In [3]:
```
from queue import PriorityQueue
v = 14
graph = [[] for i in range(v)]
# Function For Implementing Best First Search
# Gives output path having lowest cost
def best_first_search(actual_Src, target, n):
    visited = [False] * n
    pq = PriorityQueue()
    pq.put((0, actual_Src))
    visited[actual_Src] = True
    while pq.empty() == False:
        u = pq.get()[1]
        # Displaying the path having lowest cost
        print(u, end=" ")
        if u == target:
            break
        for v, c in graph[u]:
            if visited[v] == False:
                visited[v] = True
                pq.put((c, v))
    print()
# Function for adding edges to graph
def addedge(x, y, cost):
    graph[x].append((y, cost))
    graph[y].append((x, cost))
# The nodes shown in above example(by alphabets) are
# implemented using integers addedge(x,y,cost);
addedge(0, 1, 3)
addedge(0, 2, 6)
addedge(0, 3, 5)
addedge(1, 4, 9)
addedge(1, 5, 8)
addedge(2, 6, 12)
addedge(2, 7, 14)
addedge(3, 8, 7)
addedge(8, 9, 5)
addedge(8, 10, 6)
addedge(9, 11, 1)
addedge(9, 12, 10)
addedge(9, 13, 2)
source = 0
target = 6
best_first_search(source, target, v)
```

```
0 1 3 2 8 9 11 13 10 5 4 12 6
```

In [6]:
```
from collections import deque
class Graph:
    def __init__(self, adjacency_list):
        self.adjacency_list = adjacency_list
    def get_neighbors(self, v):
        return self.adjacency_list[v]
```

```
    # heuristic function with equal values for all nodes
    def h(self, n):
        H = {
            'A': 1,
            'B': 1,
            'C': 1,
            'D': 1
        }
        return H[n]
    def a_star_algorithm(self, start_node, stop_node):
        # open_list is a list of nodes which have been visited, but
        # haven't all been inspected, starts off with the start nod
        # closed_list is a list of nodes which have been visited
        # and who's neighbors have been inspected
        open_list = set([start_node])
        closed_list = set([])
        # g contains current distances from start_node to all other
        # the default value (if it's not found in the map) is +infi
        g = {}
        g[start_node] = 0
        # parents contains an adjacency map of all nodes
        parents = {}
        parents[start_node] = start_node
        while len(open_list) > 0:
            n = None
            # find a node with the lowest value of f() - evaluation
            for v in open_list:
                if n == None or g[v] + self.h(v) < g[n] + self.h(n)
                    n = v;
            if n == None:
                print('Path does not exist!')
                return None
            # if the current node is the stop_node
            # then we begin reconstructin the path from it to the s
            if n == stop_node:
                reconst_path = []
                while parents[n] != n:
                    reconst_path.append(n)
                    n = parents[n]
                reconst_path.append(start_node)
                reconst_path.reverse()
                print('Path found: {}'.format(reconst_path))
                return reconst_path
            # for all neighbors of the current node do
            for (m, weight) in self.get_neighbors(n):
                # if the current node isn't in both open_list and c
                # add it to open_list and note n as it's parent
                if m not in open_list and m not in closed_list:
                    open_list.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight
                # otherwise, check if it's quicker to first visit n
                # and if it is, update parent data and g data
                # and if the node was in the closed_list, move it t
```

```
                # and if the node was in the closed_list, move it t
                else:
                    if g[m] > g[n] + weight:
                        g[m] = g[n] + weight
                        parents[m] = n
                        if m in closed_list:
                            closed_list.remove(m)
                            open_list.add(m)
            # remove n from the open_list, and add it to closed_lis
            # because all of his neighbors were inspected
            open_list.remove(n)
            closed_list.add(n)
        print('Path does not exist!')
        return None
adjacency_list = {
    'A': [('B', 1), ('C', 3), ('D', 7)],
    'B': [('D', 5)],
    'C': [('D', 12)]
}
graph1 = Graph(adjacency_list)
graph1.a_star_algorithm('A', 'D')
```

```
Path found: ['A', 'B', 'D']
```

Out[6]: ['A', 'B', 'D']

In [ ]: