Minimum Spanning Tree

Dr Dayal Kumar Behera

- **Acyclic Graph:** *A graph that contains no cycles is called* ***acyclic graph****.*

- **Tree:** *A simple(no loop), connected, acyclic graph is called a* ***tree****.*

- ***Spanning tree*** *of a graph G (V, E) is a tree that contains all the vertices of the graph G.*

What is a Spanning Tree?

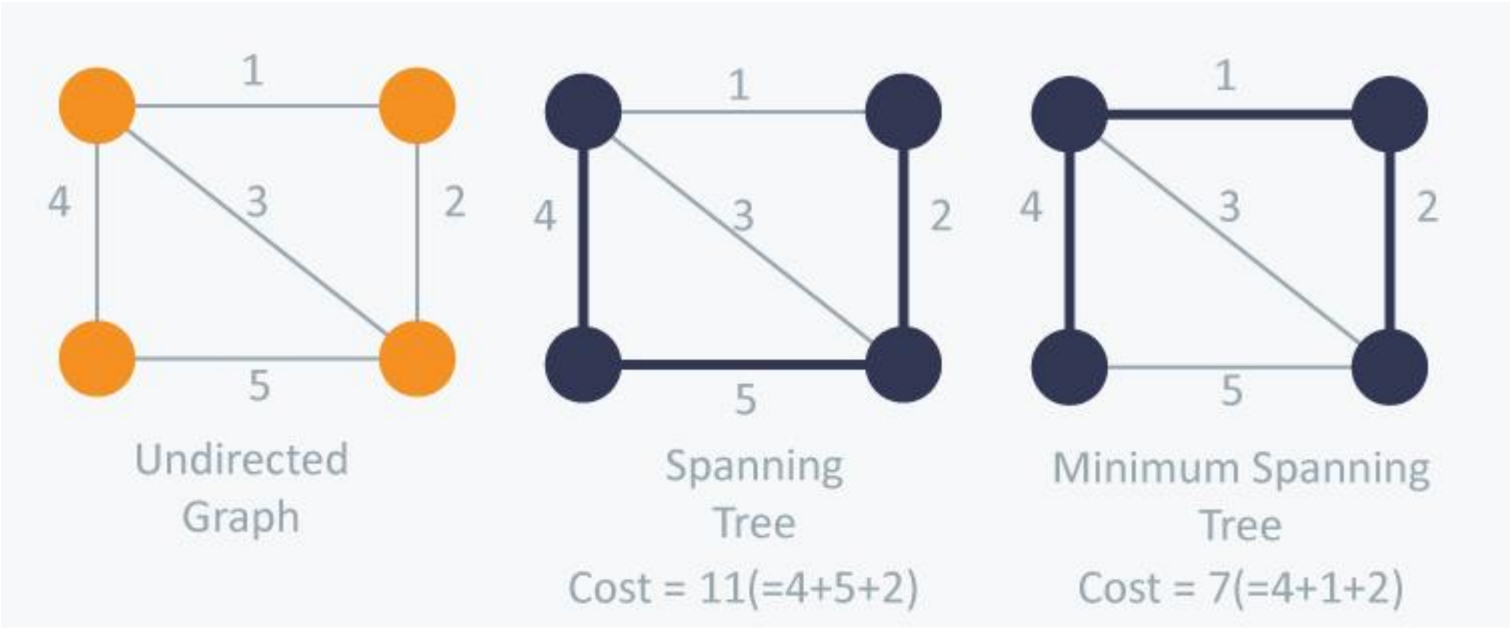
Given a connected graph G(V, E), if T is a subgraph of G and contains all the vertices but no cycles, then T is said to be a spanning tree.

Definition

Given a connected graph $G = (V, E)$,

- Spanning tree T:
  - A tree that includes all nodes from V
  - $T = (V, E')$, where $E' \subseteq E$
  - Weight of T: *sum of weights of all the edges in T.*
- Minimum spanning tree (MST):
  - A tree with minimum weight among all spanning trees.
  - MST is not unique (There can also be many minimum spanning trees)

Example



1
4
3
2
5
Undirected Graph
1
4
3
2
5
Spanning Tree
Cost = 11(=4+5+2)
1
4
3
2
5
Minimum Spanning Tree
Cost = 7(=4+1+2)

Minimum spanning trees: properties

Important properties:

- An MST is always a tree and it cannot contain a cycle
- If there are $|V|$ vertices, the MST contains exactly $|V|$ - 1 edges.
- If we add or remove an edge from an MST, it's no longer a valid MST for that graph.
- Adding an edge introduces a cycle; removing an edge means vertices are no longer connected.
- If every edge has a unique weight, there exists a unique MST.

Algorithms to find Minimum Spanning Trees

- Kruskal’s algorithm
- Prim’s algorithm

Generic Algorithm

- Framework for $G = (V, E)$:
  - Goal: build a set of edges $A \subseteq E$
  - Start with A empty
  - Add edge into A one by one
  - At any moment, A is a subset of some MST for G

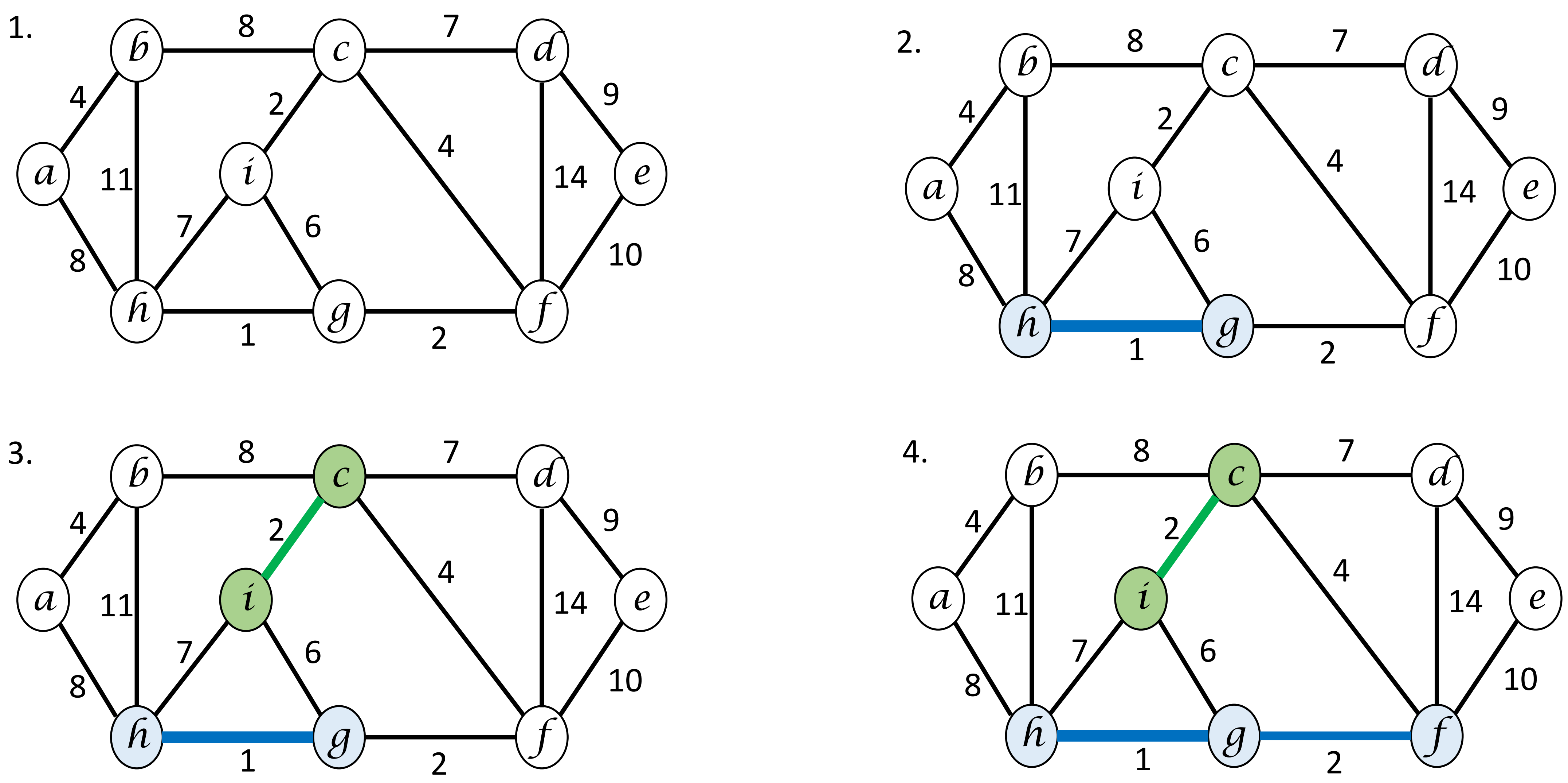
```
GENERIC-MST(G, w)
A ← Ø
while A is not a spanning tree
    do find an edge (u, v) that is safe for A
        A ← A ∪ {(u, v)}
return A
```

