currentNode].actions:
        if child not in frontier and child not in explored:
            graph[child].parent = currentNode
            if graph[child].state == goalstate:
                  return actionsequence(graph,initialstate,goalstate)
            frontier.append(child)
solution = BFS()
```

```
['arad', 'sibiu', 'rimnicu vilcea', 'pitesti', 'bucharest']
```

In [ ]:

In [ ]: