# CSE 2202
# Design and Analysis of Algorithms – I
# **Lecture 6**
# **Euler Path, Minimum Spanning Tree (Kruskal and Prims)**

# EULER GRAPH AND CIRCUITS

# The Königsberg bridge problem

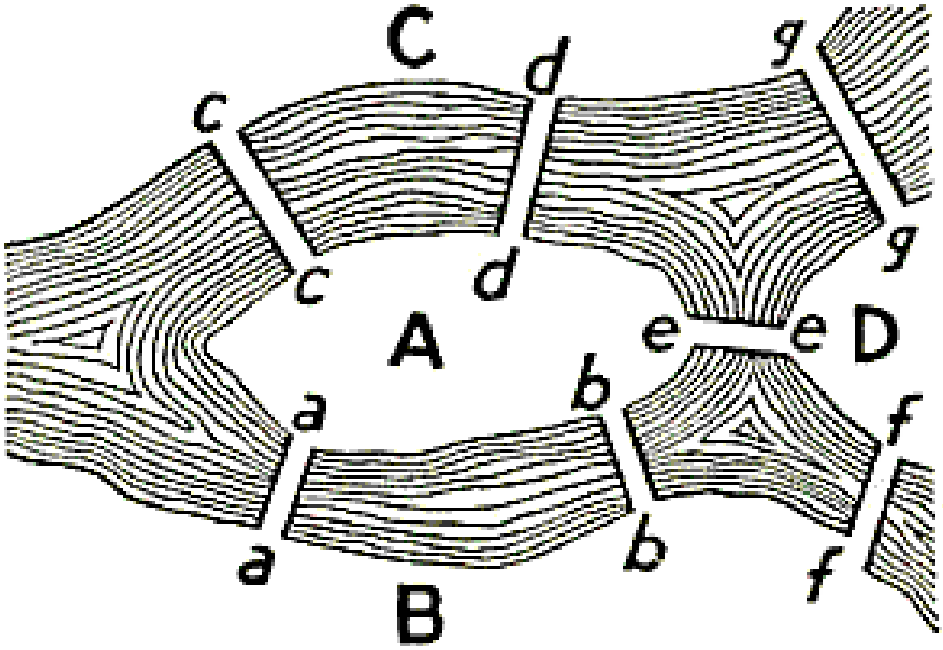
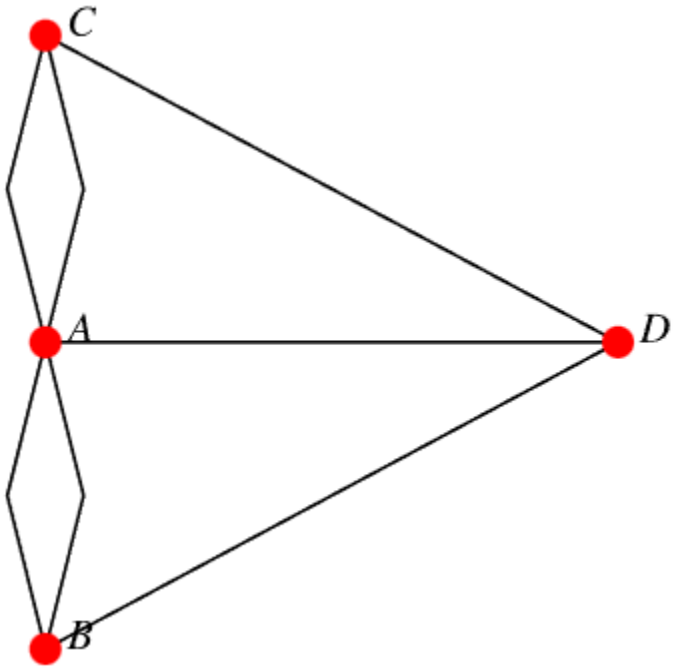
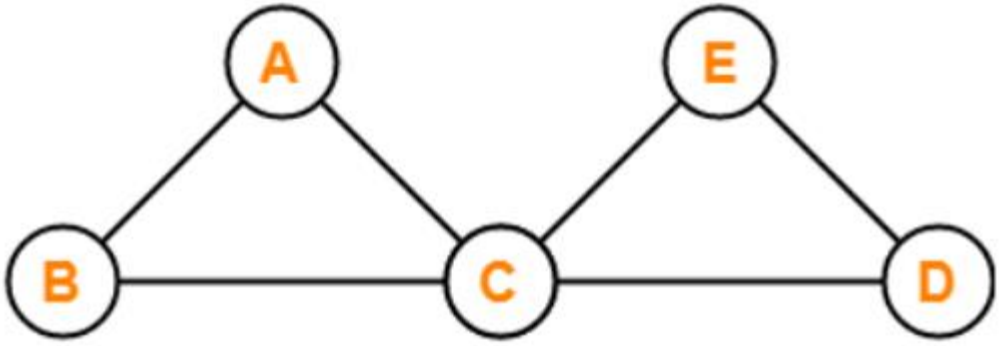



FIGURE 98. *Geographic Map: The Königsberg Bridges.*



# Walk and Trail

- A walk is defined as a finite length alternating sequence of vertices and edges.
- vertices and edges can repeat in a walk, but in trail, only vertices repetition is allowed – not edge repetition.



# Walk

- A walk in a graph is an alternating sequence of vertices and edges, starting and ending with a vertex, where:
  - Vertices represent points or nodes.
  - Edges represent connections or links between vertices.
- The **length** of a walk is the number of edges it traverses.
- In a walk, **both vertices and edges can repeat** any number of times.
- **Open or Closed:**
  - An open walk starts and ends at different vertices.
  - A closed walk starts and ends at the same vertex.

# Types of Walks

Depending on the characteristics and restrictions, walks can be further categorized:

**a. Trail:** A **trail** is a walk in which **no edge is repeated**. However, vertices can repeat. It can be open or closed.

**b. Path:** A **path** is a walk where **no vertex or edge is repeated** (unless it's closed, where the first and last vertices may be the same).

**c. Circuit:** A **circuit** is a **closed trail**, meaning it starts and ends at the same vertex but does not repeat any edge.

**d. Cycle :** A **cycle** is a closed path that starts and ends at the same vertex but doesn't repeat any other vertices or edges.

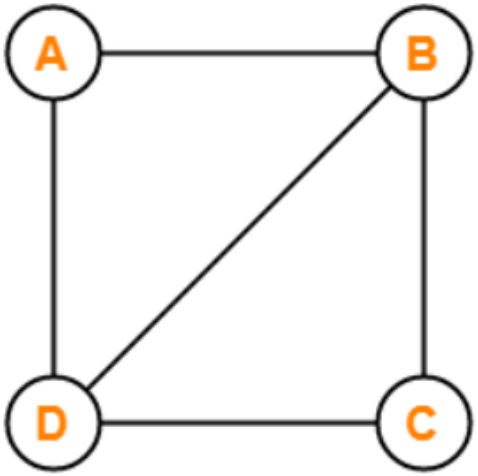
# Euler Path

- Euler path is also known as Euler Trail or Euler Walk.
- If there exists a Trail in the connected graph that contains **all the edges** of the graph, then that trail is called as an Euler trail.
  - Trail is an open walk in which no edge is repeated.
  - Vertex can be repeated.
- If there exists a walk in the connected graph that visits every edge of the graph exactly once **with or without repeating the vertices**, then such a walk is called as an Euler walk.

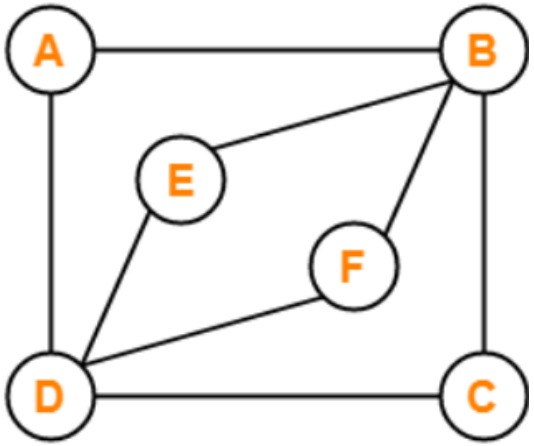
**Euler Path and Euler Circuit:** These are special types of paths and circuits that cover **every edge exactly once**:

- **Euler Path**: An open walk that covers every edge once.
- **Euler Circuit**: A closed walk that covers every edge once.

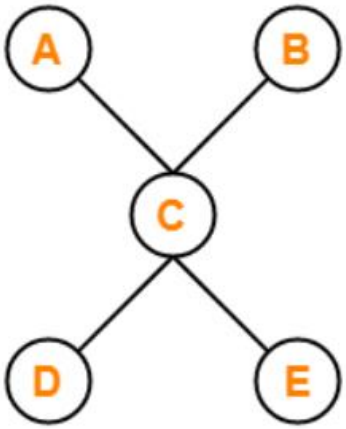
# Euler Path Examples



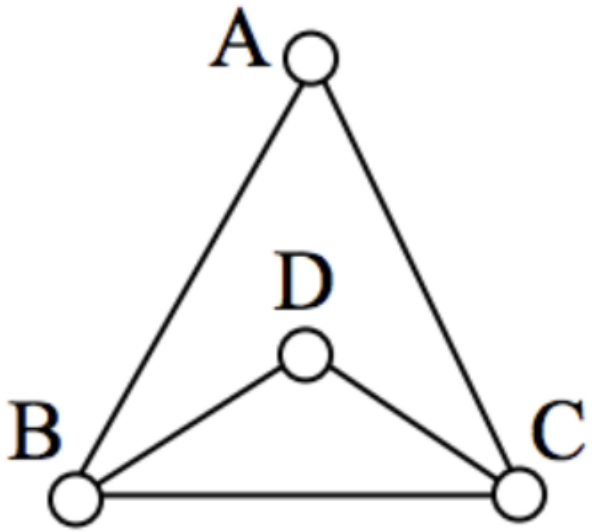

Euler Path = BCDBAD

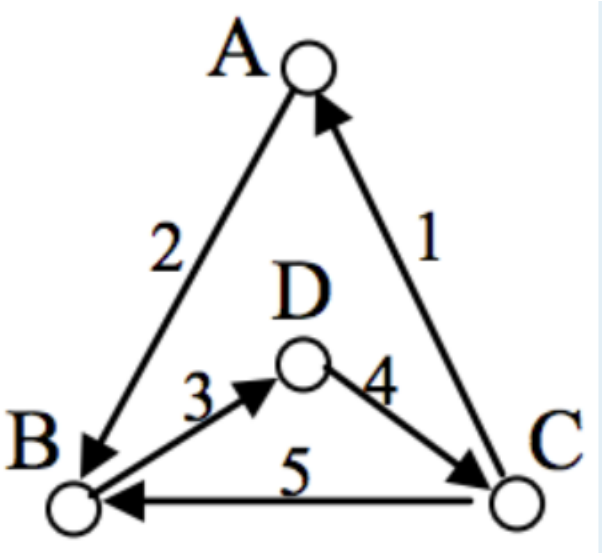



Euler Path = BCDFBEDAB




Euler Path Does Not Exist

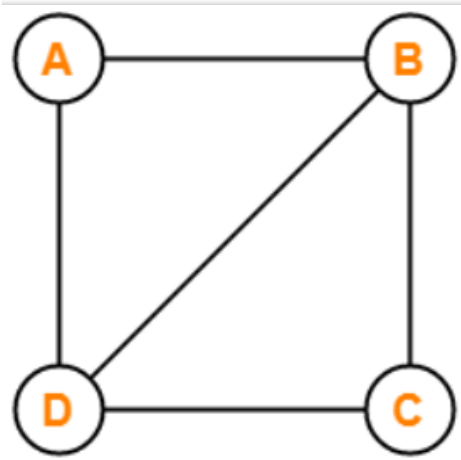


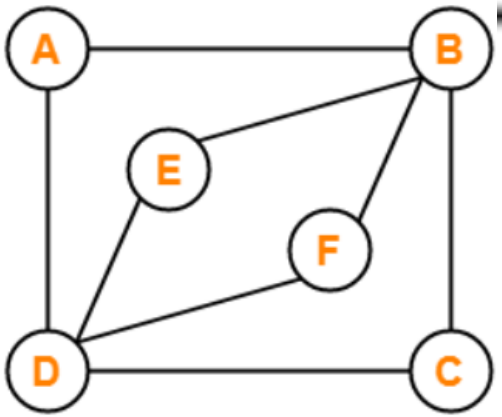
# Euler Circuit

- If there exists a walk in the connected graph
  - that starts and ends at the same vertex and
  - visits every edge of the graph exactly once with or without repeating the vertices,
- then such a walk is called as an Euler circuit.
- An Euler trail that starts and ends at the same vertex is called as an Euler circuit.
  - A closed Euler trail is called as an Euler circuit.

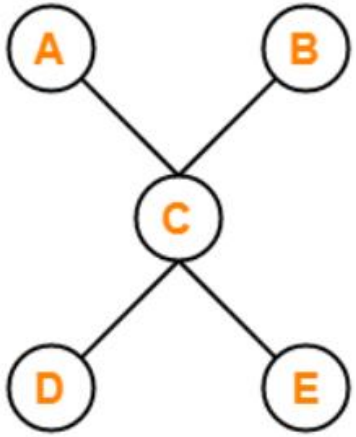
# Euler Circuit Examples




Euler Circuit Does Not Exist




Euler Circuit = ABCDFBEDA

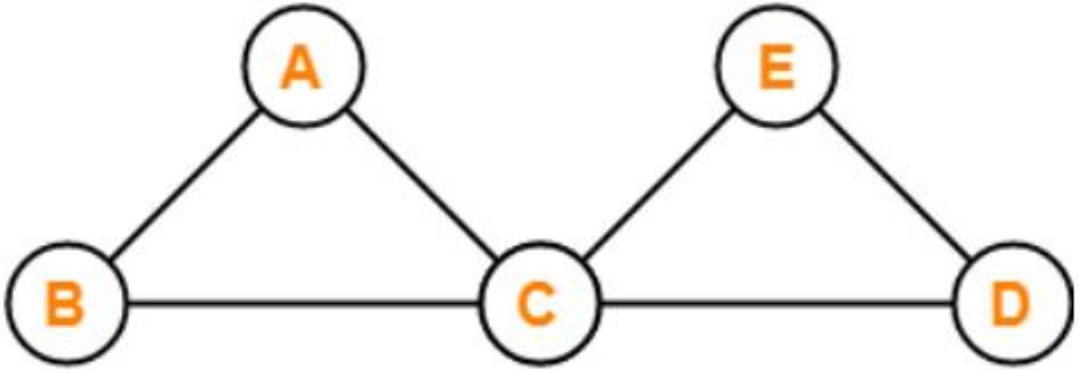



Euler Circuit Does Not Exist

**A graph will contain an Euler circuit if and only if all its vertices are of even degree.**

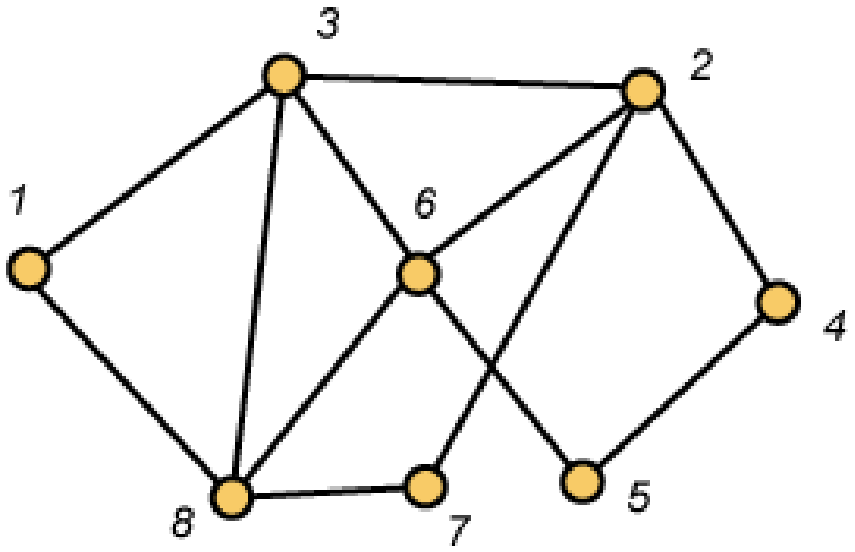
# Euler Graph

- Any connected graph is called as an Euler Graph if and only if all its **vertices** are of even degree.
- An Euler Graph is a connected graph that contains an Euler Circuit.




Example of Euler Graph

# Euler Circuits in Graphs



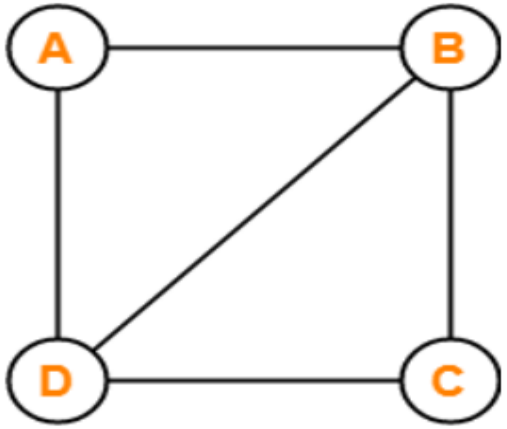

Here is an euler circuit for this graph:

(1,8,3,6,8,7,2,4,5,6,2,3,1)

# Semi Euler Graph

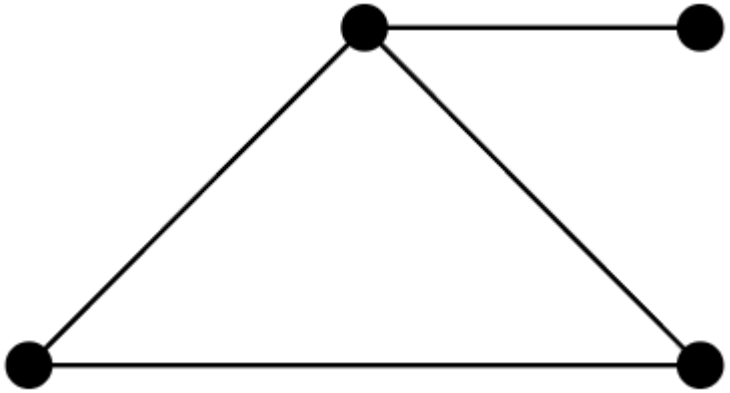
- If a connected graph contains an **Euler trail** but does not contain an **Euler circuit**, then such a graph is called as a semi-Euler graph.
- This graph contains an Euler trail BCDBAD.
- But it does not contain an Euler circuit.
- **Euler trail:** the number of vertices in the graph with odd degree are not more than 2.

A graph will contain an Euler path if it contains at most two vertices of odd degree.




Semi-Euler Graph

# Example

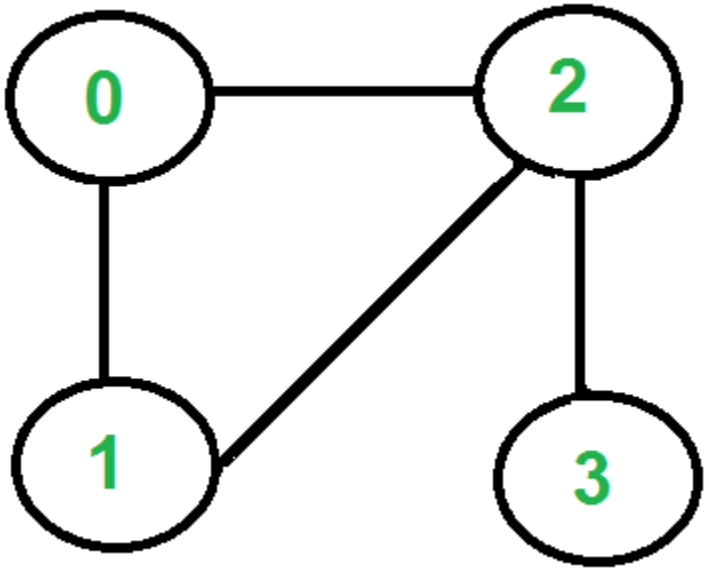


- A vertex with degree 1 is a dead end – unless you start there.
  - Even in that case, you can't do anything without repeating the edge..
- a graph has an **Euler path** and two vertices with odd degree, then the Euler path must **start at one of the odd degree** vertices and **end at the other**.
- In such a situation, every other vertex must have an even degree since we need an equal number of edges to get to those vertices as to leave them.
- But for a graph to have an Euler circuit, all vertices must have even degree.

# The Algorithm

**Printing the Eulerian trail or cycle**

- The graph should have either 0 or 2 odd vertices.
- If there are 0 odd vertices, start anywhere. If there are 2 odd vertices, start at one of them.
- Follow edges one at a time. If you have a choice between a bridge and a non-bridge, always choose the non-bridge.
- Stop when you run out of edges.

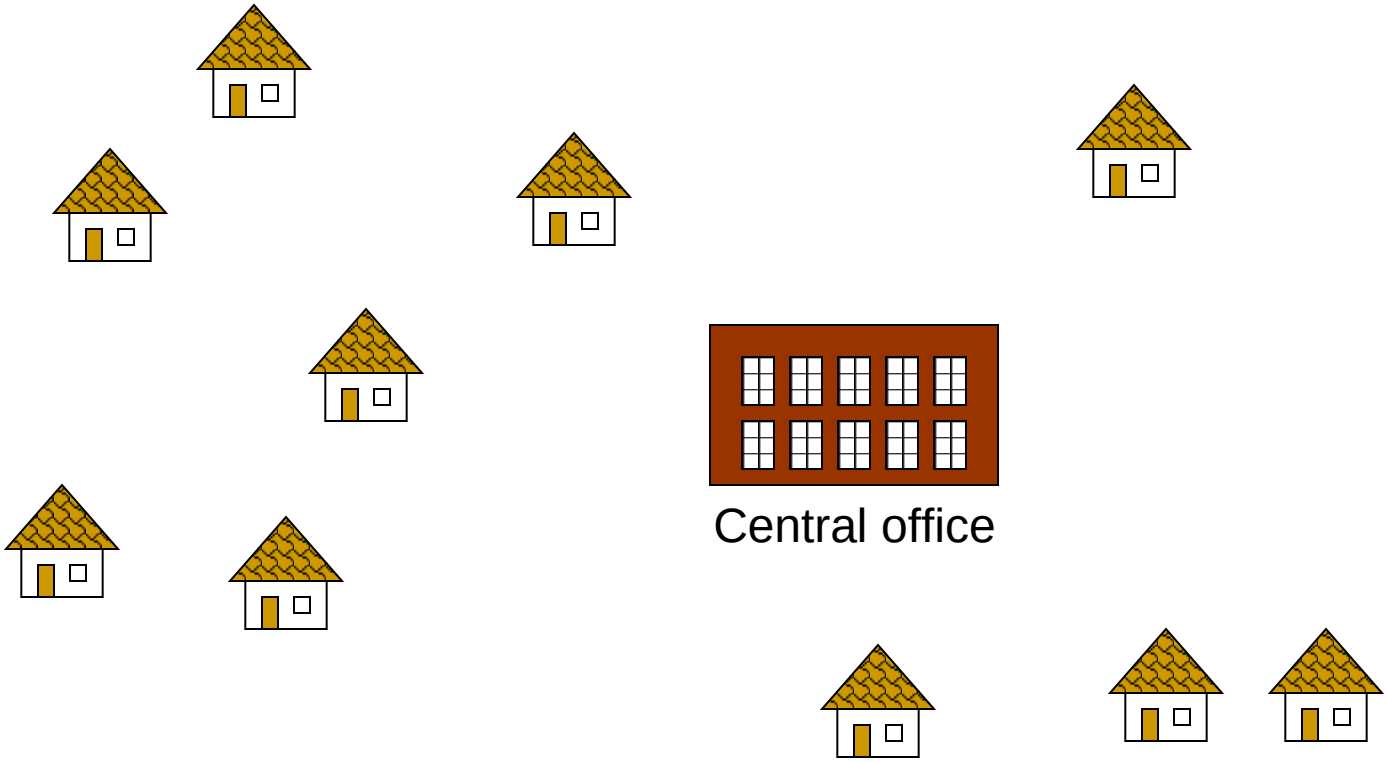


# Euler Circuit vs. Hamiltonian Cycle

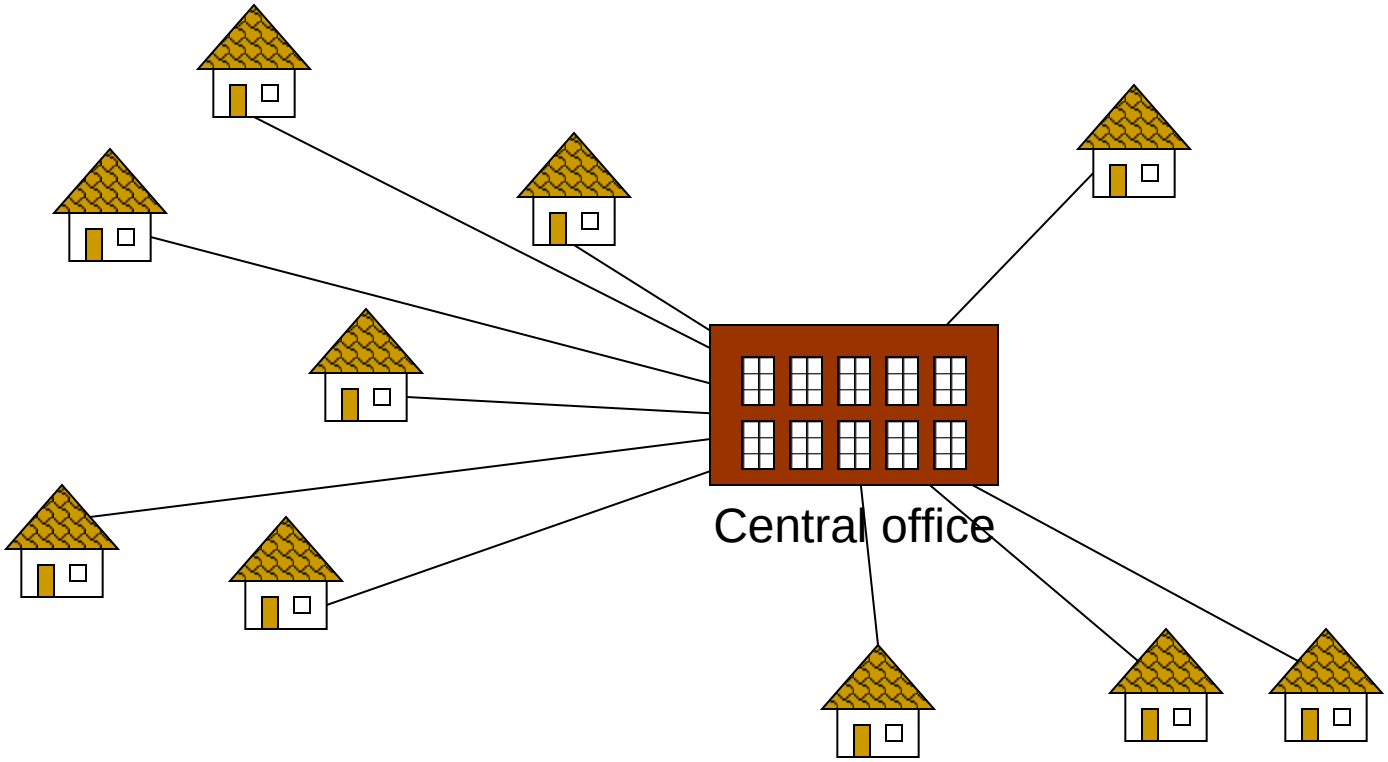
- An **Euler tour (circuit)** of a connected, directed graph G (V, E) is a cycle that traverses each edge of G exactly once, although it is allowed to visit each vertex more than once.
- Whether there is a path which visits every vertex exactly once. Such a path is called a **Hamilton path (or Hamiltonian path)**
- We could also consider Hamilton cycles, which are Hamilton paths which start and stop at the same vertex.
- Determining whether a directed graph has a hamiltonian cycle is NP-complete (will discuss this later).
- However, we can determine whether a graph has an Euler circuit in only O(E) time, in fact, we can find the edges of the circuit in O(E) time.

