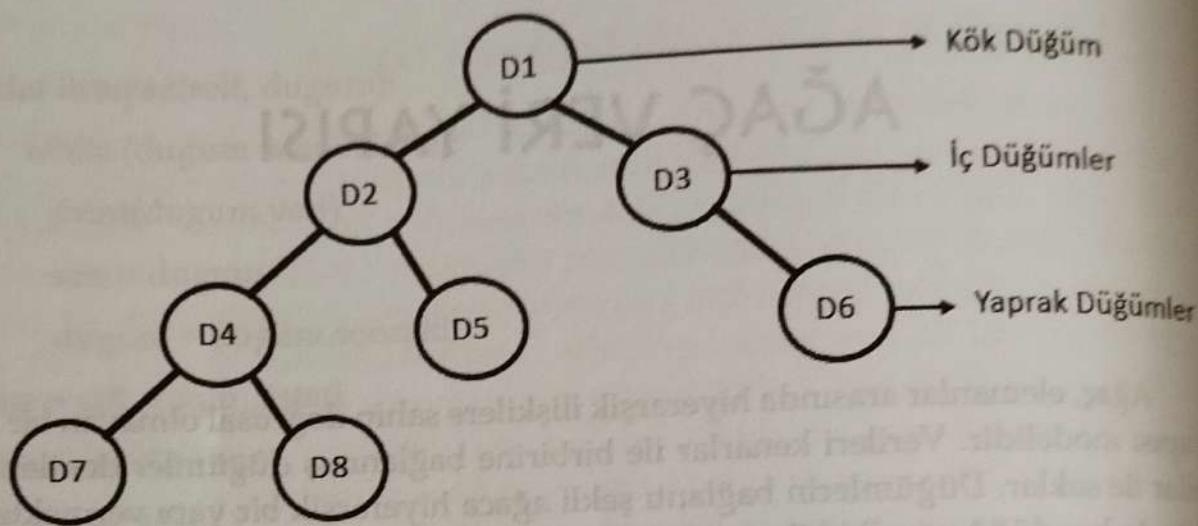
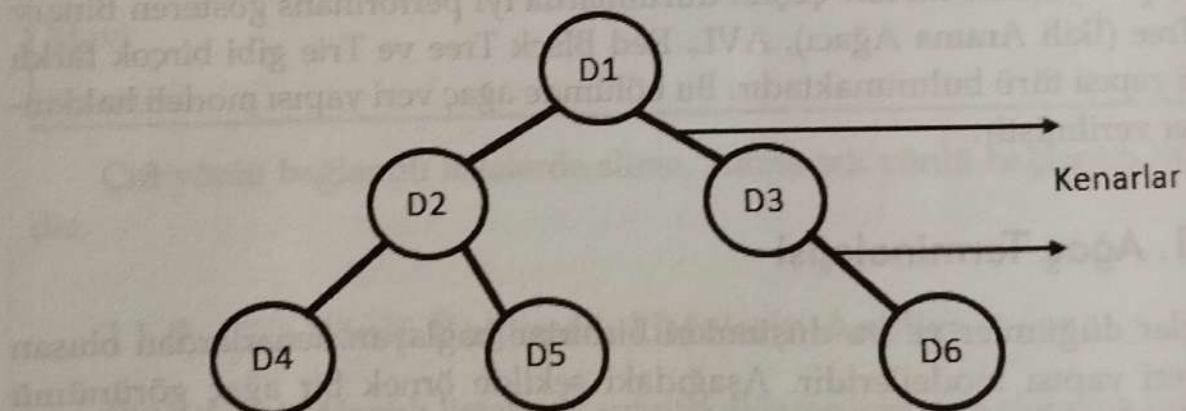


ağaçtaki herhangi bir yolun son düğümüdür. Alt düğümlere bir bağlantı/şaretçi içermezler. İç düğümler ise en az bir alt düğüme sahip olan düğümlerdir. Aşağıdaki şekilde bir ağaç yapısı ve düğümler gösterilmiştir.



Kenar/Dal/Edge: Bir ağaçın kenarı/dalı, herhangi iki düğüm arasındaki bağlantıdır. Kenarlar, düğümler arasında döngü oluşmayacak şekilde düzenlenmiştir.



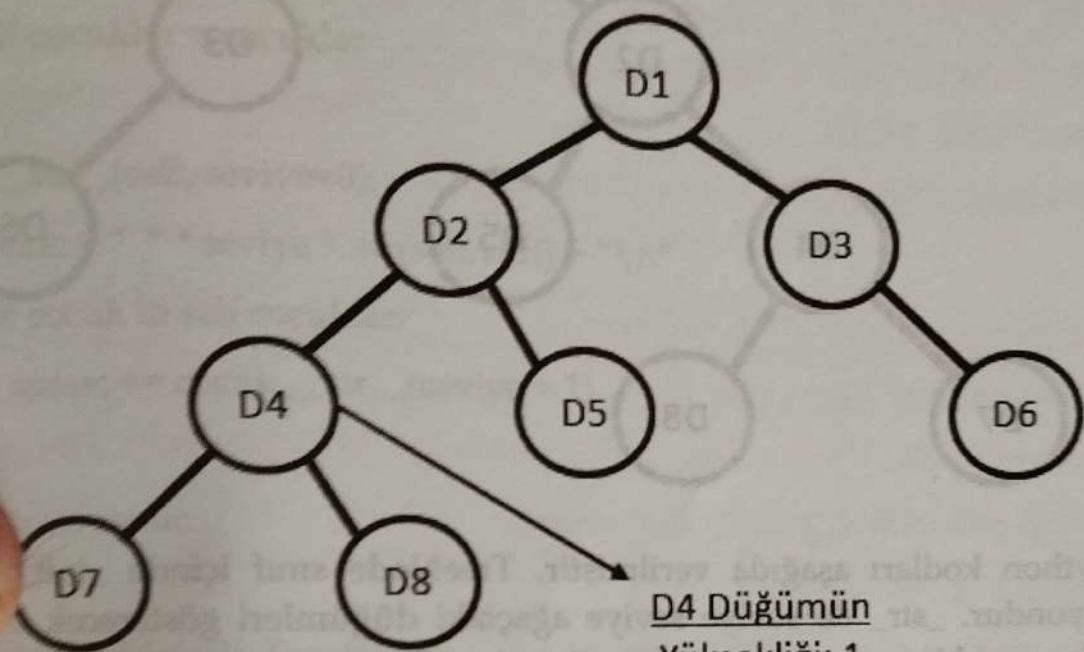
Yol: Bir ağaçın kenarları boyunca düğümlerin dizisidir.

Yükseklik: Bir düğümün yüksekliği, mevcut düğümden en derin yaprağa kadar olan kenarların sayısıdır. Diğer bir deyişle üzerinde durulan düğümden yaprak düğümlere en uzun yolu belirtir.

Derinlik: Bir düğümün derinliği, kökten düğüme kadar olan kenarların sayısıdır.

Derece: Bir düğümün derecesi, o düğümün toplam dal sayısıdır.

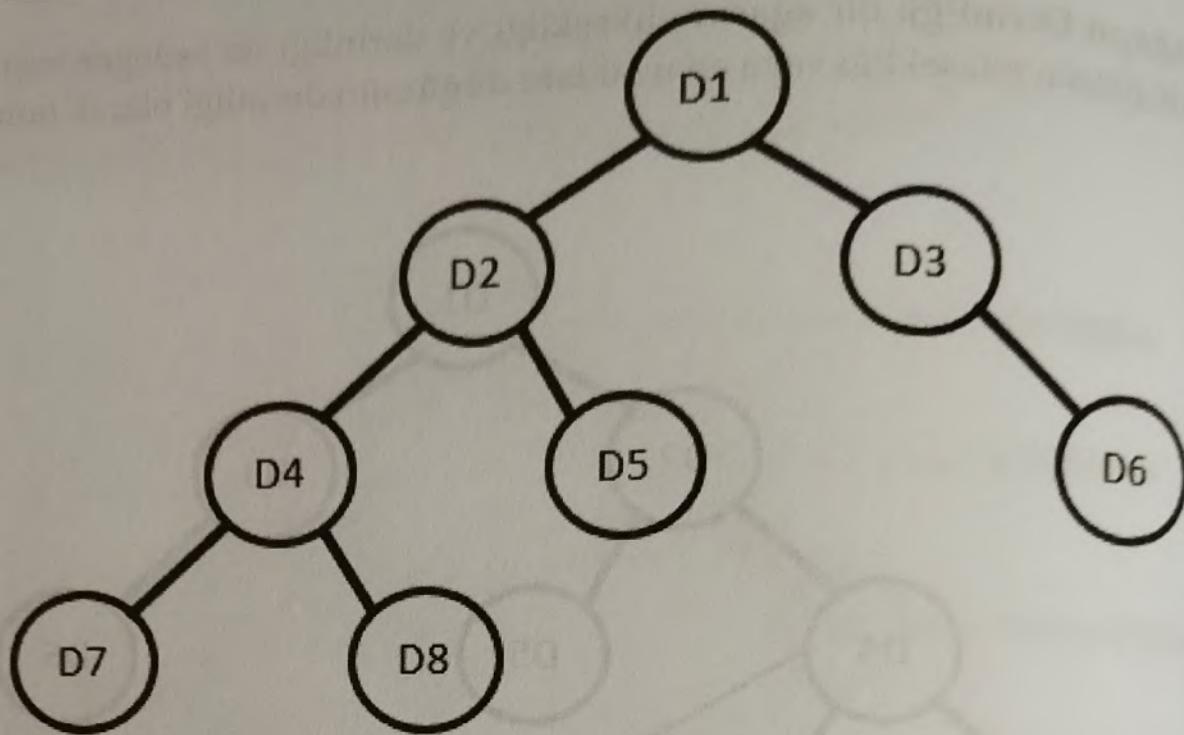
Ağacın Derinliği: Bir ağacın yüksekliği ve derinliği iki eşdeğer terimlerdir. Kök düğümün yüksekliği veya en uzaktaki düğümün derinliği olarak tanımlanır.



Yukarıdaki ağaçta D4 düğümün yüksekliği 1'dir. Çünkü D4 düğümün yaprak düğümlere olan uzaklığı 1'dir. Derinliği ise 2'dir. Çünkü kök düşümden D4 düğümüne olan uzaklıktaki kenar sayıları 2'dir. D4'ün derecesi ise 3 kenar bağlı olduğundan dolayı 3'tür. Tüm ağacın derinliği ise 3'tür.

12.2. Python Temel Ağaç İşlemleri

Python'da temel bir ağaç oluşturmak için nesneye yönelik programlama yaklaşımı kullanılmıştır. Sınıflar oluşturup bu sınıflardan nesneler oluşturularak ağaç veri yapısı modeli oluşturulmuştur. "TreeNode" isminde bir sınıf oluşturulmuştur. Bu sınıfın ağaç için düşümler oluşturmaktadır. Oluşturulan düğüme ait çocuk düğümler olacaktır. Yani bir düğüme bağlı olan alt düğümler çocuk olarak ifade edilmiştir. Aşağıda verilen şekildeki ağaç oluşturulacaktır.



Python kodları aşağıda verilmiştir. `TreeNode` sınıf içinde `_init_` kurucu fonksiyondur. `_str_` de seviye seviye ağaçtaki düğümleri gösterecek metottur. Ağaç benzeri bir sonuç gösterecektir. Sınıf içinde ağaçta düğümleri ekleyecek `coukekle()` isminde bir metot tanımlanmıştır. Diğer açıklamalar kod içinde yapılmıştır.

```
class TreeNode:  
    def __init__(self, veri, cocuklar=[]):  
        self.veri = veri  
        self.cocuklar = cocuklar  
  
    def __str__(self, seviye=0):  
        sonuc = " " * seviye + str(self.veri) + "\n"  
        for cocuk in self.cocuklar:  
            sonuc += cocuk.__str__(seviye + 1)  
  
        return sonuc  
  
    def cocukekle(self, TreeNode):  
        self.cocuklar.append(TreeNode)
```

TreeNode('D1', []) #ilk düğümün oluşturulması

#D2 ve D3 düğümlerin oluşturulması

D2=TreeNode('D2', [])

D3=TreeNode('D3', [])

#D1 düğümüne D2 ve D3 düğümün bağlanması.

D1.cocukekle(D2)

D1.cocukekle(D3)

#Diğer düğümlerin oluşturulması

```
D4=TreeNode('D4',[])
```

```
D5=TreeNode('D5',[])
```

```
D6=TreeNode('D6',[])
```

```
D7=TreeNode('D7',[])
```

```
D8=TreeNode('D8',[])
```

#D2 düğümüne D4 ve D5'in bağlanması

```
D2.cocukekle(D4)
```

```
D2.cocukekle(D5)
```

#D3 düğümüne D6 düğümün bağlanması

```
D3.cocukekle(D6)
```

#D4 düğümüne D7 ve D8'in bağlanması

```
D4.cocukekle(D7)
```

```
D4.cocukekle(D8)
```

#Kök düğümün yazdırılması. Ağaçtaki tüm düğümler ekrana yazdırılacaktır.

```
print(D1)
```

D1

D2

D4

D7

D8

D5

D3

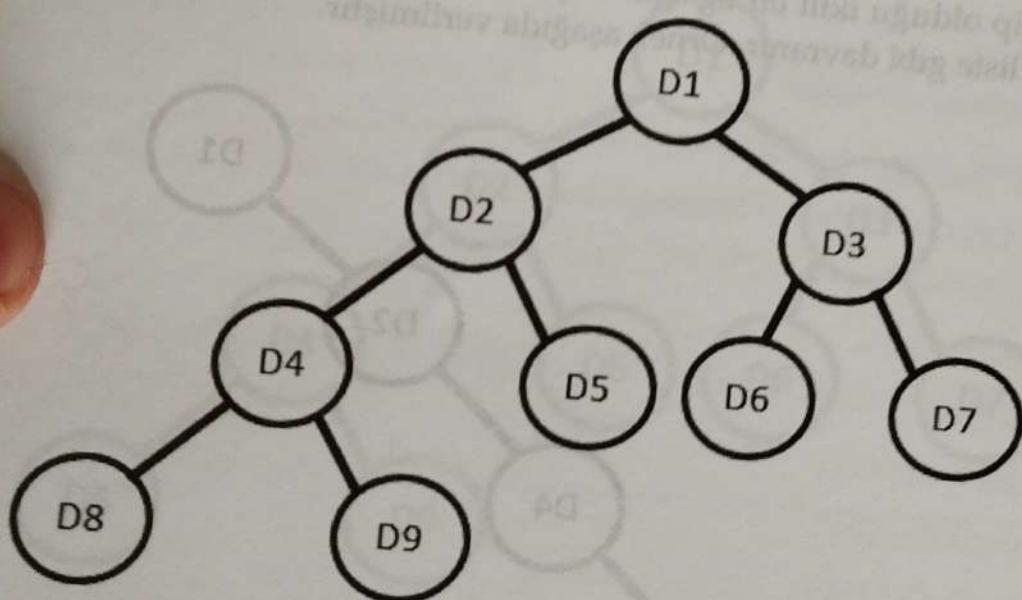
D6

12.3. İkili Ağaç Türleri

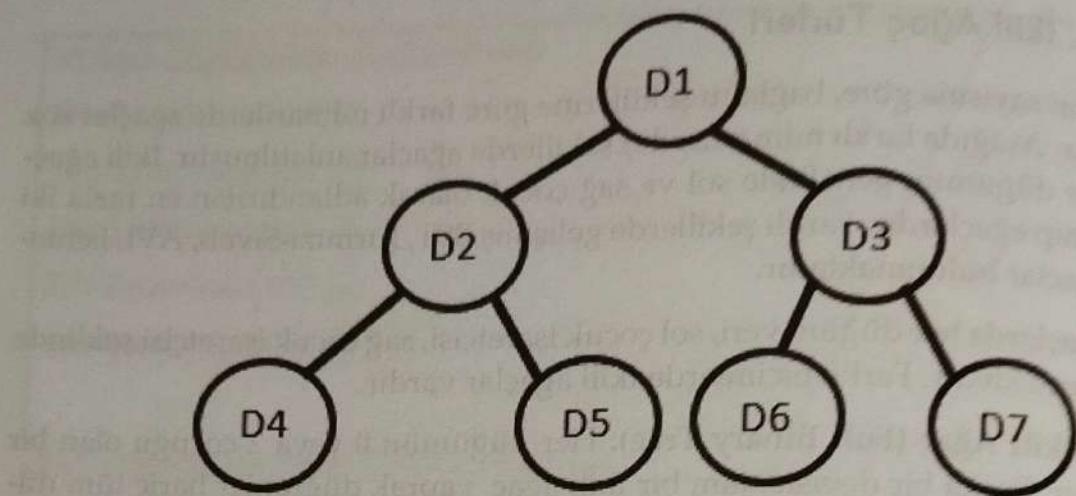
Düğüm sayısına göre, bağlantı şekillerine göre farklı mimarilerde ağaçlar söz konusudur. Aşağıda farklı mimarilerde/şekillerde ağaçlar anlatılmıştır. İkili ağaçlar, her bir düğümün genellikle sol ve sağ çocuk olarak adlandırılan en fazla iki çocuğu sahip ağaçlardır. Farklı şekillerde gelişmiş BST, Kırmızı-Siyah, AVL isminde ikili ağaçlar bulunmaktadır.

İki ağaçlarda her düğüm veri, sol çocuk işaretçisi, sağ çocuk işaretçisi şeklinde 3 bilgi içermektedir. Farklı biçimlerde ikili ağaçlar vardır.

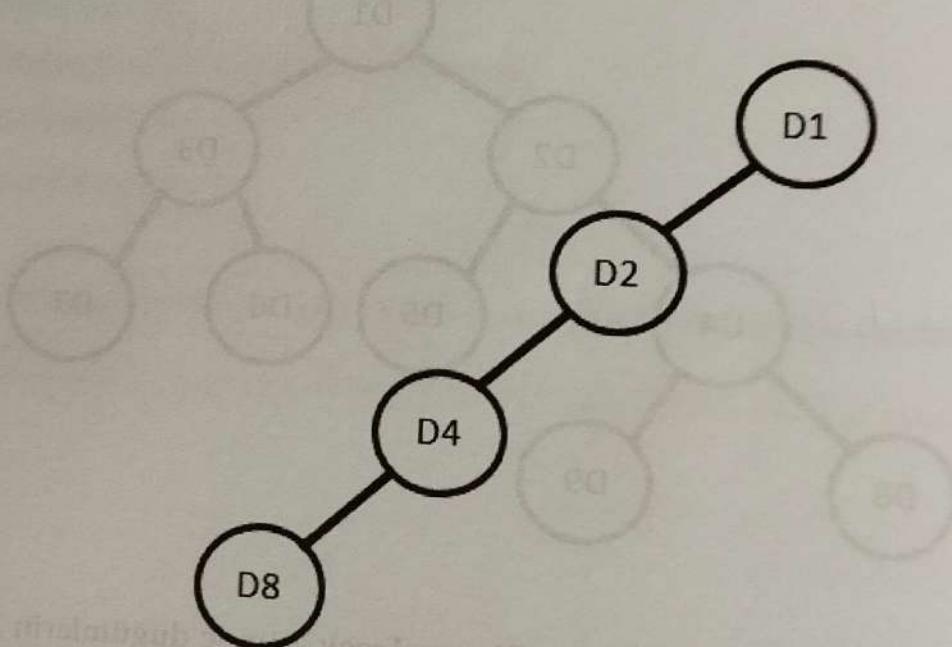
Tam İkili Ağaç (Full Binary Tree): Her düğümün 0 veya 2 çocuğu olan bir ikili ağaçtır. Başka bir deyişle, tam bir ikili ağaç, yaprak düğümler hariç tüm düğümlerin iki çocuğu olan ağaçtır. Örnek aşağıda verilmiştir. Şekilden görüldüğü gibi her düğümün iki çocuğu bulunmaktadır.