An edge is safe if adding it to A, *does not form a cycle*

Kruskal's Algorithm

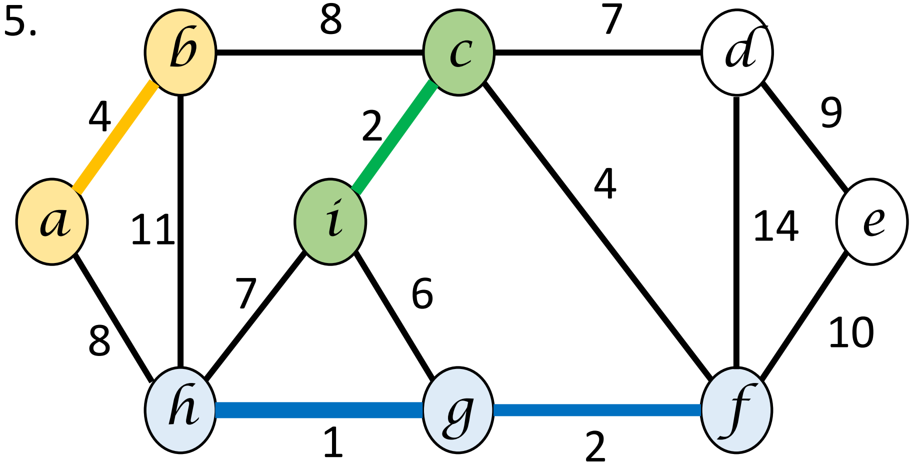
- Start with A empty, and each vertex being its own connected component
- Repeatedly merge two components by connecting them with a light edge crossing them
  - Maintain sets of components — Disjoint set data structure
  - Choose light edges — Scan edges from low to high weight

Example Illustration

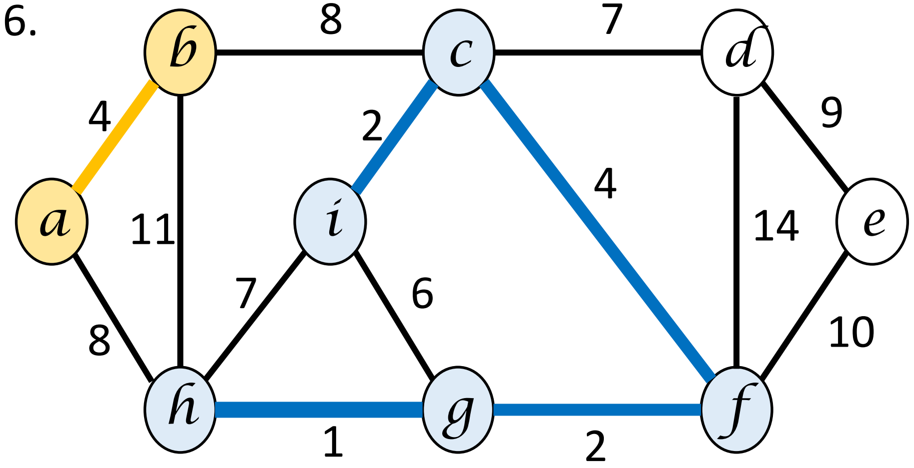


Example Illustration

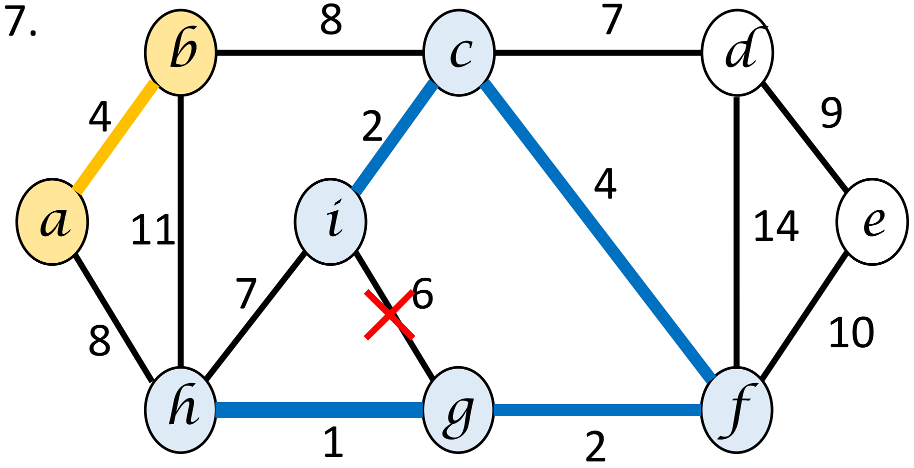
5.



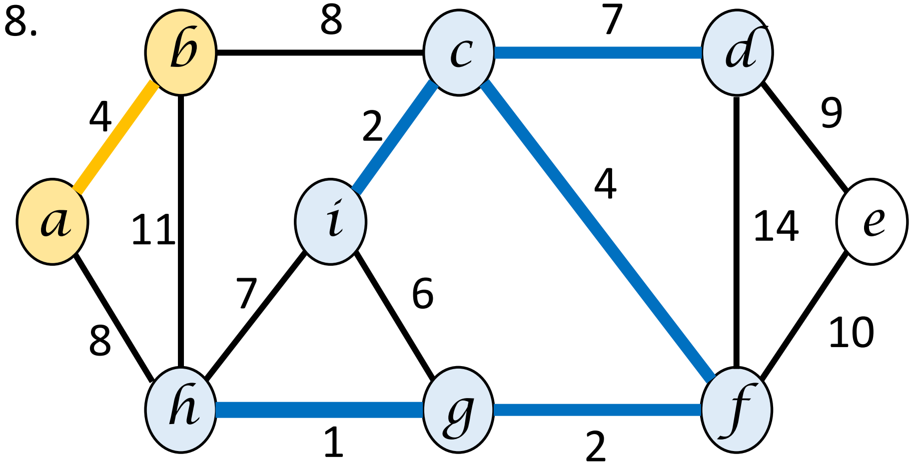
6.



7.