# MINIMUM SPANNING TREE

# Problem: Laying Telephone Wire

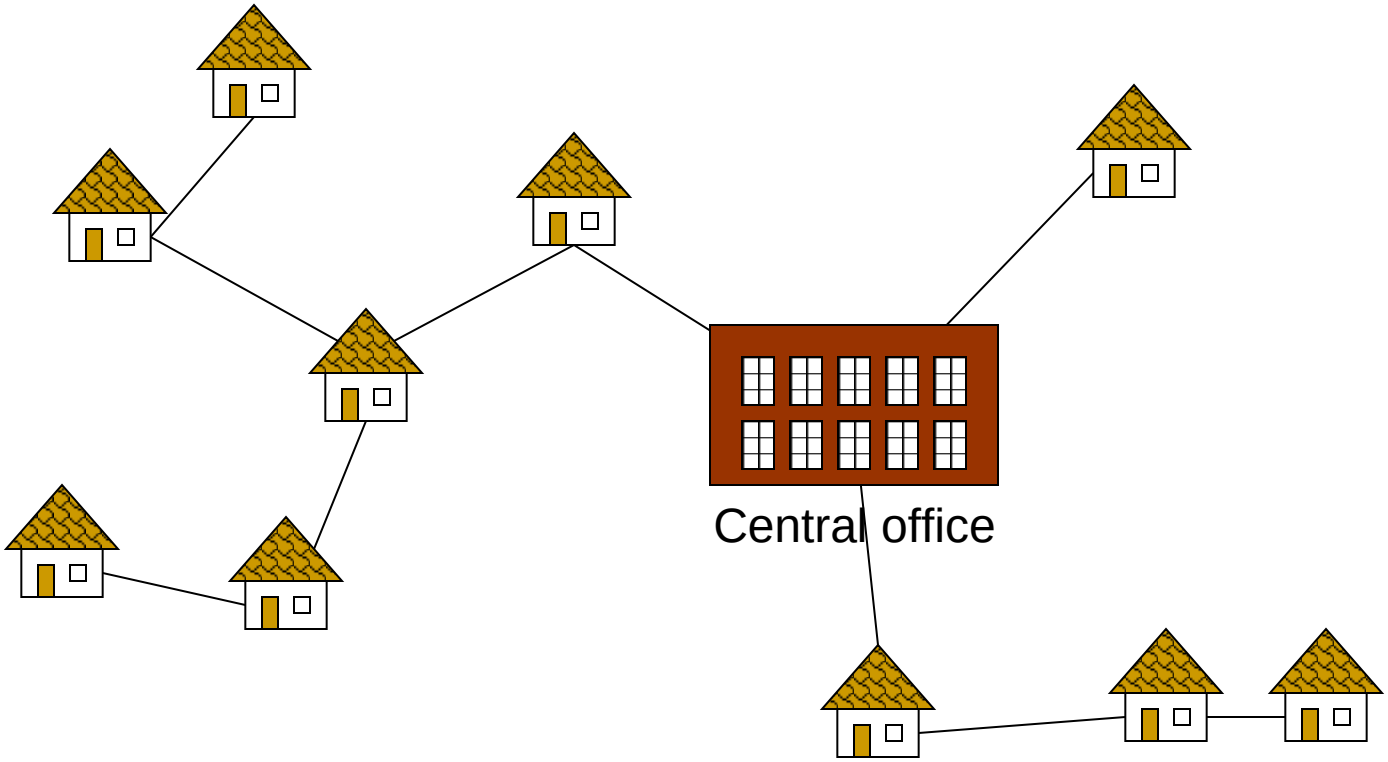


# Wiring: Naïve Approach



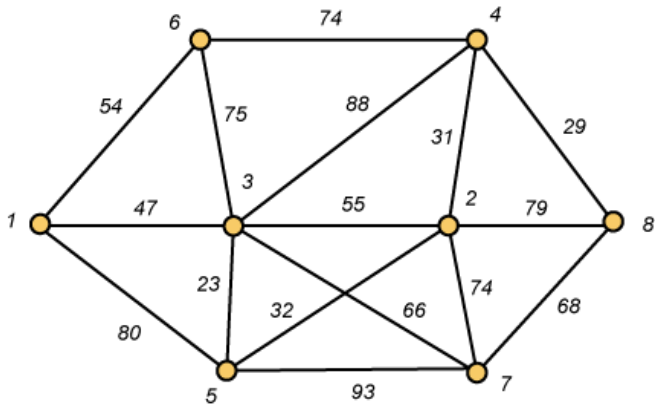

**Expensive!**

# Wiring: Better Approach



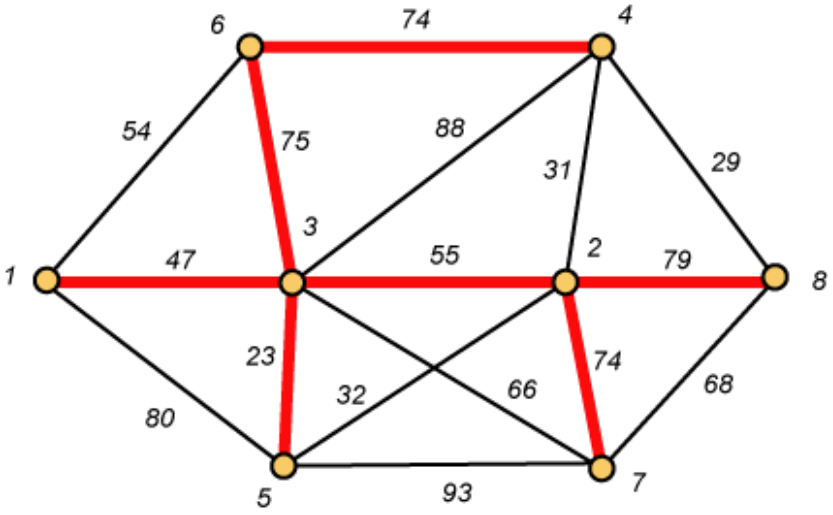

Minimize the total length of wire connecting the customers

# A Networking Problem




Problem: The vertices represent 8 regional data centers which need to be connected with high-speed data lines. Feasibility studies show that the links illustrated above are possible, and the cost in millions of dollars is shown next to the link. Which links should be constructed to enable full communication (with relays allowed) and keep the total cost minimal.

# Links Will Form a Spanning Tree

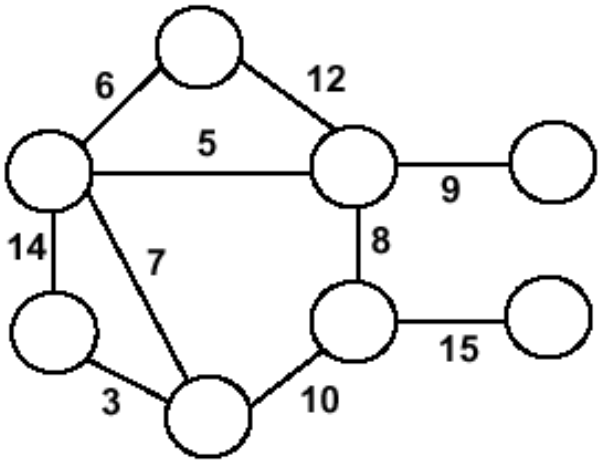



Cost (T) = 47 + 23 + 75 + 74 + 55 + 74 + 79

= 427

# Minimum Spanning Trees

- Undirected, connected graph $G = (V,E)$
- Weight function $W$: $E \rightarrow R$ (assigning cost or length or other values to edges)




- Spanning tree: tree that connects all the vertices (above?)
- Minimum spanning tree: tree that connects all the vertices and minimizes

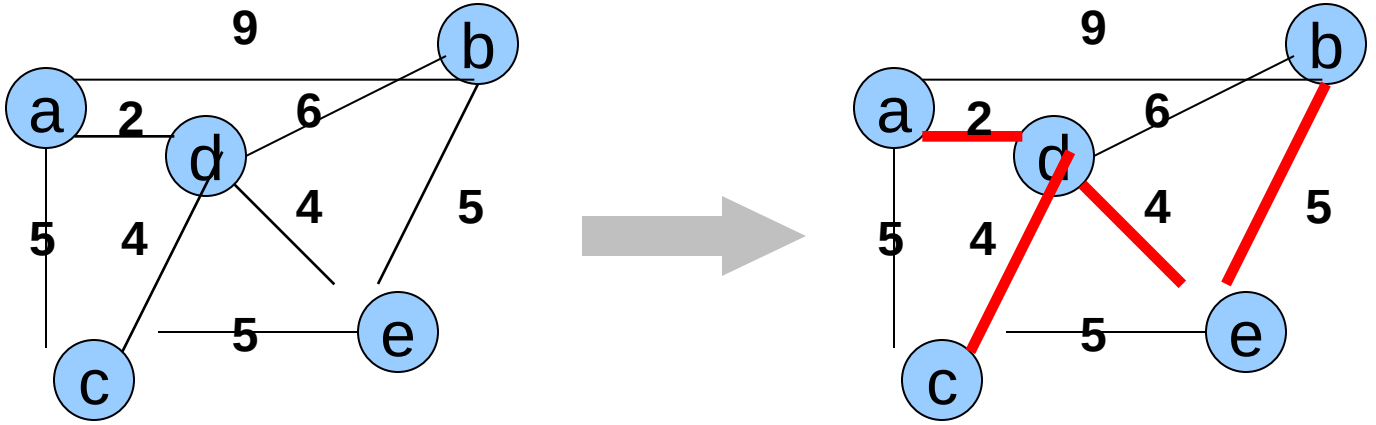
$$w(T) = \sum_{(u,v)\in T} w(u,v)$$

# Minimum Spanning Tree (MST)

A **minimum spanning tree** is a subgraph of an undirected weighted graph $G$, such that

- it is a tree (i.e., it is acyclic)
- it covers all the vertices $V$
  - contains $|V| - 1$ edges
- the total cost associated with tree edges is the minimum among all possible spanning trees
- not necessarily unique

# How Can We Generate a MST?



# Greedy Choice

We will show two ways to build a minimum spanning tree.

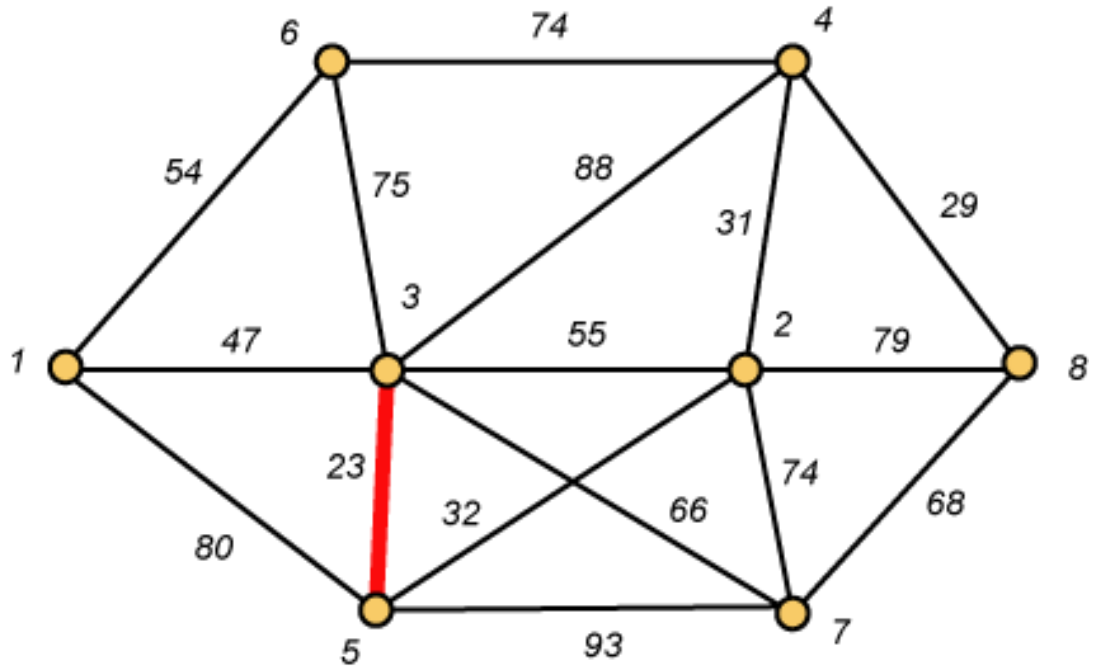
- A MST can be grown from the current spanning tree by adding the nearest vertex and the edge connecting the nearest vertex to the MST. (Prim's algorithm)

- A MST can be grown from a forest of spanning trees by adding the smallest edge connecting two spanning trees. (Kruskal's algorithm)

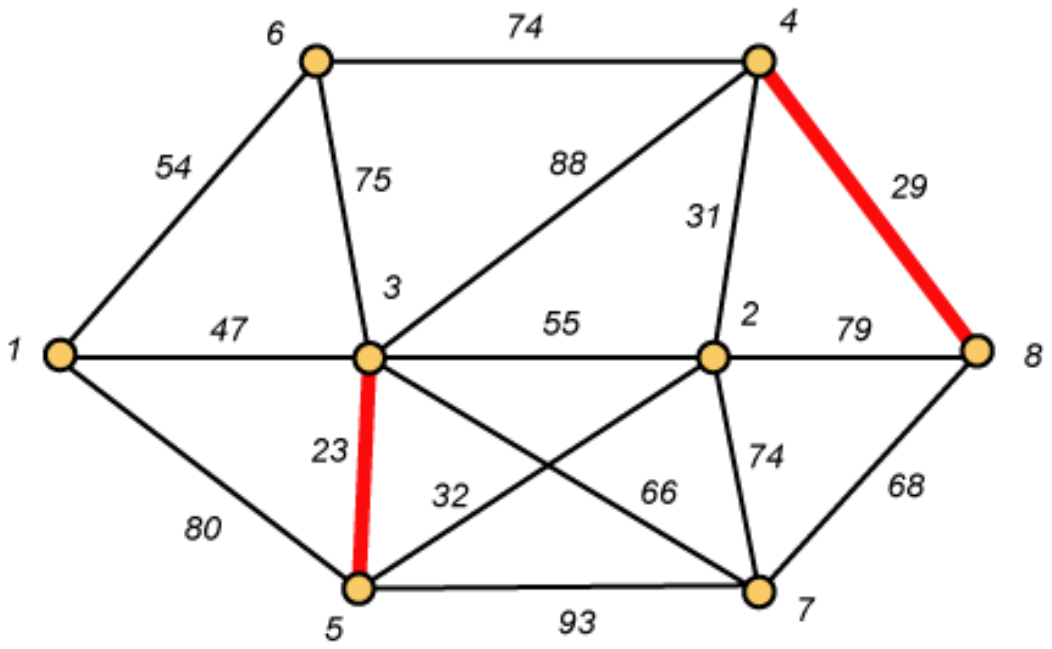
# Kruskal's Algorithm

- Edge based algorithm
- Add the edges one at a time, in increasing weight order
- The algorithm maintains *A* – a **forest of trees**. An edge is accepted it if connects vertices of distinct trees
- We need a data structure that maintains a partition, i.e.,a collection of disjoint sets
  - MakeSet(S,x) – simply creates a tree with just one node.
  - Union($S_i$,$S_j$) – causes a root of a one tree to point to the root of the other.
  - FindSet(S, x) – follows the parent's pointers until it reaches the root of the tree, and nodes visited in this simple path toward the root constitute a find path.