Mükemmel İkili Ağaç (Perfect Binary Tree): Tüm iç düğümlerin 2 çocuğu olduğu ve tüm yaprak düğümlerinin aynı derinlige sahip olduğu bir ikili ağaçtır. Örnek aşağıda verilmiştir.



Dejenere Ağaç (Degenerate Tree): Her bir üst düğümün yalnızca bir alt düğüme sahip olduğu ikili bir ağaçtır. Dejenere bir ağaç, yapısına bağlı olarak bağlantılı bir liste gibi davranışır. Örnek aşağıda verilmiştir.



12.4. Ağaçlarda Dolaşım İşlemleri

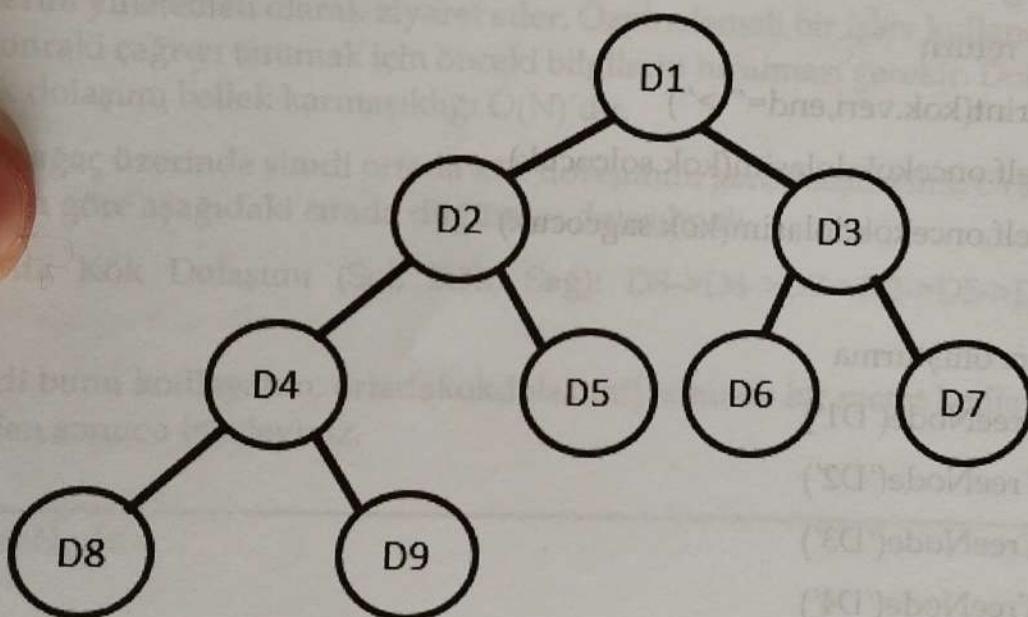
İkili ağaçlar üzerinde gerçekleştirilecek en önemli işlemlerden biri dolaşımdır. Bir ağaç dolaşmak, ağacın her düğümünü ziyaret etmek anlamına gelir. İkili bir ağaçta geçiş çeşitli şekillerde gerçekleştirilebilir.

Önce Derinlik Dolaşımı (Depth First Traversals): Farklı biçimlerde düğümlerin dolaşımıdır. 3 farklı dolaşım türü bulunmaktadır.

- **Ortada Kök (InOrder) Dolaşımı:** Düğümleri sol, kök ve sağ şeklinde dolaşımındır.
- **Önce Kök (PreOrder) Dolaşımı:** Düğümleri Kök, sol ve sağ şeklinde dolaşım biçimidir.
- **Sonda Kök (PostOrder) Dolaşımı:** Düğümleri sol, sağ ve kök şeklinde dolaşım biçimidir.

Önce Genişlik Dolaşımı (Breadth-First Traversal): Ağaç seviye dolaşımıdır. Bir düğümden diğer düşüme geçerek ağaç derinliğini çıkarmaya çalışan dolaşımındır.

Şimdi Python ile önce derinlik dolaşımlarını kodlayalım. Önce kök dolaşımı yöntemini kodlayalım. Önce derinlik dolaşımını aşağıdaki sekilden oluşan ağaç üzerinde gerçekleştirilecektir.



Bu ağaç yapısı üzerinde önce kök dolaşımı ile aşağıdaki gibi bir dolaşım gerçekleştirilecektir.

Önce Kök Dolaşımı (Kök, Sol, Sağ): D1->D2->D4->D8->D9->D5->D3->D6->D7. Şimdi bunu kodlayalım. oncekokdolasim() isminde bir metot kodlanmıştır. Elde edilen sonucu inceleyiniz.

```

class TreeNode:

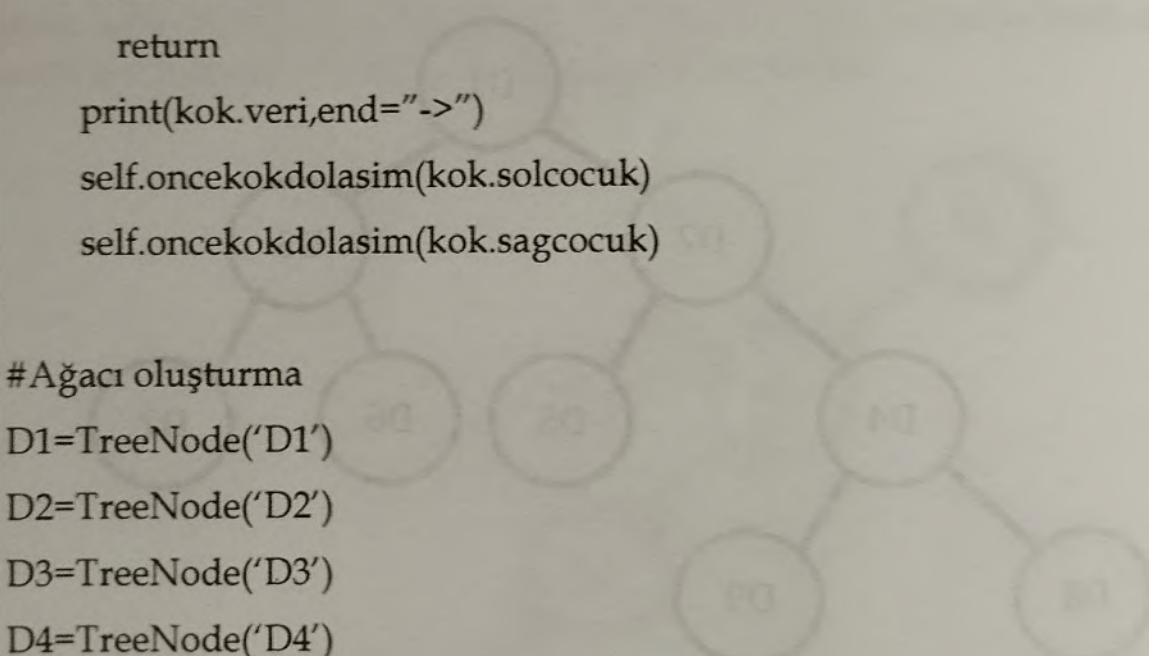
    def __init__(self, veri):
        self.veri = veri
        self.solcuk = None
        self.sagcuk = None

    def oncekokdolasim(self,kok):
        if not kok:
            return
        print(kok.veri,end="->")
        self.oncekokdolasim(kok.solcuk)
        self.oncekokdolasim(kok.sagcuk)

#Ağacı oluşturma
D1=TreeNode('D1')
D2=TreeNode('D2')
D3=TreeNode('D3')
D4=TreeNode('D4')
D5=TreeNode('D5')
D6=TreeNode('D6')
D7=TreeNode('D7')
D8=TreeNode('D8')
D9=TreeNode('D9')

#Bağlantıların Eklenmesi
D1.solcuk=D2
D1.sagcuk=D3

```



D2.solcocuk=D4

D2.sagcocuk=D5

D3.solcocuk=D6

D3.sagcocuk=D7

D4.solcocuk=D8

D4.sagcocuk=D9

D1.oncekokdolasim(D1)

D1->D2->D4->D8->D9->D5->D3->D6->D7

Önce kök dolaşımı için zaman karmaşıklığı $O(N)$ 'dir. Çünkü işlev ağacın tüm düğümlerini yinelemeli olarak ziyaret eder. Özyinelemeli bir işlev kullandığımızda, bir sonraki çağrıyı tanımak için önceki bilgilerin tutulması gereklidir. Dolayısıyla önce kök dolaşımı bellek karmaşıklığı $O(N)$ 'dir.

Aynı ağaç üzerinde şimdi ortada kök dolaşımını gerçekleştirelim. Ortada kök dolaşımına göre aşağıdaki sırada düğüler dolaşılmalı.

Ortada Kök Dolaşımı (Sol, Kök, Sağ): D8->D4->D9->D2->D5->D1->D6->D3-D7

Şimdi bunu kodlayalım. ortadakokdolasim() isminde bir metot kodlanmıştır. Elde edilen sonucu inceleyiniz.

TreeNode:

```
def __init__(self, veri):
    self.veri = veri
    self.solcocuk = None
    self.sagcocuk = None

def ortadakokdolasim(self,kok):
    if not kok:
        return
    self.ortadakokdolasim(kok.solcocuk)
    print(kok.veri,end="->")
    self.ortadakokdolasim(kok.sagcocuk)
```

```

#Ağacı oluşturma
D1=TreeNode('D1')
D2=TreeNode('D2')
D3=TreeNode('D3')
D4=TreeNode('D4')
D5=TreeNode('D5')
D6=TreeNode('D6')
D7=TreeNode('D7')
D8=TreeNode('D8')
D9=TreeNode('D9')

#Bağlantıların Eklenmesi
D1.solcocuk=D2
D1.sagcocuk=D3
D2.solcocuk=D4
D2.sagcocuk=D5
D3.solcocuk=D6
D3.sagcocuk=D7
D4.solcocuk=D8
D4.sagcocuk=D9

```

D1.ortadakokdolasim(D1)

D8->D4->D9->D2->D5->D1->D6->D3->D7

Ortada kök dolaşımı için zaman karmaşıklığı $O(N)$ 'dir. Çünkü işlev *ağacın* tüm düğümlerini yinelemeli olarak ziyaret eder. Özyinelemeli bir işlev kullandığımızda, bir sonraki çağrıyı tanıtmak için önceki bilgilerin tutulması gereklidir. Dolayısıyla ortada kök dolaşımı bellek karmaşıklığı $O(N)$ 'dir.

Son olarak aynı ağaç üzerinde sonda kök dolaşımını kodlayalım. Sonda kök dolaşımına göre aşağıdaki sırada düğümler dolaşılmalı.

Sonda Kök Dolaşımı (Sol, Sağ, Kök): D8->D9->D4->D5->D2->D6->D7->D8->D1

Şimdi bunu kodlayalım. `sondakokdolasim()` isminde bir metod kodlanmıştır. Elde edilen sonucu inceleyiniz.

`class TreeNode:`

```
def __init__(self, veri):
    self.veri = veri
    self.solcuk = None
    self.sagcuk = None
```

`def sondakokdolasim(self, kok):`

`if not kok:`

`return`

`self.sondakokdolasim(kok.solcuk)`

`self.sondakokdolasim(kok.sagcuk)`

`print(kok.veri,end="->")`

#Ağacı oluşturma

`D1=TreeNode('D1')`

`D2=TreeNode('D2')`

`D3=TreeNode('D3')`

`D4=TreeNode('D4')`

`D5=TreeNode('D5')`

`D6=TreeNode('D6')`

`D7=TreeNode('D7')`

`D8=TreeNode('D8')`

`D9=TreeNode('D9')`

#Bağlantıların Eklenmesi

`D1.solcuk=D2`

D1.sagcocuk=D3

D2.solcocuk=D4

D2.sagcocuk=D5

D3.solcocuk=D6

D3.sagcocuk=D7

D4.solcocuk=D8

D4.sagcocuk=D9

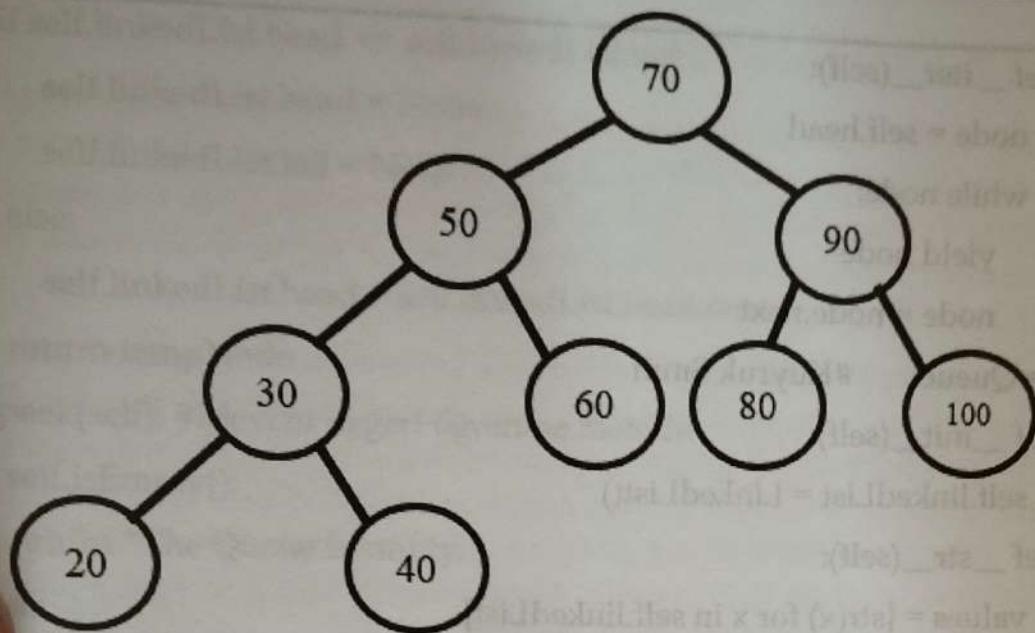
D1.sondakokdolasim(D1)

D8->D9->D4->D5->D2->D6->D7->D3->D1

Sonda kök dolaşımı için zaman karmaşıklığı $O(N)$ 'dir. Çünkü işlev ağacın tüm düğümlerini yinelemeli olarak ziyaret eder. Özyinelemeli bir işlev kullandığımızda, bir sonraki çağrıyı tanıtmak için önceki bilgilerin tutulması gereklidir. Dolayısıyla sonda kök dolaşımı bellek karmaşıklığı $O(N)$ 'dir.

12.5. İkili Arama Ağacı

İkili Arama Ağacı, düğümlerin belirli bir sırada düzenlendiği bir ağaç veri yapısı modelidir. İkili arama ağacında düğümler belirli bir düzende yerlesir. Bir düğüme bağlı sol alt ağaç değerleri düğümün değerinden küçük olmalıdır. Düğümün sağ tarafına bağlı değerler (sağ alt ağaç değerleri) düğümün değerinden büyük olmalıdır. Ağaç içindeki her düğüm bir ikili arama ağaç olmalıdır. Bir ikili arama ağacında veriler sıralıdır. İkili arama ağaçlarında veriler sıralı bir şekilde düzenlenir. İkili arama ağaçlarında ekleme, silme, arama işlemleri klasik ikili ağaçlara göre daha hızlıdır. Aşağıdaki şekilde bir ikili arama ağacı vardır. Şekilden görüldüğü gibi ağaçtaki her düğümün sol değeri kendisinden küçük ve kendi sağ çocuk değerinden küçüktür.



