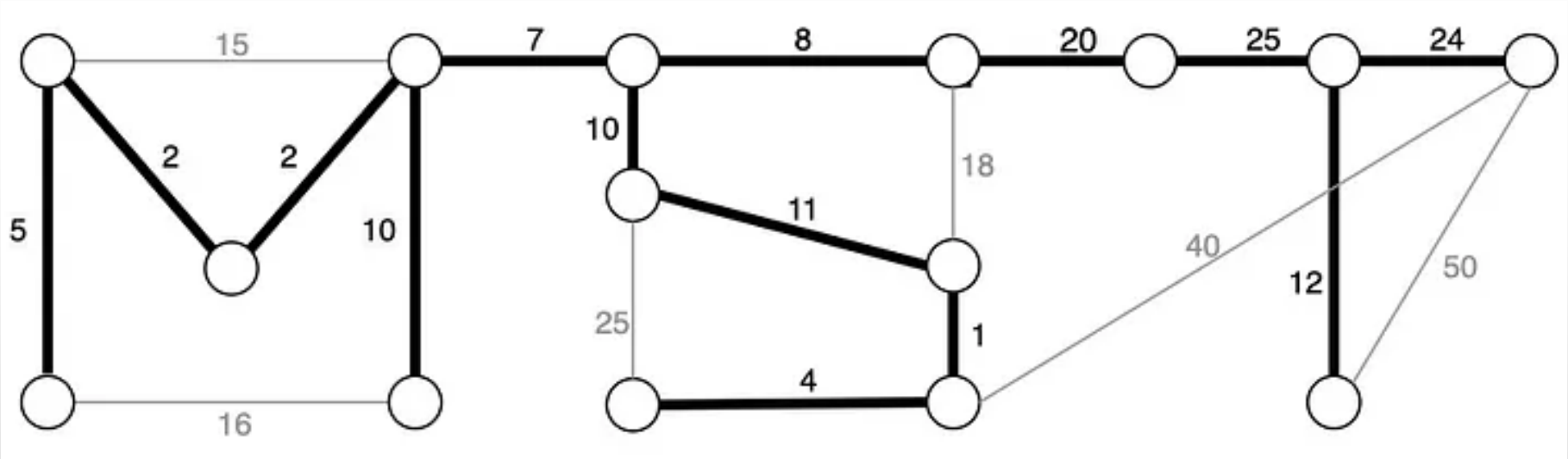
# Bölüm 4: Çizge Algoritmaları

## Algoritmalar

# Minimum Kapsayan Ağaç Algoritmaları

- Çizgedeki,
  - tüm düğümleri birbirine bağlayan ve
  - toplam kenar ağırlığının en az olduğu alt ağaçtır.
- *Kruskal*, kenarları ağırlıklarına göre sıralar ve döngü oluşturmayan kenarları seçerek ağacı oluşturur.
- *Prim*, başlangıç düğümünden başlayarak, her adımda en düşük ağırlıklı kenarı seçerek ağacı büyütür.

# Minimum Kapsayan Ağaç (MST)



# Kruskal Algoritması

- Çizgede tüm düğümleri birbirine bağlayan en kısa ağırlıklı ağacı oluşturur.
- Açgözlü (*greedy*) yaklaşım kullanır.
- *Joseph Kruskal* tarafından geliştirilmiştir.

# Algoritma İlkeleri

- Ağırlıklı çizge üzerinde çalışır.
- Tüm düğümleri en küçük ağırlıklı kenarları kullanarak birleştirir.
- Döngü oluşturmadan, minimum kapsayan ağacı oluşturur.
- Başlangıçta, her düğüm ayrı bir ağacı temsil eder.
  - Adım adım bu ağaçlar birleştirilir.

# Algoritma Adımları

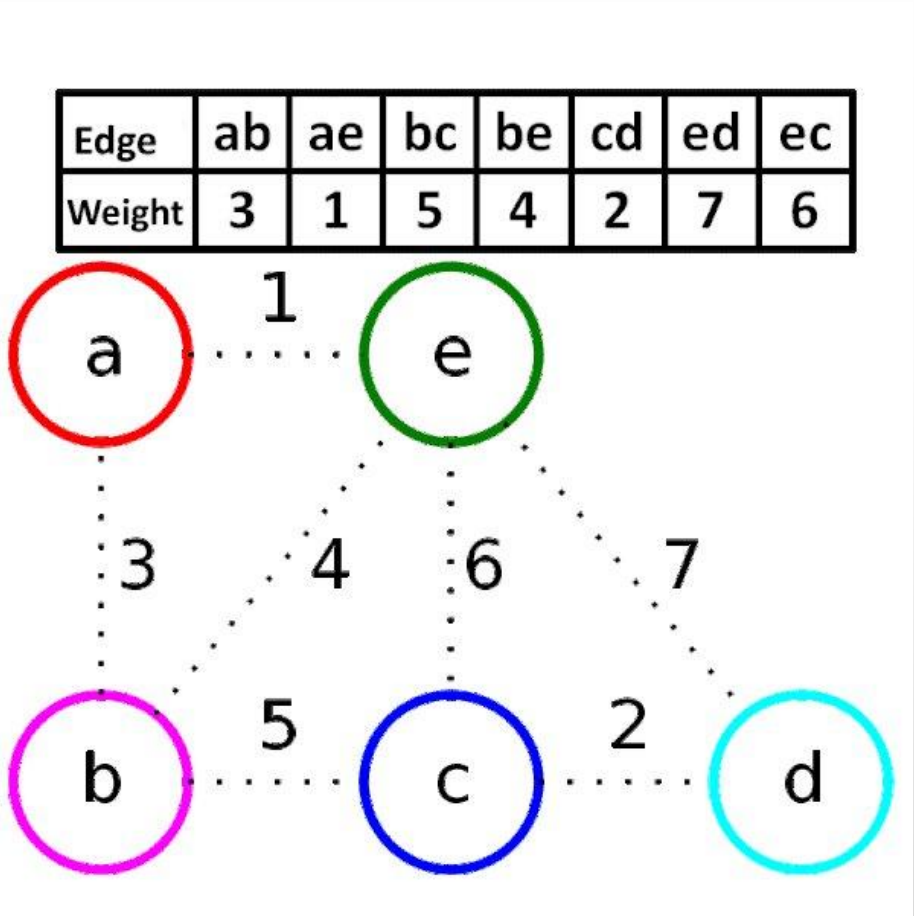
- Adım 1: Çizge içindeki tüm kenarlar ağırlıklarına göre sıralanır.
- Adım 2: Sıralı kenarlar arasından en küçük ağırlıklı kenar seçilir.
- Adım 3: Seçilen kenar, döngü oluşturmuyorsa, ağaç içine eklenir.
- Adım 4: Eğer seçilen kenar, farklı ağaçlara ait düğümleri birleştiriyorsa, bu kenar ağaç içine eklenir ve ağaçlar birleştirilir.

# Karmaşıklık Analizi

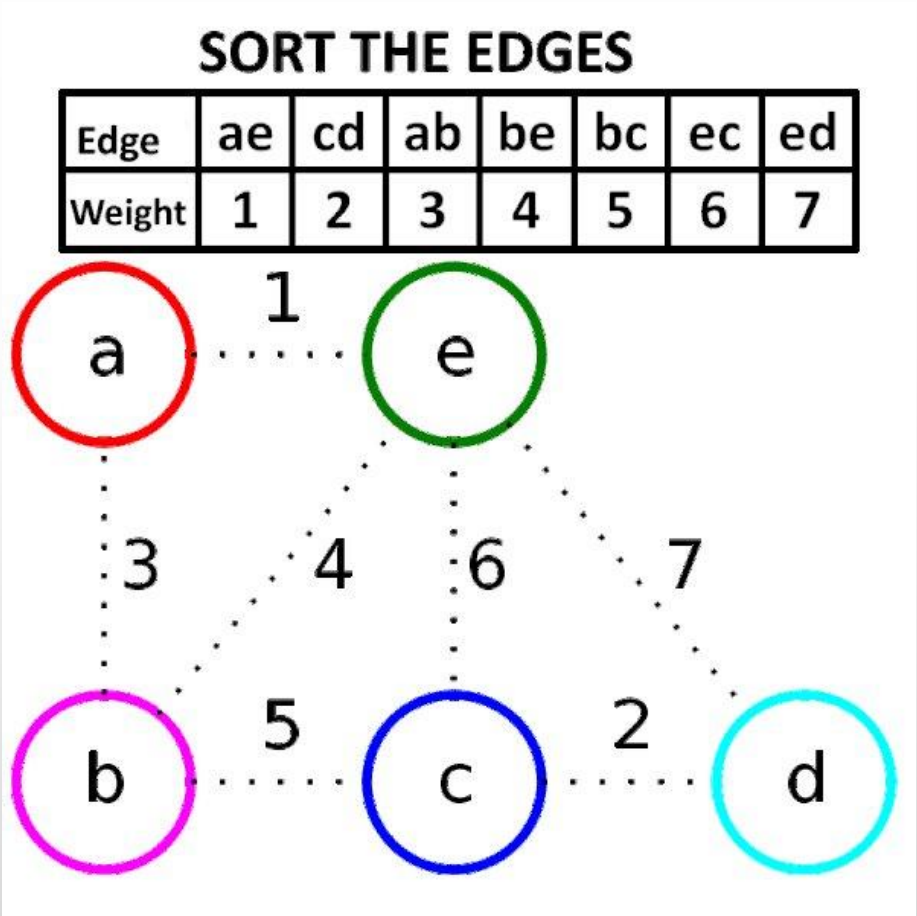
- Kenarların ağırlıklarına göre sıralanması:
  - *O(E log E)*
- Birleştirme-bulma (*union-find*) işlemleri:
  - *O(log E)* (*amortize* edilmiş)

# Kruskal

| Edge | ab | ae | bc | be | cd | ed | ec |
|---|---|---|---|---|---|---|---|
| Weight | 3 | 1 | 5 | 4 | 2 | 7 | 6 |

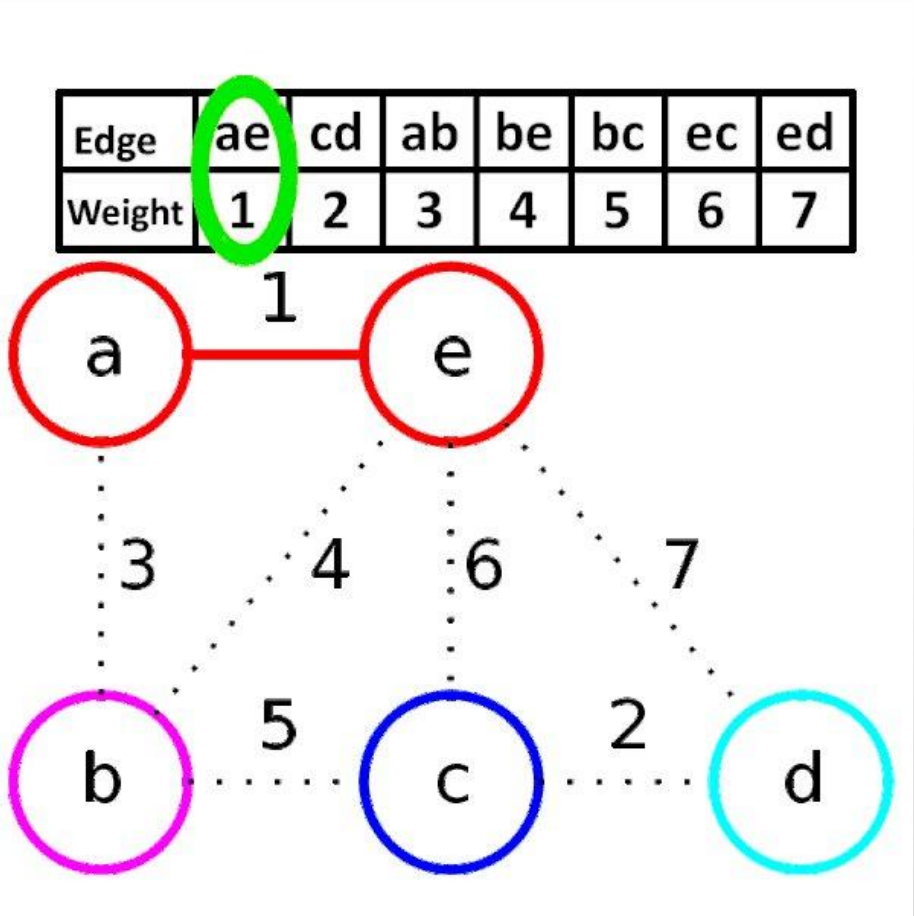


# Kruskal

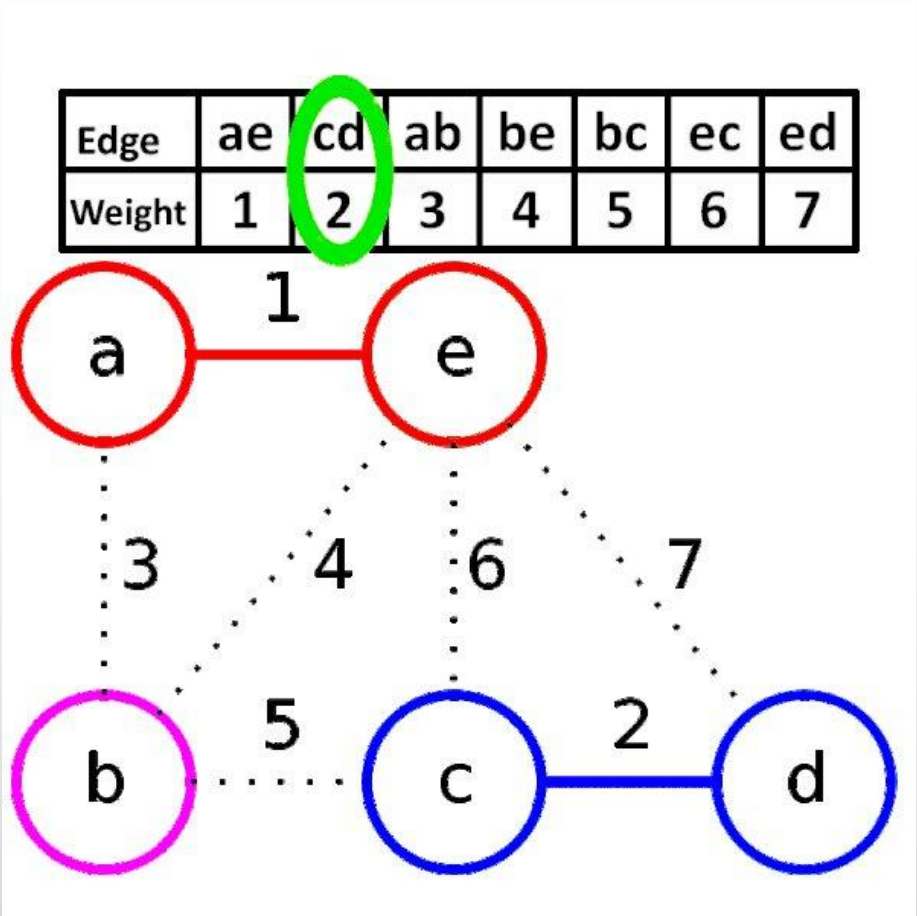


# Kruskal

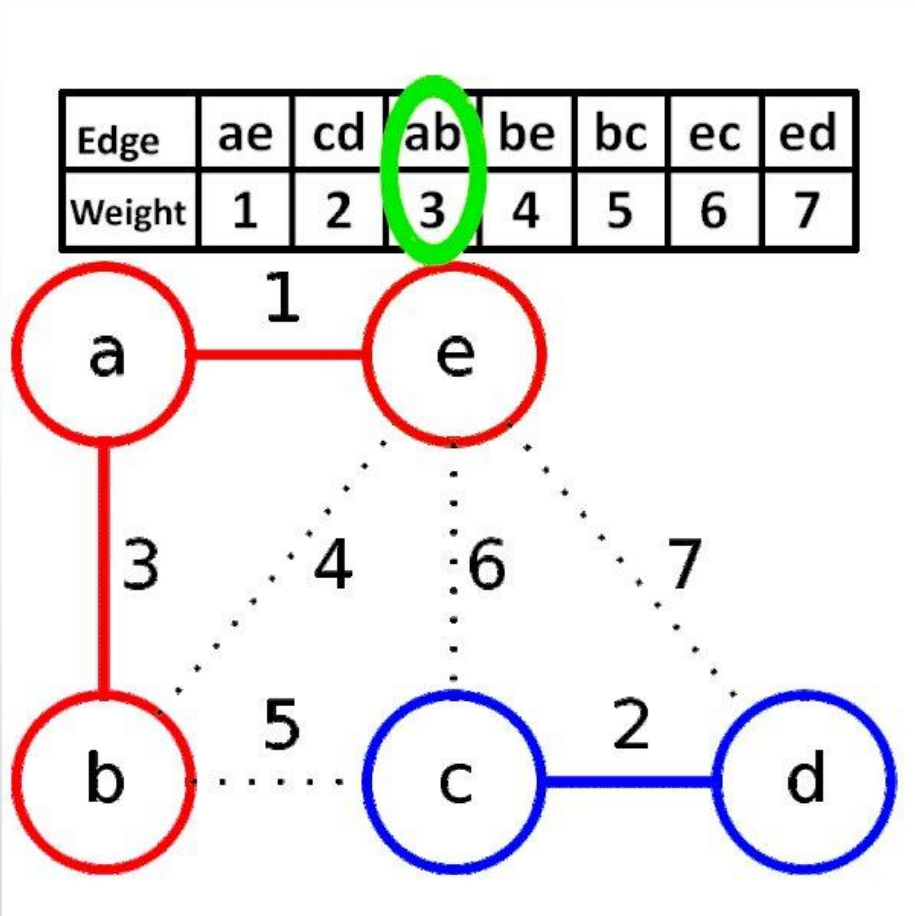
| Edge | ae | cd | ab | be | bc | ec | ed |
|---|---|---|---|---|---|---|---|
| Weight | 1 | 2 | 3 | 4 | 5 | 6 | 7 |



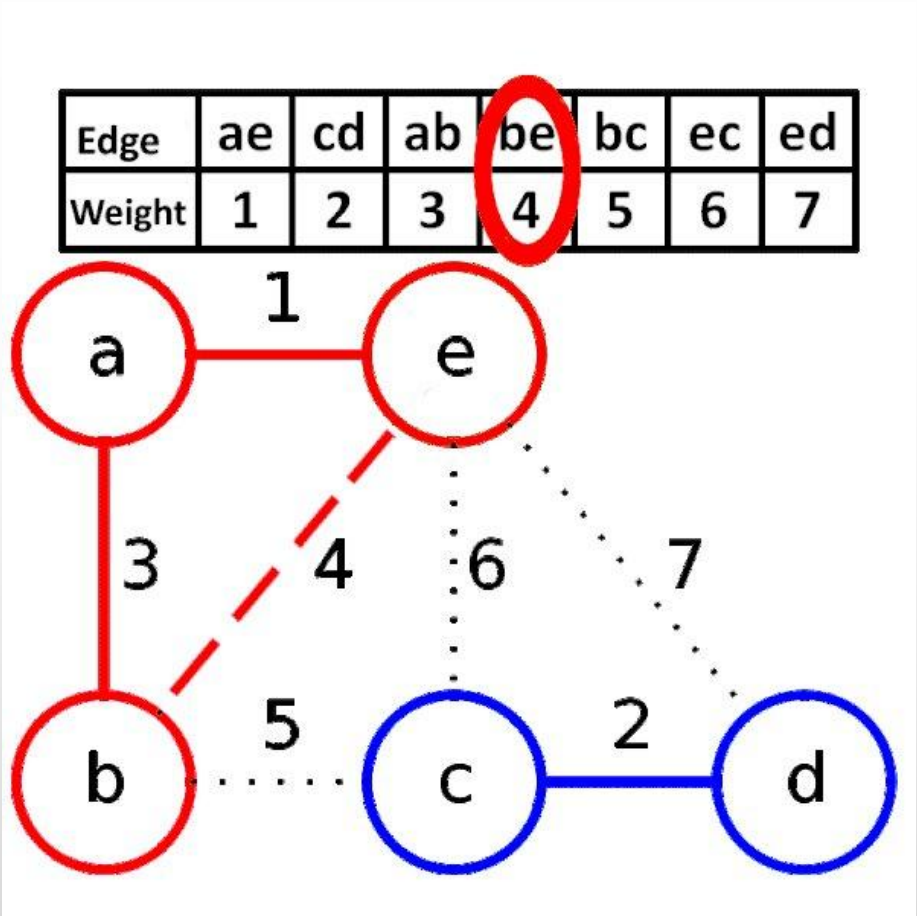
# Kruskal



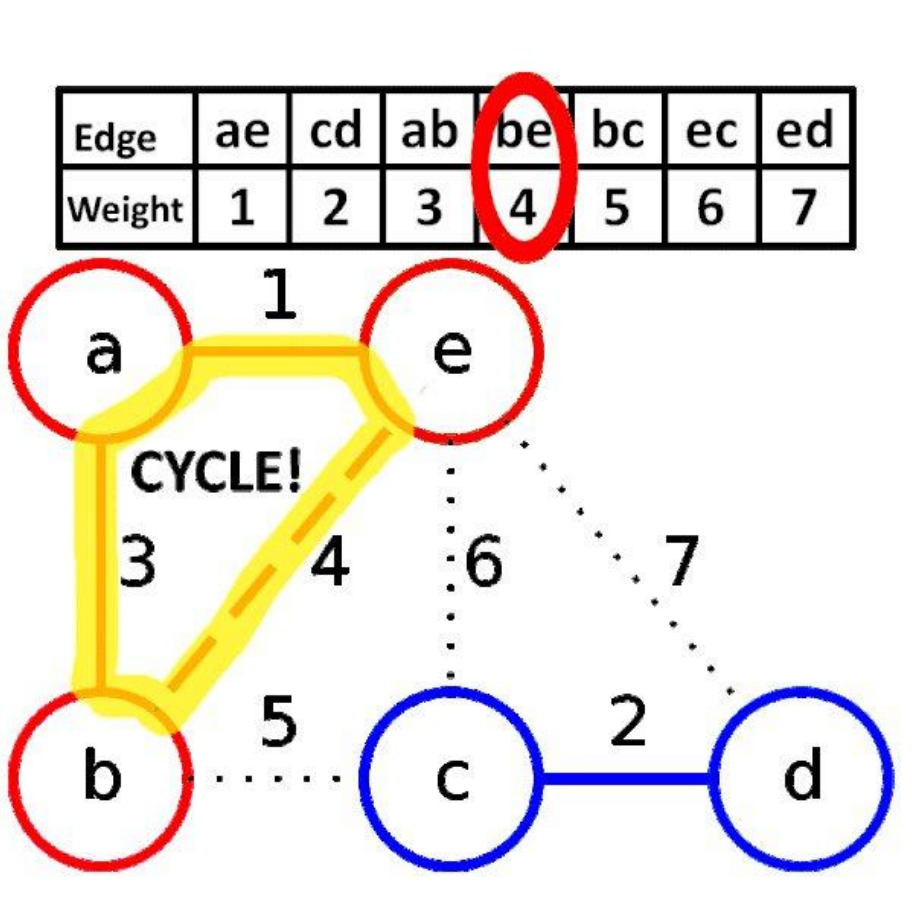
# Kruskal



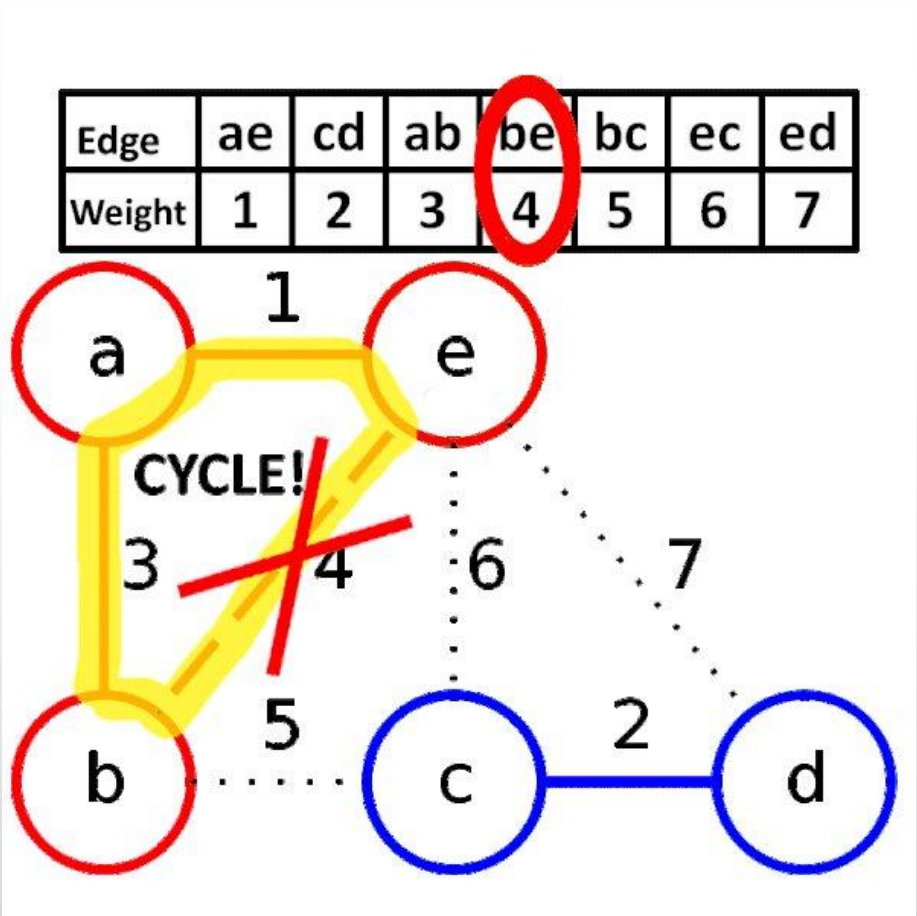
# Kruskal



# Kruskal

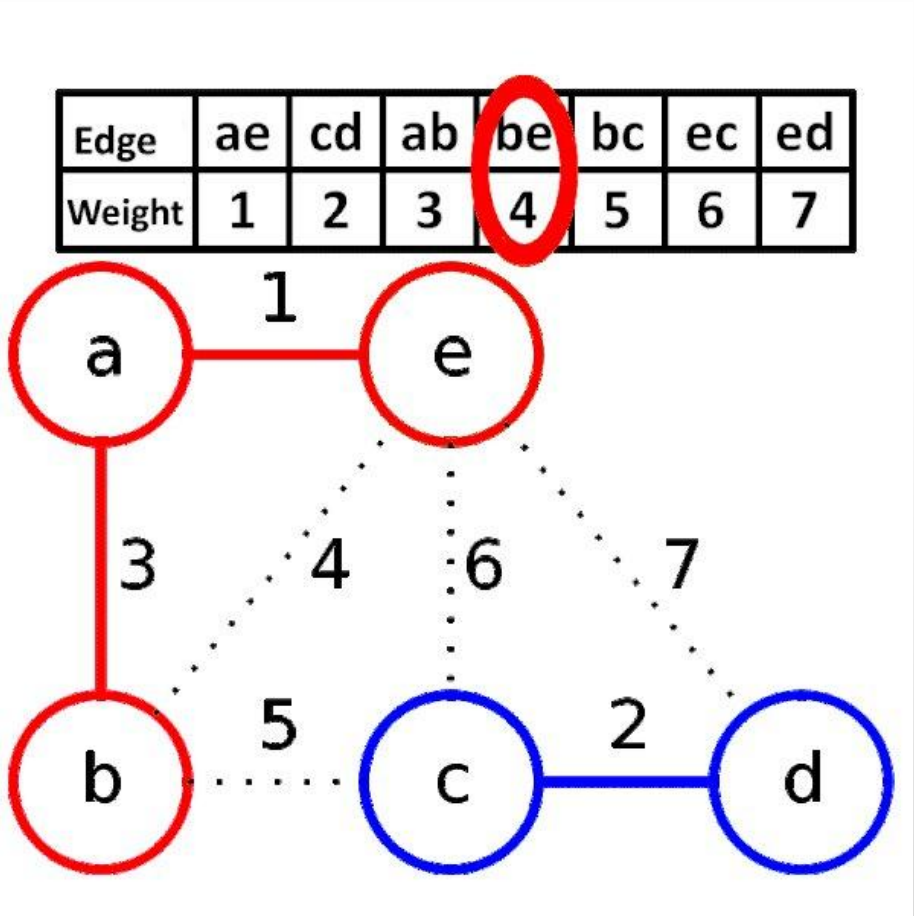


# Kruskal

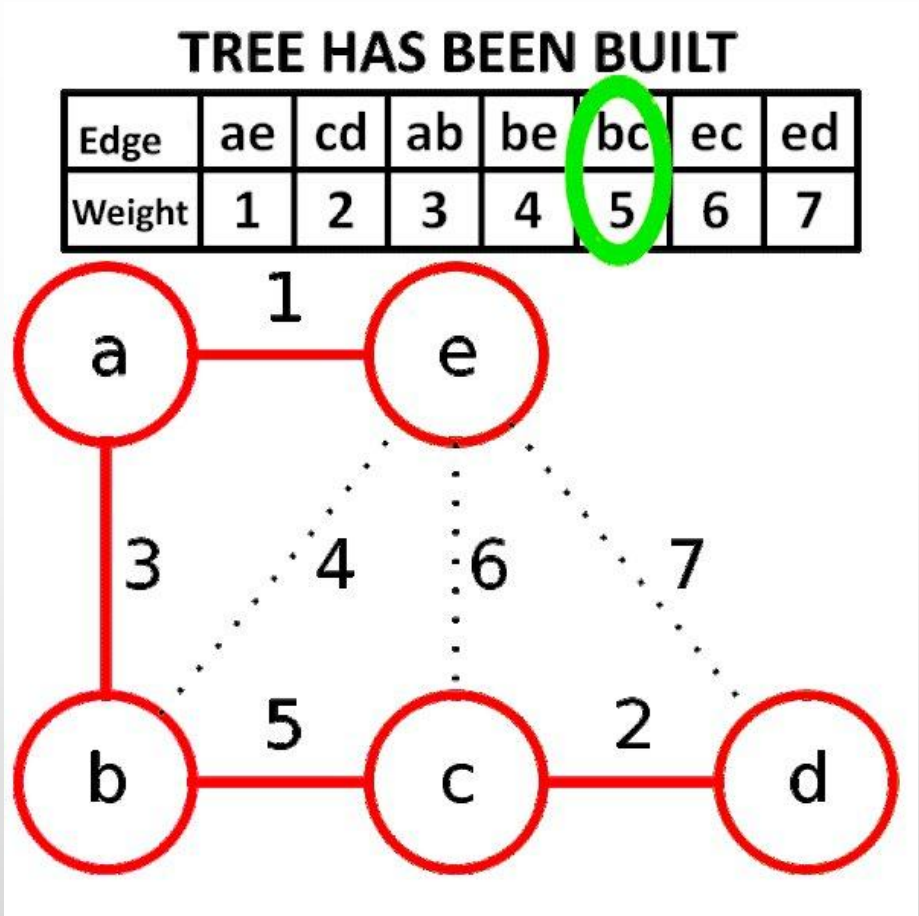


# Kruskal

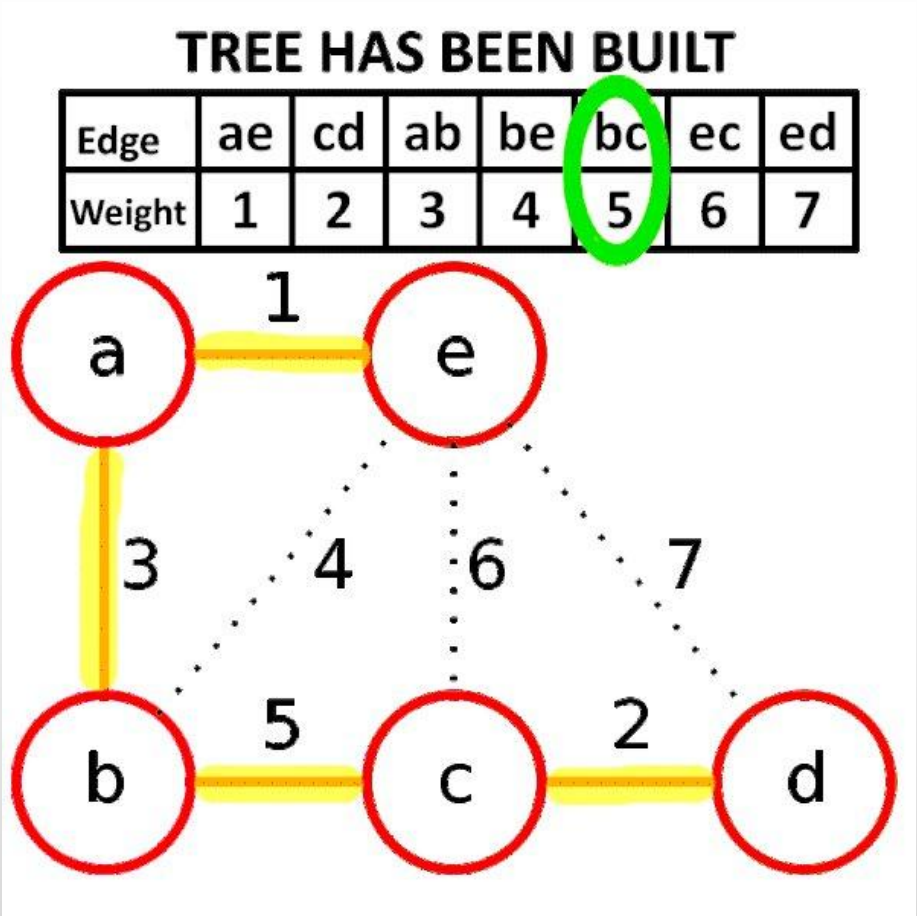
| Edge | ae | cd | ab | be | bc | ec | ed |
|---|---|---|---|---|---|---|---|
| Weight | 1 | 2 | 3 | 4 | 5 | 6 | 7 |



# Kruskal

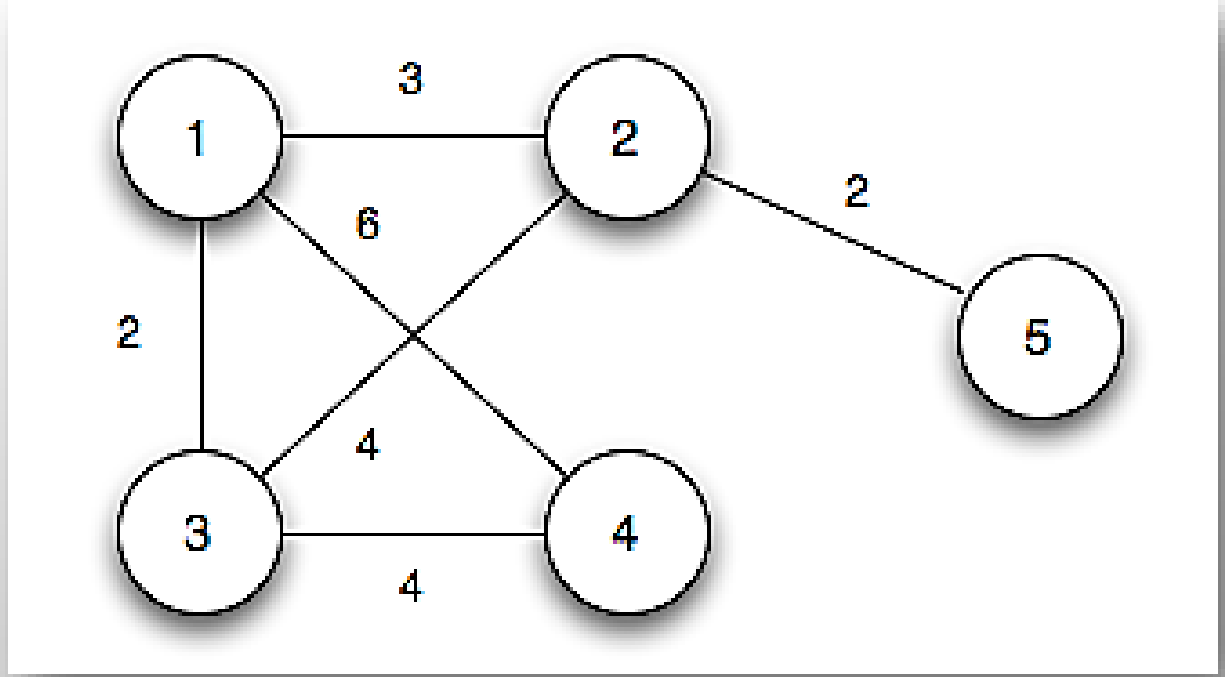


# Kruskal



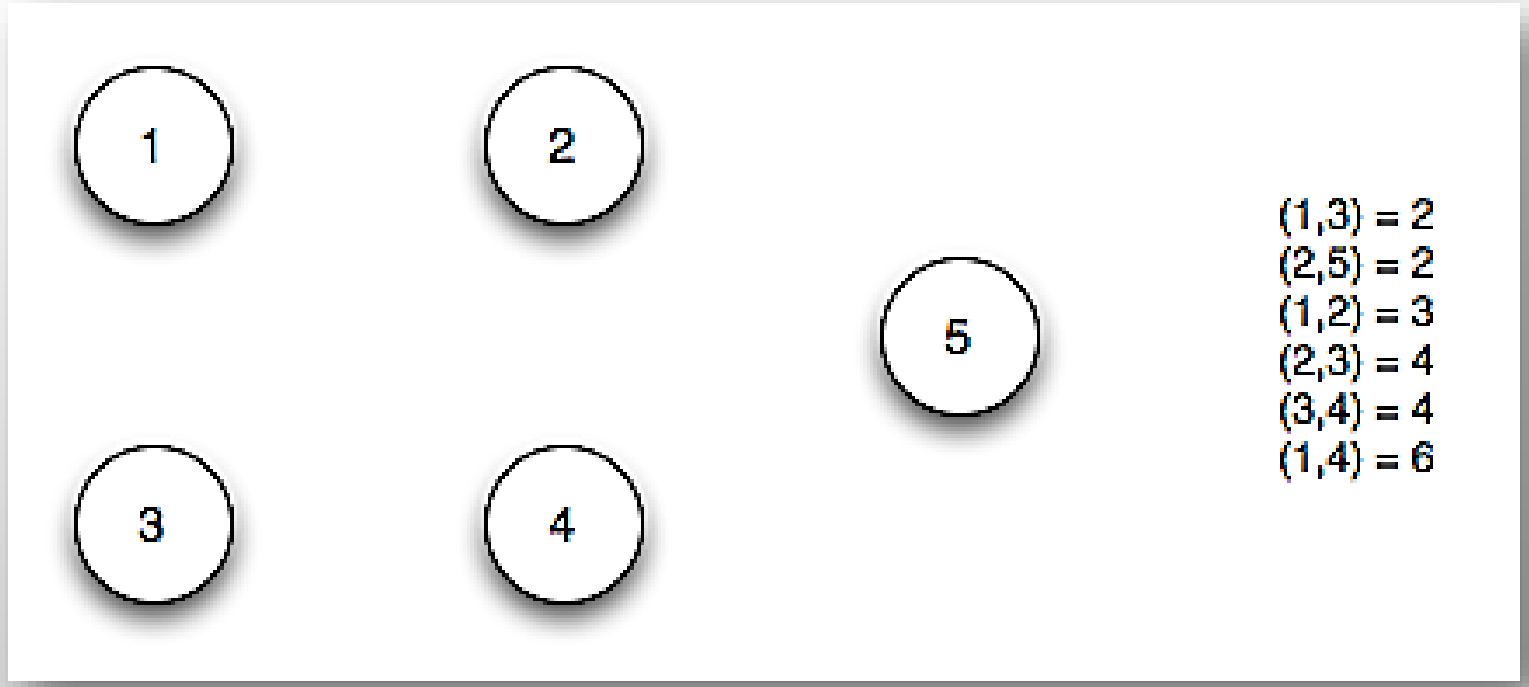
# Örnek

- Aşağıdaki yönsüz çizge verilsin.



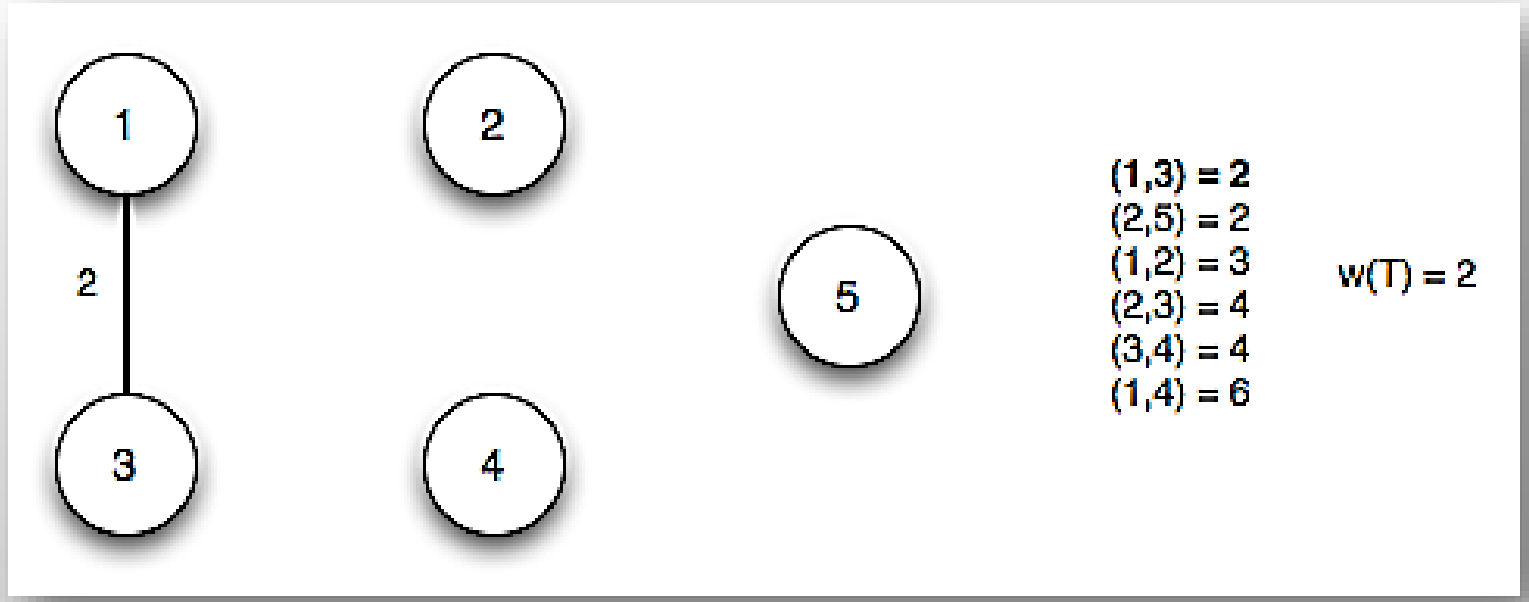
# İlklendirme Aşaması

- Her düğüm ayrı bir ağaç yapılır.
- Kenarlar ağırlıklarına göre sıralanır.



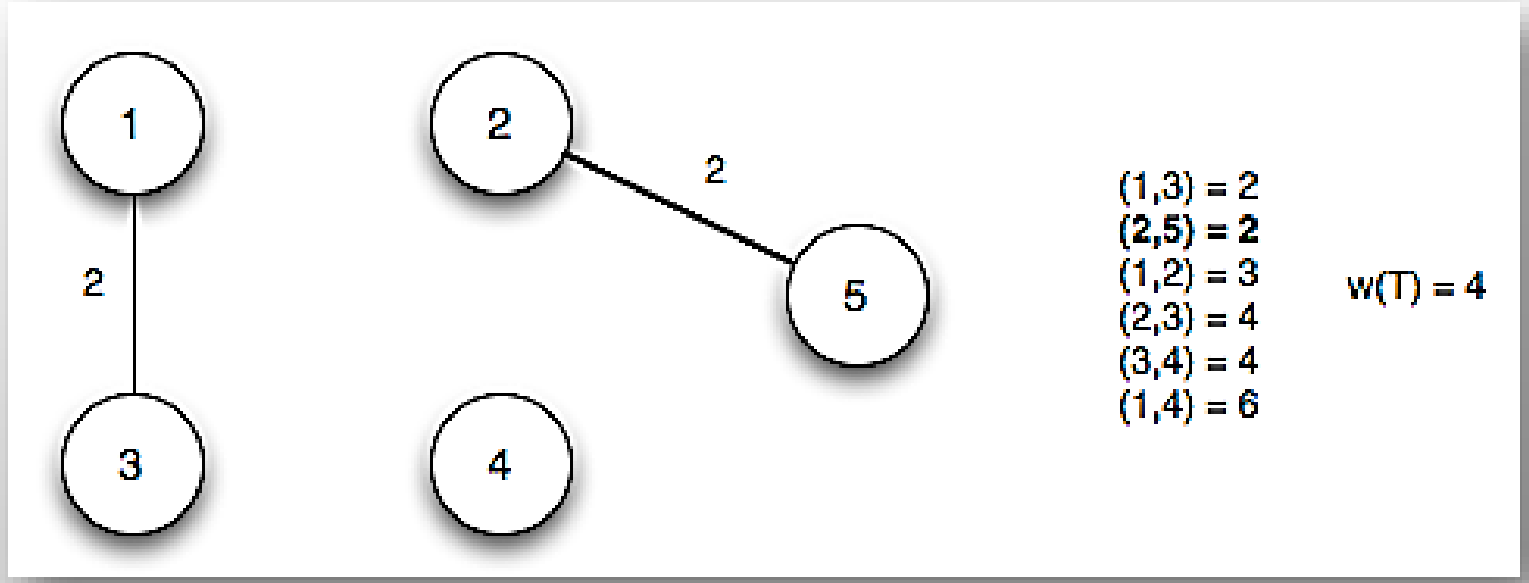
# Adım 1

- (u1,u3) kenarını ekle
- v1 ve v3 birleştir



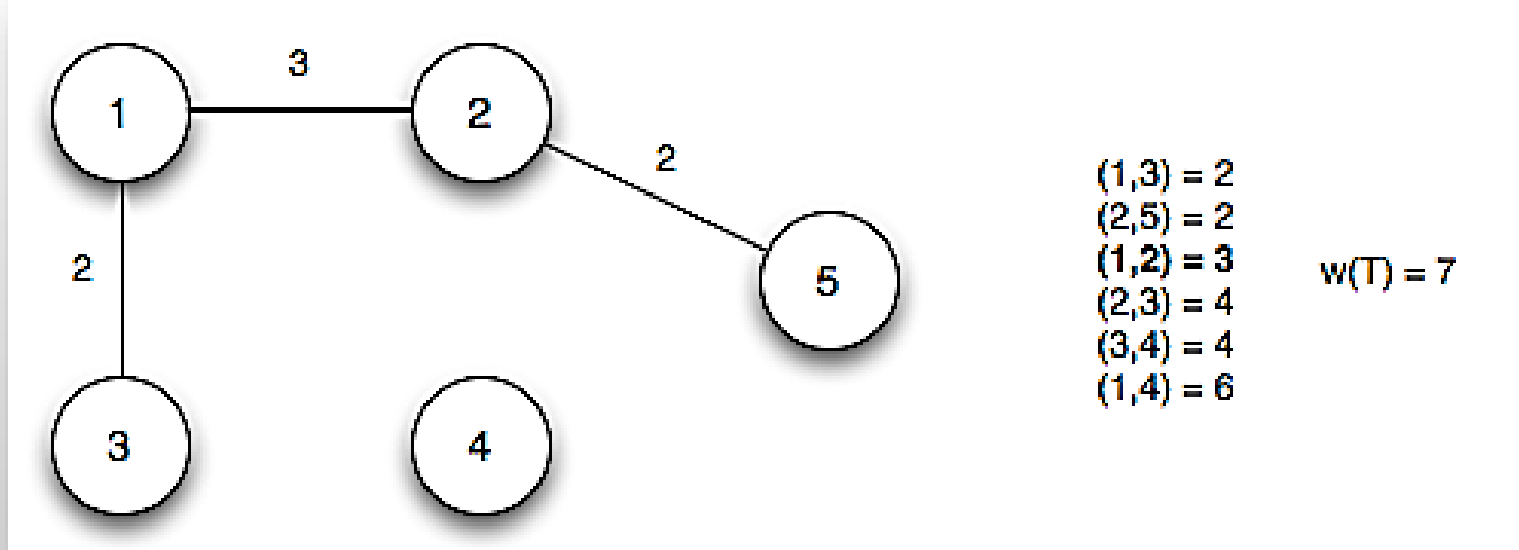
# Adım 2

- (u2,u5) kenarını ekle.
- v2 ve v5 birleştir.



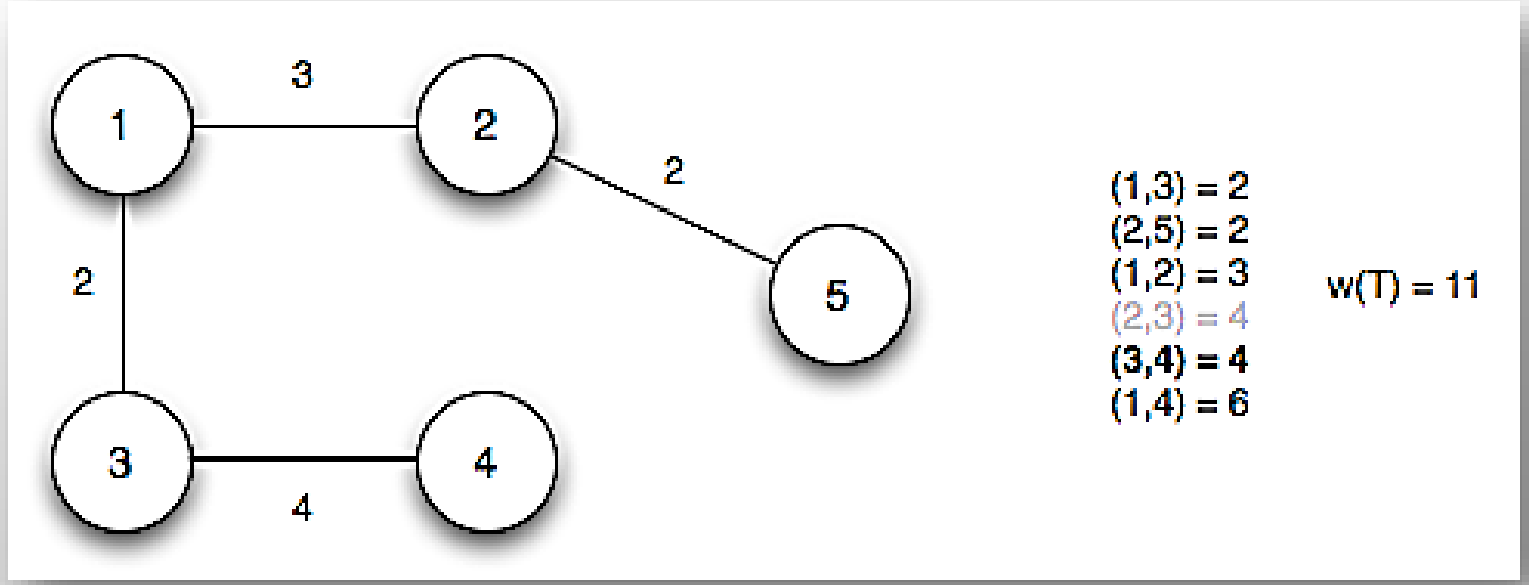
# Adım 3

- (u1,u2) kenarını ekle.
- Adım 1 ve Adım 2'de oluşan iki ağacı birleştir.



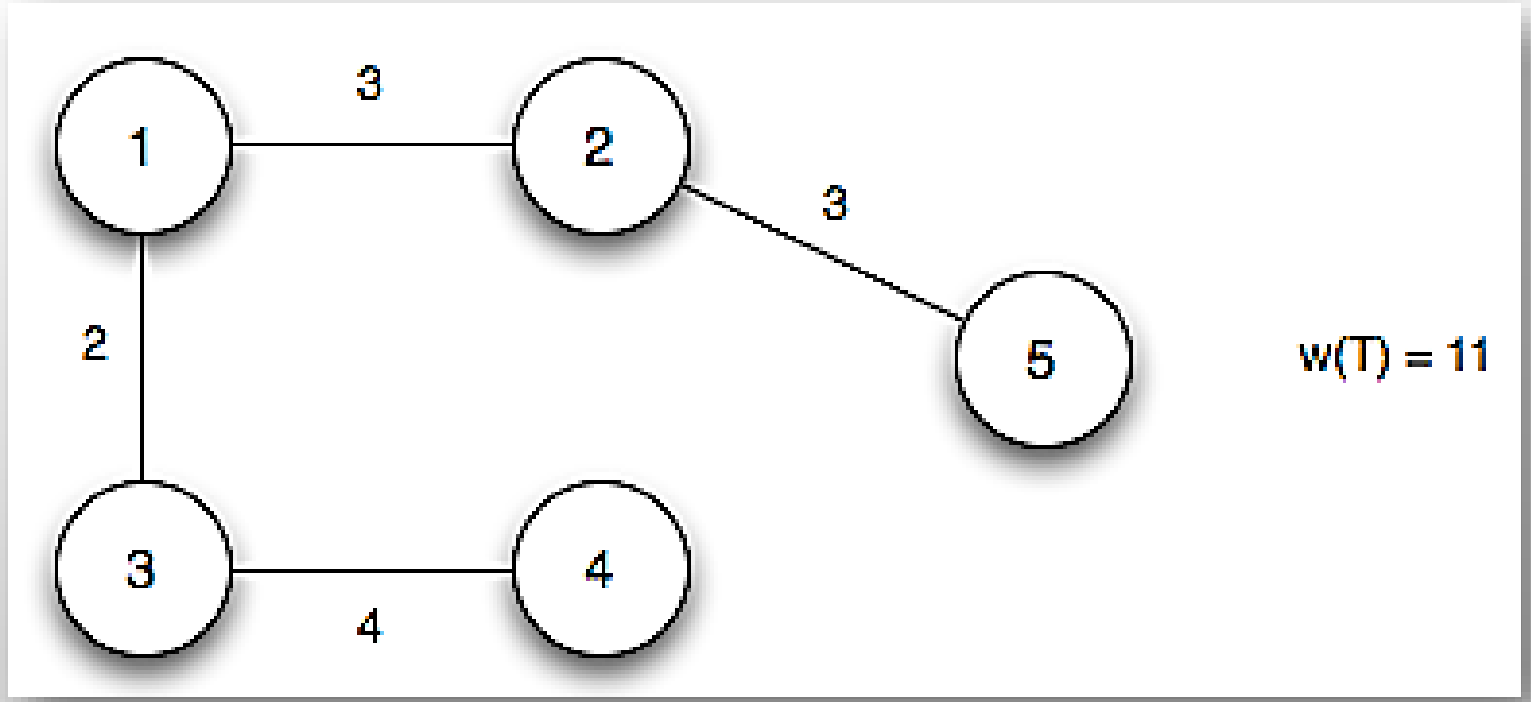
# Adım 4

- (u3,u4) kenarını ekle, v4 birleştir.
- (u2,u3) ayrı ağaçları birleştirmiyor!



# Son Durum

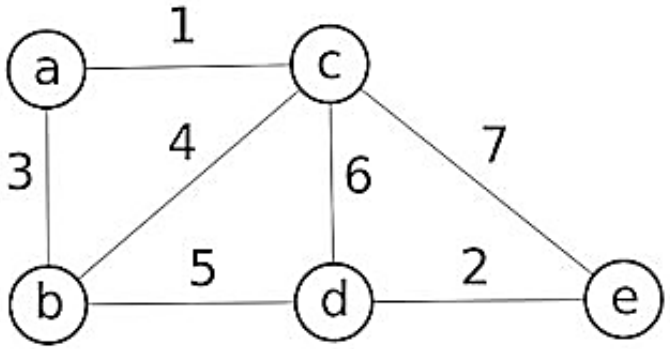
- (u1,u4) kenarı ayrı ağaçları birleştirmiyor!