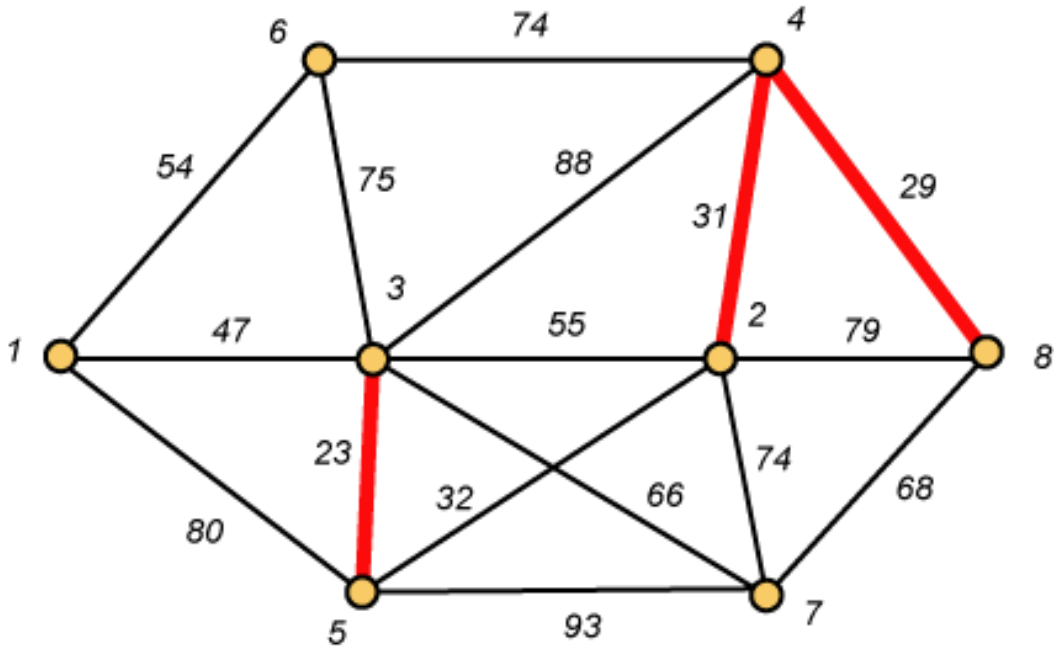
# Kruskal – Step 1



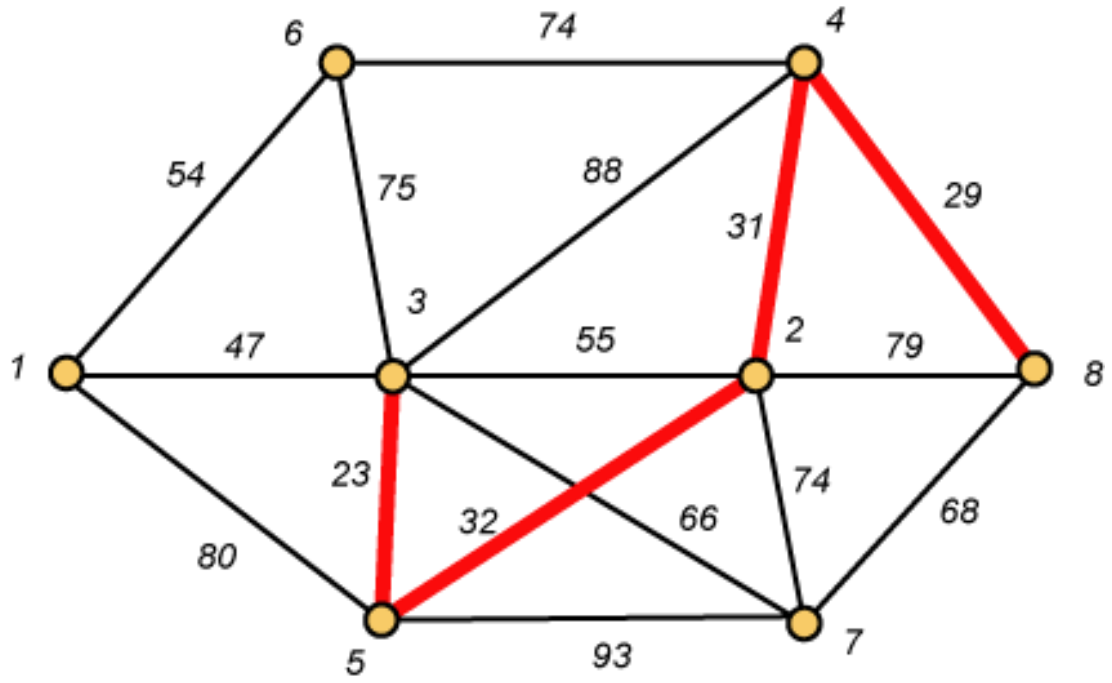
# Kruskal – Step 2



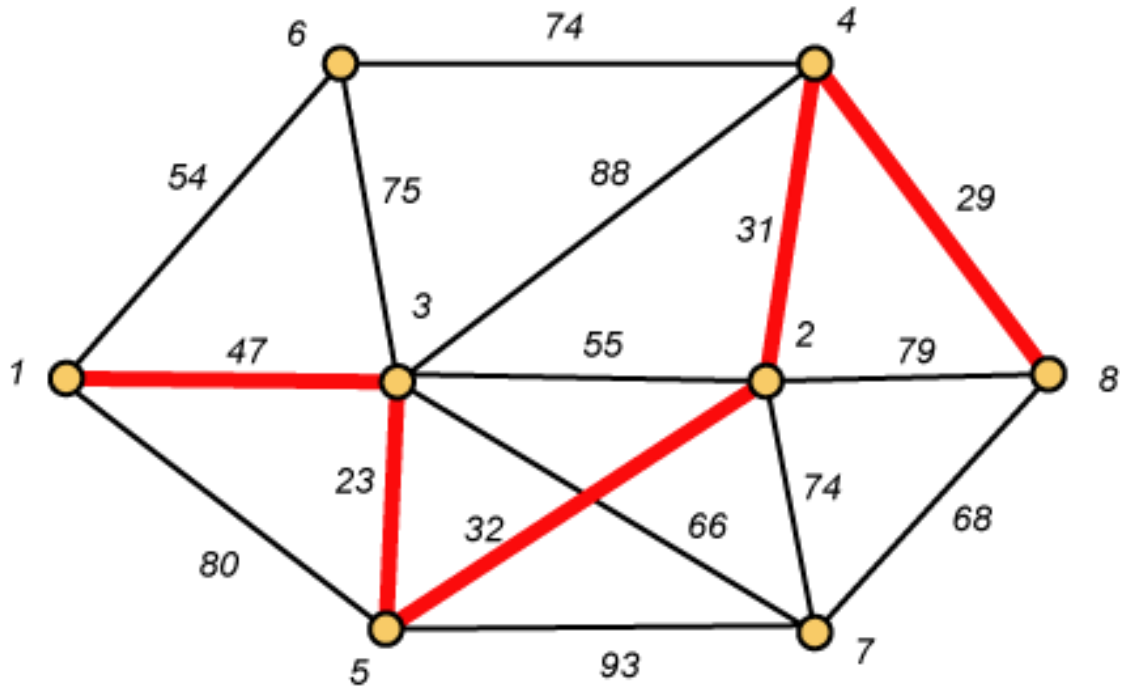
# Kruskal – Step 3



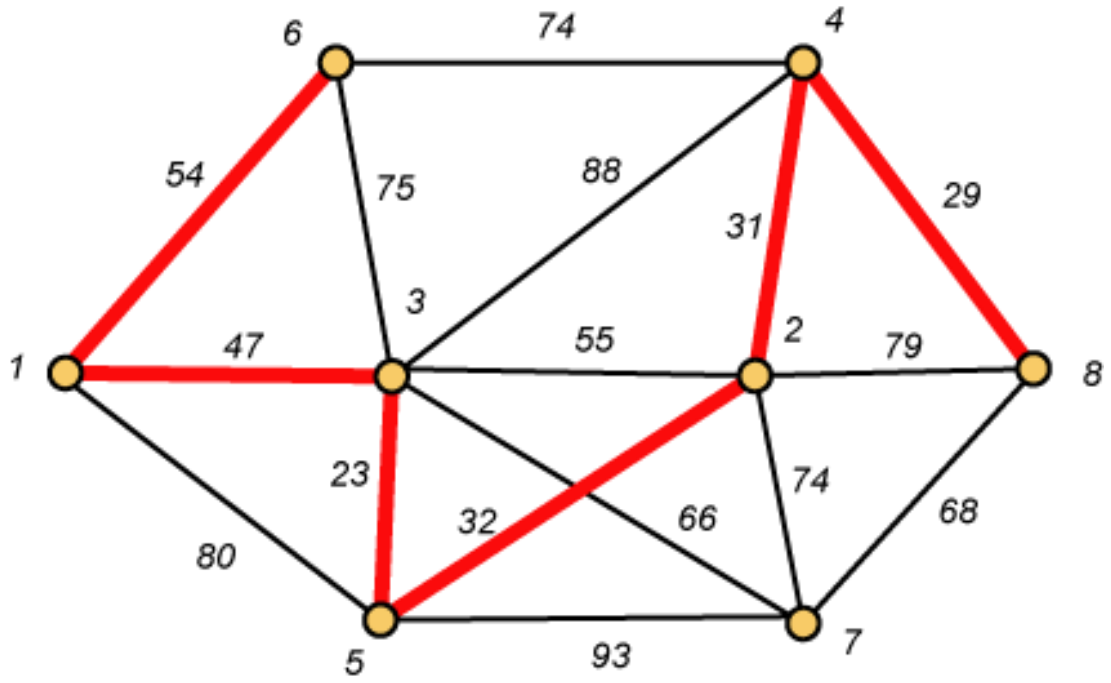
# Kruskal – Step 4



# Kruskal – Step 5



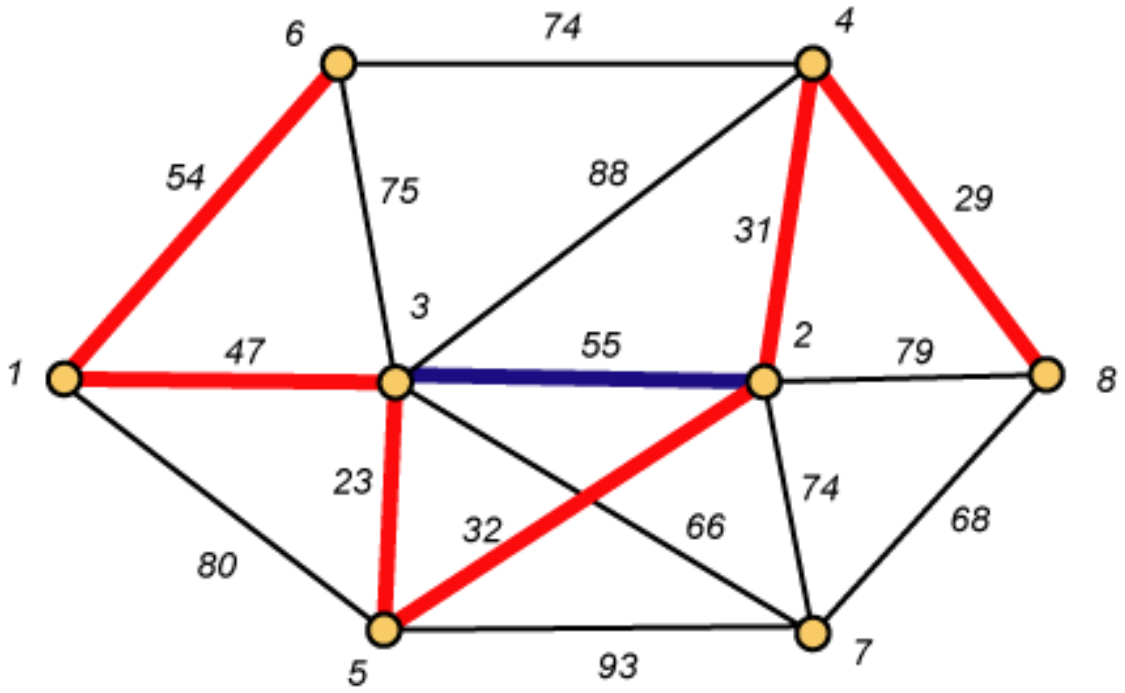
# Kruskal – Step 6



# Why Avoiding Cycles Matters

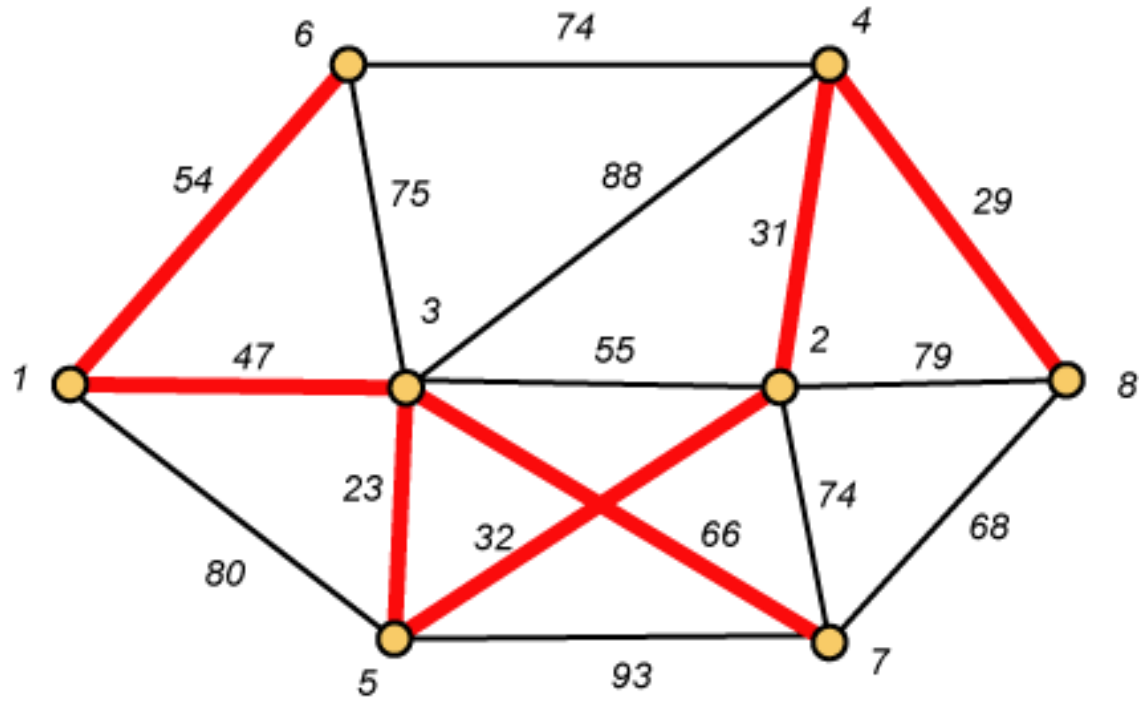
Up to this point, we have simply taken the edges in order of their weight. But now we will have to reject an edge since it forms a cycle when added to those already chosen.

# Forms a Cycle




So we cannot take the blue edge having weight 55.

# Kruskal – Step 7 *DONE!!*




Weight (T) = 23 + 29 + 31 + 32 + 47 + 54 + 66 = **282**

# Disjoint-Set

- Keep a collection of sets $S_1$, $S_2$, .., $S_k$,
  - Each $S_i$ is a set, e,g, $S_1$={$v_1$, $v_2$, $v_8$}.
- Three operations
  - Make-Set(x)-creates a new set whose only member is x.
  - Union(x, y) –unites the sets that contain x and y, say, $S_x$ and $S_y$, into a new set that is the union of the two sets.
  - Find-Set(x)-returns a pointer to the representative of the set containing x.

# Kruskal's Algorithm

- The algorithm adds the cheapest edge that connects two trees of the forest

```
MST-Kruskal(G,w)
01 A ← ∅
02 for each vertex v ∈ V[G] do                          O(V)
03    Make-Set(v)
04 sort the edges of E by non-decreasing weight w       O(ElogE)
05 for each edge (u,v) ∈ E, in order by non-            O(E)
   decreasing weight do
06    if Find-Set(u) ≠ Find-Set(v) then
07       A ← A ∪ {(u,v)}                                O(V)
08       Union(u,v)
09 return A
```

**Overall Complexity: O(VE)**

# Notation

- Tree-vertices: in the tree constructed so far
- Non-tree vertices: rest of vertices

## Prim's Selection rule

- Select the minimum weight edge between a tree-node and a non-tree node and add to the tree

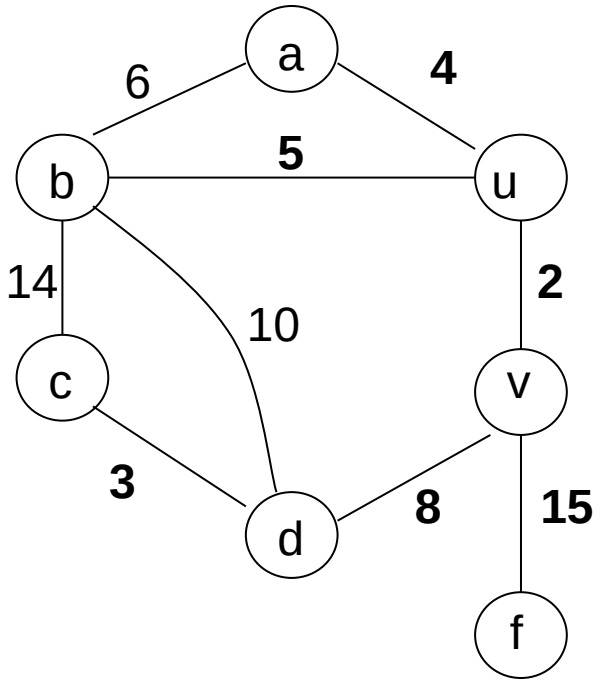
# The Prim algorithm Main Idea

Select a vertex to be a tree-node

**while** (there are non-tree vertices) {
    **if** there is no edge connecting a tree node with a non-tree node
        **return** "no spanning tree"

    select an edge of minimum weight between a tree node and a non-tree node

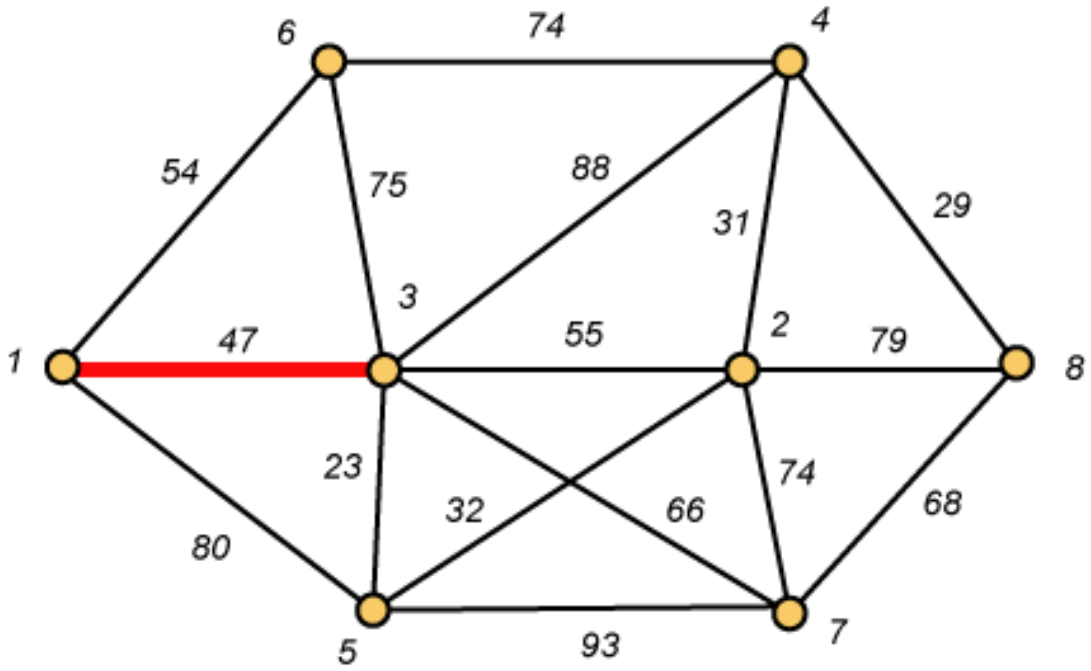
    add the selected edge and its new vertex to the tree
}
return tree