12.6. İkili Arama AĞacı Oluşturma

Bir ikili arama ağacında düğümler veri, sol ve sağ düğümlere ait referanslarдан oluşur. Bağlantılı liste kullanarak ikili arama ağaçları oluşturacağız. Öncelikle aşağıdaki kodu bir ayrı sınıf olarak kayıt edip yazacağımız kodlar içinde modül olarak çağıracağız. Kodlardaki değişken isimleri, metod isimler vs. ifadeler İngilizce olarak verilmiştir. Kodlar netten indirilmiştir. Değiştirilmeden verilmiştir. İkili arama ağacı bu kodlar üzerinden açıklanmıştır. Aşağıdaki kodlar bağlantılı liste ile ikili arama ağaçları oluşturulacaktır. Gerekli diğer açıklamalar kod içerisinde yapılmıştır.

```

class Node: #Düğüm oluşturma sınıfı ve korucu metodlar
    def __init__(self, value=None):
        self.value = value
        self.next = None
    def __str__(self):
        return str(self.value)

class LinkedList: #Bağlantılı liste sınıfı
    def __init__(self):
        self.head = None
        self.tail = None
  
```

```
def __iter__(self):
    node = self.head
    while node:
        yield node
        node = node.next

class Queue:      #Kuyruk Sınıfı
    def __init__(self):
        self.linkedList = LinkedList()
    def __str__(self):
        values = [str(x) for x in self.linkedList]
        return ' '.join(values)
    def enqueue(self, value): #Kuyruğa ekleme metodu
        new_node = Node(value)
        if self.linkedList.head == None:
            self.linkedList.head = new_node
            self.linkedList.tail = new_node
        else:
            self.linkedList.tail.next = new_node
            self.linkedList.tail = new_node
    def isEmpty(self): #Kuyruk boş olup olmadığını kontrolü
        if self.linkedList.head == None:
            return True
        else:
            return False
    def dequeue(self): #Kuyruktan alma metodu
        if self.isEmpty():
            return "The Queue is empty."
        else:
            tempNode = self.linkedList.head
```

```

if self.linkedList.head == self.linkedList.tail:
    self.linkedList.head = None
    self.linkedList.tail = None
else:
    self.linkedList.head = self.linkedList.head.next
return tempNode

def peek(self): #Mevcut değeri öğrenme metodu
    if self.isEmpty():
        return "The Queue is empty."
    else:
        return self.linkedList.head

def delete(self):
    self.linkedList.head = None
    self.linkedList.tail = None

```

Yukarıdaki kod “kuyruksınıf” şeklinde ayrı bir dosyada saklandıktan sonra ikili arama ağacını oluşturalım. Yukarıdaki kodlar modül olarak yeni oluşturulan dosyalarda çağrılacaktır. Aşağıdaki kod içinde kuyruksınıf modülü çağrılmıştır. Ve BSTNode sınıfı ile ikili arama ağacı düğümleri tanımlamak için eklenmiştir.

```

import kuyruksınıf as queue

class BSTNode:
    def __init__(self, data):
        self.data = data
        self.leftChild = None
        self.rightChild = None
    newTree = BSTNode(None)

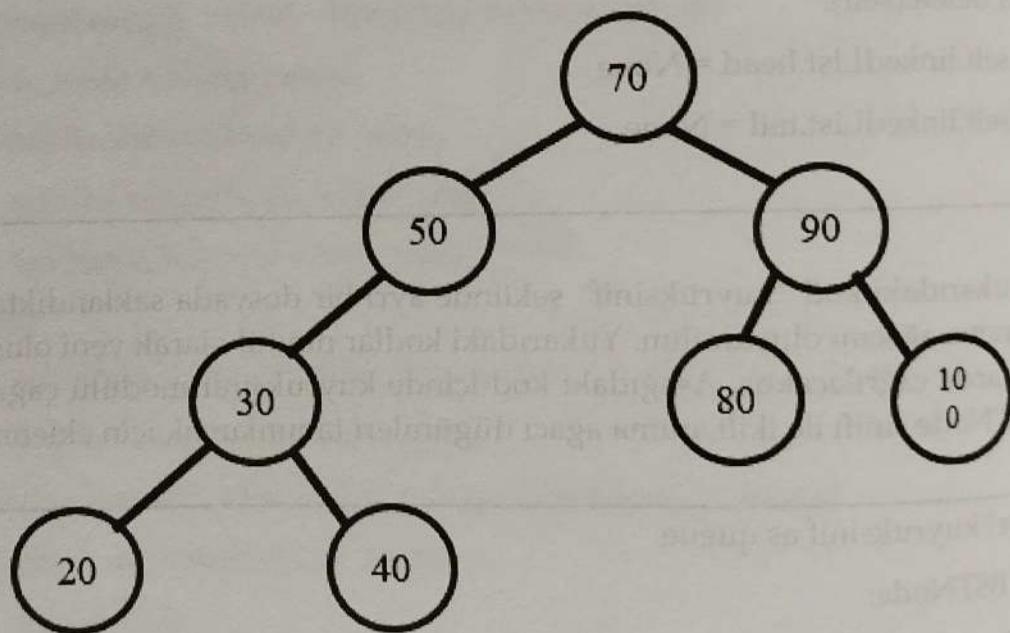
```

İkili arama ağacının oluşturulması için zaman ve bellek karmaşıklıkları $O(1)$ 'dir.

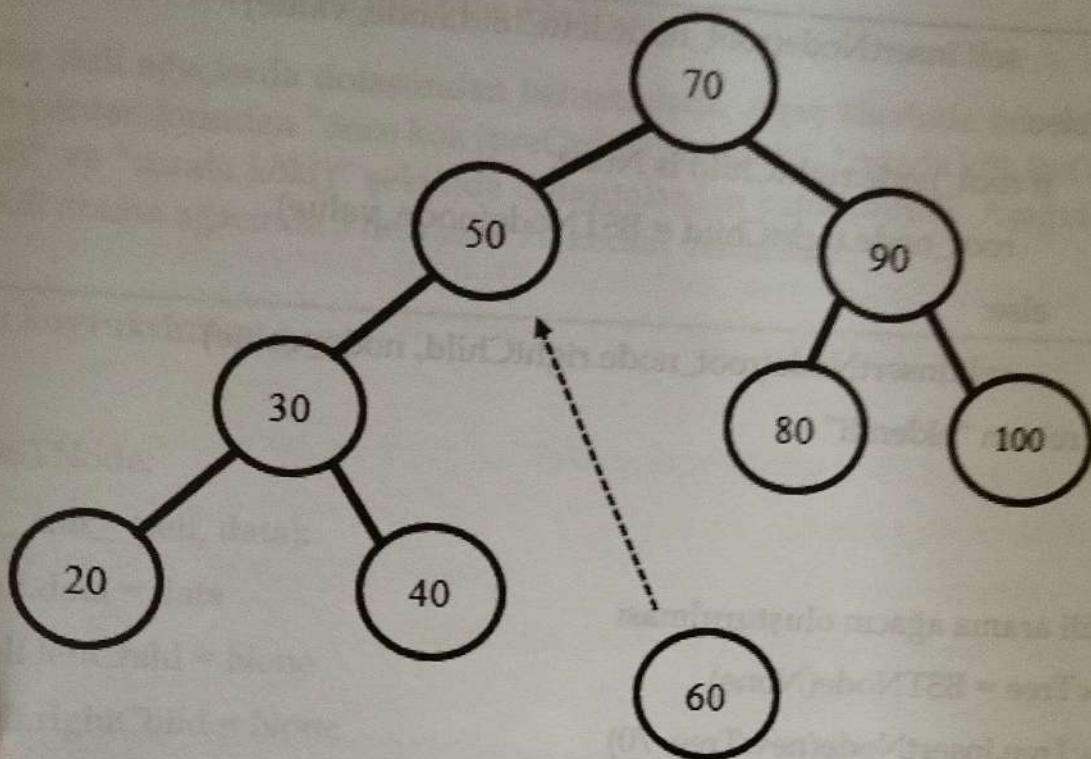
12.7. İkili Arama Ağacına Ekleme

İkili arama ağacına eklenen değer rastgele bir yere eklenmez. Ağaç yapısını bozmadan ilgili yere eklenmesi gereklidir. İkili arama ağacına bir düğüm eklerken düğümün ekleneceği doğru yeri, sahip olduğu değere göre bulmamız gereklidir. Ekleme sırasında ziyaret ettiğimiz düğümlerin değerlerini eklenecek düğümün değeri ile karşılaştırırız.

İkili ağaç boşsa yeni düğüm eklenir. Eklenen değer ikili arama ağacının kökü olarak ayarlanır. Eğer ağaç boş değilse ve yeni düğümün değeri üst düğümün değerinden küçükse, sol alt ağaca gidilir ve doğru konum bulunana kadar karşılaştırma işlemeye devam edilir. Eğer ağaç boş değilse ve yeni düğümün değeri üst ögenin değerinden büyükse sağ alt ağaca gidilir ve doğru konum bulunana kadar karşılaştırma işlemeye devam edilir. Örneğin aşağıdaki ikili arama ağacına "60" değerini eklemek isteyelim.



Eklenecek "60" değerinin kök sırası ile düğümler ile karşılaştırılır. Kök (70) düğümden küçük olduğu için ikili arama ağacın sol alt ağacına geçilir. Hemen sol alt ağacta "50" ile karşılaştırılır. Bu değerden büyük olmasından dolayı bu düğümün sağına hareket edilir. Sağ tarafında herhangi bir düğüm olmamasından dolayı "50" düğümün sağ tarafına bağlanır.



```

import kuyruksinif as queue

class BSTNode:
    def __init__(self, data):
        self.data = data
        self.leftChild = None
        self.rightChild = None

    def insertNode(self, root_node, node_value):
        if root_node.data == None:
            root_node.data = node_value
        elif node_value <= root_node.data:
            if root_node.leftChild is None:
                root_node.leftChild = BSTNode(node_value)
            else:
                self.insertNode(root_node.leftChild, node_value)
        else:
            if root_node.rightChild is None:
                root_node.rightChild = BSTNode(node_value)
            else:
                self.insertNode(root_node.rightChild, node_value)
  
```

```

    self.insertNode(root_node.leftChild, node_value)
else:
    if root_node.rightChild is None:
        root_node.rightChild = BSTNode(node_value)
    else:
        self.insertNode(root_node.rightChild, node_value)
return "eklendi"

```

#İkili arama ağacın oluşturulması

```

newTree = BSTNode(None)
newTree.insertNode(newTree, 70)
newTree.insertNode(newTree, 50)
newTree.insertNode(newTree, 90)
newTree.insertNode(newTree, 30)
newTree.insertNode(newTree, 80)
newTree.insertNode(newTree, 100)
newTree.insertNode(newTree, 20)
newTree.insertNode(newTree, 40)

```

```
print(newTree.insertNode(newTree, 60))
```

Son satırda 60 değeri eklenmiştir. Eklenen her değer ilgili konuma yerleştirilecektir.

Ekleme işlemi özyinelemeli olarak ağacı alt düzeydeki alt ağaçları bölgerek ve düğümün değerine bağlı olarak bir sonraki düzeye geçerek doğru konumu arar. Bu şekilde, fonksiyon çağrılarının sayısı $\log N$ şekline alır. Bu nedenle, ekleme için zaman karmaşıklığı $O(\log N)$ 'dir. Bellek karmaşıklığı da $O(\log N)$ 'dir. Çünkü metod her çağrılığında, bir sonraki çağrımda kendisini tekrar çağırabilmesi için kendisini bellekteki yığında tutmalıdır.

12.8. İkili Arama Ağacında Dolaşım

Daha ikili ağaçlarda dolaşımdan bahsetmiştik. Ağaç üzerinde önce derinlik dolaşım yöntemlerinden "önce kök (preOrderTraversal)", "ortada kök (inOrderTraversal)" ve "sonda kök()" şeklinde dolaşılardan bahsetmiştik. Aşağıdaki örnekte ikili arama ağacında 3 dolaşımı ait kodlar verilmiştir.

```
import kuyruksinif as queue
```

```
class BSTNode:
```

```
    def __init__(self, data):
```

```
        self.data = data
```

```
        self.leftChild = None
```

```
        self.rightChild = None
```

```
    def insertNode(self, root_node, node_value):
```

```
        if root_node.data == None:
```

```
            root_node.data = node_value
```

```
        elif node_value <= root_node.data:
```

```
            if root_node.leftChild is None:
```

```
                root_node.leftChild = BSTNode(node_value)
```

```
            else:
```

```
                self.insertNode(root_node.leftChild, node_value)
```

```
        else:
```

```
            if root_node.rightChild is None:
```

```
                root_node.rightChild = BSTNode(node_value)
```

```
            else:
```

```
                self.insertNode(root_node.rightChild, node_value)
```

```
    return "eklendi"
```

```
def preOrderTraversal(self,root_node):
    if not root_node:
        return
    print(root_node.data,end='->')
    self.preOrderTraversal(root_node.leftChild)
    self.preOrderTraversal(root_node.rightChild)

def inOrderTraversal(self,root_node):
    if not root_node:
        return
    self.inOrderTraversal(root_node.leftChild)
    print(root_node.data,end='->')
    self.inOrderTraversal(root_node.rightChild)

def postOrderTraversal(self,root_node):
    if not root_node:
        return
    self.postOrderTraversal(root_node.leftChild)
    self.postOrderTraversal(root_node.rightChild)
    print(root_node.data,end='->')

#ağacın Oluşturulması
newTree = BSTNode(None)
newTree.insertNode(newTree, 70)
newTree.insertNode(newTree, 50)
newTree.insertNode(newTree, 90)
newTree.insertNode(newTree, 30)
newTree.insertNode(newTree, 80)
newTree.insertNode(newTree, 100)
newTree.insertNode(newTree, 20)
newTree.insertNode(newTree, 40)
```

```
print(newTree.insertNode(newTree, 60))

#PreOrder Dolaşım
print("Önce Kök Dolaşımı")
newTree.preOrderTraversal(newTree)
print("\n\n")

print("Ortada Kök Dolaşımı")
#InOrder Dolaşım
newTree.inOrderTraversal(newTree)
print("\n\n")

print("Sonda Kök Dolaşımı")
#PostOrder Dolaşım
newTree.postOrderTraversal(newTree)
print("\n\n")

def preOrderTraversal(self,root_node):
    if not root_node:
        return
    print(root_node.data,end='->')
    self.preOrderTraversal(root_node.leftChild)
    self.preOrderTraversal(root_node.rightChild)

def inOrderTraversal(self,root_node):
    if not root_node:
        return
    self.inOrderTraversal(root_node.leftChild)
    print(root_node.data,end='->')
    self.inOrderTraversal(root_node.rightChild)
```

```
def postOrderTraversal(self,root_node):  
    if not root_node:  
        return  
    self.postOrderTraversal(root_node.leftChild)  
    self.postOrderTraversal(root_node.rightChild)  
    print(root_node.data,end='->')
```

#ağacın Oluşturulması

```
newTree = BSTNode(None)  
newTree.insertNode(newTree, 70)  
newTree.insertNode(newTree, 50)  
newTree.insertNode(newTree, 90)  
newTree.insertNode(newTree, 30)  
newTree.insertNode(newTree, 80)  
newTree.insertNode(newTree, 100)  
newTree.insertNode(newTree, 20)  
newTree.insertNode(newTree, 40)
```

```
print(newTree.insertNode(newTree, 60))
```

#PreOrder Dolaşım

```
print("Önce Kök Dolaşımı")  
newTree.preOrderTraversal(newTree)  
print("\n\n")
```

print("Ortada Kök Dolaşımı")

```
#InOrder Dolaşım  
newTree.inOrderTraversal(newTree)  
print("\n\n")
```

```
print("Sonda Kök Dolaşımı")
#PostOrder Dolaşım
newTree.postOrderTraversal(newTree)
print("\n\n")

def preOrderTraversal(self,root_node):
    if not root_node:
        return
    print(root_node.data,end='->')
    self.preOrderTraversal(root_node.leftChild)
    self.preOrderTraversal(root_node.rightChild)

def inOrderTraversal(self,root_node):
    if not root_node:
        return
    self.inOrderTraversal(root_node.leftChild)
    print(root_node.data,end='->')
    self.inOrderTraversal(root_node.rightChild)

def postOrderTraversal(self,root_node):
    if not root_node:
        return
    self.postOrderTraversal(root_node.leftChild)
    self.postOrderTraversal(root_node.rightChild)
    print(root_node.data,end='->')

#ağacın Oluşturulması
newTree = BSTNode(None)
newTree.insertNode(newTree, 70)
newTree.insertNode(newTree, 50)
```

```
newTree.insertNode(newTree, 90)
newTree.insertNode(newTree, 30)
newTree.insertNode(newTree, 80)
newTree.insertNode(newTree, 100)
newTree.insertNode(newTree, 20)
newTree.insertNode(newTree, 40)
```

```
print(newTree.insertNode(newTree, 60))
```

```
#PreOrder Dolaşım
```

```
print("Önce Kök Dolaşımı")
```

```
newTree.preOrderTraversal(newTree)
```

```
print("\n\n")
```

```
print("Ortada Kök Dolaşımı")
```

```
#InOrder Dolaşım
```

```
newTree.inOrderTraversal(newTree)
```

```
print("\n\n")
```

```
print("Sonda Kök Dolaşımı")
```

```
#PostOrder Dolaşım
```

```
newTree.postOrderTraversal(newTree)
```

```
print("\n\n")
```

Önce Kök Dolaşımı

70->50->30->20->40->60->90->80->100

Ortada Kök Dolaşımı

20->30->40->50->60->70->80->90->100

Sonda Kök Dolaşımı

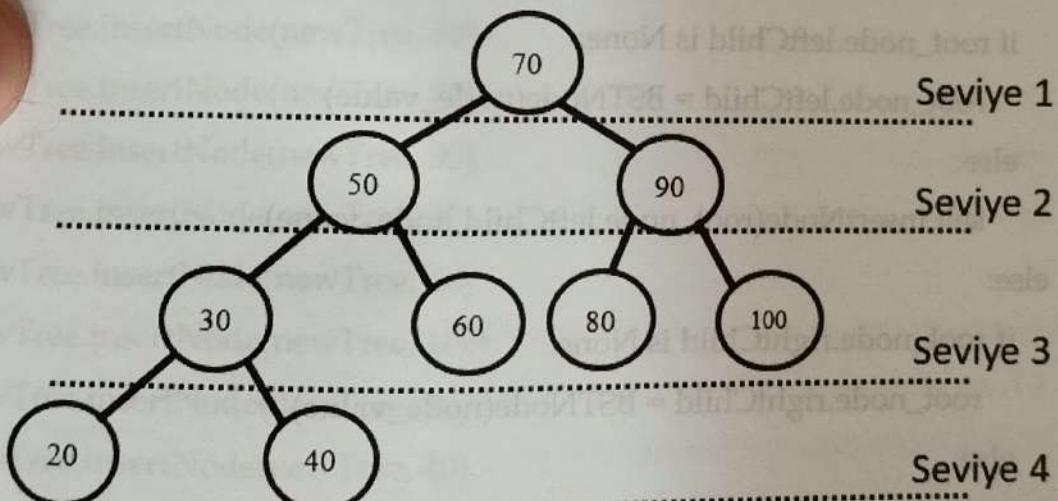
20->40->30->60->50->80->100->90->70

Önce kök, ortada kök ve sonda kök dolaşım yöntemlerinde zaman karmaşıklığı $O(N)$ 'dir. Çünkü ikili arama ağacı üzerindeki tüm düğümler özyinelemeli olarak ziyaret edilmektedir.

Özyinelemeli bir metot kullandığımızda, bir sonraki çağrıyı yapabilmek için önceki çağrı sonuçlarını bellekte tutmak gerekiyor. Bu nedenle, önce kök, ortada kök ve sonda kök dolaşım yöntemlerinde bellek karmaşıklığı da $O(N)$ 'dir.

12.9. İkili Arama AĞACINDA ÖNCE GENİŞLİK DOLAŞIMI

Her düzeyde bulunan düğümler sırası ile kuyruğa alınarak düğümlere ulaşmaktadır. Tüm düğümler sırası ile ziyaret edildiğinden düzey dolaşımı zaman karmaşıklığı $O(N)$ 'dir. Düzey dolaşımı için kuyruk veri yapısı modelinden faydalananlığı için tüm düğümler kuyrukta tutulmuştur. Bu yüzden bellek karmaşıklığı da $O(N)$ 'dir. Düzey dolaşımı örneği aşağıdaki şekilde verilmiştir. Her düzeyde soldan sağa düğümlere tek tek erişilmektedir.



