In [1]:

```
import heapq
li = [5,7,9,1,3]
heapq.heapify(li)

print("the created heap is: ", end=" ")
print(list(li))
```

```
the created heap is:  [1, 3, 9, 7, 5]
```

In [2]:

```
heapq.heappush(li, 4)
print("the modified heap push after push is: ", end=" ")
print(list(li))
```

```
the modified heap push after push is:  [1, 3, 4, 7, 5, 9]
```

In [5]:

```
import heapq
from collections import defaultdict

def shortestPath(graph, src, dest):
    # Priority queue for the Dijkstra's algorithm (min-heap)
    h = []
    # Dictionary to store the shortest path cost to each vertex
    dist = {src: 0}
    # Dictionary to store the previous vertex on the shortest path
    parent = {src: None}

    # Push the source into the heap with cost 0
    heapq.heappush(h, (0, src))

    while h:
        # Pop the vertex with the smallest distance (cost)
        curr_cost, curr_vtx = heapq.heappop(h)

        # If we reach the destination, reconstruct the path
        if curr_vtx == dest:
            path = []
            while curr_vtx is not None:
                path.append(curr_vtx)
                curr_vtx = parent[curr_vtx]
            path.reverse()
            print(f"Path Exists from {src} to {dest} with cost {curr_co
            print("Path:", " -> ".join(path))
            return

        # Visit each neighbor of the current vertex
        for neigh, neigh_cost in graph[curr_vtx]:
            new_cost = curr_cost + neigh_cost
            # If the new cost is better (smaller), update and push to h
            if neigh not in dist or new_cost < dist[neigh]:
                dist[neigh] = new_cost
                parent[neigh] = curr_vtx
                heapq.heappush(h, (new_cost, neigh))

    print(f"No path exists from {src} to {dest}.")

# Input graph
graph = defaultdict(list)
v, e = map(int, input("Enter No. of vertices and edges: ").split())
for _ in range(e):
    u, v, w = map(str, input("Enter edge and weight (u v w): ").split()
    graph[u].append((v, int(w)))

src, dest = map(str, input("Enter Source and Destination: ").split())

# Call the shortestPath function
shortestPath(graph, src, dest)
```

```
Enter No. of vertices and edges: 6 9
Enter edge and weight (u v w): a b 18
Enter edge and weight (u v w): a d 15
Enter edge and weight (u v w): d b 6
Enter edge and weight (u v w): d e 7
Enter edge and weight (u v w): d c 14
Enter edge and weight (u v w): b c 9
Enter edge and weight (u v w): e c 10
Enter edge and weight (u v w): c f 28
Enter edge and weight (u v w): e f 36
Enter Source and Destination: a f
Path Exists from a to f with cost 55.
Path: a -> b -> c -> f
```

In [6]:
```python
INF = float('inf')

def printmatrix(m):
    # Print a matrix in a readable format
    r, c = len(m), len(m[0])
    for i in range(r):
        for j in range(c):
            # Print each element with a width of 5 for better alignment
            if m[i][j] == INF:
                print("INF", end="\t")
            else:
                print(f"{m[i][j]:5}", end="\t")
        print()

def floydWarshall(v, e):
    # Create a distance matrix with infinity and 0 on the diagonal
    m = [[INF] * v for _ in range(v)]

    for i in range(v):
        m[i][i] = 0  # Distance from a vertex to itself is 0

    # Take input edges and weights
    for _ in range(e):
        src, dest, wt = map(int, input("Enter edge (src, dest, weight):
        m[src][dest] = wt

    print("Initial Matrix:")
    printmatrix(m)
    print("..............................")

    # Floyd-Warshall Algorithm: O(v^3) time complexity
    for k in range(v):
        for i in range(v):
            for j in range(v):
                # If a shorter path exists through vertex k, update the
                if m[i][k] + m[k][j] < m[i][j]:
                    m[i][j] = m[i][k] + m[k][j]

        # After each k iteration, print the updated matrix
        print(f"After iteration {k+1}:")
        printmatrix(m)
        print("..............................")

    return m

def main():
    # Input number of vertices and edges
    v, e = map(int, input("Enter number of Vertices and Edges: ").split

    # Run Floyd-Warshall Algorithm
    m = floydWarshall(v, e)

    # You can further display the final shortest path matrix:
    print("Final Shortest Path Matrix:")
    printmatrix(m)

if __name__ == "__main__":
    main()
```

```
59
```

```
Enter number of Vertices and Edges: 4 4
Enter edge (src, dest, weight): 0 3 10
Enter edge (src, dest, weight): 0 1 5
Enter edge (src, dest, weight): 1 2 3
Enter edge (src, dest, weight): 2 3 1
Initial Matrix:
    0       5   INF      10
INF       0       3   INF
INF     INF       0       1
INF     INF     INF       0
..............................
After iteration 1:
    0       5   INF      10
INF       0       3   INF
INF     INF       0       1
INF     INF     INF       0
.............................
After iteration 2:
    0       5       8      10
INF       0       3   INF
INF     INF       0       1
INF     INF     INF       0
.............................
After iteration 3:
    0       5       8       9
INF       0       3       4
INF     INF       0       1
INF     INF     INF       0
.............................
After iteration 4:
    0       5       8       9
INF       0       3       4
INF     INF       0       1
INF     INF     INF       0
.............................
Final Shortest Path Matrix:
    0       5       8       9
INF       0       3       4
INF     INF       0       1
INF     INF     INF       0
```

In [ ]:
```
1
```