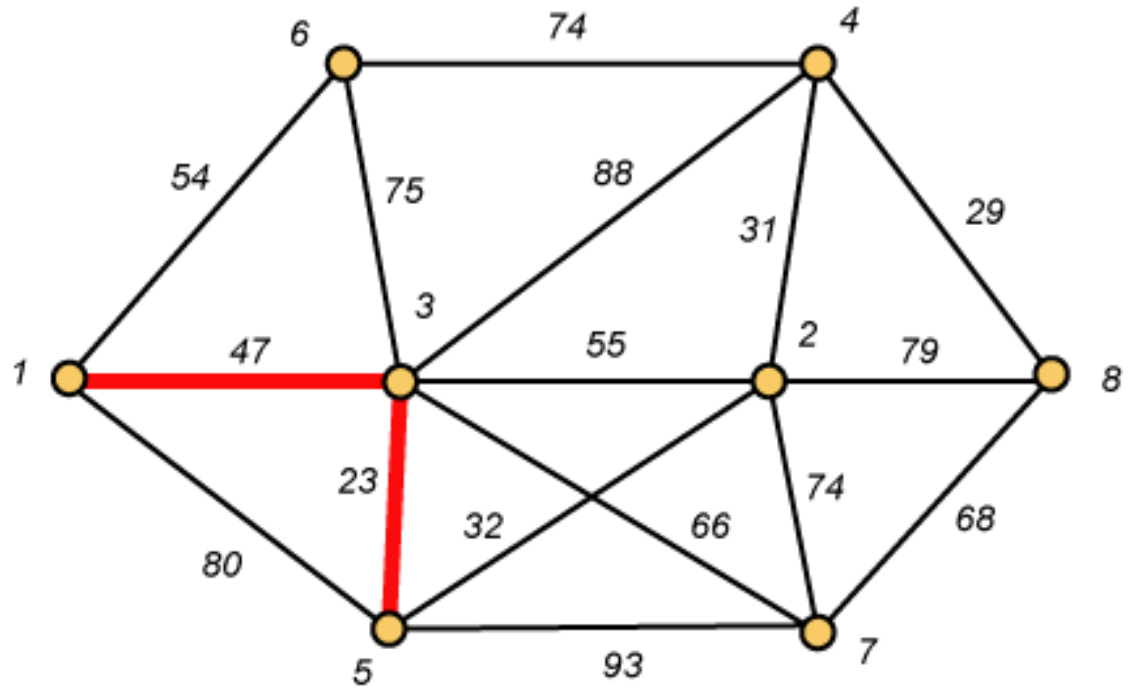
# Prim's Algorithm

- Vertex based algorithm
- Grows one tree T, **one vertex at a time**

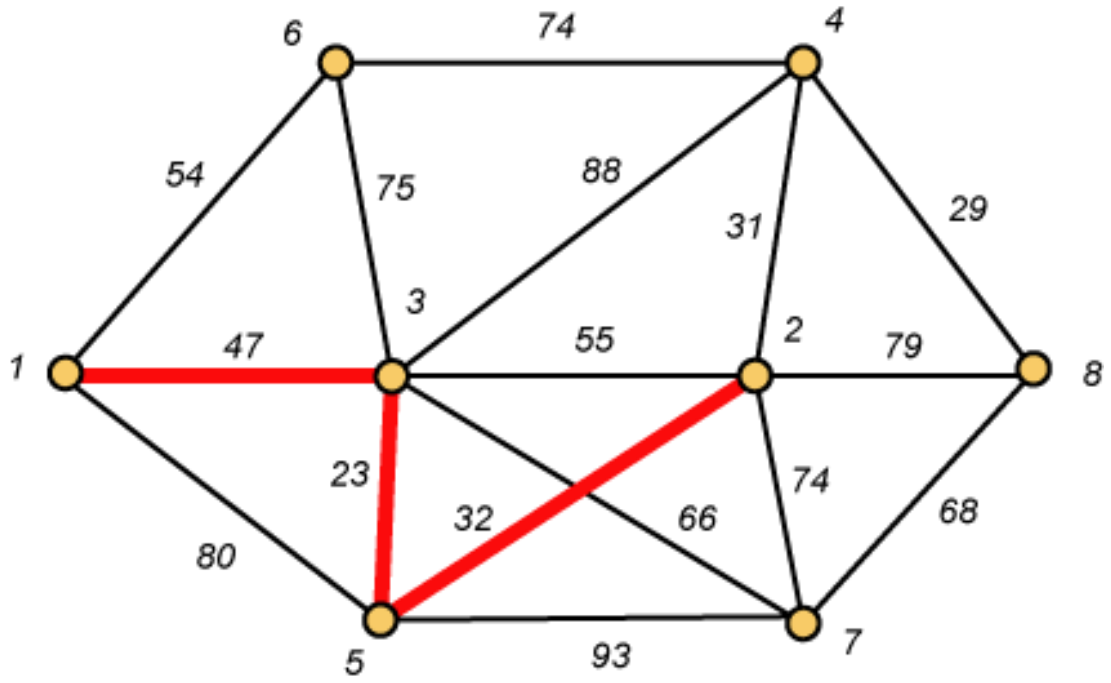
# Prim – Step 1



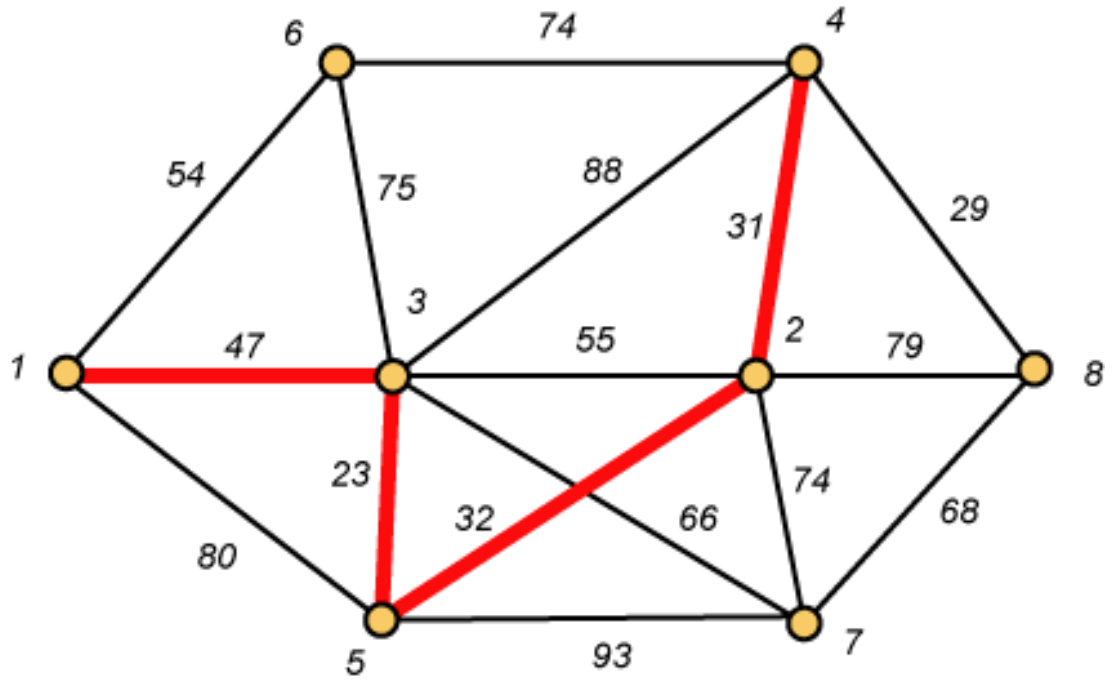
# Prim – Step 2



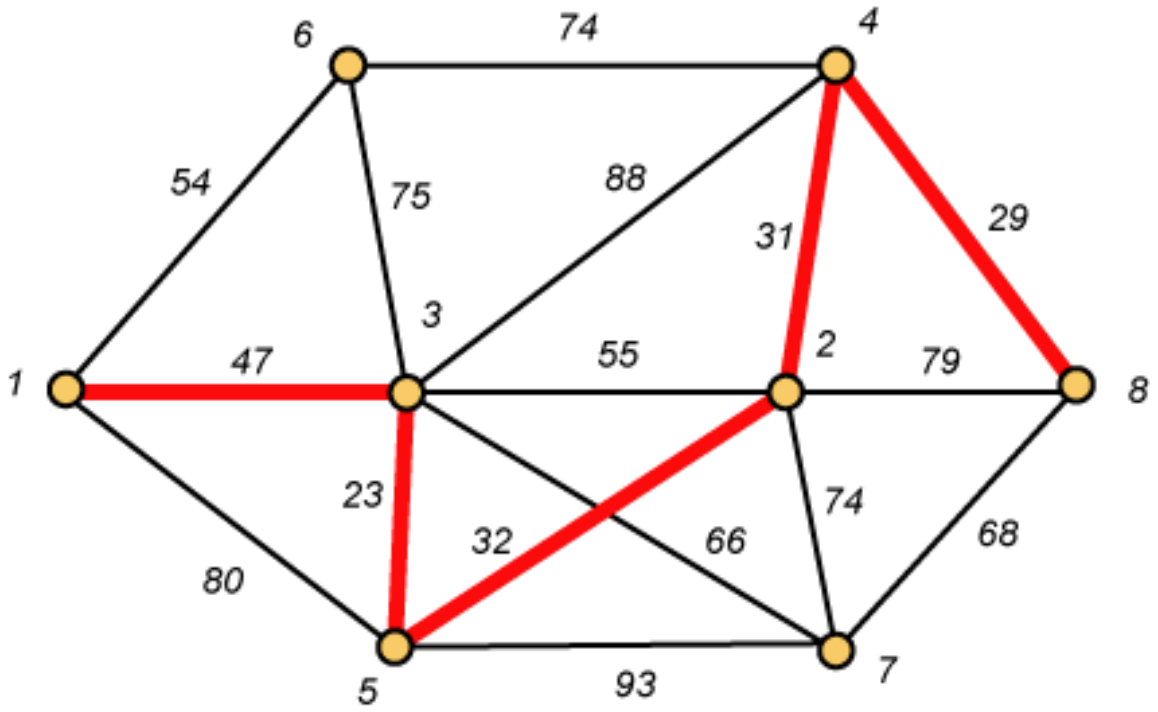
# Prim – Step 3



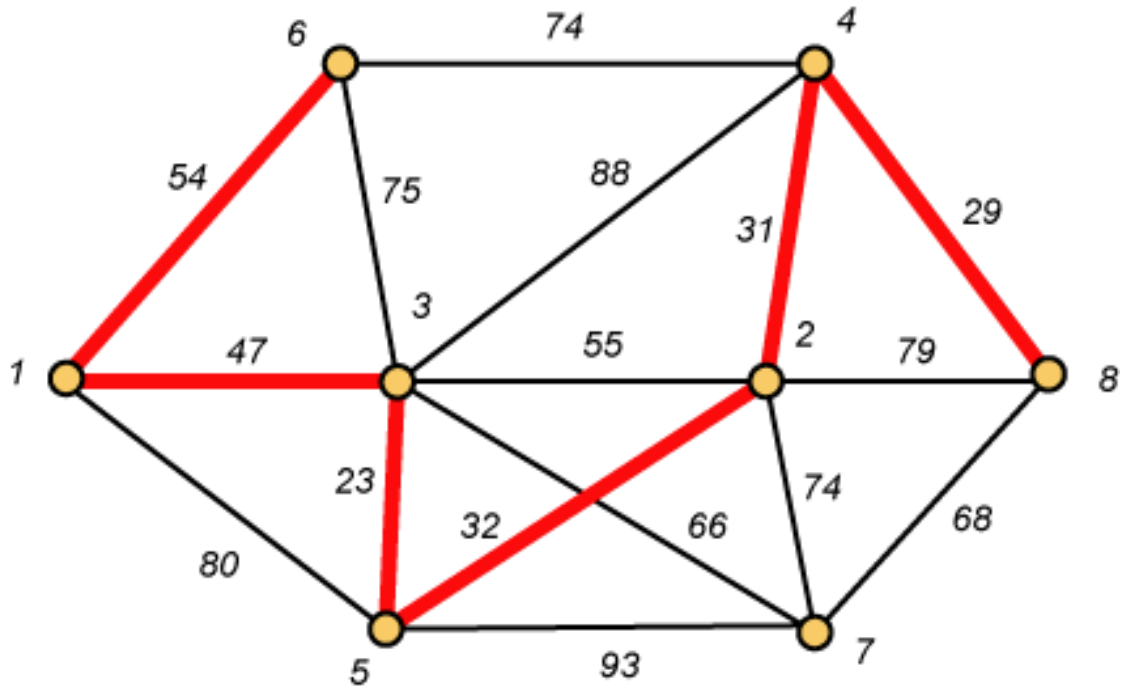
# Prim – Step 4



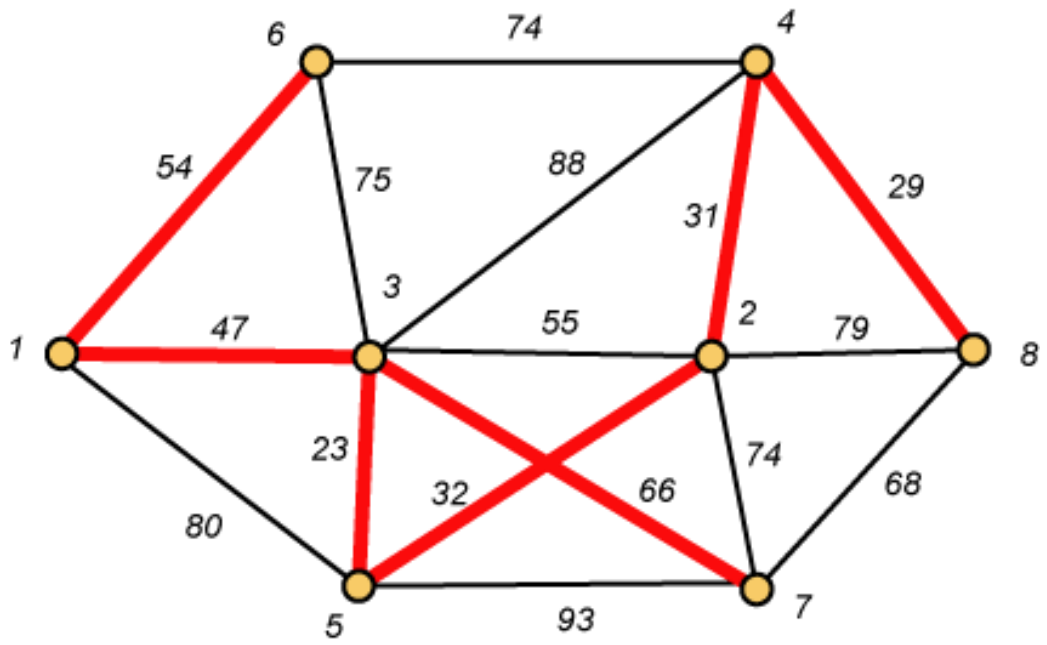
# Prim – Step 5



# Prim – Step 6



# Prim – Step 7 *Done!!*




Weight (T) = 23 + 29 + 31 + 32 + 47 + 54 + 66 = **282**

# Prim Algorithm:Variables

- r:
  - Grow the minimum spanning tree from the **root vertex "r"**.
- Q:
  - is a priority queue, holding all vertices that are not in the tree now.
- key[v]:
  - is the minimum weight of any edge connecting v to a vertex already in the tree.
- π [v]:
  - names the parent of v in the tree.
- T[v] –
  - Vertex v is already included in MST if T[v]==1, otherwise, it is not included yet.