Python kodları aşağıda verilmiştir.

```
import kuyruksinif as queue

class BSTNode:
    def __init__(self, data):
        self.data = data
        self.leftChild = None
        self.rightChild = None

    def insertNode(self, root_node, node_value):
        if root_node.data == None:
            root_node.data = node_value
        elif node_value <= root_node.data:
            if root_node.leftChild is None:
                root_node.leftChild = BSTNode(node_value)
            else:
                self.insertNode(root_node.leftChild, node_value)
        else:
            if root_node.rightChild is None:
                root_node.rightChild = BSTNode(node_value)
            else:
                self.insertNode(root_node.rightChild, node_value)
        return "eklendi"

    def levelOrderTraversal(self, root_node):
        if not root_node:
            return
        else:
```

```

customQueue = queue.Queue()
customQueue.enqueue(root_node)
while not(customQueue.isEmpty()):
    root = customQueue.dequeue()
    print(root.value.data,end='->')
    if root.value.leftChild is not None:
        customQueue.enqueue(root.value.leftChild)
    if root.value.rightChild is not None:
        customQueue.enqueue(root.value.rightChild)

```

#Ağacın Oluşturulması

```

newTree = BSTNode(None)
newTree.insertNode(newTree, 70)
newTree.insertNode(newTree, 50)
newTree.insertNode(newTree, 90)
newTree.insertNode(newTree, 30)
newTree.insertNode(newTree, 80)
newTree.insertNode(newTree, 100)
newTree.insertNode(newTree, 20)
newTree.insertNode(newTree, 40)

print(newTree.insertNode(newTree, 60))

```

#Düzen Dolaşım

```

print("\n\n")
newTree.levelOrderTraversal(newTree)
70->50->90->30->60->80->100->20->40

```

12.10. İkili Arama Ağacında Arama

İkili arama ağacında, bir düğümün sol alt düğümün değeri kendisinden küçükdür. Benzer şekilde bir düğümün sağ alt düğüm değeri de kendisinden büyuktur. Böylece arama yaparken aranılan değeri bir düğüm ile karşılaştırıp büyük olması durumunda düğümün sağ tarafındaki alt ağaçta, küçük olması durumunda düğümün sol alt ağacında arama işlemine devam edilir. Bu şekilde arama işlemi doğrusal arama işlemine göre daha hızlıdır. Çünkü ağaç her düzeyde bölünmektedir. Aşağıdaki Python kodunda ikili arama ağacında arama işlemi `searchNode()` isminde bir metod ile gerçekleştirilmektedir. Arama işlemi özyinelemeli olarak ağaçları bölerek ve düğümün değerine bağlı olarak sonraki düzeye geçildiğinden fonksiyonun kendi kendini çağrıma sayısı $\log N$ kadardır. Dolayısıyla zaman karmaşıklığı $O(\log N)$ 'dir.

```
import kuyruksinif as queue

class BSTNode:

    def __init__(self, data):
        self.data = data
        self.leftChild = None
        self.rightChild = None

    def insertNode(self, root_node, node_value):
        if root_node.data == None:
            root_node.data = node_value
        elif node_value <= root_node.data:
            if root_node.leftChild is None:
                root_node.leftChild = BSTNode(node_value)
            else:
                self.insertNode(root_node.leftChild, node_value)
        else:
            if root_node.rightChild is None:
                root_node.rightChild = BSTNode(node_value)
            else:
```

```

    self.insertNode(root_node.rightChild, node_value)
    return "eklendi"

def searchNode(self,root_node, node_value):
    if root_node.data == node_value:
        print("Kök Düğümde Bulundu")
    elif node_value < root_node.data:
        if root_node.leftChild.data == node_value:
            print("Eleman Bulunmadı")
        else:
            self.searchNode(root_node.leftChild, node_value)
    else:
        if root_node.rightChild.data == node_value:
            print("Eleman Bulundu.")
        else:
            self.searchNode(root_node.rightChild, node_value)

```

#Ağacı Oluştur

```

newTree = BSTNode(None)
newTree.insertNode(newTree, 70)
newTree.insertNode(newTree, 50)
newTree.insertNode(newTree, 90)
newTree.insertNode(newTree, 30)
newTree.insertNode(newTree, 80)
newTree.insertNode(newTree, 100)
newTree.insertNode(newTree, 20)
newTree.insertNode(newTree, 40)
newTree.insertNode(newTree, 60)

```

#arama

```
newTree.searchNode(newTree,70)
```

12.11. İkili Arama Ağacında Silme

Bir ikili arama ağacında silme işlemi yaparken aşağıdaki üç durumdan herhangi biriyle karşılaşabiliriz:

- Silinecek düğüm bir yaprak düğümdür.
- Silinecek düğümün bir alt düğümü var.
- Silinecek düğümün iki çocuğu var.

Silinecek bir düğümün yaprak olması durumunda direkt silinir. Silinecek düğümün bir alt düğümü varsa bu durumda alt düğümü silinecek düğüme kopyalayıp alt düğüm silinir. Silinecek düğümün iki çocuğunun olması durumunda mevcut düğümün sağ alt ağacında minimum değeri buluyoruz ve onu silinecek mevcut düğümle değiştiriyoruz.

Silme işlemi her düzeydeki alt ağaçları bölerek ve düğümün değerine bağlı olarak bir sonraki düzeye geçerek silinecek düğüm aranır. Bu şekilde, fonksiyon çağrılarının sayısı $\log N$ şeklini alır. Bu nedenle, silme için zaman karmaşıklığı $O(\log N)$ 'dir. Silme için gerekli metodlar verilmiştir.

```
import kuyruksinif as queue

#Diğer Gerekli kodlar

def minValueNode(self,bstNode):
    current = bstNode
    while (current.leftChild is not None):
        current = current.leftChild
    return current

def deleteNode(self,root_node, node_value):
    if root_node is None:
        return root_node
    if node_value < root_node.data:
        root_node.leftChild = self.deleteNode(root_node.leftChild, node_value)
    elif node_value > root_node.data:
```

```

root_node.rightChild = self.deleteNode(root_node.rightChild, node_value)
else:
    if root_node.leftChild is None:
        temp = root_node.rightChild
        root_node = None
        return temp

    if root_node.rightChild is None:
        temp = root_node.leftChild
        root_node = None
        return temp

    temp = self.minValueNode(root_node.rightChild)
    root_node.data = temp.data
    root_node.rightChild = self.deleteNode(root_node.rightChild, temp.data)
return root_node

```

#Ekleme İşlemleri

```

print("\n\n")
newTree.levelOrderTraversal(newTree)
newTree.deleteNode(newTree, 50)

```

#LevelOrder Dolaşım

```

print("\n\n")
newTree.levelOrderTraversal(newTree)
70->50->90->30->60->80->100->20->40
70->60->90->30->80->100->20->40

```

12.12. İkili Arama Ağacı'nın Silinmesi

İkili arama ağacının tamamını silmek için, kök düğümü ve sol ve sağ alt referansları Null olarak ayarlamamız yeterlidir. Bu şekilde ayrılan bellek alanını boşaltıyoruz.

```
def deleteBST(self,root_node):
    root_node.data = None
    root_node.leftChild = None
    root_node.rightChild = None
    return "Silindi"
```

İkili arama ağacı üzerinde gerçekleştirilen işlemlere ait zaman ve bellek karmaşıklıkları aşağıdaki tabloda özetlenmiştir.

İşlem	Zaman Karmaşıklığı	Bellek Karmaşıklığı
İkili Arama Ağacı Oluşturma	O(1)	O(1)
Ağaca Ekleme	O(LogN)	O(LogN)
İkili Ağaçta Dolaşım	O(N)	O(N)
Arama İşlemi	O(LogN)	O(LogN)
Silme İşlemi	O(LogN)	O(LogN)
İkili Arama Ağacı Silme	O(1)	O(1)

12.13. Heap Ağacı

Heap üst düğümün kendi çocuk düğümlerden büyük ve küçük olma durumuna göre farklı isimlendirilen özel ağaç yapılarıdır. Heap ağacı, her üst düğümün alt düğümden küçük veya ona eşit olduğu ağaç yapısına Min-Heap ağacı olarak isimlendirilir. Bunun tam tersi her üst düğüm, alt düğümden büyük veya ona eşitse maksimum-heap ağacı olarak isimlendirilir. Python'da heap ağacı için "heapq" modülü kullanılmaktadır. Bu modül içinde heap ağacı için temel işlemler bulunmaktadır. Bu metodlar aşağıda verilmiştir.

heapify(): Bir listeden bir heap ağacı oluşturan metottur. Min-Heap ağacı için bu metodun uygulanması durumunda liste içindeki en küçük eleman 0. sıraya yerleştirilir. Ancak diğer elemanlar sıralanmaz.

heappush(): Tanımlanmış olan heap ağaca bir eleman ekler.

heappop(): Tanımlanmış olan heap ağacından en küçük değeri alır. İlk sıradaki elemanı alır.

heapreplace(): Bu metod, en küçük elemanı yeni bir değerle değiştirir.
Şimdi yukarıdaki metodlara bir örnek verelim.

```
import heapq
```

```
liste = [21,16,72,56,65,34]
```

```
#Heap ağacı oluşturma
```

```
heapq.heapify(liste)
```

```
print(liste)
```

```
#ağaçta eleman ekleme
```

```
heapq.heappush(liste,12)
```

```
print(liste)
```

```
#Ağaçtan silme
```

```
heapq.heappop(liste)
```

```
print(liste)
```

```
#değiştirme
```

```
heapq.heapreplace(liste,10)
```

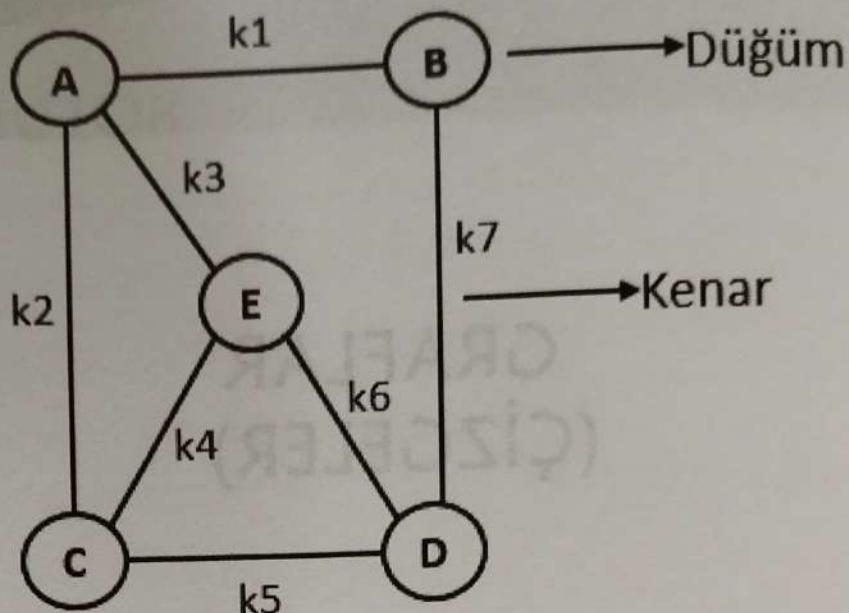
```
print(liste)
```

```
[16, 21, 34, 56, 65, 72]
```

```
[12, 21, 16, 56, 65, 72, 34]
```

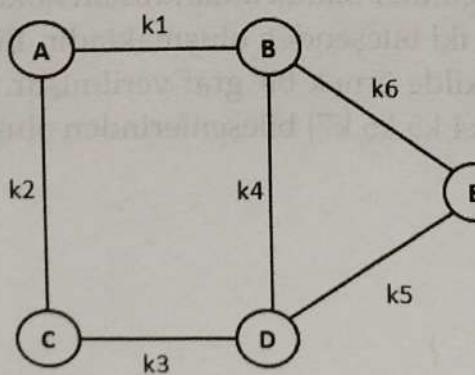
```
[16, 21, 34, 56, 65, 72]
```

```
[10, 21, 34, 56, 65, 72]
```

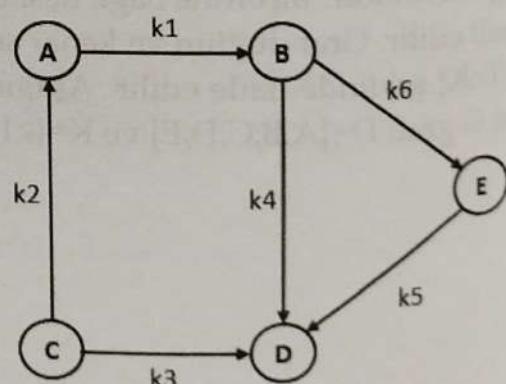


Çeşitli graf türleri bulunmaktadır. Bu graf türleri aşağıda açıklanmıştır.

Yönlendirilmiş (Directed) Graflar: Bu tür graflarda birbirine bağlı düğümler arasında hangi yönde bağlantının olduğu belli graflardır. İki düğüm arasında bağlantı ok işaretleri ile gösterildiği graflar. Aşağıdaki şekilde yönlendirilmiş bir graf verilmiştir. Şekilden görüldüğü gibi düğümler arasında kenar bağlantısının hangi yönde olduğu bellidir. Örneğin A düğümü B düğümüne k1 kenarı ile $A \rightarrow B$ bağlantı söz konusudur. Benzer şekilde E ile D düğümleri arasında bağlantı E'den D'yedir. Şekildeki graf için bağlantı kümesi $B = \{(A,B), (B,E), (E,D), (B,D), (C,A), (C,D)\}$ bağlantı çiftleri bulunmaktadır. Aşağıda bağlantı ve komşuluk matrisleri de verilmiştir. Bağlantı matrisi hangi düğümün hangi düğüme bağlı olduğunu göstermektedir. Komşuluk matrisi ise hangi düğümler arasında bağlantı olduğunu göstermektedir.



Yönsüz Graf



Yönlendirilmiş Graf

Komşuluk Matrisi						Bağlantı Matrisi					
	A	B	C	D	E		A	B	C	D	E
A	-	1	1	-	-	A	-	1	-	-	-
B	1	-	-	1	1	B	-	-	-	1	1
C	1	-	-	1	-	C	1	-	-	1	-
D	-	1	-	-	1	D	-	-	-	-	-
E	-	1	-	1	-	E	-	-	-	1	-

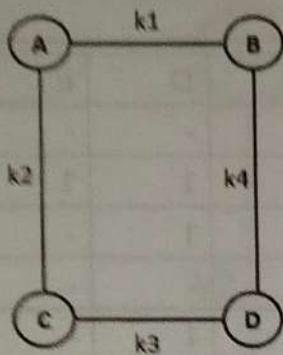
Komşuluk matrisinden görüldüğü gibi hangi düğümler arasında bağlantı varsa "1" olarak işaretlenmiştir. Örneğin A düğümü B ve C düğümlerine bağlılığı olduğu için komşu olarak ifade edilir. Yönlendirilmiş graflarda bağlantı hangi yönde ise ona göre matris oluşturulur. Örneğin A düğümünden B düğümüne bir bağlantı söz konusudur. Dolayısıyla (A,B) hücresi "1" olarak belirtilmiştir. A'dan B'ye bir bağlantı olduğu belirtilmiştir. Ancak B düğümünden A düğümüne bir bağlantı yok.

Yönsüz (Undirected) Graf: Düğüm ve kenarlardan oluşan graflardır. Düğümler arasında bağlantı yönü belli olmayan graf türüdür. Yukarıdaki şekilde görülmektedir.

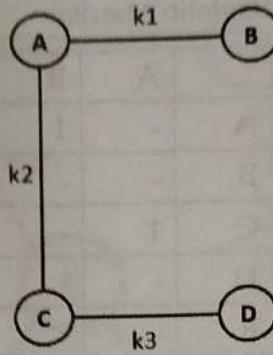
Döngüsel(Cycle) Graflar: Grafın düğümleri arasında kapalı bir çevrimin gerçekleştiği graf türleridir. Graf'ta bazı düğümleri kapalı bir zincirde bağlanacağı anlamına gelir. Başlangıç ve bitiş noktaları aynıdır.

Döngüsel Olmayan Graf: Graf döngüsü olmayan bir graf türüdür. Bağlı bir döngüsel olmayan graf ağaç olarak bilinir.

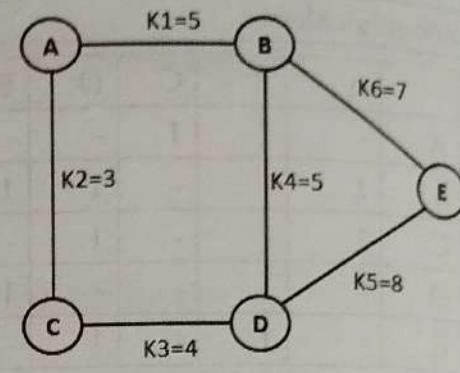
Ağırlıklı (maliyetli) Graf: Her bir kenarına sayısal bir ağırlık verilen bir graf türüdür. Diğer bir tanımla kenarların pozitif sayılar olduğu özel bir etiketli graf türüdür. Aşağıdaki şekilde örnekler verilmiştir.



Döngüsel Graf



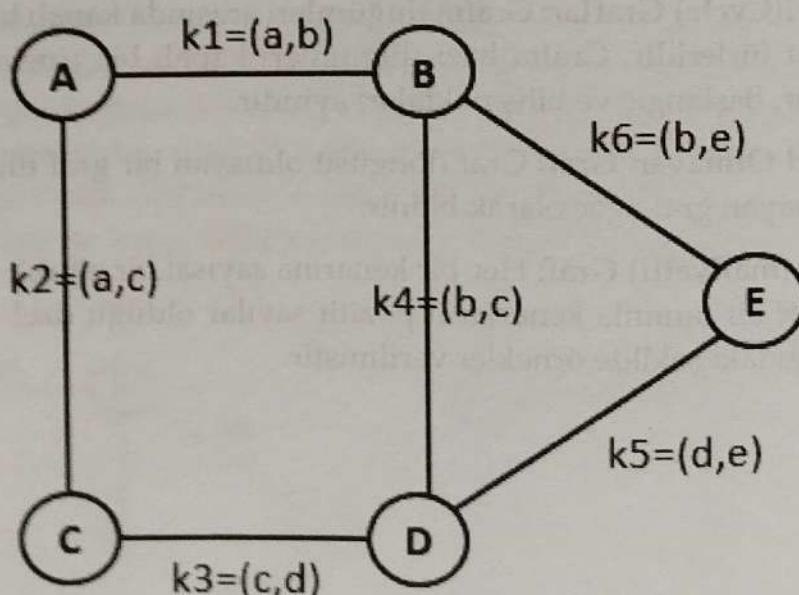
Döngüsel Olmayan Graf



Maliyetli Graf

13.2. Python ile Graf Oluşturma

Bu bölümde Python ile grafların nasıl oluşturulacağını ve graflara ait bir kısım işlemlerin nasıl yapılacağını göreceğiz. Graflar üzerinde temel olarak düğümlerin gösterimi, kenarların gösterimi, düğüm ekleme, kenar ekleme gibi işlemler gerçekleştirilmektedir. Python'da grafları programlayabilmek için sözlük koleksiyonlarından faydalanılmıştır. Sözlükler anahtar-değer mantığı ile çalışan koleksiyonlardır. Düğümler anahtarları, düğümler arasındaki bağlantılar (kenarlar) ise değer olarak belirteceğiz. Python örnekleri aşağıdaki graf üzerinde gerçekleştirilmiştir.