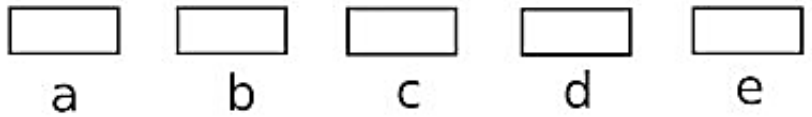


# Sözde Kod

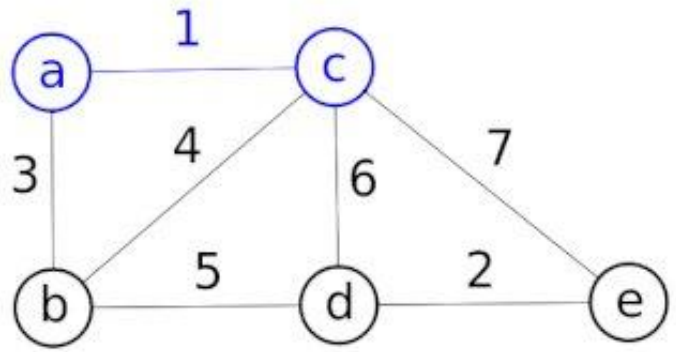
```
KRUSKAL(G):
        MST = [ ]
        union-find.oluştur(V)
        kenarlar.sırala(artana göre)

        her bir (u, v, ağırlık) için kenarlar içinde:
                eğer union-find.bul(u) ≠ union-find.bul(v):
                        MST.ekle((u, v, ağırlık))
                         union-find.birleştir(u, v)
        döndür MST
```

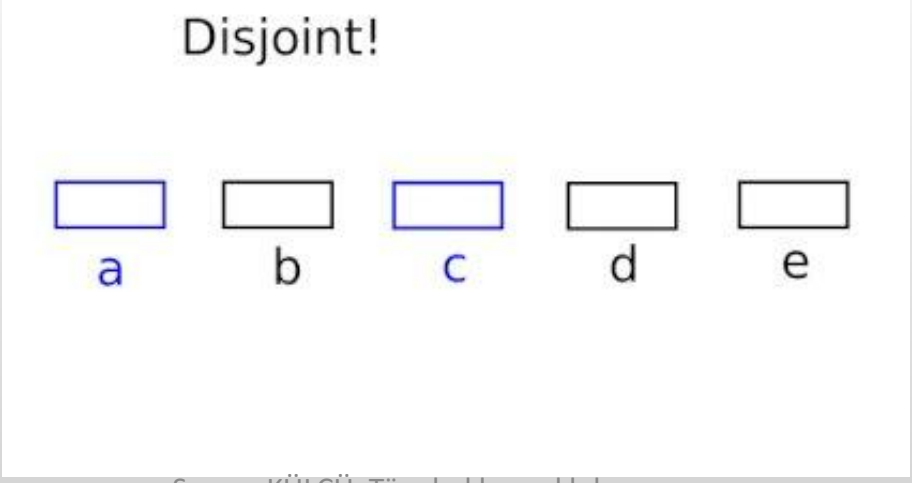
# Union Find



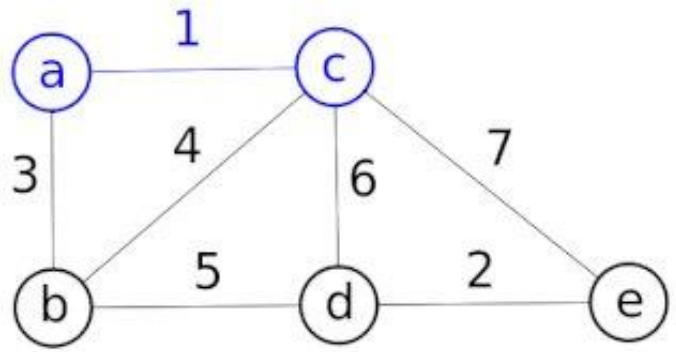



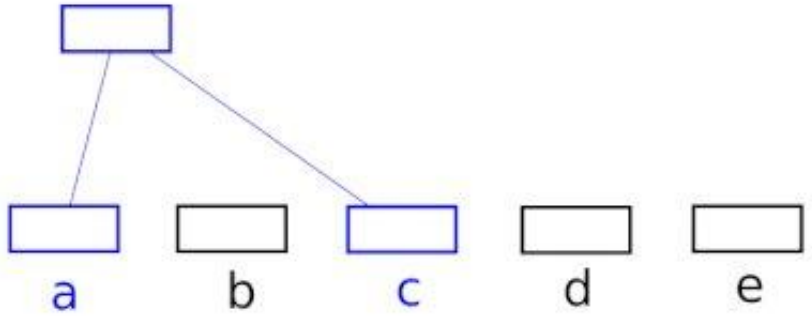
# Union Find



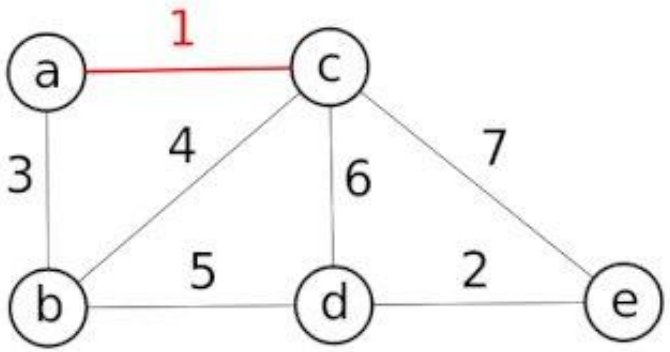



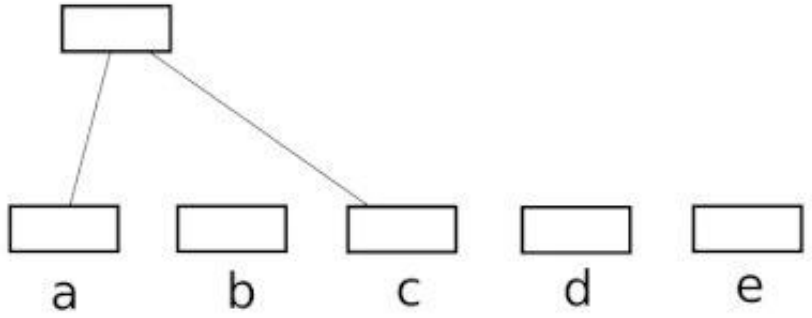
# Union Find



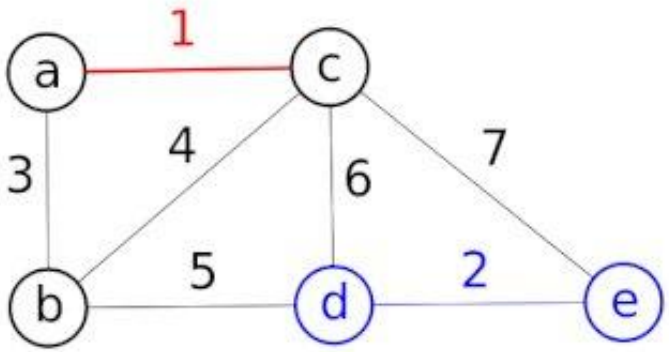



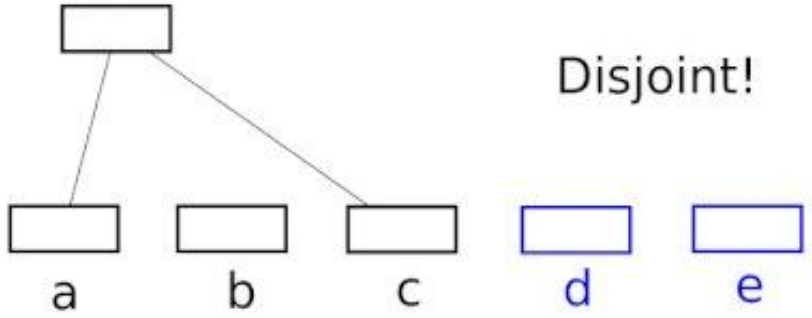
# Union Find



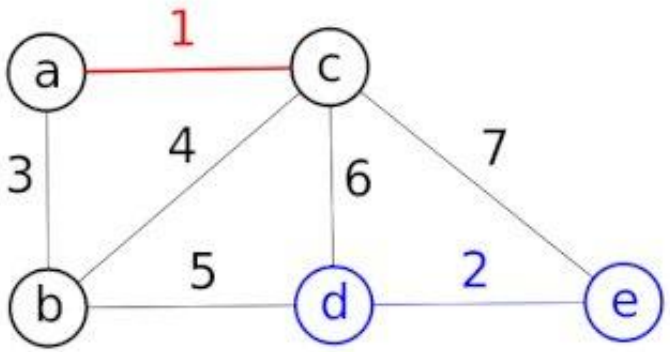



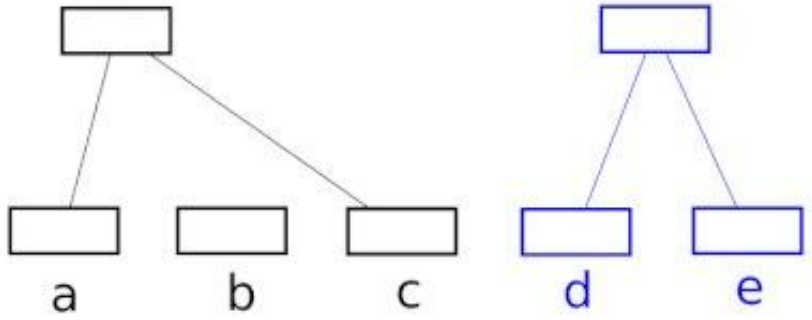
# Union Find



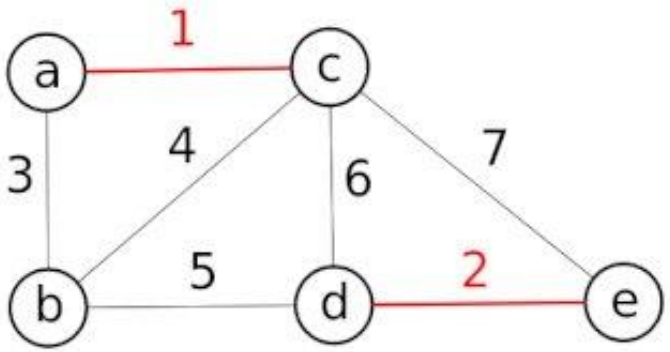



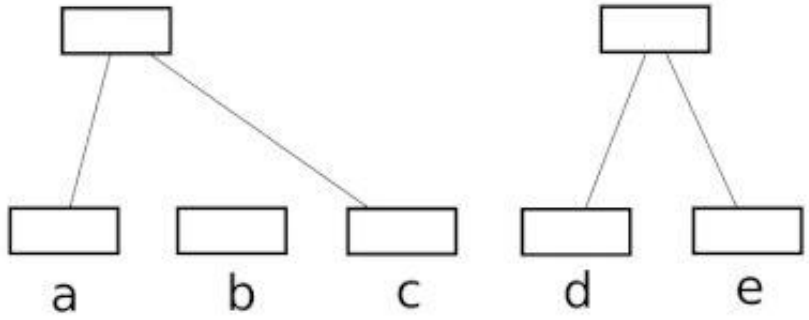
# Union Find



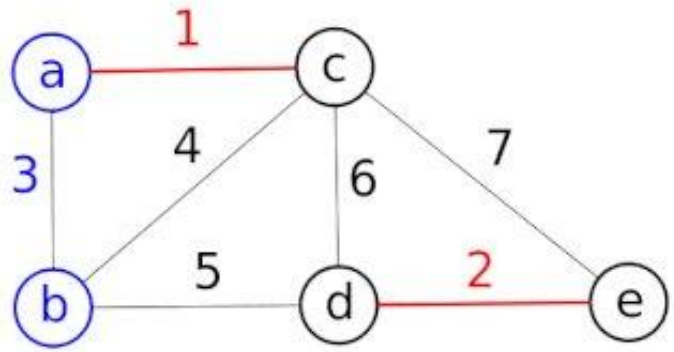



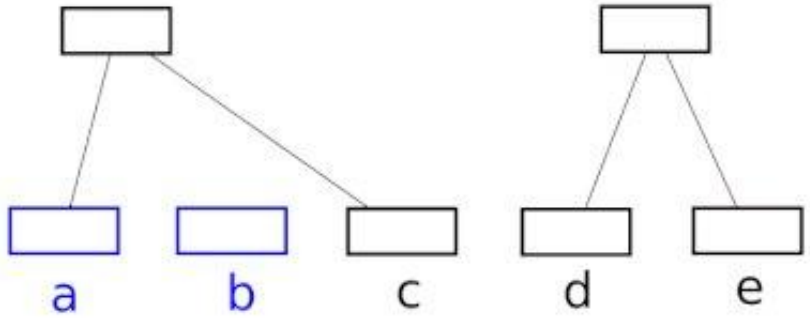
# Union Find



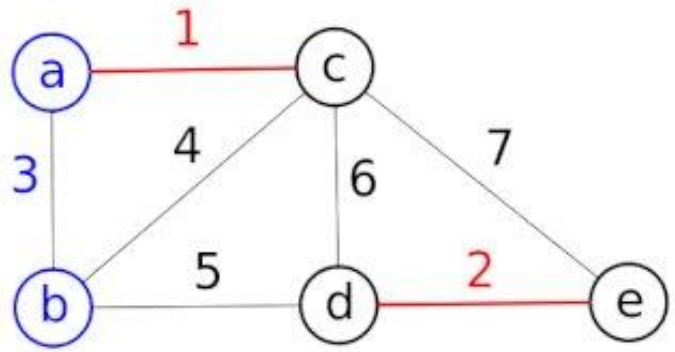



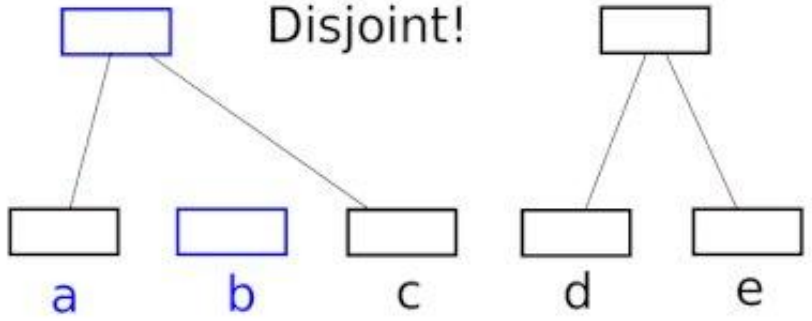
# Union Find



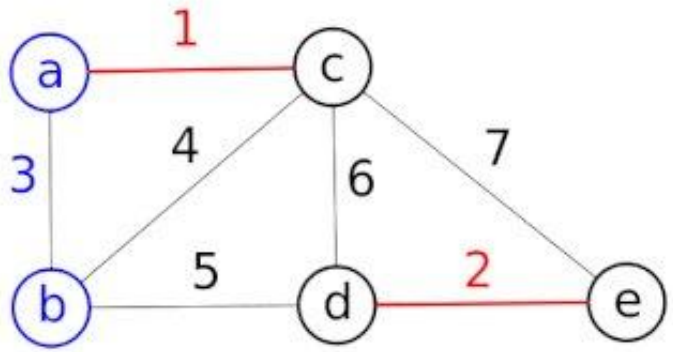



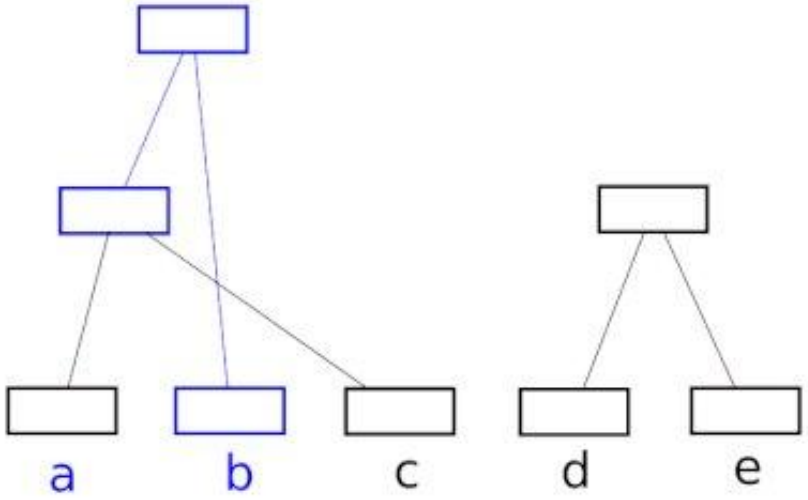
# Union Find



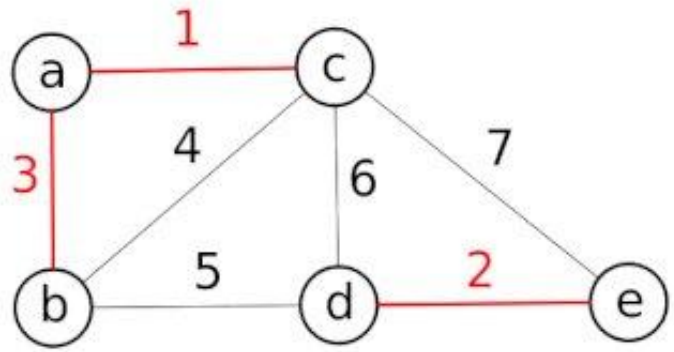



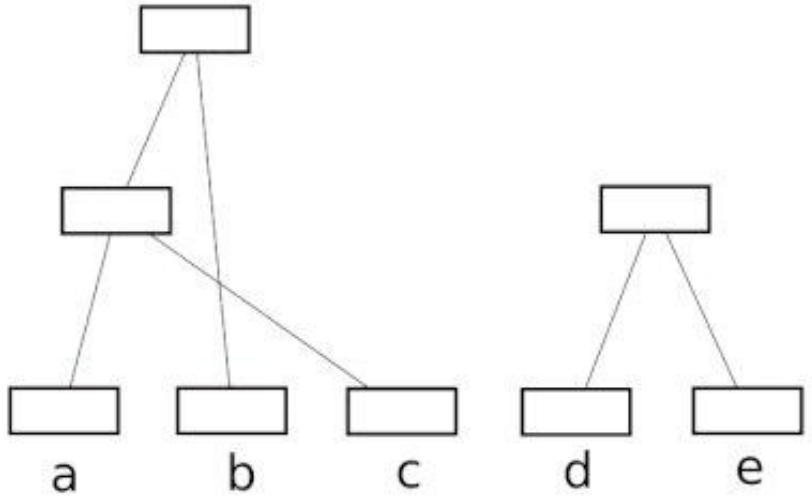
# Union Find



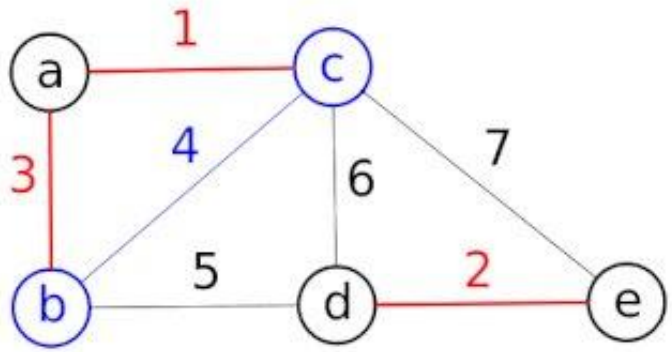



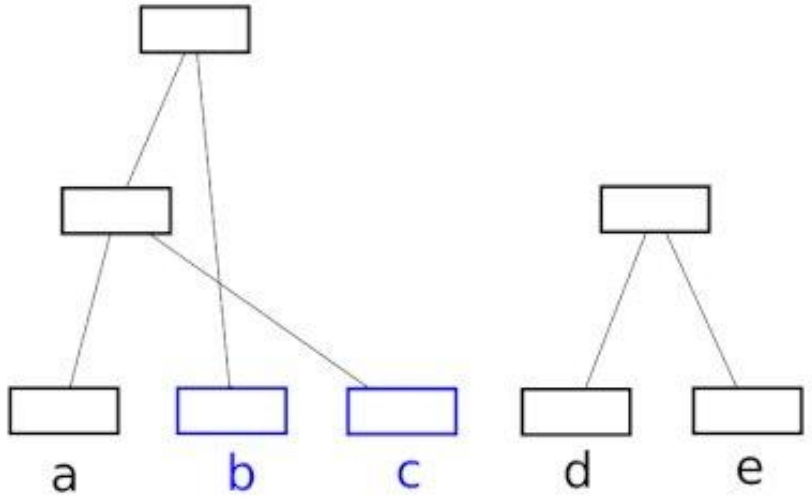
# Union Find



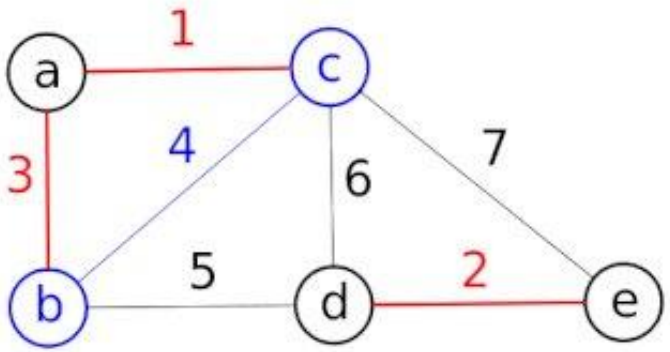



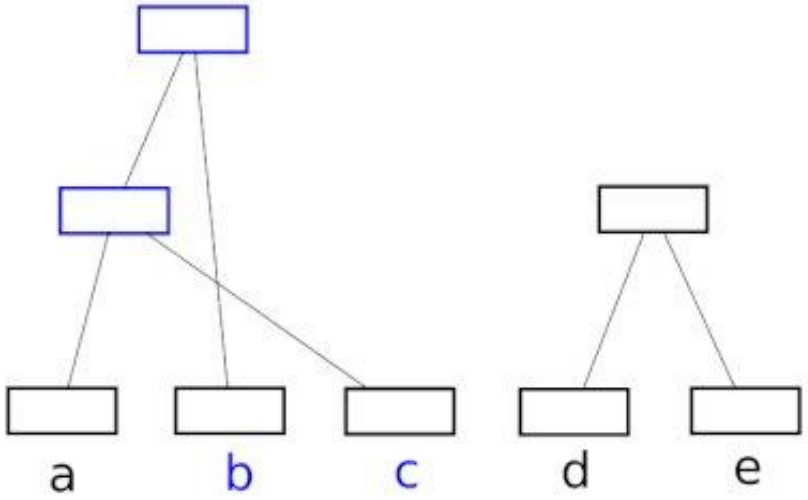
# Union Find



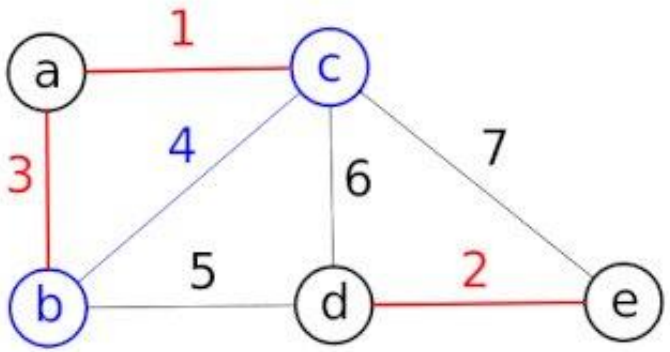



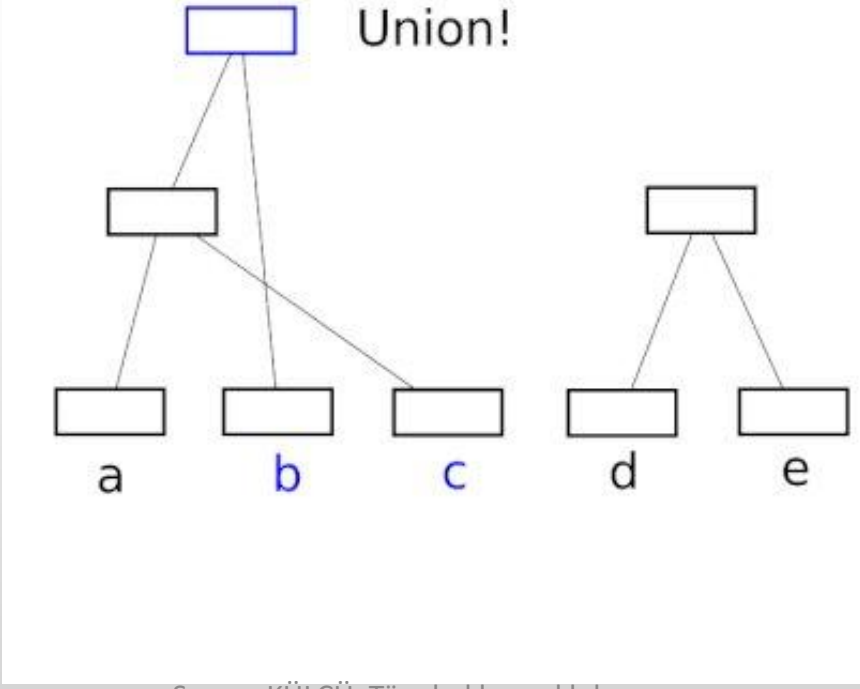
# Union Find



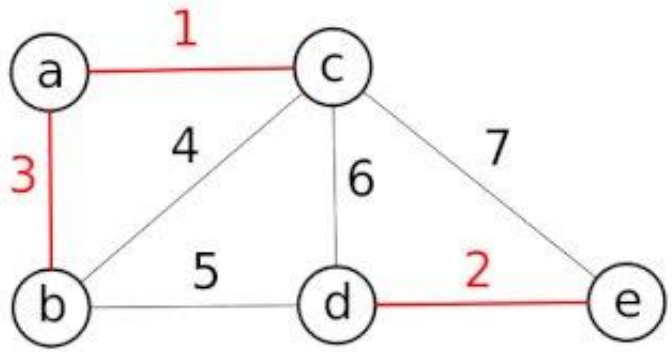



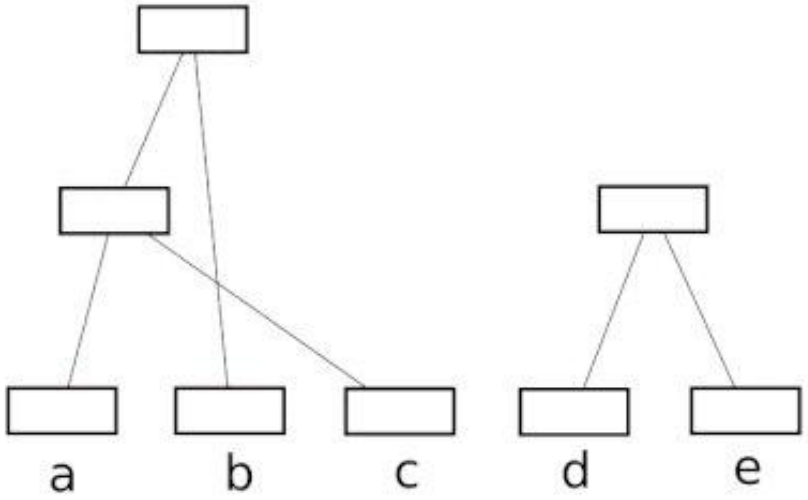
# Union Find



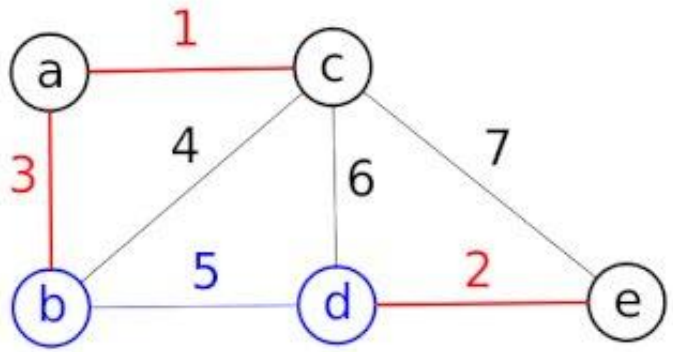



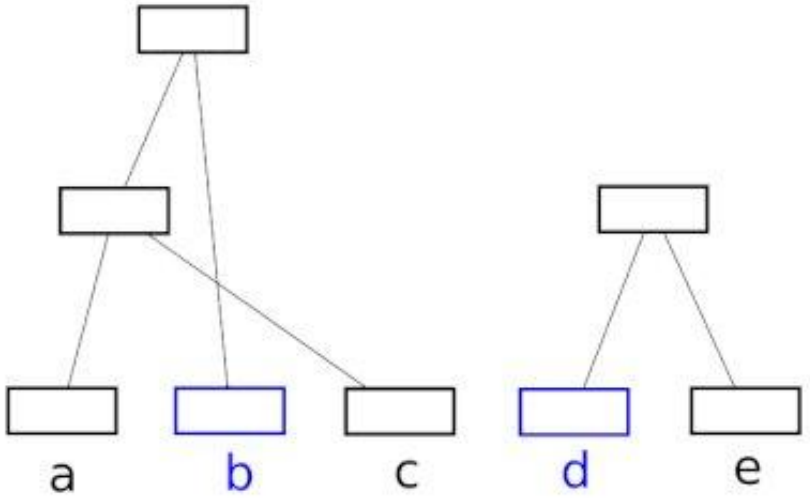
# Union Find



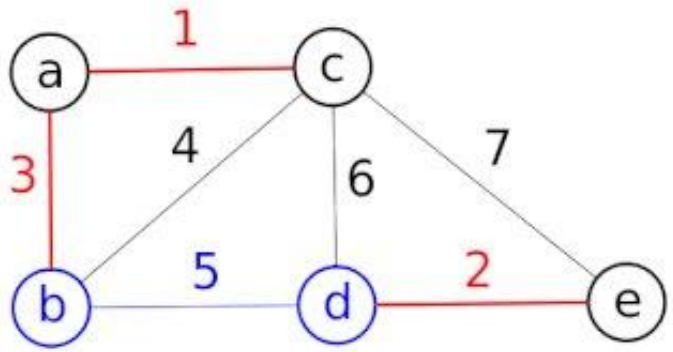



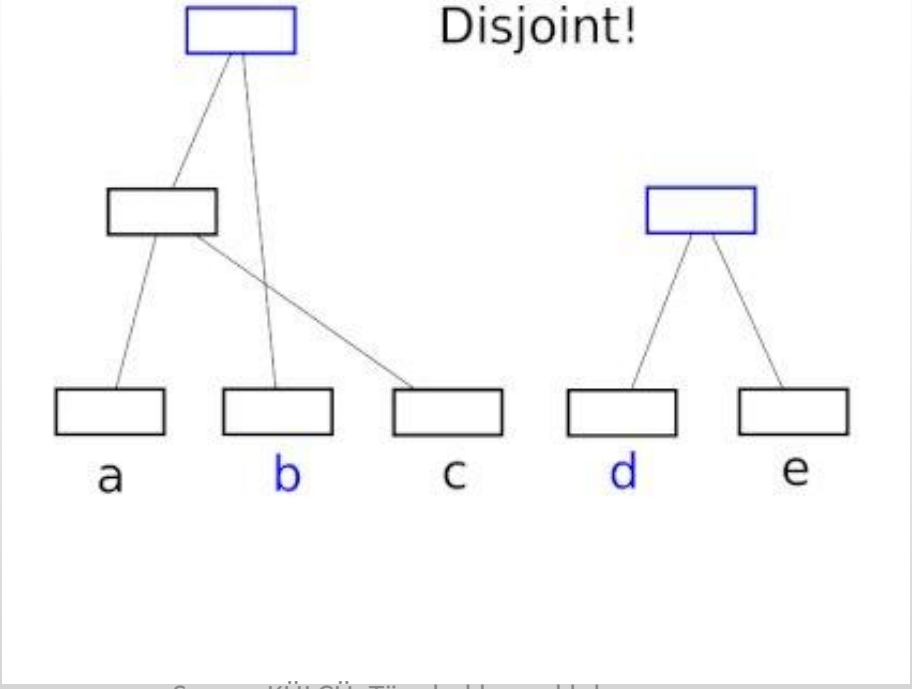
# Union Find



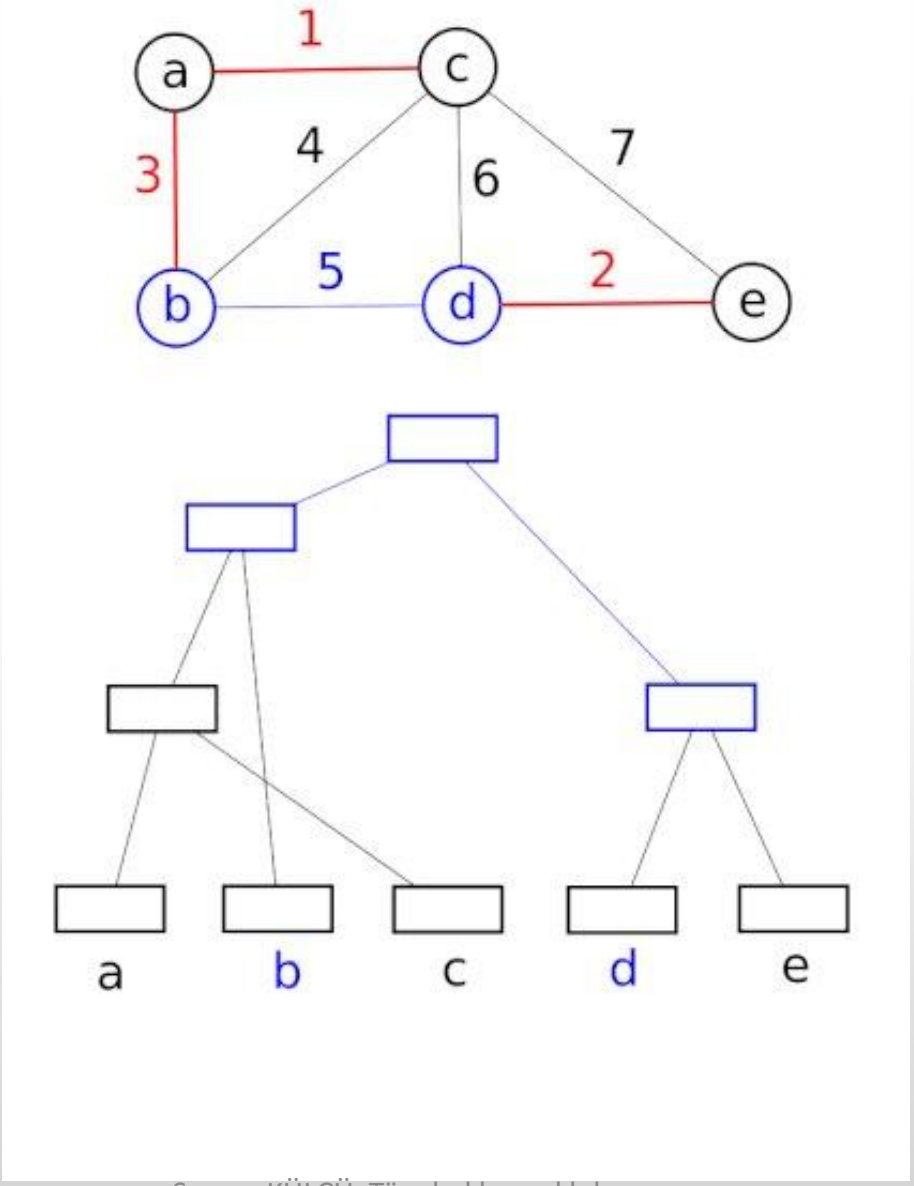



# Union Find

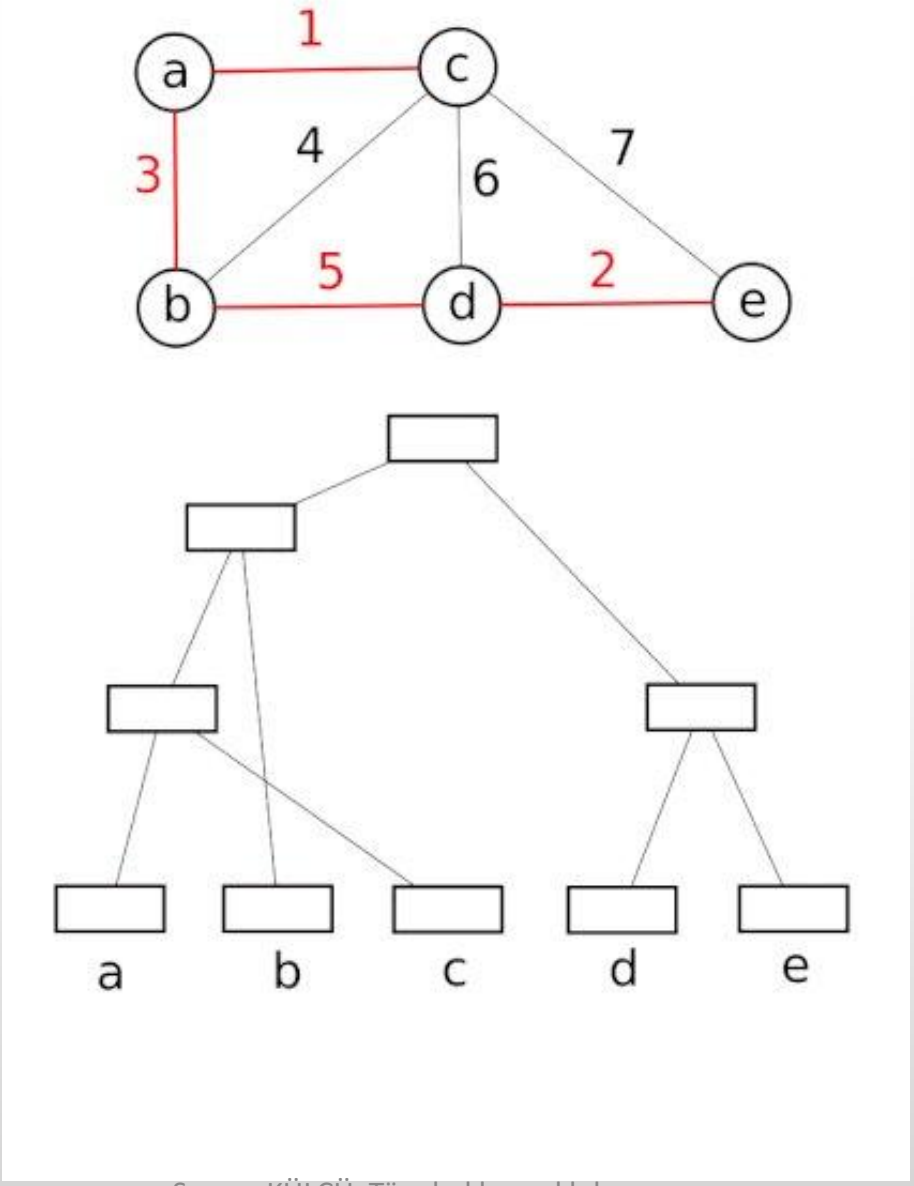


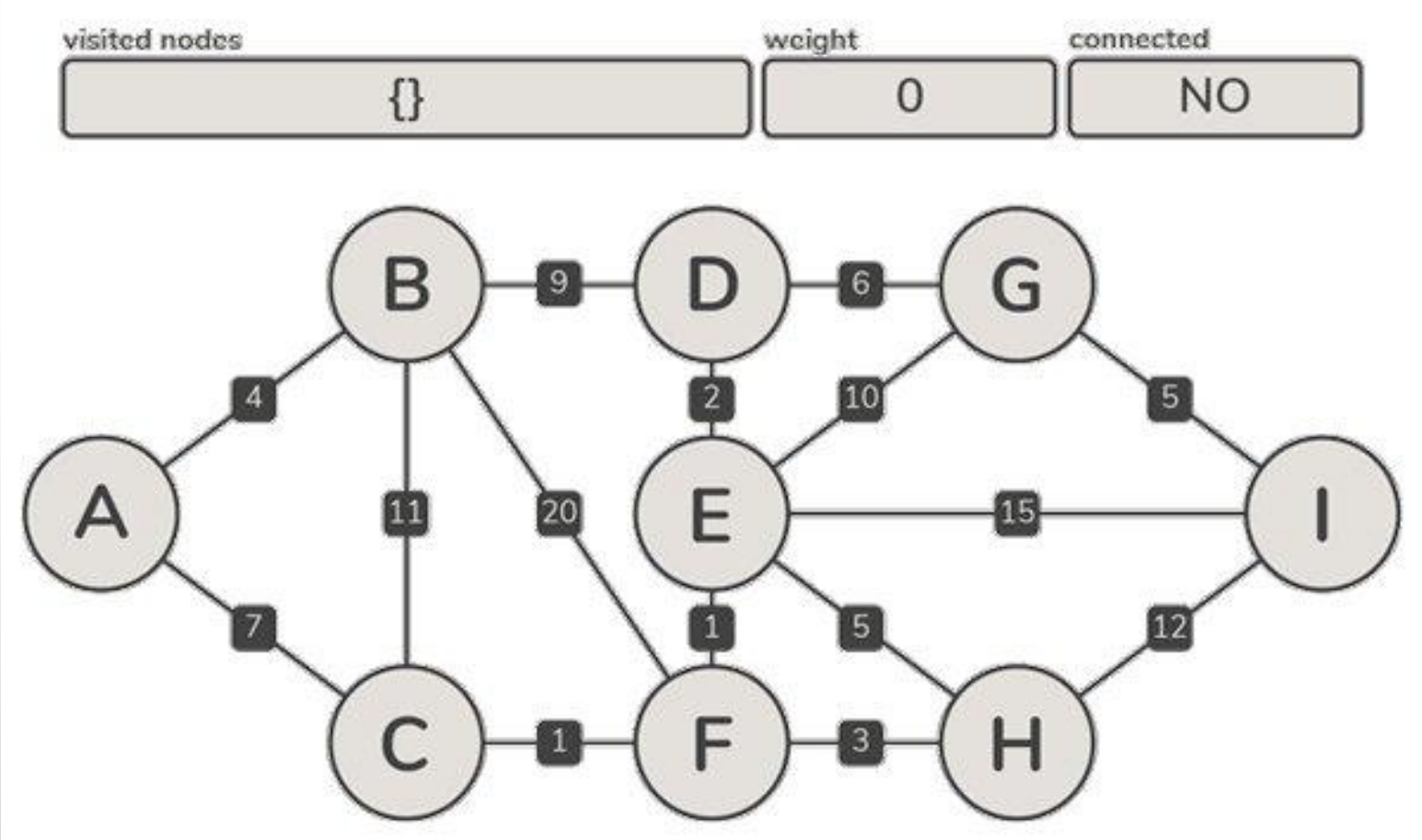
# Union Find



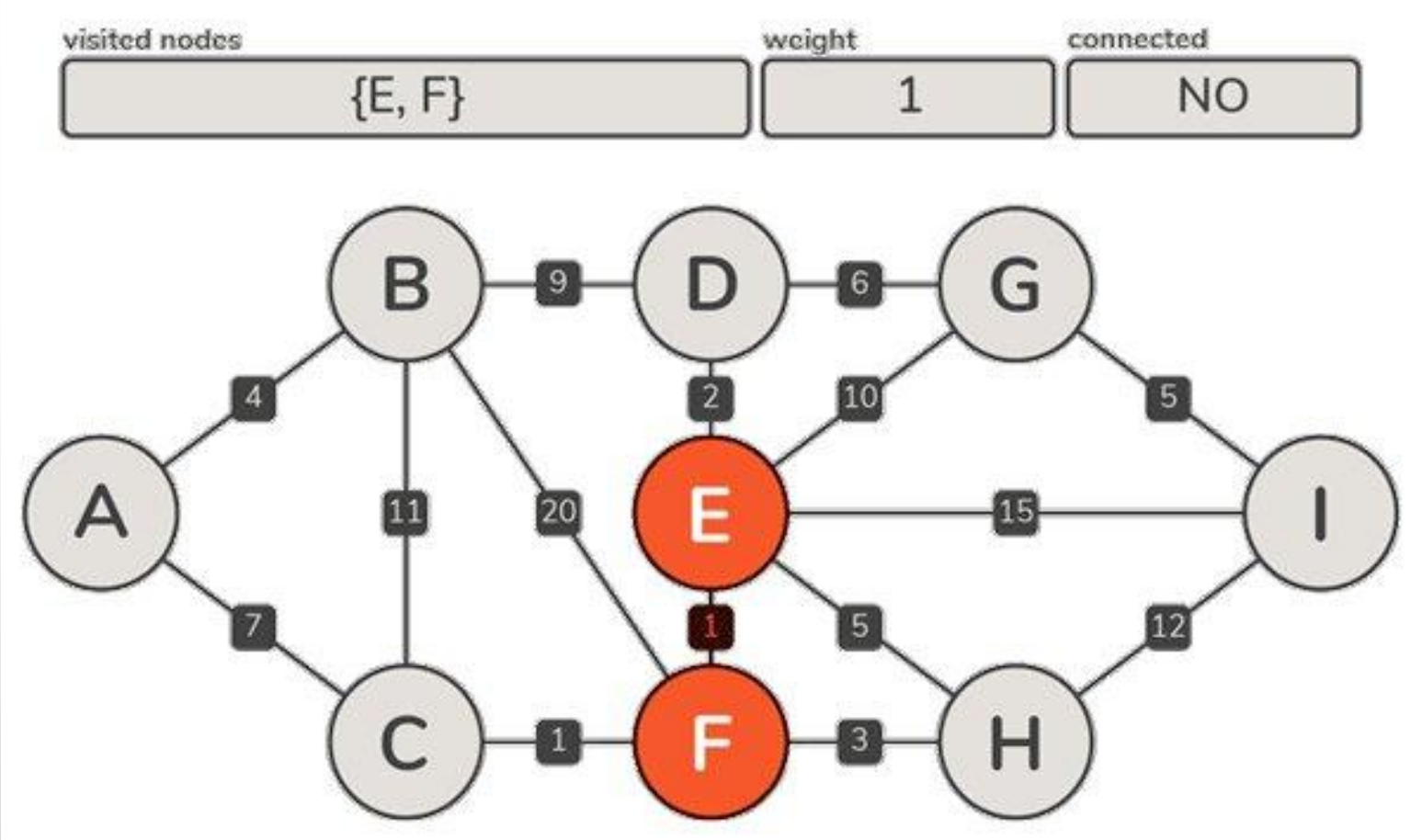

# Union Find



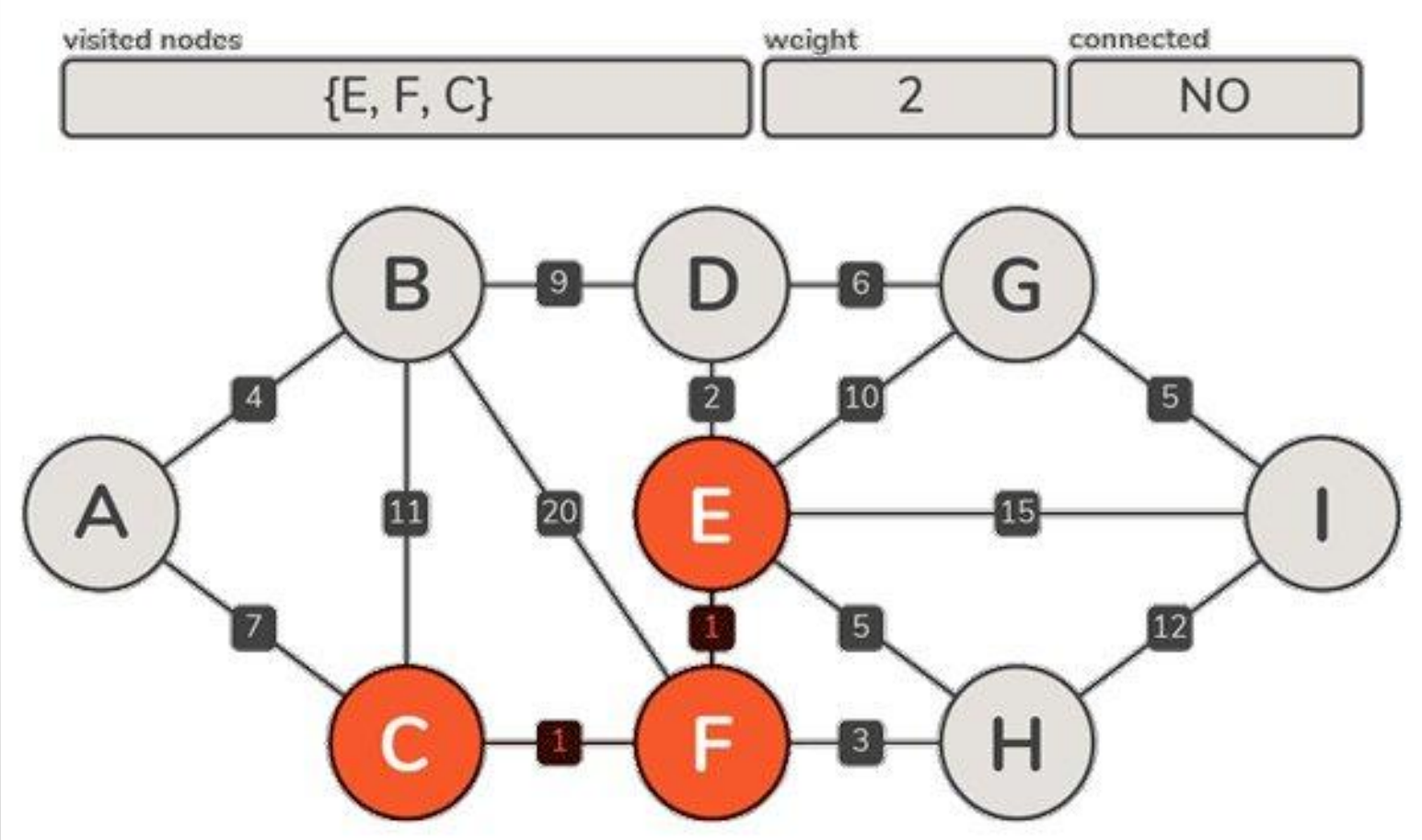
# Örnek Kruskal



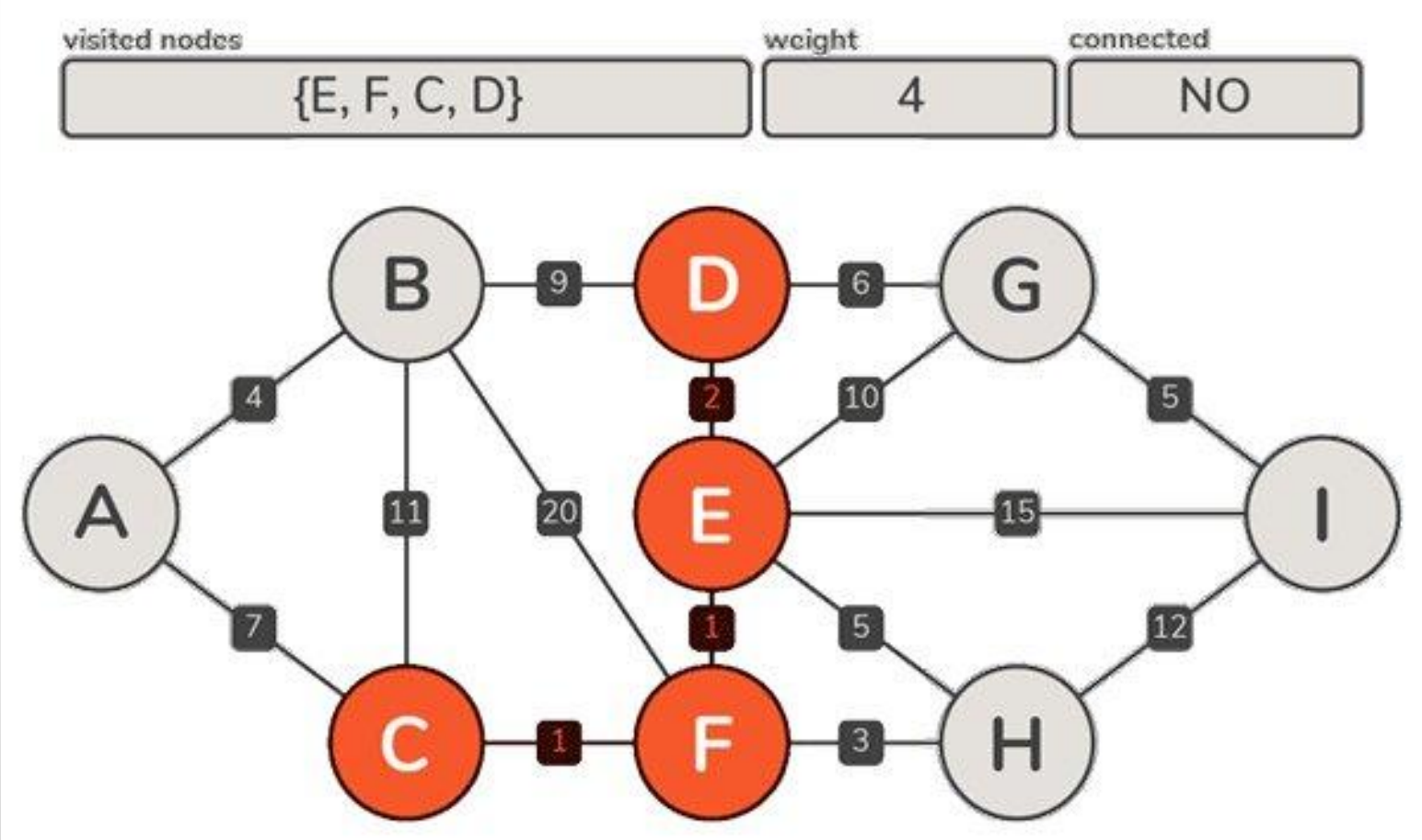
# Örnek Kruskal



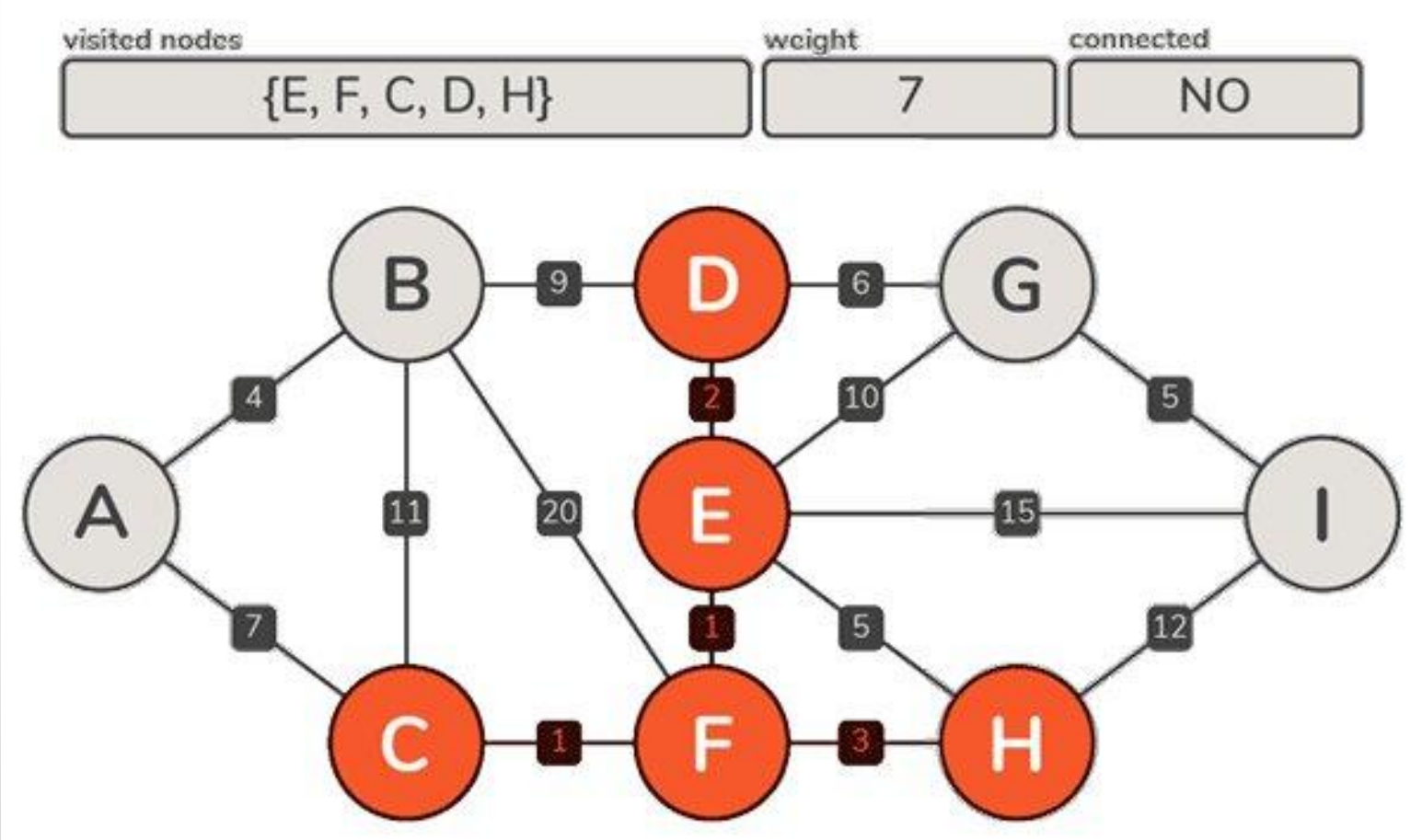
# Örnek Kruskal



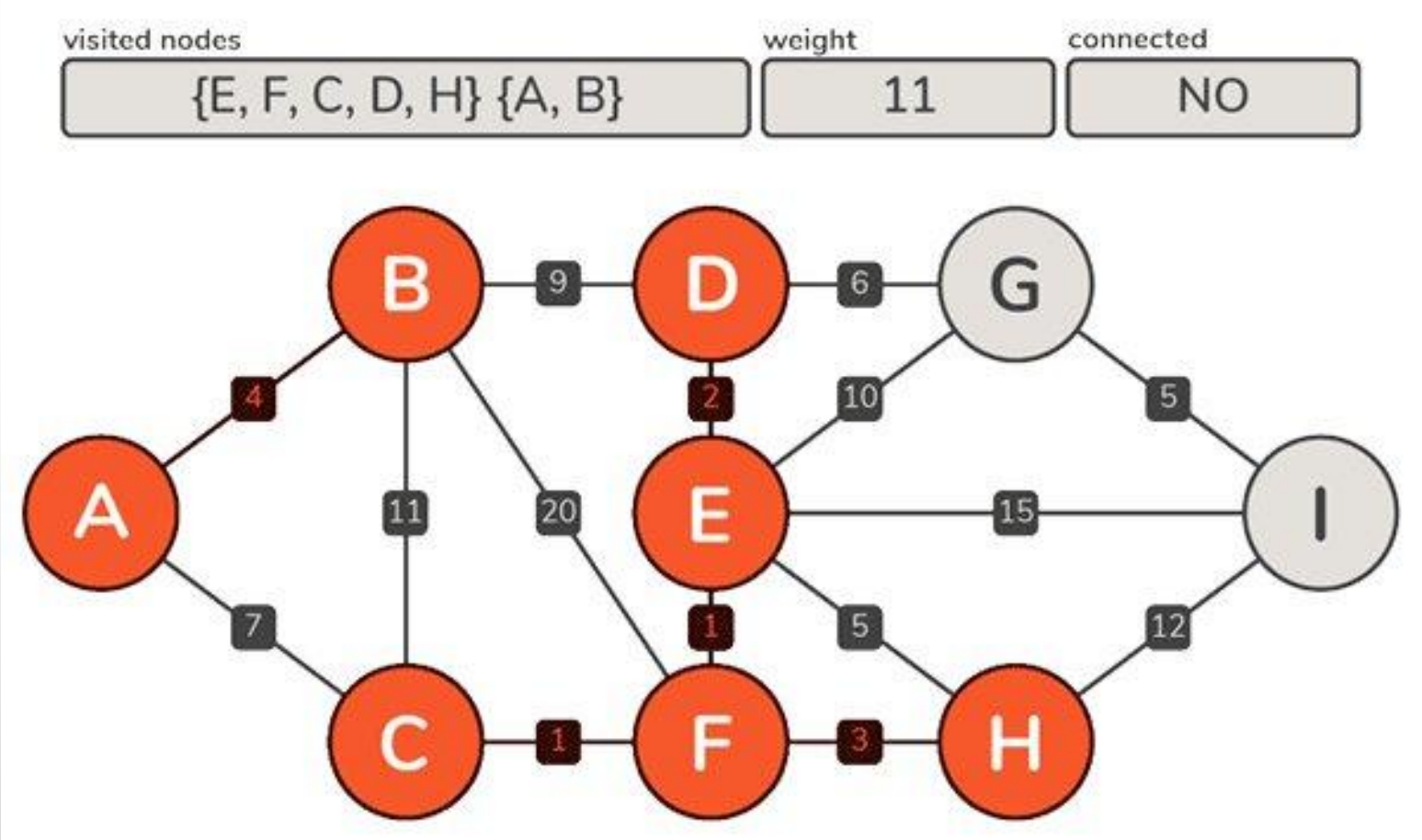
# Örnek Kruskal



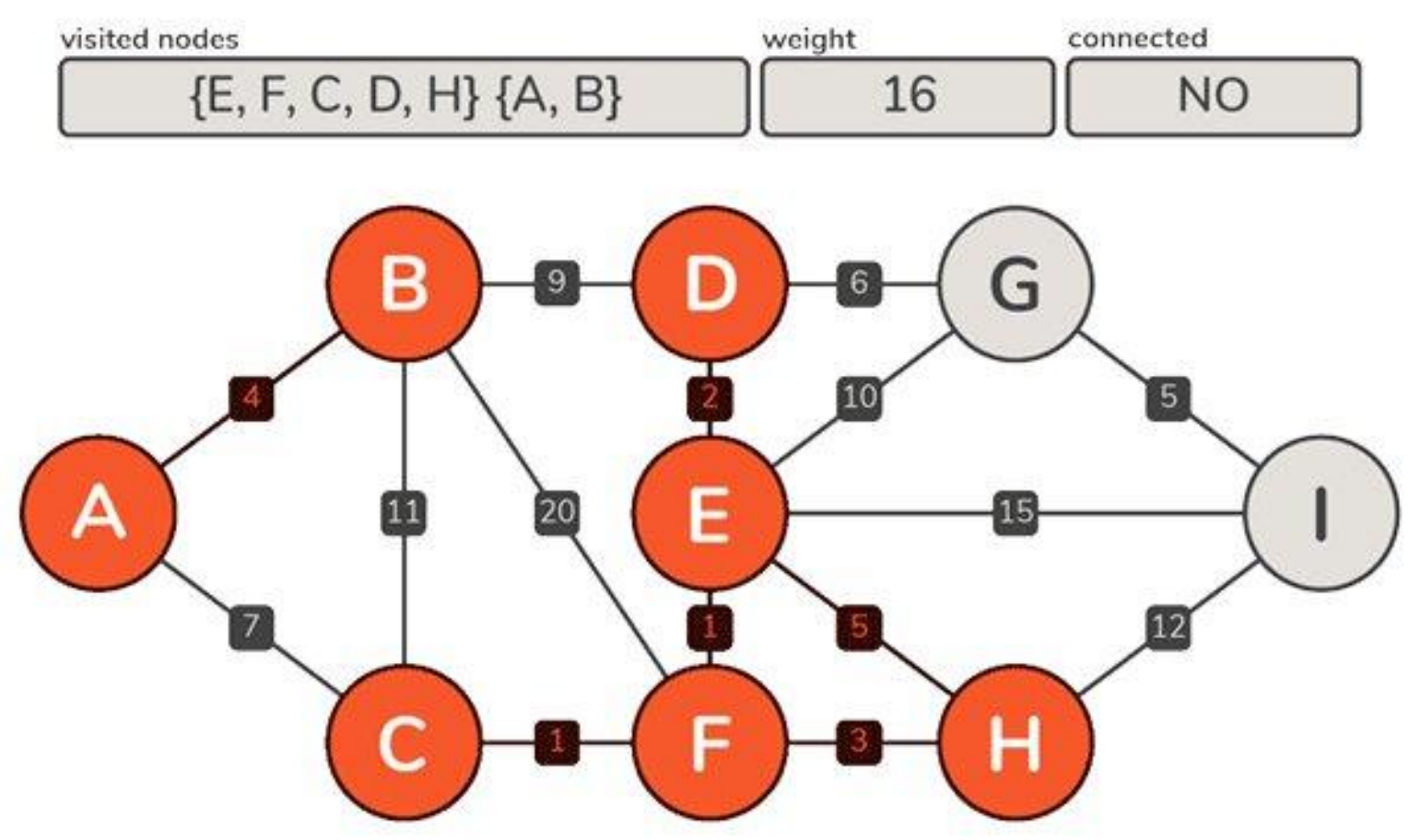
# Örnek Kruskal



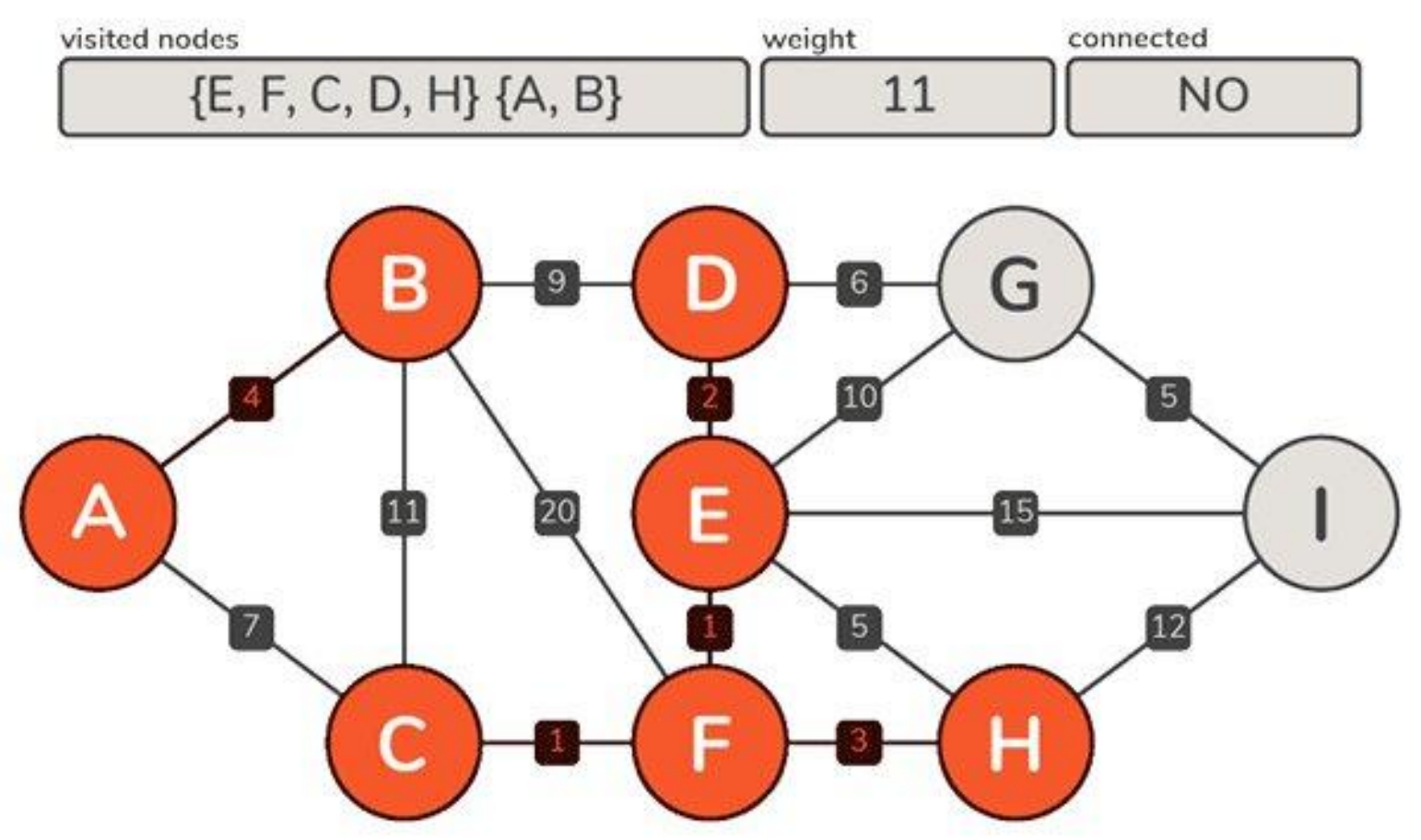
# Örnek Kruskal



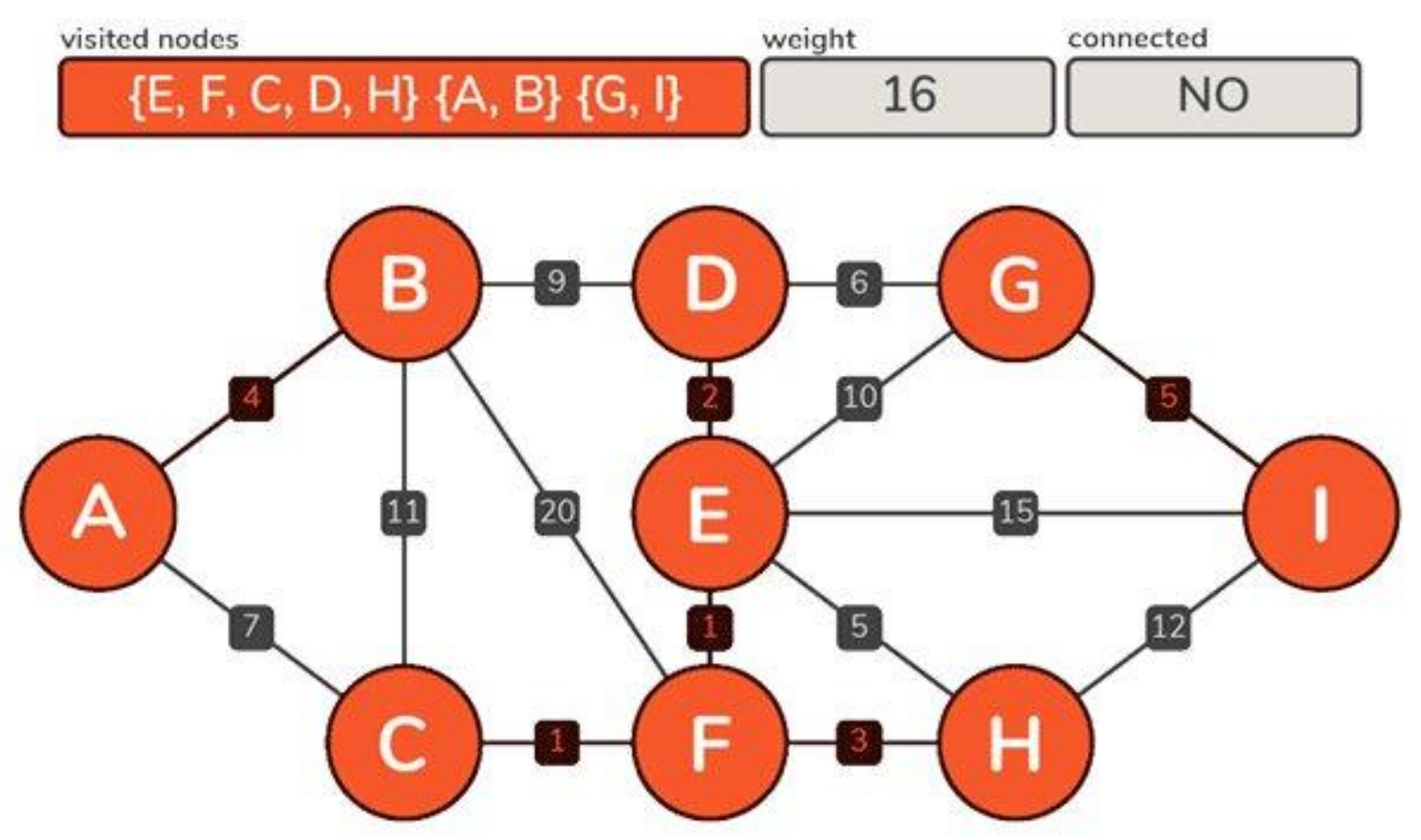
# Örnek Kruskal



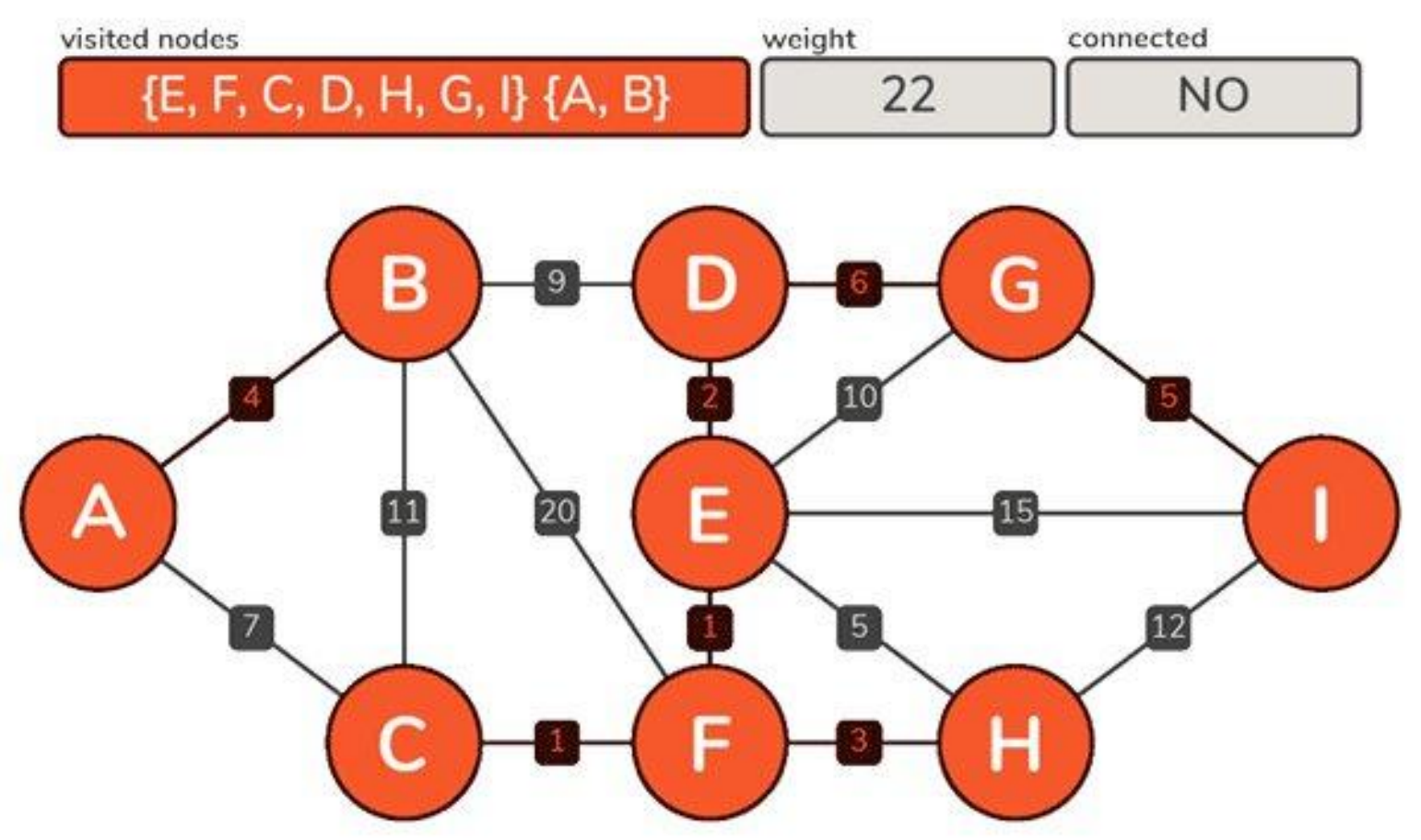
# Örnek Kruskal



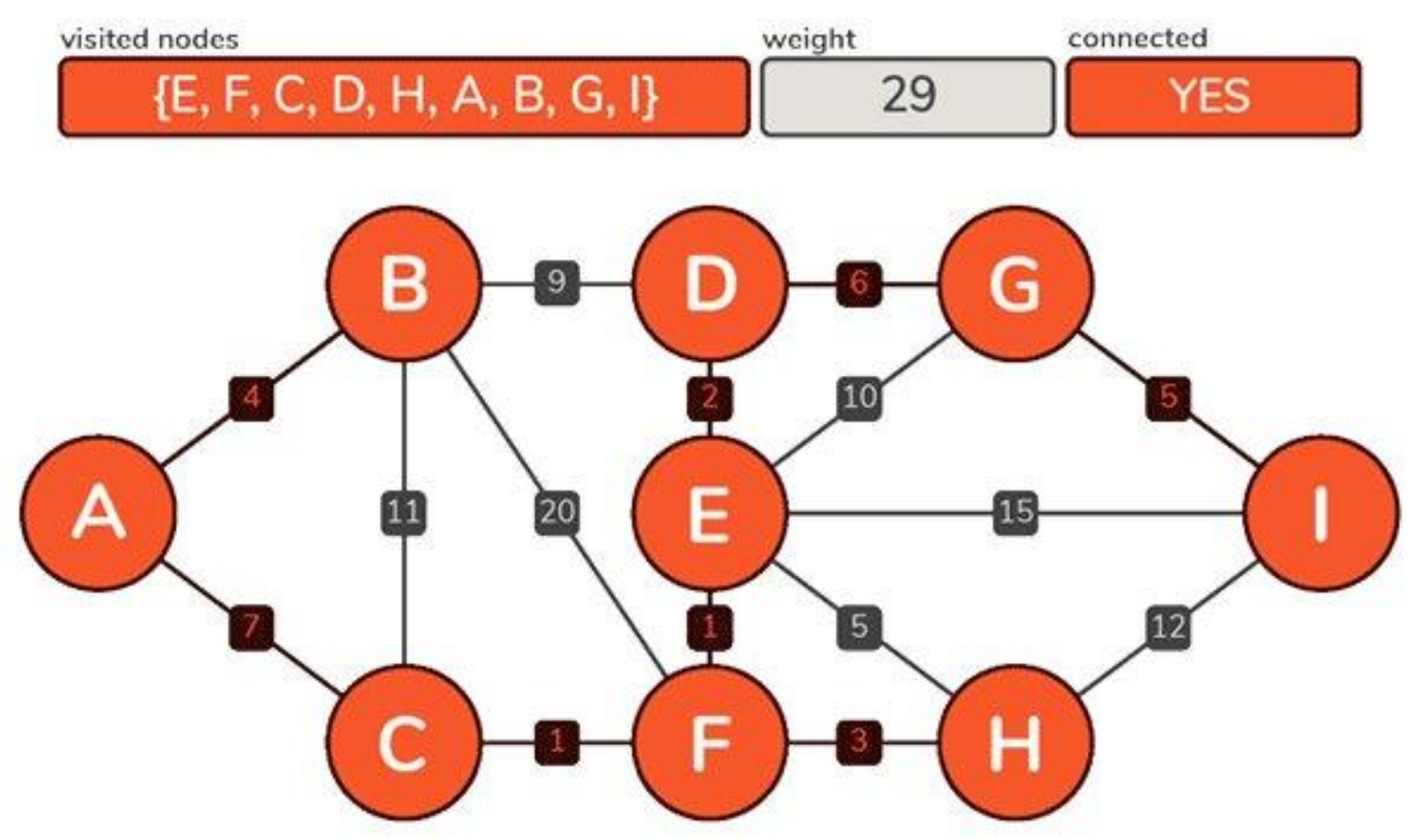
# Örnek Kruskal



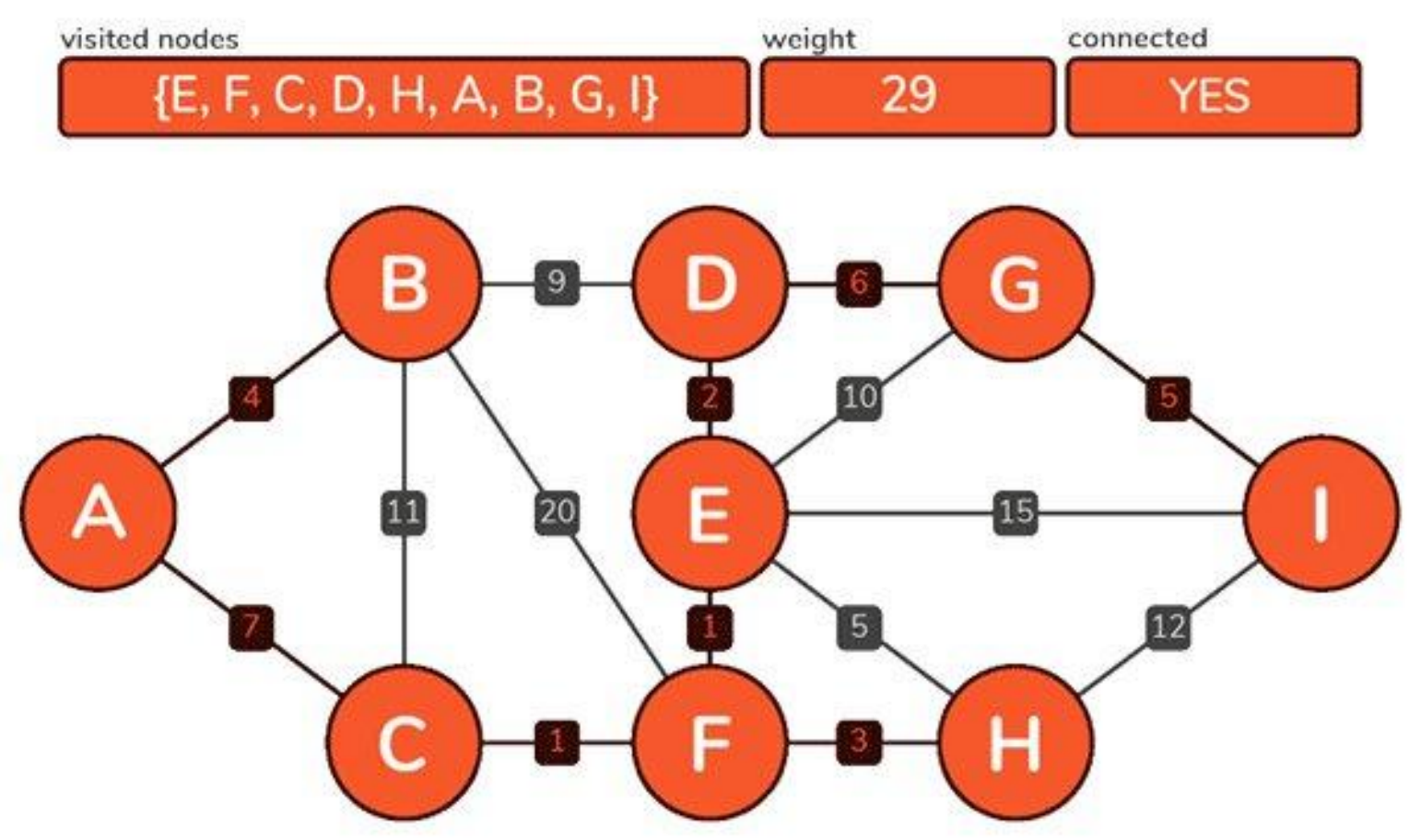
# Örnek Kruskal



# Örnek Kruskal



# Örnek Kruskal



# Prim Algoritması

- Tüm düğümleri bağlayan en küçük ağırlıklı ağacı oluşturmayı amaçlar.
- Açgözlü (*greedy*) bir algoritmadır.
- *Robert C. Prim* tarafından geliştirilmiştir.

# Algoritma İlkeleri

- Ağırlıklı çizge üzerinde çalışır.
- Tüm düğümleri en küçük ağırlıklı kenarlar ile birleştirir.
- Başlangıçta, bir düğüm seçilir ve ağacın başlangıç düğümü kabul edilir.
- Her adımda,
  - ağaç içinde olmayan düğümler arasından bir tanesi,
  - en küçük ağırlıklı kenar ile ağaca eklenir.

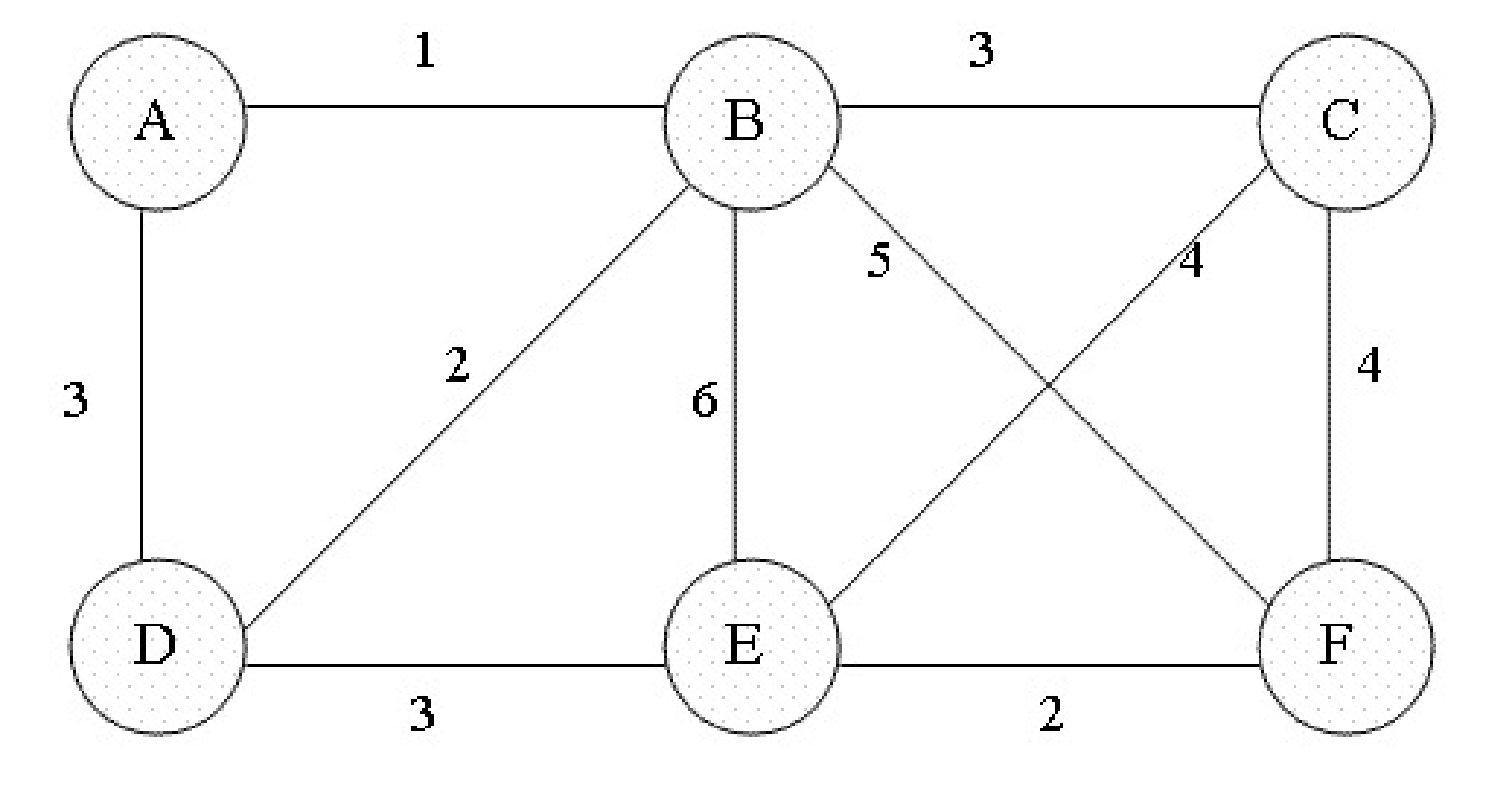
# Algoritma Adımları

- Adım 1: Başlangıç düğümü seçilir ve bu düğüme ait olan tüm kenarlar *öncelik kuyruğuna* eklenir.
- Adım 2: Kuyruktan, ağaçta olmayan en küçük ağırlıklı kenar seçilir.
- Adım 3: Seçilen kenar ile bağlantılı olan yeni düğüm ağaca eklenir.
- Adım 4: Yeni eklenen düğüme bağlı kenarlar *öncelik kuyruğuna* eklenir.

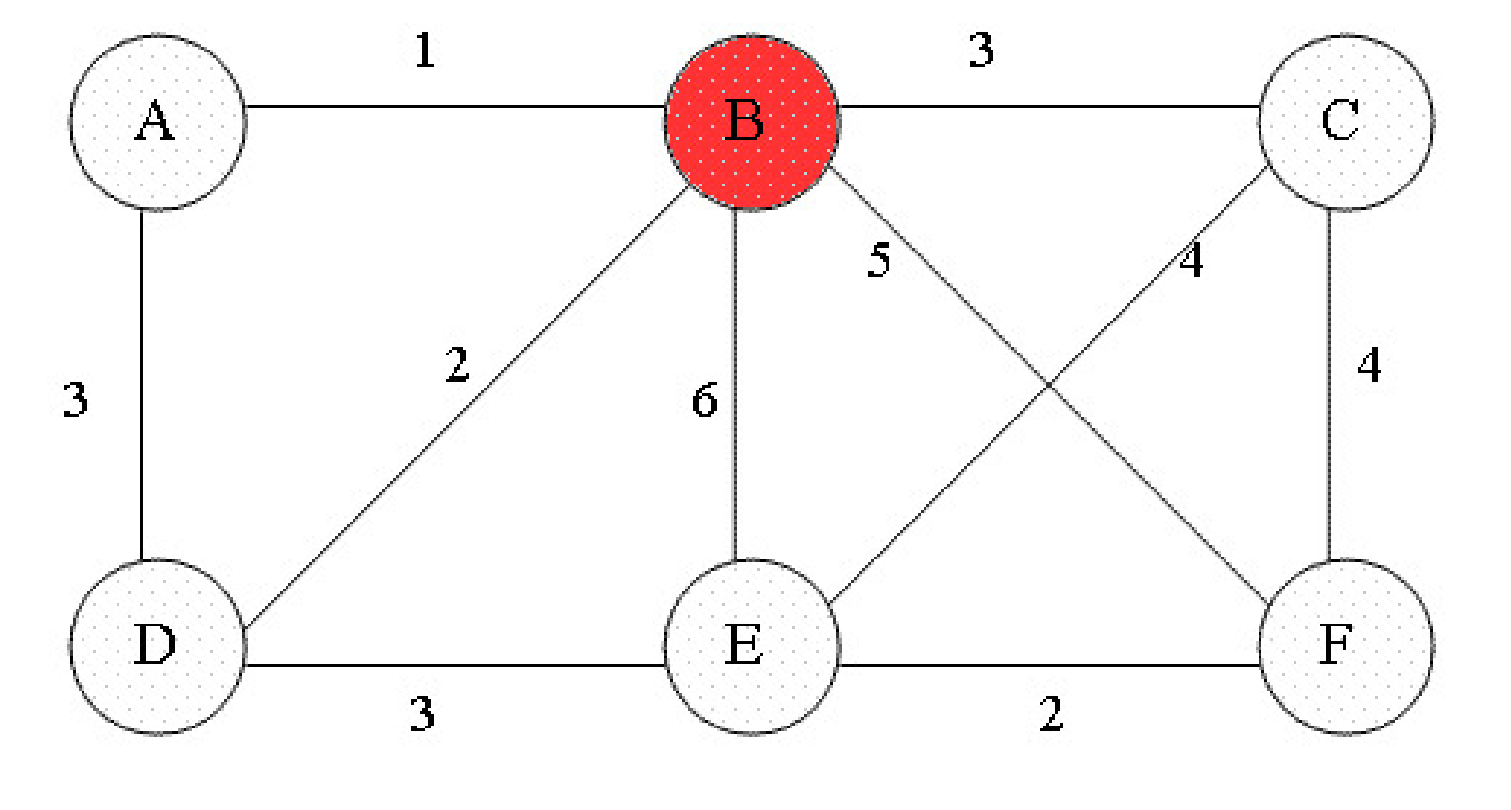
# Karmaşıklık Analizi

- Prim Algoritması'nın karmaşıklığı
  - *O((E + V) log V)* (*öncelik kuyruğu kullanılarak*)
    - E kenar sayısını,
    - V düğüm sayısını temsil eder.

