5.6 ikil arama ağaçları

→ anahtar-değer: liste, çırpı

→ bunlara haritalama yapıları denir

5.6.1 arama ağacı işlemleri

python sözlük yapısıyla özdeş

BinaryTree(): boş ikil ağaç

put(key, value): ağaca key:value çiftini ekle

get(key): ağaçtan key'e denk düşen değeri döndür. yoksa None

delete_key(key): key'i bul ve sil.

length(): kaç çift var

has_key(key): key var mı?

operators: [] ve in (has_key())

5.6.2 arama ağacı gerçeklemesi

→ buna **BST** (binary search tree) denir

→ özelliği ağacın sol tarafı kökten küçük,

→ sağ tarafı kökten büyük değerli

BST

- → gelen veriler: [70,31,93,94,14923,73]
- → $70 \Longrightarrow$ *köke*

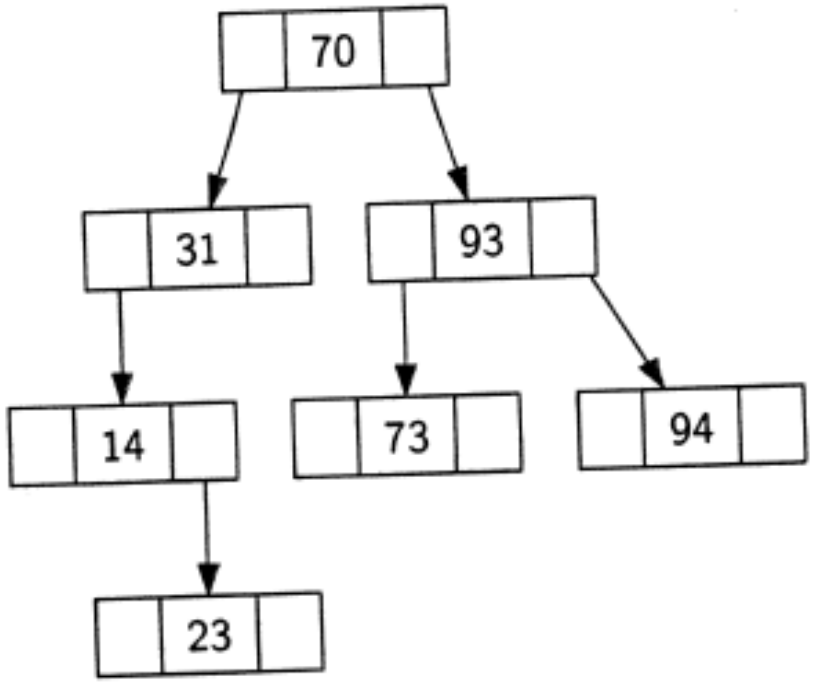
→ $31 < 70 \Longrightarrow$ *sola*

→ $93 > 70 \Longrightarrow$ *sağa*

→ $94 > 70$
$\Longrightarrow$ *sağa*; $94 > 93 \Longrightarrow$ *sağa*

→ ...

sonuç



gerçekleme

gerçekleme

```
1       class BinarySearchTree:
2           def __init__(self):
3               self.root = None
4               self.size = 0
5
6           def put(self,key,val):
7               if self.root:
8                   self.root.put(key,val)
9               else:
10                  self.root = TreeNode(key,val)
11              self.size = self.size + 1
12
13          def __setitem__(self,k,v):
14              self.put(k,v)
15
16          def get(self,key):
17              if self.root:
18                  return self.root.get(key)
19              else:
20                  return None
21
22          def __getitem__(self,key):
23              return self.get(key)
```

gerçekleme

gerçekleme

```
1           def has_key(self,key):
2               if self.root.get(key):
3                   return True
4               else:
5                   return False
6
7           def length(self):
8               return self.size
9
10          def __len__(self):
11              return self.size
12
13          def delete_key(self,key):
14              if self.size > 1:
15                  self.root.delete_key(key)
16                  self.size = self.size-1
17              elif self.root.key == key:
18                  self.root = None
19                  self.size = self.size - 1
20              else:
21                  print 'error, bad key'
```

TreeNode gerçekleme

gerçekleme

```
1       def __init__(self,key,val,parent=None,
2                                   left=None,right=None):
3           self.key = key
4           self.payload = val
5           self.leftChild = left
6           self.rightChild = right
7           self.parent = parent
```