Bu grafta $D=\{A,B,C,D,E\}$ ve $K=\{k1,k2,k3,k4,k5,k6\}$ şeklindedir. Aşağıda grafi tanımlarken hangi düğümün hangi düğümlere bağlı olduğunu düğüm isimleri ile belirtilemiştir. Bu graf sözlük koleksiyonu ile aşağıdaki gibi ifade edilebilir.

```
graf = {  
    "A": ["a", "c"],  
    "B": ["a", "d", "e"],  
    "C": ["a", "d"],  
    "D": ["b", "c", "e"],  
    "E": ["b", "d"]  
}  
  
print(graf)
```

```
{'A': ['a', 'c'], 'B': ['a', 'd', 'e'], 'C': ['a', 'd'], 'D': ['b', 'c', 'e'], 'E': ['b', 'd']}
```

13.3. Düğüm ve Kenarların Gösterilmesi

Graf işlemleri için nesneye yönelik programlama yaklaşımları kullanılmıştır. Aşağıda bir "Graf" sınıfı ve bu sınıf içinde kurucu metot verilmiştir. Kurucu metot içinde görüldüğü gibi Sözlük (Dictionary) koleksiyonu tanımlanmıştır. Düğüm ve kenarları listelemek için dugumListele() ve kenarListele() şeklinde iki metot yazılmıştır.

```
class graph:  
  
    def __init__(self, sozluk=None):  
        if sozluk is None:  
            sozluk = {}  
        self.sozluk = sozluk  
  
    # Düğümlerin Yazdırılması  
  
    def dugumListele(self):  
        return list(self.sozluk.keys())  
  
    # Kenarların Listelenmesi  
  
    def kenarlar(self):  
        return self.kenarListele()
```

```

def kenarListele(self):
    kenarismileri = []
    for d in self.sozluk:
        for dd in self.sozluk[d]:
            if {dd, d} not in kenarismileri:
                kenarismileri.append({d, dd})
    return kenarismileri

graf = {
    "A": ["a", "c"],
    "B": ["a", "d", "e"],
    "C": ["a", "d"],
    "D": ["b", "c", "e"],
    "E": ["b", "d"]
}

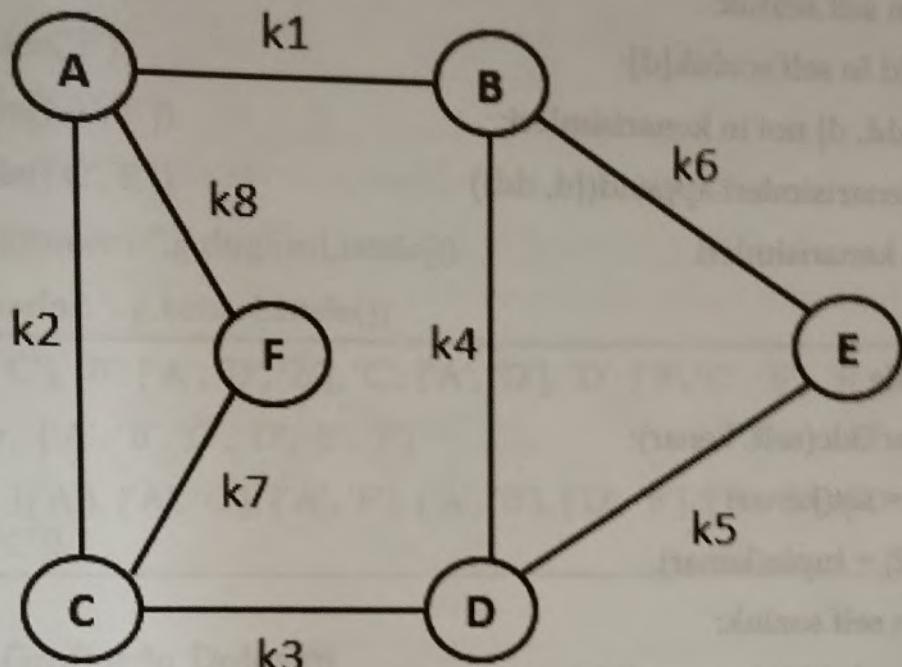
print(graf)
g = graph(graf)
print("Düğümler : ", g.dugumListele())
print('Kenarlar: ', g.kenarListele())
Düğümler: ['A', 'B', 'C', 'D', 'E']
Kenarlar: [{‘a’, ‘A’}, {‘c’, ‘A’}, {‘a’, ‘B’}, {‘d’, ‘B’}, {‘e’, ‘B’}, {‘a’, ‘C’}, {‘C’, ‘d’}, {‘D’, ‘b’}, {‘D’, ‘c’}, {‘e’, ‘D’}, {‘E’, ‘b’}, {‘E’, ‘d’}]

```

13.4. Düğüm ve Kenar Ekleme

Düğüm ve kenar eklemek sözlüğe bir değer eklemektir. Ancak eklenen düğüm bağı olduğu düğüm sayısının birden fazla olması durumunda birden fazla bağlantı veya kenar eklenmiş olacaktır. Dolayısıyla eklenen kenarlar demet veya liste şeklinde verilmesi gereklidir. Yukarıda verilen grafa E düğümü ve k7, k8 kenarların eklenmesini gerçekleştirelim. F düğümünün eklenmesi durumunda A ve C

düğümlerine yeni bağlantılarla eklenmiş olacaktır. A ve F düğümü arasında k8 (a,f), C ile F arasında k7 (c,f) eklenmiş olacaktır. Aşağıdaki graf elde edilmiş olacaktır. Tüm kodlar aşağıdaki örnekte verilmiştir.



class graph:

```

def __init__(self,sozluk=None):
    if sozluk is None:
        sozluk = {}
    self.sozluk = sozluk
  
```

Düğümlerin Yazdırılması

```

def dugumListele(self):
    return list(self.sozluk.keys())
  
```

Kenarların Listelenmesi

```

def kenarlar(self):
    return self.kenarListele()
  
```

```
def kenarListele(self):  
    kenarisimleri = []  
    for d in self.sozluk:  
        for dd in self.sozluk[d]:  
            if {dd, d} not in kenarisimleri:  
                kenarisimleri.append({d, dd})  
    return kenarisimleri
```

#Kenar ekle

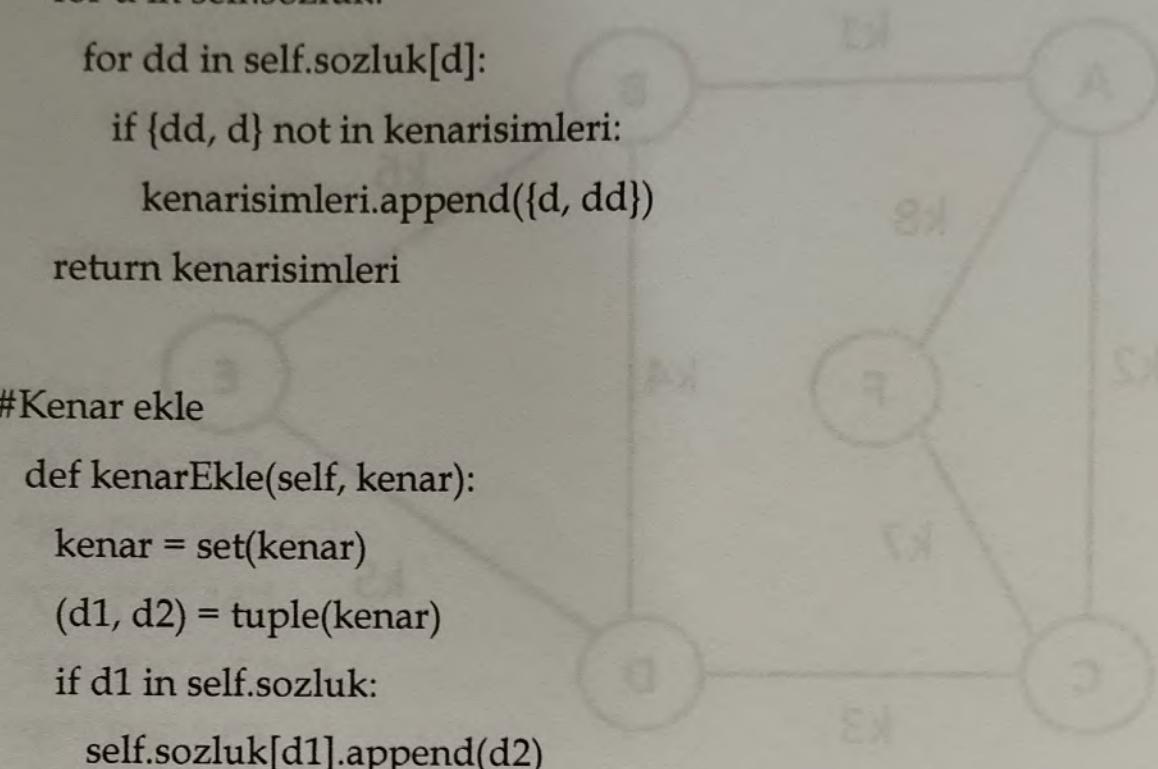
```
def kenarEkle(self, kenar):  
    kenar = set(kenar)  
    (d1, d2) = tuple(kenar)  
    if d1 in self.sozluk:  
        self.sozluk[d1].append(d2)  
    else:  
        self.sozluk[d1] = [d2]
```

#düğüm Ekle

```
def dugumEkle(self, dugum):  
    if dugum not in self.sozluk:  
        self.sozluk[dugum] = []
```

graf = {

```
    "A": ["A", "C"],  
    "B": ["A", "D", "E"],  
    "C": ["A", "D"],  
    "D": ["B", "C", "E"],  
    "E": ["B", "D"]  
}
```



```

print(graf)
g = graph(graf)

g.dugumEkle('F')
g.kenarEkle([('A','F')])
g.kenarEkle([('C','F')])
print("Düğümler : ",g.dugumListele())
print('Kenarlar: ', g.kenarListele())
{'A': ['A', 'C'], 'B': ['A', 'D', 'E'], 'C': ['A', 'D'], 'D': ['B', 'C', 'E'], 'E': ['B', 'D']}
Düğümler: ['A', 'B', 'C', 'D', 'E', 'F']
Kenarlar: [[{'A'}, {'A', 'C'}, {'A', 'F'}, {'A', 'B'}, {'D', 'B'}, {'E', 'B'}, {D, 'C'}, {E, 'D'}, {F, 'C'}]]

```

13.5. Graflarda Dolaşım

Graf düğümleri arasında dolaşım için genel olarak iki yaklaşım kullanılmaktadır.

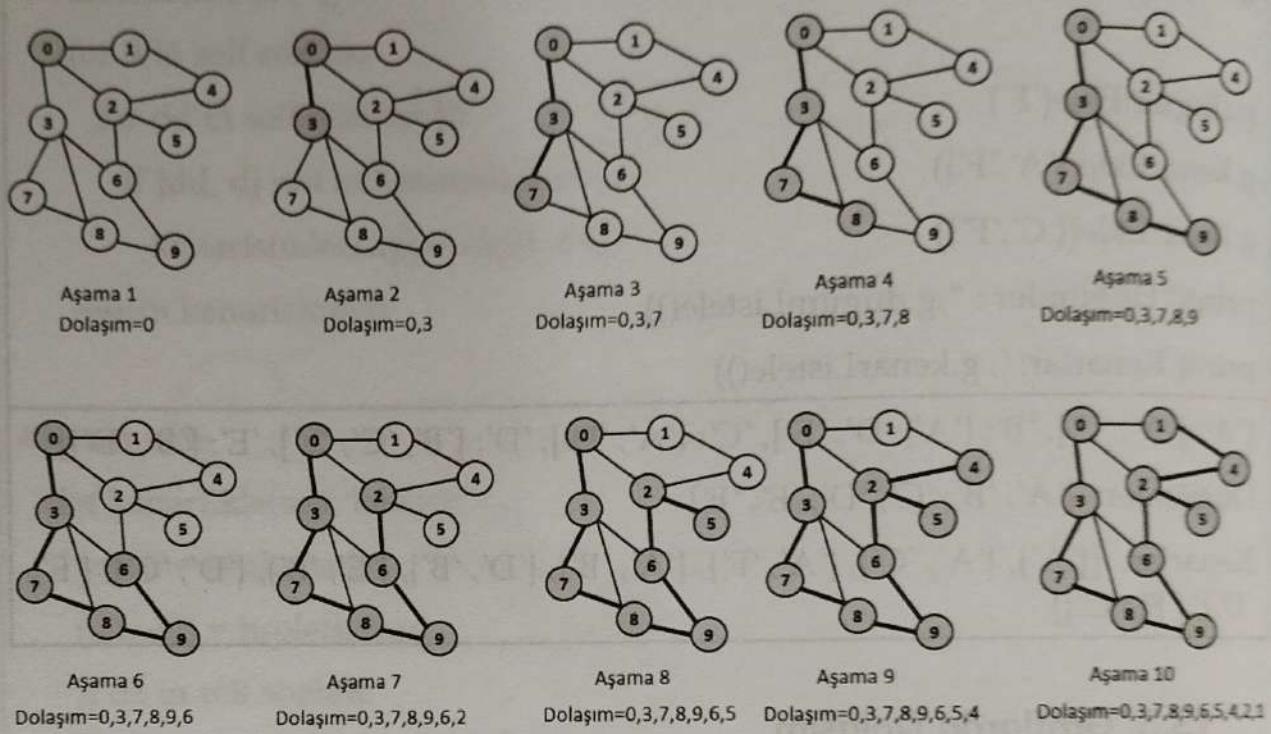
- **Önce Derinlik (Depth first) dolaşımı:** Graf üzerindeki bir X düşümüne gidildikten sonra X düşümün bir komşusu seçilir ve ziyaret edilir. Ardından onun bir komşusu seçilir ve ard arda komşu seçimi yapılarak devam edilir. Komşu kalmayincaya kadar devam edilir.
- **Önce Genişlik (Breadth first) dolaşımı:** Graf üzerindeki bir X düşümüne gidildikten sonra X düşümün sırasıyla tüm komşu düğümlerine gidilir ardından tüm komşu düğümlerin komşu düğümlerine gidilir.

13.5.1. Önce Derinlik (Depth First) Dolaşımı İşlem Adımları

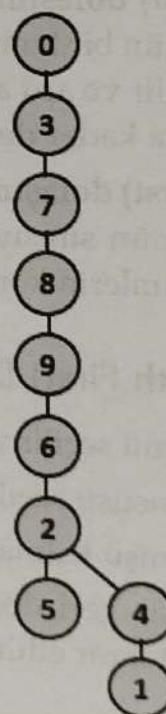
- Önce bir başlangıç X düşümü seçilir ve ziyaret edilir.
- Seçilen X düşümün bir komşusu seçilir ve ziyaret edilir.
- 2. adım ziyaret edilecek komşu kalmayincaya kadar tekrar edilir.
- Komşu kalmadığında tekrar geri dönülür ve önceki ziyaret edilmemiş düğümler için adım 2 ve 3 tekrar edilir.

Once derinlik dolaşımı için bir örnek aşağıda verilmiştir. Örnek graf üzerinde başlangıçta "0" düşümü seçilmiştir. Daha sonra komşuları olan "3" düşümüne

onun komşusu "7" düğümüne şeklinde bir düşümün komşularından birine geçip ziyaret edilmemiş tüm düğümlere ulaşılır.



Yukarıdaki örnek graf üzerinde önce derinlik dolaşımı sonucunda ziyaret edilmiş düğümler aşağıdaki ağaç yapısını oluşturacaktır.

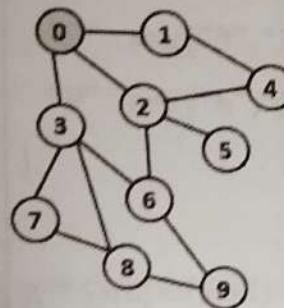


13.5.2. Önce Genişlik (Breadth First) Dolaşımı İşlem Adımları

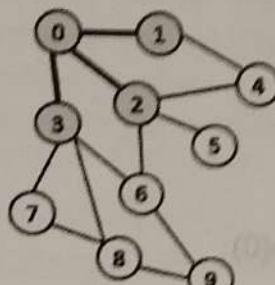
Önce genişlik dolaşımı ağaçlardaki seviye dolaşımı benzemektedir.

- Rastgele bir düğüm seçilir
- Seçilen düğümün sırası ile komşuları ziyaret edilir. Ziyaret edilen her komşu bir kuyruk koleksiyonuna atılır.
- Hiçbir komşu kalmayınca kuyruk içindeki ilk düğüme geçilir. Bu şekilde tüm düğümler ziyaret edilinceye kadar devam edilir.

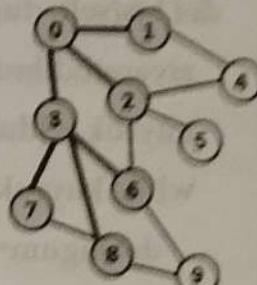
Önce genişlik dolaşımı için bir örnek aşağıda verilmiştir. Örnek graf üzerinde başlangıçta "0" düğümü seçilmiştir. Daha sonra komşuları sırası ile ziyaret edilir. Bu işlem tüm düğümler ziyaret edilinceye kadar devam eder.



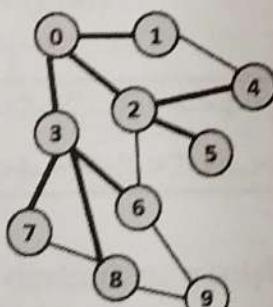
Aşama 1



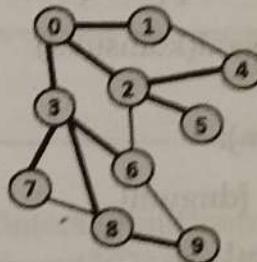
Aşama 2



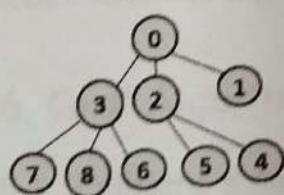
Aşama 3



Aşama 4



Aşama 5



Şimdi yukarıdaki grafi oluşturup üzerinde önce derinlik ve genişlik dolaşım-lara ait kodları yazalım.

```
class Graph:  
    def __init__(self, sozluk=None):  
        if sozluk is None:  
            sozluk = {}  
        self.sozluk = sozluk  
  
    def addEdge(self, dugum, kenar):  
        self.sozluk[dugum].append(kenar)  
  
    def bfs(self, dugum):  
        ziyaretEdilmis = [dugum]  
        kuyruk = [dugum]  
        while kuyruk:  
            deDugum= kuyruk.pop(0)  
            print(deDugum,end="->")  
            for komsular in self.sozluk[deDugum]:  
                if komsular not in ziyaretEdilmis:  
                    ziyaretEdilmis.append(komsular)  
                    kuyruk.append(komsular)  
  
    def dfs(self, dugum):  
        ziyaretEdilmis = [dugum]  
        kuyruk = [dugum]  
        while kuyruk:  
            deDugum = kuyruk.pop()  
            print(deDugum,end="->")  
            for komsular in self.sozluk[deDugum]:  
                if komsular not in ziyaretEdilmis:  
                    ziyaretEdilmis.append(komsular)  
                    kuyruk.append(komsular)
```

```
graf = {
    "0": set(["1", "2", "3"]),
    "1": set(["0", "4"]),
    "2": set(["0", "4", "5", "6"]),
    "3": set(["0", "6", "7", "8"]),
    "4": set(["1", "2"]),
    "5": set(["2"]),
    "6": set(["2", "3", "9"]),
    "7": set(["3", "8"]),
    "8": set(["3", "7", "9"]),
    "9": set(["6", "8"])
}
```

```
g = Graph(graf)
g.bfs("0")
print("\n")
g = Graph(graf)
g.dfs("0")
0->3->1->2->7->8->6->4->5->9
0->2->4->6->9->8->7->5->1->3
```

Önce derinlik ve genişlik dolaşımı yöntemlerinin zaman ve bellek karmaşılığına bakıldığından zaman karmaşıklıklarının $O(\text{Düğüm sayısı} + \text{Kenar sayısı})$ kadardır. Bellek karmaşaklılığı da aynıdır.