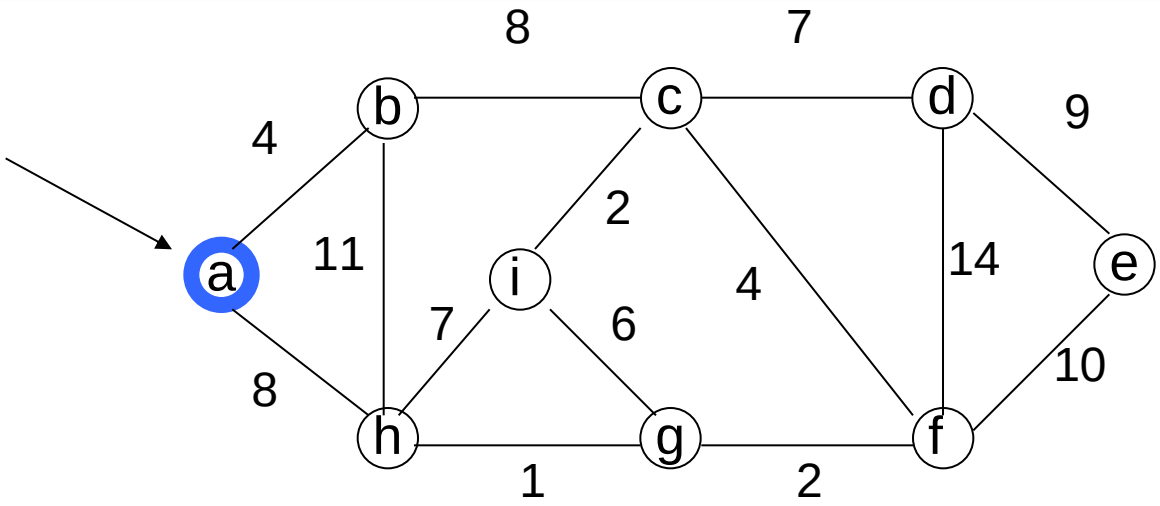
# Prim Algorithm (2)

```
MST-Prim(G,w,r)
for each u ∈ Q
    key[u] ← ∞
key[r] ← 0                          // r is the first tree
   node, let r=1
π[r] ← NIL
Q.insert((key[r], r))
while Q ≠ Ø do
     u ← Q.ExtractMin()
     make u part of T
     for each v ∈ Adj[u] do
          if v ∈ T and w(u,v) < key[v] then
               π[v] ← u
               key[v] ← w(u,v)

        Q.insert((key[v], v));
```

# The execution of Prim's algorithm(moderate part)

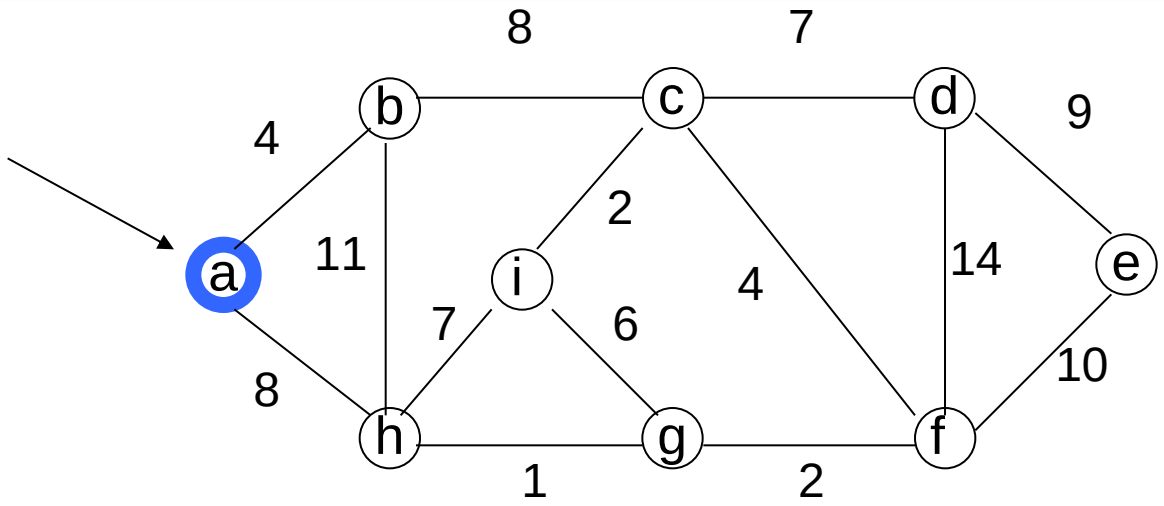
the root vertex




| V | a | b | c | d | e | f | g | h | i |
|---|---|---|---|---|---|---|---|---|---|
| T | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Key | 0 | - | - | - | - | - | - | - | - |
| π | -1 | - | - | - | - | - | - | - | - |

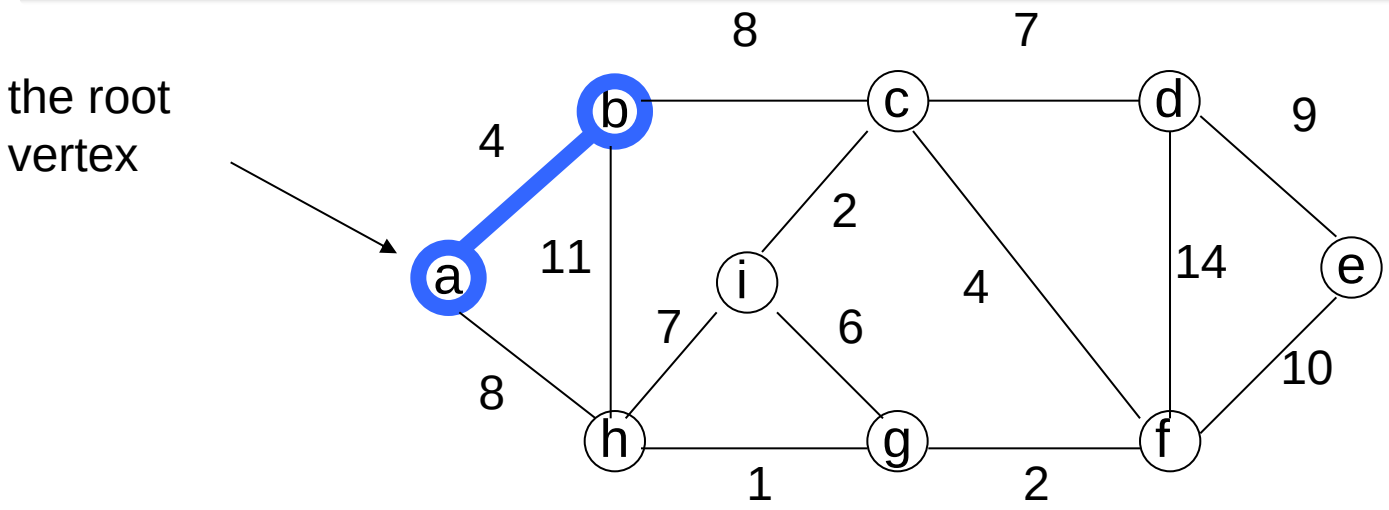
# The execution of Prim's algorithm(moderate part)

the root vertex




| V | a | b | c | d | e | f | g | h | i |
|---|---|---|---|---|---|---|---|---|---|
| T | 1 | **0** | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Key | 0 | **4** | - | - | - | - | - | 8 | - |
| π | -1 | **a** | - | - | - | - | - | a | - |

# The execution of Prim's algorithm(moderate part)

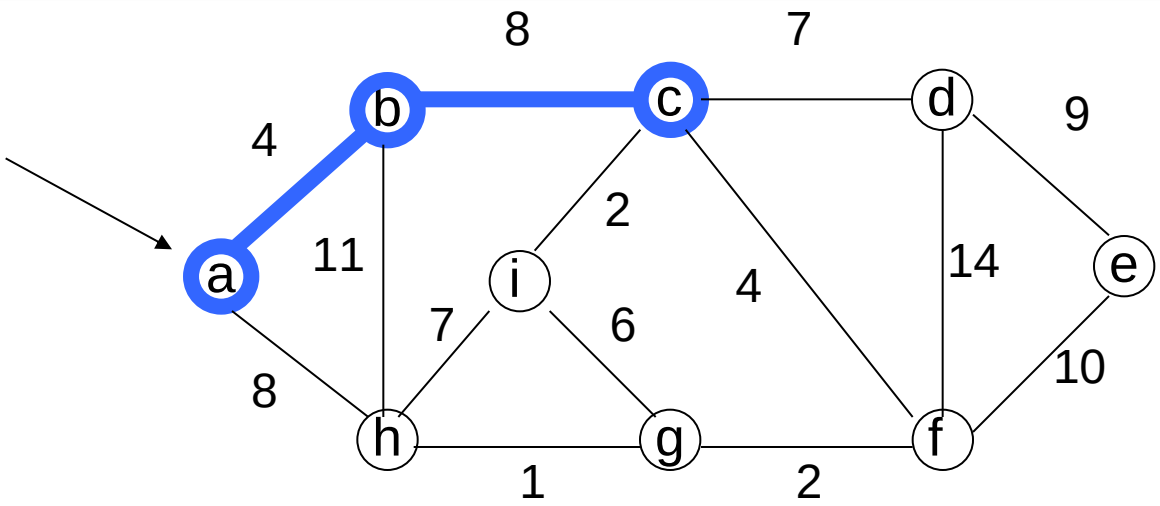



**Important:** Update Key[v] only if T[v]==0

| V | a | b | c | d | e | f | g | h | i |
|---|---|---|---|---|---|---|---|---|---|
| T | 1 | 1 | **0** | 0 | 0 | 0 | 0 | 0 | 0 |
| Key | 0 | 4 | **8** | - | - | - | - | 8 | - |
| π | -1 | a | **b** | - | - | - | - | a | - |

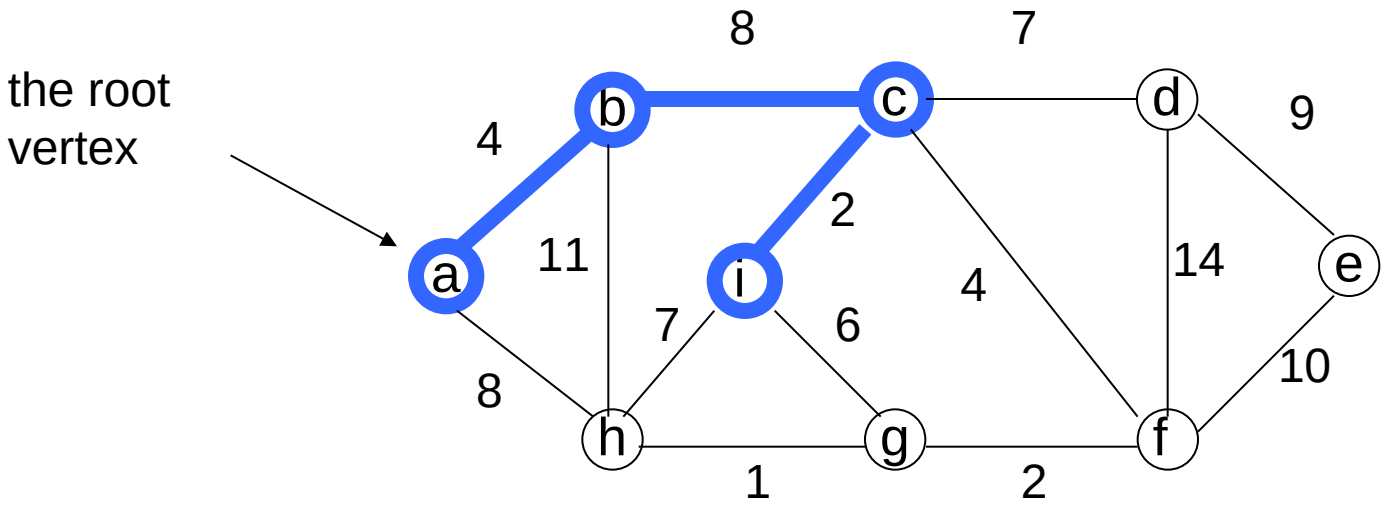
# The execution of Prim's algorithm(moderate part)

the root vertex




| V | a | b | c | d | e | f | g | h | i |
|---|---|---|---|---|---|---|---|---|---|
| T | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | **0** |
| Key | 0 | 4 | 8 | 7 | - | 4 | - | 8 | **2** |
| π | -1 | a | b | c | - | c | - | a | **c** |