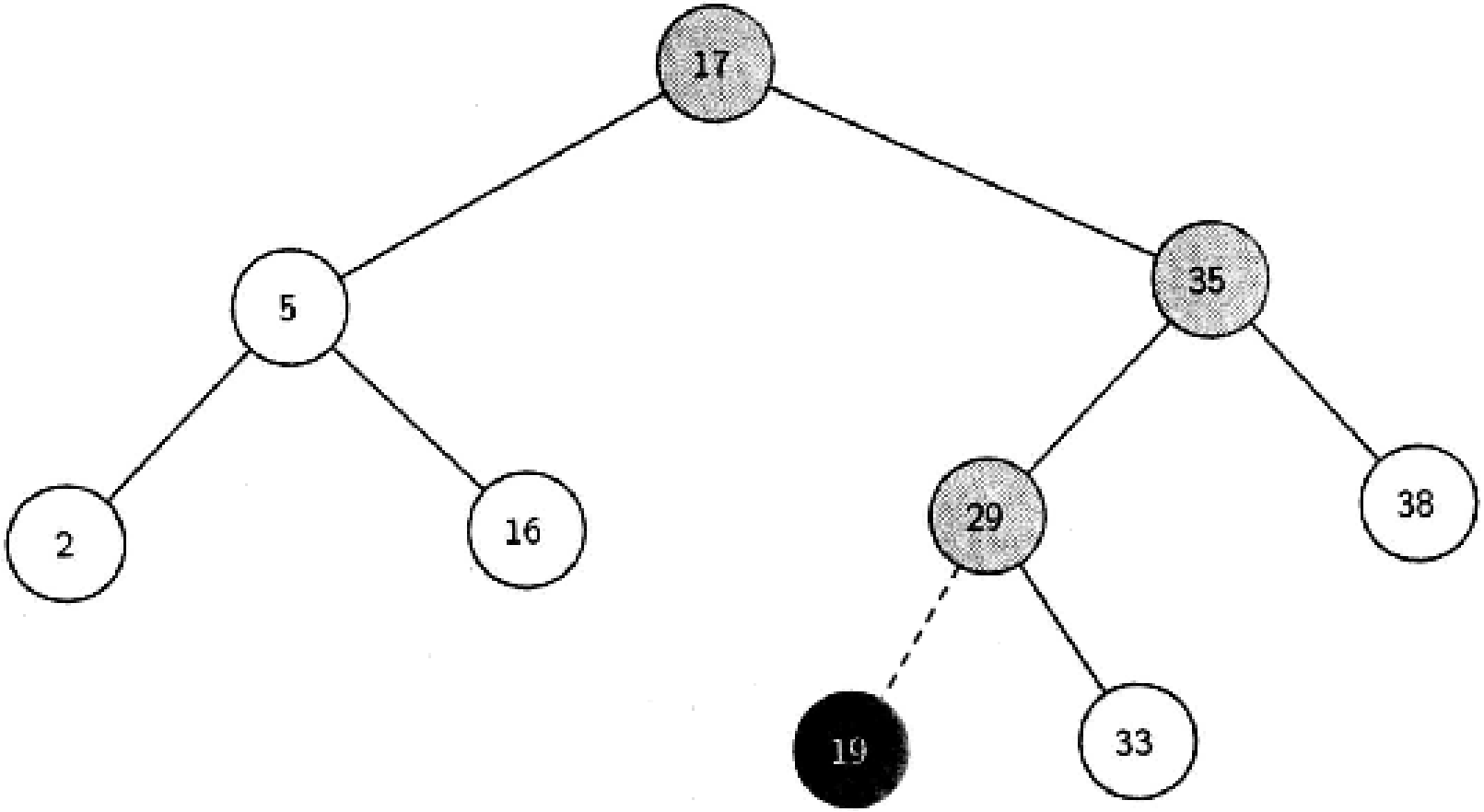
put: gerçekleme

gerçekleme

```
1       def put(self,key,val):
2           if key < self.key:
3               if self.leftChild:
4                   self.leftChild.put(key,val)
5               else:
6                   self.leftChild = TreeNode(key,val,self)
7           else:
8               if self.rightChild:
9                   self.rightChild.put(key,val)
10              else:
11                  self.rightChild = TreeNode(key,val,self)
```

yeni düğüm ekleme

→ anahtar değeri 19 olan yeni düğüm ekleme



→ gri olan ziyaret edilenler

ekstralar

→ [] işlecine overload için

```
1           def __setitem__(self,k,v):
2               self.put(k,v)
```

ekstralar

→ in işlecine overload için

```
1       def get(self,key):
2           if key == self.key:
3               return self.payload
4           elif key < self.key:
5               if self.leftChild:
6                   return self.leftChild.get(key)
7               else:
8                   return None
9           elif key > self.key:
10              if self.rightChild:
11                  return self.rightChild.get(key)
12              else:
13                  return None
14          else:
15              print 'error: this line should never be executed'
16
17      def __getitem__(self,key):
18          return self.get(key)
```

ekstralar

→ length işlevine overload için

```
1           def length(self):
2               return self.size
3
4           def __len__(self):
5               return self.size
```