13.6. Graf Algoritmaları

13.6.1. Dijkstra's Algoritması

Dijkstra algoritması verilen bir düğümün diğer düğümlere olan en kısa yolu bulmak için kullanılan bir graf algoritmasıdır. Günümüzde oldukça popüler olan bu algoritma, Google Maps, OSPF(Open Shortest Path First) ağ protokolünde, sos-

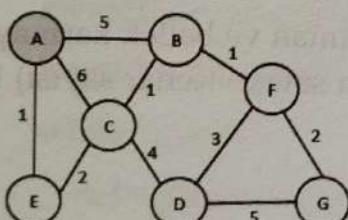
yal ağlarda, telefon ağlarda, oyun programlamada veya ulaşım ağlarında kullanmaktadır.

Algoritmanın temel amacı graf üzerinde bir düğümden başka bir düğüme giderken en ucuz maliyetle nasıl gidilebileceği hesaplamaktır. Negatif olmayan maliyetli graflarda kullanılır.

Algoritma aşağıdaki adımlardan oluşur.

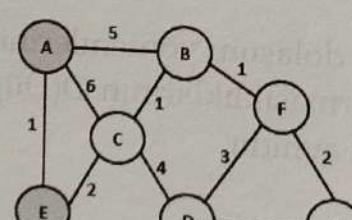
- Algoritma s kaynak düğümünden diğer düğümlere olan mesafeleri ilk değerlerle başlar.
- Algoritma adım adım çalışırken, her adımda mesafe değerlerini günceller.
- Her bir adımda, s düğümünden diğer düğümlere olan en kısa mesafe belirlenir.
- Algoritma, her bir düğümün durumu belirler. Bir düğümün durumu iki özellik içerir. Bunlar, mesafe değeri ve durum etiketidir. Bir düğümün mesafe değeri, s kaynağından o düğüme olan mesafesini gösteren bir sayıdır. Durum etiketi ise bir düğümün mesafe değerinin s düğümüne en kısa yol olup olmadığını belirten etikettir.
- Eğer, bir düğümün mesafe değeri, s düğümünden olan en kısa yolu gösteriyorsa o düğümün durum etiketi **Kalıcıdır**. Diğer durumda, o düğümün durum etiketi **Geçicidir**.
- Algoritma, düğümlerin durumlarını adım adım günceller.
- Her bir adımda bir düğüm **Şu Anki (Current)** olarak işaretlenir.

Dijkstra's Algoritmasını aşağıdaki graf üzerinde anlatalım. A düğümünden G düğümüne en kısa yoldan nasıl gidileceğini hesaplayalım. Aşama aşama A düğümünden G düğümüne geçiş aşağıdaki şekillerde gösterilmiştir. Her aşamada gerçekleştirilen işlemler aşağıda verilmiştir.



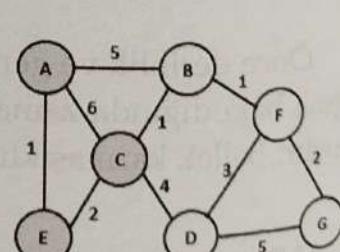
Durum	A	B	C	D	E	F	G
Baş.	0	-	-	-	-	-	-

Aşama 1



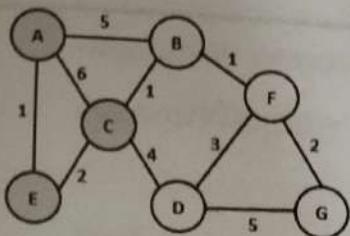
Durum	A	B	C	D	E	F	G
Baş.	0	-	-	-	-	-	-
A	5	6	-	1	-	-	-

Aşama 2



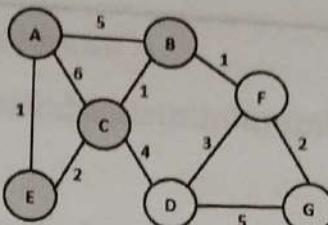
Durum	A	B	C	D	E	F	G
Baş.	0	-	-	-	-	-	-
A(0)	5	6	-	1	-	-	-
E(1)	5	3	-	-	-	-	-

Aşama 3



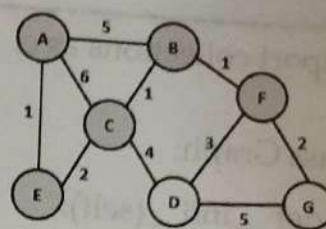
Durum	A	B	C	D	E	F	G
Baş.	0	-	-	-	-	-	-
A(0)	5	6	-	-	-	-	-
E(1)	5	3	-	-	-	-	-
C(3)	4		7	-	-	-	-
B(4)						1	-

Aşama 4



Durum	A	B	C	D	E	F	G
Baş.	0	-	-	-	-	-	-
A(0)	5	6	-	-	-	-	-
E(1)	5	3	-	-	-	-	-
C(3)	4		7	-	-	-	-
B(4)						1	-

Aşama 5



Durum	A	B	C	D	E	F	G
Baş.	0	-	-	-	-	-	-
A(0)	5	6	-	-	-	-	-
E(1)	5	3	-	-	-	-	-
C(3)	4		7	-	-	-	-
B(4)						1	-
F(5)						7	-

Aşama 6

Aşama 1: Başlangıç düğümümüz A olarak belirlenmiştir.

Aşama 2: A düğümünün komşularına olan uzaklıklarını tabloya kayıt edilmiştir. A düğümü B, C ve E düğümlerine komşudur. A düğümü en yakın E düğümüne yakındır için E düğümüne geçiyoruz.

Aşama 3: E düğümünün C düğümüne komşuluğu bulunmaktadır. C düğümünün yeni değeri $1+2=3$ olur. Onceki gelinen mesafe korunur. C değerine E üzerinden gitmek daha az maliyetli olduğu için C değeri üzerinde güncelleme yapıldı. Bir sonraki düzüme geçmek için en küçük mesafeye sahip ve daha önceden ziyaret edilmemiş olan C düğümünden devam edilecektir.

Aşama 4: C düğümün B ve D düğümlerine komşuluğu bulunmaktadır. B düğümün yeni değeri $3+1=4 < 5$ olduğu için B düğümün mesafe değerini değiştiriyoruz. D düğümün yeni değeri $3+4=7$ oldu. Durum tablosunda B düğümünün değeri en düşük olduğu için B düğümünden devam ediyoruz.

Aşama 5: Bu aşamada B düğümünün F düğümüne komşuluğu bulunmaktadır ve $4+1=5'$ ten F düğümün yeni değerini değiştiriyoruz.

Aşama 6: F düğümü D ve H düğümüne komşuluğu bulunmaktadır. H düğümünün değeri $5+2=7$ olmaktadır. D düğümünün mesafe değeri 7 iken $5+3=8$ olması gerekmektedir ama $8>7$ olduğundan mevcut değerini korudu.

Sonuç olarak A düğümünden G düğümüne geçiş A-E-C-B-F-G şeklindeki rota ile 7 birim mesafe uzaklıktadır.

Dijkstra algoritmasına ait Python kodu aşağıdadır. Yukarıdaki graf örnek alınmıştır. Bulunan sonuçların graf için elde edilen değerlere eşit olduğu görülmektedir.

```
import collections as cl

class Graph:
    def __init__(self):
        self.dugumler = set()
        self.kenarlar = cl.defaultdict(list)
        self.mesafeler = {}

    def dugumEkle(self,deger):
        self.dugumler.add(deger)

    def kenarEkle(self, kaynak, hedef, mesafe):
        self.kenarlar[kaynak].append(hedef)
        self.mesafeler[(kaynak, hedef)] = mesafe

    def dijkstra(self,graph, baslangic):
        ziyaretEdilmis = {baslangic : 0}
        yol = cl.defaultdict(list)

        dugumler = set(graph.dugumler)

        while dugumler:
            minimumD = None
            for node in dugumler:
                if node in ziyaretEdilmis:
                    if minimumD is None:
                        minimumD = node
                    elif ziyaretEdilmis[node] < ziyaretEdilmis[minimumD]:
                        minimumD = node
                if minimumD is None:
                    break
                yol[minimumD].append(node)
                del ziyaretEdilmis[minimumD]
                for kenar in graph.kenarlar[minimumD]:
                    if kenar not in ziyaretEdilmis:
                        ziyaretEdilmis[kenar] = ziyaretEdilmis[minimumD] + graph.mesafeler[(minimumD,kenar)]
```

```

dugumler.remove(minimumD)
currentWeight = ziyaretEdilmis[minimumD]

for kenar in graph.kenarlar[minimumD]:
    Amesafe = currentWeight + graph.mesafeler[(minimumD, kenar)]
    if kenar not in ziyaretEdilmis or Amesafe < ziyaretEdilmis[kenar]:
        ziyaretEdilmis[kenar] = Amesafe
        yol[kenar].append(minimumD)
return ziyaretEdilmis, yol

graf = Graph()
graf.dugumEkle("A")
graf.dugumEkle("B")
graf.dugumEkle("C")
graf.dugumEkle("D")
graf.dugumEkle("E")
graf.dugumEkle("F")
graf.dugumEkle("G")
graf.kenarEkle("A", "E", 1)
graf.kenarEkle("A", "C", 6)
graf.kenarEkle("A", "B", 5)
graf.kenarEkle("B", "A", 5)
graf.kenarEkle("B", "C", 1)
graf.kenarEkle("B", "F", 1)
graf.kenarEkle("C", "A", 6)
graf.kenarEkle("C", "B", 1)
graf.kenarEkle("C", "E", 2)
graf.kenarEkle("C", "D", 4)

```

```
graf.kenarEkle("D", "C", 4)
```

```
graf.kenarEkle("D", "F", 3)
```

```
graf.kenarEkle("D", "G", 5)
```

```
graf.kenarEkle("E", "A", 1)
```

```
graf.kenarEkle("E", "C", 2)
```

```
graf.kenarEkle("F", "B", 1)
```

```
graf.kenarEkle("F", "D", 3)
```

```
graf.kenarEkle("F", "G", 2)
```

```
graf.kenarEkle("G", "D", 5)
```

```
graf.kenarEkle("G", "F", 2)
```

```
print(graf.dijkstra(graf, "A"))
```

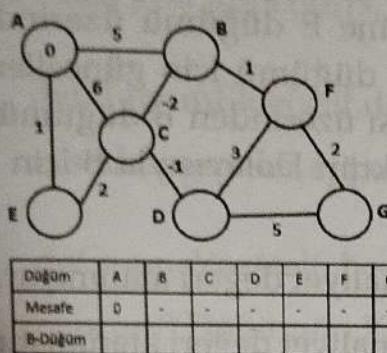
```
{'A': 0, 'E': 1, 'C': 3, 'B': 4, 'D': 7, 'F': 5, 'G': 7}
```

13.6.2. Bellman-Ford's Algoritması

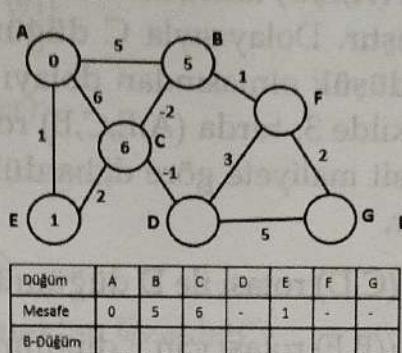
Graflarda en kısa yolu bulmak için kullanılan bir yaklaşımdır. Bir graftaki belirli bir köşe (kaynak olarak adlandırılır) ile diğer tüm köşeler (hedef olarak adlandırılır) arasındaki yolları, aralarındaki toplam mesafe minimum olacak şekilde rota bilgilerini çıkaran bir graf algoritmasıdır. İki düğüm arasındaki en kısa yoldan geçilirken her düğümün ziyaret edilmesi gerekli değildir. Bellman-Ford'un algoritması, dinamik programmanın bir örnek yaklaşımıdır. Bir başlangıç düğüm ile başlar ve bir kenarın ulaşabileceği diğer düğüm uzaklıklarını hesaplar. Bellman-Ford'un Algoritması, Dijital Haritalama Hizmetleri, Sosyal Ağ Uygulamaları, Telefon Ağları, Bilgisayar Ağları gibi çeşitli gerçek uygulamaları mevcuttur. Algoritma maliyetli graflar üzerinde çalışır ve bir anlamda Dijkstra algoritmasının iyileştirilmiş olarak düşünülebilir. Algoritma aslında Dijkstra algoritmasından daha kötü bir performansa sahiptir ancak graftaki maliyetlerin eksisi(-) olması durumunda Dijkstra'nın tersine başarılı çalışır. Algoritmanın temel aşamaları aşağıda verilmiştir.

- Maliyetli bir graf için başlangıç ve hedef düğümler belirlenir. Başlangıç düğümü maliyeti sıfır(0) diğer düğümler için maliyet= ∞ olarak atanır.
- Graftaki tüm düğümler için bir uzaklık tahmini oluşturulur.
- En az maliyetli yol hesaplanana kadar tüm düğümler üzerinden değişiklikler yapılır.

Algoritmanın çalışmasını bir örnek graf üzerinde gösterelim. Graf ve algoritmanın aşamalarına ait şekiller aşağıda verilmiştir.

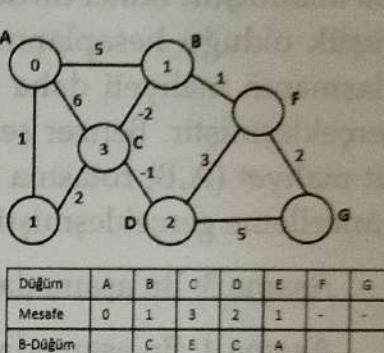


Aşama 1

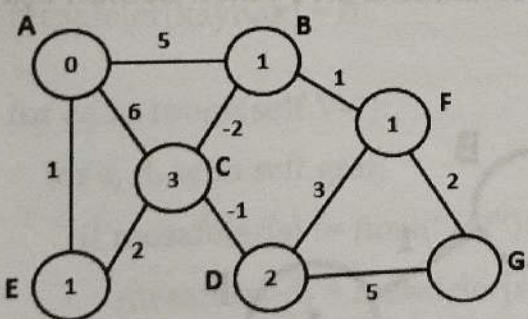


Düyüküm	A	B	C	D	E	F	G
Mesafe	0	5	3	-	1	-	-
B-Düyüküm			E		A		

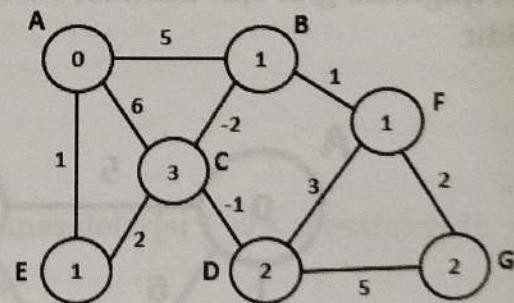
Aşama 3



Aşama 2



Aşama 4



Aşama 5

Düyüküm	A	B	C	D	E	F	G
Mesafe	0	1	3	2	1	1	3
B-Düyüküm	C	E		A	B	F	

Yukarıdaki eksik değerlere sahip graf üzerinde algoritmanın çalışma aşamaları aşağıda anlatılmıştır. A (kaynak) düğümünden G (hedef) düğümüne geçilecektir.

Aşama 1: Bu aşamada düğümlere değer ataması yapılıyor. A başlangıç düğümüne 0, doğrudan erişilen düğümlere erişim değerleri ve erişilemeyen düğümlere “-” sonsuz değeri atanmıştır.

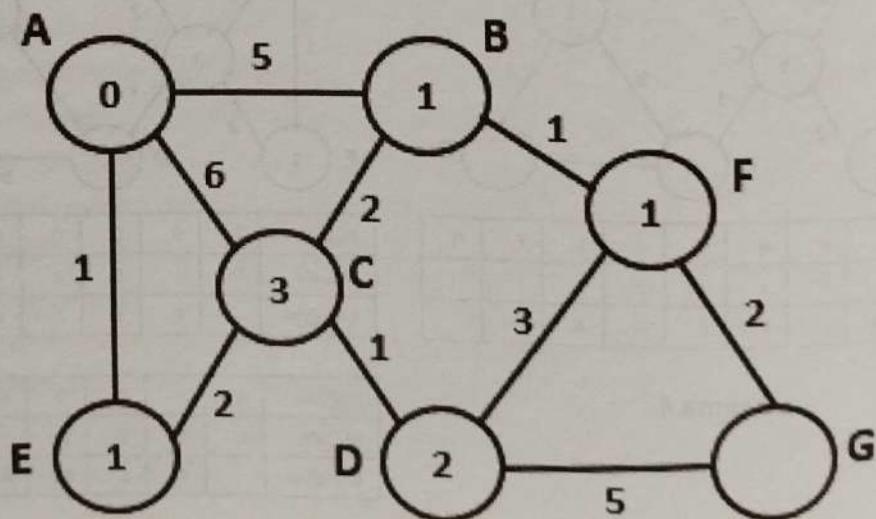
Aşama 2: Bu aşamada A düşümünün komşu olduğu B, C ve E düğümlerine maliyet değerleri atanacaktır. Birinci turda $(A,B)=5$, $(A,C)=6$ ve $(A,E)=1$ olarak atanmıştır. İkinci turda (A,E,C) üzerinden maliyetin (A,C) maliyetinden daha düşük olduğu hesaplanmıştır. Dolayısıyla C düşümüne E düşümü üzerinden ulaşmanın maliyeti daha düşük olmasından dolayı C düşümü için güncelleme gerçekleşmiştir. Benzer şekilde 3. turda (A,E,C,B) rotası üzerinden B düşümüne ait maliyet (A,B) rotasına ait maliyete göre daha düşüktür. Dolayısıyla B için de güncelleme gerçekleşmiştir.

Aşama 3: Bu aşamada (C,D) rotası ile D düşümü maliyet değeri atanmıştır.

Aşama 4: Bu aşamada $((B,F)$ rotası için F düşümü maliyet değeri atanmıştır.

