Yığın Sıralaması

→ Heap Sort, sıralama algoritmalarından biridir ve temel olarak bir heap veri yapısı kullanarak çalışır. Heap, özellikle öncelik sırasına göre elemanlara erişim sağlamak için kullanılan bir ağaç yapısıdır. Heap Sort; Min Heap ve Maks Heap olarak ikiye ayrılır. Mantıkları aynıdır sadece Min Heap küçükten büyüğe sıralarken Maks Heap büyükten küçüğe sıralar.

1. İlk Adım: Verilen diziyi bir maksimum heap yapısı haline getiriyoruz. Maksimum heap, her düğümün kendisinden daha küçük alt düğümlere sahip olduğu bir yapıyı ifade eder. Bu adım, diziyi heap yapısına dönüştürmek için "heapify" adı verilen bir işlemi içerir.

2- İkinci Adım: Heap yapısını oluşturduktan sonra, dizinin en büyük elemanı (kök) en üstte olur. Bu elemanı dizinin son elemanıyla yer değiştiririz, böylece en büyük eleman dizinin sonunda yer alır. Daha sonra, son elemanı heap yapısından çıkararak düzeni bozmamak için tekrar heapify işlemi yaparız.

3- Son Adım: Bu adımları son eleman heap yapısından çıkarılana kadar tekrar ederiz. Her adımda en büyük elemanı dizinin sonundaki elemanla yer değiştiririz ve heap yapısını sürdürmek için heapify işlemi uygularız.

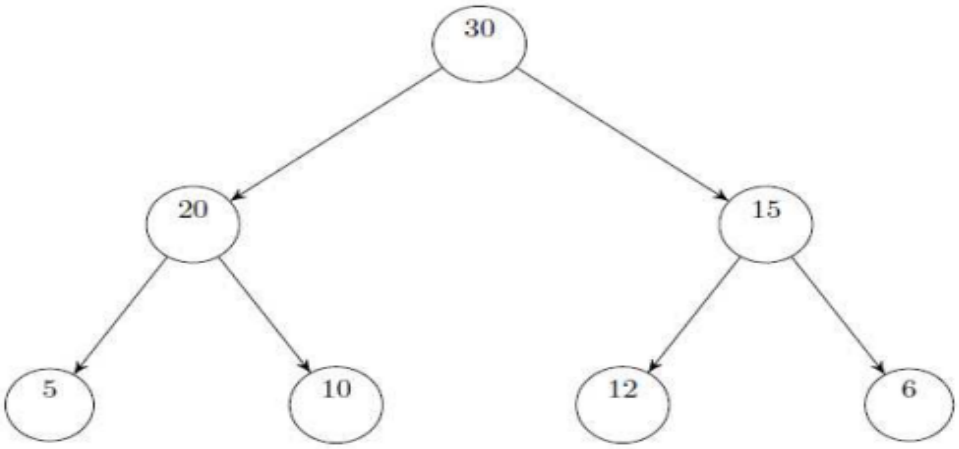
Zaman Karmaşıklığı: Heap Sort' un ortalama ve en kötü durum zaman karmaşıklığı O(n log n) olarak kabul edilir. Heapify işlemi ve dizideki elemanların yer değiştirmeleri bu karmaşıklığı etkiler.

Not: Heap Sort'un avantajlarından biri, kararlı olmasıdır, yani aynı değere sahip elemanların sırasının değişmemesi anlamına gelir. Ancak, ekstra bellek kullanımı gerektiren ve diğer sıralama algoritmalarına (örneğin Quick Sort veya Merge Sort) kıyasla daha yavaş olabilen bir algoritmadır.