özetçe

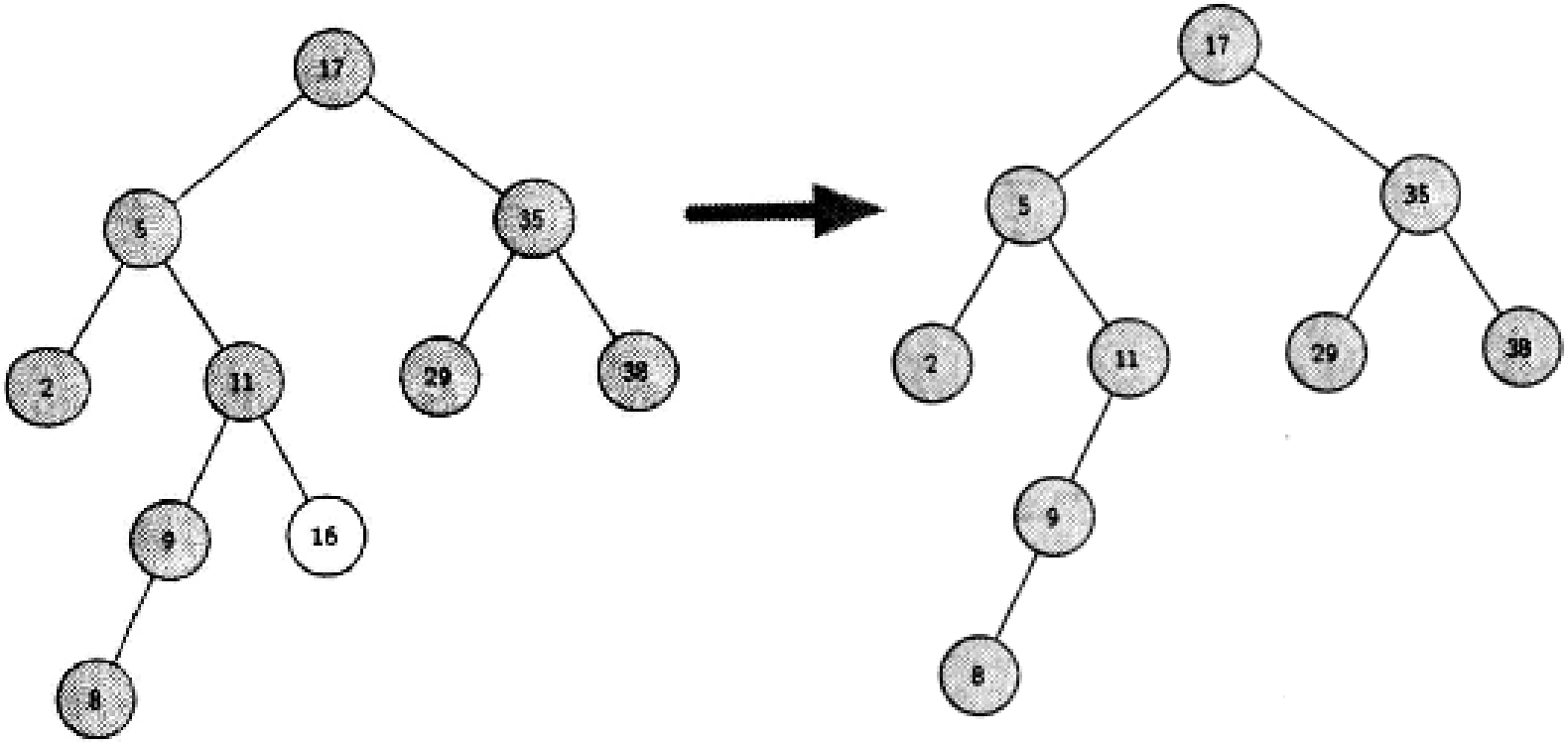
aşağıdaki kalıplar kullanılabilir

→ böylelikle: if/for in sozluk:

→ veya foo=Sozluk['+']