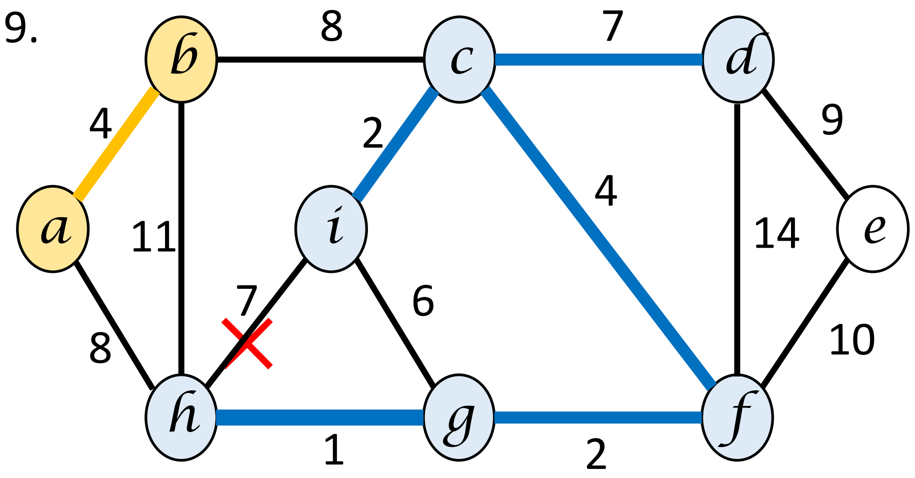


8.

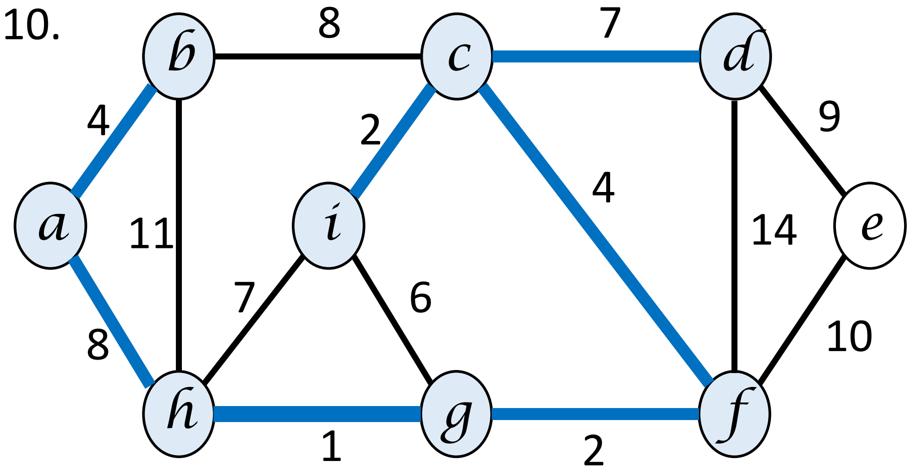


Example Illustration

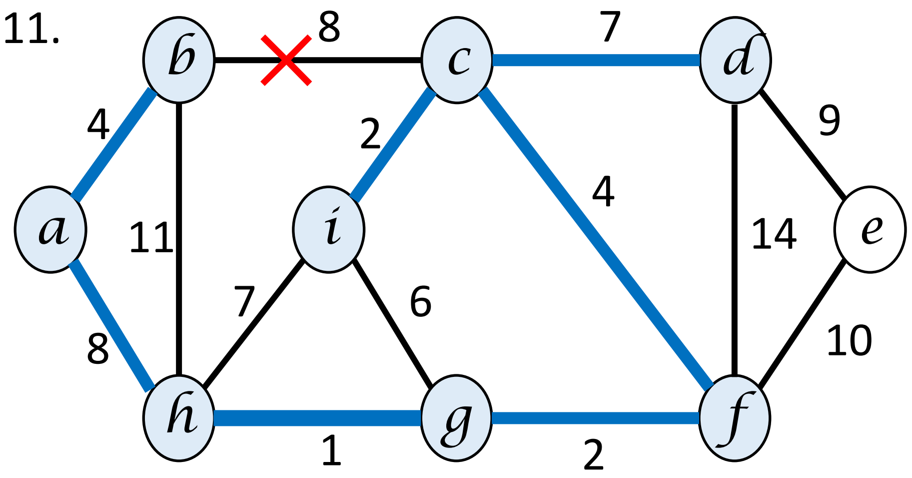
9.



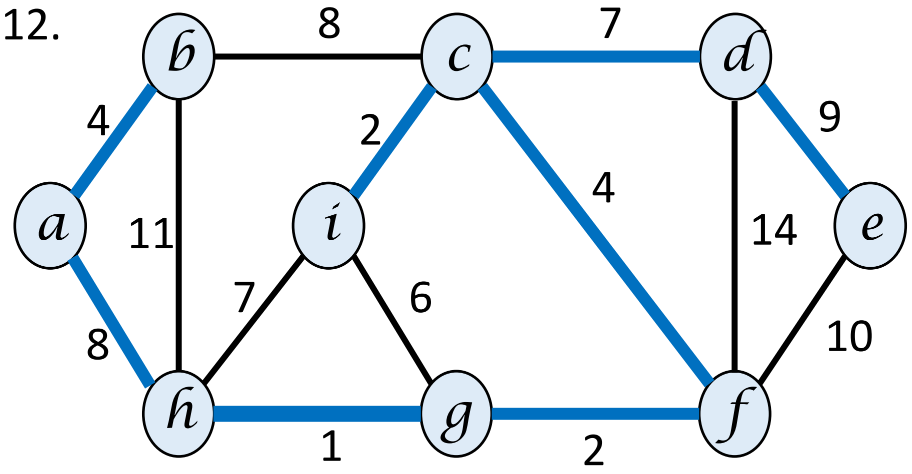
10.



11.

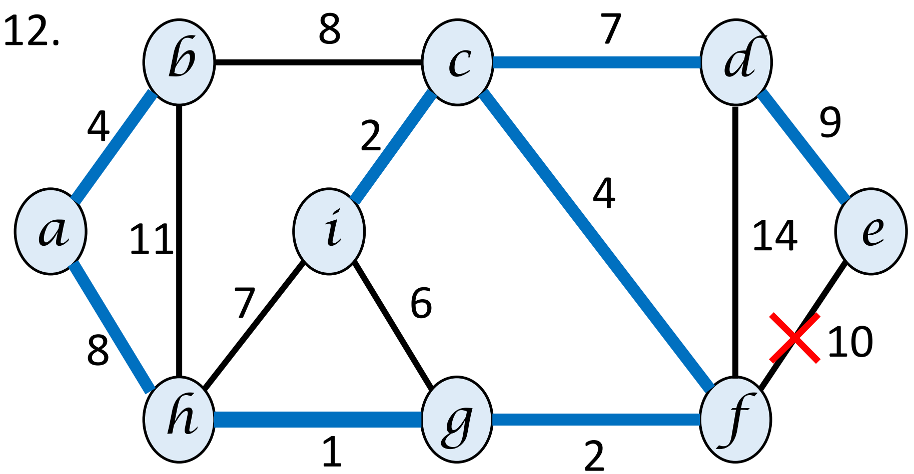


12.