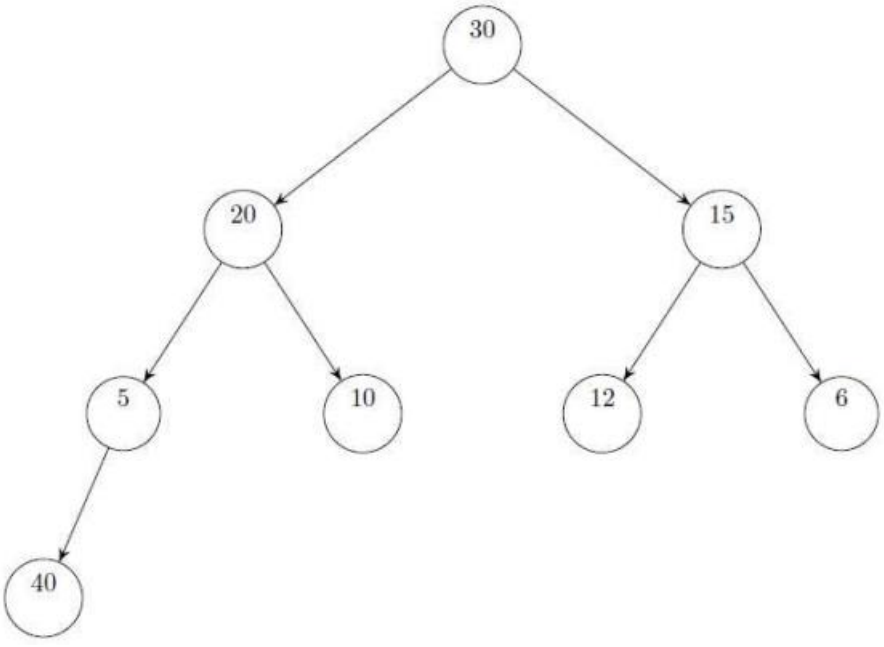
Örnek 1: Eleman Ekleme

→ Eleman ekleme işleminden sonra ağacımız kurala uygun şekilde değil ise tekrardan kurala uygun olacak şekilde yer değişimleri yapıyoruz.

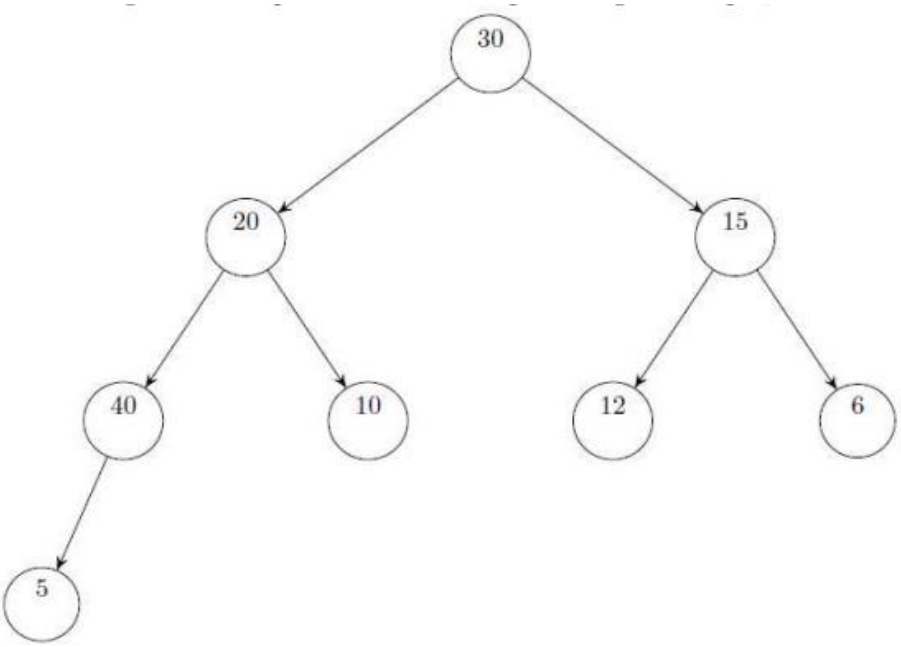
Dizi: [30, 20, 15, 5, 10, 12, 6]