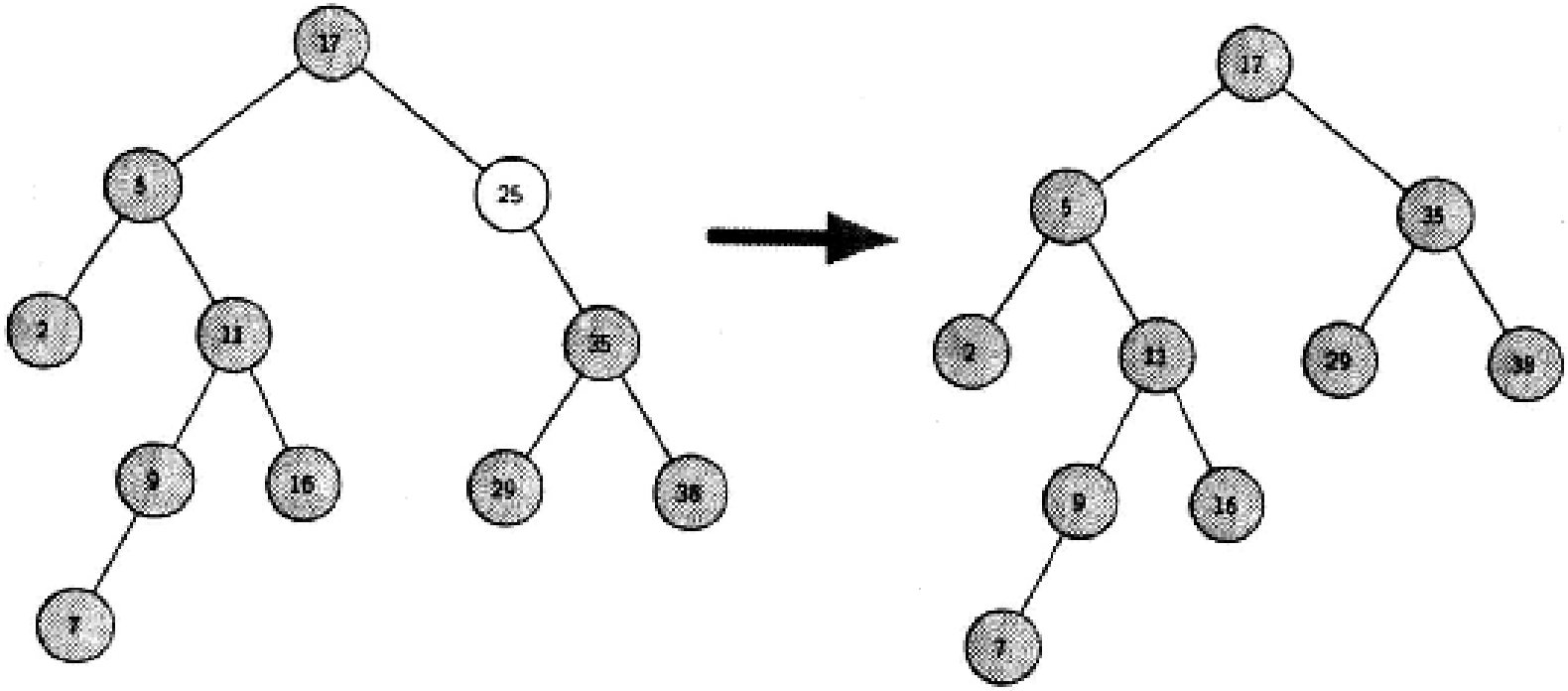
ağaçtan key i silme

→ çocuğu olmayan düğümün silinmesi. Ör. düğüm: 16



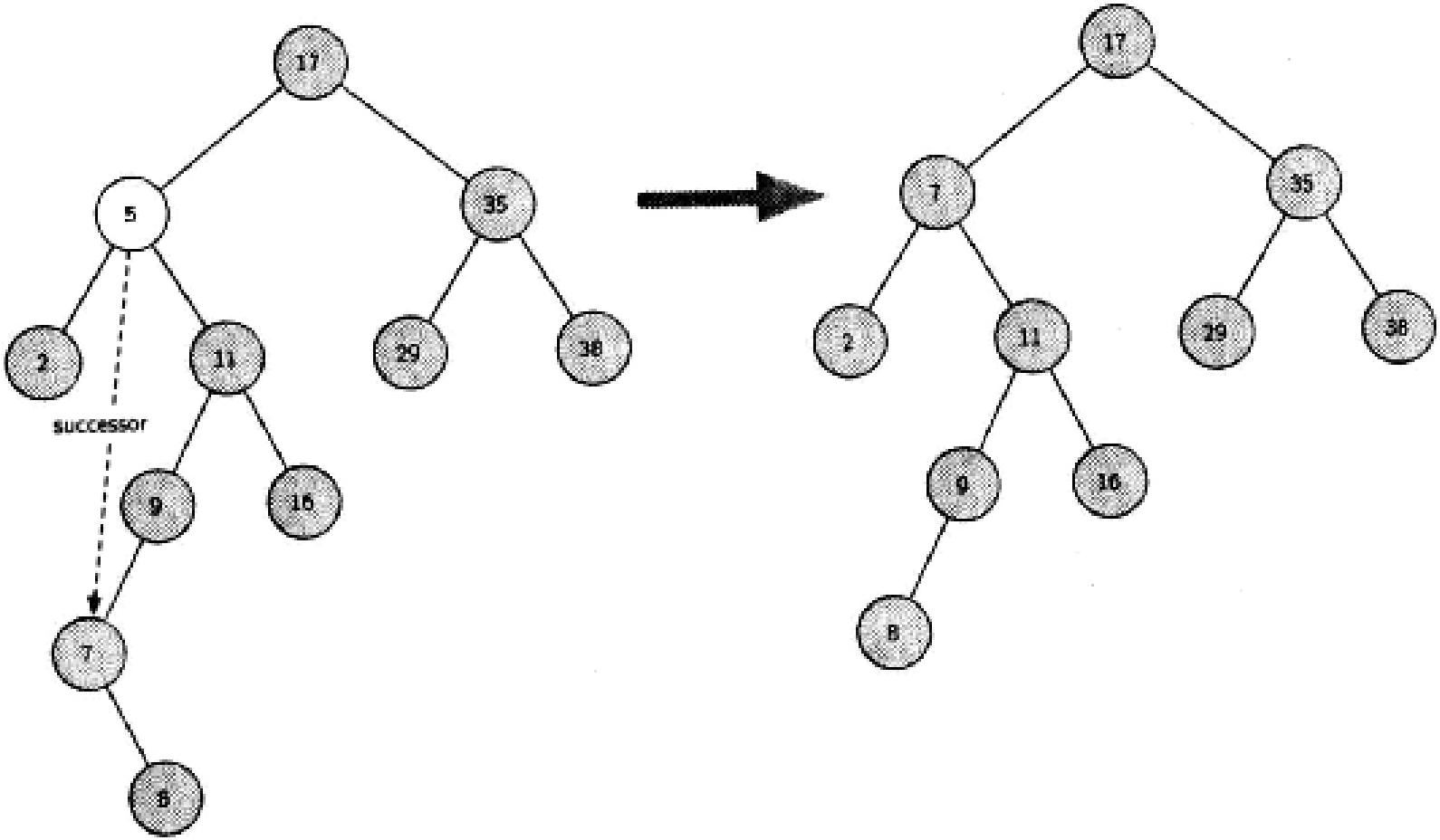
ağaçtan key i silme

→ tek çocukluysa. Ör. düğüm: 25



ağaçtan key i silme