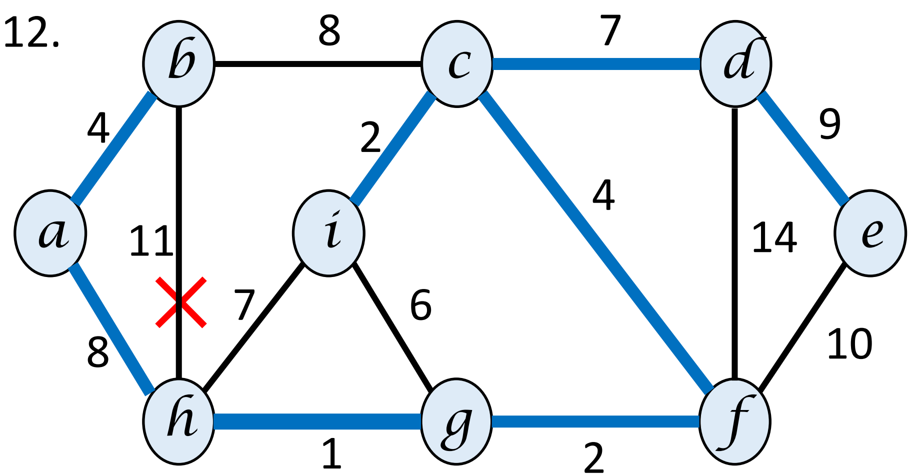


Example Illustration

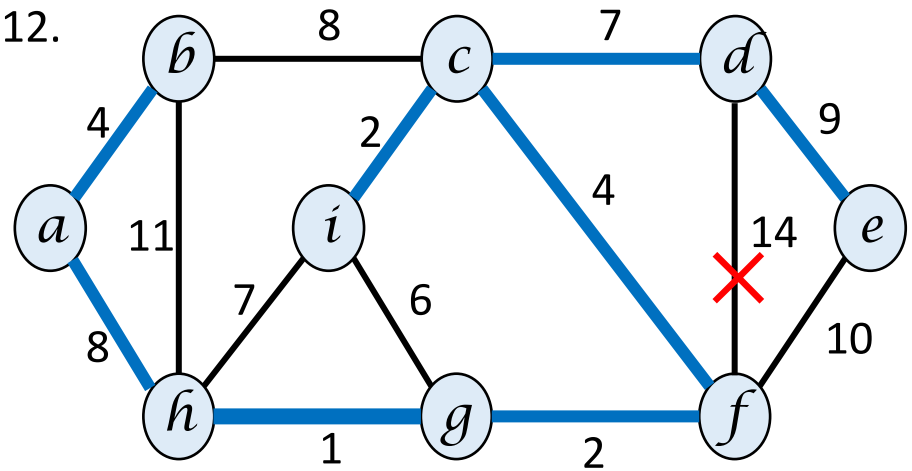
12.



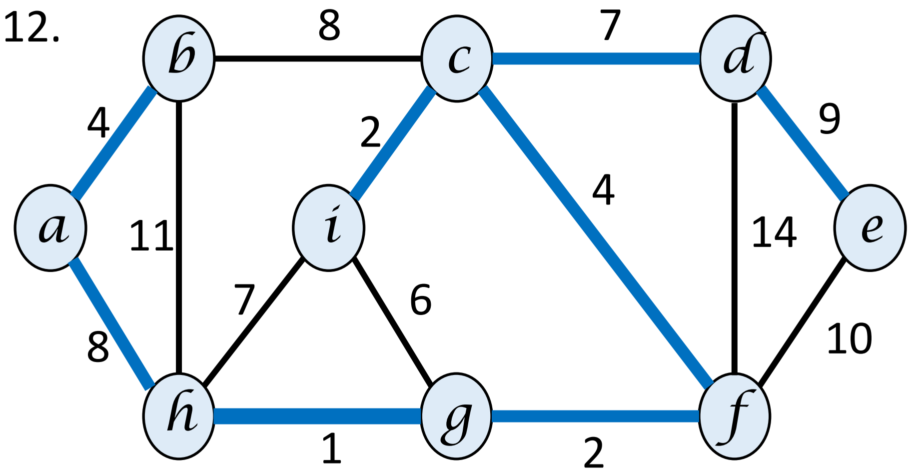
12.



12.



12.



Total Weight = 37

KRUSKAL: Pseudo-code

```
KRUSKAL(V, E, w)
A ← Ø
for each vertex v ∈ V
    do MAKE-SET(v)
sort E into nondecreasing order by weight w
for each (u, v) taken from the sorted list
    do if FIND-SET(u) ≠ FIND-SET(v)
        then A ← A ∪ {(u, v)}
             UNION(u, v)
return A
```

Analysis

- The sorting of edges will take $O(|E|\log|E|)$ time. Next for each edge, disjoint-set functions are called. This requires $O(|E|\log|V|)$ time.

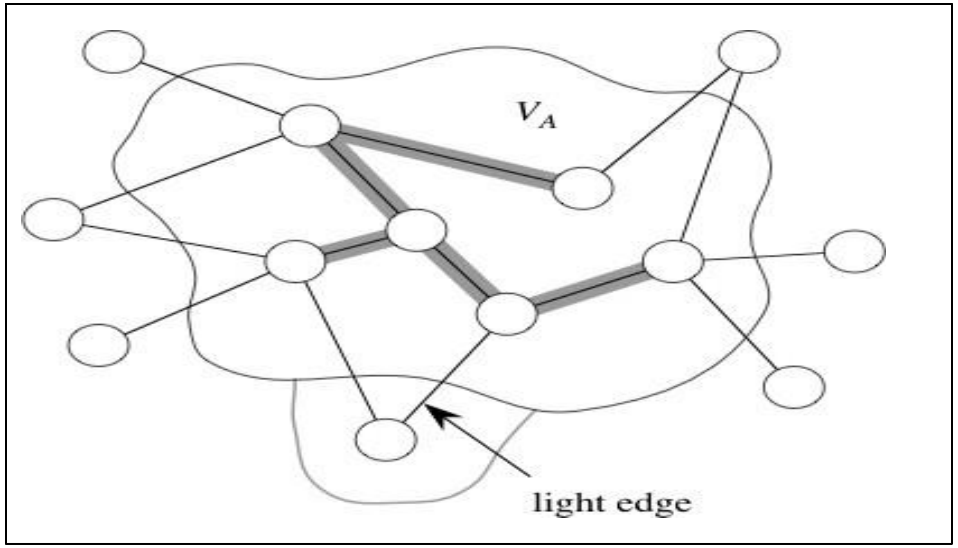
- Since $|E|$ is at most $|V|^2$ and $\log|V|^2 = 2\log|V|$,

  $$O(|E|\log|E|) = O(|E|\log|V|).$$

- The running time for Kruskal's algorithm is thus $O(|E|\log|V|)$.

Prim's Algorithm

- Start with an **arbitrary node** from *V*
- Instead of maintaining a forest, grow a MST
  - At any time, maintain a MST for $S \subseteq V$
- At any moment, find a light edge connecting *S* with *(V-S) i*.e., the edge with smallest weight connecting some vertex in *S* with some vertex in *V-S*



Prim's Algorithm

For a given graph G(V, E) with 'n' no of vertices.

1. Select an arbitrary vertex 'u' and put it in S.
    - S ← { u }
    - $\bar{E}$ ← {}
2. Select an edge (u, v) with minimum weight such that $u \in S$ and $v \in V - S$
3. Modify $\bar{E} = \bar{E} \cup \{(u, v)\}$ and S = S U { v }
4. if $|S| = n$ $i.e. V = S$ then Stop

    else goto step 2

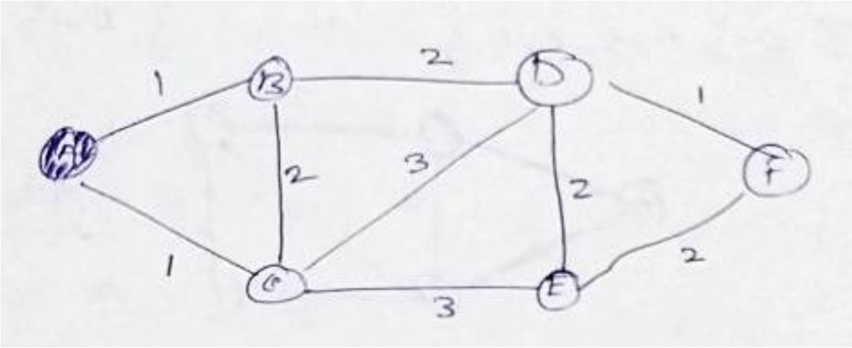
Output: T (V, $\bar{E}$) - minimum spanning tree

Prim's Algorithm cont.