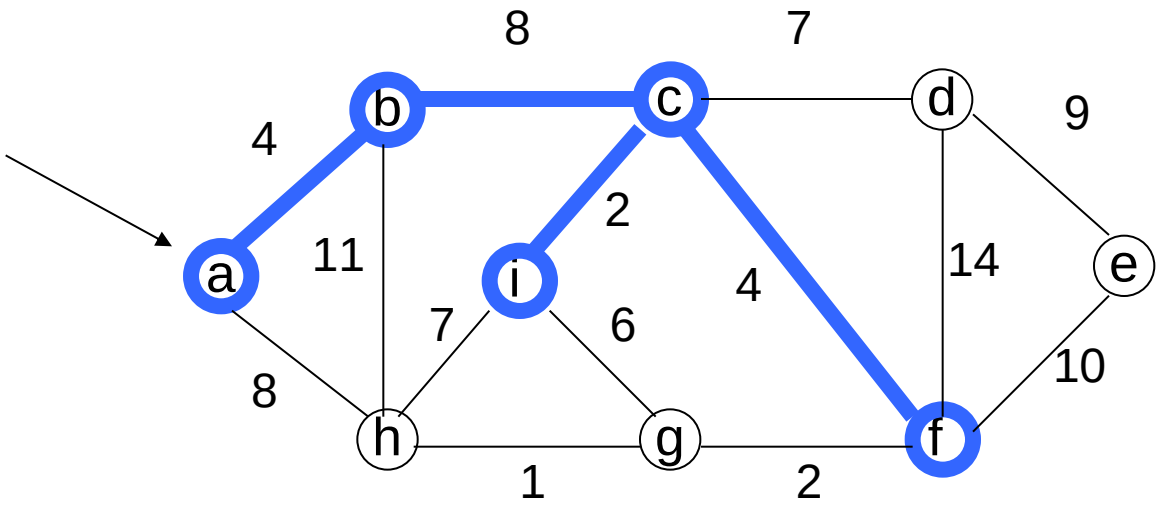
# The execution of Prim's algorithm(moderate part)




| V | a | b | c | d | e | f | g | h | i |
|---|---|---|---|---|---|---|---|---|---|
| T | 1 | 1 | 1 | 0 | 0 | **0** | 0 | 0 | 1 |
| Key | 0 | 4 | 8 | 7 | - | **4** | 6 | 7 | 2 |
| π | -1 | a | b | c | - | **c** | i | i | c |

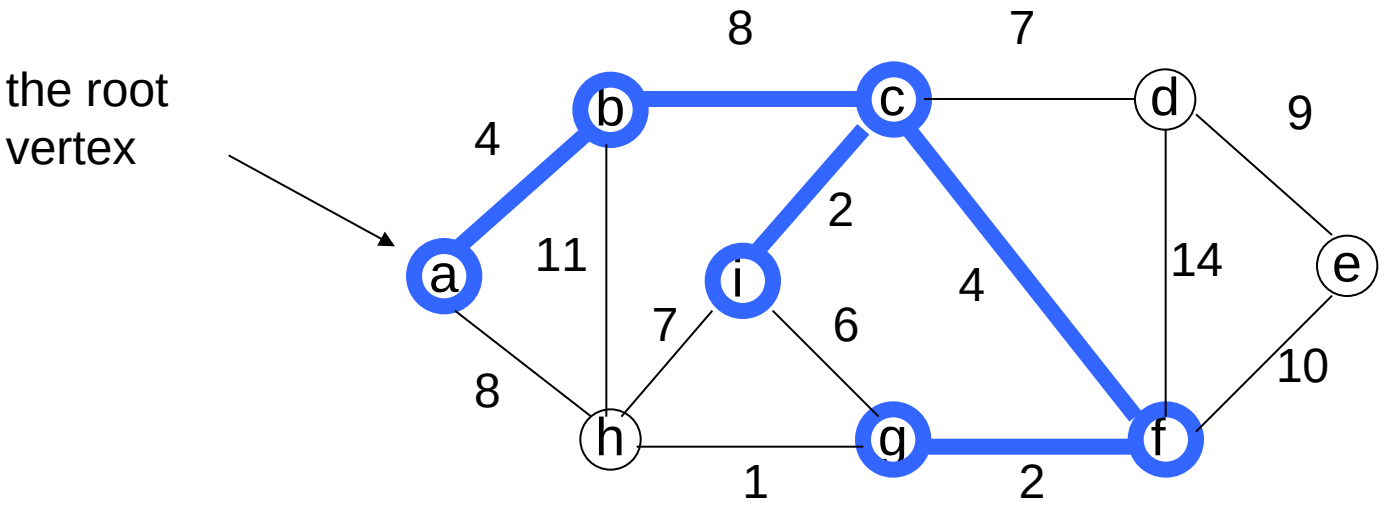
# The execution of Prim's algorithm(moderate part)

the root vertex




| V | a | b | c | d | e | f | g | h | i |
|---|---|---|---|---|---|---|---|---|---|
| T | 1 | 1 | 1 | 0 | 0 | **1** | **0** | 0 | 1 |
| Key | 0 | 4 | 8 | 7 | 10 | **4** | **2** | 7 | 2 |
| π | -1 | a | b | c | f | **c** | **f** | i | c |

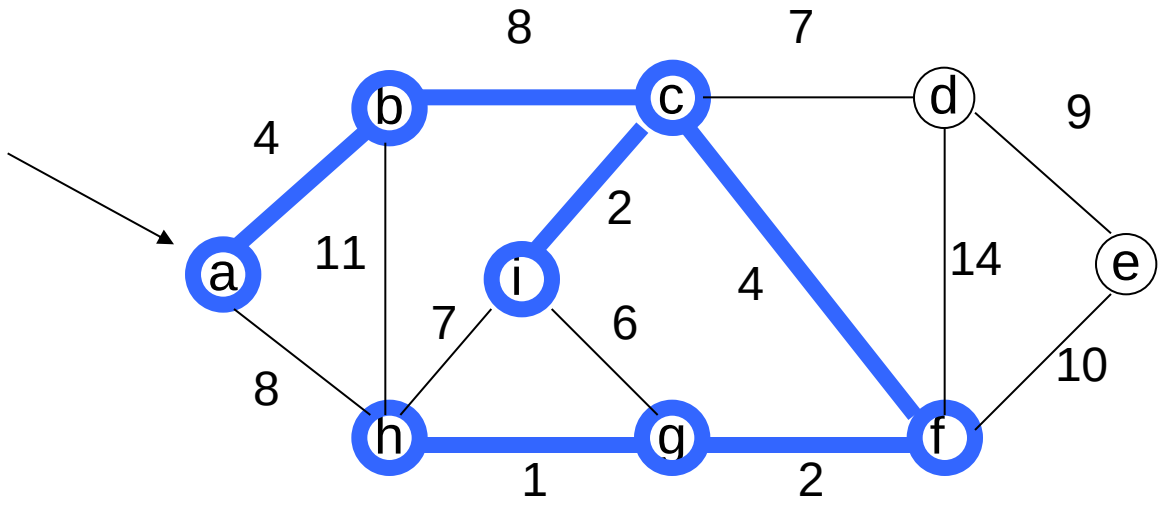
# The execution of Prim's algorithm(moderate part)




| V | a | b | c | d | e | f | g | h | i |
|---|---|---|---|---|---|---|---|---|---|
| T | 1 | 1 | 1 | 0 | 0 | 1 | **1** | **0** | 1 |
| Key | 0 | 4 | 8 | 7 | 10 | 4 | **2** | **1** | 2 |
| π | -1 | a | b | c | f | c | **f** | **g** | c |

# The execution of Prim's algorithm(moderate part)

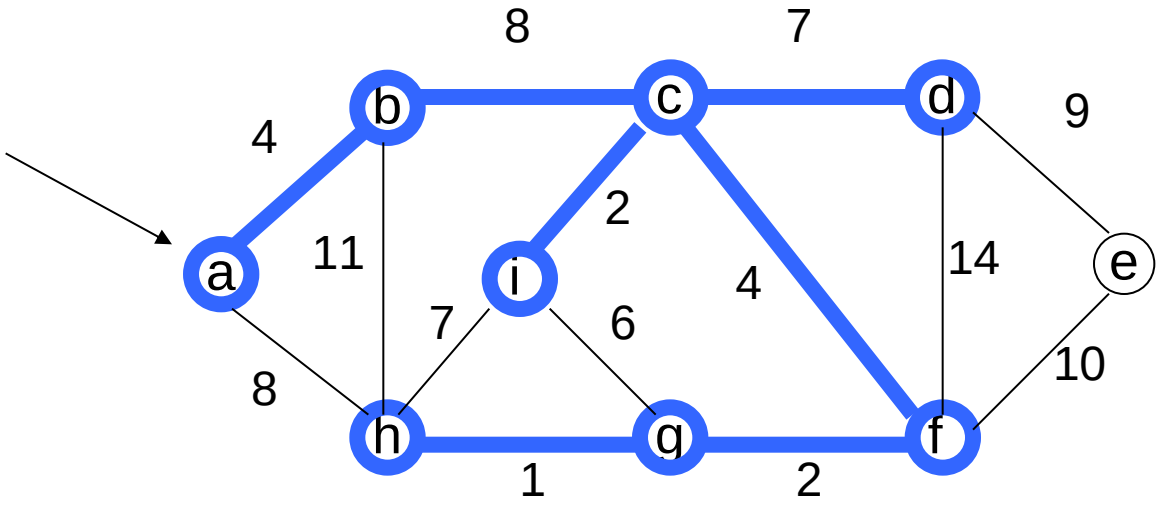
the root vertex




| V | a | b | c | d | e | f | g | h | i |
|---|---|---|---|---|---|---|---|---|---|
| T | 1 | 1 | 1 | **0** | 0 | 1 | 1 | **1** | 1 |
| Key | 0 | 4 | 8 | **7** | 10 | 4 | 2 | **1** | 2 |
| π | -1 | a | b | **c** | f | c | f | **g** | c |

# The execution of Prim's algorithm(moderate part)

the root vertex



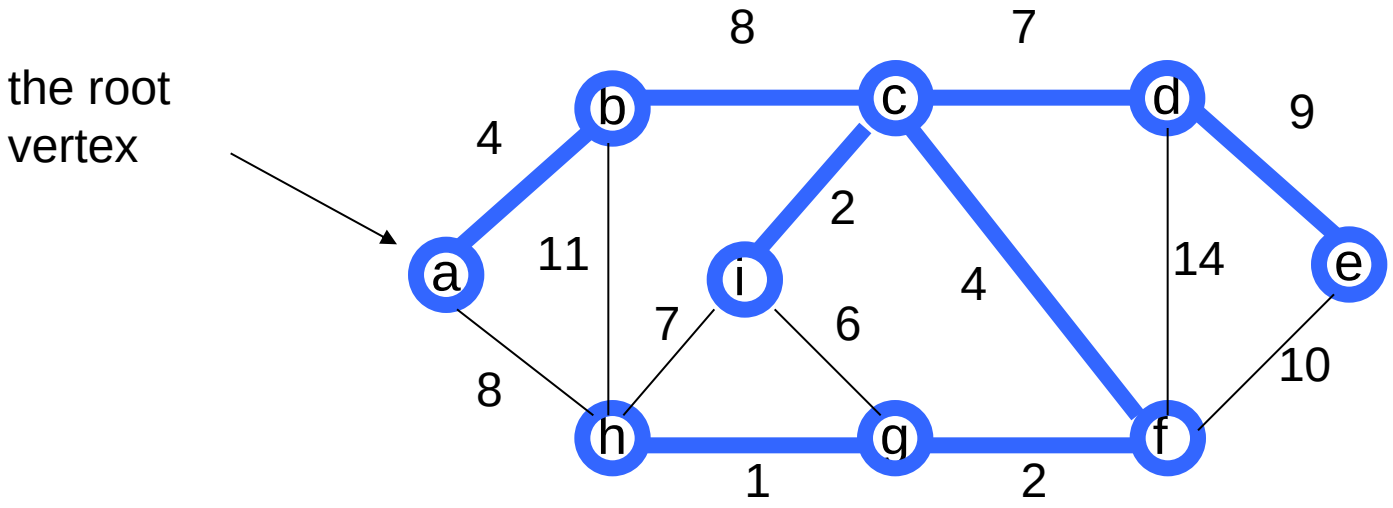

| V | a | b | c | d | e | f | g | h | i |
|---|---|---|---|---|---|---|---|---|---|
| T | 1 | 1 | 1 | **1** | **0** | 1 | 1 | 1 | 1 |
| Key | 0 | 4 | 8 | **7** | **9** | 4 | 2 | 1 | 2 |
| π | -1 | a | b | **c** | **d** | c | f | g | c |

# Prim Algorithm (2)

```
MST-Prim(G,w,r)
for each u ∈ Q                                          O(V)
   key[u] ← ∞
key[r] ← 0                          // r is the first tree
   node, let r=1
π[r] ← NIL                                              Heap: O(lgV)
Q.insert((key[r], r))                                   O(V)
while Q ≠ Ø do                                          Heap: O(lgV)
    u ← Q.ExtractMin()
    make u part of T                                    Overall: O(E)
    for each v ∈ Adj[u] do
        if v ∈ T and w(u,v) < key[v] then
           π[v] ← u
           key[v] ← w(u,v)                              Heap: O(lgV)
```

Overall complexity: O(***V***)+O(***V*** lg ***V***+***E*** lg ***V***) = O(***E*** lg ***V***)

# The execution of Prim's algorithm(moderate part)




| V | a | b | c | d | e | f | g | h | i |
|---|---|---|---|---|---|---|---|---|---|
| T | 1 | 1 | 1 | 1 | **1** | 1 | 1 | 1 | 1 |
| Key | 0 | 4 | 8 | 7 | **9** | 4 | 2 | 1 | 2 |
| π | -1 | a | b | c | **d** | c | f | g | c |

# Overall Complexity Analysis - Prims

- $O(n^2)$
  - When we don't use heap
  - To find the minimum element, we traverse the "KEY" array from beginning to end
  - We use adjacency matrix to update KEY.
- O(ElogV)
  - When min-heap is used to find the minimum element from "KEY".
- O(E+VlogV)
  - When fibonacci heap is used to find the minimum element from "KEY".

# Question

- Exercise
  - 23.2.7 – Add new vertices & edges in a graph
  - 23-1: Second best minimum spanning tree