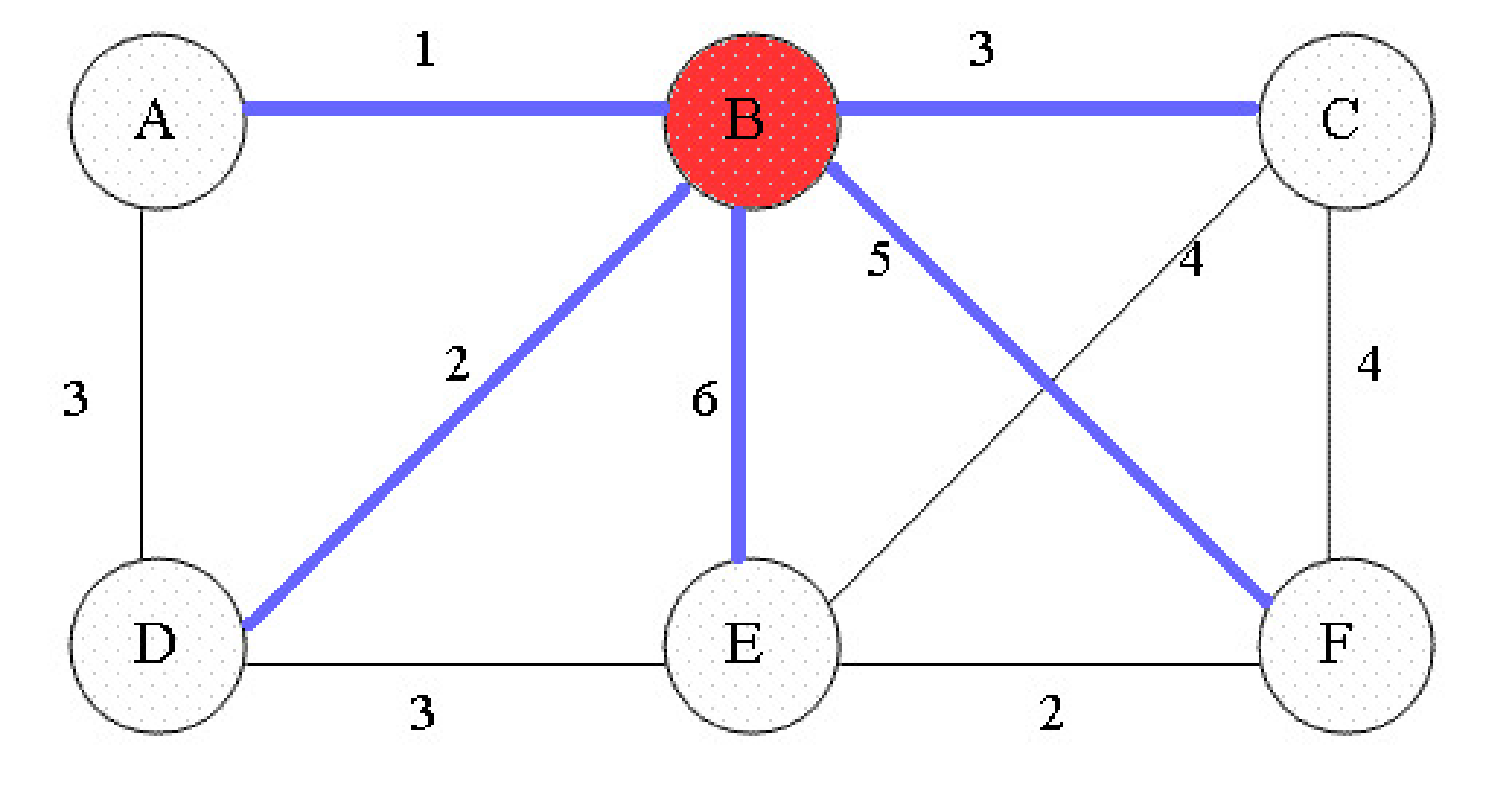
# Prim



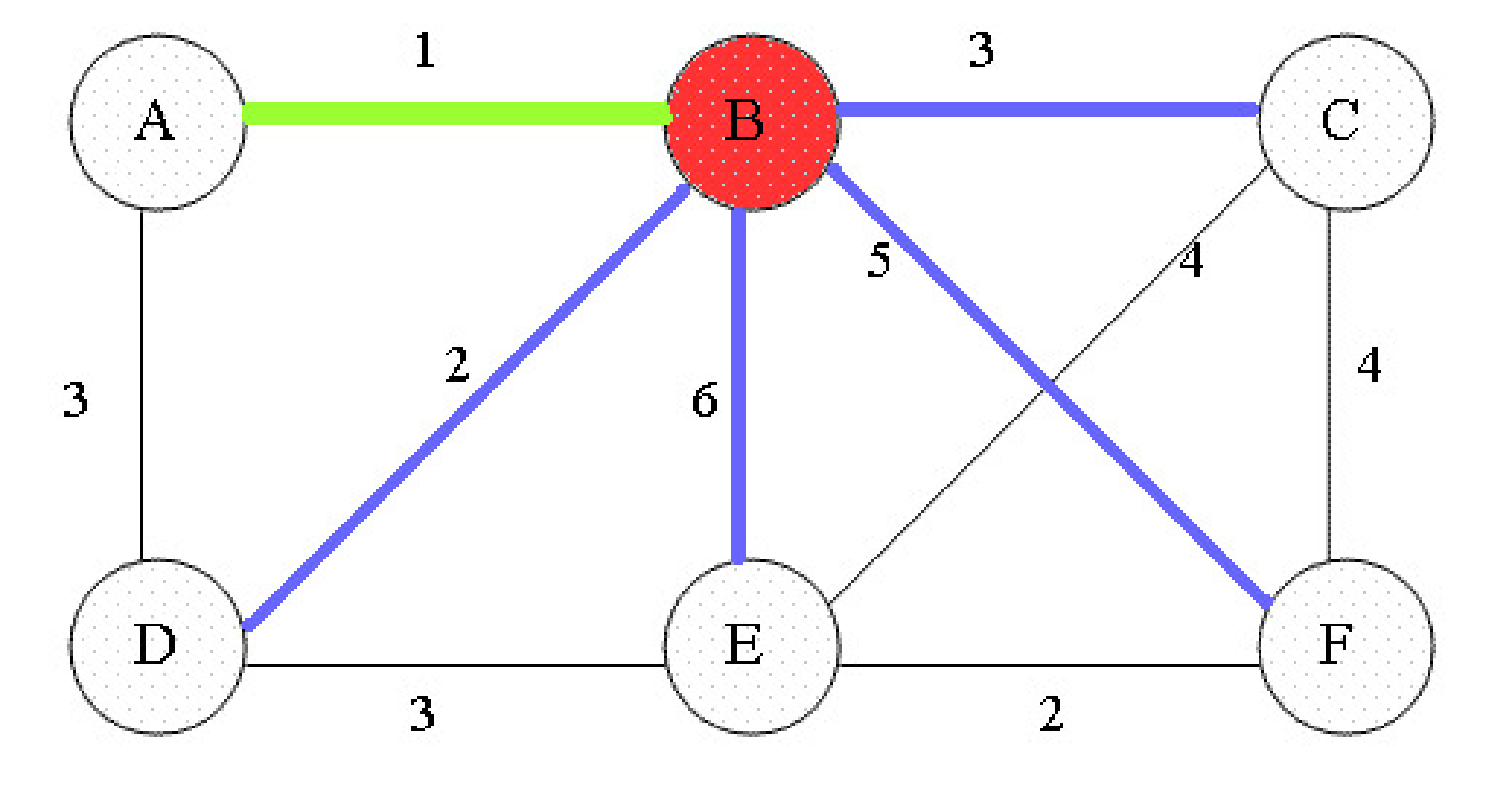
# Prim



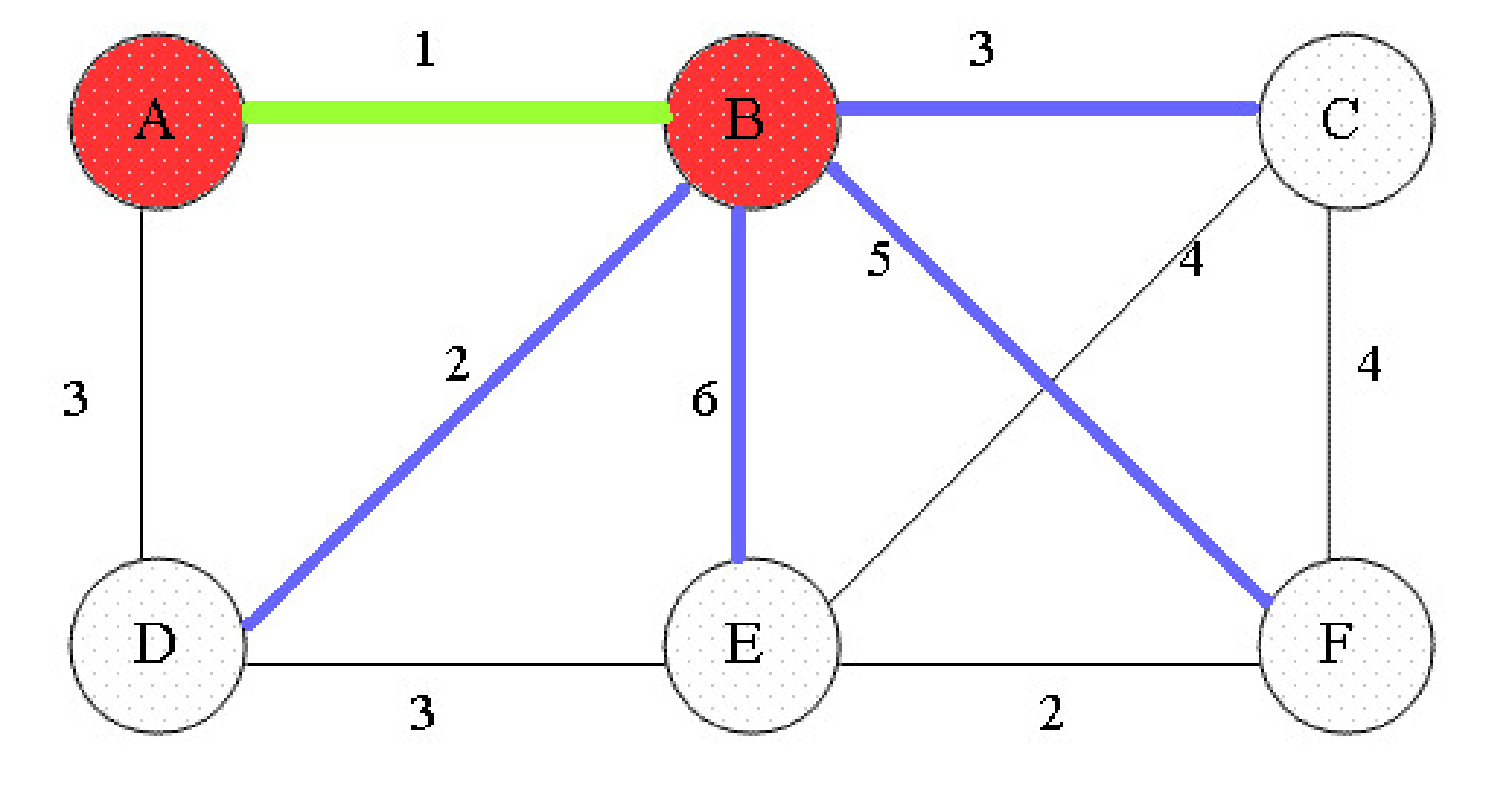
# Prim



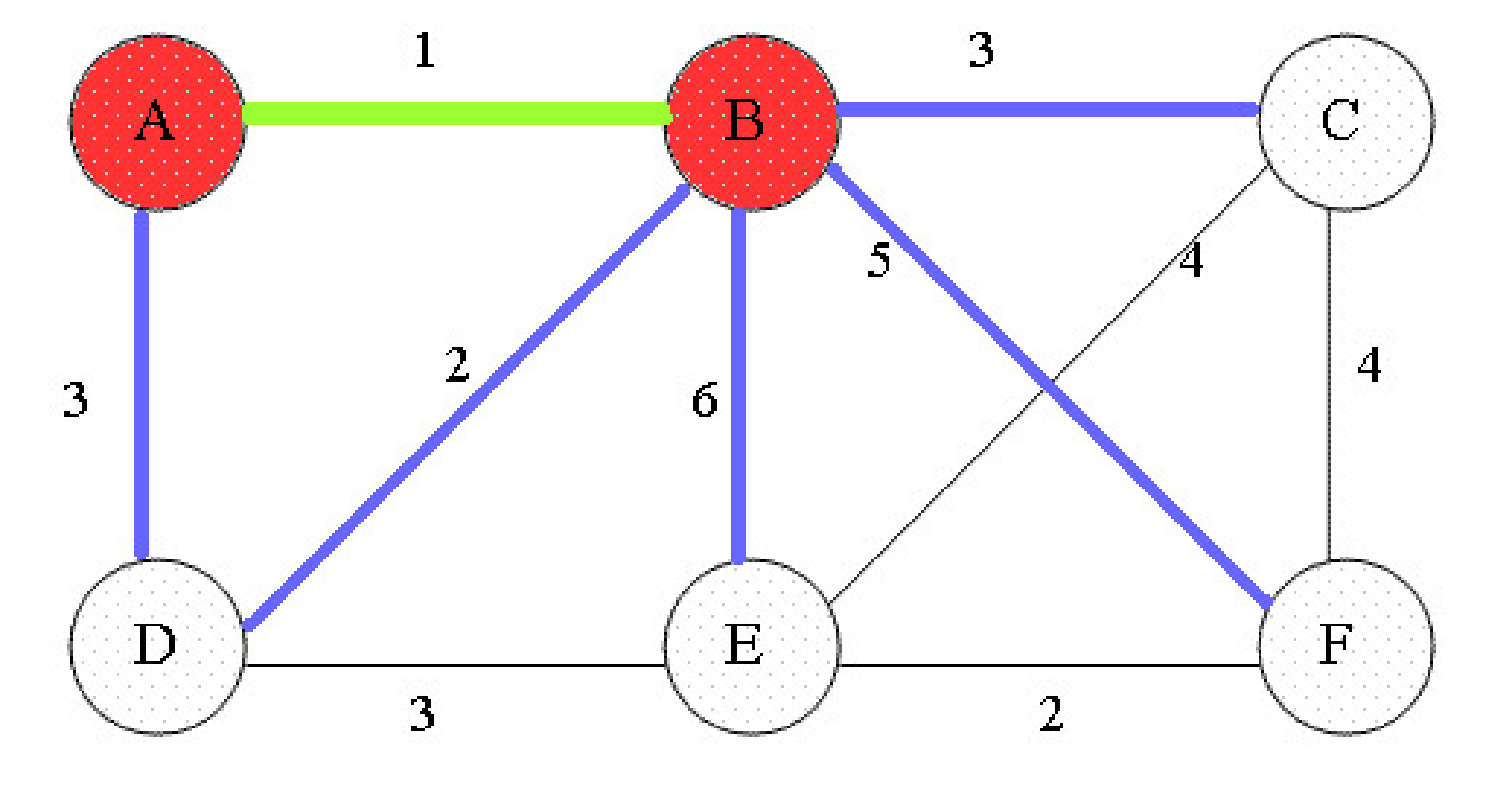
# Prim



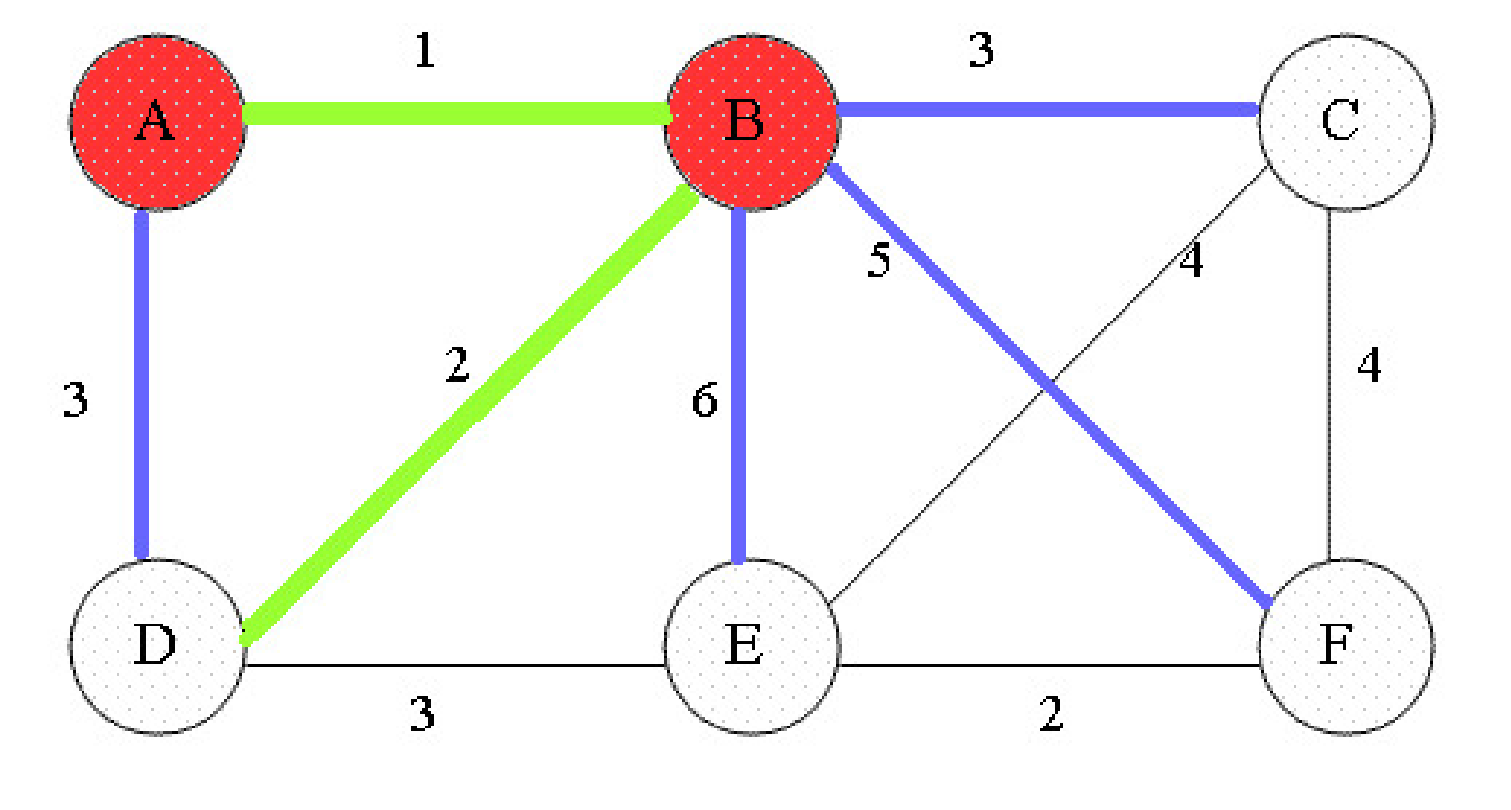
# Prim



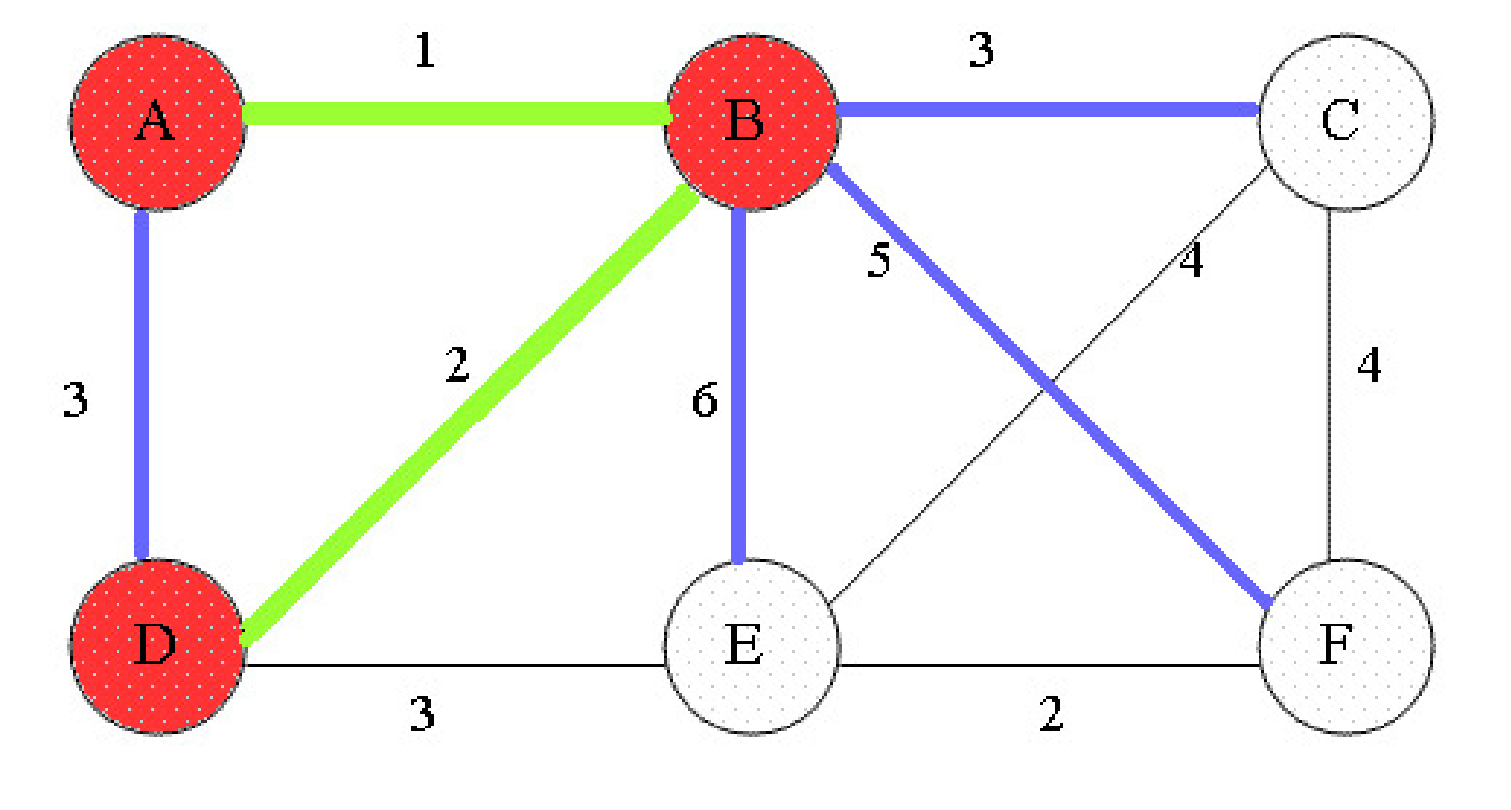
# Prim



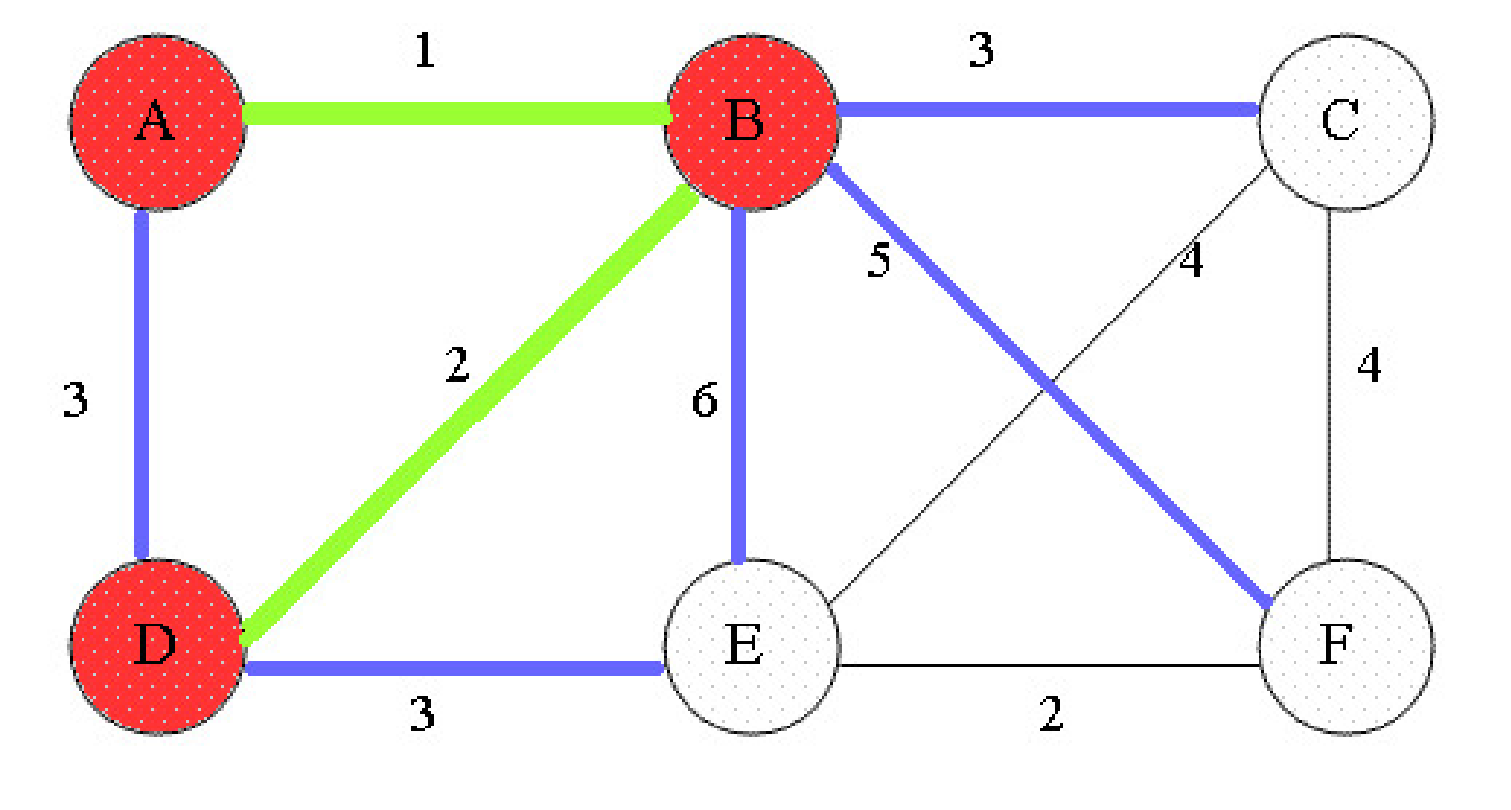
# Prim



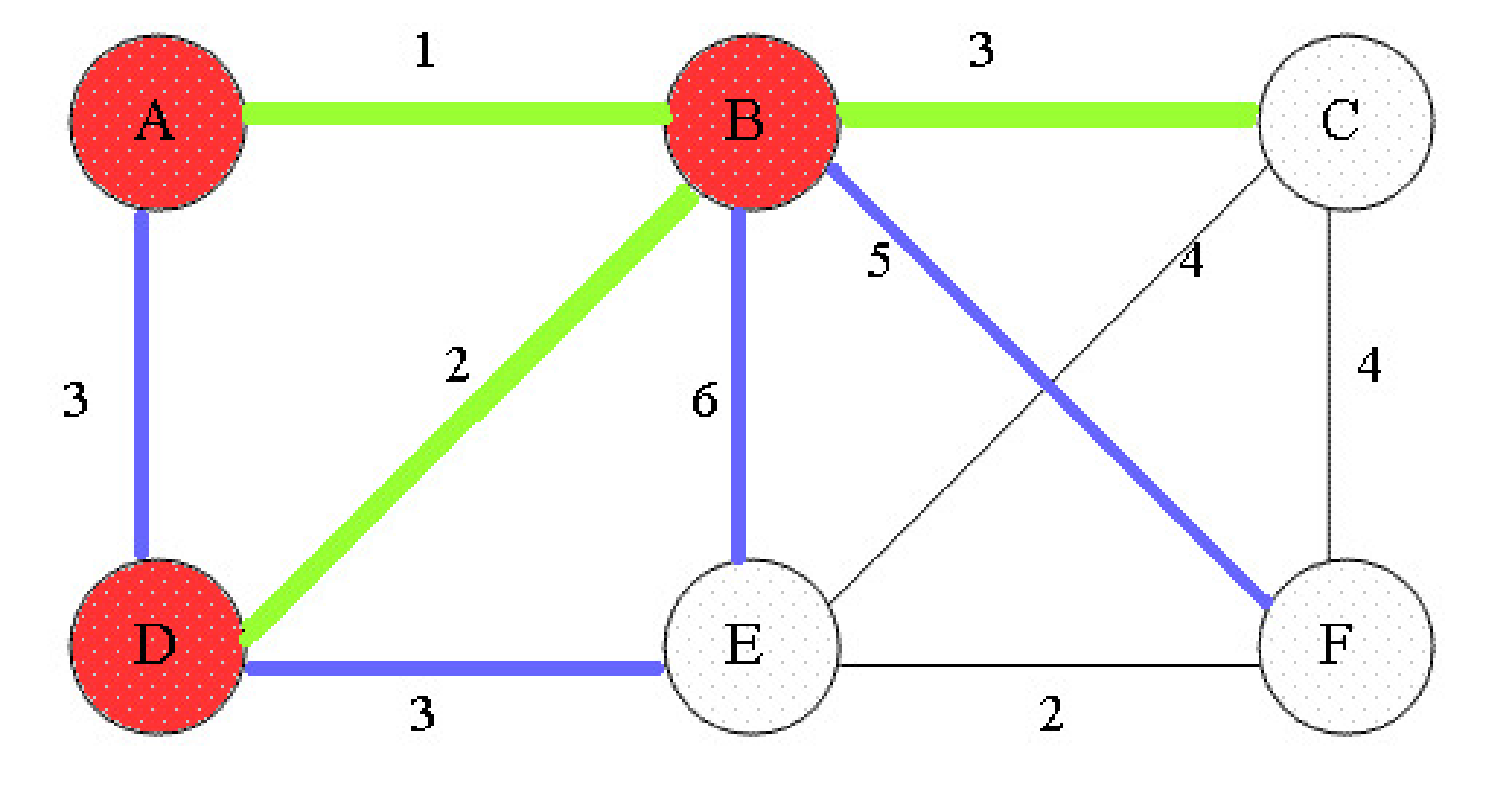
# Prim



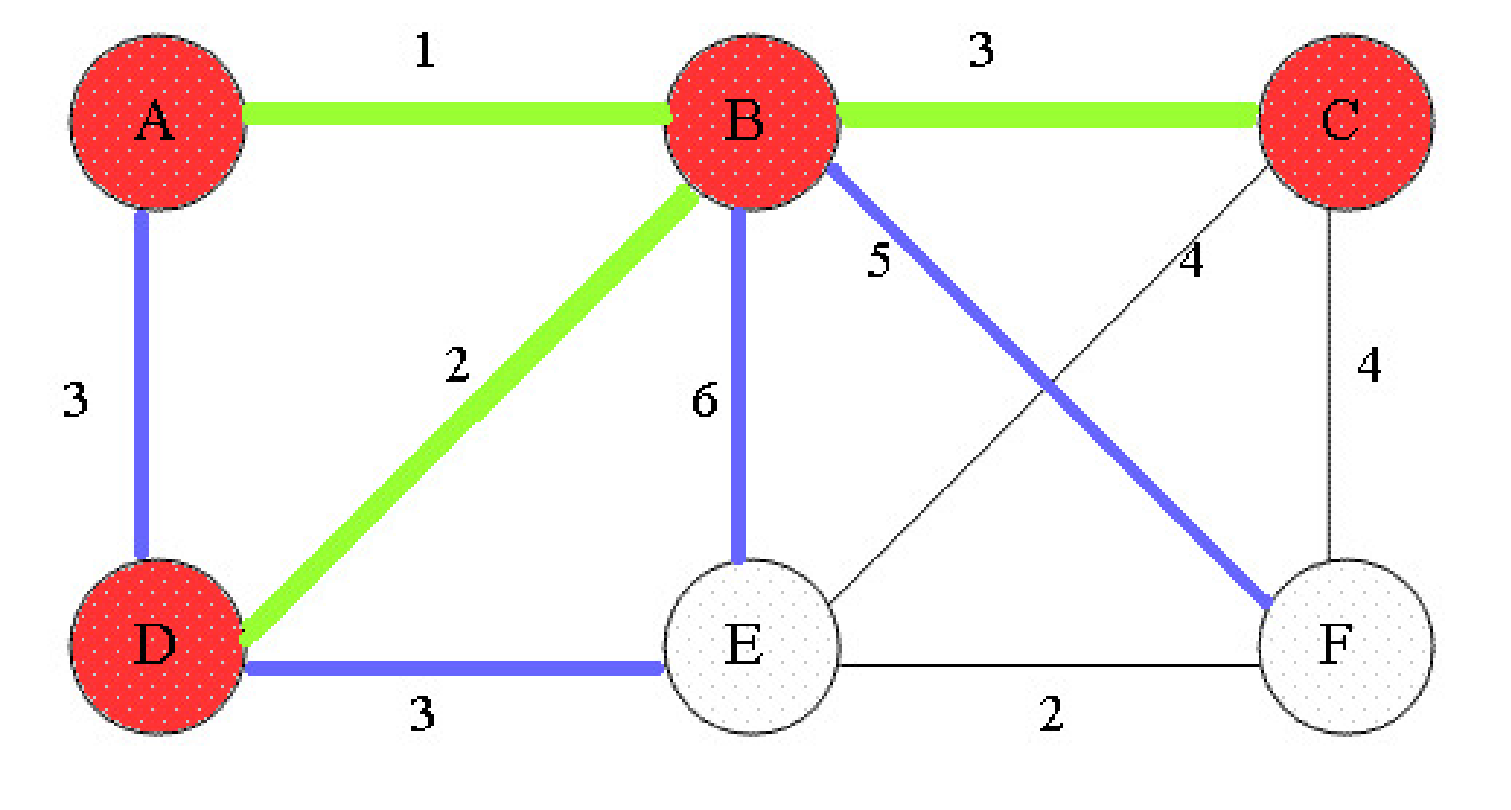
# Prim



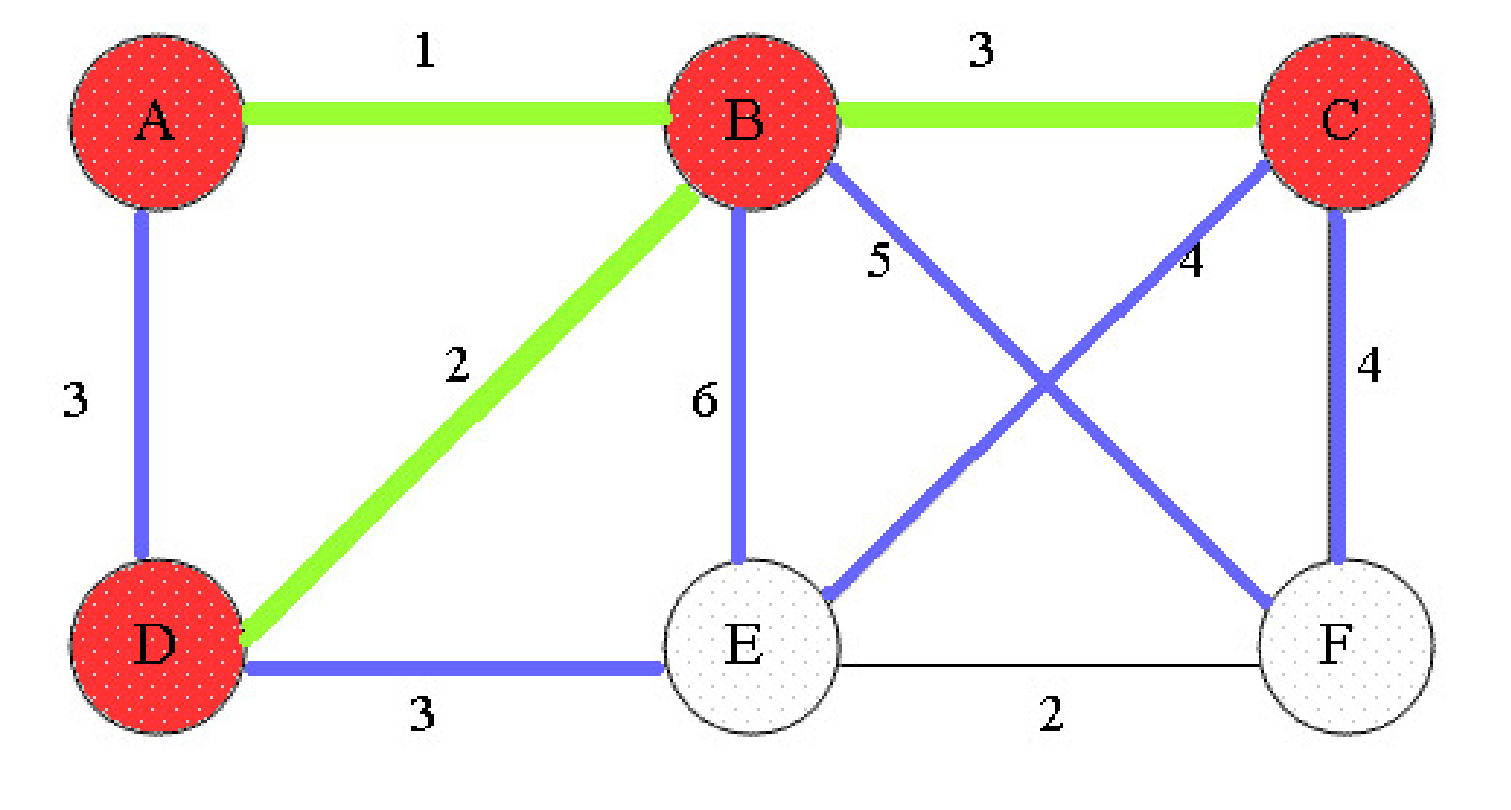
# Prim



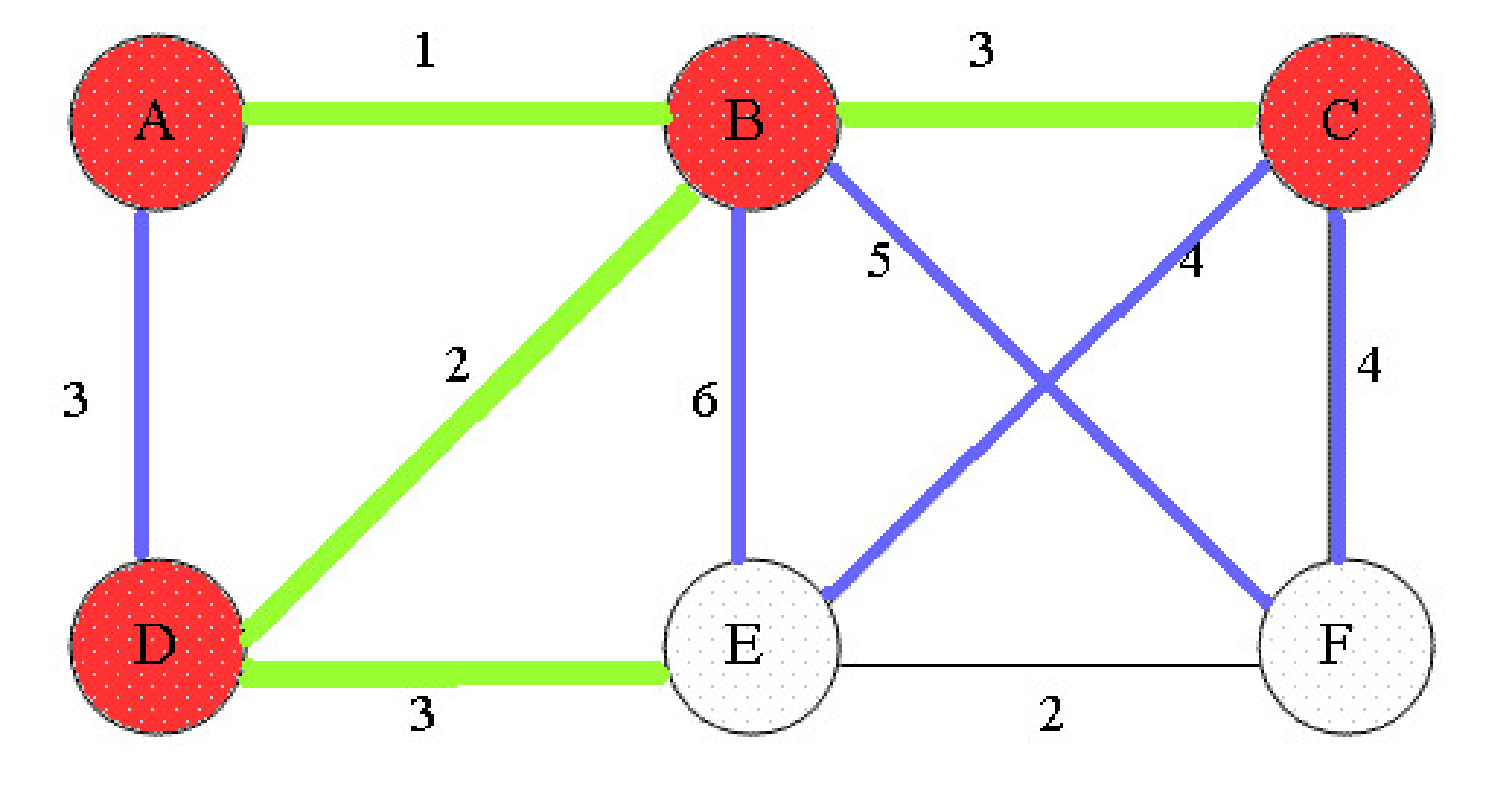
# Prim



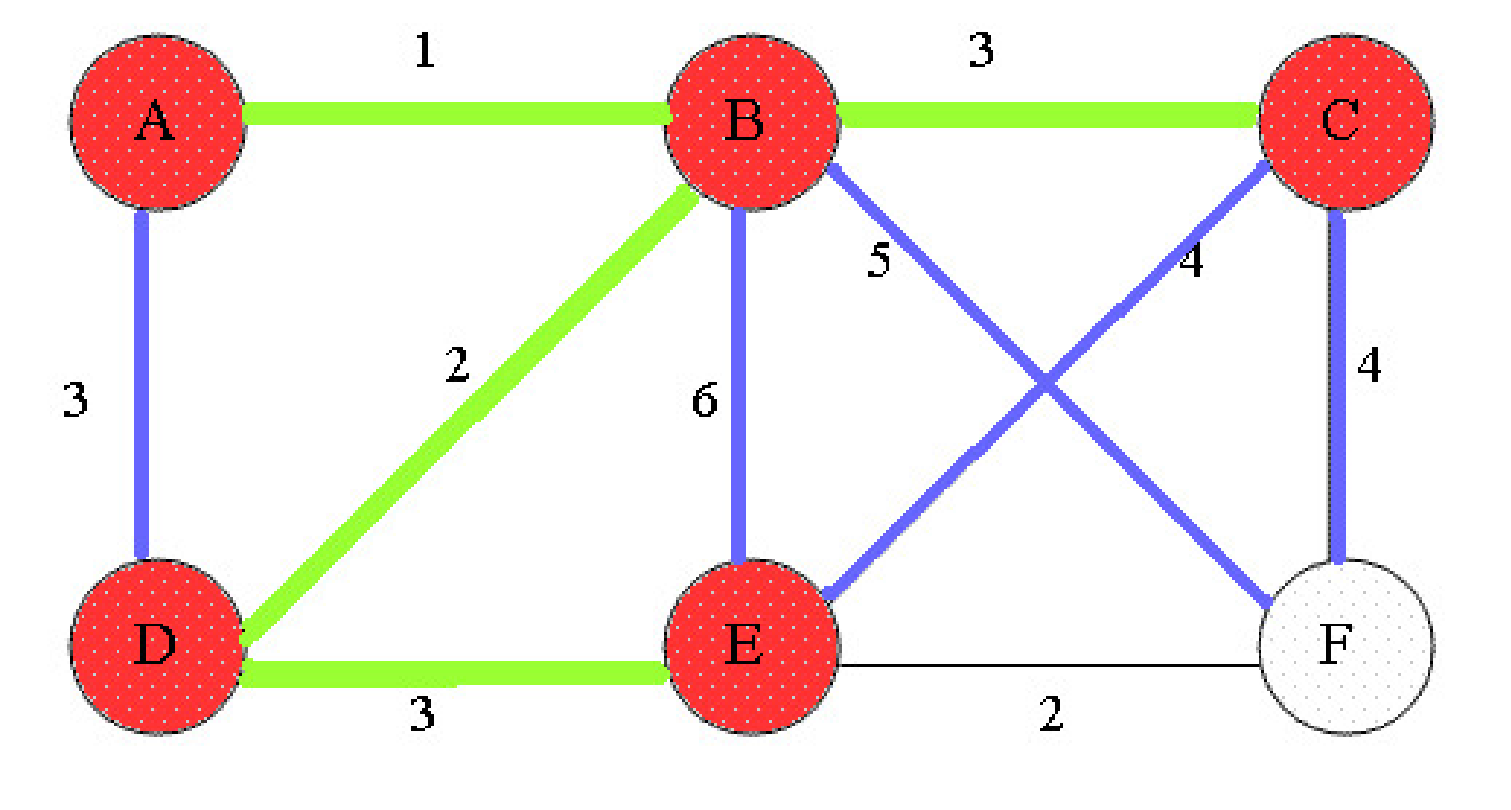
# Prim



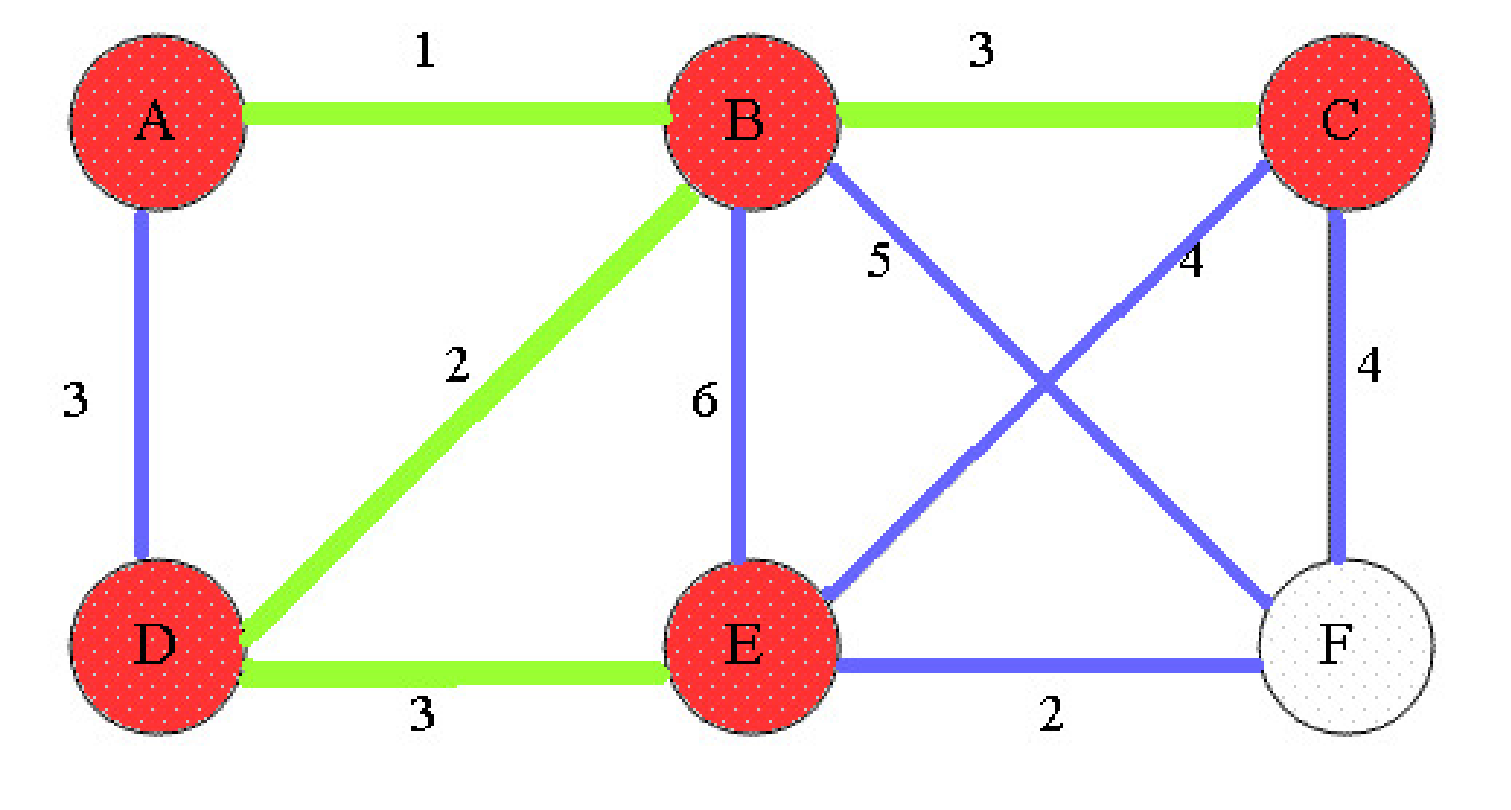
# Prim



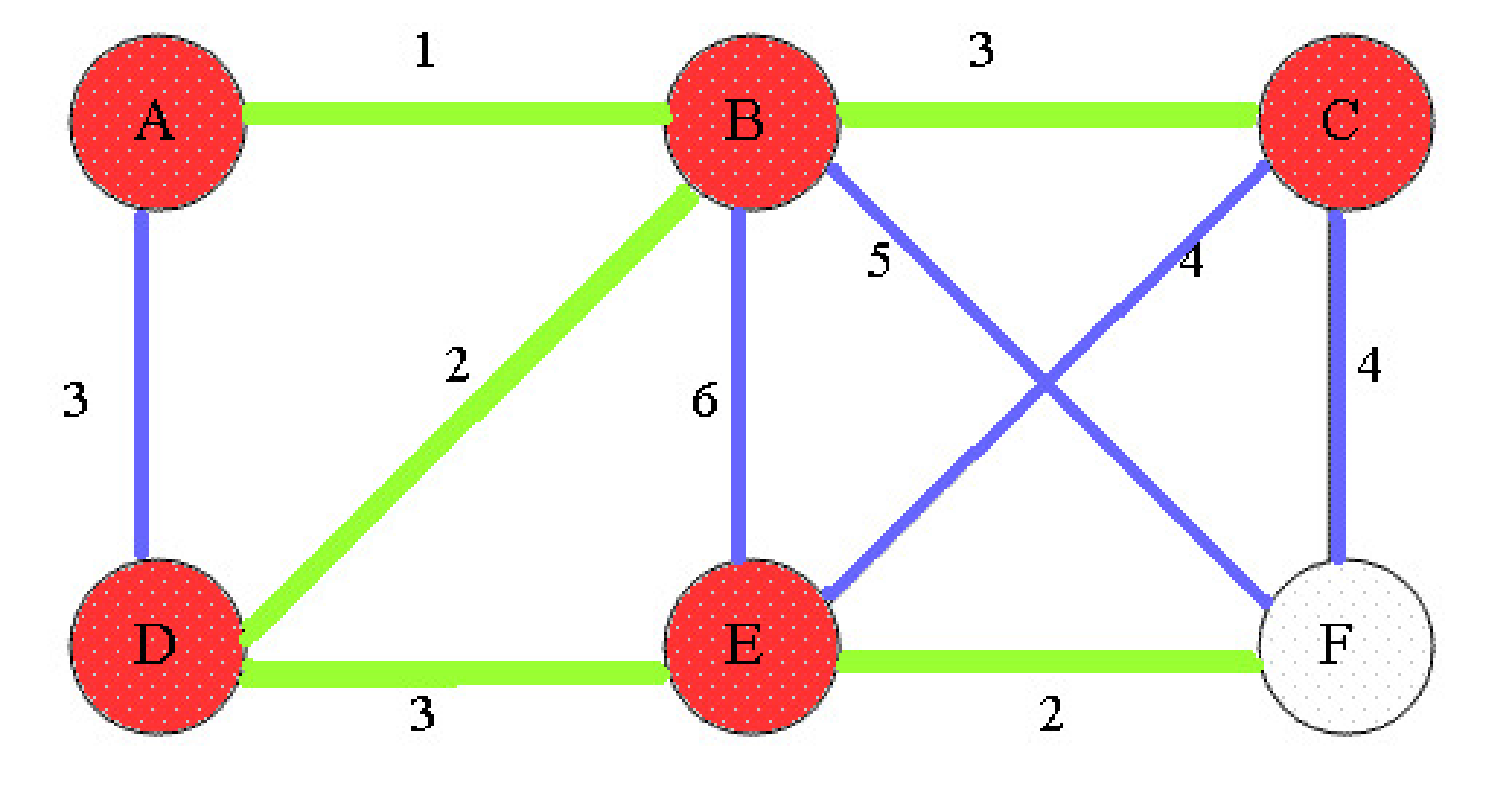
# Prim



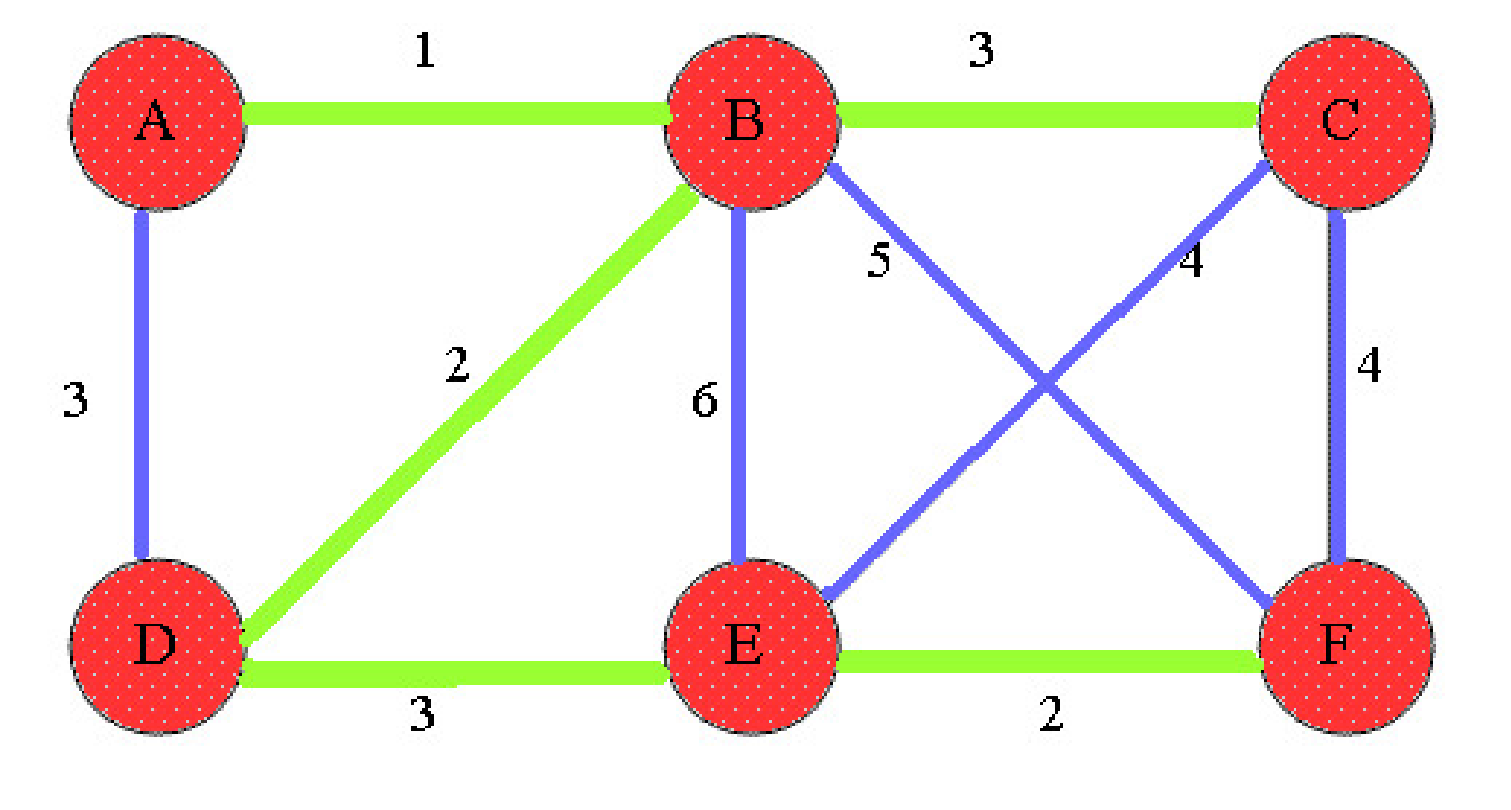
# Prim



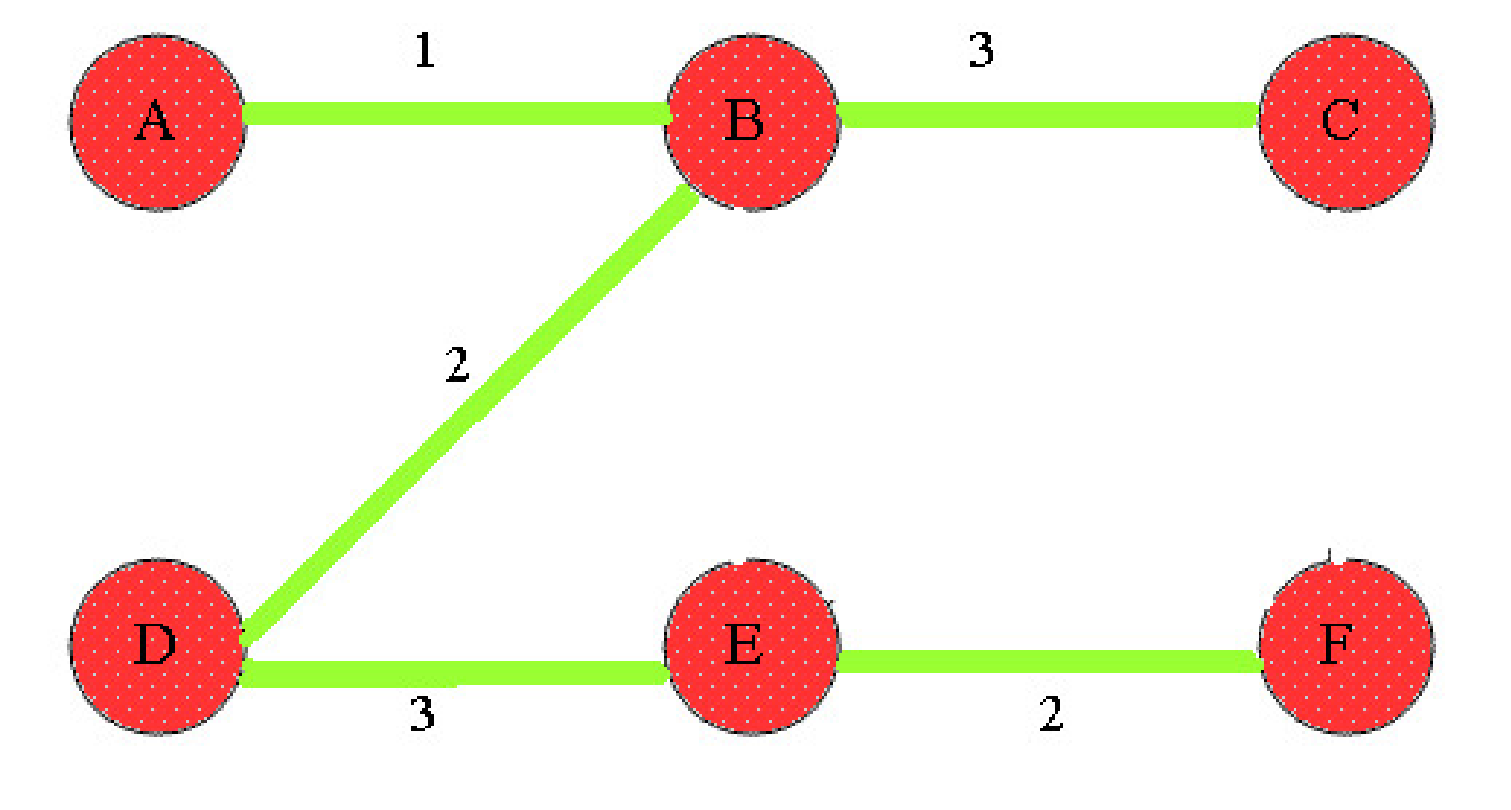
# Prim



# Prim

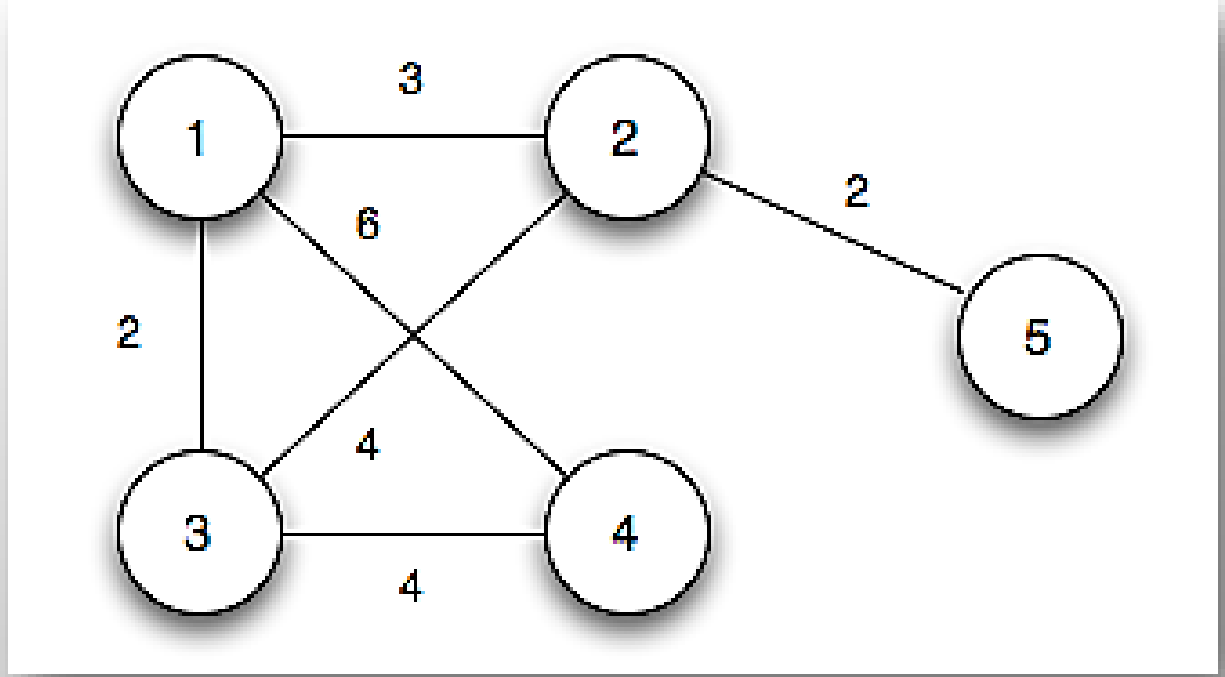


# Prim



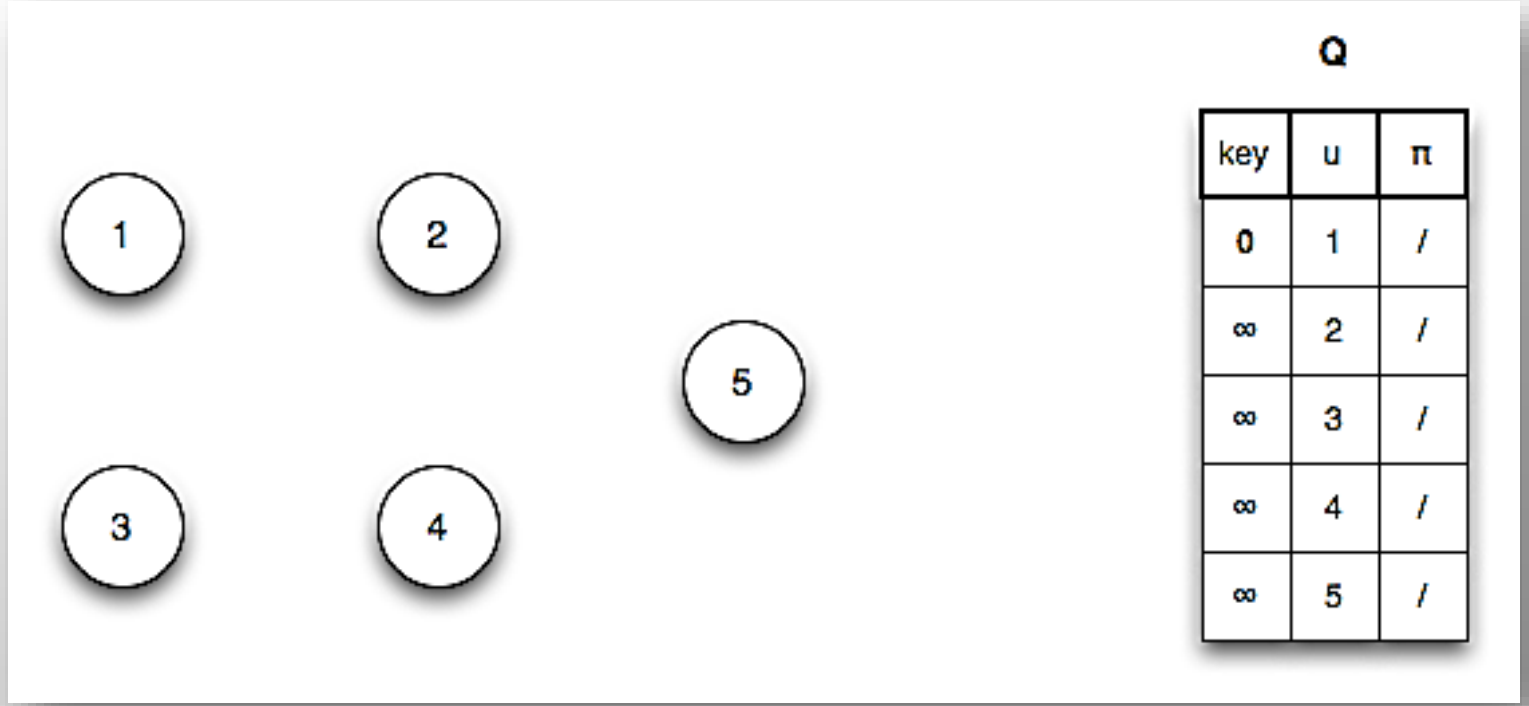
# Örnek

- Aşağıdaki yönsüz çizge verilsin.



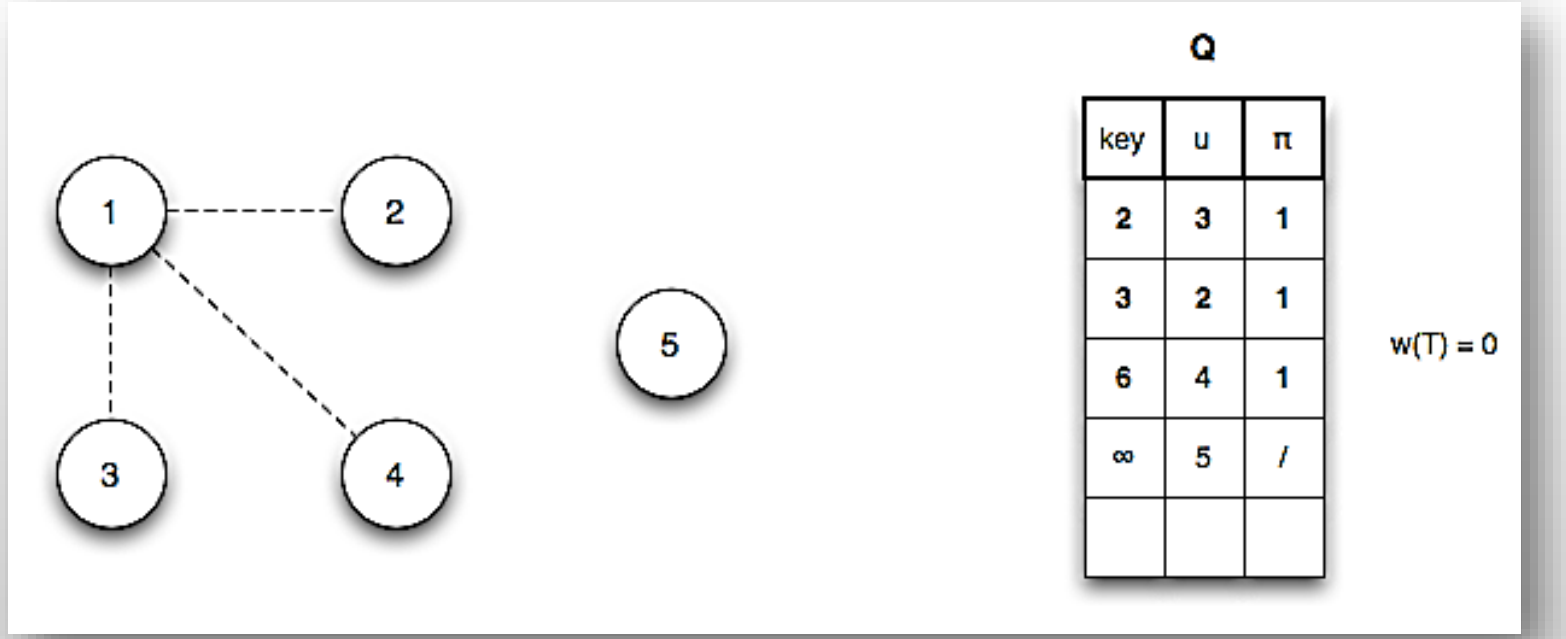
# İlklendirme Aşaması

- Düğüm 1 ile kuyruk başlatılır.



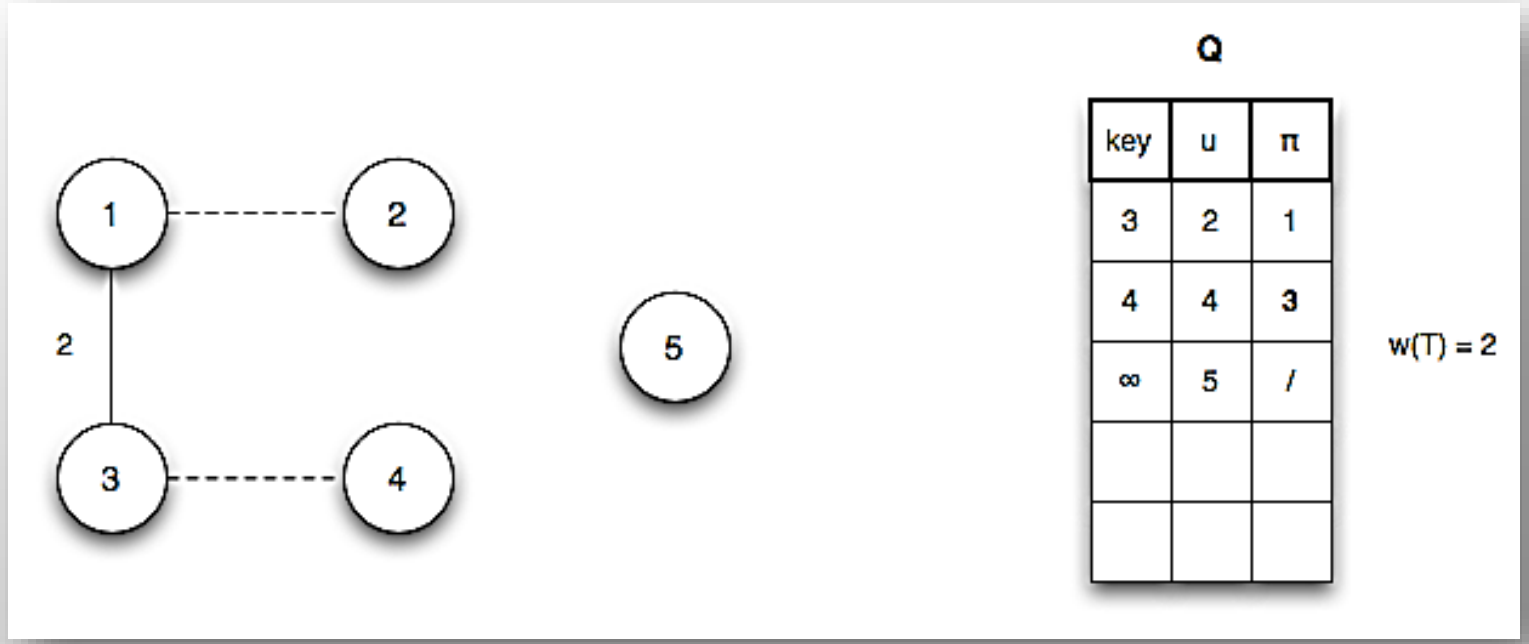
# Adım 1

- Kuyruktan düğüm 1'i al, Q'yu güncelle.
- u3.key = 2 ((u1,u3)), u2.key = 3 ((u1,u2)), u4.key = 6 ((u1,u4))



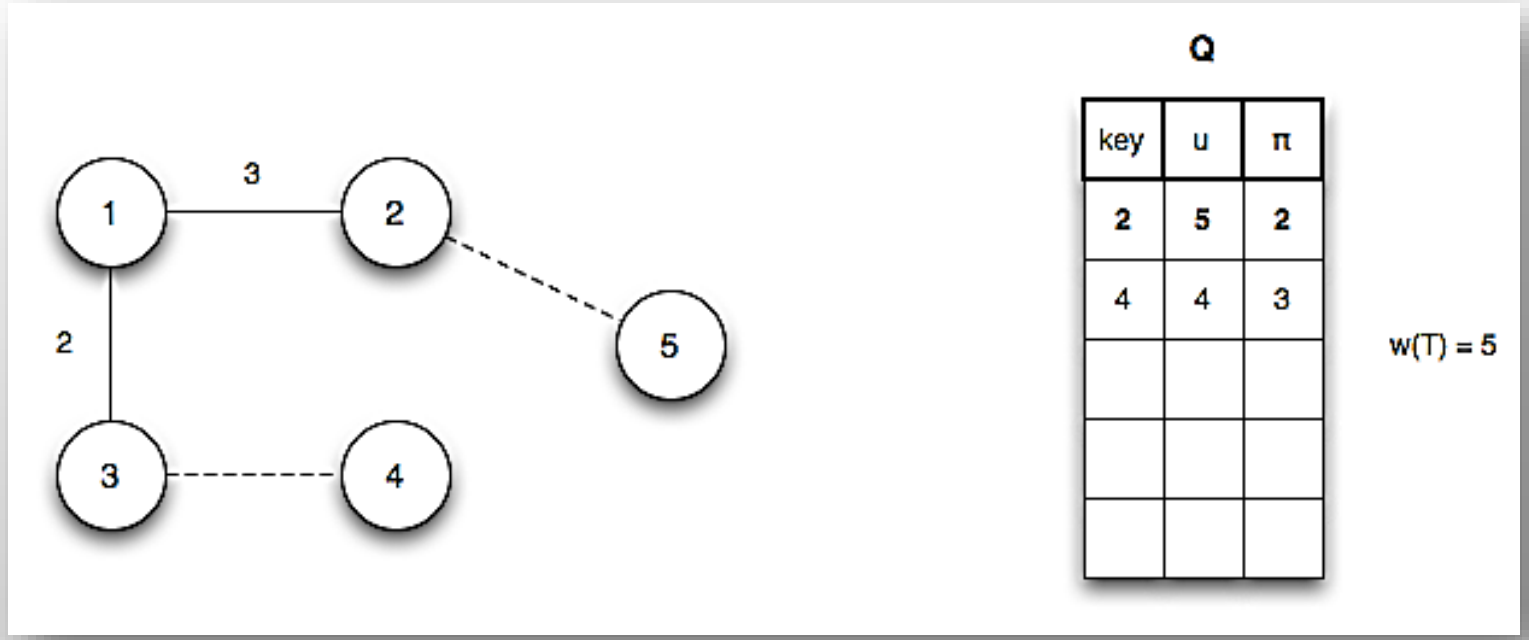
# Adım 2

- Kuyruktan düğüm 3'ü al, T'ye ($u_1$,$u_3$) kenarını ekle. Q'yu güncelle.
- u4.key = 4 ((u3,u4))



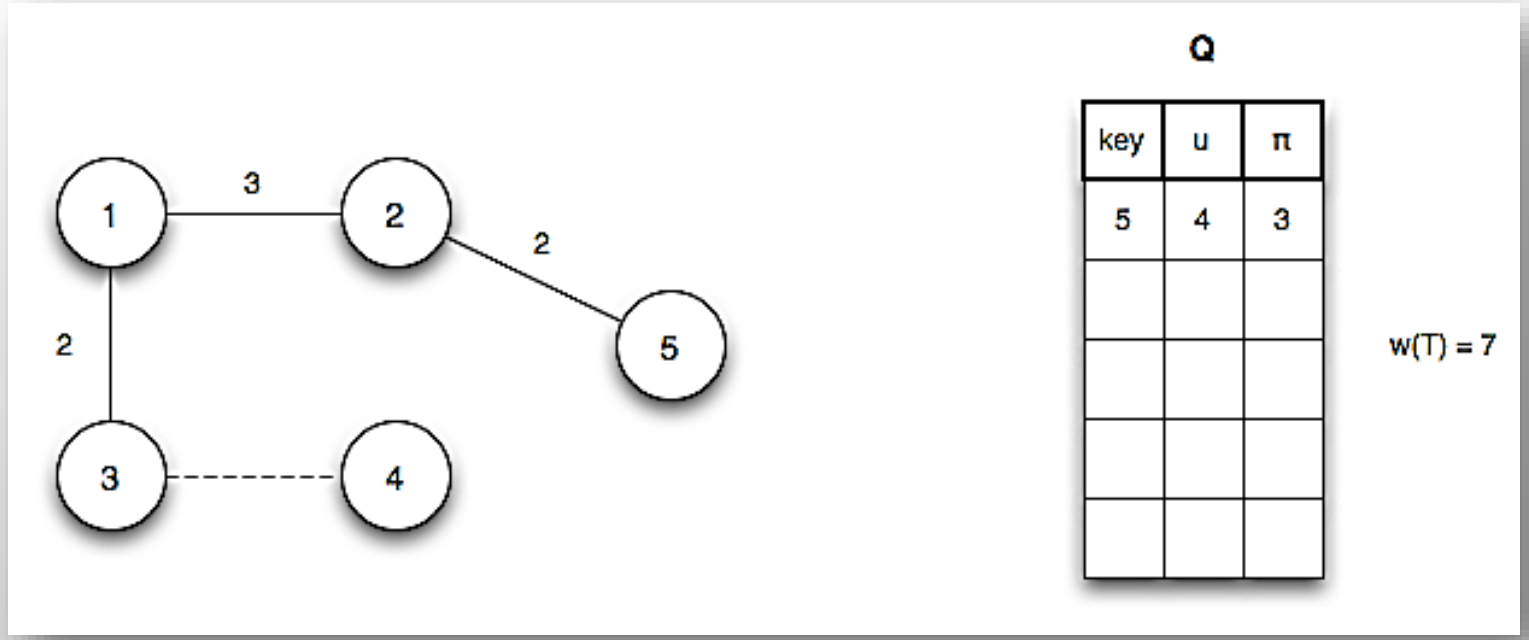
# Adım 3

- Kuyruktan düğüm 2'yi al, T'ye (u1,u2) kenarını ekle. Q'yu güncelle.
- u5.key = 2 ((u2,u5))



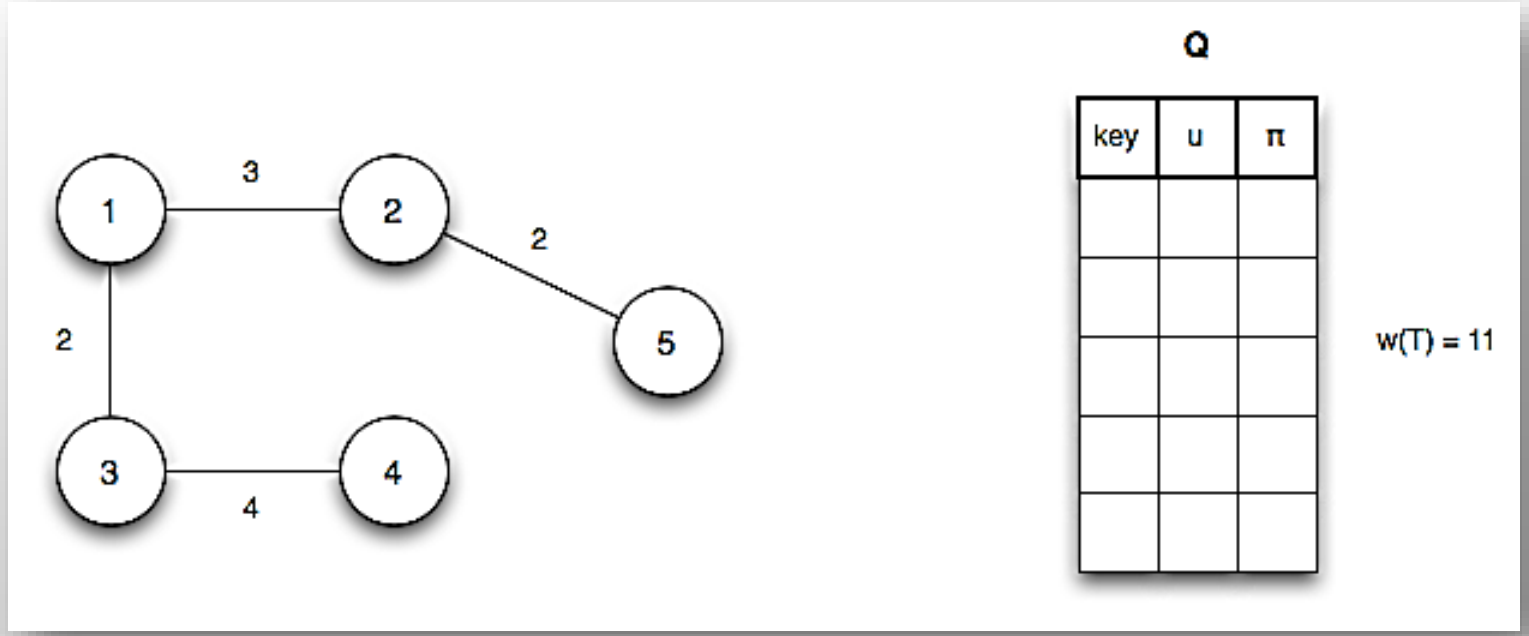
# Adım 4

- Kuyruktan düğüm 5'i al, T'ye (u2,u5) kenarını ekle. Q'da güncelleme yok.



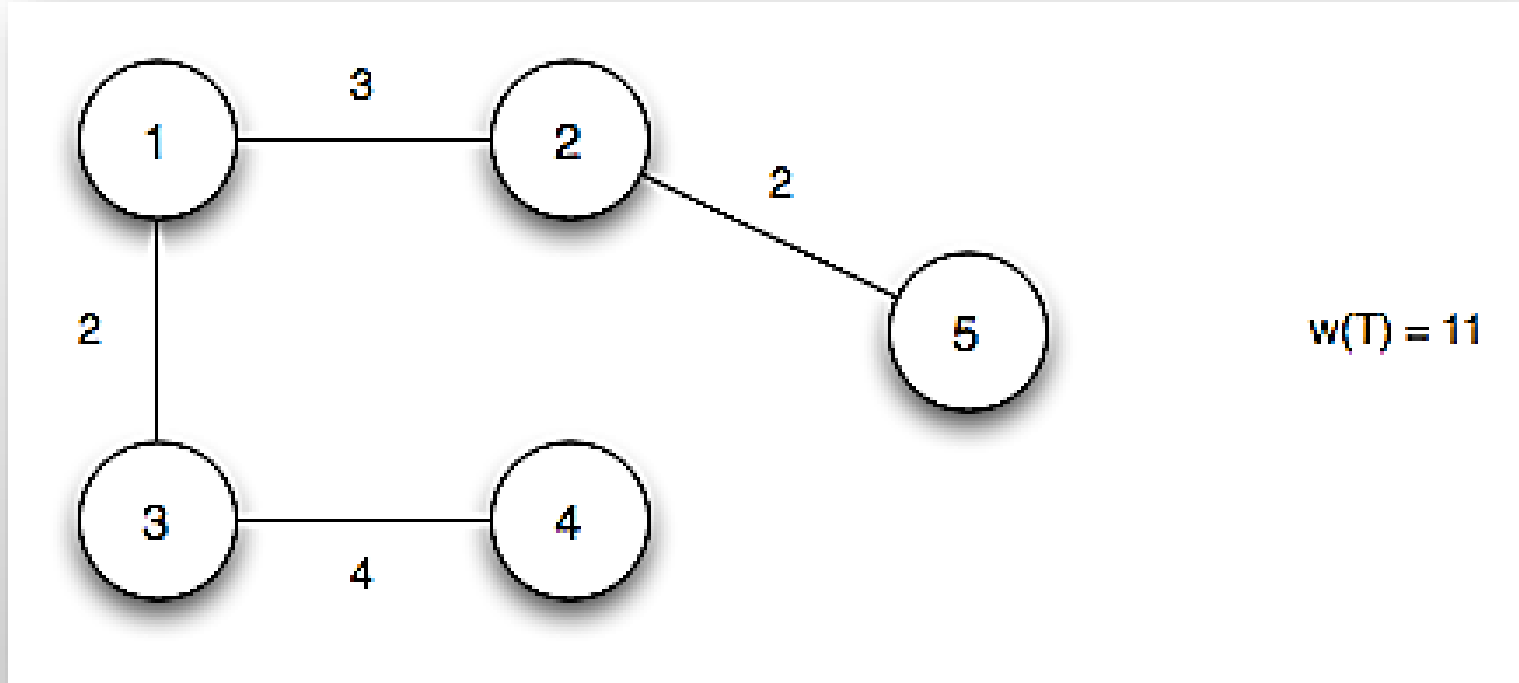
# Adım 5

- Kuyruktan düğüm 4'ü al, T'ye (u3,u4) kenarını ekle. Q'da güncelleme yok.