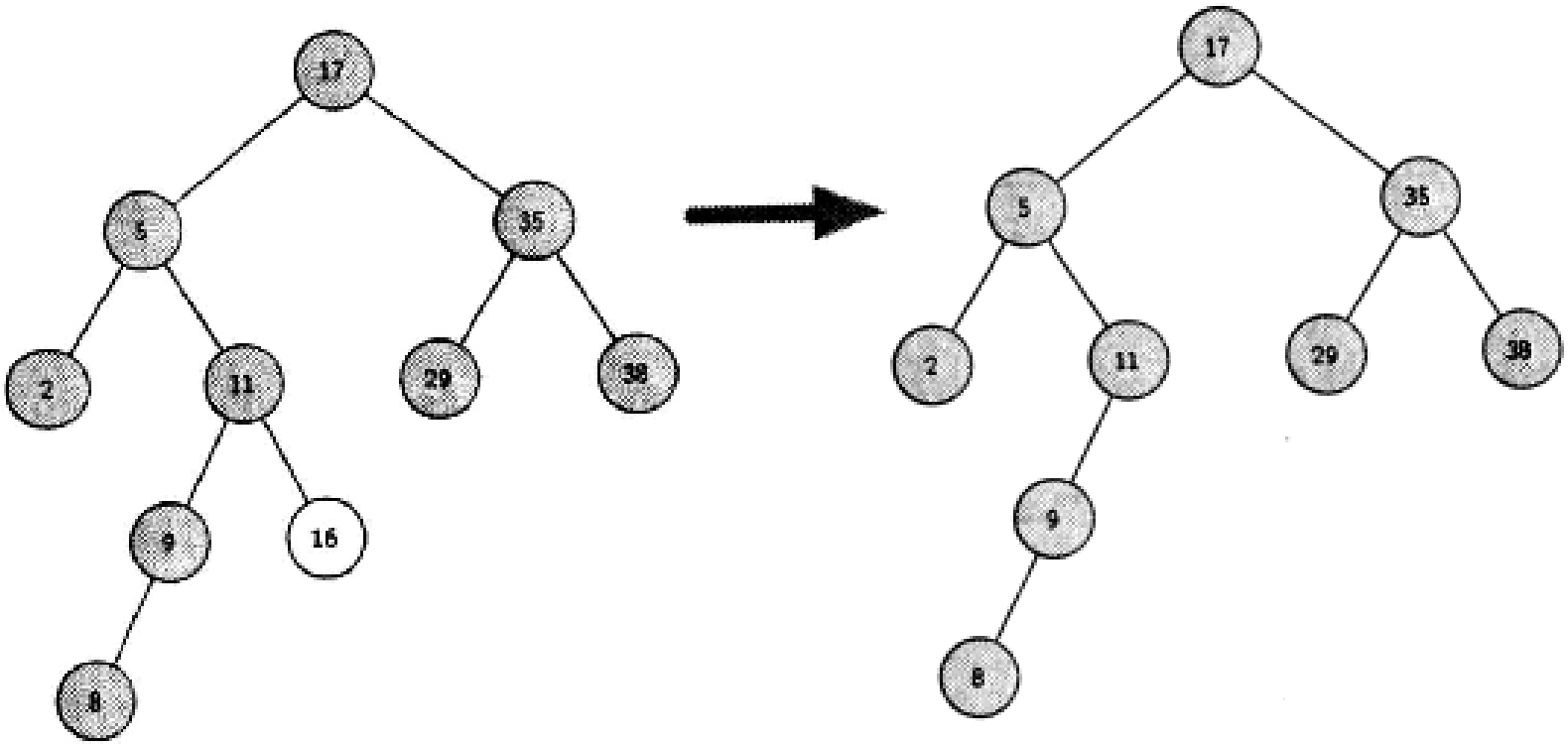
→ iki çocukluysa. Ör. düğüm: 5



silme: çocuksuz düğümü silme

→ durum 1: çocuksuz düğüm

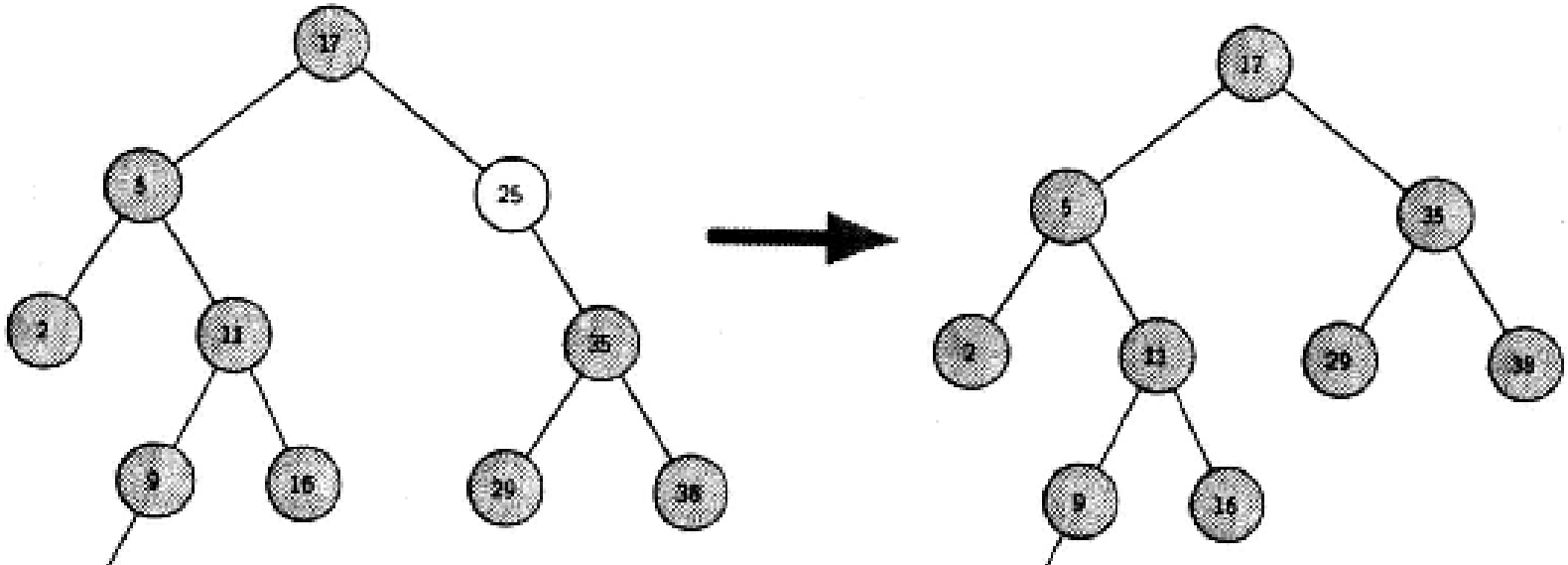
```
1       if not (self.leftChild or self.rightChild):
2           print "removing a node with no childrend"
3           if self == self.parent.leftChild:
4               self.parent.leftChild = None
5           else:
6               self.parent.rightChild = None
```