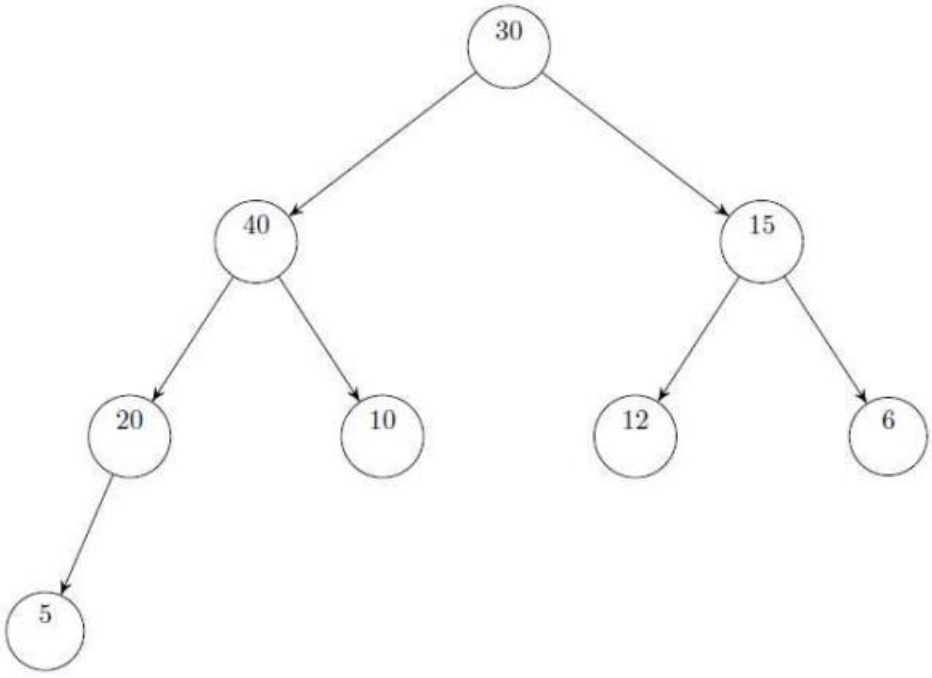
Dizi: [30, 20, 15, 5, 10, 12, 6, 40]



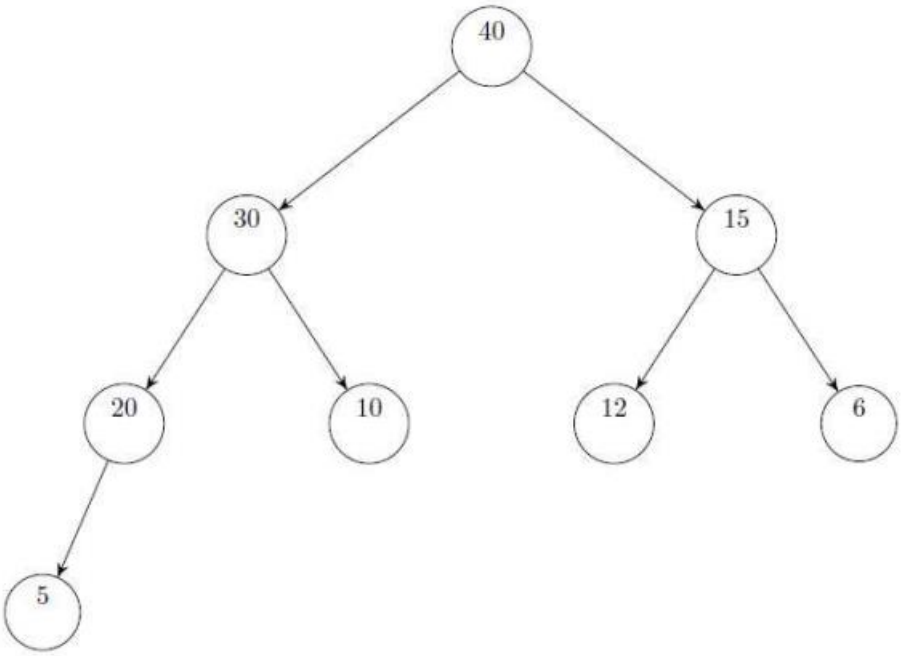
Dizi: [30, 20, 15, 40, 10, 12, 6, 5]



Dizi: [30, 40, 15, 20, 10, 12, 6, 5]



