In [14]:
```
# sample graph implemented as a dictionary
graph = {'A': ['B', 'C', 'E'],
         'B': ['A','D', 'E'],
         'C': ['A', 'F', 'G'],
         'D': ['B'],
         'E': ['A', 'B','D'],
         'F': ['C'],
         'G': ['C']}
```

In [17]:
```
# finds shortest path between 2 nodes of a graph using BFS
def bfs_shortest_path(graph, start, goal):
    # keep track of explored nodes
    explored = []
    # keep track of all the paths to be checked
    queue = [[start]]

    # return path if start is goal
    if start == goal:
        return "That was easy! Start = goal"

    # keeps looping until all possible paths have been checked
    while queue:
        # pop the first path from the queue
        path = queue.pop(0)
        # get the last node from the path
        node = path[-1]
        if node not in explored:
            neighbours = graph[node]
            # go through all neighbour nodes, construct a new path and
            # push it into the queue
            for neighbour in neighbours:
                new_path = list(path)
                new_path.append(neighbour)
                queue.append(new_path)
                # return path if neighbour is goal
                if neighbour == goal:
                    return new_path

            # mark node as explored
            explored.append(node)

    # in case there's no path between the 2 nodes
    return "So sorry, but a connecting path doesn't exist :("
```

In [18]:
```
bfs_shortest_path(graph, 'G', 'D')  # returns ['G', 'C', 'A', 'B', 'D']
```

```
<class 'list'>
```

Out[18]:
```
['G', 'C', 'A', 'B', 'D']
```

DFS

In [34]:
```
# Using a Python dictionary to act as an adjacency list
graphs = {
    'A' : ['B','C'],
    'B' : ['D', 'E'],
    'C' : ['F'],
    'D' : [],
```

```
    'E' : ['F'],
    'F' : []
}

#visited = [] # Set to keep track of visited nodes.
#visited = set() # Lab work
def dfs(visited, graphs, node):
    if node not in visited:
        # print(node)
        #visited.add(node)
        visited.append(node)
        for neighbour in graphs[node]:
            dfs(visited, graphs, neighbour)
    return visited

# Driver Code
visit = dfs([], graphs, 'A')
print(visit)
```

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

UCS

In [44]:

```
"""
Uniform Cost Search Implementation using PriorityQueue

Map and input taken from
http://www.massey.ac.nz/~mjjohnso/notes/59302/l04.html

Author: Jayesh Chandrapal
Version: 1.0
"""

import queue as Q

def search(graph, start, end):
    if start not in graph:
        raise TypeError(str(start) + ' not found in graph !')
        return
    if end not in graph:
        raise TypeError(str(end) + ' not found in graph !')
        return

    queue = Q.PriorityQueue()
    queue.put((0, [start]))

    while not queue.empty():
        node = queue.get()
        current = node[1][len(node[1]) - 1]

        if end in node[1]:
            print("Path found: " + str(node[1]) + ", Cost = " + str(node[0]))
            break

        cost = node[0]
        for neighbor in graph[current]:
            temp = node[1][:]
            temp.append(neighbor)
            queue.put((cost + graph[current][neighbor], temp))
```

```
def readGraph():
    lines = int( input() )
    graph = {}

    for line in range(lines):
        line = input()

        tokens = line.split()
        node = tokens[0]
        graph[node] = {}
        print

        for i in range(1, len(tokens) - 1, 2):
            print(node, tokens[i], tokens[i + 1])
            #graph.addEdge(node, tokens[i], int(tokens[i + 1]))
            #graph[node][tokens[i]] = int(tokens[i + 1])
    return graph


"""
Sample Map Input:

14
Arad Zerind 75 Timisoara 118 Sibiu 140
Zerind Oradea 71 Arad 75
Timisoara Arad 118 Lugoj 111
Sibiu Arad 140 Oradea 151 Fagaras 99 RimnicuVilcea 80
Oradea Zerind 71 Sibiu 151
Lugoj Timisoara 111 Mehadia 70
RimnicuVilcea Sibiu 80 Pitesti 97 Craiova 146
Mehadia Lugoj 70 Dobreta 75
Craiova Dobreta 120 RimnicuVilcea 146 Pitesti 138
Pitesti RimnicuVilcea 97 Craiova 138 Bucharest 101
Fagaras Sibiu 99 Bucharest 211
Dobreta Mehadia 75 Craiova 120
Bucharest Fagaras 211 Pitesti 101 Giurgiu 90
Giurgiu Bucharest 90
"""
```

Out[44]:

```
'\nSample Map Input:\n\n14\nArad Zerind 75 Timisoara 118 Sibiu 140\nZerind Oradea 71 Ara
d 75\nTimisoara Arad 118 Lugoj 111\nSibiu Arad 140 Oradea 151 Fagaras 99 RimnicuVilcea 8
0\nOradea Zerind 71 Sibiu 151\nLugoj Timisoara 111 Mehadia 70\nRimnicuVilcea Sibiu 80 Pi
testi 97 Craiova 146\nMehadia Lugoj 70 Dobreta 75\nCraiova Dobreta 120 RimnicuVilcea 146
Pitesti 138\nPitesti RimnicuVilcea 97 Craiova 138 Bucharest 101\nFagaras Sibiu 99 Buchar
est 211\nDobreta Mehadia 75 Craiova 120\nBucharest Fagaras 211 Pitesti 101 Giurgiu 90\nG
iurgiu Bucharest 90\n'
```

In [45]:

```
graph = readGraph()
```

```
2
Arad 75
Arad Zerind 75
Arad Zerind 75
```

Out[45]:

```
dict
```

In [40]:

```
search(graph, 'Arad', 'Bucharest')
```

```
Path found: ['Arad', 'Sibiu', 'RimnicuVilcea', 'Pitesti', 'Bucharest'], Cost = 418
```

A*

In [ ]:

```

```