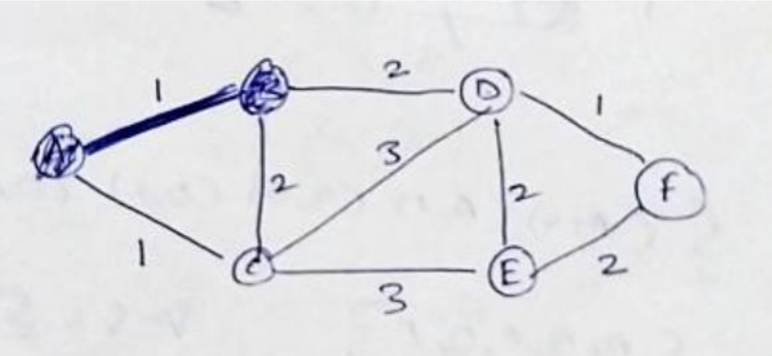
- Maintain the tree already build at any moment
  - Easy: simply a tree rooted at r : the starting node
- Find the next light edge efficiently
  - For $v \in V - S$, define $key(v)$ = the min distance between v and some node from S
  - At any moment, find the node with min key.

Use a Min priority queue (Q)

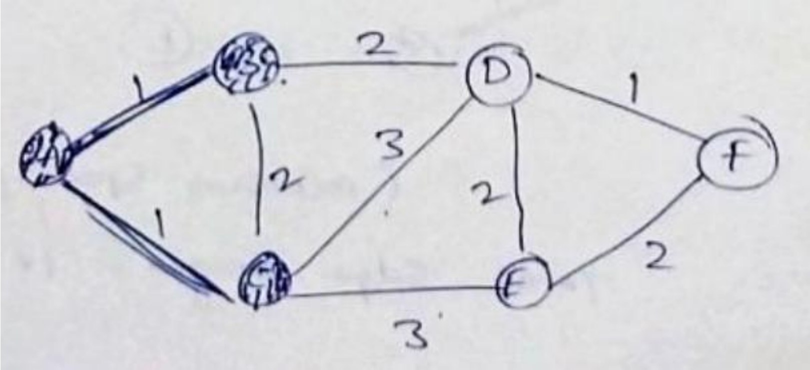
Example 1



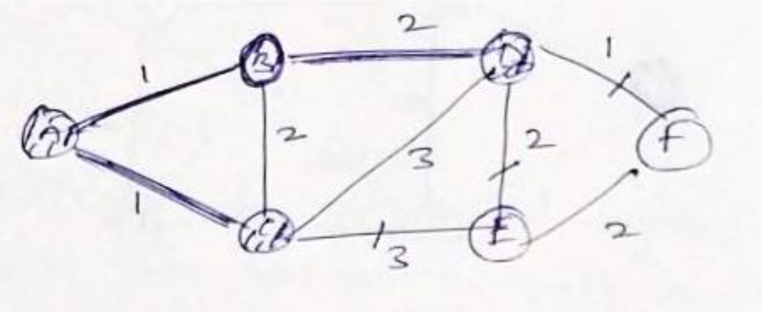
S = {A} V – S = {B, C, D, E, F}
$\bar{E}$ = {}



S = {A, B} V – S = {C, D, E, F}
$\bar{E}$ = {(A,B)}

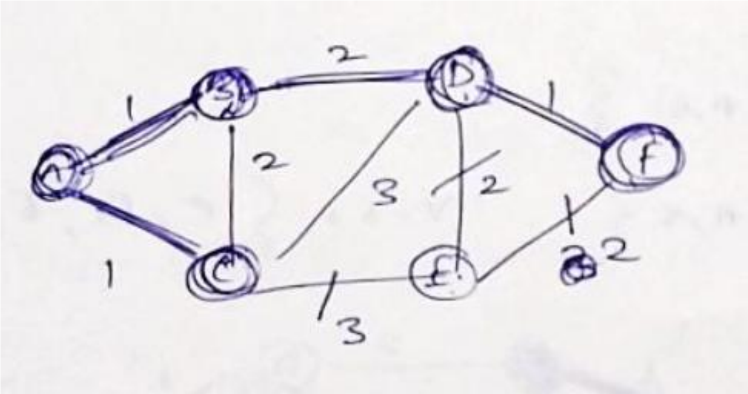


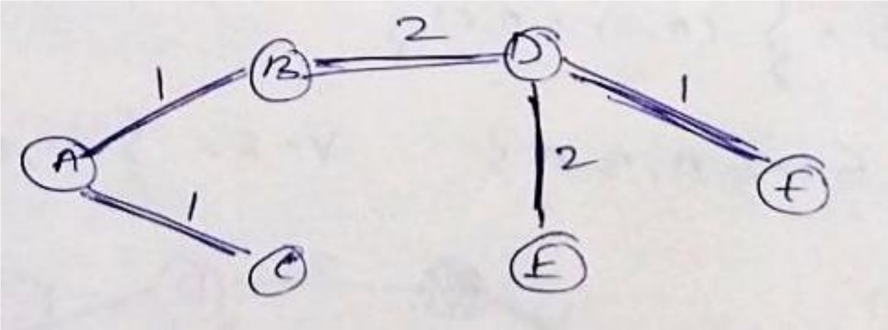
S = {A, B, C} V – S = {D, E, F}
$\bar{E}$ = {(A,B), (A,C)}



S = {A, B, C, D} V – S = {E, F}
$\bar{E}$ = {(A,B), (A,C), (B,D)}

Example 1





S = {A, B, C, D, F} V – S = {E}

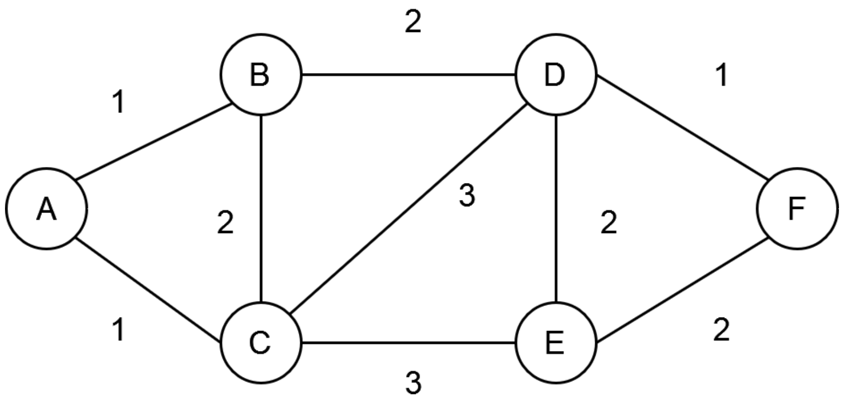
$\bar{E}$ = {(A,B), (A,C), (B,D), (D,F)}