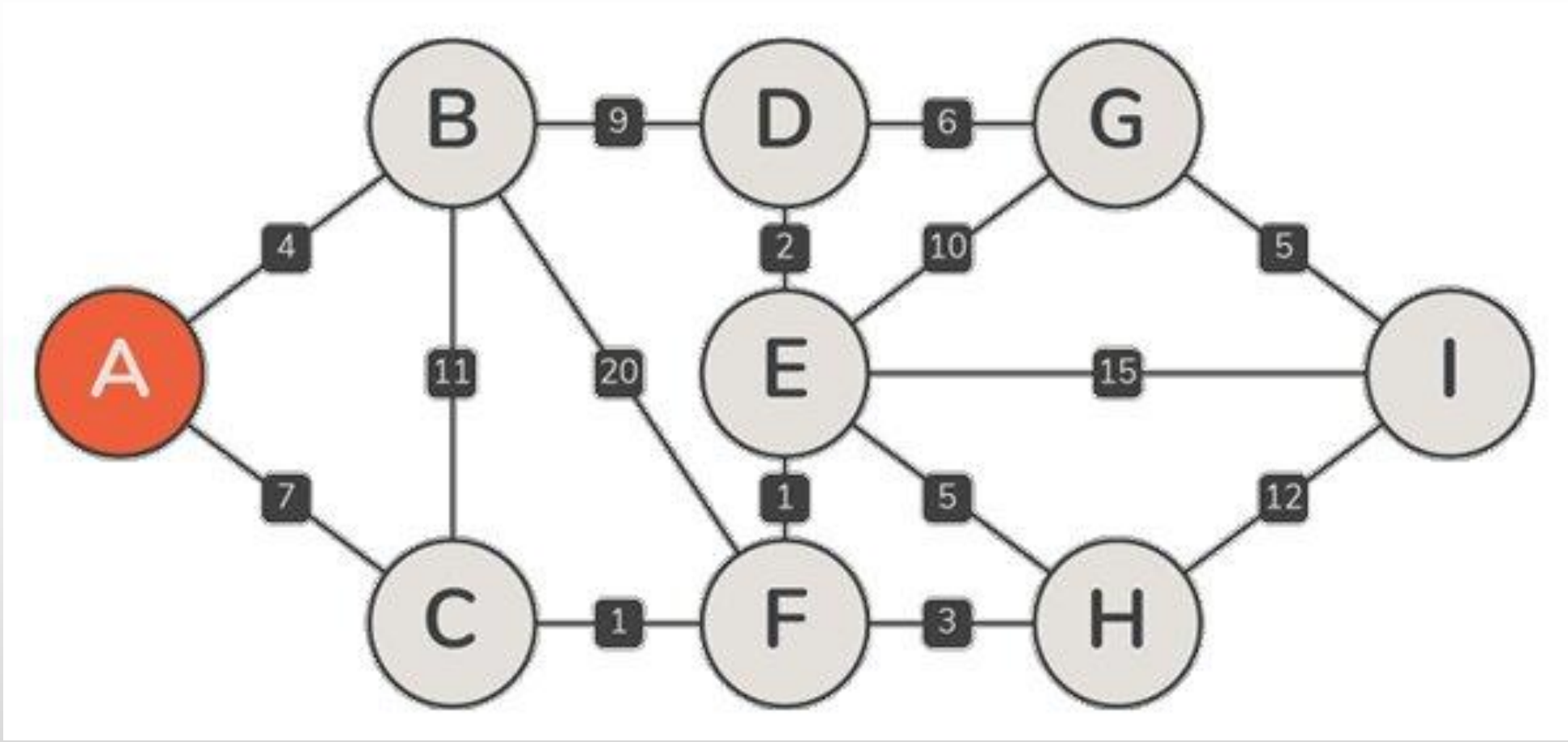


# Son Durum

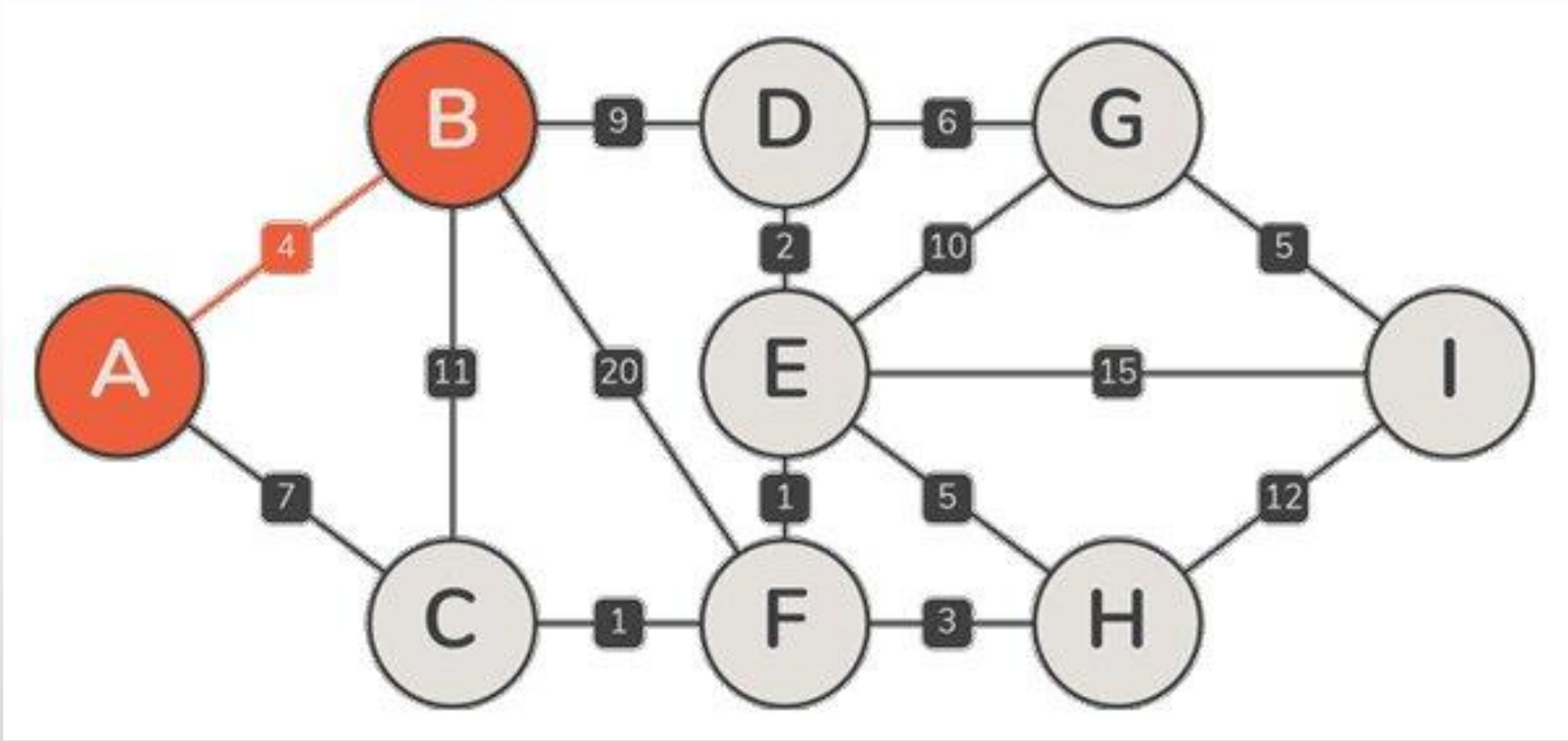
- Q = ∅



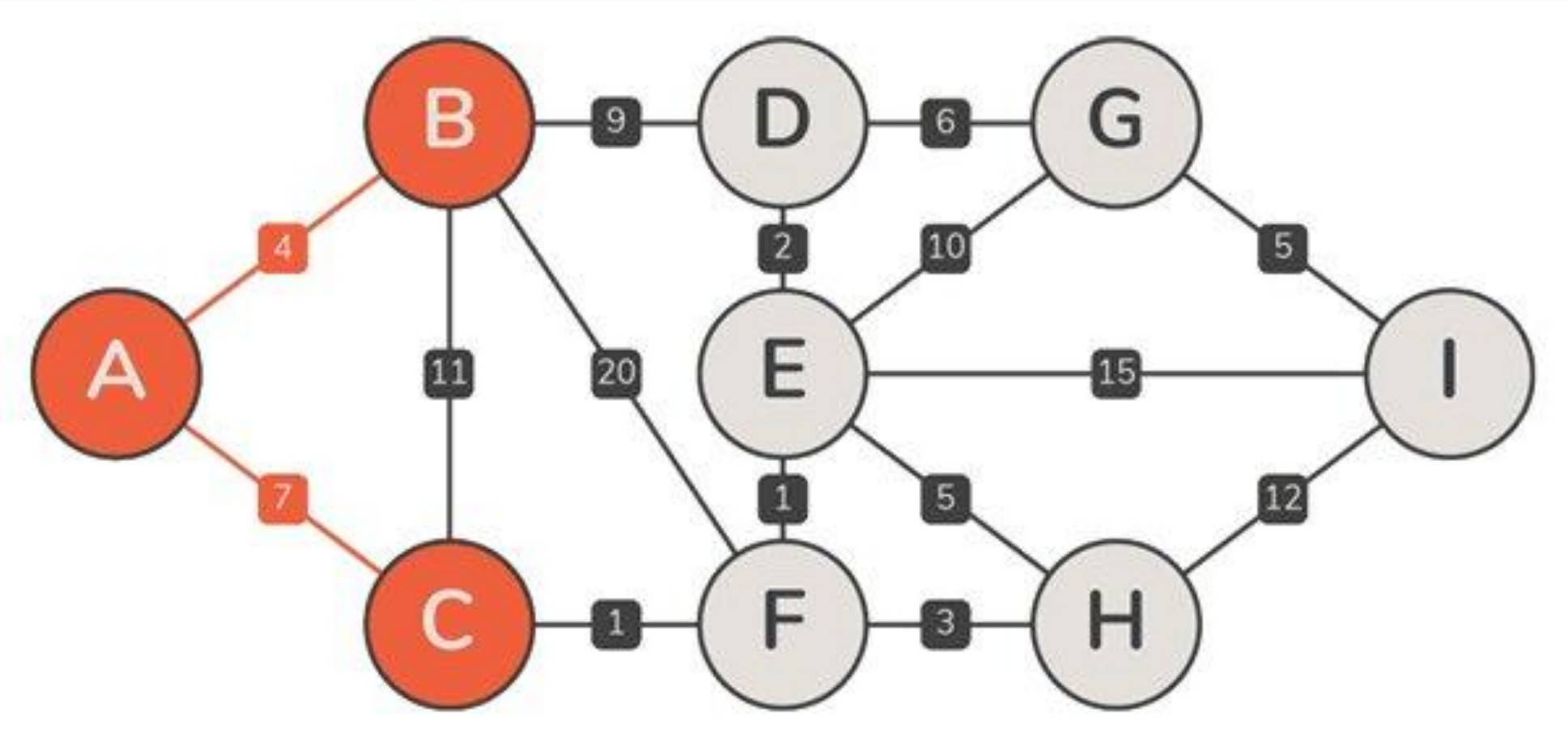
# Örnek Prim



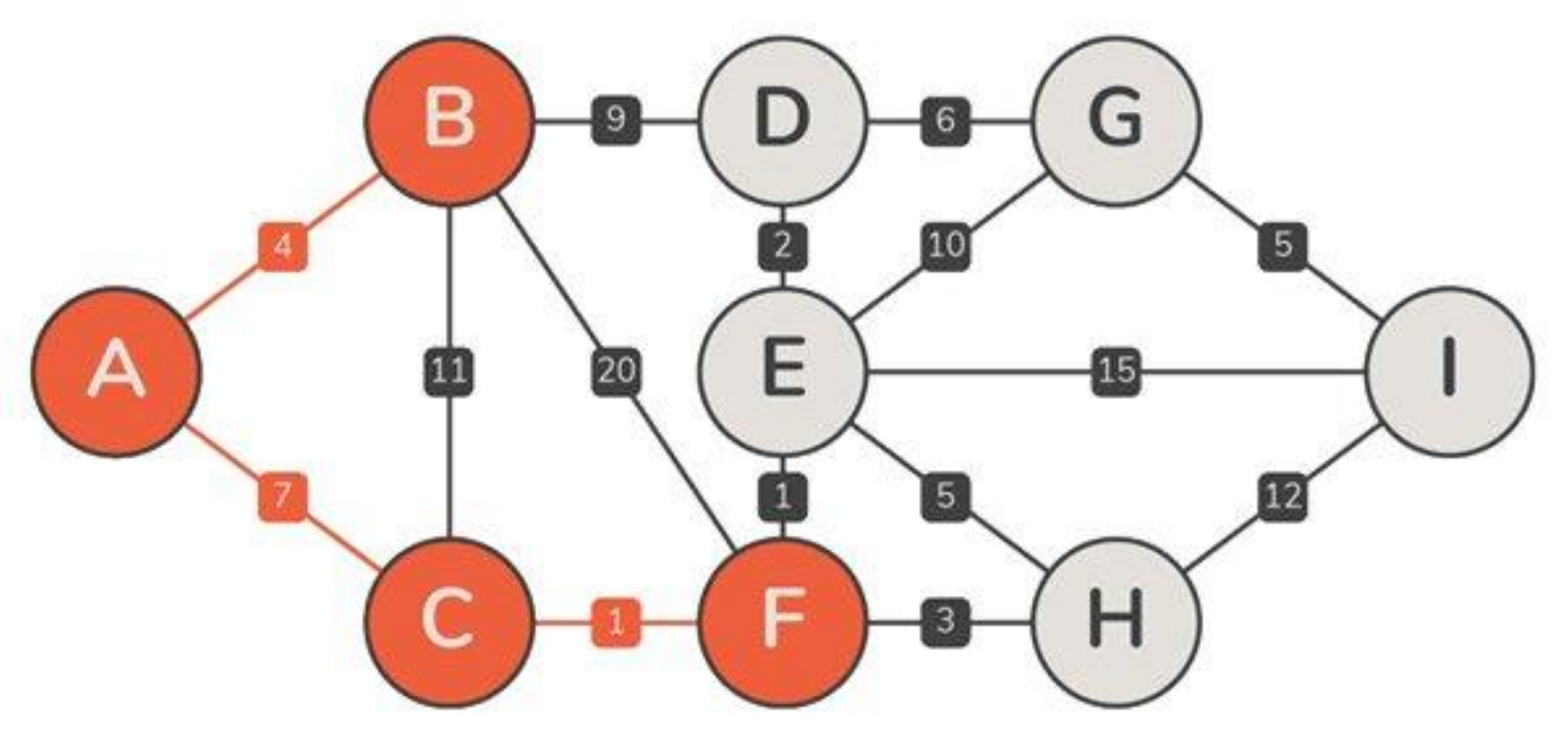
# Örnek Prim



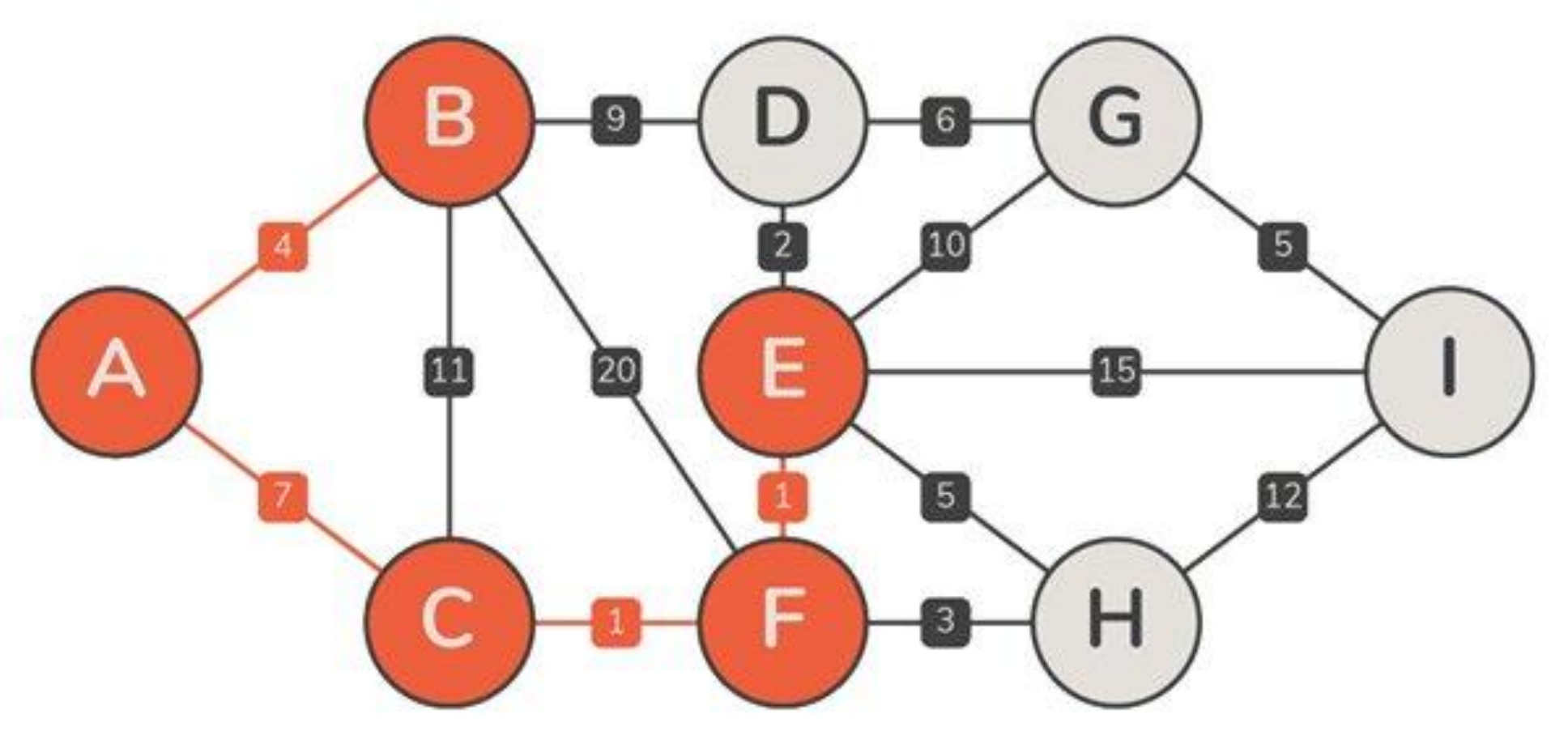
# Örnek Prim



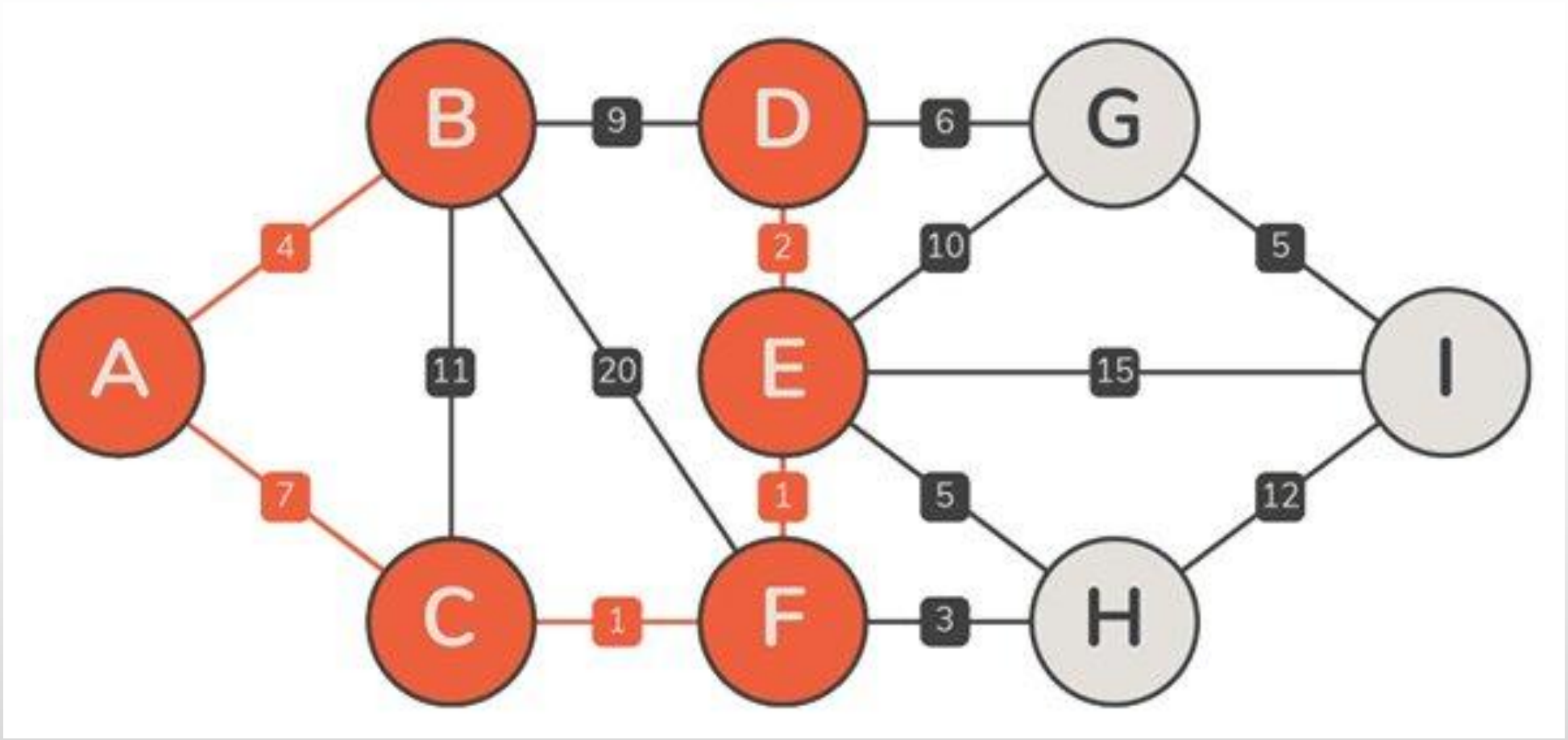
# Örnek Prim



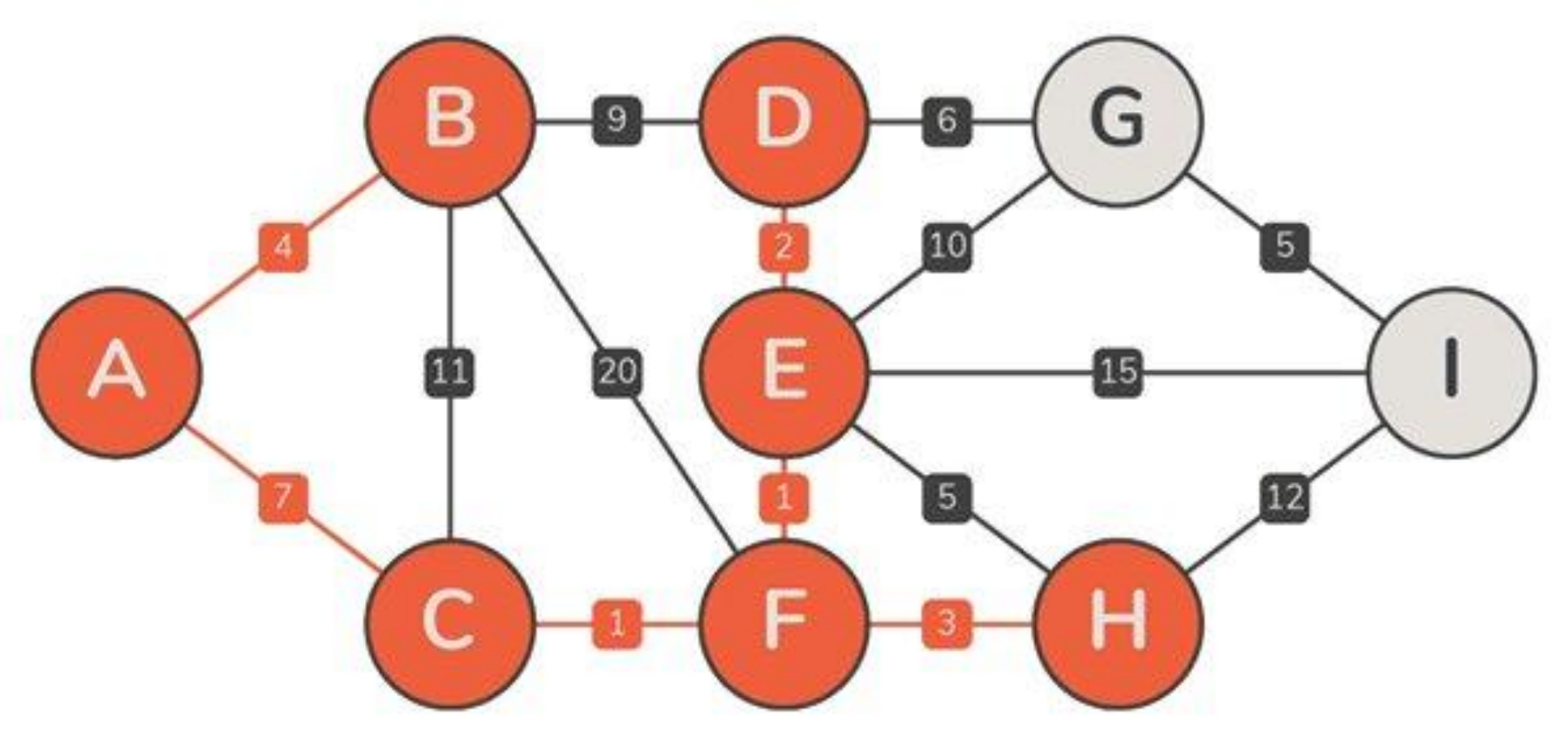
# Örnek Prim



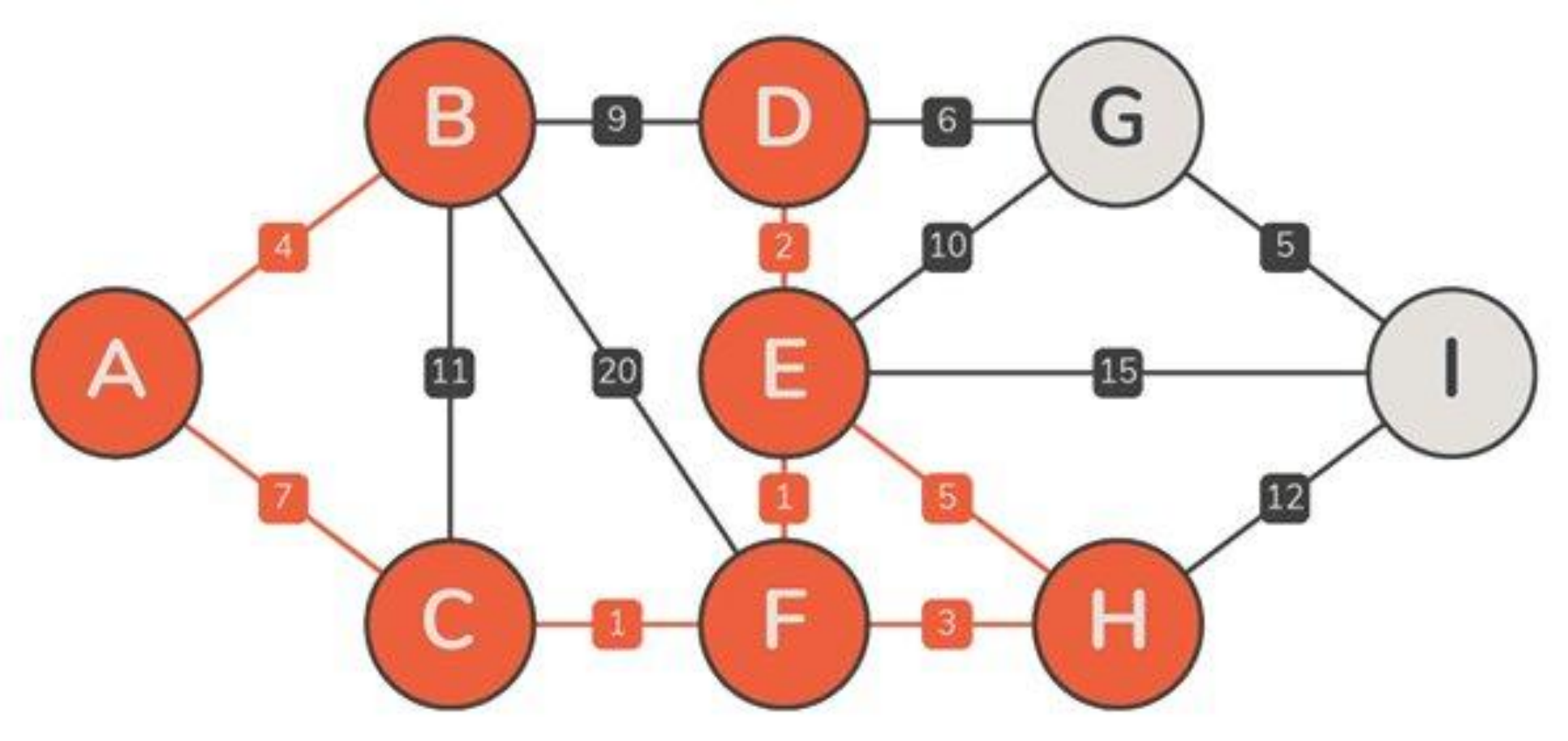
# Örnek Prim



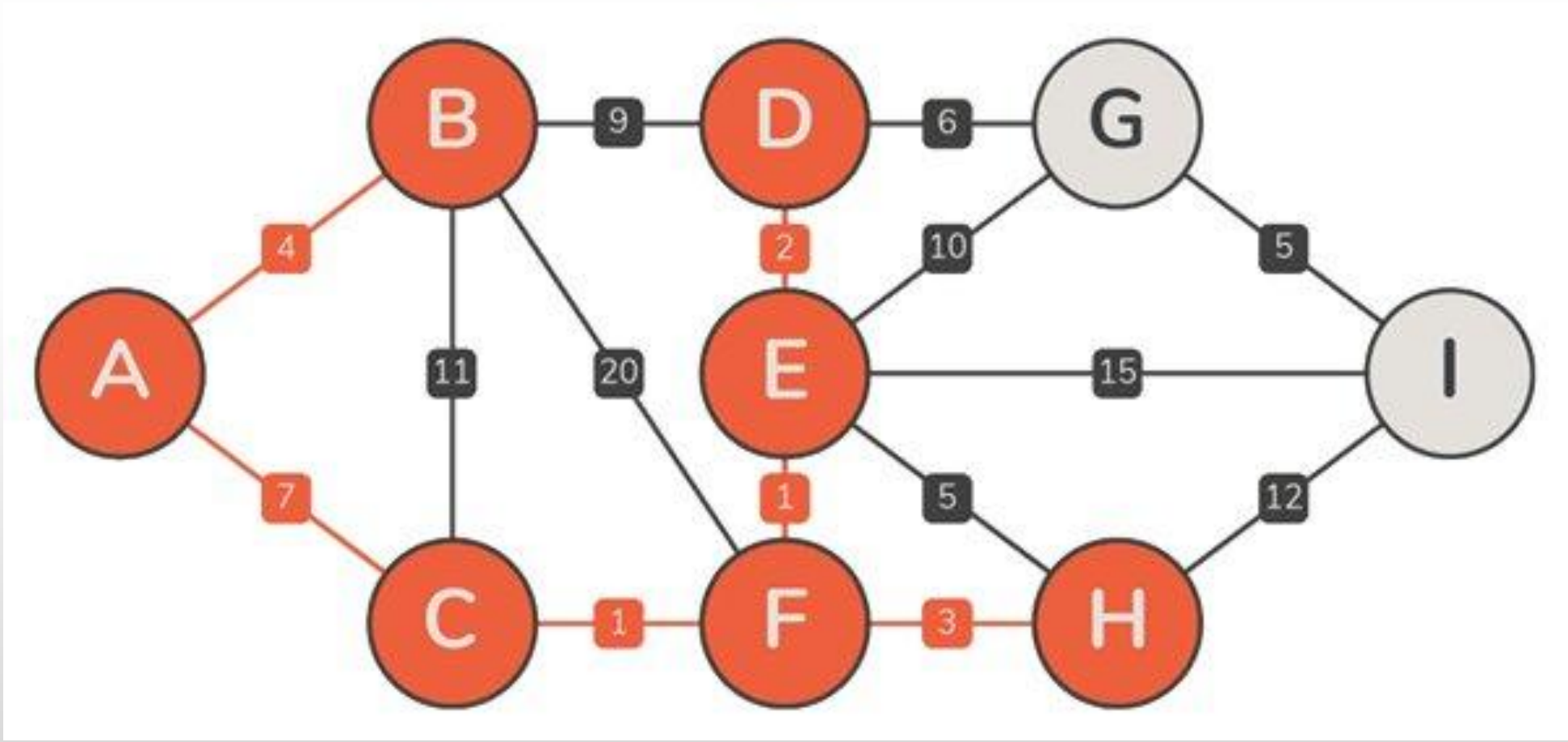
# Örnek Prim



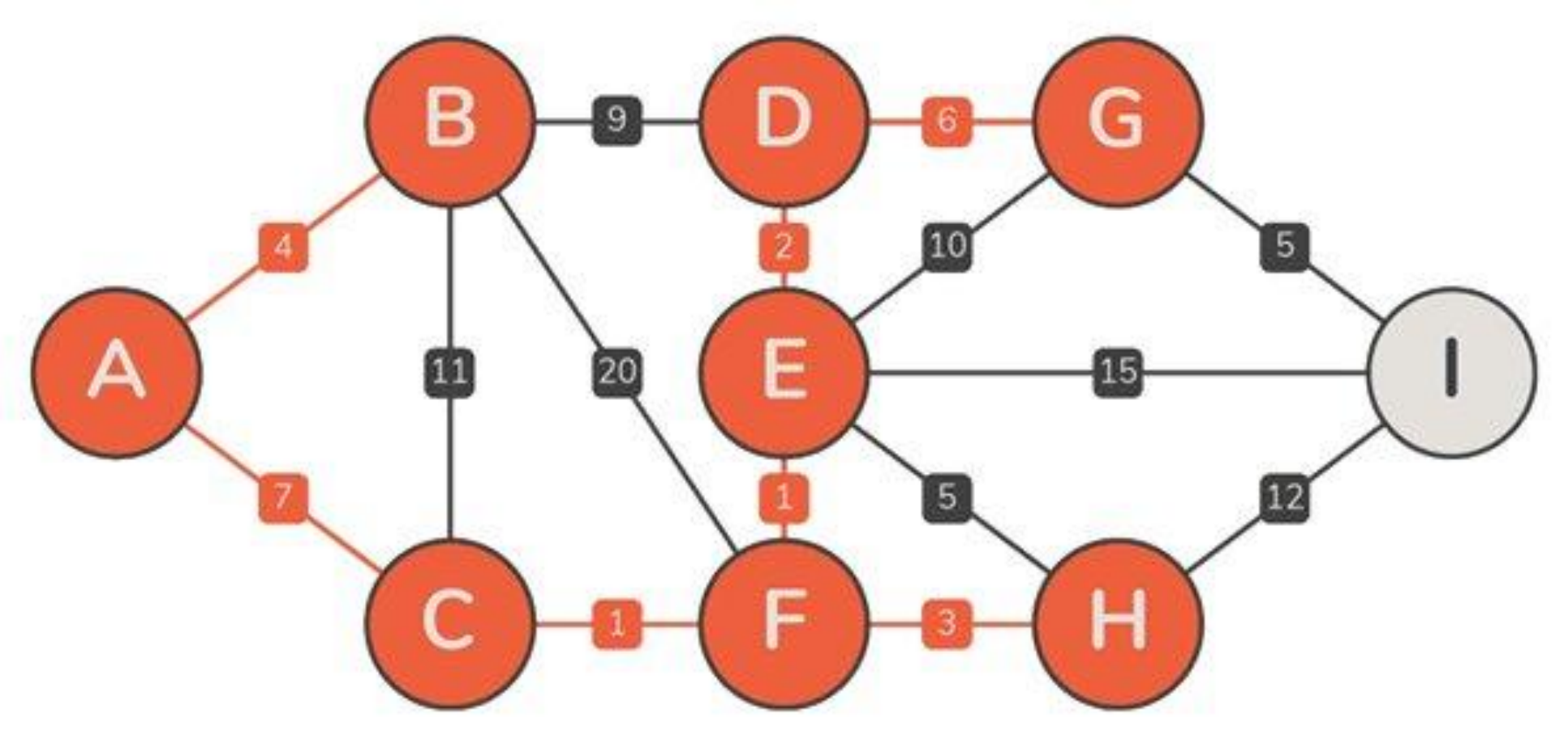
# Örnek Prim



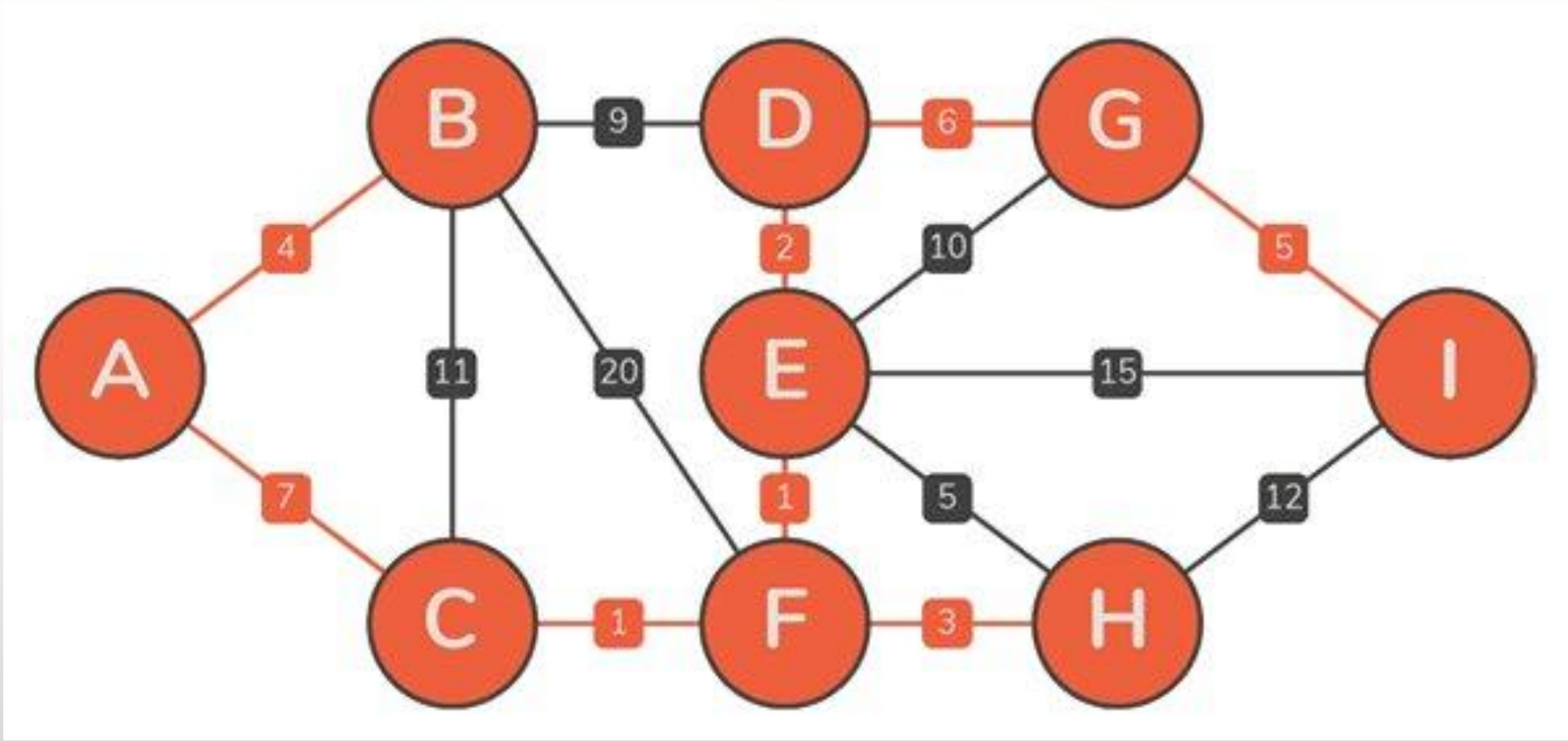
# Örnek Prim



# Örnek Prim



# Örnek Prim



# Sözde Kod

```
PRIM(G, kaynak):
        MST = [ ]
        mesafeler = [ ]
        ata = [ ]
        ziyaretEdildi = [ ]

        her bir v için V içinde:
                mesafeler[v] = ∞
                ziyaretEdildi[v] = 0
        mesafeler[kaynak] = 0
        öncelikKuyruğu.ekle(kaynak, 0)
```

# Sözde Kod (2)

```
döngü öncelikKuyruğu boş değil iken:
        u = öncelikKuyruğu.çıkar()
        eğer ziyaretEdildi[u] == 0:
                ziyaretEdildi[u] = 1
                her bir (u, v, ağırlık) için G[u] içinde:
                        eğer ziyaretEdildi[v] == 0 ve ağırlık < mesafeler[v]:
                                mesafeler[v] = ağırlık
                                ata[v] = u
                                öncelikKuyruğu.ekle(v, ağırlık)
her bir v için V içinde (kaynak hariç):
        MST.ekle((ata[v], v, mesafeler[v]))
döndür MST
```

# Boruvka Algoritması

- Minimum kapsayan ağaçları bulmak için kullanılır.
- Çizgenin tüm düğümlerini kapsayan ve ağırlığı minimum alt çizgeyi bulur.
- Nasıl Çalışır?
  - Başlangıçta her düğüm birbirinden bağımsız bileşen kabul edilir.
  - Her bileşen için en ucuz kenarı seçer ve ekler.
  - Union-Find yapısı ile bileşenleri birleştirir.
  - Tek bir bileşen kalana kadar devam eder.
  - Bağlantılı çizge için |V| - 1 kenar üretir.

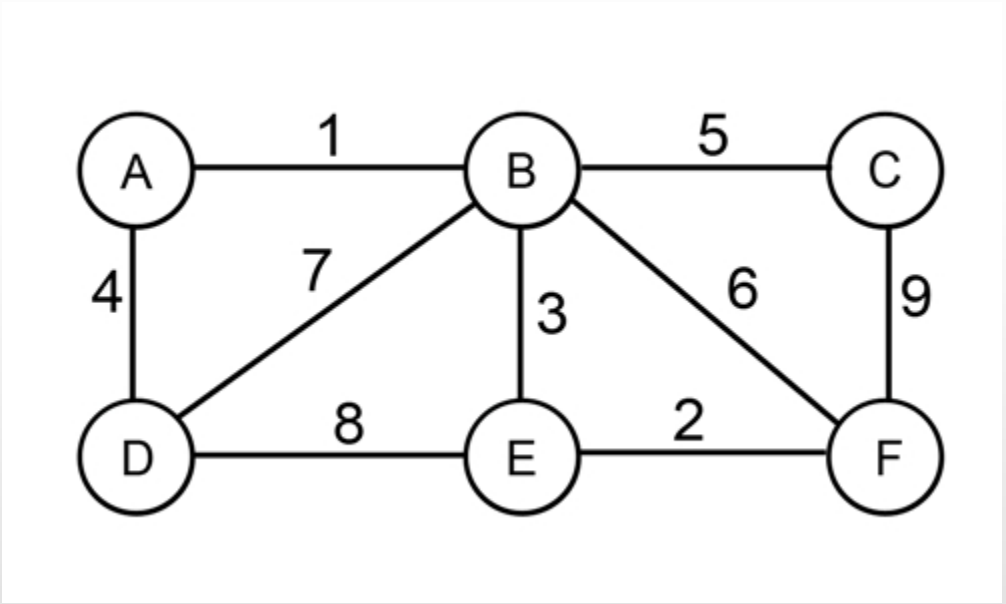
# Algoritma Adımları

- Başlangıçta her düğümü tek başına bir küme olarak ele al.
- En küçük ağırlıklı kenarı seç ve bu kenarı bağlayan düğümleri birleştir.
- Her adımda oluşan alt küme ağacının ağırlığını güncelle.
- Tüm düğümler birleşinceye kadar 2 ve 3. adımları tekrarla.

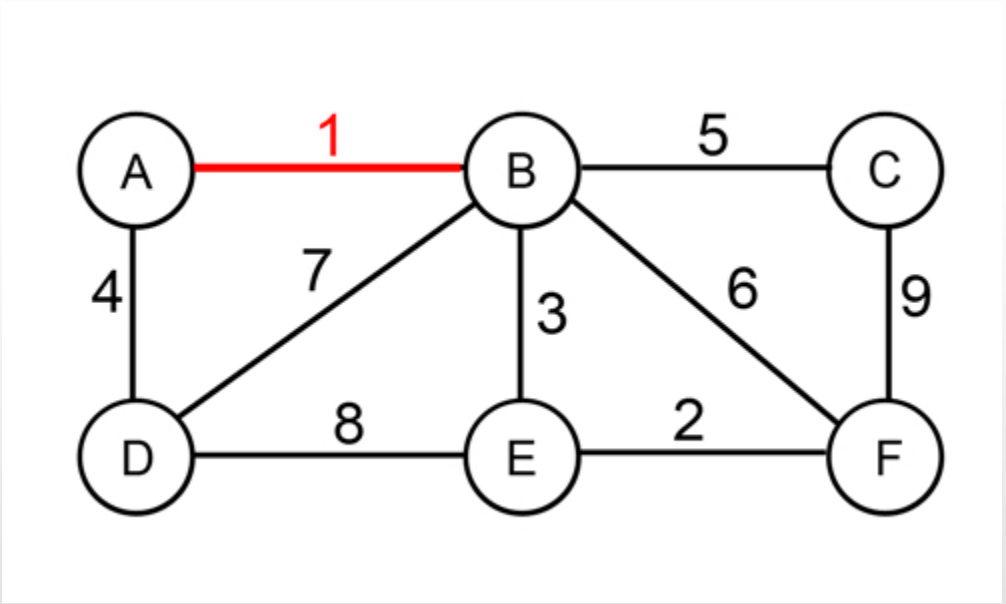
# Özellikleri

- Ağırlık sıralaması yaparak minimum ağaçları bulur.
- Aç gözlü (*Greedy*) bir algoritmadır.
- Karmaşıklığı *O(E log V)*'dir,
  - E kenar sayısı,
  - V düğüm sayısıdır.