Aşama 5: Son aşamada 1. turda (D,G) rotası ile G düşümü maliyet değeri atanmıştır. Ancak 2. turda (F,G) düşümü rotası için maliyet değerinin daha düşük olduğu hesaplanmıştır. Buna göre güncellemeler gerçekleştirilmiştir.

Sonuç olarak A düşümünden diğer düğümlere ulaşım için gerekli mesafe bilgileri $(A,B)=1$, $(A,C)=3$, $(A,D)=2$, $(A,E)=1$, $(A,F)=1$ ve $(A,G)=3$ olarak bulunmuştur. Aşağıdaki graf için Bellman-Ford algoritmasına ait Python kodları aşağıdaki gibidir.



```
class Graph:
```

```
    def __init__(self, kose):
```

```
        self.V = kose
```

```
        self.graf = []
```

```
        self.dugumler = []
```

```
    def kenarEkle(self, s, d, w):
```

```
        self.graf.append([s, d, w])
```

```
    def dugumEkle(self, deger):
```

```
        self.dugumler.append(deger)
```

```
    def sonucYaz(self, hedef):
```

```
        print("Düğümün Kaynaktan Uzaklığı")
```

```
        for anahtar, deger in hedef.items():
```

```
            print(' ' + anahtar, ': ', deger)
```

```
    def bellmanFord(self, kaynak):
```

```
        mesafeler = {i : float("Inf") for i in self.dugumler}
```

```
        mesafeler[kaynak] = 0
```

```
        for aa in range(self.V-1):
```

```
            for s, d, w in self.graf:
```

```
                if mesafeler[s] != float("Inf") and mesafeler[s] + w < mesafeler[d]:
```

```
                    mesafeler[d] = mesafeler[s] + w
```

```
        for s, d, w in self.graf:
```

```
            if mesafeler[s] != float("Inf") and mesafeler[s] + w < mesafeler[d]:
```

```
                print("Negatif Çevrim")
```

```
            return
```

self.sonucYaz(mesafeler)

```

graf = Graph(7)

graf.dugumEkle("A")
graf.dugumEkle("B")
graf.dugumEkle("C")
graf.dugumEkle("D")
graf.dugumEkle("E")
graf.dugumEkle("F")
graf.dugumEkle("G")

graf.dugumEkle("A", "B", 1)
graf.dugumEkle("A", "C", 6)
graf.dugumEkle("A", "D", 5)
graf.dugumEkle("B", "C", 2)
graf.dugumEkle("B", "D", 3)
graf.dugumEkle("B", "E", 1)
graf.dugumEkle("C", "D", 4)
graf.dugumEkle("C", "E", 5)
graf.dugumEkle("C", "F", 6)
graf.dugumEkle("D", "E", 2)
graf.dugumEkle("D", "F", 3)
graf.dugumEkle("E", "F", 1)
graf.dugumEkle("F", "G", 1)

graf.kenarEkle("A", "B", 5)
graf.kenarEkle("A", "C", 6)
graf.kenarEkle("B", "C", 2)
graf.kenarEkle("B", "D", 3)
graf.kenarEkle("B", "E", 1)
graf.kenarEkle("C", "D", 4)
graf.kenarEkle("C", "E", 5)
graf.kenarEkle("C", "F", 6)
graf.kenarEkle("D", "E", 2)
graf.kenarEkle("D", "F", 3)
graf.kenarEkle("E", "F", 1)
graf.kenarEkle("F", "G", 1)

graf.kenarEkle("A", "B", 5)
graf.kenarEkle("A", "C", 6)
graf.kenarEkle("B", "C", 2)
graf.kenarEkle("B", "D", 3)
graf.kenarEkle("B", "E", 1)
graf.kenarEkle("C", "D", 4)
graf.kenarEkle("C", "E", 5)
graf.kenarEkle("C", "F", 6)
graf.kenarEkle("D", "E", 2)
graf.kenarEkle("D", "F", 3)
graf.kenarEkle("E", "F", 1)
graf.kenarEkle("F", "G", 1)

```

graf.kenarEkle("P", "B", 1)
graf.kenarEkle("P", "D", 3)
graf.kenarEkle("P", "G", 2)

graf.kenarEkle("G", "D", 5)
graf.kenarEkle("G", "P", 2)

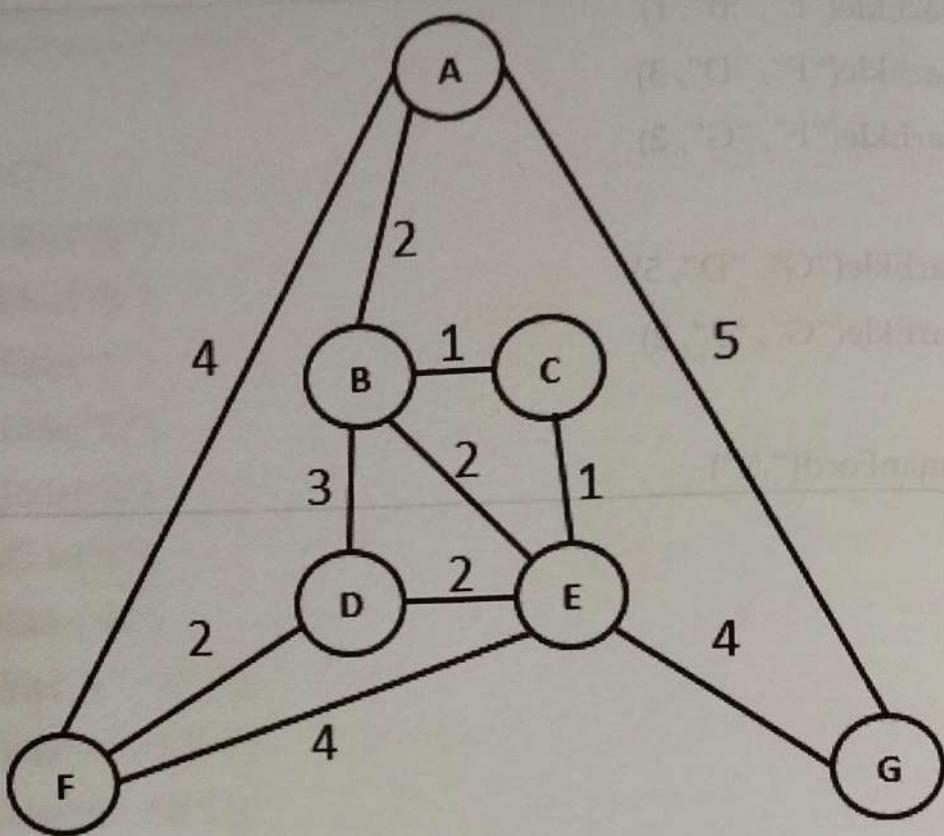
graf.bellmanFord("A")

A :	0
B :	1
C :	3
D :	4
E :	1
F :	2
G :	4

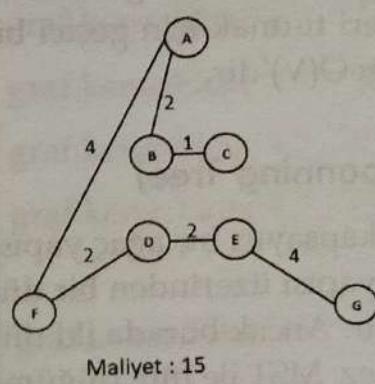
Bellman-Ford'un algoritmasının zaman karmaşıklığı $O(V^2)$ dir. Burada "V" düğüm sayısı ve "E" grafın kenar sayısıdır. Bu karmaşıklığın nedeni biri düğümler üzerinde, diğer de kenarlar üzerinde yineLENEN, içe geçmiş iki döngü kullanılmıştır. Algoritmanın yürütülmESİ strasunda düğümleri tutmak için geçici bir veri yapısına ihtiyacımız olduğundan, bellek karmaşıklığı $O(V)$ 'dir.

13.6.3. Minimum Kapsama Ağacı (Minimum Spanning Tree)

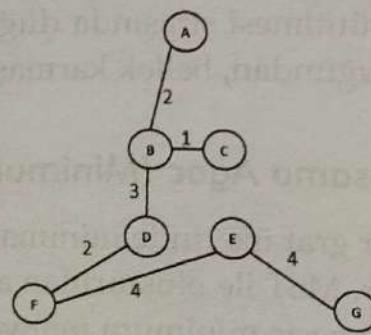
MST, ağırlıklara sahip bir graf üzerinde minimal ve kapsayıcı bir ağaç yapısı ortaya koyan bir algoritmadır. MST ile oluşturulmuş ağaç yapısı üzerinden bir düğüm istenilen bir düzgüme geçiş minimum maliyetlidir. Ancak burada iki düğüm arasındaki maliyetin minimum olmasına dikkat etmez. MST ile tüm düğümlerin dolasımının minimum olması amaçlanmaktadır. Kapsayan ağaç (spanning tree) graf içindeki tüm düğümleri kapsayan ve çevrim olmayan bir ağaç yapısıdır. Bir graf yapısı üzerinden birden fazla ağaç yapısı oluşturulabilir. Ancak MST ile düğümler arası toplam mesafenin minimum ağaç yapısı edilmektedir. MST algoritmları telefon kablolama alt yapılarında, lojistik, kanser görüntüleme, astronomi gibi gerçek hayat uygulamalarında kullanılmaktadır.



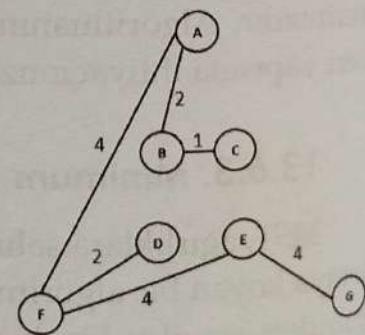
Yukarıda verilen graf örneğinden çeşitli şekillerde ağaçlar oluşturulabilir. Birkaç örnek ağaç aşağıdaki şekilde verilmiştir. Ancak minimum maliyetli ağaçın bulunması gereklidir. Telefon alt yapısı gibi gerçek uygulamalara bakıldığında MST algoritmalarının ne kadar önemli olduğu anlaşılmaktadır.



Maliyet : 15



Maliyet : 16



Maliyet : 17

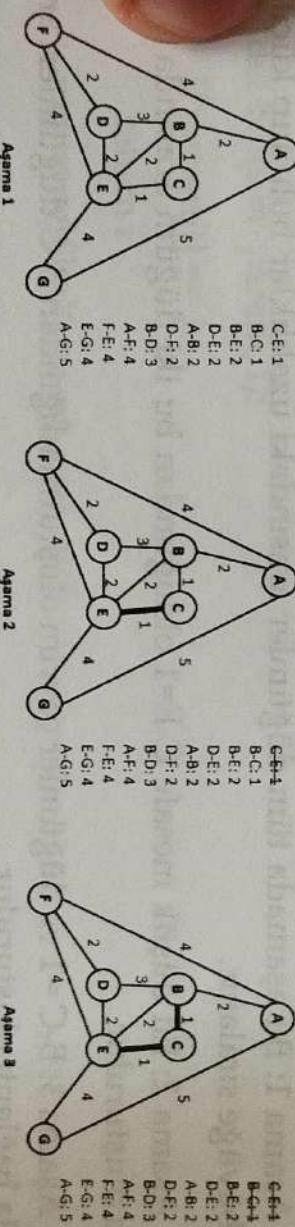
Yukarıdaki graftan görüldüğü çeşitli maliyetlerde alt ağaçlar oluşturulabilir.

13.6.4. Kruskal Algoritması

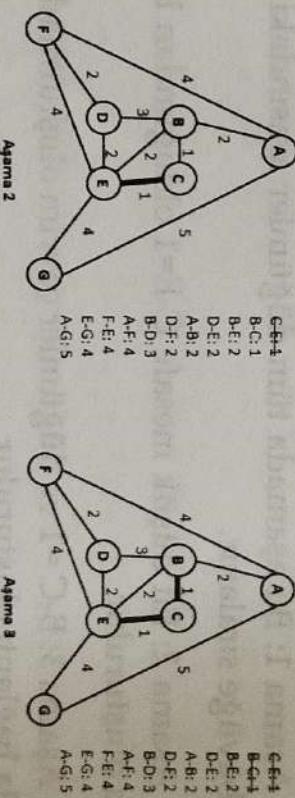
Kruskal algoritması bir G grafi içinde minimum kapsama ağacının bulunmasını sağlayan bir algoritmadır. Asıl amaç tüm düğümleri dolaşım sağlayacak minimum maliyetli bir ağaç oluşturmak. Kruskal algoritması ile minimum kapsama ağacının bulunma aşamaları şu şekildedir.

1. Kenar uzunlukları küçükten büyüğe doğru olacak şekilde düğümler alt alta sıralanır.
2. En Küçük kenardan seçilir, graf içinde döngü oluşturup oluşturulmadığı kontrol edilir. Eğer döngü oluşuyorsa kenar dahil edilir. Fakat bir döngü oluşuyorsa o kenar kullanılmaz.
3. ($N-1$) kadar kenar sayısı elde edilene kadar 2. adım tekrarlanır.

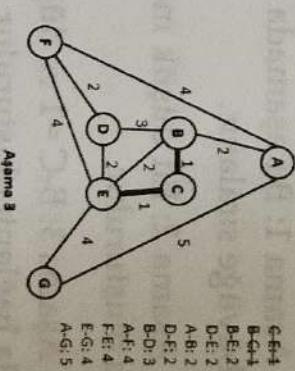
Şimdi Kruskal algoritmasını örnek bir graf üzerinde aşama aşama nasıl çalıştığını göstermiş olalım.



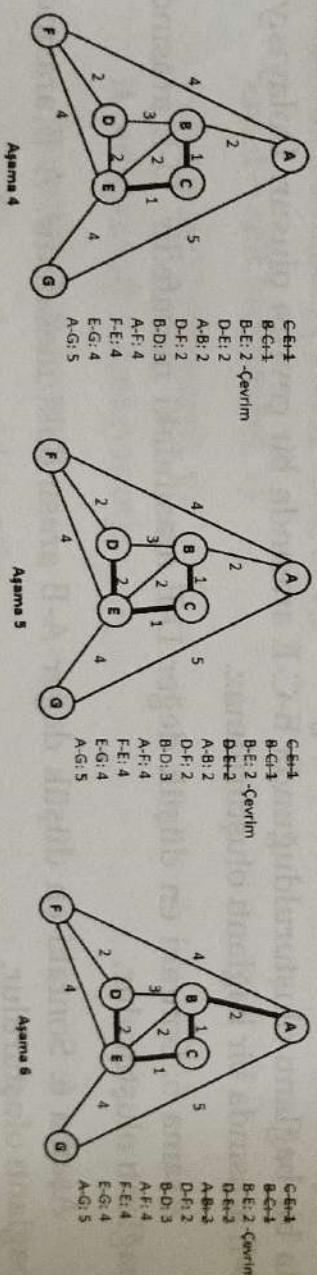
Aşama 1



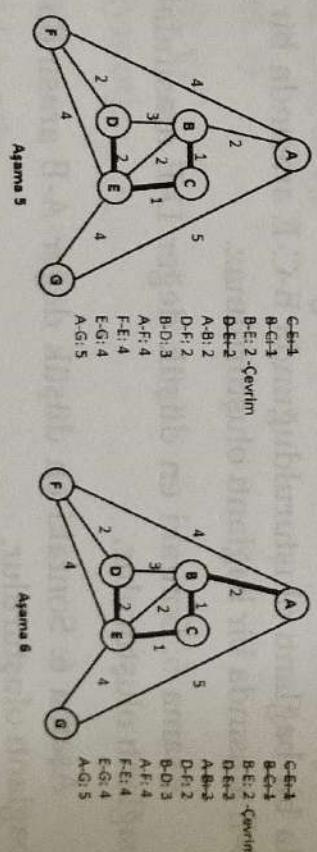
Aşama 2



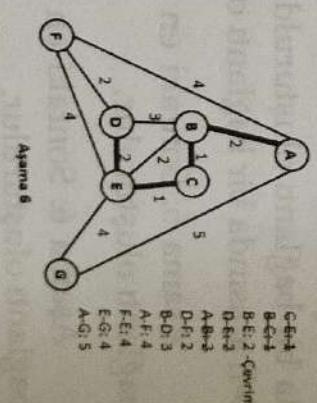
Aşama 3



Aşama 4



Aşama 5



Aşama 6

C-E:1

B-C:1

B-E:2

D-E:2

A-B:2

D-F:2

B-D:3

A-F:4

F-E:4

E-G:4

A-G:5

C-E:1

B-C:1

B-E:2

D-E:2

A-B:2

D-F:2

B-D:3

A-F:4

F-E:4

E-G:4

A-G:5

C-E:1

B-C:1

B-E:2

D-E:2

A-B:2

D-F:2

B-D:3

A-F:4

F-E:4

E-G:4

A-G:5

C-E:1

B-C:1

B-E:2

D-E:2

A-B:2

D-F:2

B-D:3

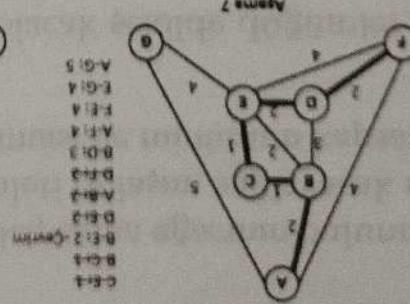
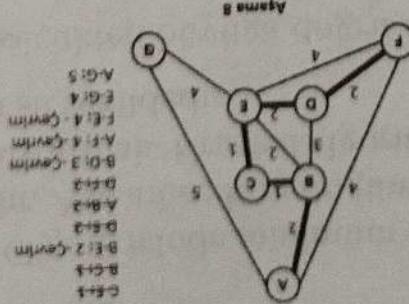
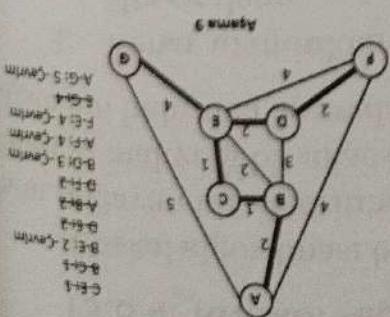
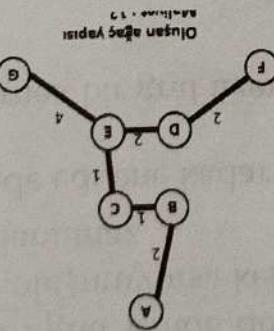
A-F:4

F-E:4

E-G:4

A-G:5

- Asama 1: Bu asamada tüm diğimler arasındaki uzaklıklar bulunup kılınır.
 Asama 2: En dişik mesafe C-E=1 olduğundan bu iki diğim arasımda bağıt olusturulur.
 Asama 3: B-C=1 ve diğimler gevrim olusturmadığında B-C diğimler arasıbağıt olusturulur.
 Asama 4: Sonraki en dişik deger B-E arasımdaki mesafedir ancak B-E arasımda bir baglantı olusturulmaz.
 Asama 5: Sonraki en dişik deger D-E arasımdaki mesafedir. D-E arasımda B-E arasımda bir baglantı olusturulmaz.
 Asama 6: Sonraki en dişik deger A-B arasımdaki mesafedir. A-B arasımda baglantı olusturulur.
 Asama 7: Sonraki en dişik deger D-F arasımdaki mesafedir. D-F arasımda baglantı olusturulur.
 Asama 8: Sonraki asama da en dişik mesafeler B-D, A-F ve F-E dişim gibi arasımdadır. Ancak bunlardan biri arasımda bile baglantı gereklesitirildiginde gevrim arasımdadır.
 Asama 9: Sonraki en dişik deger E-G arasımda A-C arasımda bir baglantı olusturulur. E-G arasımda gevrim baglantı olusturulur. Son asama da A-C arasımda bir baglantı olusturulursa gevrim baglantı olusturulur.