S = {A, B, C, D, E, F} V – S = {}

$\bar{E}$ = {(A,B), (A,C), (B,D), (D,F), (D,E)}

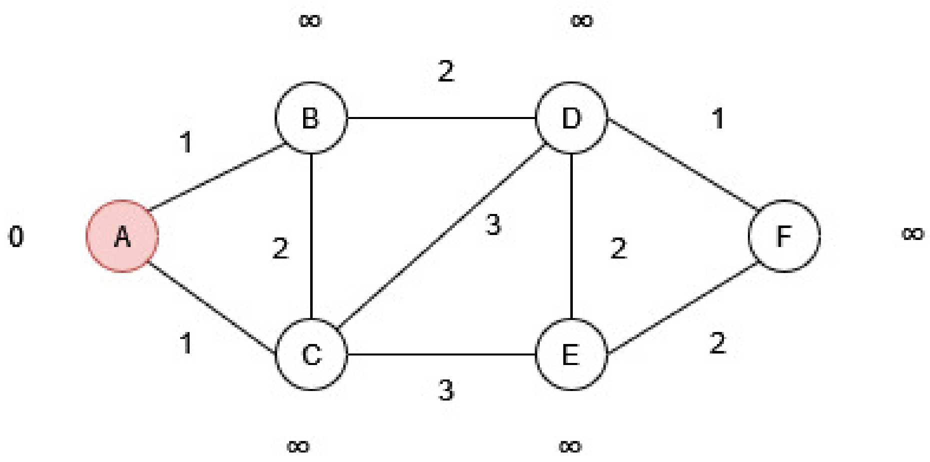
Total edge weights (Cost) = 7

Example 1: with priority queue (Q)



Q

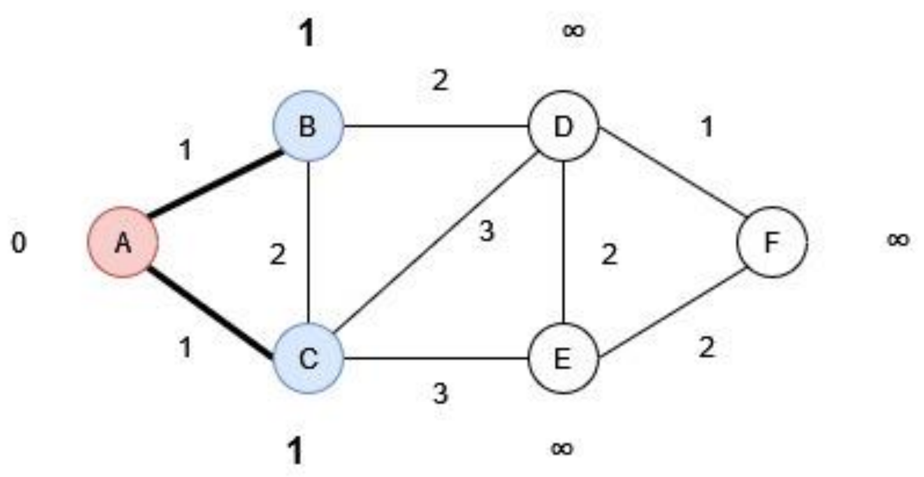
A	B	C	D	E	F
-	-	-	-	-	-

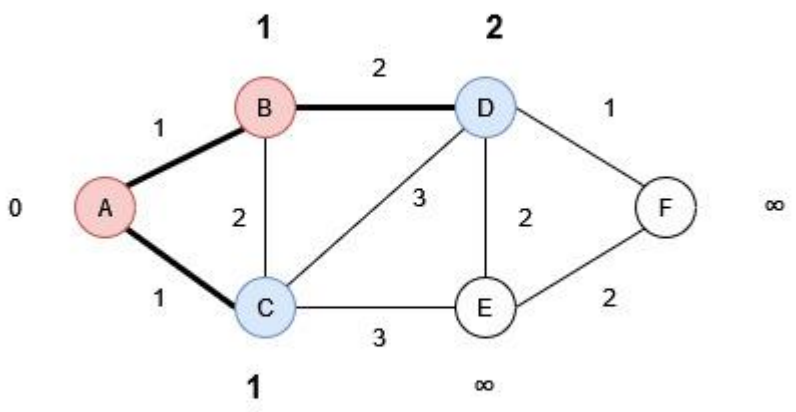


Q

A	B	C	D	E	F
0	∞	∞	∞	∞	∞

Example 1: with priority queue (Q)





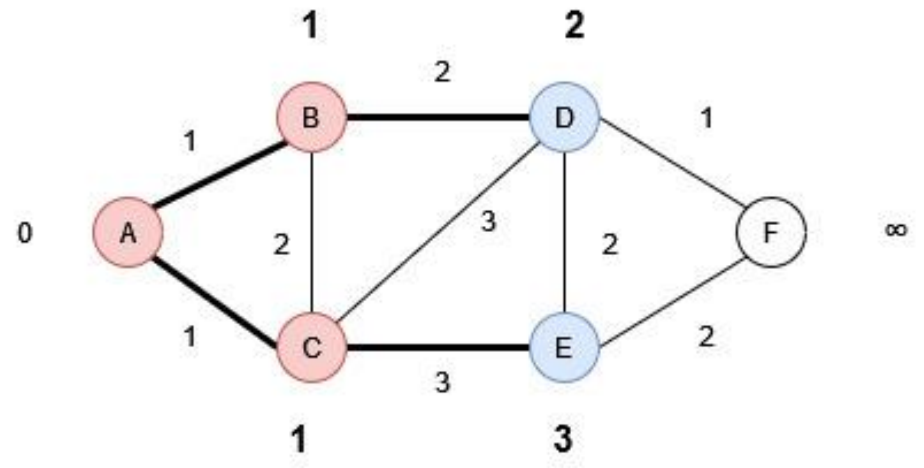
Q

B	C	D	E	F
1	1	∞	∞	∞

Q

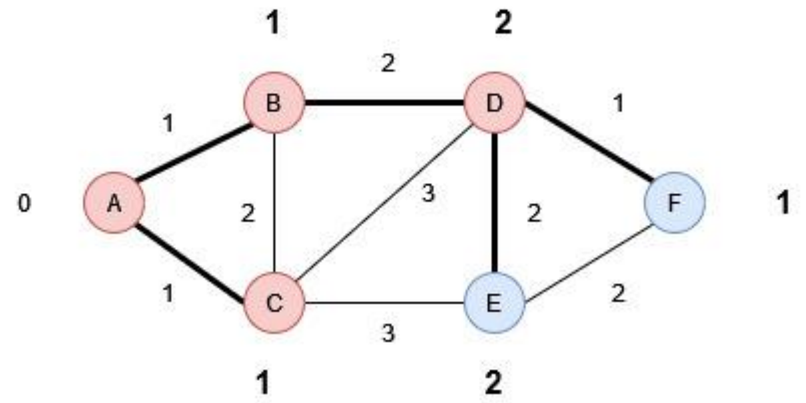
C	D	E	F
1	2	∞	∞

Example 1: with priority queue (Q)