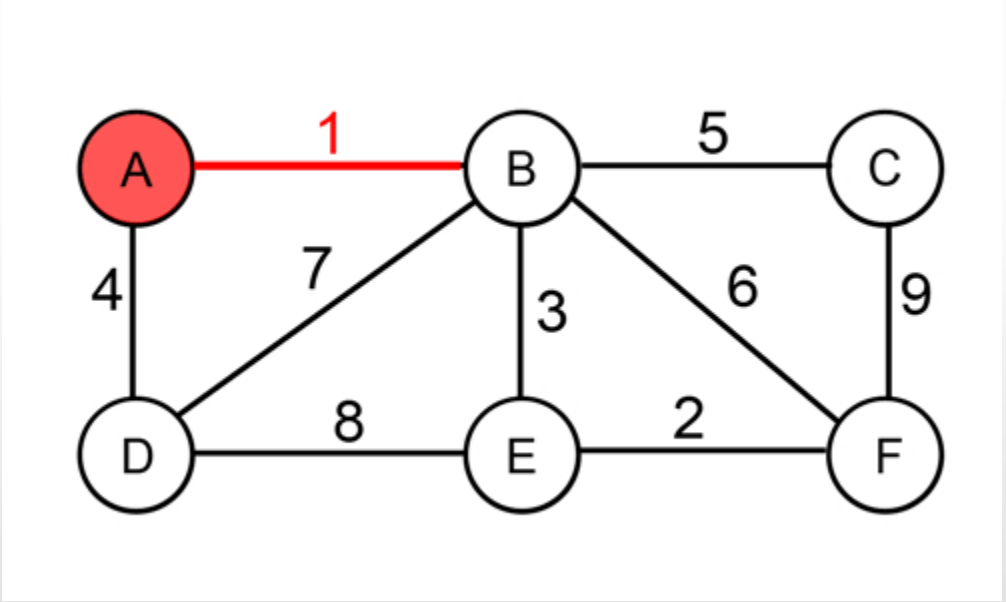
# Örnek Boruvka



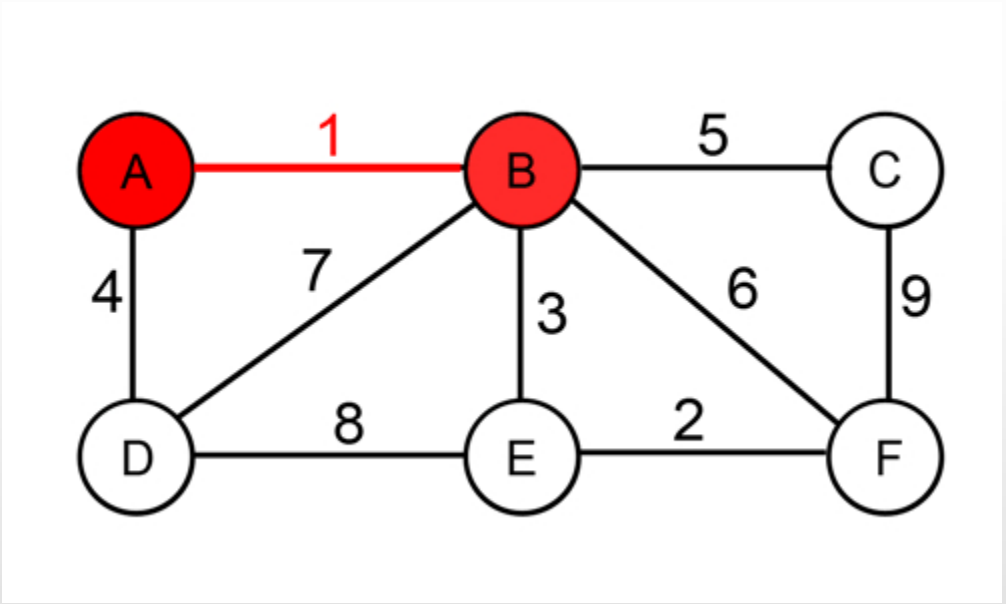
# Örnek Boruvka



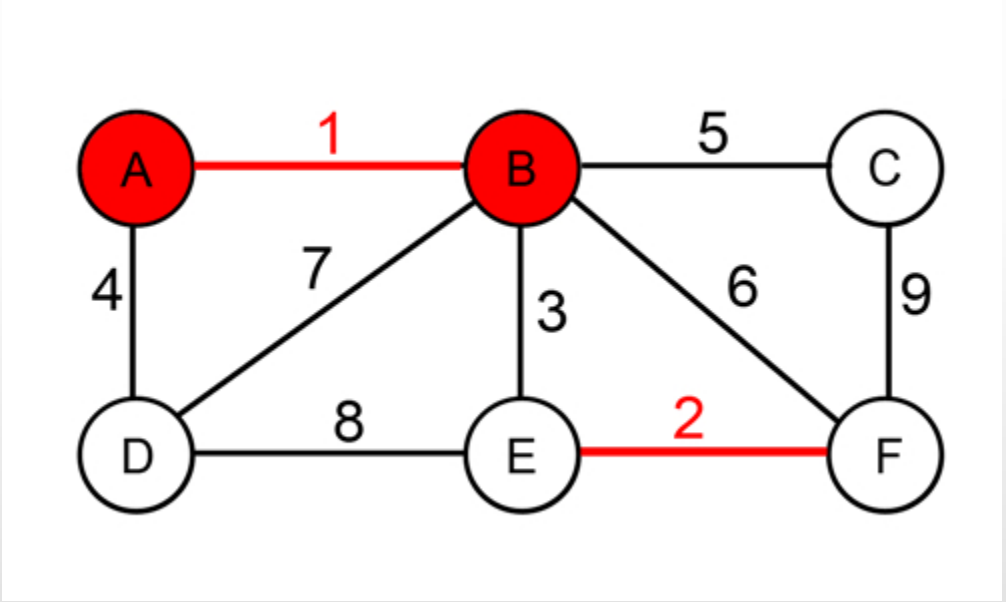
# Örnek Boruvka



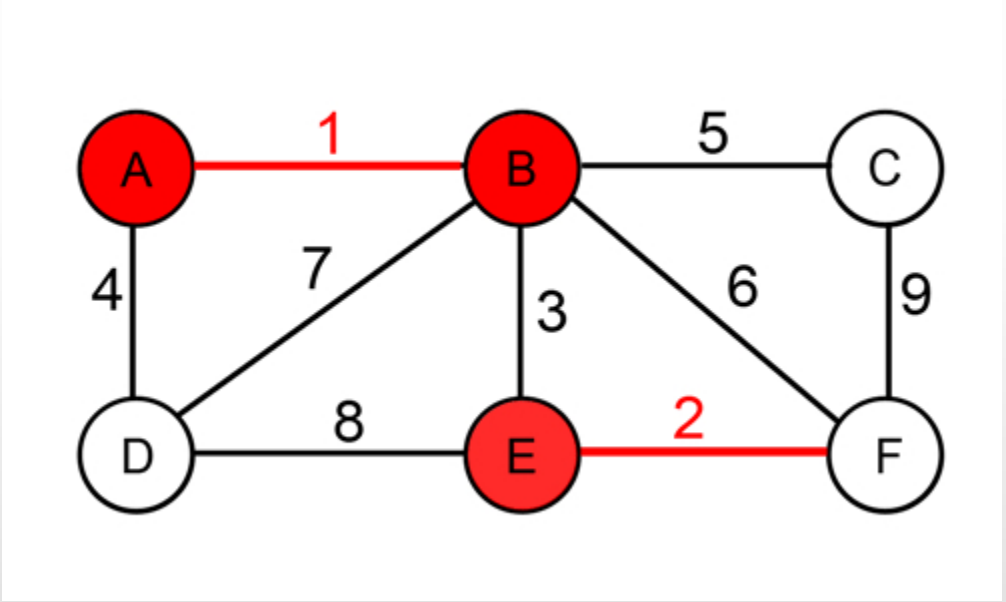
# Örnek Boruvka



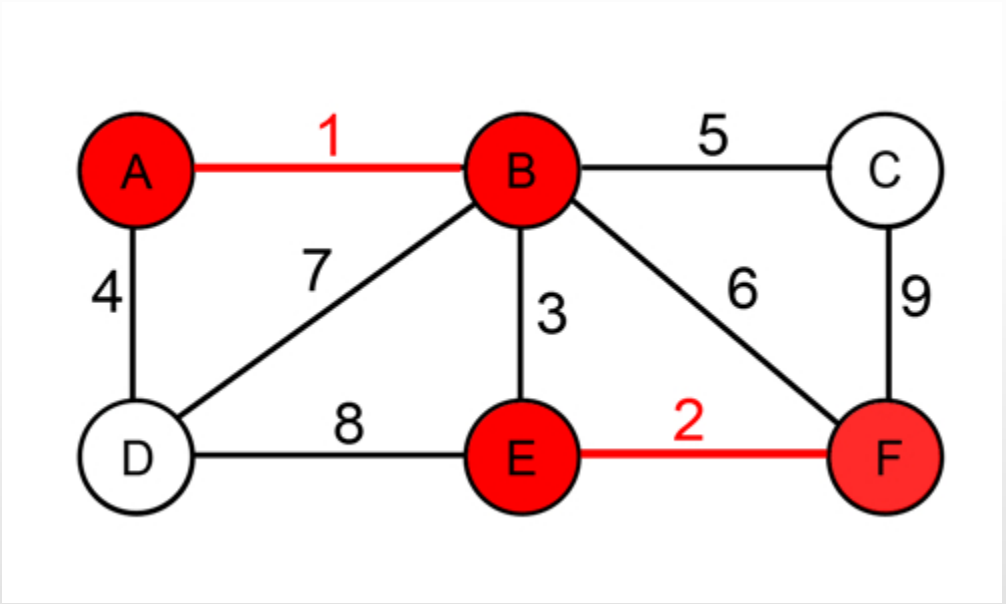
# Örnek Boruvka



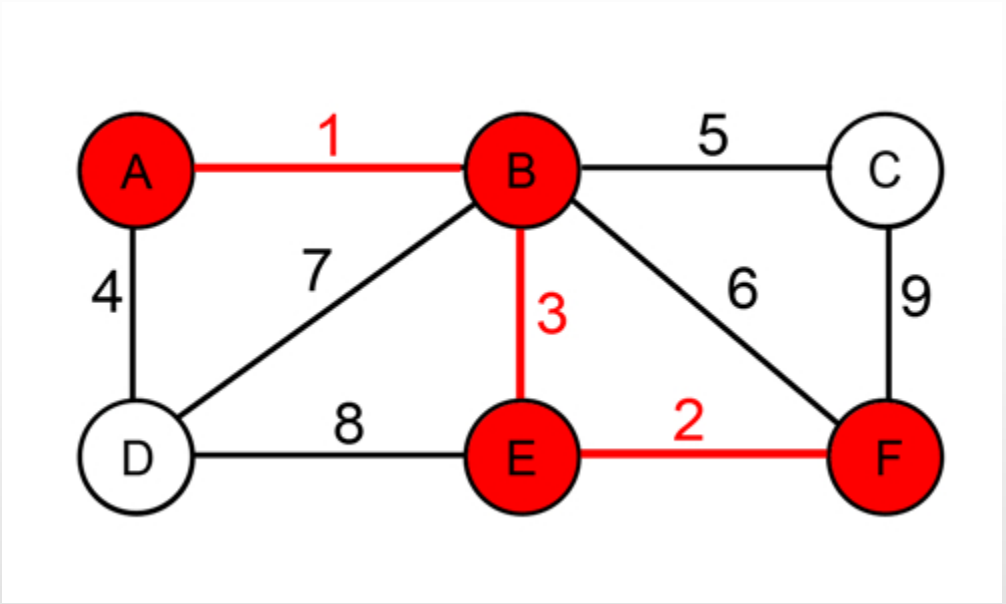
# Örnek Boruvka



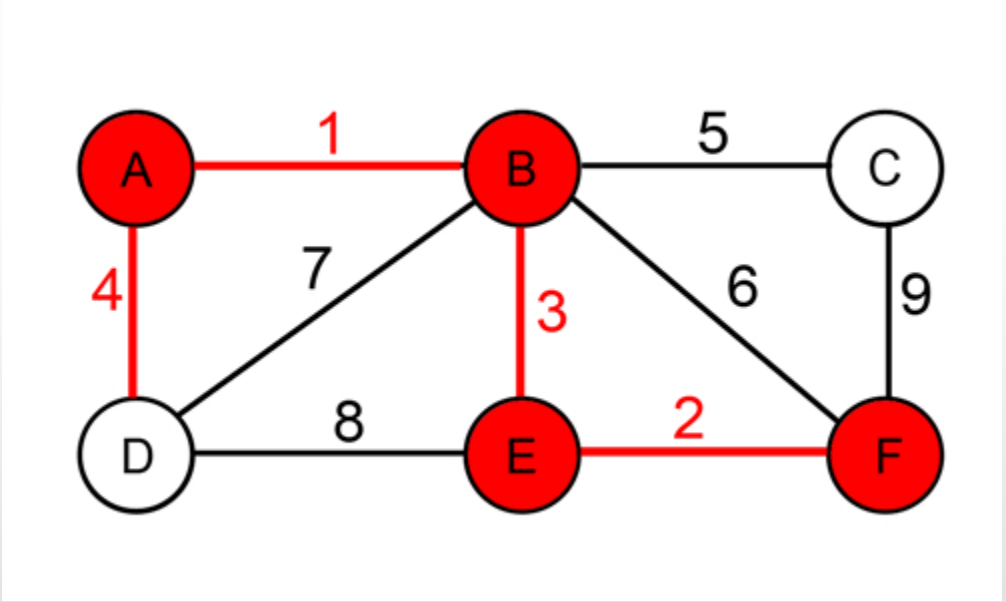
# Örnek Boruvka



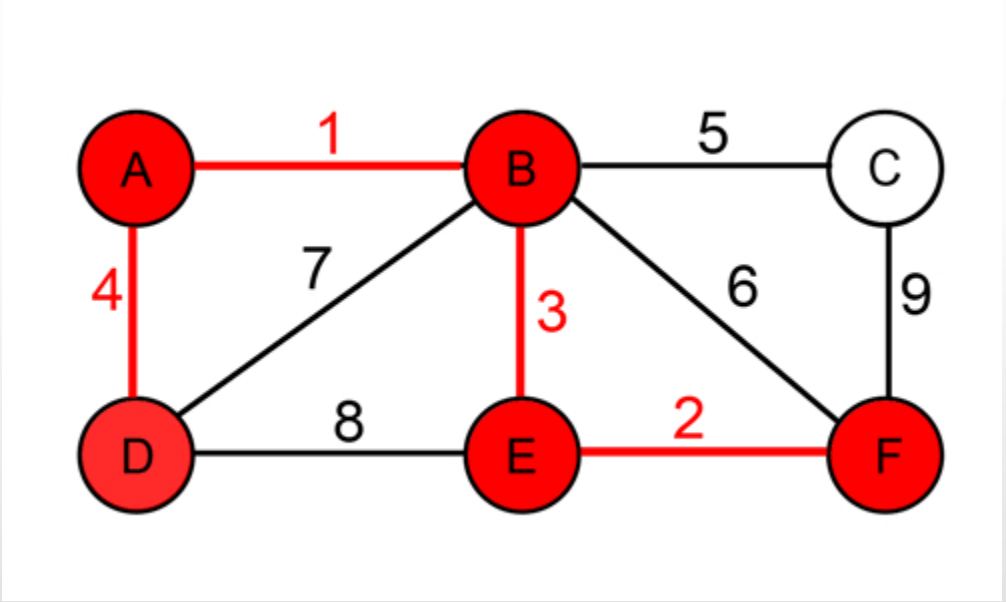
# Örnek Boruvka



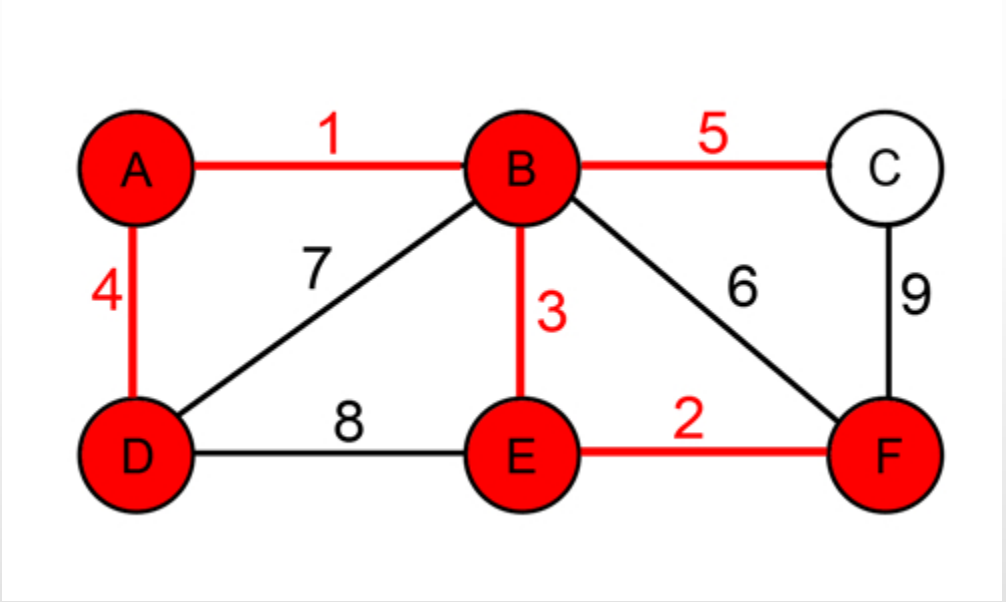
# Örnek Boruvka



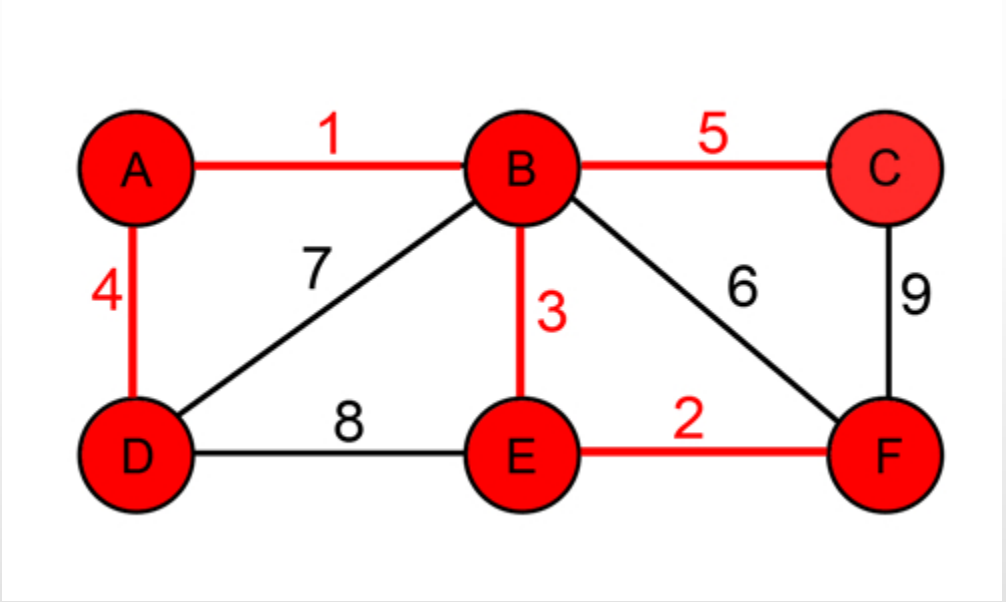
# Örnek Boruvka



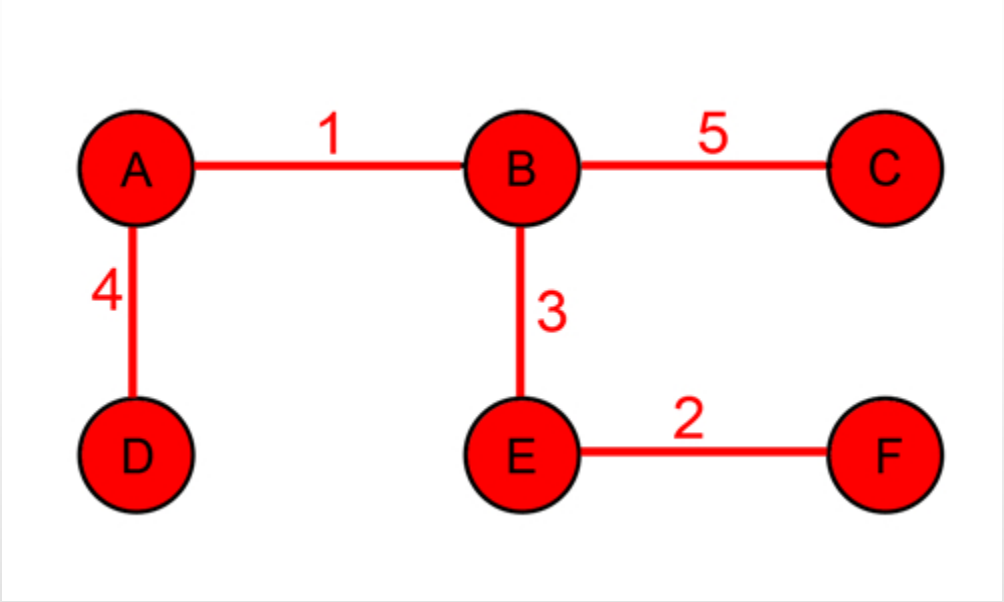
# Örnek Boruvka



# Örnek Boruvka



# Örnek Boruvka



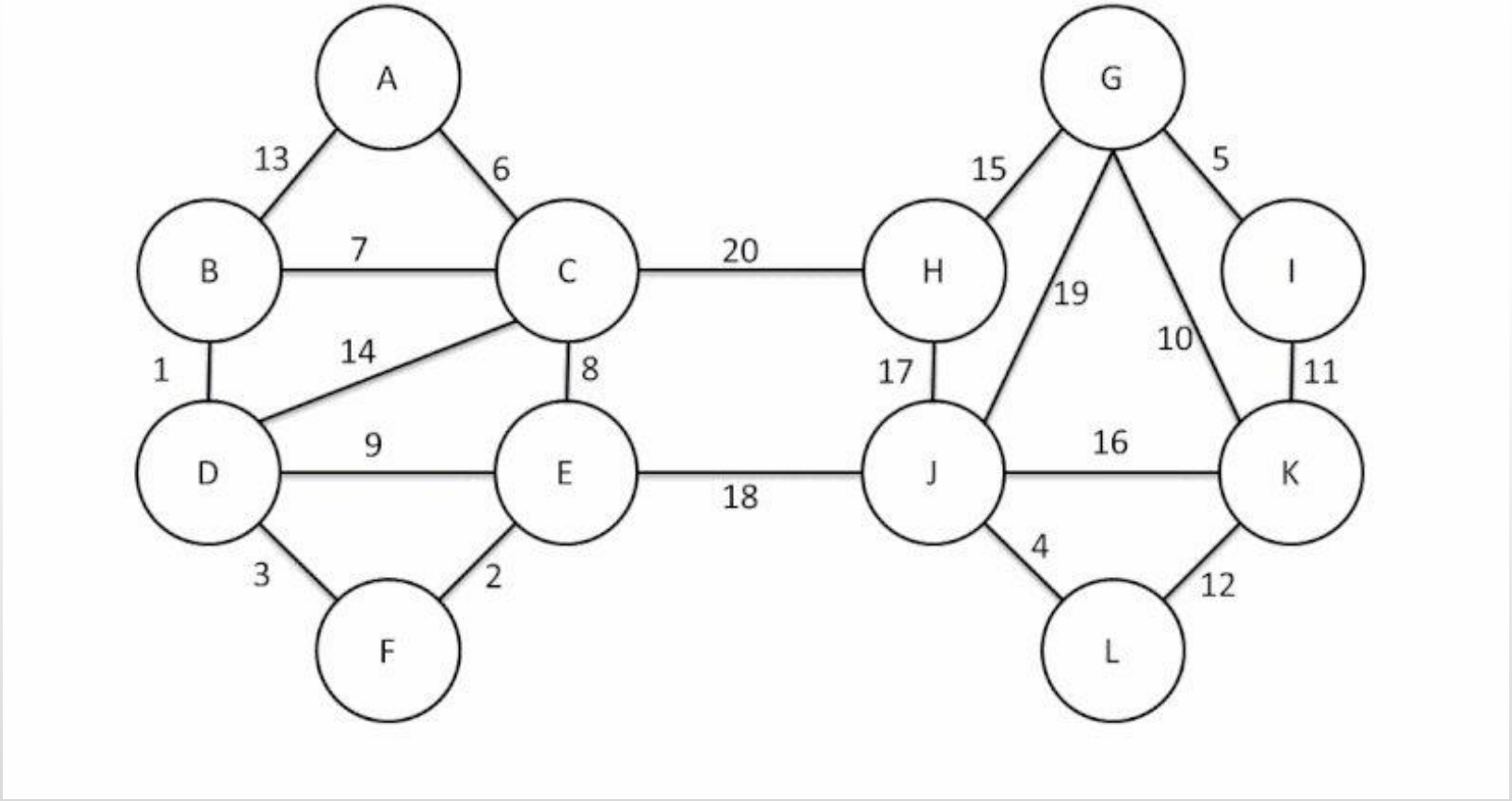
# Sözde Kod

```
BORUVKA(G):
MST = [ ]
bileşenSayısı = |V|
döngü bileşenSayısı > 1 iken:
        enUcuzKenar = [ ]  // Her bileşen için en ucuz kenar
        her bir (u, v, ağırlık) için E içinde:
                cu = union-find.bul(u)
                cv = union-find.bul(v)
                eğer cu ≠ cv:
                        enUcuzKenar[cu] = min(enUcuzKenar[cu], (u, v, ağırlık))
                        enUcuzKenar[cv] = min(enUcuzKenar[cv], (u, v, ağırlık))
```

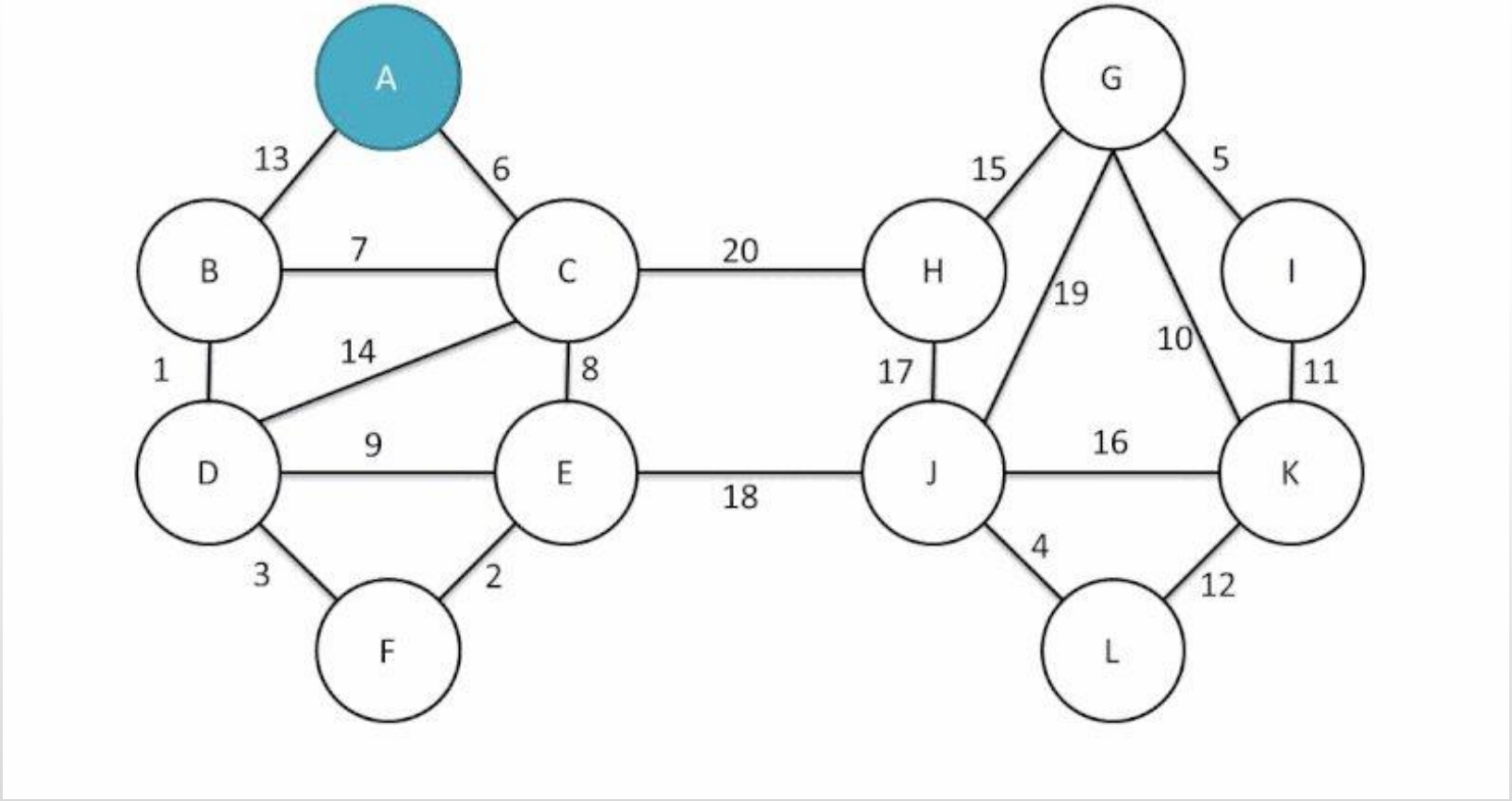
# Sözde Kod (2)

```
her bir (u, v, ağırlık) için enUcuzKenar içinde:
        eğer union-find.bul(u) ≠ union-find.bul(v):
                MST.ekle((u, v, ağırlık))
                union-find.birleştir(u, v)
                bileşenSayısı -= 1
döndür MST
```