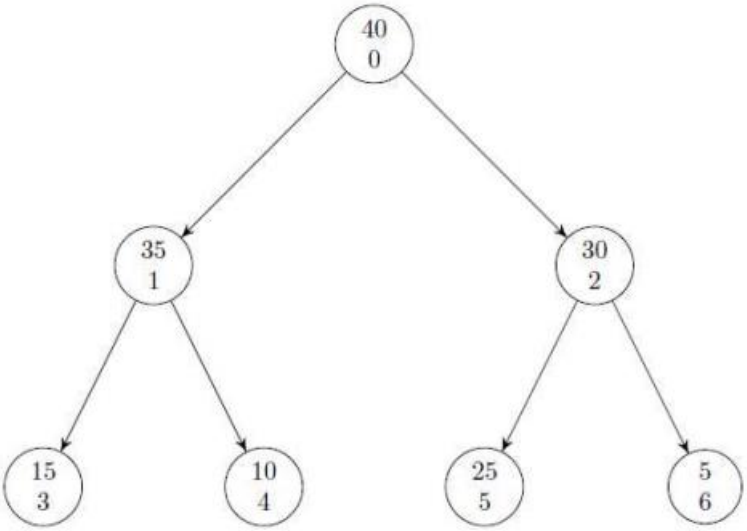
Dizi: [40, 30, 15, 20, 10, 12, 6, 5]



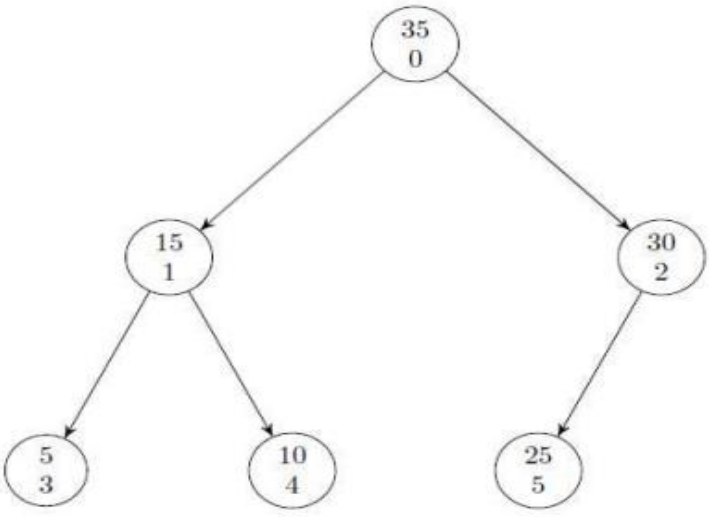
Örnek 2: Eleman Silme

→ Tepedeki elemanı silerek başka bir diziye atadıktan sonra en sağ alt taraftaki değeri sildiğimiz elemanın yerine koyarız(yani kök kısmına) sonrasında ise eğer ağacımız kurala uygun şekilde değil ise yer değişimleri yaparak kurala uygun hale getirmemiz gerekmektedir.

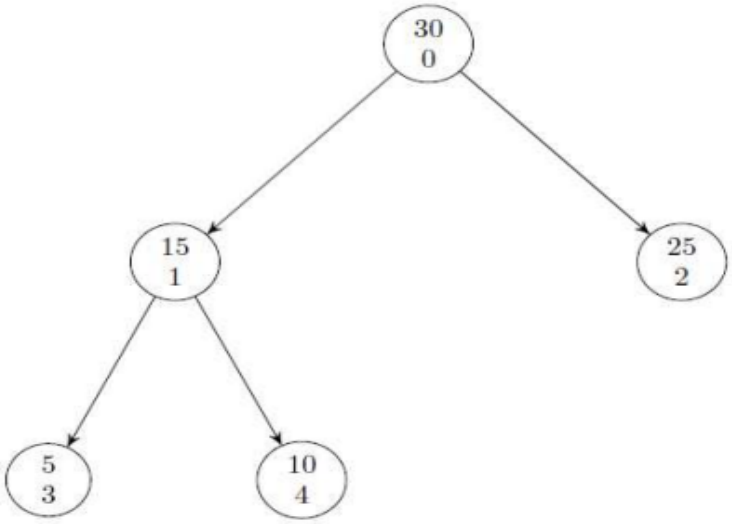
Dizi: [40, 35, 30, 15, 10, 25, 5]