silme: tek çocuklu

→ durum 2: tek çocuklu düğüm

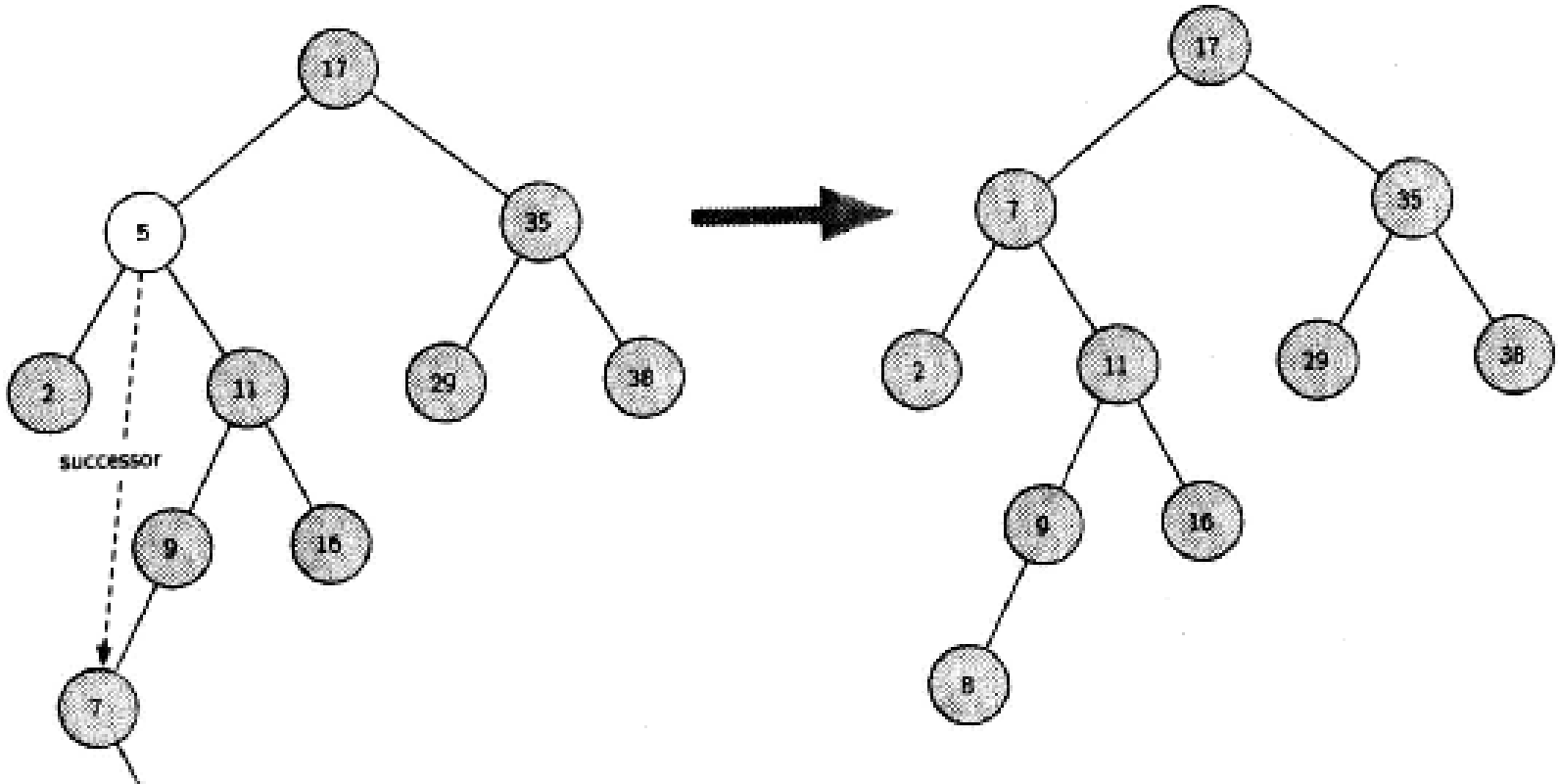
```
1       elif (self.leftChild or self.rightChild) \
2             and (not (self.leftChild and self.rightChild)):
3           print "removing a node with one child"
4           if self.leftChild:
5               if self == self.parent.leftChild:
6                   self.parent.leftChild = self.leftChild
7               else:
8                   self.parent.rightChild = self.leftChild
9           else:
10              if self == self.parent.leftChild:
11                  self.parent.leftChild = self.rightChild
12              else:
13                  self.parent.rightChild = self.rightChild
```



silme: iki çocuklu

→ durum 3: iki çocuklu düğüm

```
1       else:
2           succ = self.findSuccessor()
3           succ.spliceOut()
4           if self == self.parent.leftChild:
5               self.parent.leftChild = succ
6           else:
7               self.parent.rightChild = succ
8           succ.leftChild = self.leftChild
9           succ.rightChild = self.rightChild
```



silme: iki çocuklu: successor

→ BST özelliğini korumak için successor kullanılır

```
1           def findSuccessor(self):
2               succ = None
3               if self.rightChild:
4                   succ = self.rightChild.findMin()
5               else:
6                   if self.parent.leftChild == self:
7                       succ = self.parent
8                   else:
9                       self.parent.rightChild = None
10                      succ = self.parent.findSuccessor()
11                      self.parent.rightChild = self
12              return succ
13
14          def findMin(self):
15              n = self
16              while n.leftChild:
17                  n = n.leftChild
18              print 'found min, key = ', n.key
19              return n
```

successor

$\longrightarrow$ defterdenm çizim (s17)

spliceOut

→ spliceOut

```
1       def spliceOut(self):
2           if (not self.leftChild and not self.rightChild):
3               if self == self.parent.leftChild:
4                   self.parent.leftChild = None
5               else:
6                   self.parent.rightchild = None
7           elif (self.leftChild or self.rightChild):
8               if self.leftChild:
9                   if self == self.parent.leftChild:
10                      self.parent.leftChild = self.leftChild
11                  else:
12                      self.parent.rightChild = self.leftChild
13              else:
14                  if self == self.parent.leftChild:
15                      self.parent.leftChild = self.rightChild
16                  else:
17                      self.parent.rightChild = self.rightChild
```

spliceOut

$\longrightarrow$ defterden çizim (s18)