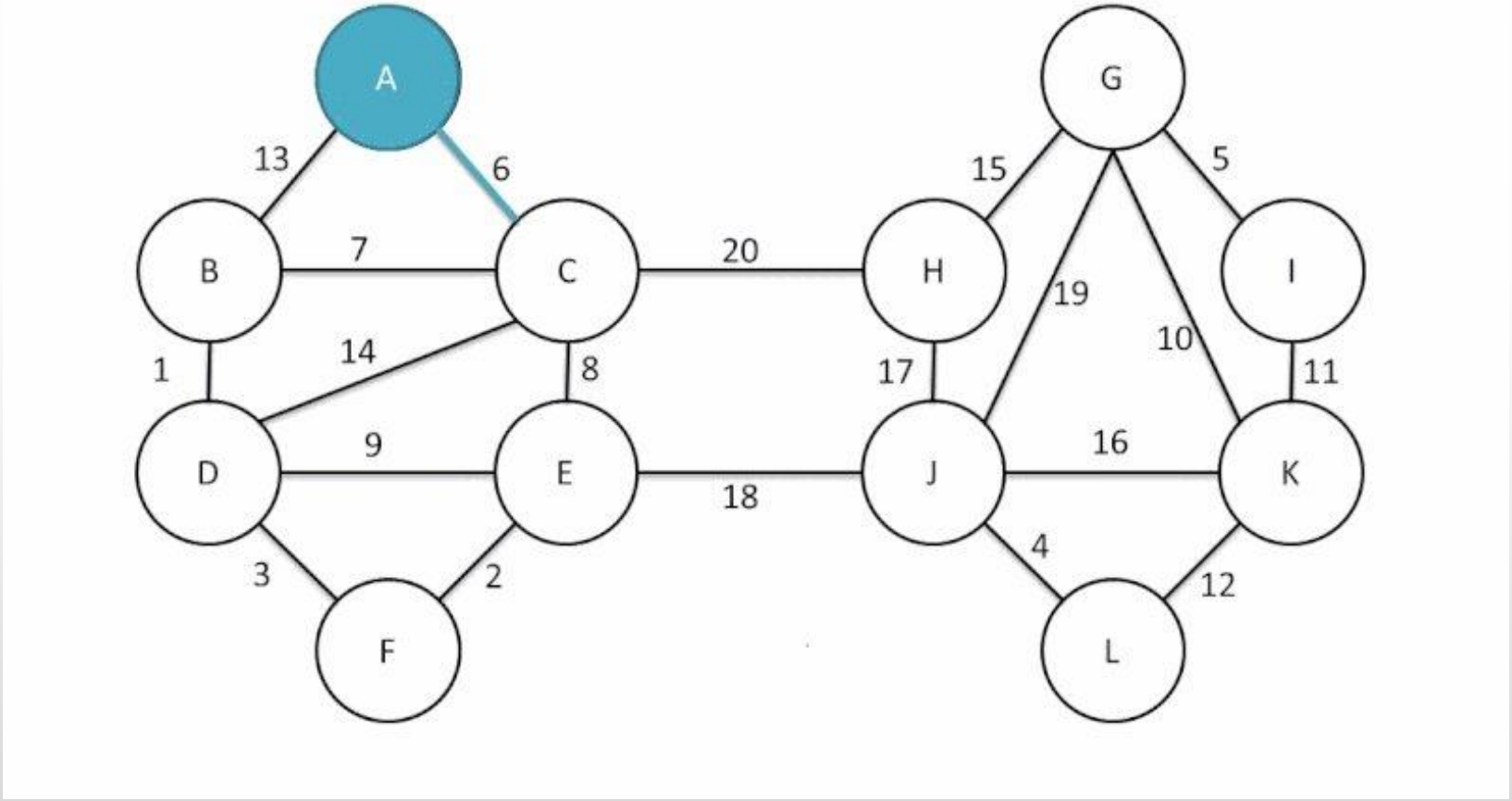
# Boruvka's Algorithm



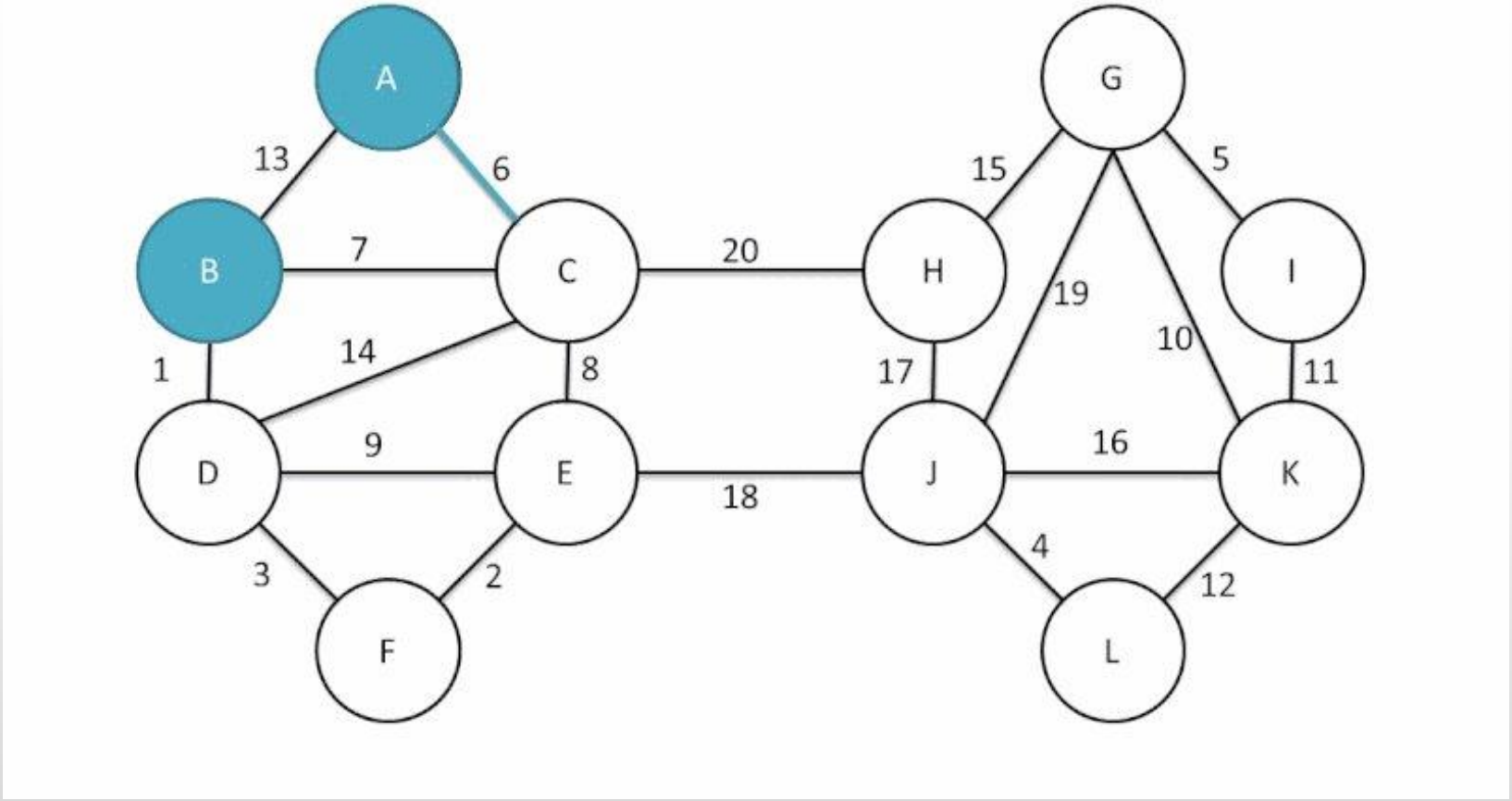
# Boruvka's Algorithm



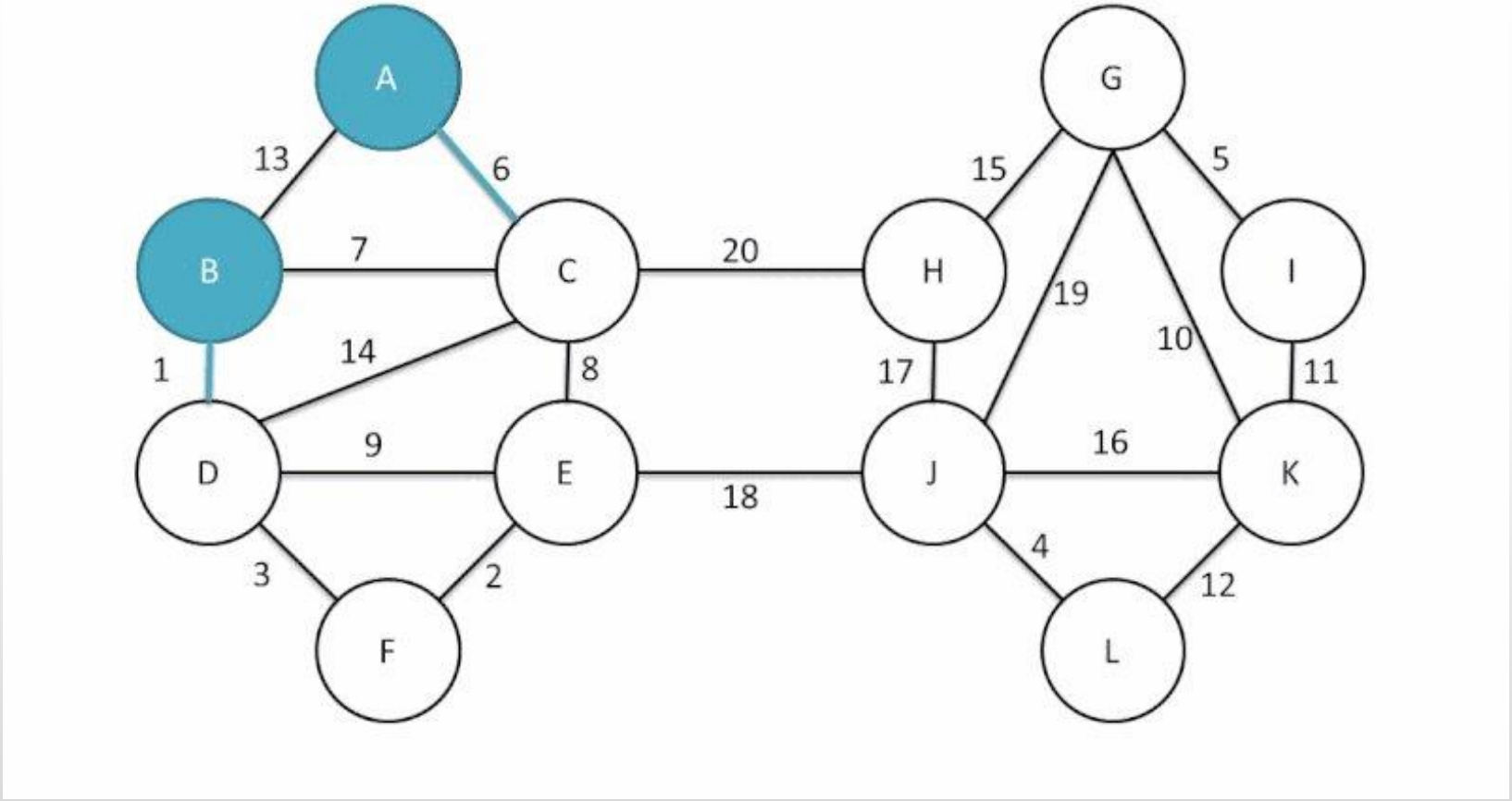
# Boruvka's Algorithm



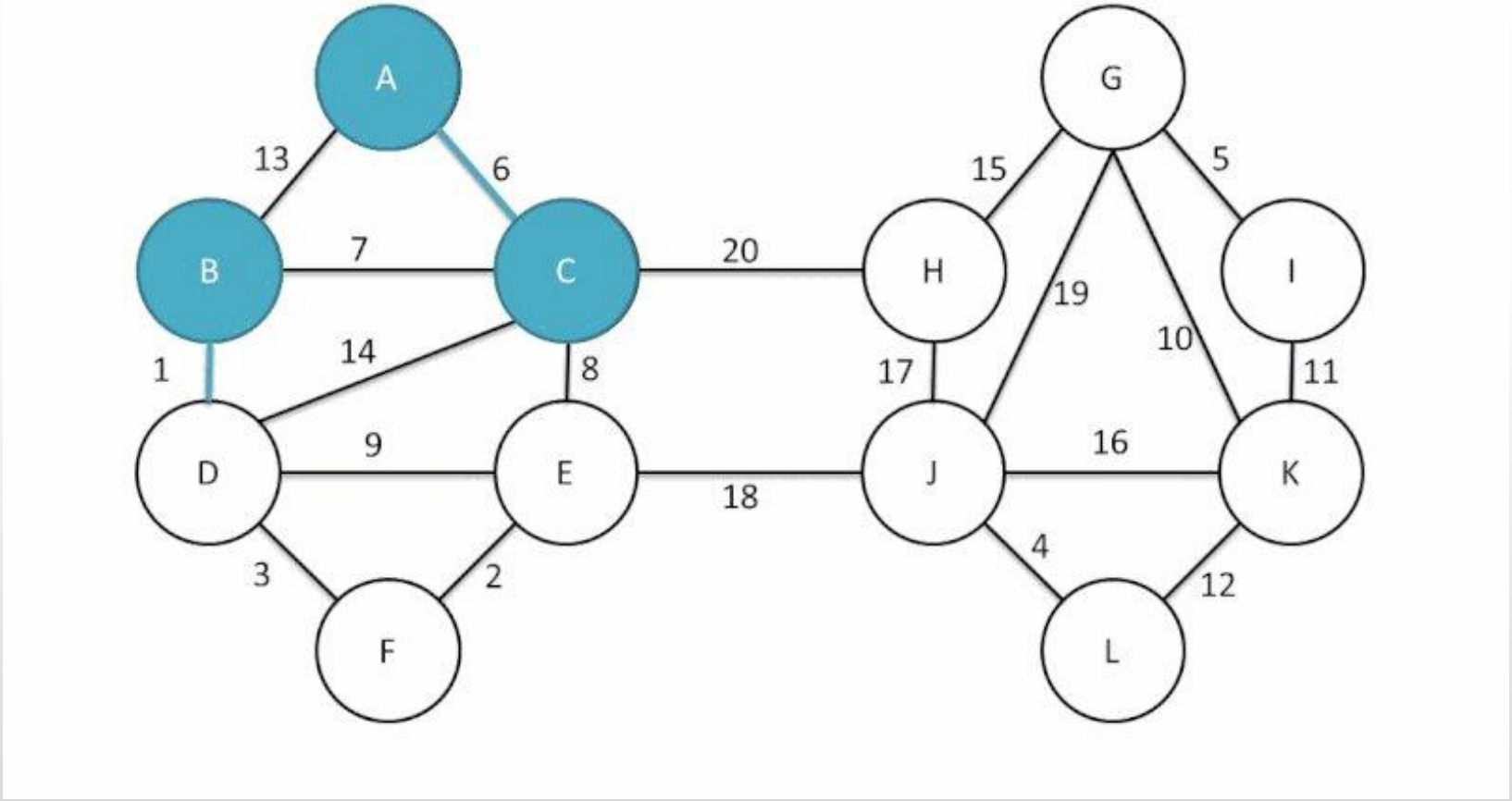
# Boruvka's Algorithm



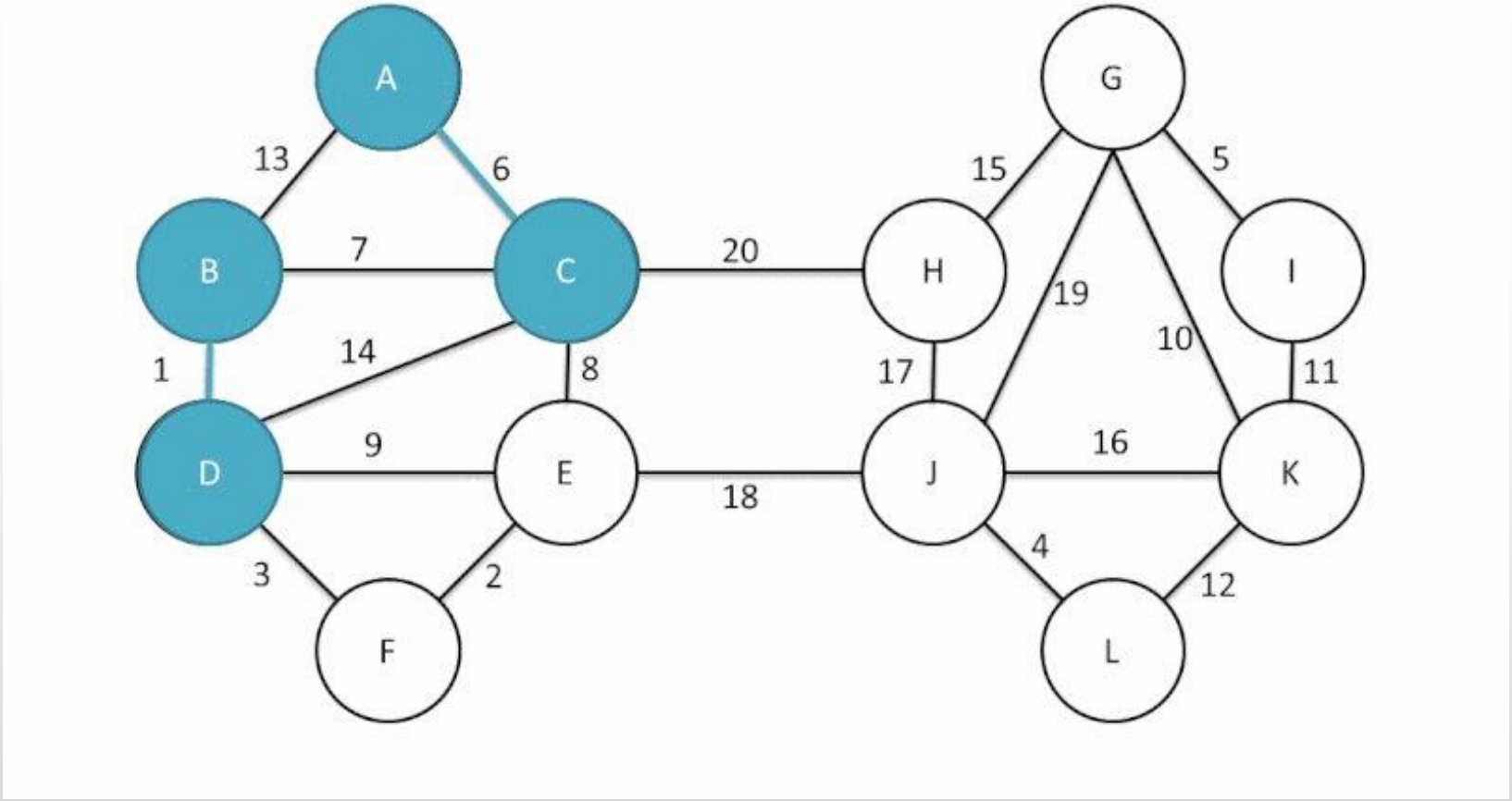
# Boruvka's Algorithm



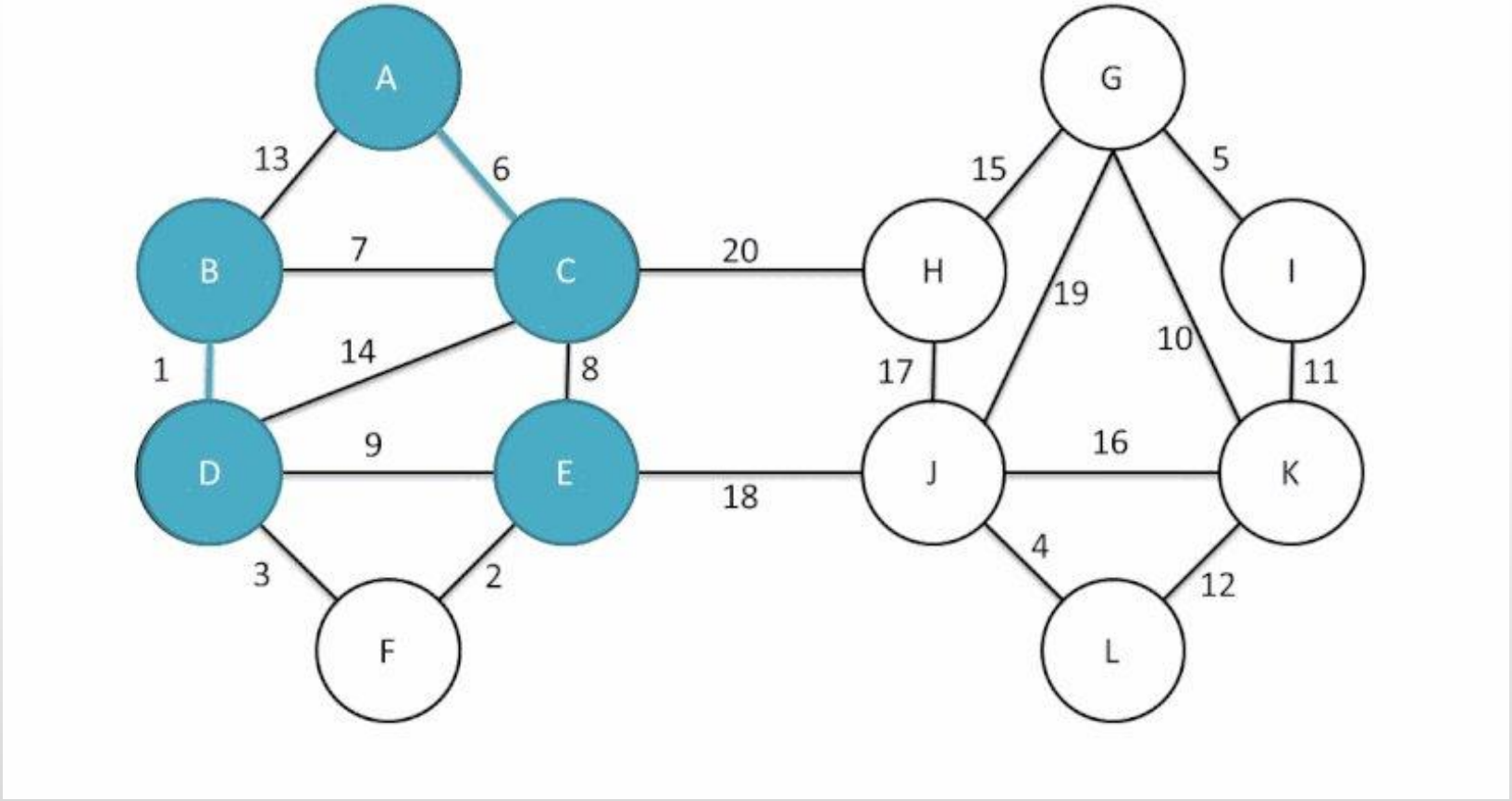
# Boruvka's Algorithm



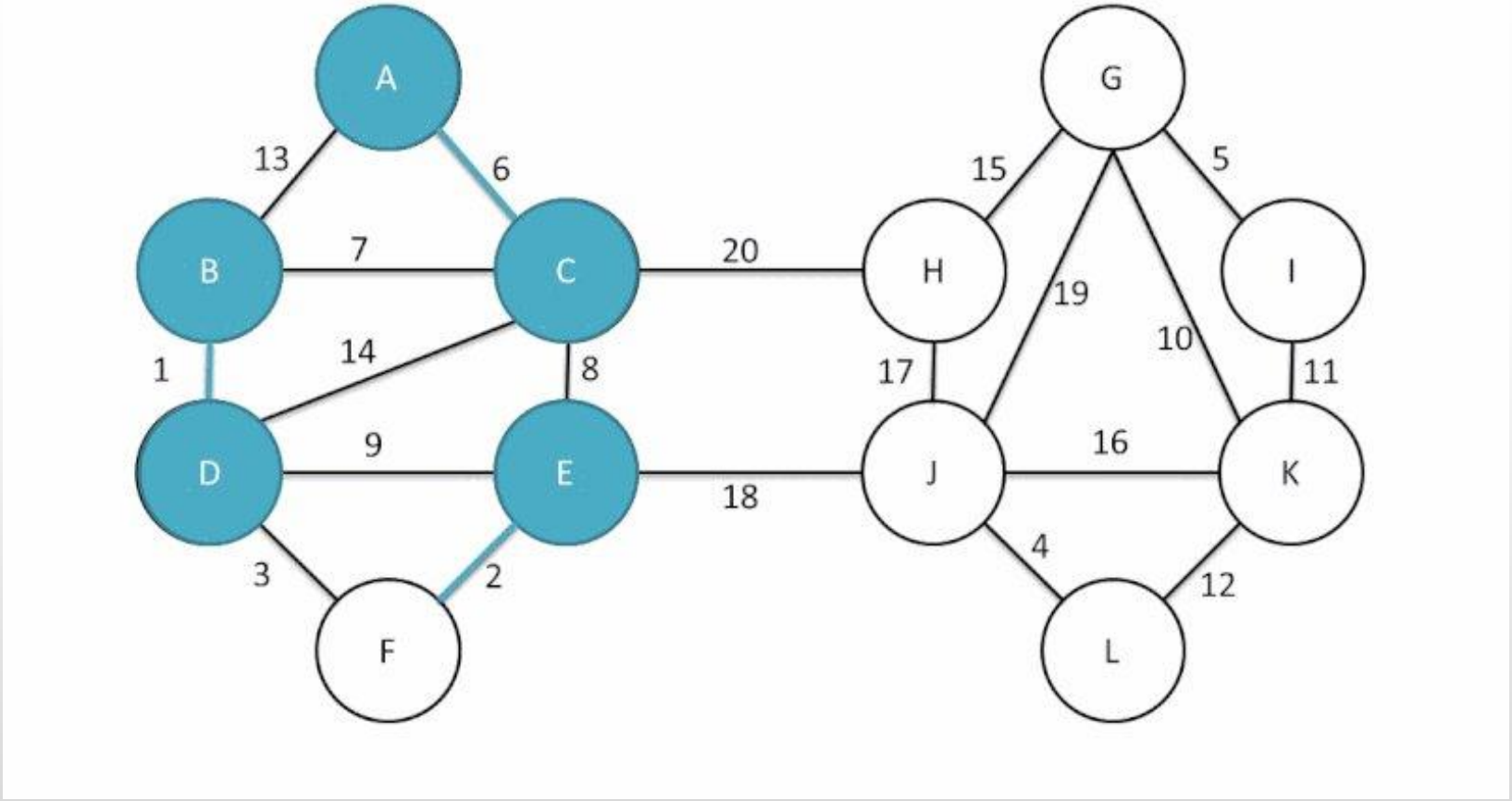
# Boruvka's Algorithm



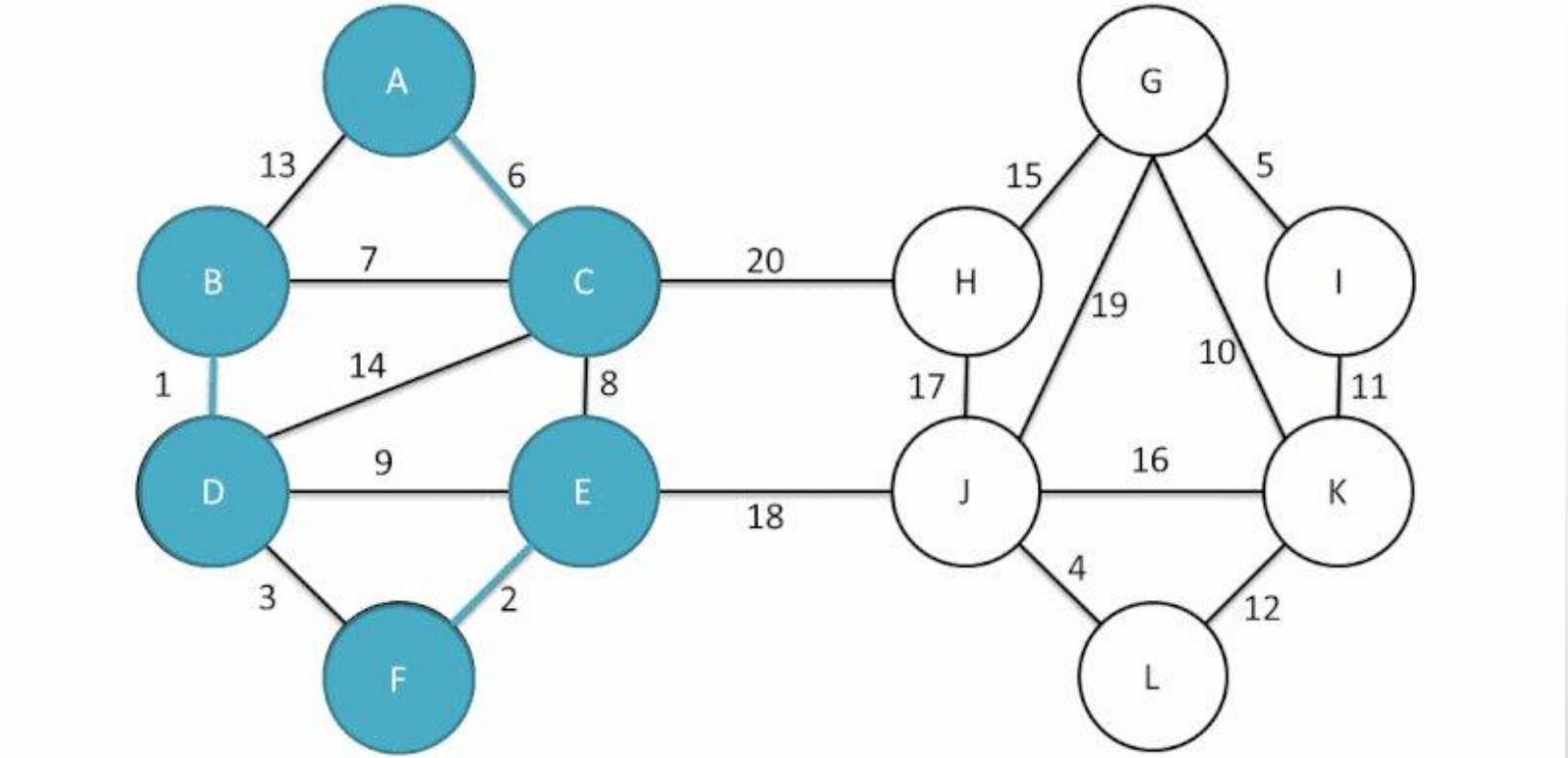
# Boruvka's Algorithm



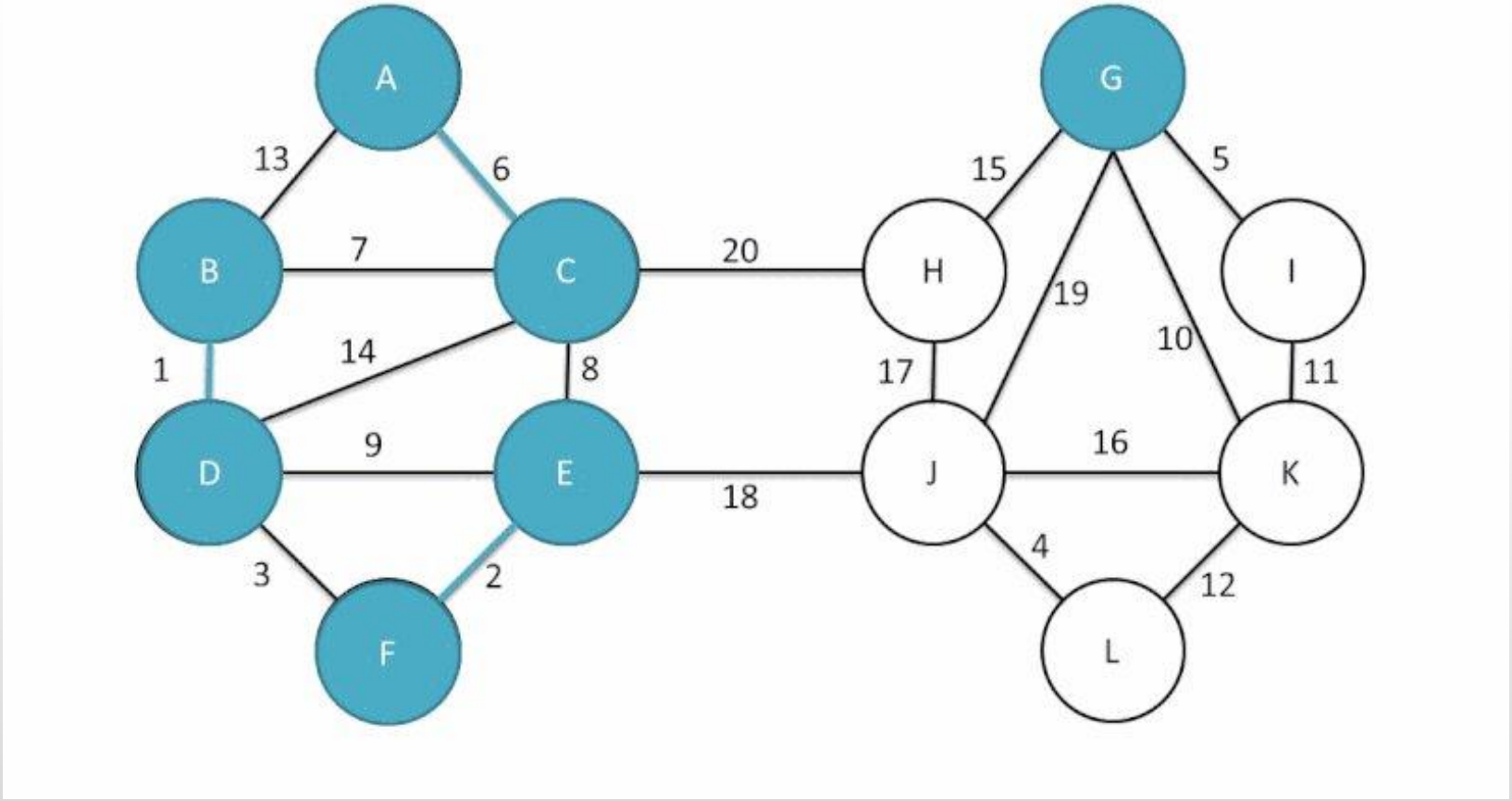
# Boruvka's Algorithm



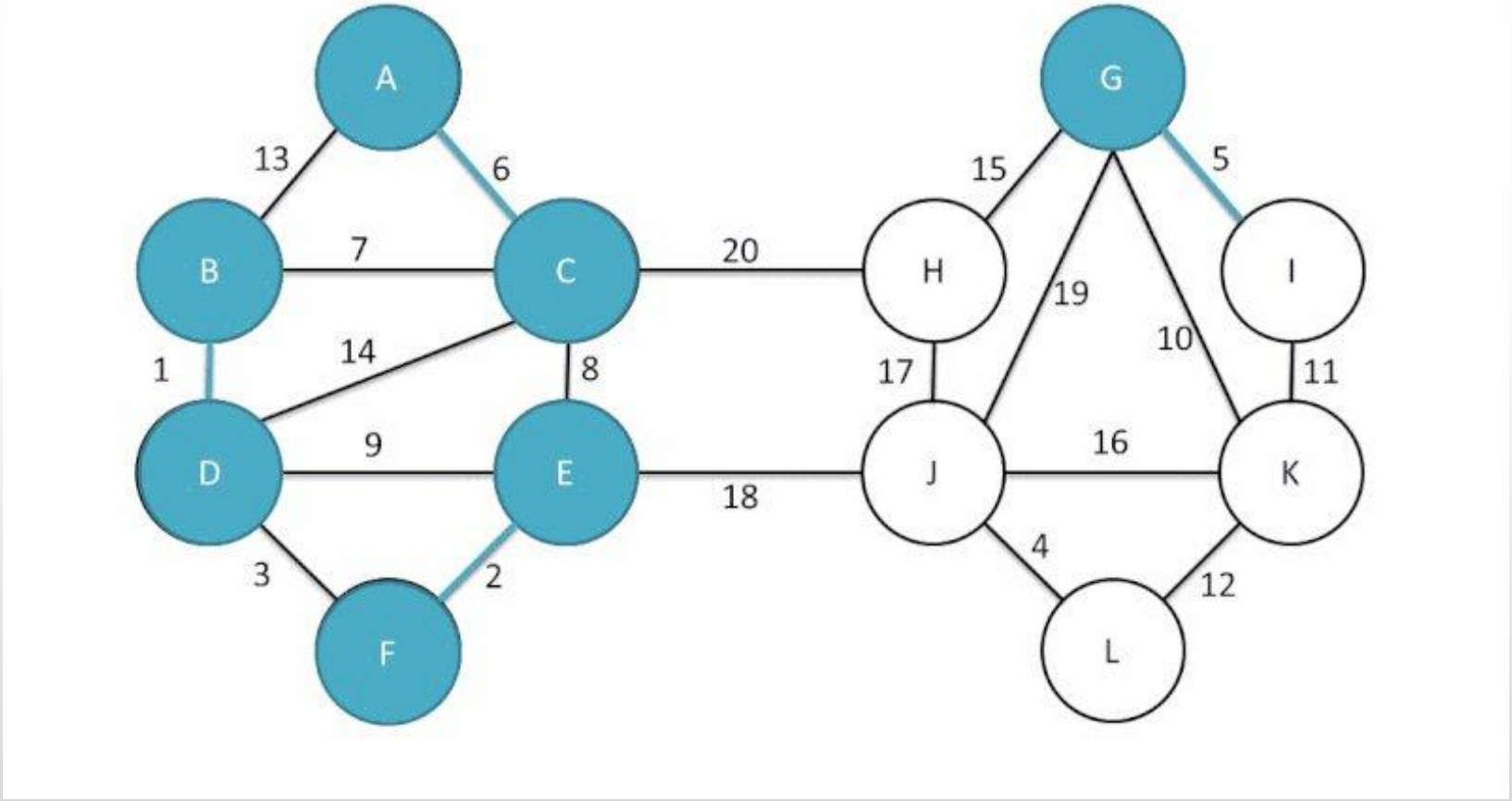
# Boruvka's Algorithm



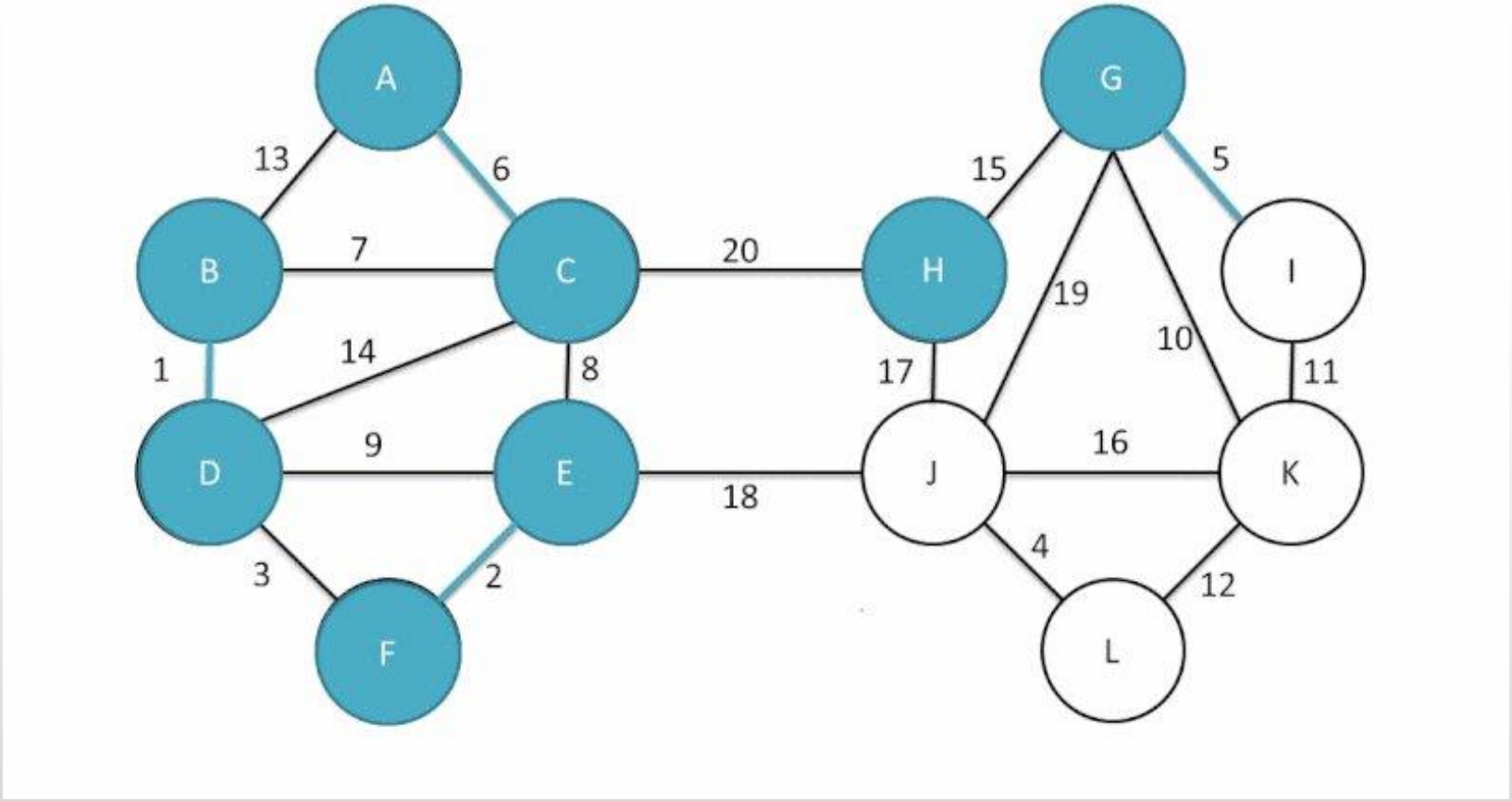
# Boruvka's Algorithm



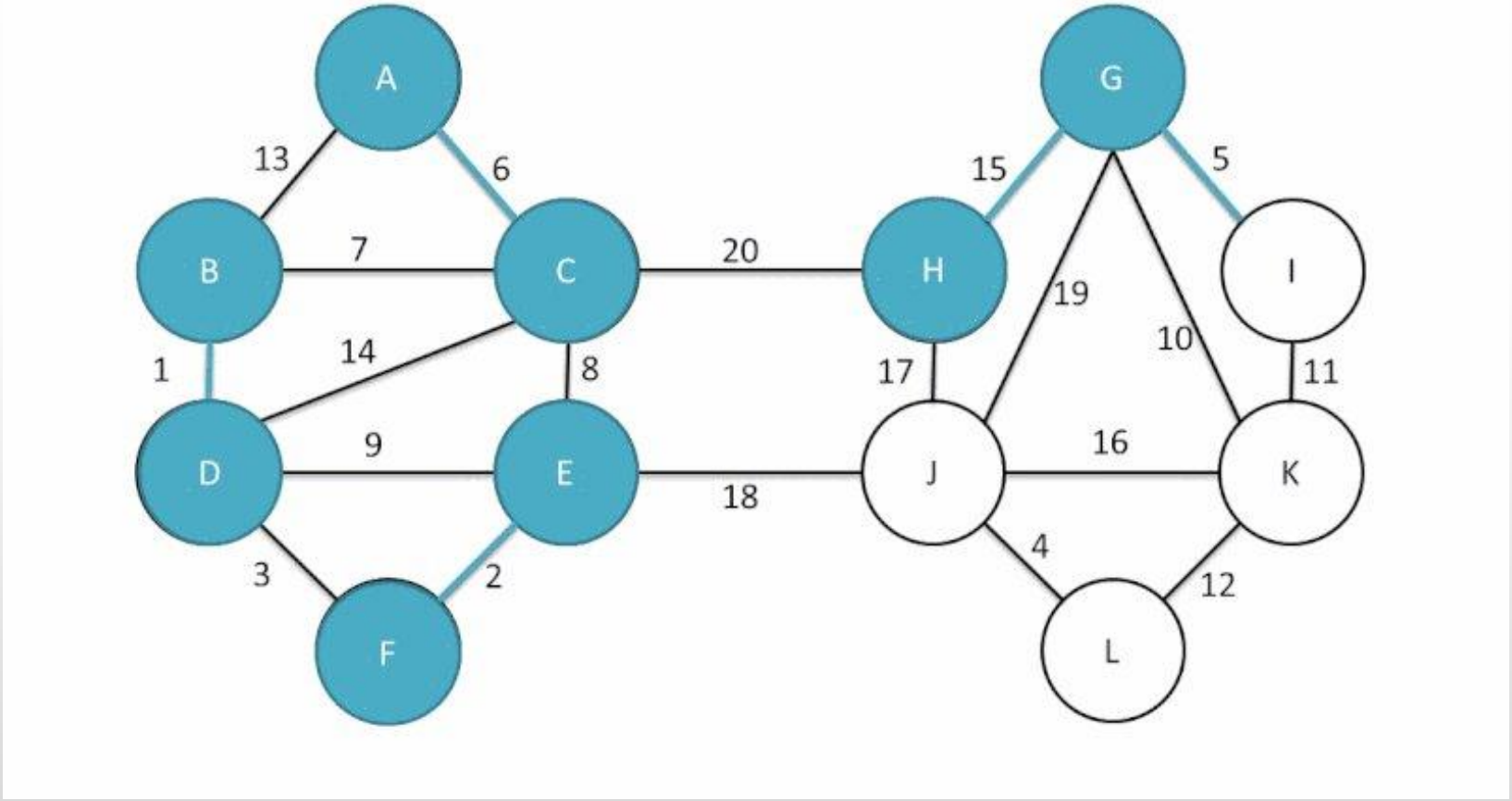
# Boruvka's Algorithm



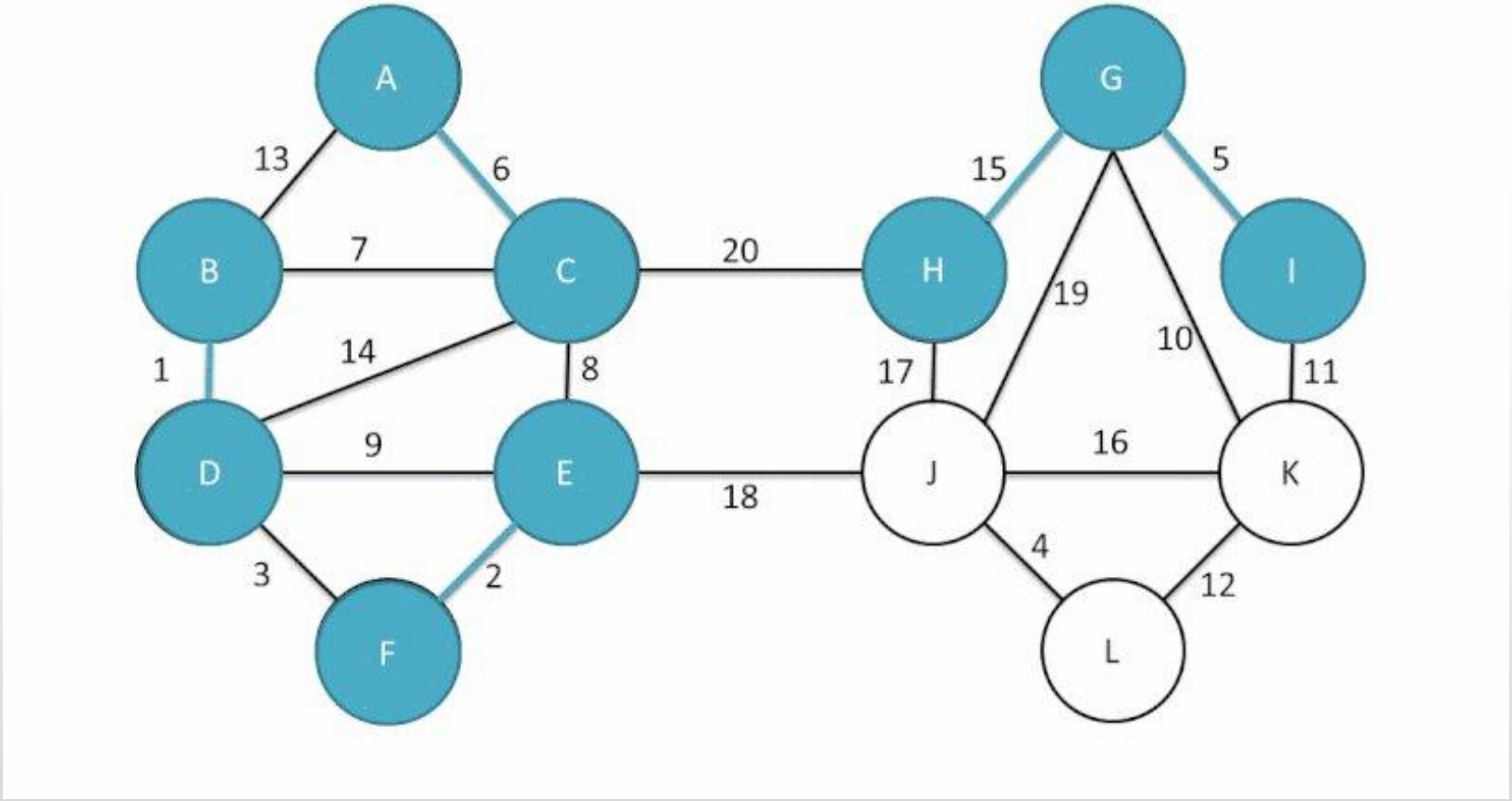
# Boruvka's Algorithm



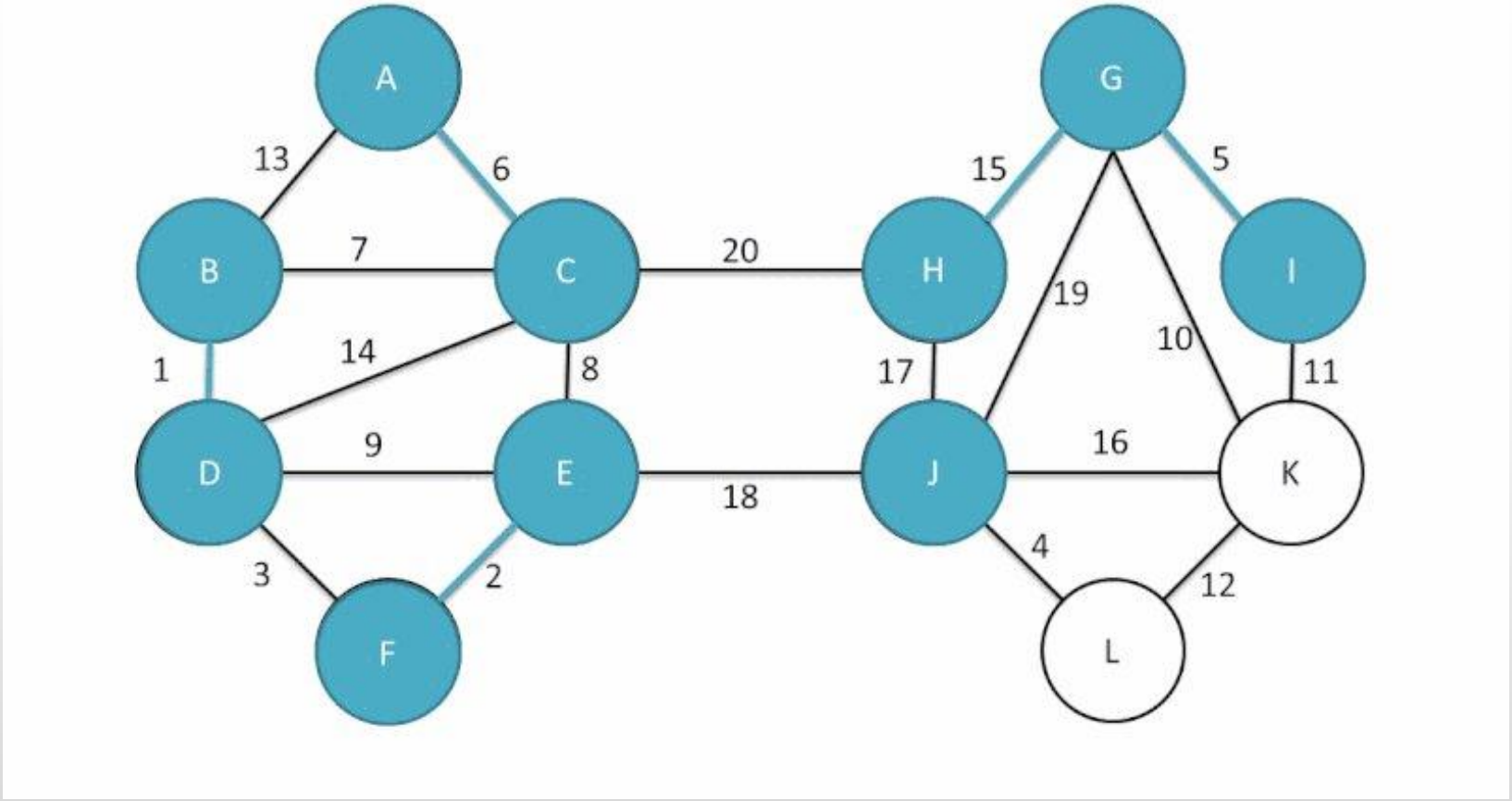
# Boruvka's Algorithm



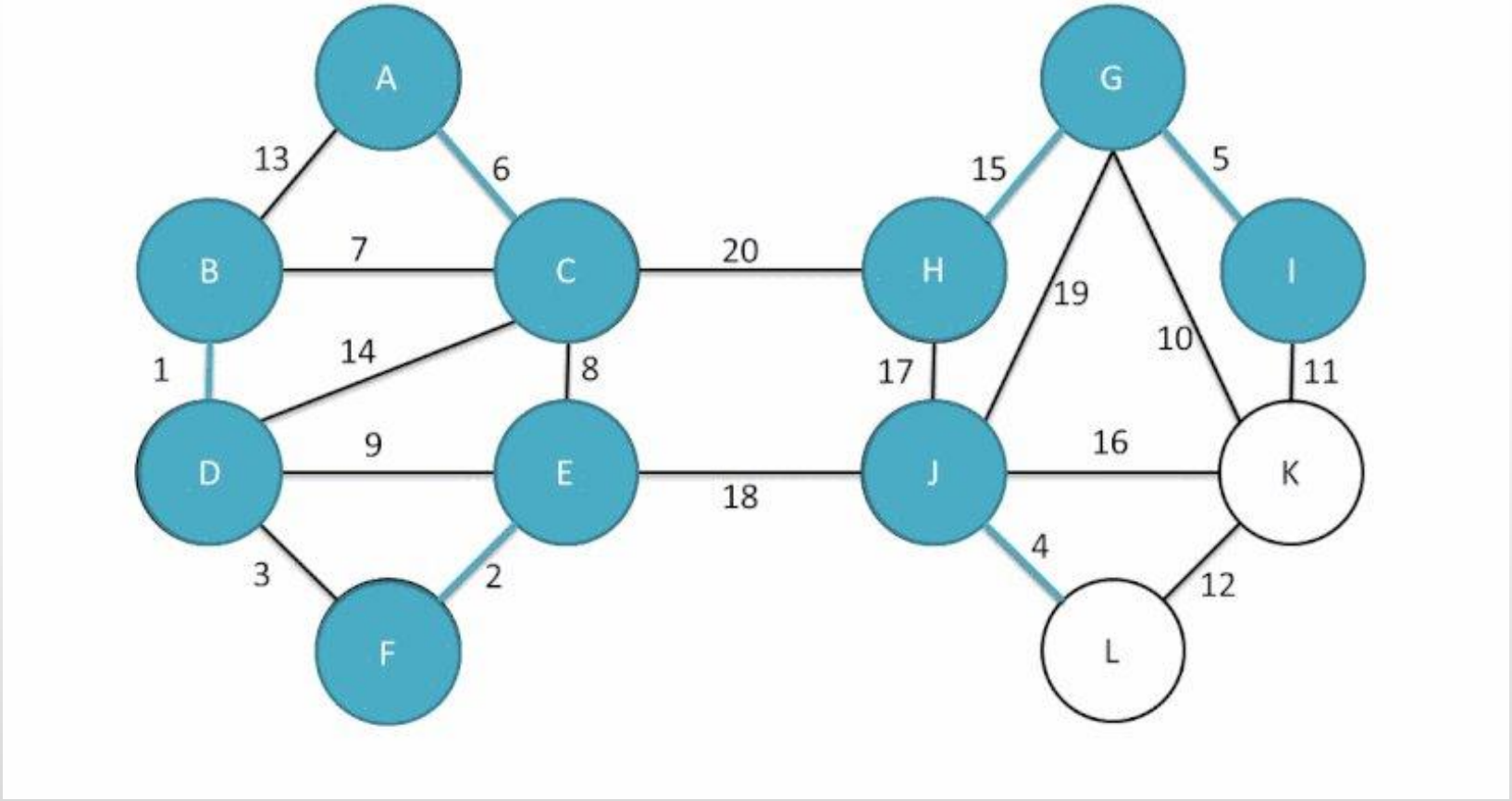
# Boruvka's Algorithm



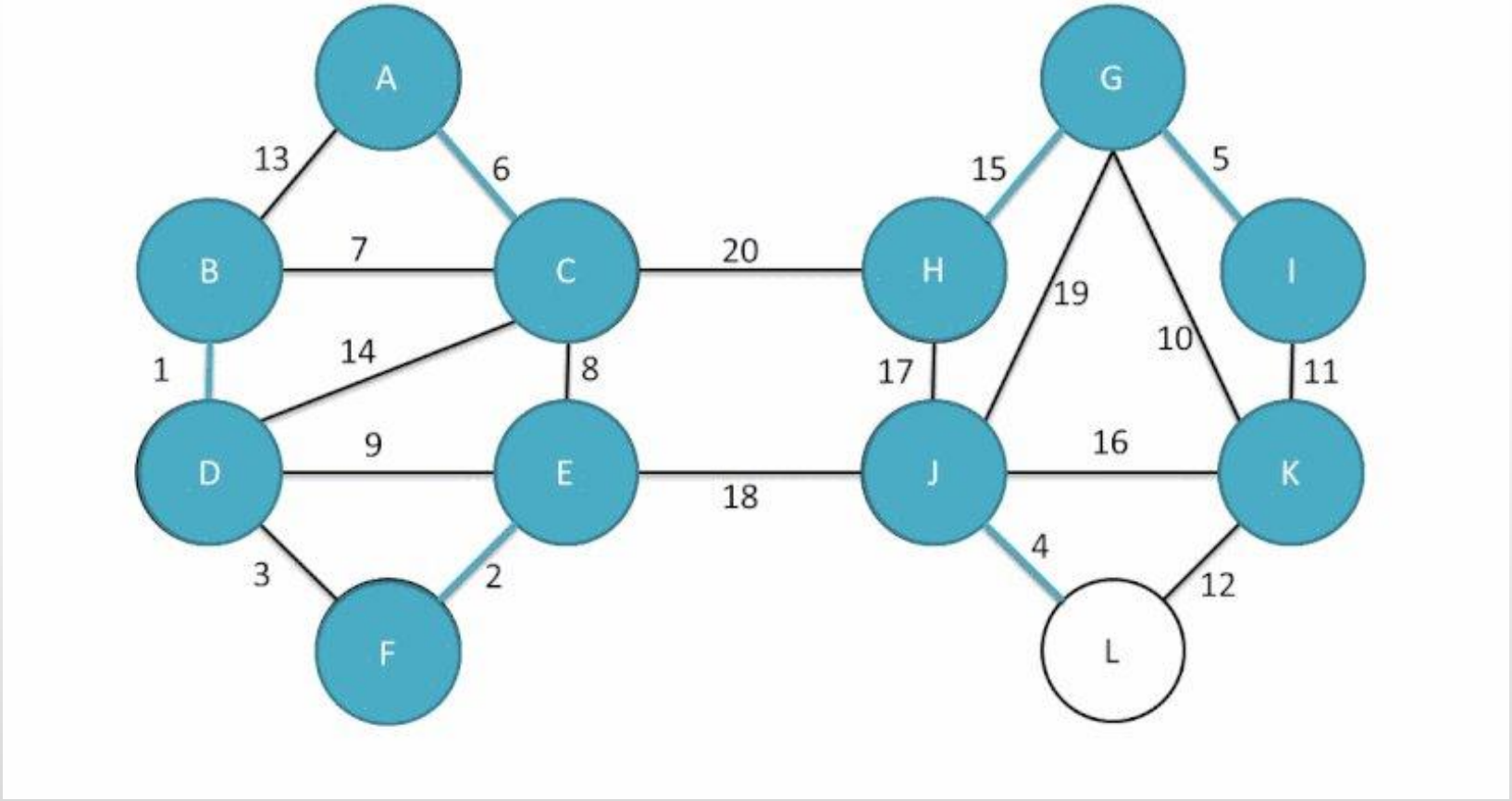
# Boruvka's Algorithm



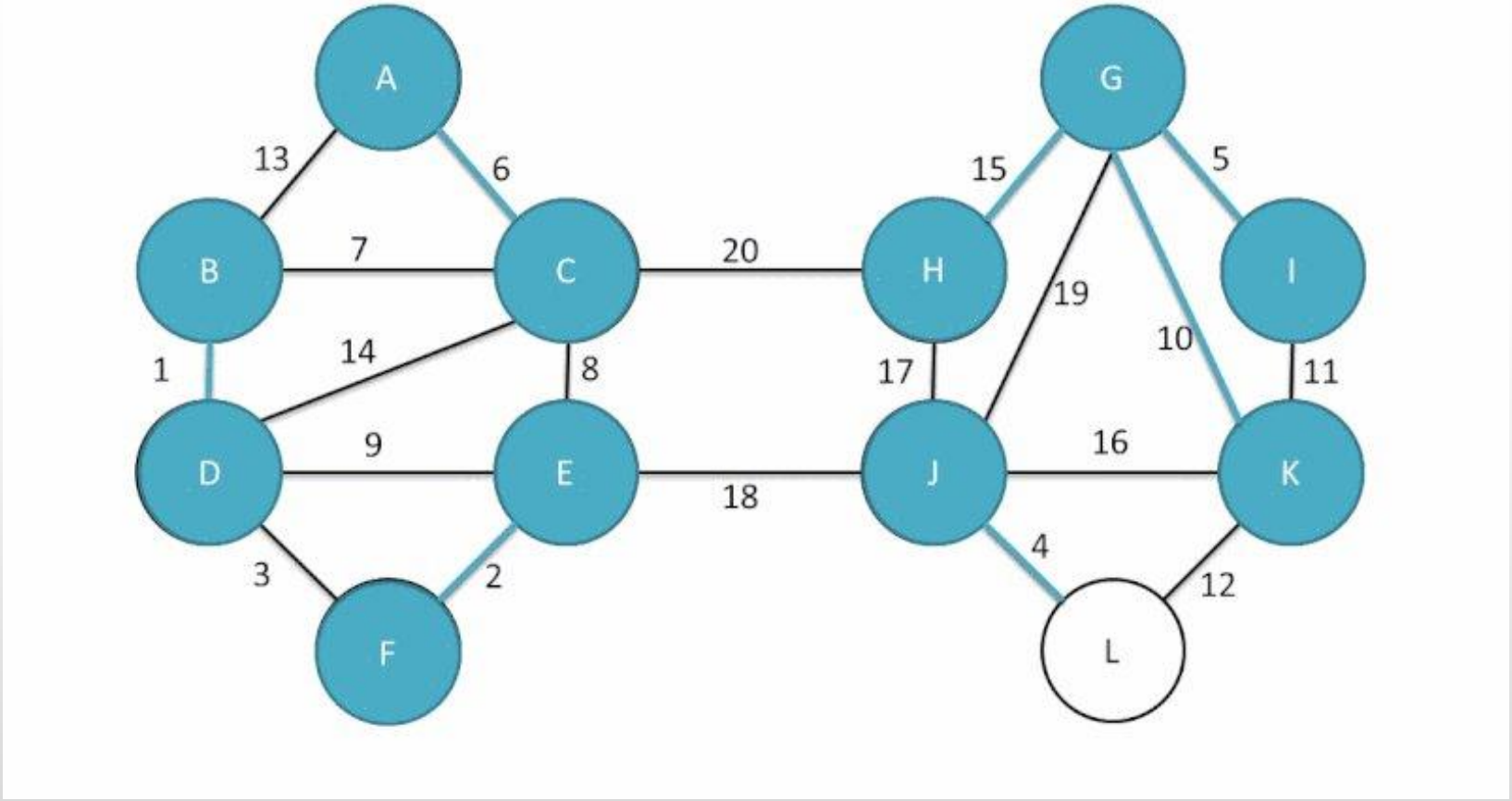
# Boruvka's Algorithm



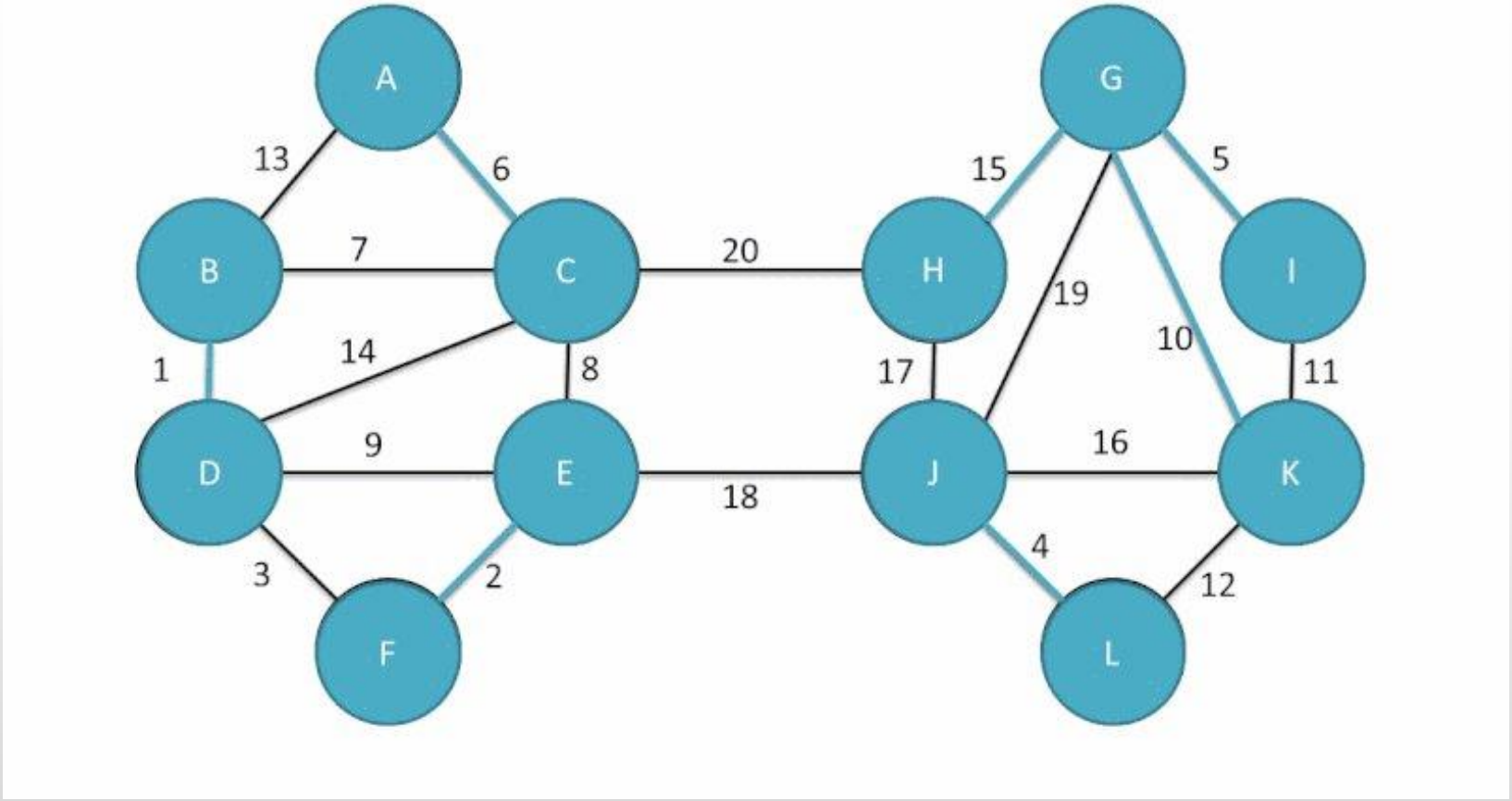
# Boruvka's Algorithm



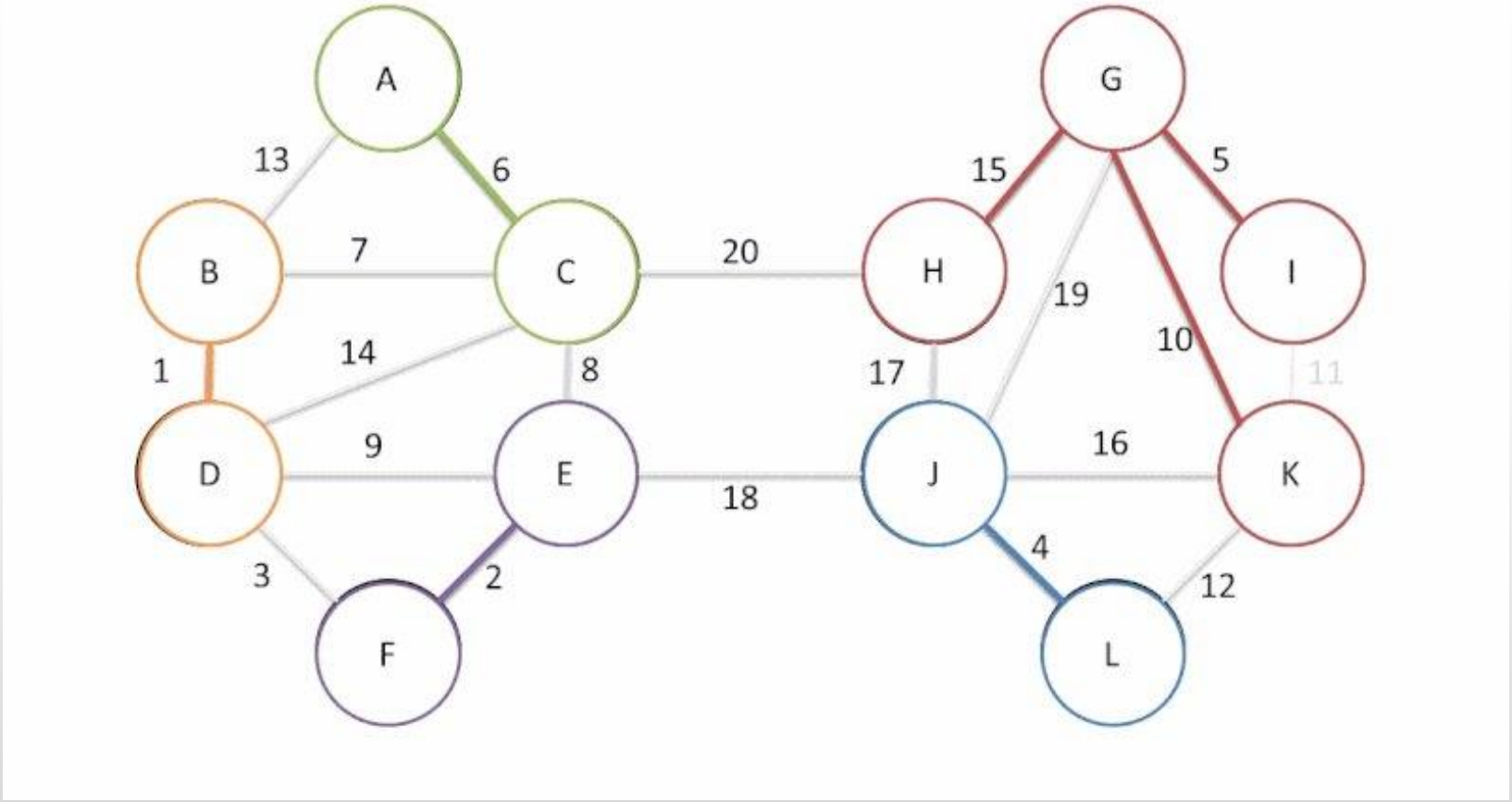
# Boruvka's Algorithm



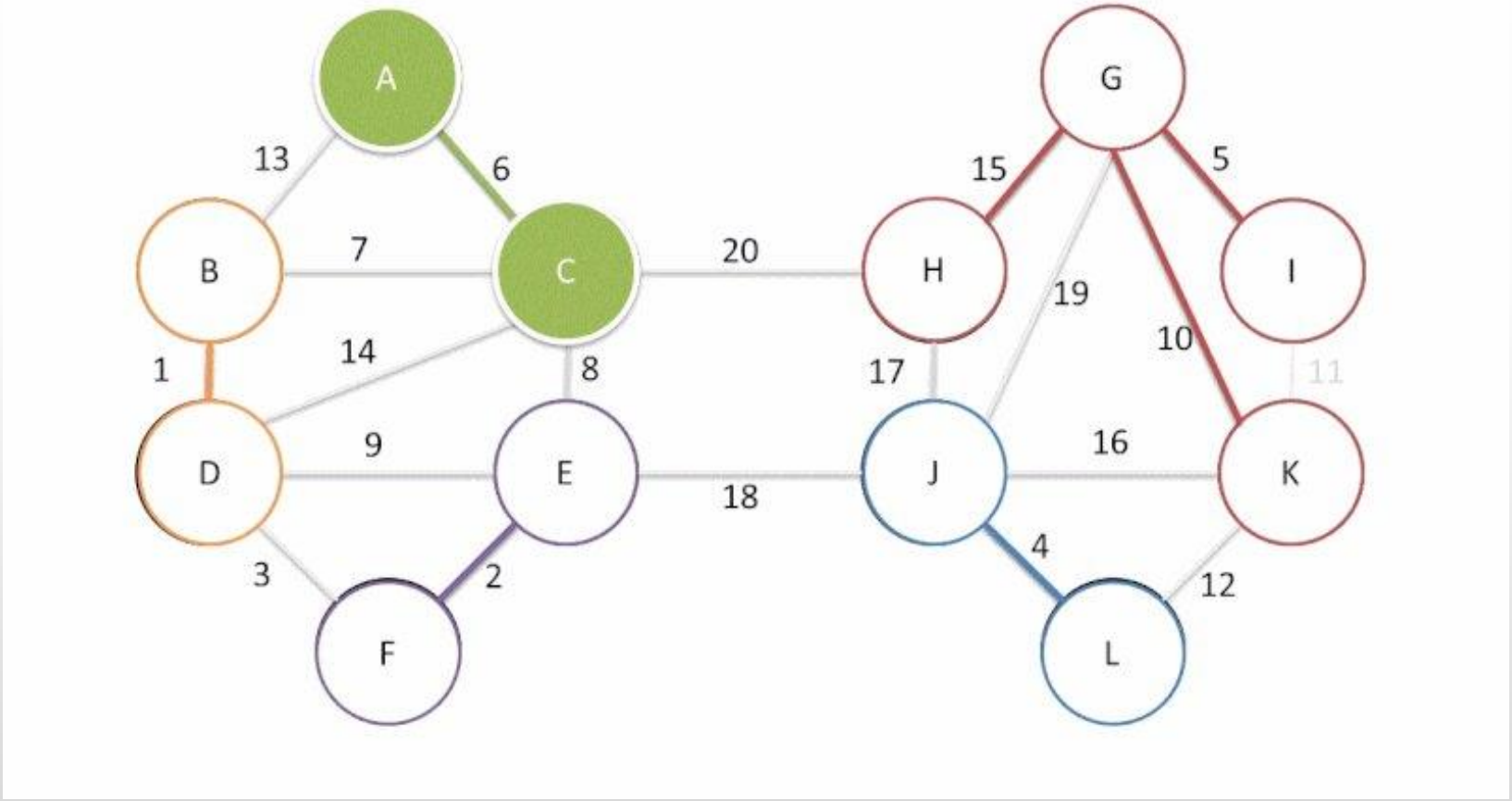
# Boruvka's Algorithm



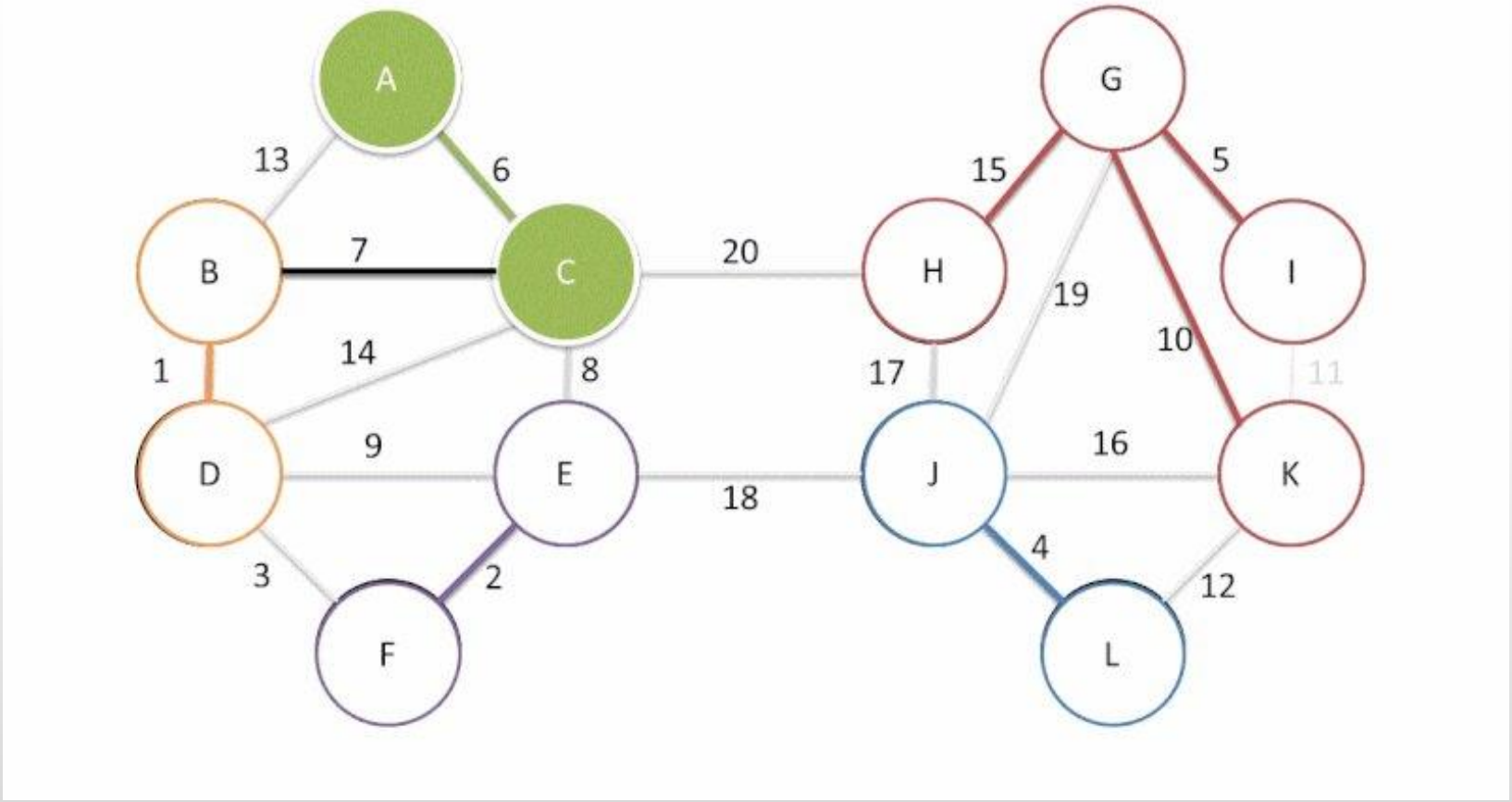
# Boruvka's Algorithm



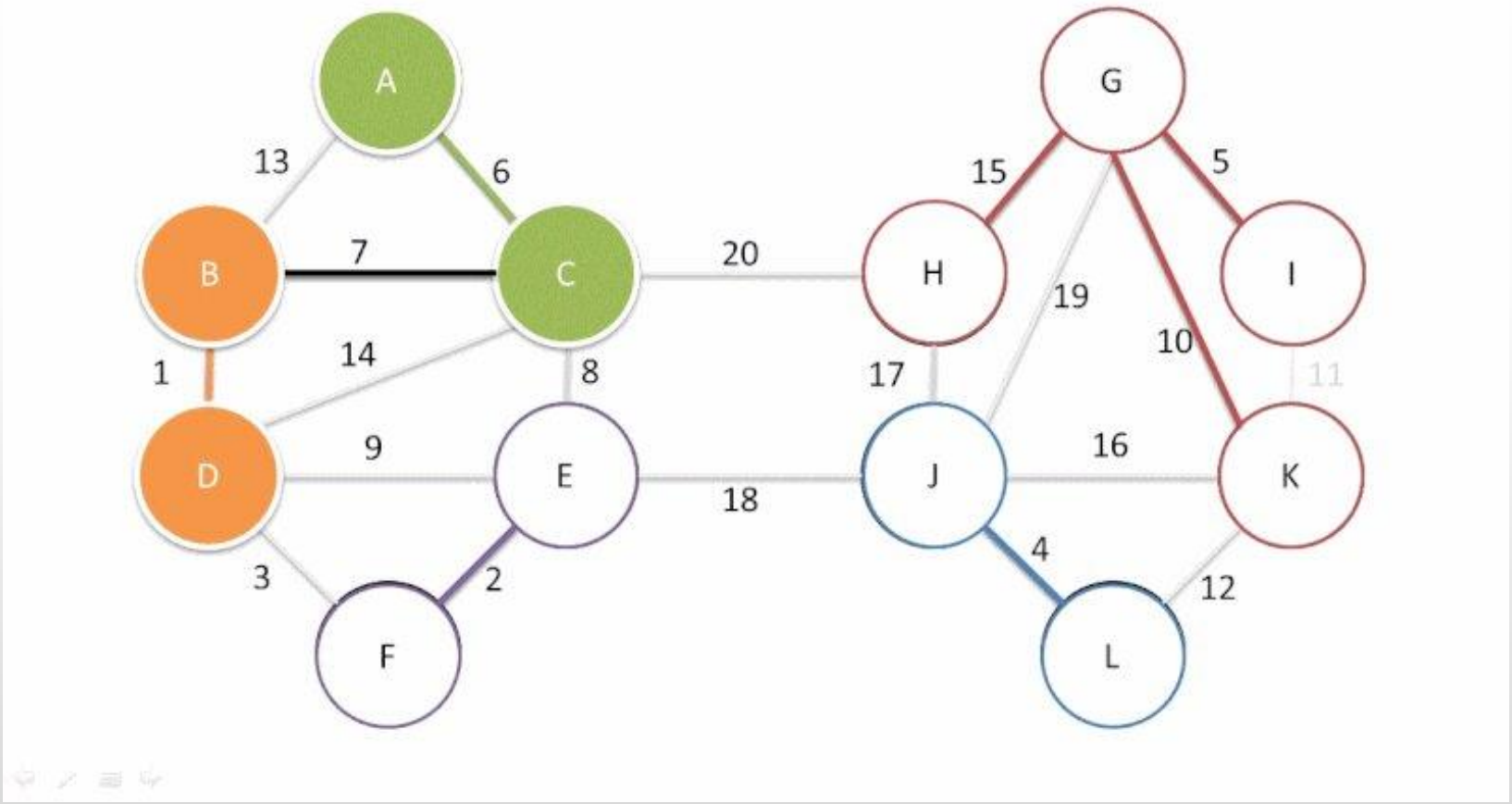
# Boruvka's Algorithm



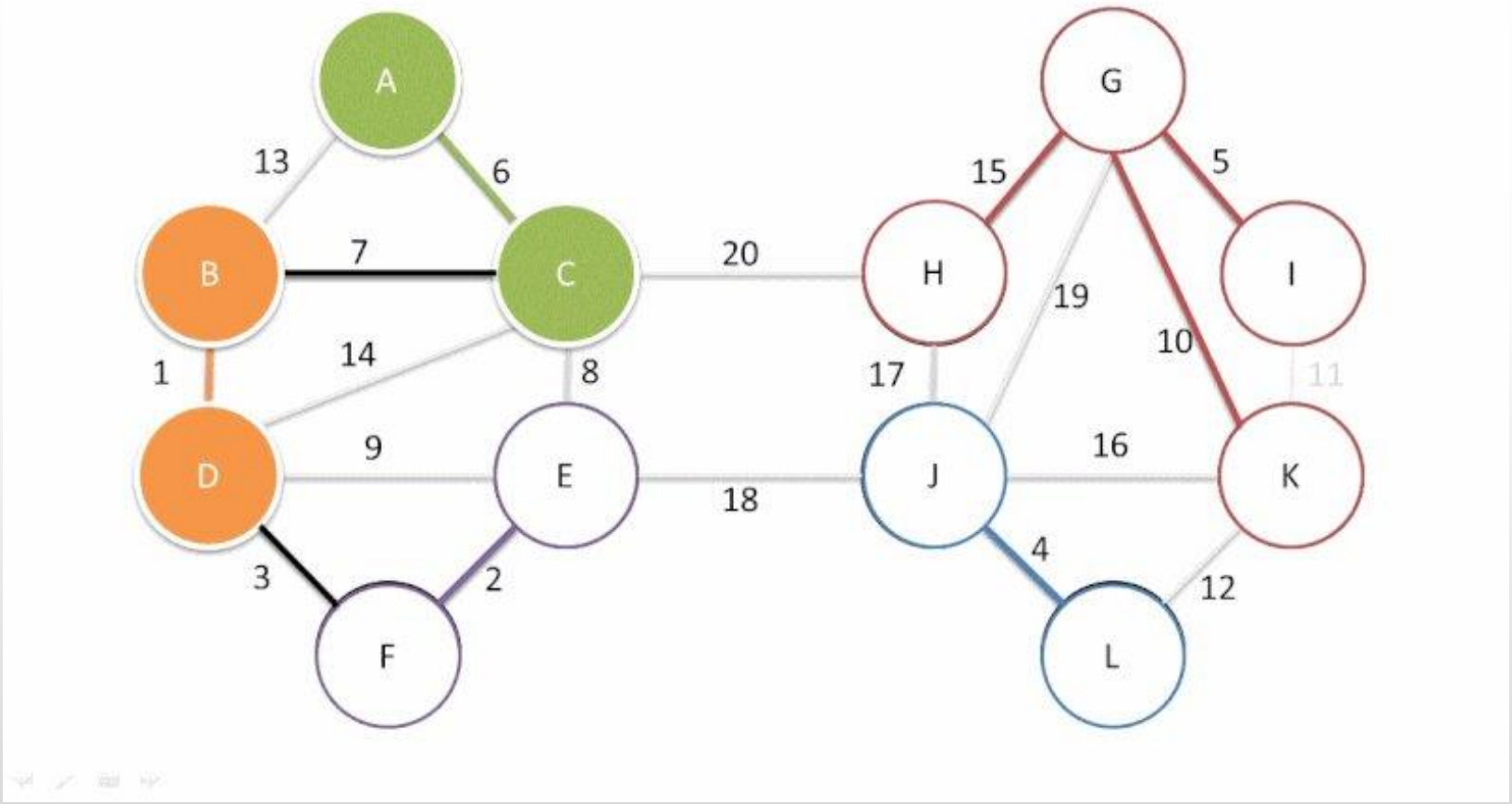
# Boruvka's Algorithm



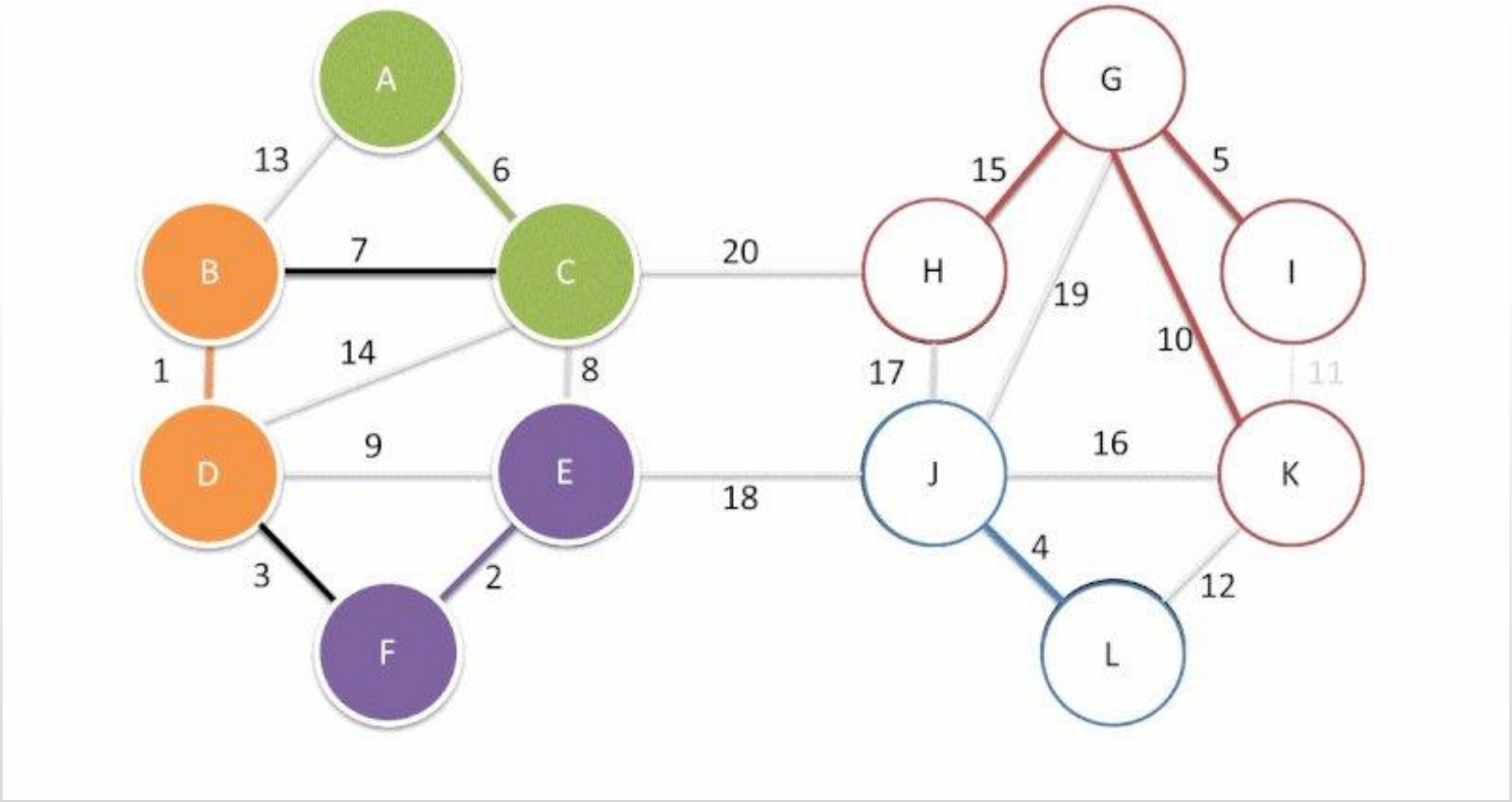
# Boruvka's Algorithm



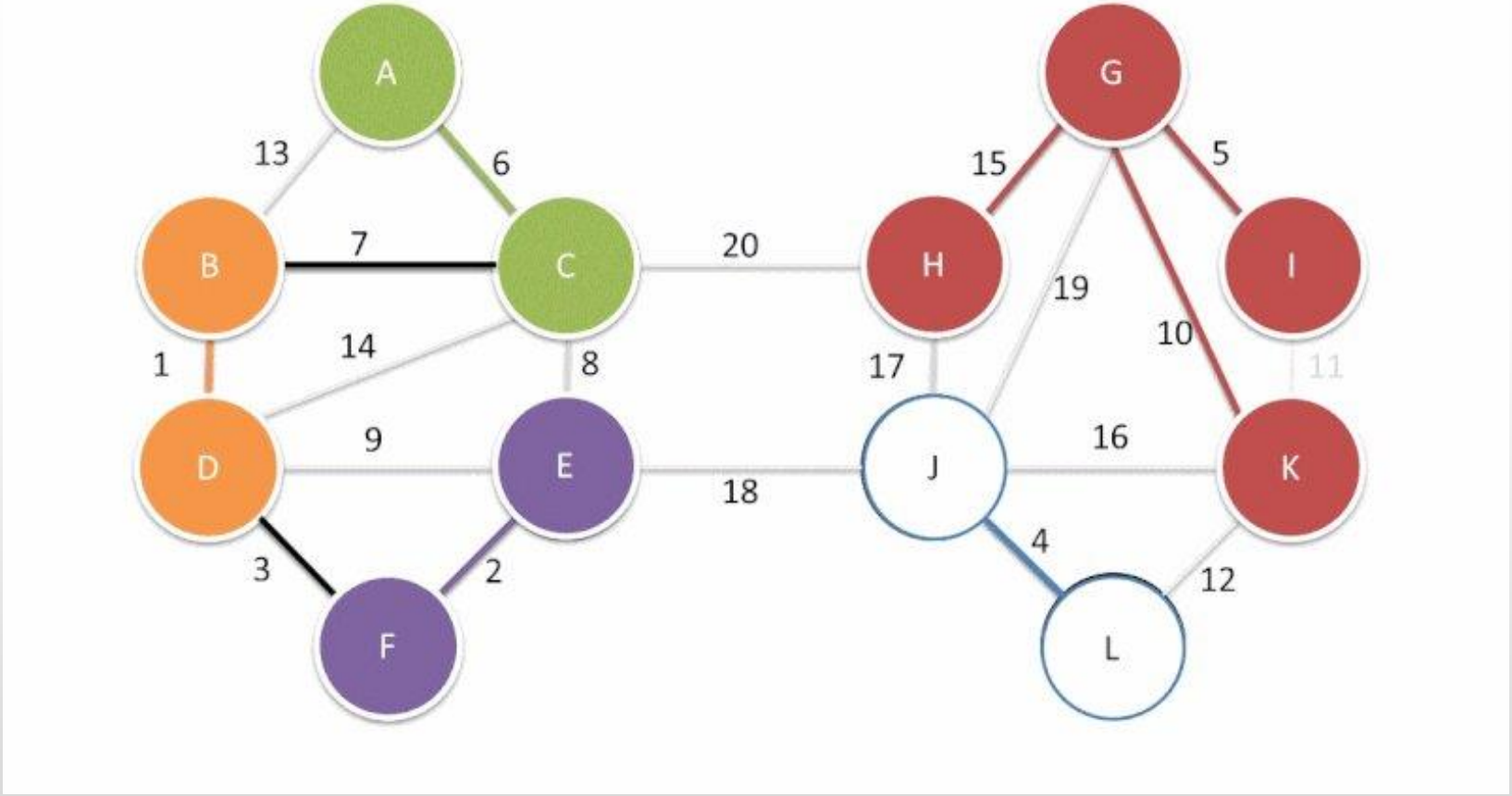
# Boruvka's Algorithm



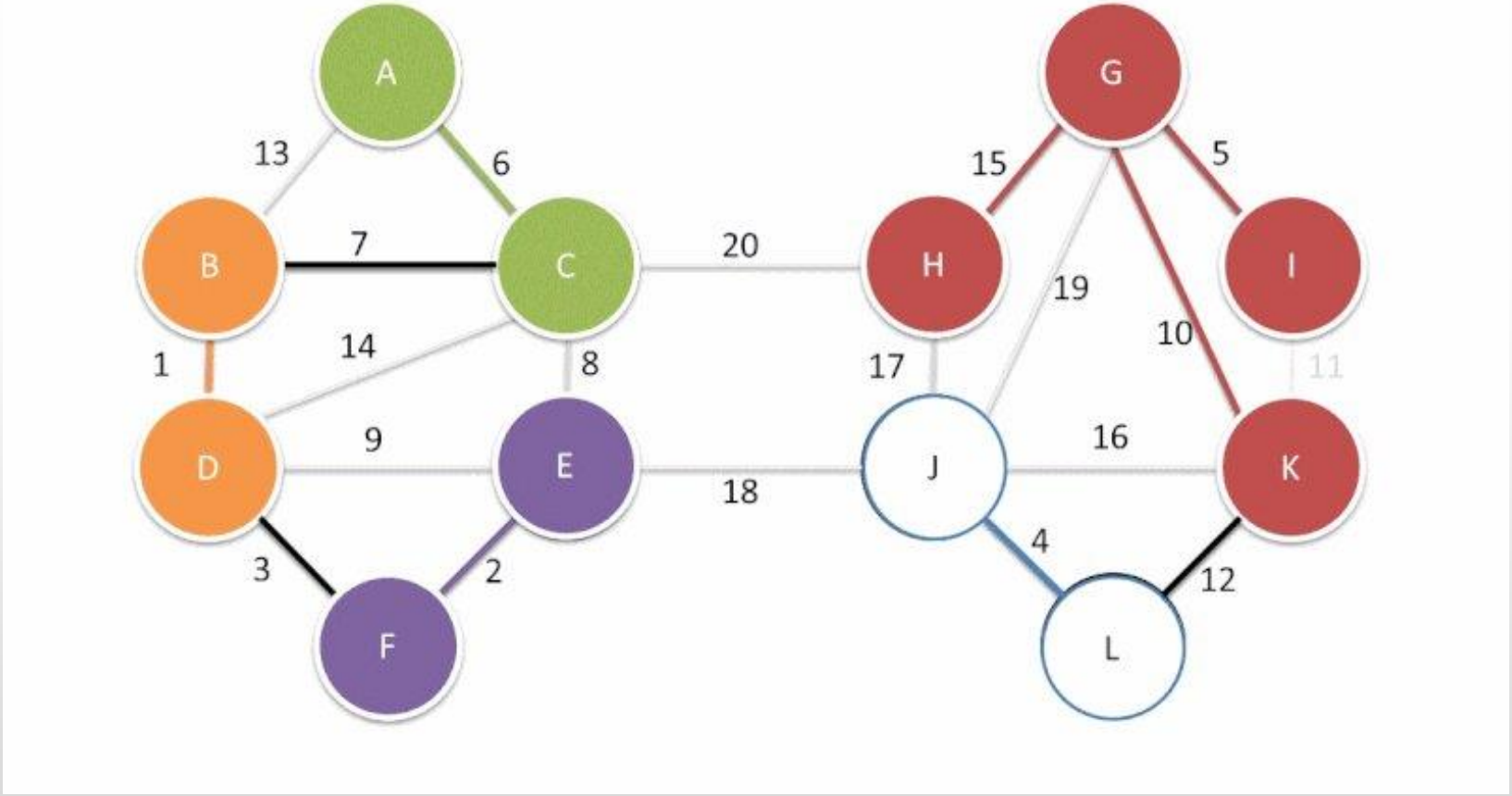
# Boruvka's Algorithm



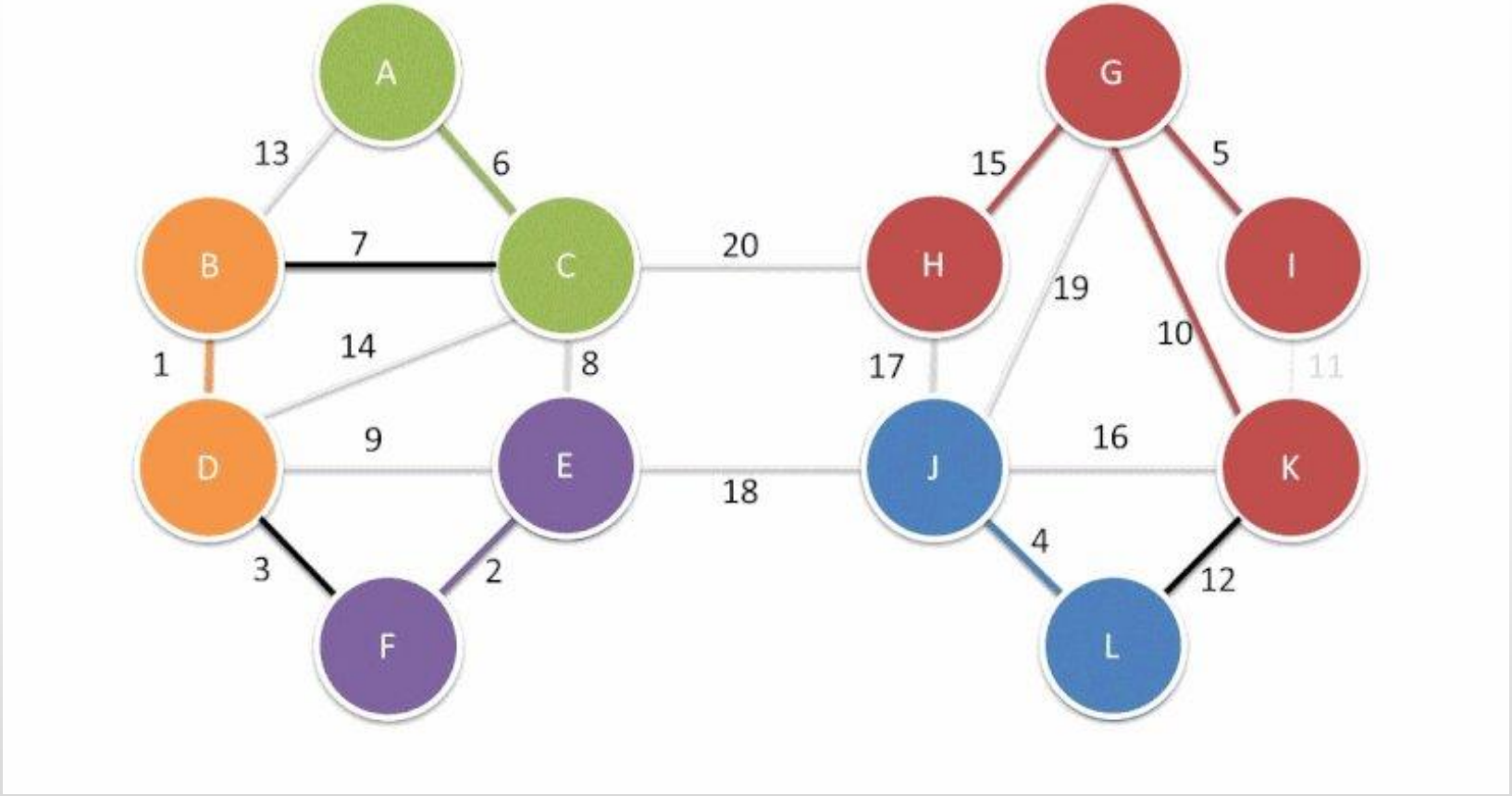
# Boruvka's Algorithm



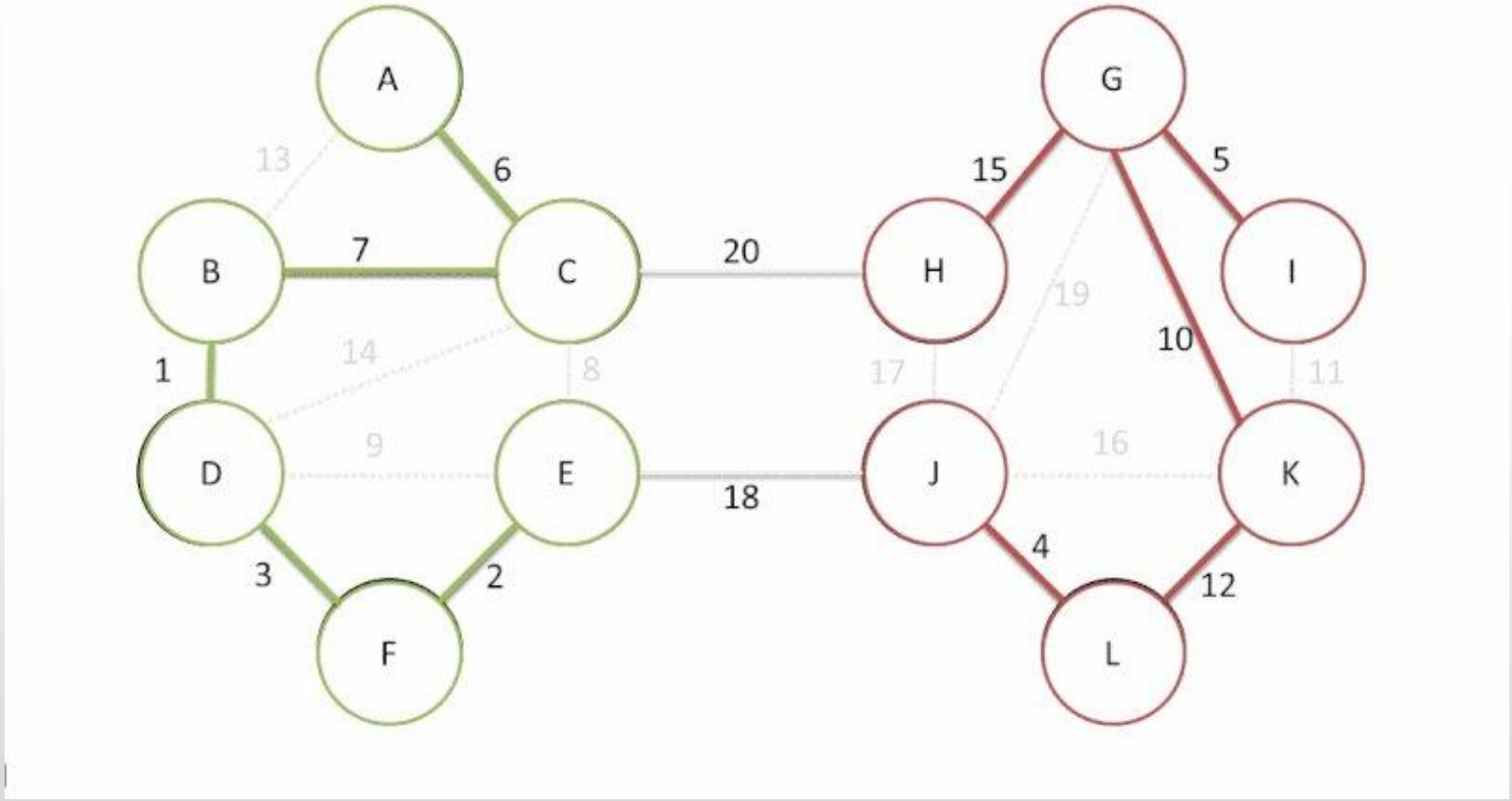
# Boruvka's Algorithm



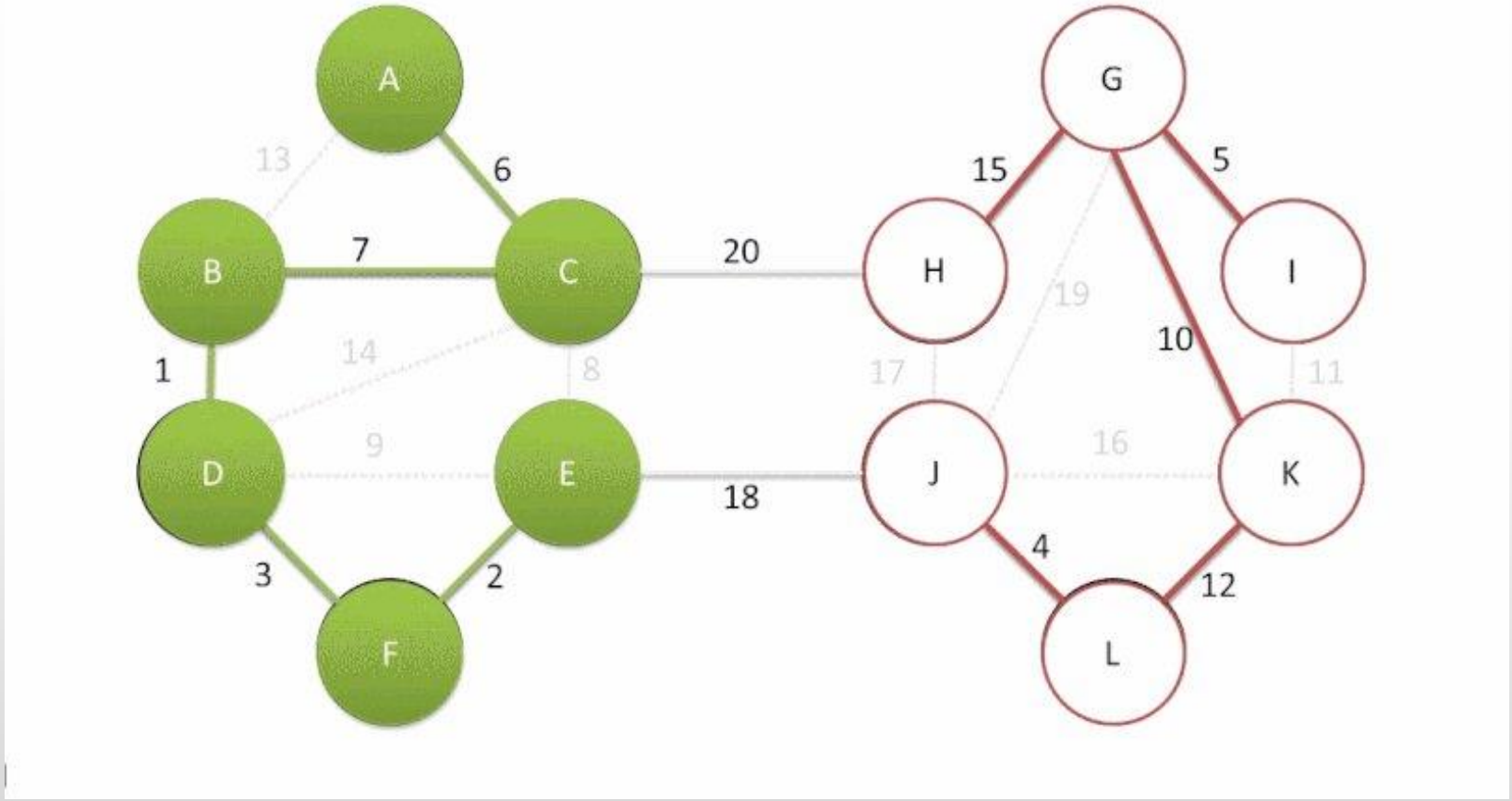
# Boruvka's Algorithm



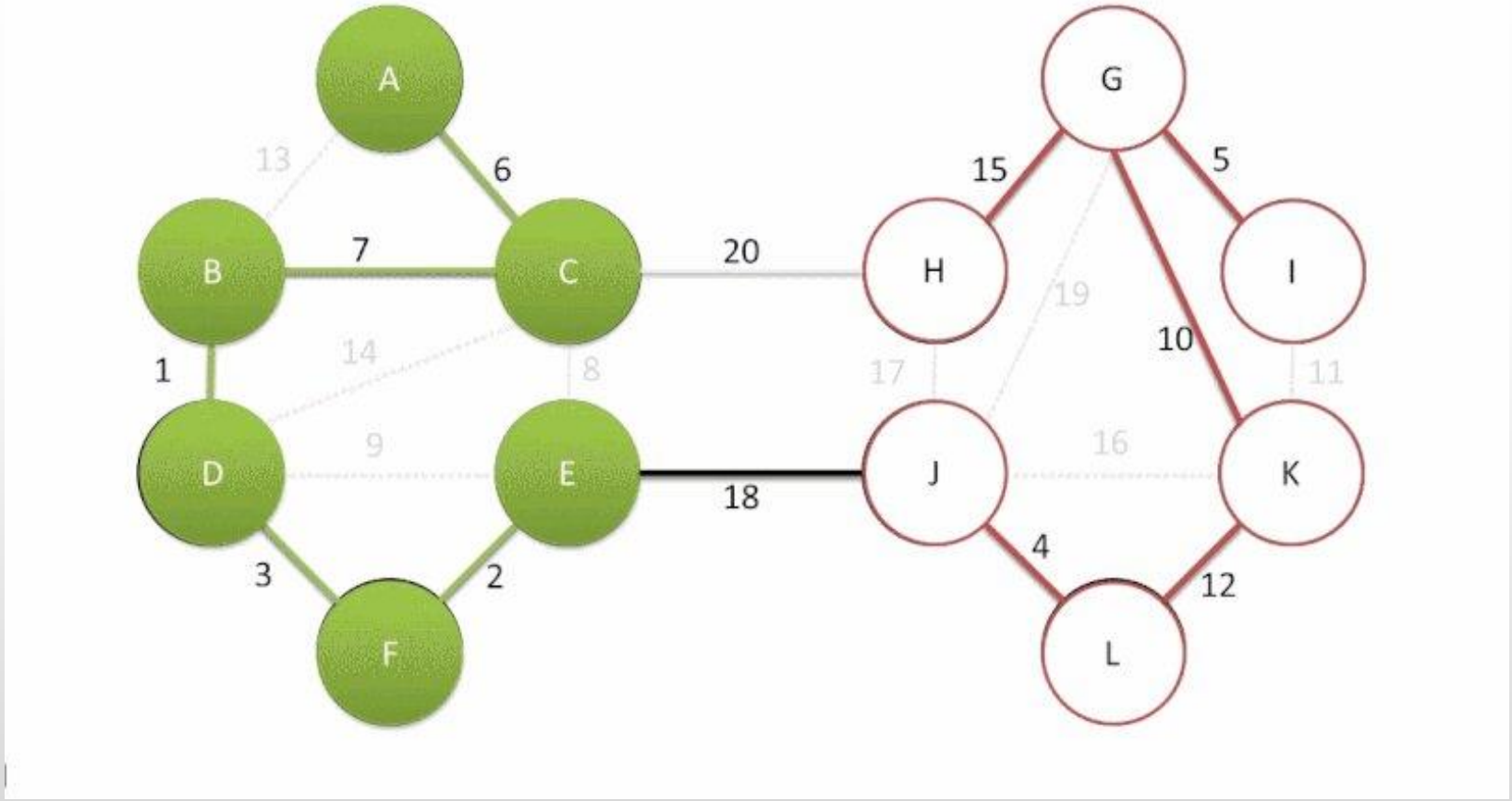
# Boruvka's Algorithm



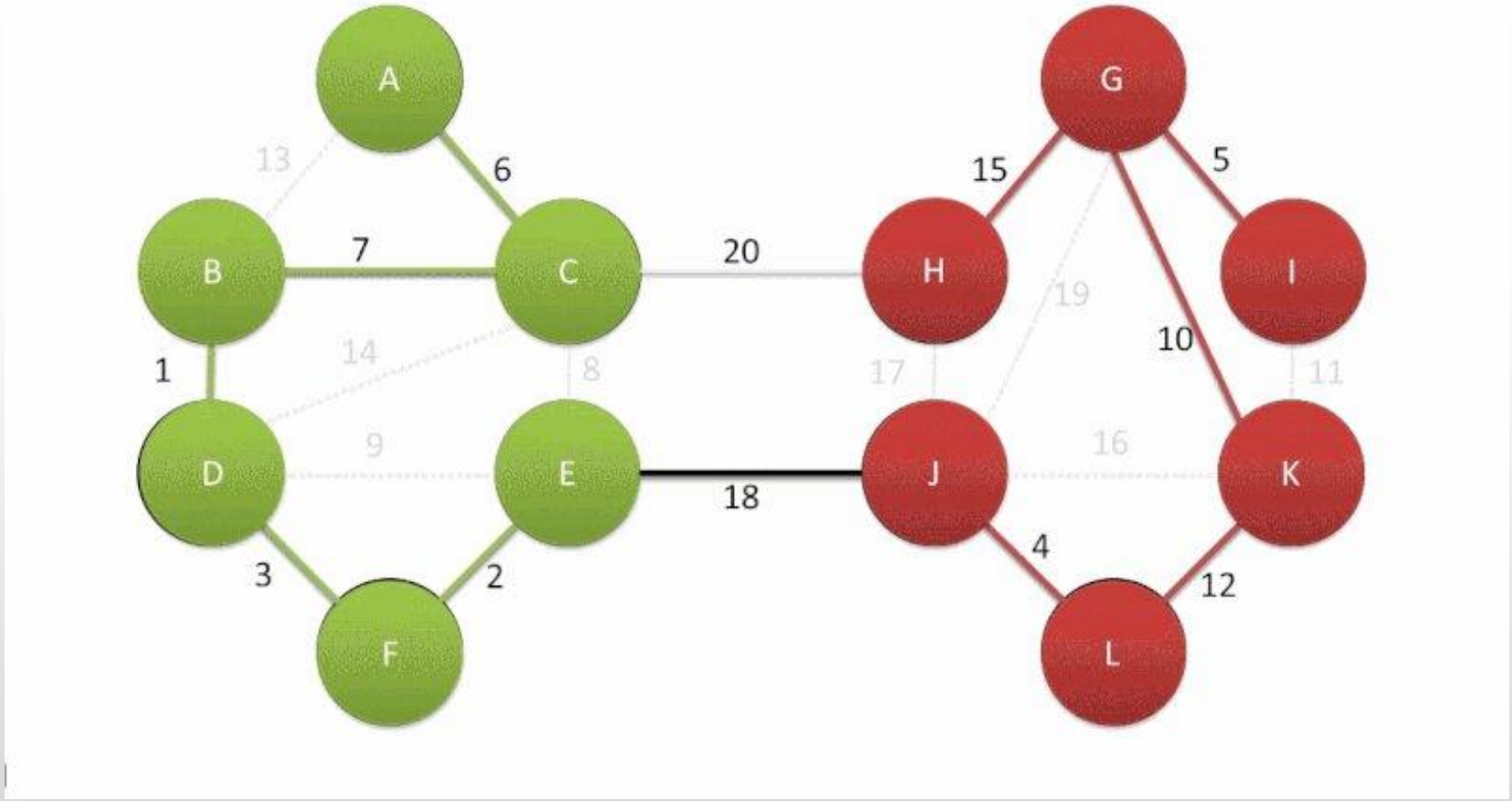
# Boruvka's Algorithm



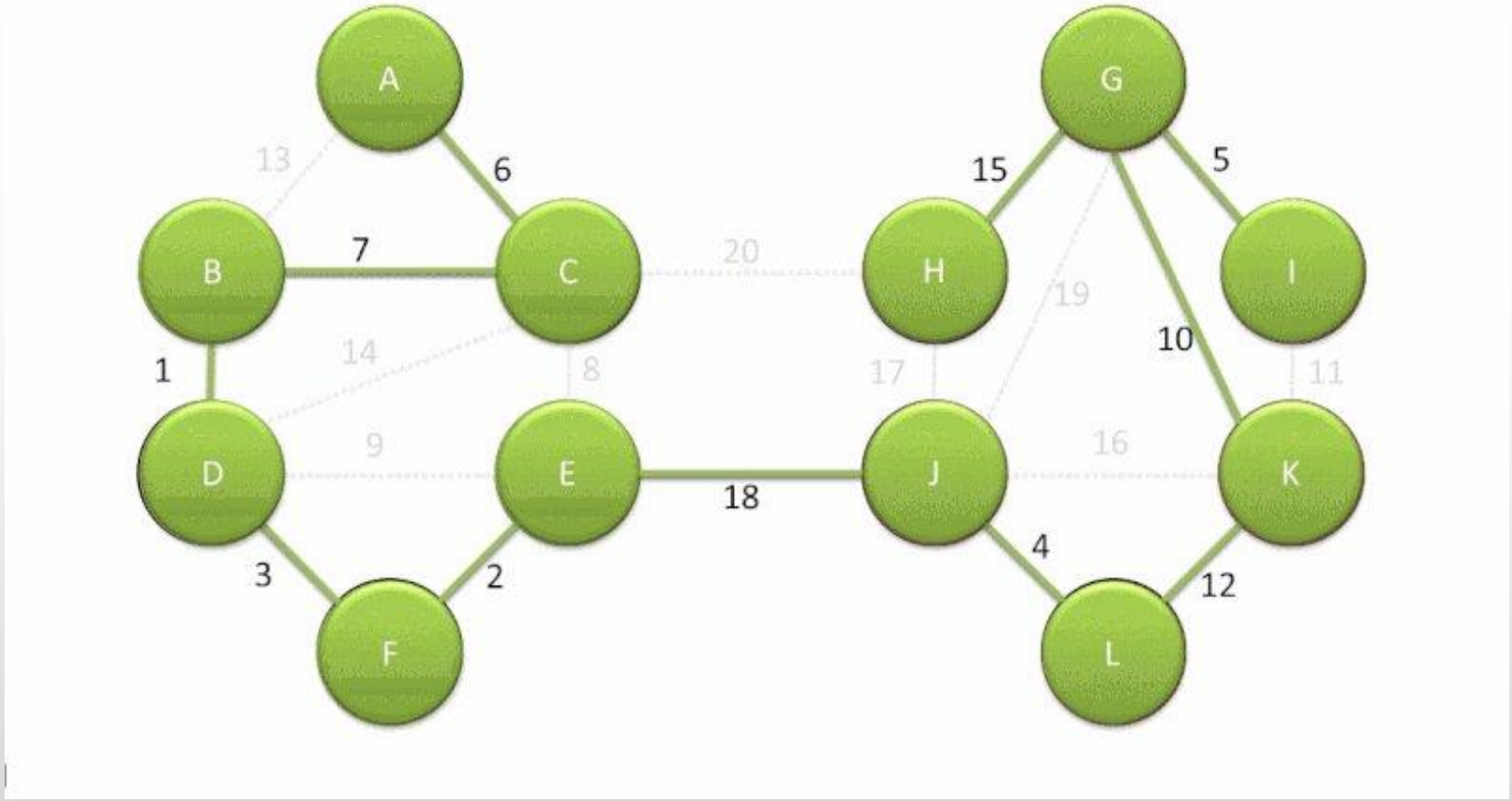
# Boruvka's Algorithm



# Boruvka's Algorithm



# Boruvka's Algorithm



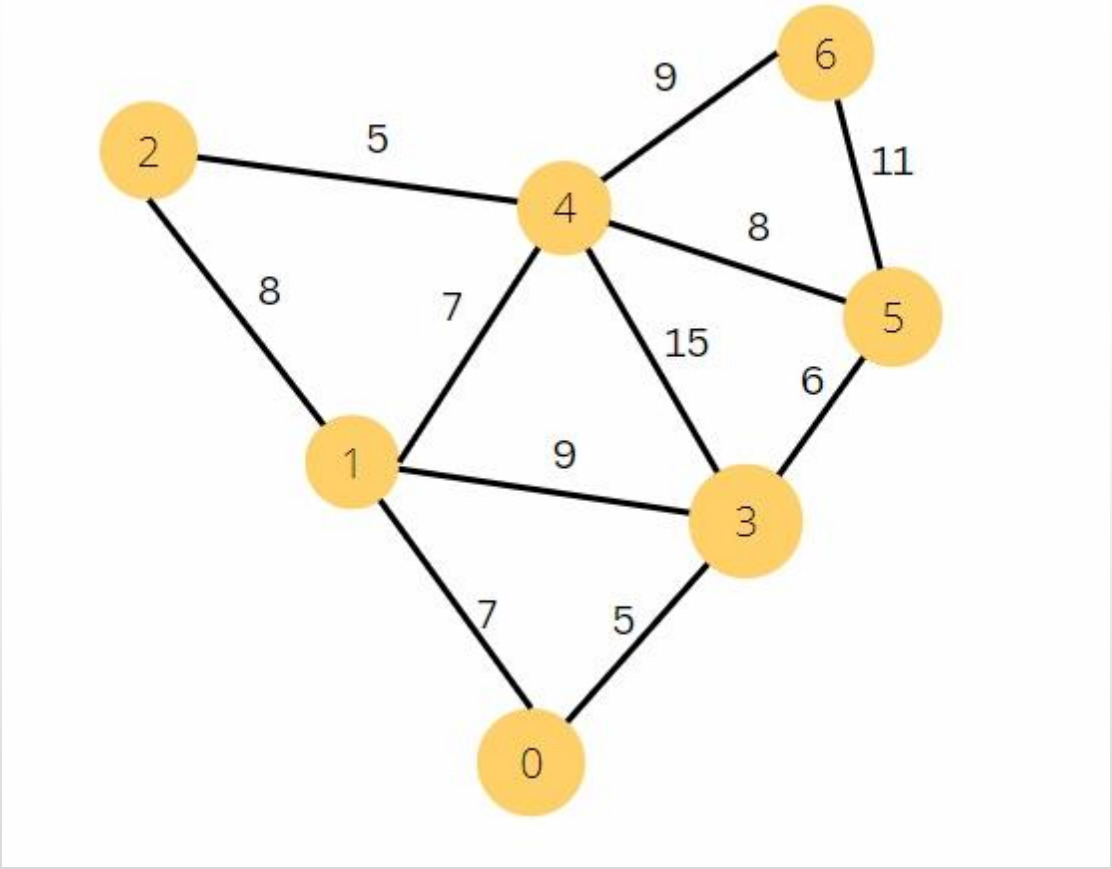
# Tersine Silme Algoritması

- *Reverse Delete* algorithm
- Minimum kapsayan ağacı bulur.