Q

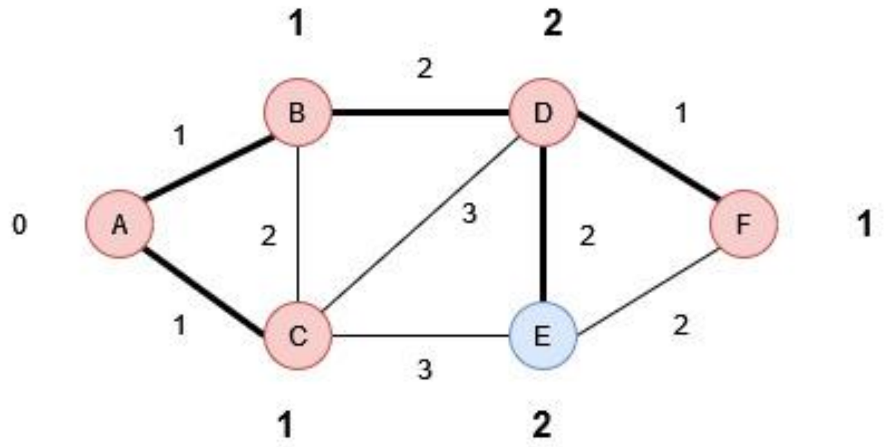
D	E	F
2	3	∞

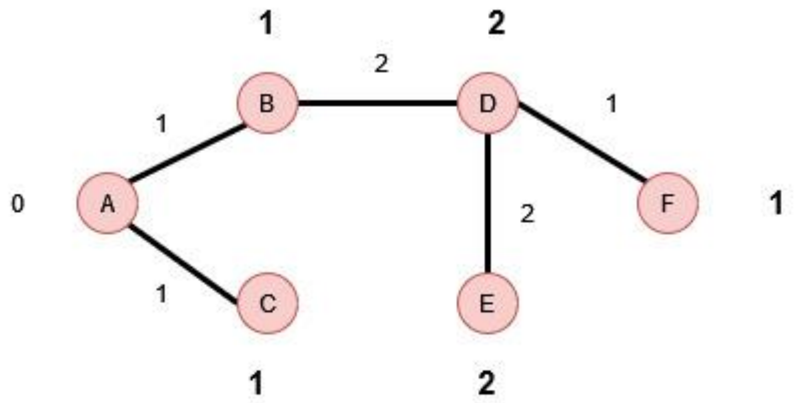


Q

E	F
2	1

Example 1: with priority queue (Q)



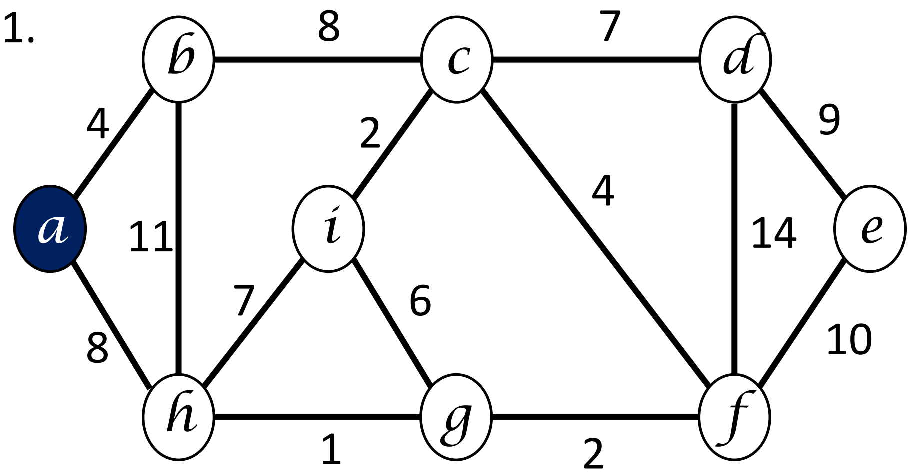


Q	E
	2

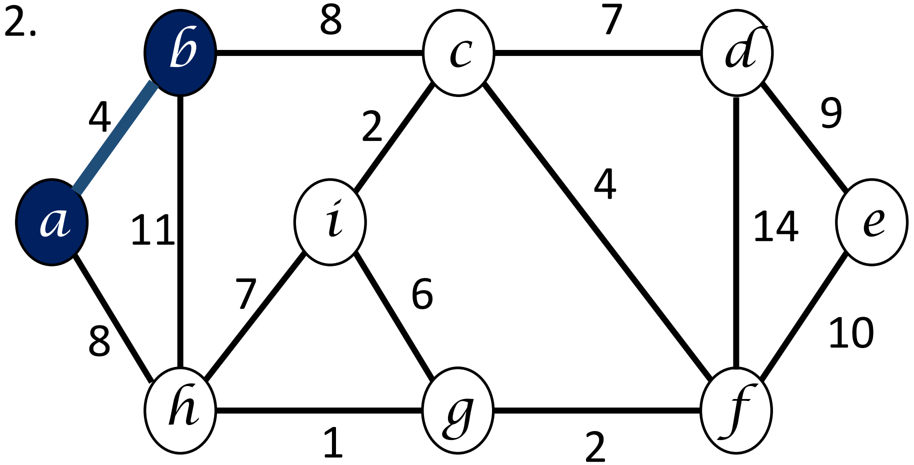
Q = {}

Example 2

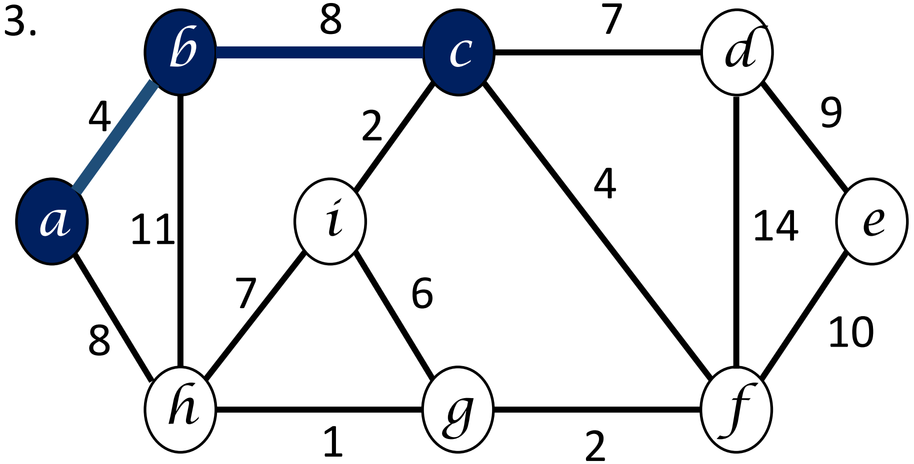
1.



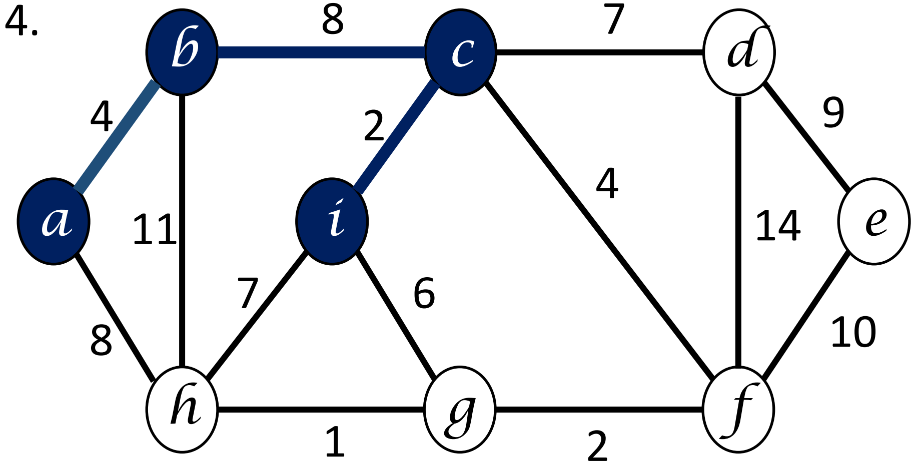
2.



3.