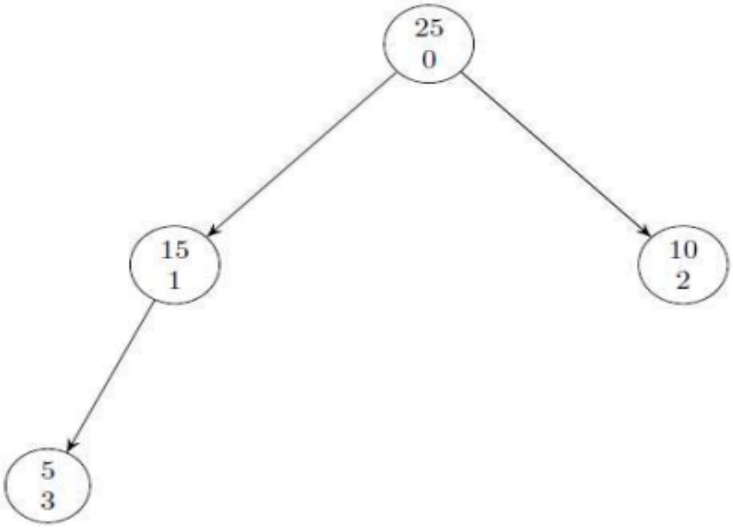
Yeni Dizi: [40]



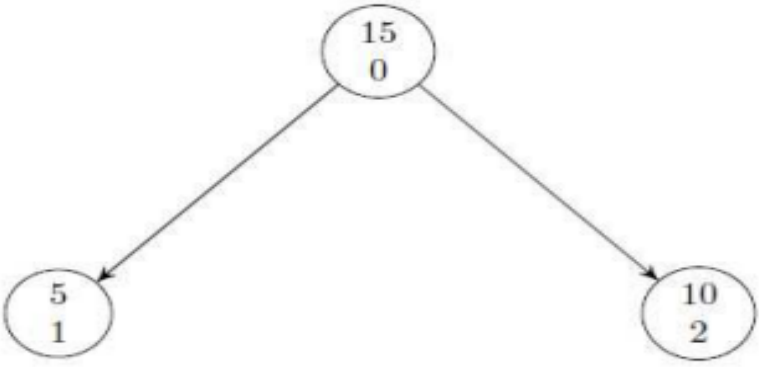
Yeni Dizi: [40, 35]



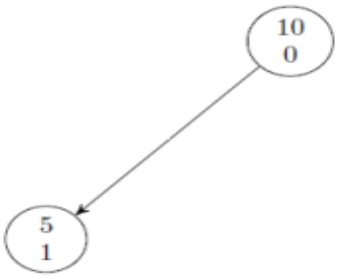
Yeni Dizi: [40, 35, 30]



Yeni Dizi: [40, 35, 30, 25]



Yeni Dizi: [40, 35, 30, 25, 15]



Yeni Dizi: [40, 35, 30, 25, 15, 10]

Yeni Dizi: [40, 35, 30, 25, 15, 10, 5]

Pseudocode Kod:

```
Max_Heapify(A, i):
    left = 2 * i + 1
    right = 2 * i + 2
    largest = i
    if left < heap_size(A) and A[left] > A[largest]:
        largest = left

    if right < heap_size(A) and A[right] > A[largest]:
        largest = right

    if largest != i:
        swap(A[i], A[largest])
        Max_Heapify(A, largest)

Build_Max_Heap(A):
    for i from floor(heap_size(A) / 2) down to 0:
        Max_Heapify(A, i)

Max_Heap_Sort(A):
    Build_Max_Heap(A)
    for i from heap_size(A) - 1 down to 1:
        swap(A[0], A[i])
        heap_size(A) = heap_size(A) - 1
        Max_Heapify(A, 0)
```

Python Kod:

```
def max_heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[left] > arr[largest]:
        largest = left

    if right < n and arr[right] > arr[largest]:
        largest = right

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        max_heapify(arr, n, largest)

def build_max_heap(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        max_heapify(arr, n, i)

def max_heap_sort(arr):
    n = len(arr)
    build_max_heap(arr)
```

```
    for i in range(n - 1, 0, -1):
        arr[0], arr[i] = arr[i], arr[0]
        max_heapify(arr, i, 0)

# Örnek kullanım
arr = [4, 10, 3, 5, 1]
max_heap_sort(arr)
print("Sıralanmış dizi:", arr)
```