Cevrim oluşturan düğümler arasında bir bağlantı gerçekleştirilemez. Aşamalarınacunda maliyeti=12 olan bir ağaç elde edilmiş olacaktır.

Yukarıdaki graf örneği için Kruskal algoritmasına ait Python kodları aşağıda verilmiştir. A,B,C,..G düğümleri için 0-6 arasındaki numaralar kullanılmıştır.

```
class Graph:  
    def __init__(self, dugumler):  
        self.V = dugumler  
        self.graf = []  
  
    def kenarEkle(self, u, v, w):  
        self.graf.append([u, v, w])  
  
    def bul(self, parent, i):  
        if parent[i] == i:  
            return i  
        return self.bul(parent, parent[i])  
  
    def birlesim(self, parent, rank, x, y):  
        xroot = self.bul(parent, x)  
        yroot = self.bul(parent, y)  
        if rank[xroot] < rank[yroot]:  
            parent[xroot] = yroot  
        elif rank[xroot] > rank[yroot]:  
            parent[yroot] = xroot  
        else:  
            parent[yroot] = xroot  
            rank[xroot] += 1
```

```
# Kruskal algoritması  
def kruskal_algo(self):  
    sonuc = []  
    i, e = 0, 0  
    self.graf = sorted(self.graf, key=lambda item: item[2])  
    parent = []  
    rank = []  
    for node in range(self.V):  
        parent.append(node)  
        rank.append(0)  
    while e < self.V - 1:  
        u, v, w = self.graf[i]  
        i = i + 1  
        x = self.bul(parent, u)  
        y = self.bul(parent, v)  
        if x != y:  
            e = e + 1  
            sonuc.append([u, v, w])  
            self.birlesim(parent, rank, x, y)  
    for u, v, weight in sonuc:  
        print("%d - %d: %d" % (u, v, weight))
```

g = Graph(7)

```
g.kenarEkle(0, 1, 2)  
g.kenarEkle(0, 5, 4)  
g.kenarEkle(0, 6, 5)
```

g.kenarEkle(1, 0, 2)

g.kenarEkle(1, 2, 1)

g.kenarEkle(1, 3, 3)

g.kenarEkle(1, 4, 2)

g.kenarEkle(2, 1, 1)

g.kenarEkle(2, 4, 2)

g.kenarEkle(3, 1, 3)

g.kenarEkle(3, 4, 2)

g.kenarEkle(3, 5, 2)

g.kenarEkle(4, 1, 2)

g.kenarEkle(4, 2, 1)

g.kenarEkle(4, 3, 2)

g.kenarEkle(4, 5, 4)

g.kenarEkle(4, 6, 4)

g.kenarEkle(5, 0, 4)

g.kenarEkle(5, 3, 2)

g.kenarEkle(5, 4, 4)

g.kenarEkle(6, 0, 5)

g.kenarEkle(6, 4, 4)

g.kruskal_algo()

1(B) - 2(C): 1

4(E) - 2(C): 1

0(A) - 1(B): 2

3(D) - 4(E): 2

3(D) - 5(F): 2

4(E) - 6(G): 4

Bulunan sonuçlar ile şekil üzerinde bulunan en kısa kapsama ağacının aynı olduğu görülmektedir.

Kruskal algoritmasının zaman karmaşıklığı $O(V + E \log E + EV)$ şeklindektir, burada "V" düğüm sayısı ve "E" graftaki kenar sayısıdır. Veri öğelerini geçici olarak depolamak için ek belleğe ihtiyacımız olduğundan, bellek karmaşıklığı $O(V * E^2)$ dir.

13.6.5. Prim Algoritması

Prim'in algoritması, bir graf içinde minimum kapsama ağacı bulmak için kullanılan bir algoritmadır. Verilen ağırlıklı yönsüz graf için bir minimum kapsayan ağaç bulur. Minimum kapsama ağacı, mümkün olan en az kenar sayısına sahip grafın tüm düğümlerini içeren, yönsüz ağaç veya alt graftır Oluşturulan ağaç graftaki tüm düğümleri içermelidir. Herhangi bir düğüm atlanırsa bu bir kapsama ağacı olmayacağındır. Kapsama ağacında çevrimler olmaz. Grafın N sayıda düğümü varsa tüm graftan oluşturulan kapsama ağaçlarının toplam sayısı $N^{(N-2)}$ kadardır. Ancak kenarların toplamı mümkün olduğunda minimum olan kapsama ağacın bulunması gerekdir. Bir grafın birden çok kapsayan ağacı olabilir, ancak yalnızca bir benzersiz minimum yayılan ağacı olabilir. Prim'in algoritması, tek kaynak düğümle başlar ve daha sonra kaynak düğümün tüm bitişik düğümlerini tüm bağlı kenarlarıyla birlikte tarar. Grafları incelerken minimum ağırlığa sahip ve grafta döngülere neden olmayacak kenarlar seçilmeli.

Prim algoritması, en uygun çözümü bulmak için temel olarak açgözlü algoritma yaklaşımını kullanmaktadır. Bu algoritma çeşitli gerçek hayat problemlerinde kullanılmaktadır. Ağ tasarımindan, demiryolu, karayolu hatlarında, elektrik alt şebekelerinde, sulama kanal tasarımindan, kümeleme işlemlerinde, gezgin satıcı probleminde kullanılmaktadır. Prim algoritmasının uygulanması aşağıdaki adımlardan oluşmaktadır.

- Herhangi bir düğümü kaynak olarak alıp ağırlığını sıfır (0) olarak ayarlayın.
- Diğer tüm düğümlerin değerlerini “-” olarak ayarlayın. Seçilen düğüme komşu her düğüm için, mevcut ağırlık mevcut kenarın maliyet değerinden büyükse o zaman onu mevcut kenarın maliyeti ile değiştiririz. Ardından, mevcut düğümü ziyaret edildi olarak işaretliyoruz.
- Bu adımları verilen tüm düğümler için artan maliyet sırasına göre tekrarlayın.

Şimdi örnek bir graf üzerinde Prim algoritmasının işleyişini anlatalım. Aşağıdaki graf örneğinde aşama aşama gerçekleşen işlemler;

Aşama
olan Aşama
Aşam
leştirilir. E
saplanır.
Aşam
sında ba
eklenir.
Aşan
olduğu i
ger düğ
Aşa
dan bu
Aşa
bu düğ
Aş
düğün
So
kapsay

Aşama 0: Örnek graf üzerinde rastgele bir düğüm seçilir.

Aşama 1: C düğümü rastgele seçilmiştir. C düğümün komşu olan düğümlere olan uzaklıklarları belirlenir.

Aşama 2: C düğümün en yakın olduğu E düğümü arasında bağlantı gerçekleştirilir. E'nin de komşu olduğu ziyaret edilmemiş düğümler ile uzaklıklarları hesaplanır.

Aşama 3: Diğer en küçük mesafe C-B arasında olduğu için bu düğümler arasında bağlantı sağlanır. B düğümün diğer düğümler ile olan mesafeleri listeye eklenir.

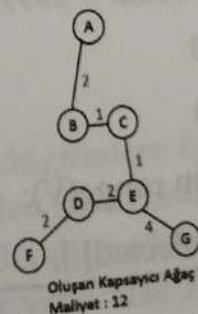
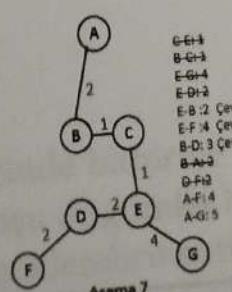
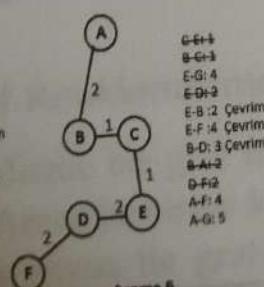
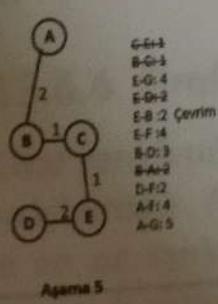
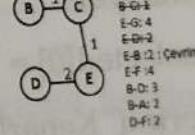
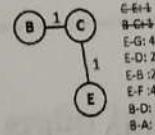
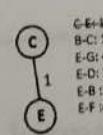
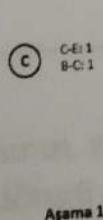
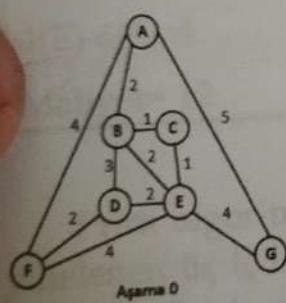
Aşama 4: Mesafe listesindeki en küçük diğer mesafe E-D düğümleri arasında olduğu için bu düğümler arasında bağlantı sağlanır. D'nin ziyaret edilmemiş diğer düğümlere olan mesafeler listeye eklenir.

Aşama 5: Listede diğer en küçük mesafe B-A düğümleri arasında olduğundan bu düğümler bağlanır.

Aşama 6: Listede diğer en küçük mesafe D-F düğümleri arasında olduğundan bu düğümler bağlanır. Mesafeler listesi güncellenir.

Aşama 7: Diğer en küçük ve çevrim oluşturmayacak mesafe E-G arasında olduğundan bu düğümler bağlanır.

Sonuç olarak maliyeti 12 olan daha önce Kruskal algoritması ile elde edilen kapsayıcı ağaç elde edilmiş olur.



Şimdi Python'da Prim algoritmasına ait kodları verelim. Yukarıdaki örnek graf için kodlar aşağıdadır. Grafa ait maliyet matrisi graf olarak verilmiş, bu matris üzerinden Prim algoritması ile minimum kapsayıcı ağaç bulunmuştur.

```
# Prim Algoritması
```

```
V = 7 #Düğüm Sayısı
```

```
#A,B,C,D,E,F,G
```

```
G = [[0,2,0,0,0,4,5],  
     [2,0,1,3,2,0,0],  
     [0,1,0,0,1,0,0],  
     [0,3,0,0,2,2,0],  
     [0,2,1,2,0,0,4],  
     [4,0,0,2,4,0,0],  
     [5,0,0,0,4,0,0]]
```

```
selected = [0, 0, 0, 0, 0, 0, 0]
```

```
# baslangıcta kenar sayısı 0
```

```
no_edge = 0
```

```
maliyet=0
```

```
selected[0] = True
```

```
print("Kenarlar : Ağırlık\n")
```

```
while (no_edge < V - 1):
```

```
    minimum = 99999
```

```
    x = 0
```

```
    y = 0
```

```
    for i in range(V):
```

```
        if selected[i]:
```

```

for j in range(V):
    if ((not selected[j]) and G[i][j]):
        * not in selected and there is an edge
        if minimum > G[i][j]:
            minimum = G[i][j]
            x = i
            y = j
maliyet=maliyet+G[x][y]
print(str(x) + "-" + str(y) + ":" + str(G[x][y]))
selected[y] = True
no_edge += 1
print("Maliyet=",maliyet)

```

Kenarlar : Ağırlık

0(A)-1(B):2

1(B)-2(C):1

2(C)-4(E):1

4(E)-3(D):2

3(D)-5(F):2

4(E)-6(G):4

Maliyet= 12

Prim algoritmasının zaman karmaşıklığı, grafin her bir düğümü üzerinde yinelenen üç iç içe döngü kullandığımız için $O(V^3)$ 'tür. Graf kenar değerlerini depolamak için ek belleğe ihtiyacımız olduğundan, bellek karmaşıklığı $O(V)$ 'dir. V düğüm sayısıdır.

13.6.6. Graf Renklendirme

Graf renklendirme bir graf üzerinde birbirine komşu düğümlere farklı renk atama işlemidir. Amaç birbirine komşu düğümler birbirinden farklı renkte olmak üzere en az renk sayısı ile grafi renklendirmektir. Renklendirmede kullanılan toplam renk sayısı kromatik sayı olarak ifade edilir. Şehir veya ülke haritalama

sistemlerinde kullanılmaktadır. Ancak farklı problemlerin çözümünde de kullanılan bir yaklaşımdır. Bu bölümde Welch ve Powel graf renklendirme algoritması anlatılacaktır. Yaklaşımı ait basamaklar aşağıdaki gibidir.

1. Graftaki tüm düğümlerin kenar sayıları bulunarak büyükten küçüğe sıralanır. Düğüm dereceleri bulunur.
2. Derecesi en büyük olan istenilen bir renk verilir. Bu düğüme komşu olmayan diğer düğümlere de (birbirlerine komşu olmayacak şekilde) aynı renk verilir.
3. Sonraki renklendirilmeyen düğüme geçilir.

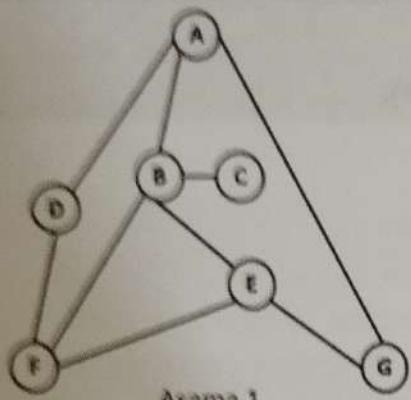
Yukarıdaki 2-3 aşamalar tüm düğümler renklendirilinceye kadar devam edilir. Aşağıda örnek bir graf üzerinde renklendirme işlemi gerçekleştirılmıştır. Şimdi bu aşamalarda gerçekleştirilen işlemleri basamak basamak anlatalım.

Aşama 1: Graftaki her düğümün derecesi bulunur. Derece bir düğüme bağlı kenar sayısıdır. Bu dereceler büyükten küçüğe sıralanır.

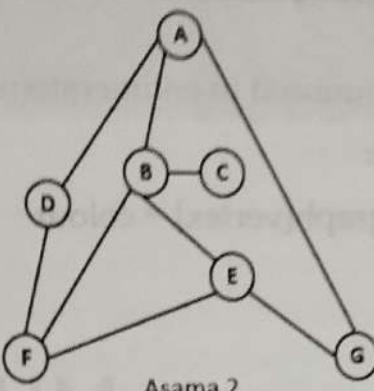
Aşama 2: Bu aşamada en yüksek dereceli B düğümü "kırmızı" olarak renklendirilmiştir. Daha sonraki komşu olmayan düğümlere bakılmıştır. A, F, E ve C düğümleri komşu olduğu için bunlar geçilmiştir. D ve G düğümleri hem B'ye komşu değil hem de kendi aralarında komşu olmadıklarından dolayı bu düğümler de "kırmızı" olarak renklendirilmiştir.

Aşama 3: Kalınan yerden devam edilerek A düğümü "Mavi" olarak renklendirilmiştir Aynı şekilde E ve C düğümler A düğümüne komşu olmadıkları gibi kendi aralarında da komşu olmadıklarından "Mavi" olarak renklendirilmiştir.

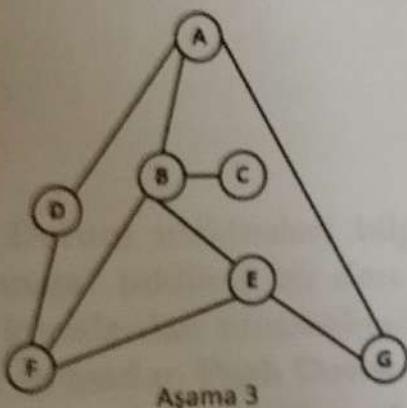
Aşama 4: Son olarak kalınan F düğümü "Sarı" olarak renklendirilmiştir.



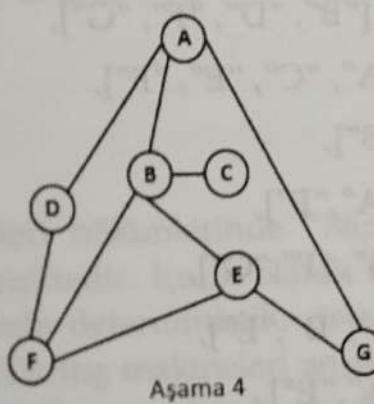
B:4
A:3
E:3
F:3
D:2
G:2
C:1



B:4 : Kırmızı
A:3
E:3
F:3
D:2 : Kırmızı
G:2 : Kırmızı
C:1



B:4 : Kırmızı
A:3 : Mavi
E:3 : Mavi
F:3 :
D:2 : Kırmızı
G:2 : Kırmızı
C:1 : Mavi



B:4 : Kırmızı
A:3 : Mavi
E:3 : Mavi
F:3 : Sarı
D:2 : Kırmızı
G:2 : Kırmızı
C:1 : Mavi

Şimdi Welch ve Powel graf renklendirme yaklaşımına ait Python kodunu ve- relim. Yıkardaki graf örnek alınmıştır.

```
def colour_vertices(graph):
    vertices = sorted((list(graph.keys())))
    colour_graph = {}

    for vertex in vertices:
        unused_colours = len(vertices) * [True]

        for neighbor in graph[vertex]:
            if neighbor in colour_graph:
                colour = colour_graph[neighbor]
                unused_colours[colour] = False

        if unused_colours:
            colour = unused_colours.index(True)
            colour_graph[vertex] = colour
```

```
for colour, unused in enumerate(unused_colours):  
    if unused:  
        colour_graph[vertex] = colour  
        break  
  
return colour_graph
```

```
graf = { "A" : ["B", "D", "F", "G"],  
        "B" : ["A", "C", "E", "F"],  
        "C" : ["B"],  
        "D" : ["A", "F"],  
        "E" : ["B", "F", "G"],  
        "F" : ["B", "D", "E"],  
        "G" : ["A", "E"],  
      }  
sonuclar = (colour_vertices(graf))
```

```
print(sonuclar)  
{'A': 0, 'B': 1, 'C': 0, 'D': 1, 'E': 0, 'F': 2, 'G': 1}
```

Yukarıdaki kod sonuçlarına bakıldığında A, C, E düğümlerin aynı renk ile renklendirildiği görülmektedir. Benzer şekilde B, D ve G düğümleri de aynı renk ile renklendirilmiştir. Son olarak F düğümü farklı renk ile renklendirilmiştir.