- Açgözlü (*Greedy*) bir algoritmadır.
- Kenarları ağırlıklarına göre azalan şekilde sıralar.
- Çizgenin bağlılığını koruyacak şekilde kenarları siler.
- Karmaşıklık: O (E log E)

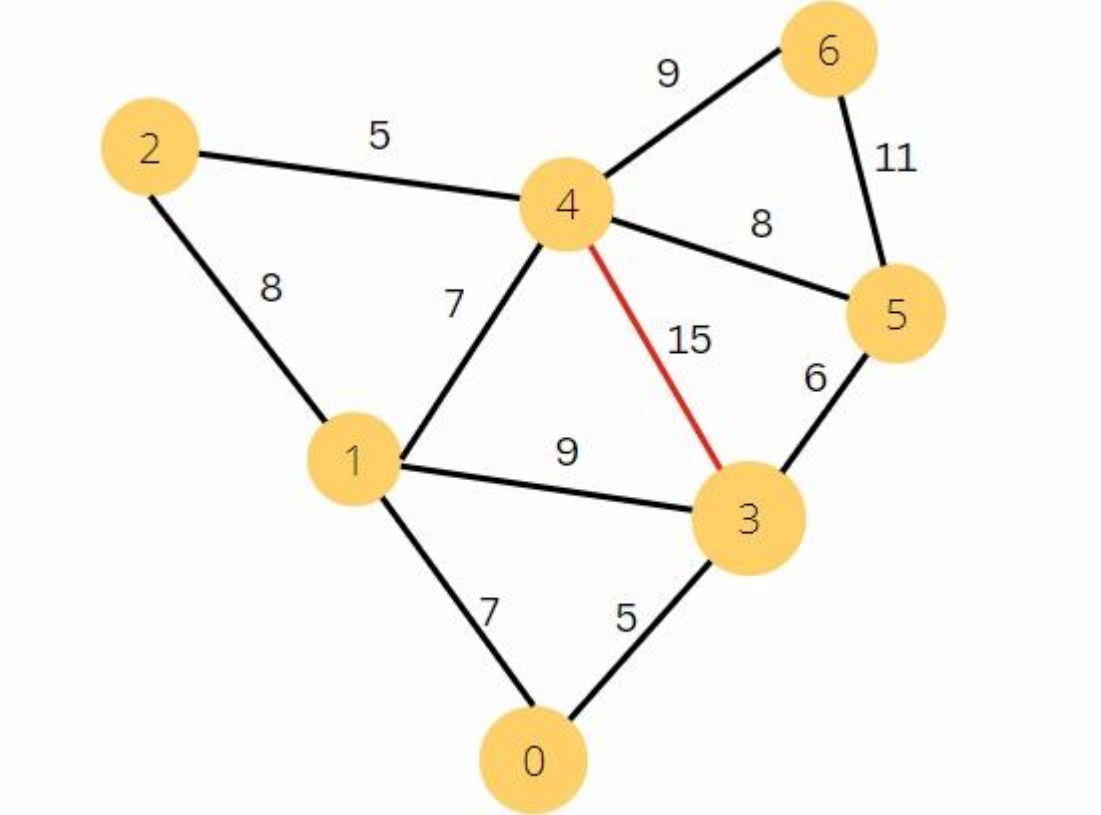
# Algoritma Adımları

- Tüm kenarları ağırlığına göre azalan şekilde sırala.
- En yüksek ağırlığa sahip kenarı sil.
- Eğer çizge hala bağlı değilse, silinen kenarı geri ekle.
- Tüm kenarlar işlenene kadar 2. ve 3. adımları tekrarla.

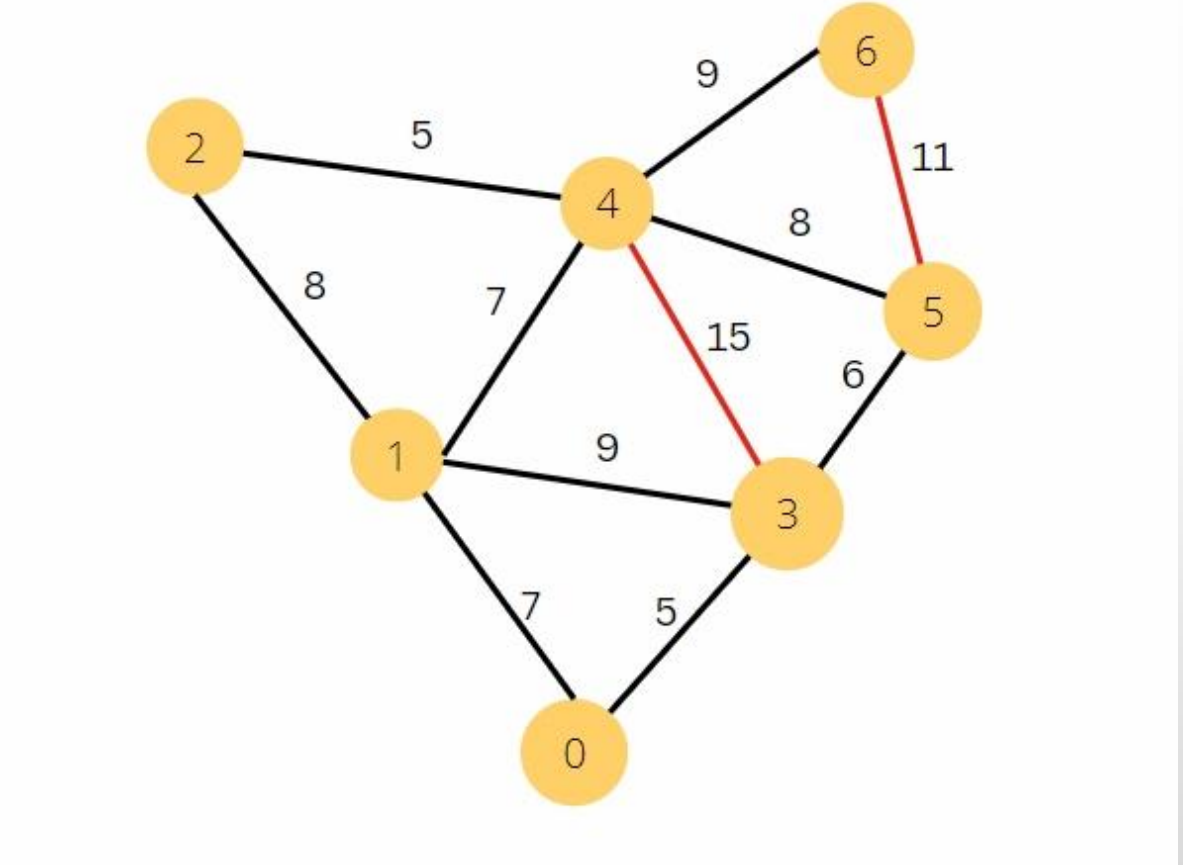
# Silmeyi Geri Al



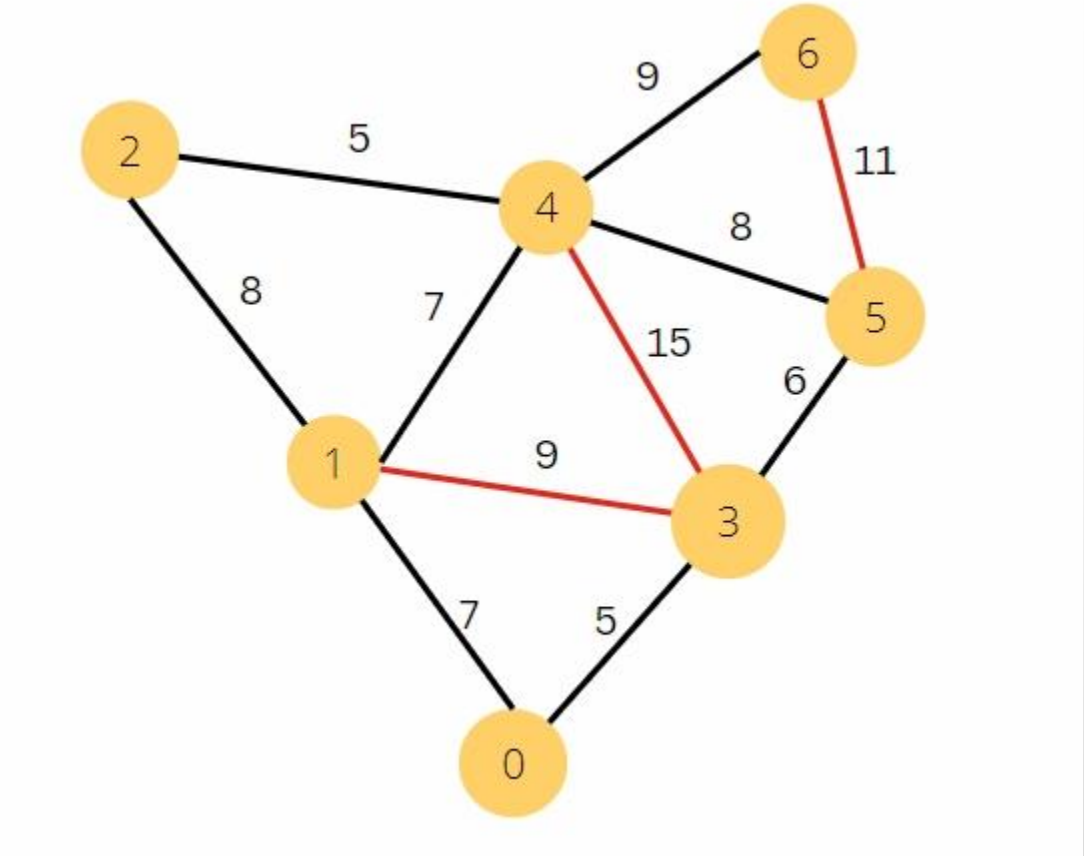
# Silmeyi Geri Al



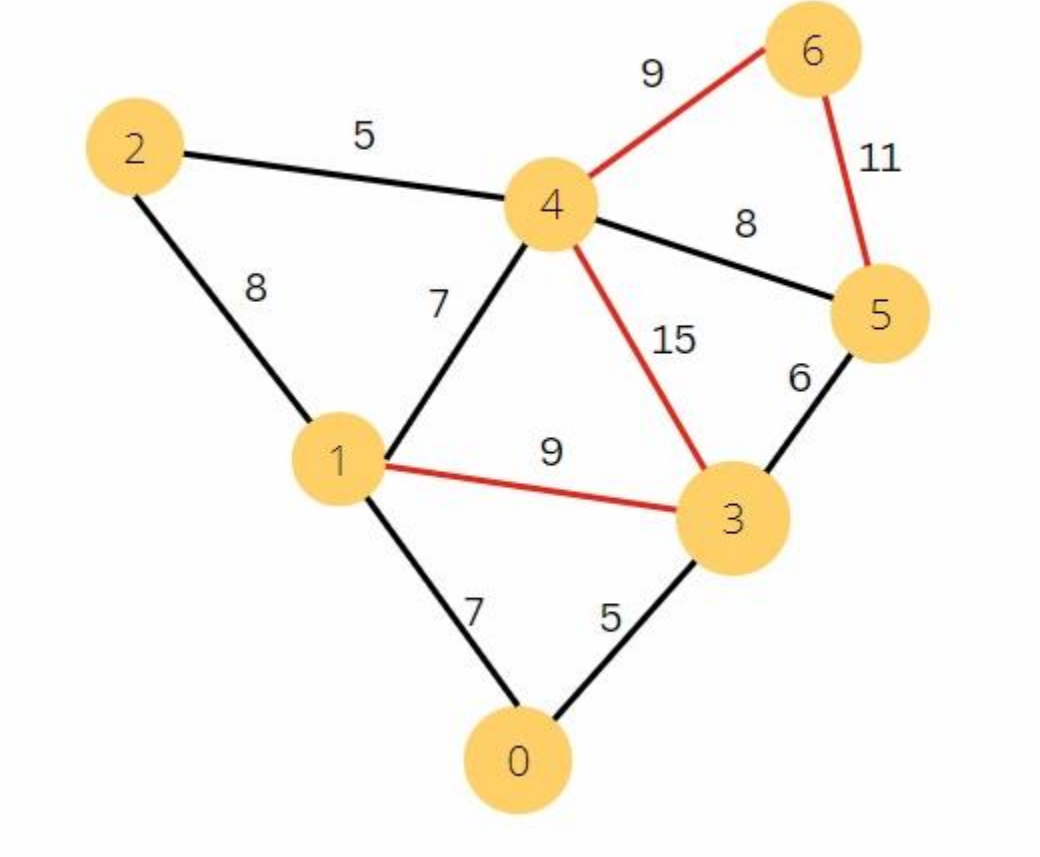
# Silmeyi Geri Al



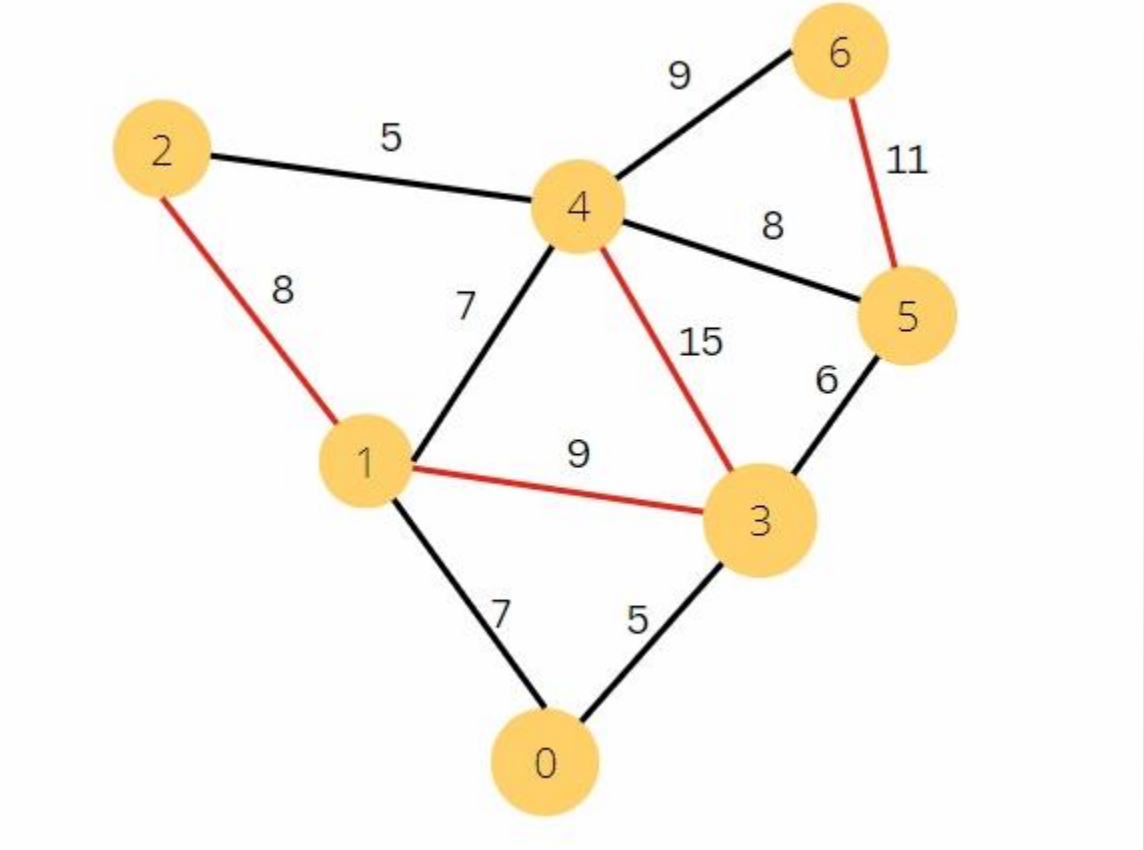
# Silmeyi Geri Al



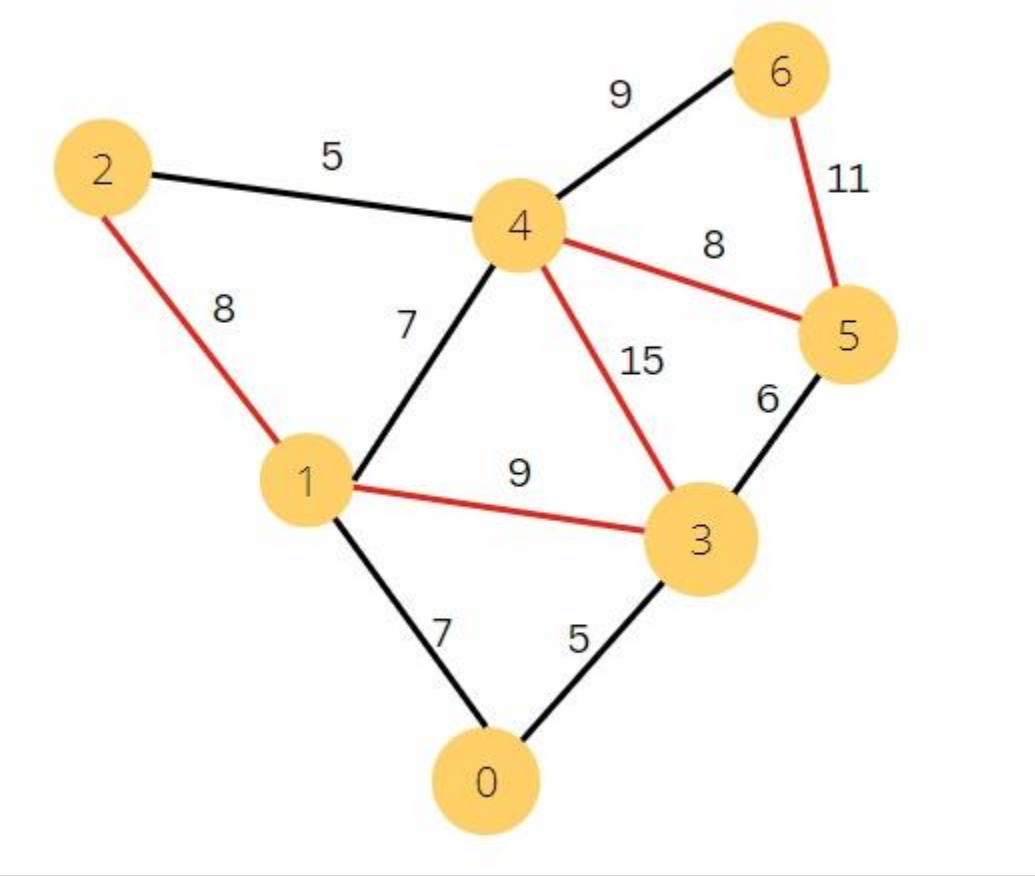
# Silmeyi Geri Al



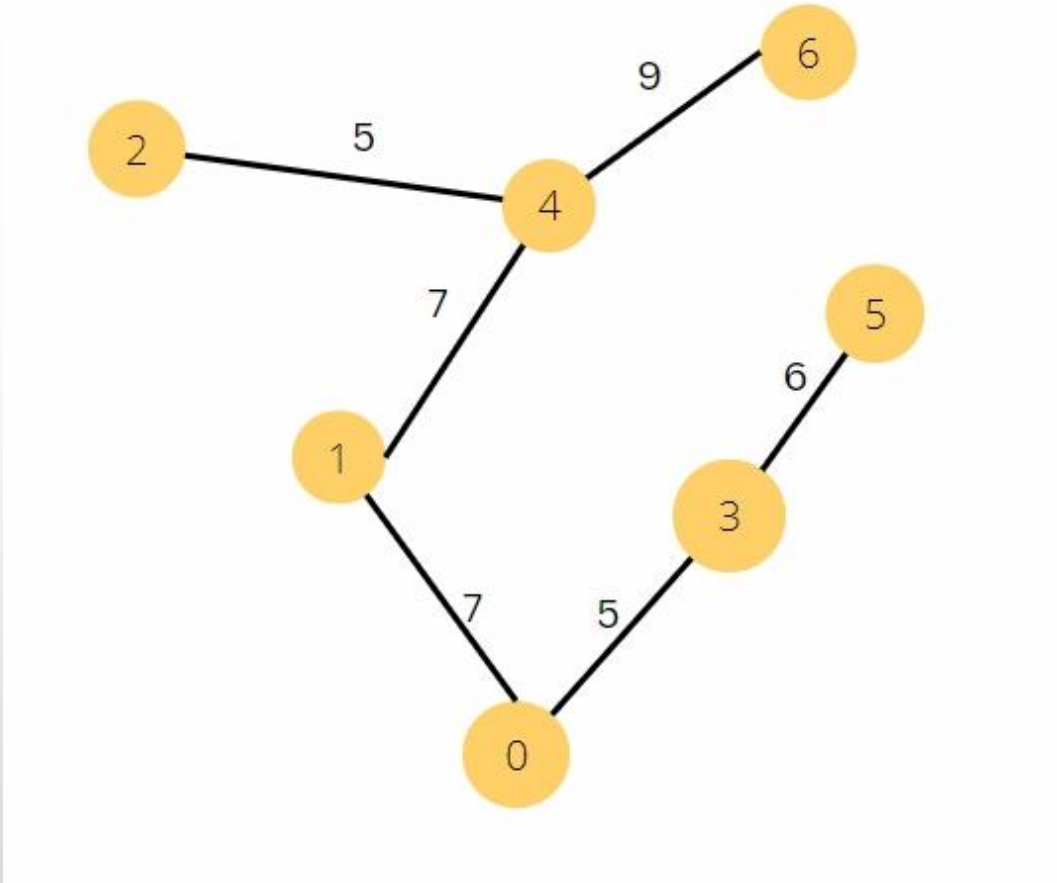
# Silmeyi Geri Al



# Silmeyi Geri Al

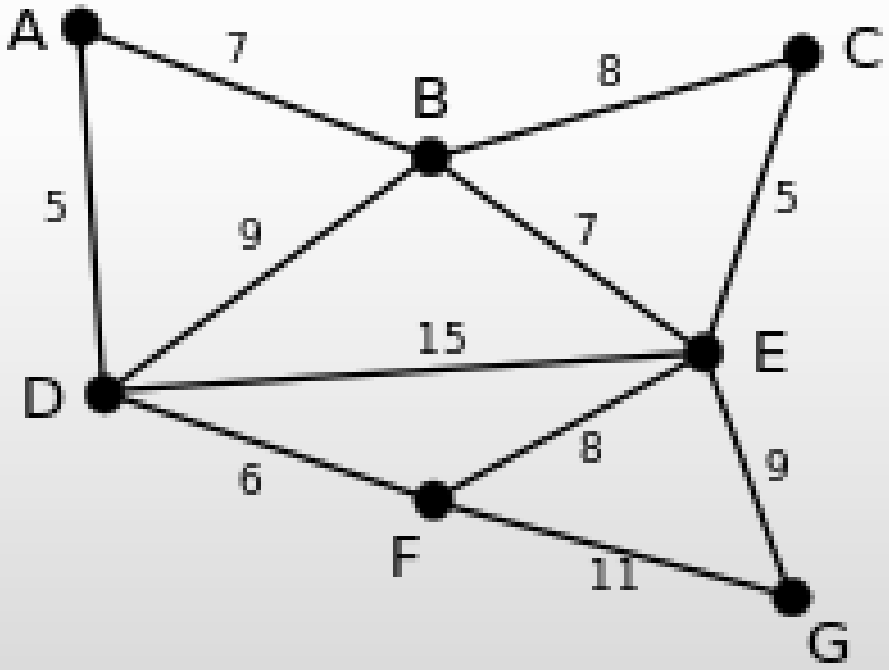


# Silmeyi Geri Al



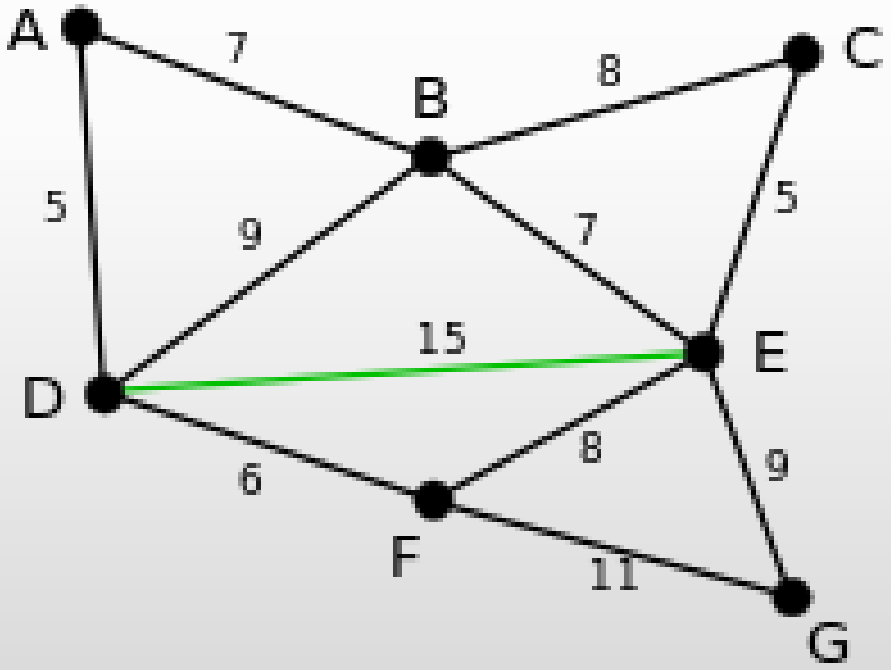
# Reverse Delete

- Aşağıdaki çizge verilsin. Sayılar kenarların ağırlıklarını gösterir.



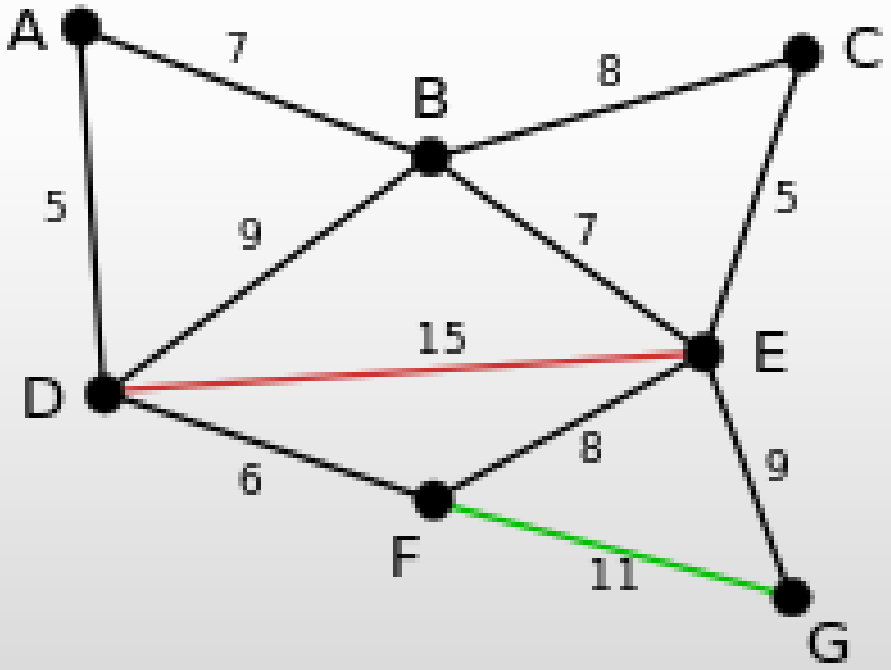
# Reverse Delete

- En yüksek ağırlıklı DE kenarı incelenir.



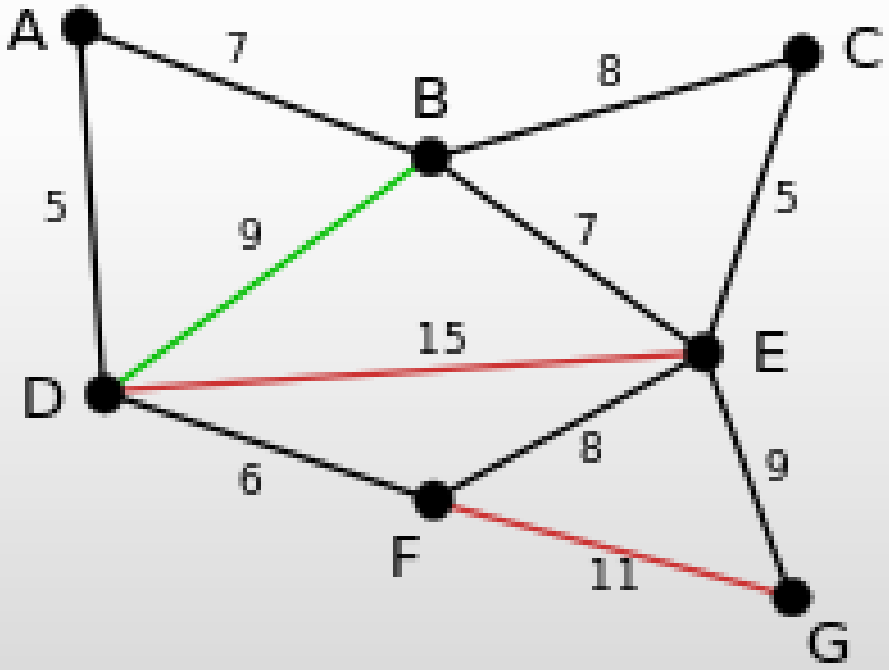
# Reverse Delete

- FG kenarı çıkarılırsa çizge bağlılığı bozuluyor mu diye kontrol edilir.



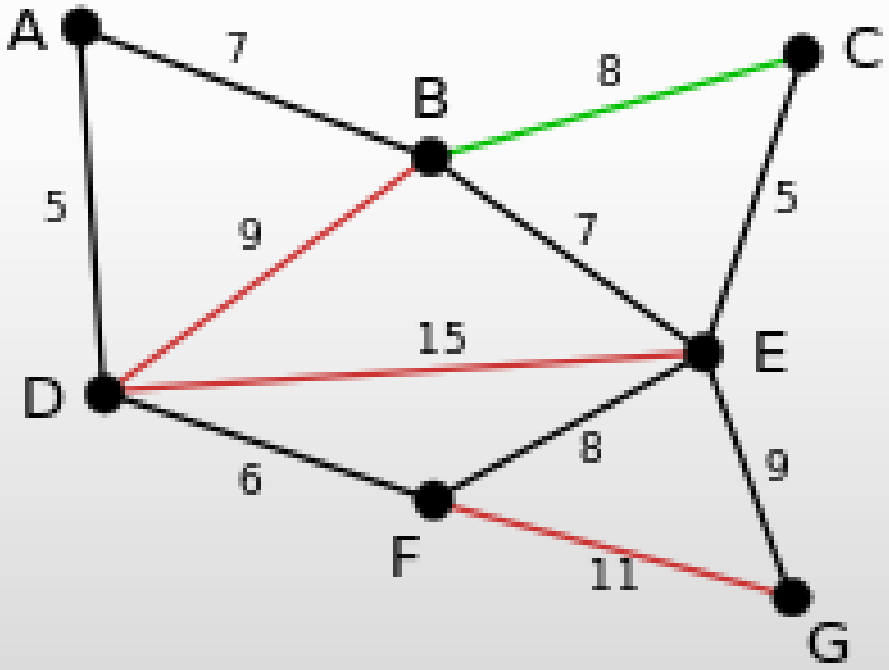
# Reverse Delete

- Bir sonraki en ağırlıklı kenar BD kontrol edilir.



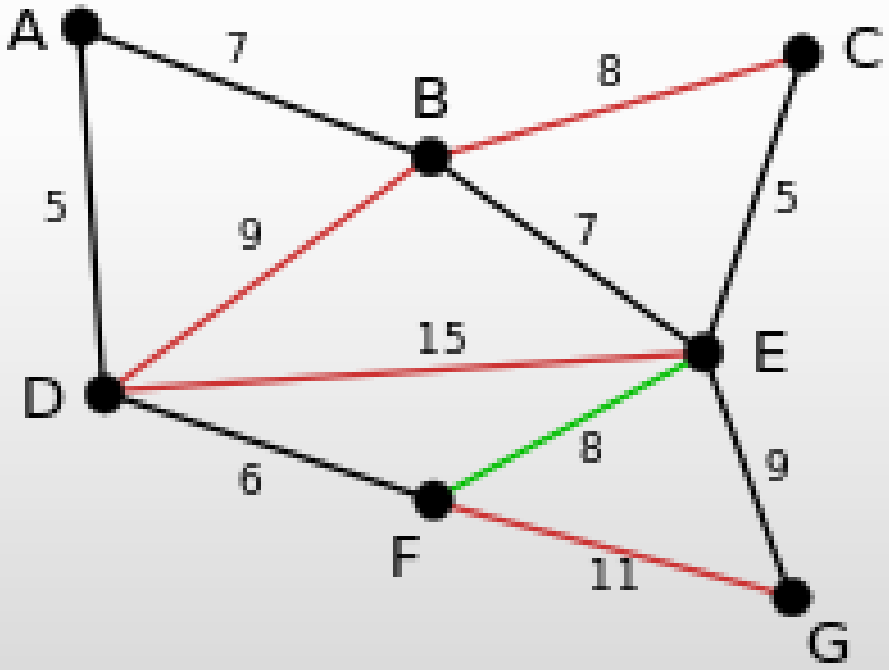
# Reverse Delete

- EG kenarı kontrol edilir, silinemeyeceği görülür. BC kenarı incelenir.



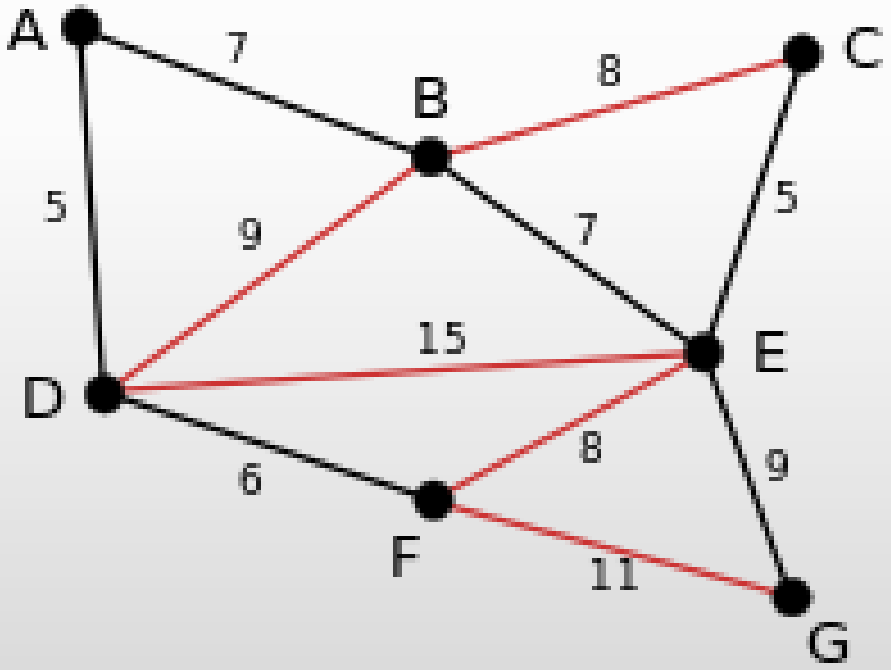
# Reverse Delete

- EF kenarı incelenir.



# Reverse Delete

- Silinecek başka bir kenar bulunamaz.



# Sözde Kod

```
REVERSE_DELETE(G):
        MST = tüm kenarlar  // Başlangıçta tüm kenarları içerir
        union-find = union-find.oluştur(V)
        kenarlar.sırala(azalana göre)

        her bir (u, v, ağırlık) için kenarlar içinde:
                MST.sil((u, v, ağırlık))
                union-find.sıfırla()
                her bir (x, y) için MST içinde:
                        union-find.birleştir(x, y)
```

# Sözde Kod (2)

```
kök = union-find.bul(V[0])
her bir v için V içinde:
        eğer union-find.bul(v) ≠ kök:
                MST.ekle((u, v, ağırlık))
döndür MST
```

SON