delete_key

→ parçaları bir araya getir

```
1       def delete_key(self,key):
2           if self.key == key:  # do the removal
3               if not (self.leftChild or self.rightChild):
4                   if self == self.parent.leftChild:
5                       self.parent.leftChild = None
6                   else:
7                       self.parent.rightChild = None
8               elif (self.leftChild or self.rightChild) and \
9                        (not (self.leftChild and self.rightChild)):
10                  if self.leftChild:
11                      if self == self.parent.leftChild:
12                          self.parent.leftChild = self.leftChild
13                      else:
14                          self.parent.rightChild = self.leftChild
15                  else:
16                      if self == self.parent.leftChild:
17                          self.parent.leftChild = self.rightChild
18                      else:
19                          self.parent.rightChild = self.rightChild
20              else:  # replace self with successor
21                  succ = self.findSuccessor()
22                  succ.spliceOut()
23                  if self == self.parent.leftChild:
24                      self.parent.leftChild = succ
```

__iter__

- → __iter__: ağacı inorder ile ziyaret eder
- → kalıp: for i in range(10):
- → inorder: sol-kök-sağ

```
        def __iter__(self):
            if self:
                if self.leftChild:
                    for elem in self.leftChild:
                        yield elem
                yield self.key
                if self.rightChild:
                    for elem in self.rightChild:
                        yield elem
```

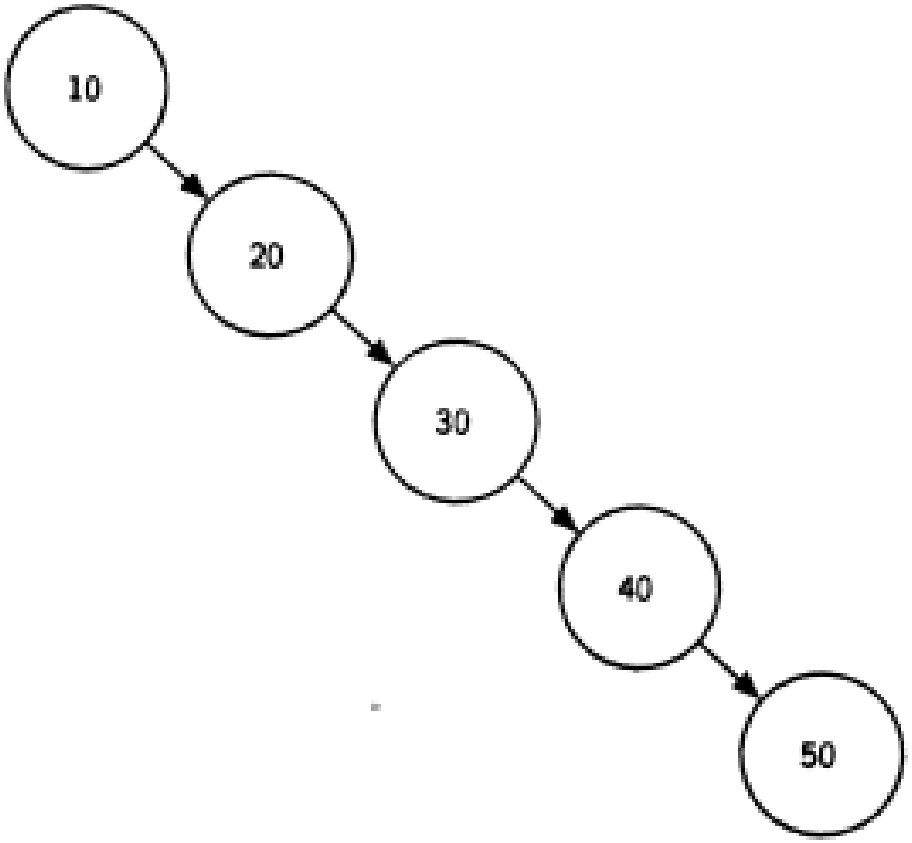
- → defterden çizim (s19)
- → sıra sizde: demo

analiz

- → **put** yöntemi: performansı kilitleyen faktör, ikil ağacuın yüksekliği
- → yükseklik ise, kökle en derin yaprak arasındaki dal sayısı
- → yükseklikle sınırlı çünkü put etmek için uygun yeri buluncaya değin aramaya ihtiyaç duyuluyor
- → ikil ağaç hangi yükseklikte olabilir
- → anahtarı ağaca nasıl ekliyoruz?
- → anahtarlar rastgele sırada eklenirse yükseklik: **log_2 n** (n, düğüm sayısı)
- → bu durumda kabaca sağ ve sol altağaçlar aynı sayıda düğüm içerir
- → böylesi ağaca **dengeli** ağaç deriz
- → dengeli ağaçta **put** yönteminin karmaşıklığı **O(log n)**

analiz

→ en kötü durum



→ bu durumda karmaşıklık: **O(n)**

→ ağacın dengesizliği sorundur

→ bunun önüne geçmek için: AVL, kırmızı-siyah ağaçlar kullanılır

→ diğer işlevler (get, has_key, delete_key) de benzer şekilde yükseklikle sınırlanır