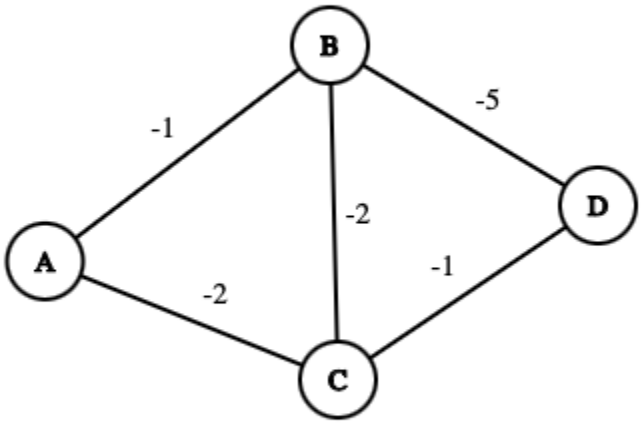
1. **Show that the program Dijkstra does not work correctly if arc costs can be negative.**

   The algorithm assumes that once a vertex is selected as having the minimum distance from the source, this distance cannot decrease in subsequent algorithm steps. This assumption is valid only if all edge weights are non-negative. If an edge with a negative weight is present, the algorithm might select a vertex too early before finding a path leading to a shorter distance through a negative-weight edge.

   Example of a graph where it fails:



   Initialize predecessor array (π) with a length of 4 and all elements set to ø and distance array (d) with a length of 4 and all elements ∞

   Starting node: Let us choose A, i.e. d[A] = 0

   Initialize priority queue as a function of d

   Iteration1: Current Node = A
   d[B] = -1 π[B]=A
   d[C] = -2 π[C]=A

   Iteration2: Current Node = C (∵ d[C] < d[B])
   d[B] = -2 + (-2) = -4 π[B]=C
   d[D] = -2 + (-1) = -3 π[D]=C

   Iteration3: Current Node = B (∵ d[B] < d[D])
   d[D] = -4 + (-5) = -9 π[D]=B

   We get a tree:
   A -> C -> B -> D

Here
d[B] = -4

There is a lesser cost to reach B.

Just by looking at the graph, we can tell that the least possible value of d[B] is -8 if we traverse
A -> C -> D -> B

However, the greedy part of the algorithm does not see this.

∴ The program Dijkstra may not work correctly if arc costs are negative.

2. **Describe an algorithm to insert and delete edges in the adjacency list representation for an undirected graph. Remember that an edge (i, j) appears on the vertex i and j adjacency list.**

   Insert:
   - **Check for an Existing Edge:** Before inserting, optionally check if the edge already exists between vertices i and j to prevent duplication. This step involves searching through the adjacency list of vertex i to see if j is present and vice versa.
   - **Insert Edge:** Add j to the adjacency list of vertex i and i to the adjacency list of vertex j. This step ensures that the edge is represented in both vertices' adjacency lists, adhering to the undirected nature of the graph.

   Pseudocode:

```
function insertEdge(Graph, i, j):
    if j not in Graph[i]:
        Graph[i].append(j)
    if i not in Graph[j]:
        Graph[j].append(i)
```

   Delete:
   - **Check for the Edge's Existence:** Verify that the edge between vertices i and j exists. This step involves searching through the adjacency list of vertex i for j and vice versa.
   - **Remove Edge:** If the edge exists, remove j from the adjacency list of vertex i and i from the adjacency list of vertex j. This operation removes the edge from both vertices' adjacency lists.

   Pseudocode:

```
function deleteEdge(Graph, i, j):
    if j in Graph[i]:
        Graph[i].remove(j)
    if i in Graph[j]:
        Graph[j].remove(i)
```

3. **Modify the adjacency list representation for an undirected graph so that the first edge on the adjacency list for a vertex can be deleted in constant time. Using your new representation, write an algorithm to delete the first edge at a vertex. Hint. How do you arrange the two cells representing edge (i, j) so that they can be found quickly from one another?**

   Modified Adjacency List Representation
   Adjacency List as a Double-Linked List: Each vertex's adjacency list is a double-linked list where each node contains the following:
   - The target vertex of the edge.
   - A reference (or pointer) to the corresponding node in the target vertex's adjacency list.
   - Standard double-linked list pointers: `next` and `prev` to navigate to the next and previous nodes in the list.

   Reference to Corresponding Node: When an edge (i, j) is inserted, the node in i's adjacency list that points to j will also reference the node in j's adjacency list that points to i, and vice versa. This mutual reference allows for constant access to the corresponding node when deletion is necessary.

   Algorithm to Delete the First Edge at a Vertex (Pseudocode)

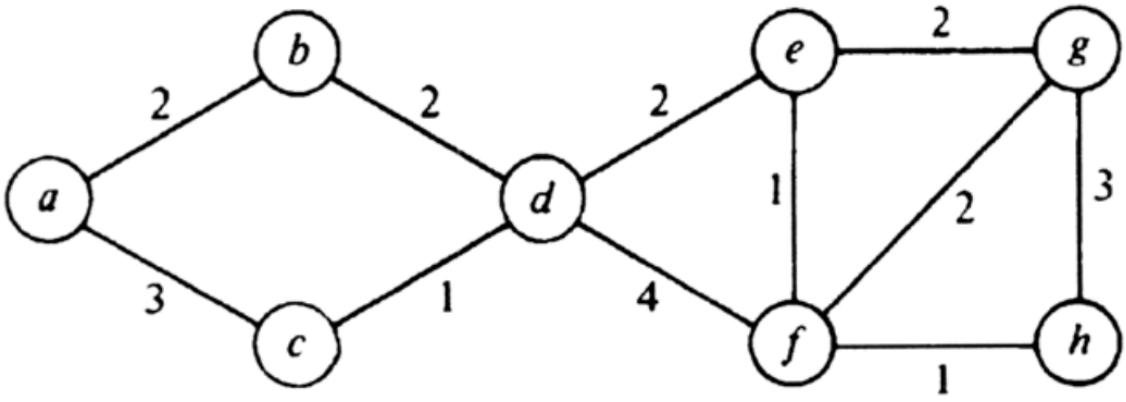
```
function deleteFirstEdge(Graph, i):
    if Graph[i] is empty:
        return

    firstEdgeNode = Graph[i].head
    correspondingNode = firstEdgeNode.correspondingNode

    # Delete the node from i's adjacency list
    if firstEdgeNode.next is not null:
        firstEdgeNode.next.prev = null
    Graph[i].head = firstEdgeNode.next  # Update the head of the list

    # Delete the corresponding node from j's adjacency list
    if correspondingNode.next is not null:
        correspondingNode.next.prev = correspondingNode.prev
    if correspondingNode.prev is not null:
        correspondingNode.prev.next = correspondingNode.next
    else:
        Graph[correspondingNode.vertex].head = correspondingNode.next  # Update the
head if it was the first node
```

4.



a. Find a minimum-cost spanning tree by Prim's algorithm.

a↔b↔d↔e↔g
↕ ↕
c f↔h

b. Find a minimum-cost spanning tree by Kruskal's algorithm.

a↔b↔d↔e↔g
↕ ↕
c f↔h

c. Find a depth-first spanning tree starting at a and at d.

Starting at 'a':
a↔b↔d↔c
↕
e↔f↔g↔h

Starting at 'd':
d↔b↔a↔c
↕
e↔f↔g↔h

d. Find a breadth-first spanning tree starting at a and at d.

Starting at ‘a’:

a↔b↔d↔e↔g↔h
↕ ↕
c f

Starting at ‘d’:

e↔g↔h
↕
a↔b↔d↔f
↕
c