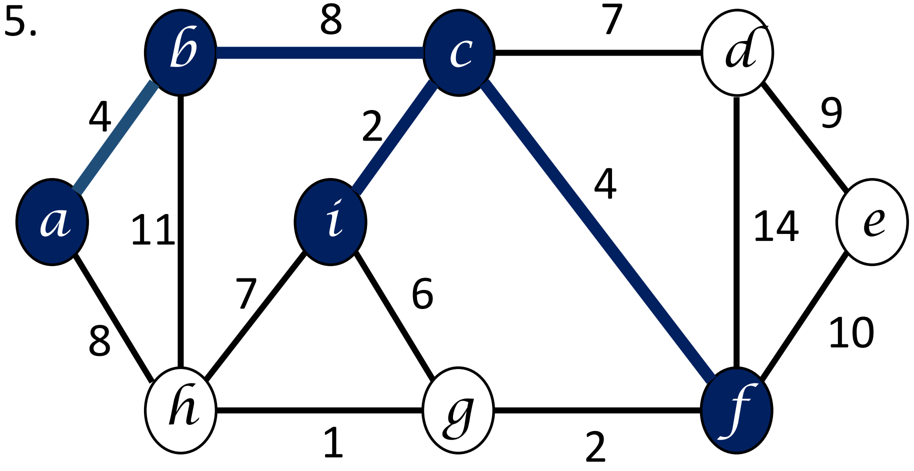


4.

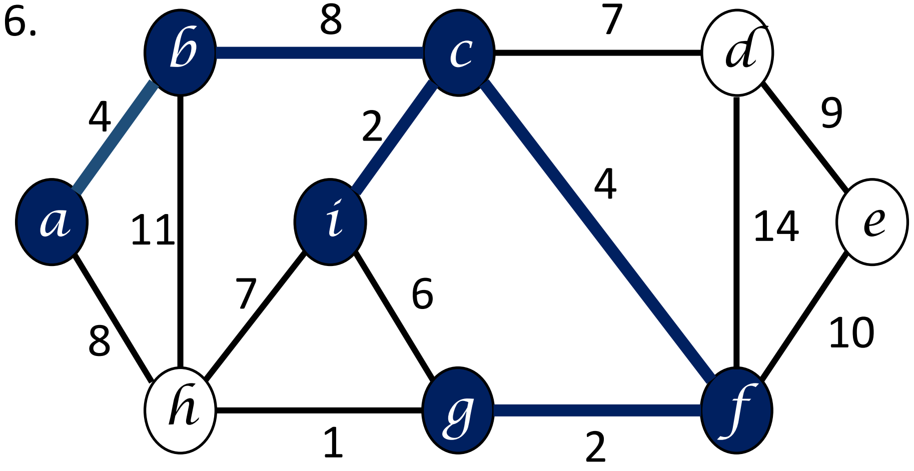


Example 2

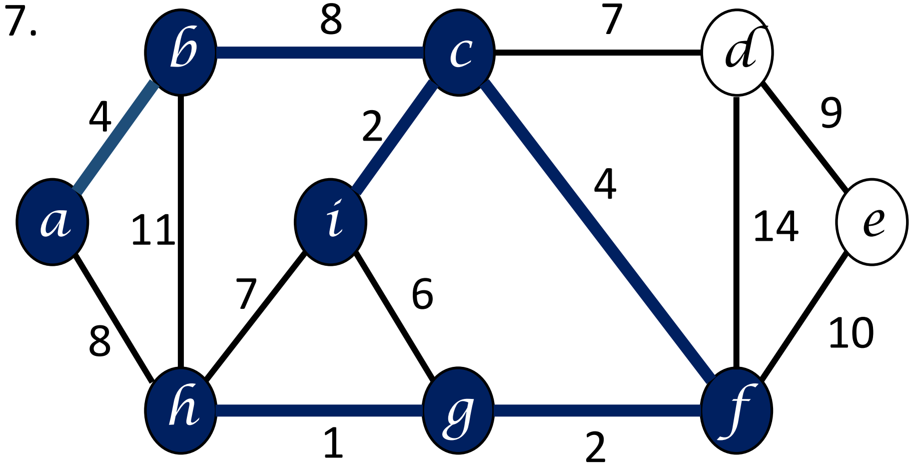
5.



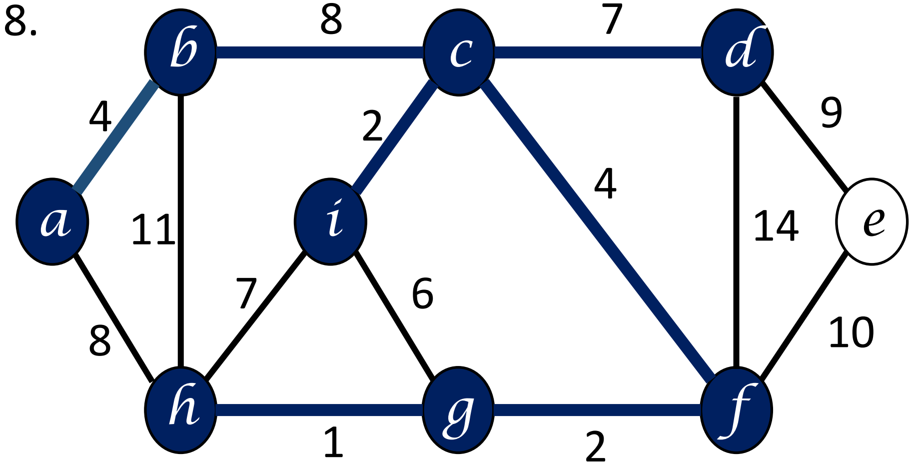
6.



7.

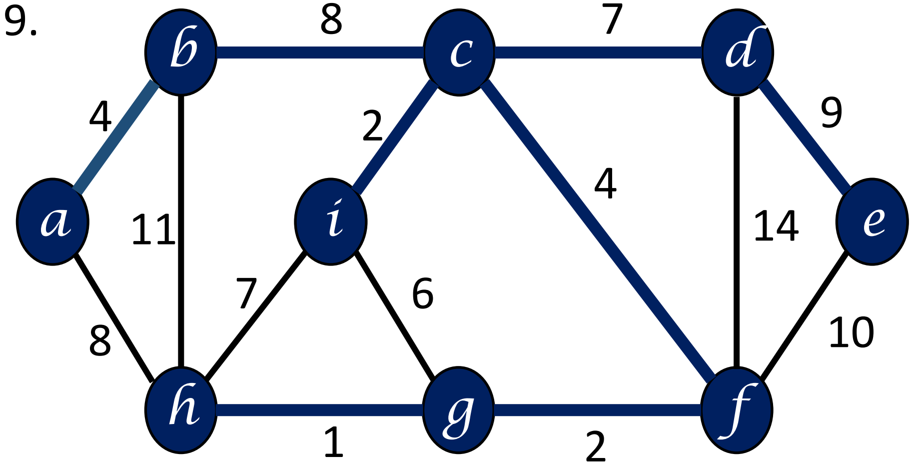


8.



Example 2

9.



Total Weight = 37

Pseudo-code

```
MST-PRIM(G, w, r)
for each u ∈ G.V
    u.key = ∞
    u.π = NIL
r.key = 0
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```

Pseudo-code Using Binary Heap as Priority Queue

```
PRIM(V, E, w, r)
Q ← Ø
for each u ∈ V
     do key[u] ← ∞
        π[u] ← NIL
        INSERT(Q, u)
DECREASE-KEY(Q, r, 0)     ▷ key[r] ← 0
while Q ≠ Ø
     do u ← EXTRACT-MIN(Q)
        for each v ∈ Adj[u]
            do if v ∈ Q and w(u, v) < key[v]
                  then π[v] ← u
                       DECREASE-KEY(Q, v, w(u, v))
```

Complexity Analysis

- Using heap for priority queue:
  - Each operation is $O(\log |V|)$
- Time complexity
  - # insert:
    - $O(\log |V|)$
  - # Decrease-Key:
    - $O(\log |V|)$
  - # Extract-Min
    - $O(\log |V|)$
- Total time complexity: Dominated by Last DECREASE-KEY operation for |E| no of times

  Total = $O(|E| \log |V|)$

Thank You