Algoritma Analizi

Ders 6: Sıralama Algoritmaları

Doç. Dr. Mehmet Dinçer Erbaş
Bolu Abant İzzet Baysal Üniversitesi
Mühendislik Fakültesi
Bilgisayar Mühendisliği Bölümü

Sıralama algoritmaları

- Bu bölümde bazı farklı sıralama algoritmalarını inceleyeceğiz.
- Sıralama algoritmaları, algoritma konusunda araştırmalarda oldukça önemli bir yer tutar. Çünkü:
  - Birçok algoritmanın içerisinde sıralama işlemi bulunur.
    - Örneğin bir banka için hesap dökümü çıkaran bir program yazacaksanız yapılan işlemlerin sıralanması gerekir.
  - Sıralama işlemi farklı algoritmaların kullanılmasından önce yapılabilir.
    - Örneğin bir oyuna ait grafik nesneleri tanımlarken, bu nesneleri birbirlerinin üzerinde olma durumuna göre sıralamamız gerekebilir.
  - Birçok farklı teknik kullanılarak oluşturulmuş sıralama algoritması vardır.
    - Bu teknikleri öğrenmek genel algoritma oluşturma konusunda yardımcı olur

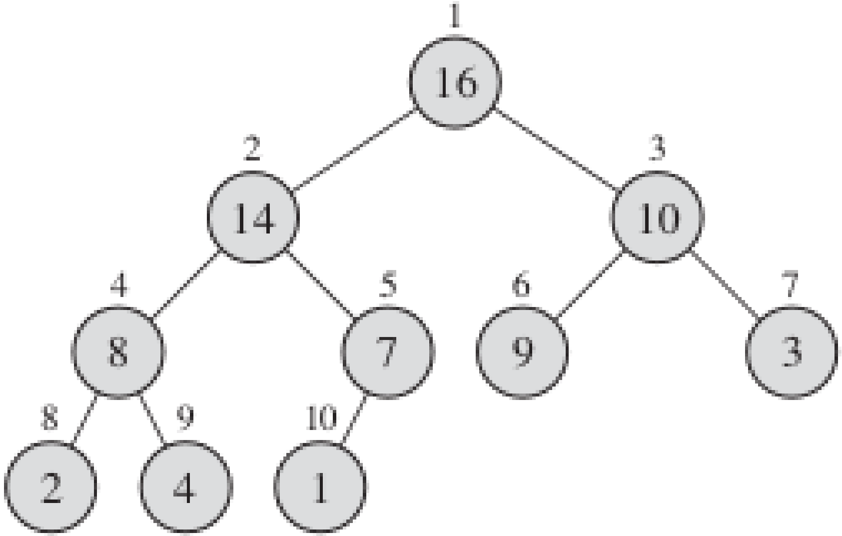
Yığın sıralama

- Bu kısımda Yığın sıralama (Heapsort) algoritmasını örneceğiz.
- Yapacağımız analizde göreceğimiz üzere Yığın sıralama algoritması O(n lg n) sürede çalışır.
  - Bu açıdan Yığın sıralama birleştirmeli sıralama algoritmasına benzer.
- Yığın sıralama, yerinde sıralama yapar, yani sıralanacak elemanlar aynı dizi içerisinde hareket ettirilir.
  - Bu açıdan Yığın sıralama eklemeli sıralama algoritmasına benzer.
- Bu iki durumu göz önünde bulundurduğumuzda Yığın sıralama algoritması için merge sort ve insertion sort algoritmalarının iyi yanlarını almıştır diyebiliriz.
- Yığın sıralama ayrıca heap (yığın) isminde bir veri yapısı kullanır.
  - Tabiki bu Java veya Lisp dillerinde tanımlanan heap terimi ile karıştırılmamalıdır.

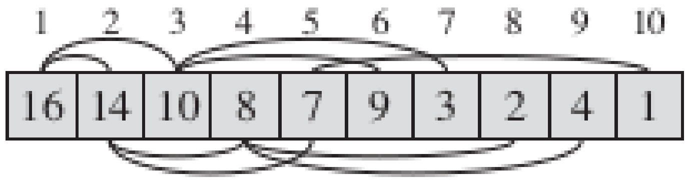
Yığın sıralama

- Bir binary (ikili) veri yapısı olan heap, neredeyse tamamlanmış bir ikili ağaçtan oluşan bir nesne dizisi olarak tanımlanabilir.
- Ağaç üzerindeki her bir düğüm (ing: node), diziye ait bir elemanı simgeler.
- Ağacın son seviyesi dışındaki tüm seviyeleri tam olarak dolmuştur.
  - Son seviye ise soldan sağa doğru doldurulur.
- Heap, bir A dizisi tarafından iki değişken ile temsil edilir:
  - 1. length[A]: dizi içerisindeki eleman sayısı.
  - 2. heap-size[A]: dizi içerisinde heap‘e ait eleman sayısı.
    - Bu durumda dizi içerisinde length[A] kadar eleman olsa bile heap-size[A] sonrası elemanlar heap‘e ait değildir.

Yığın sıralama



(a)



(b)

Yığın sıralama

- Heap ağacı şu şekilde oluşturulur.
  - Ağacın kök düğümü A[1]‘dir.
  - Belirtilen i indeksli düğümün üst, sol alt ve sağ alt düğümüne indeksleri aşağıdaki şekilde bulunabilir.
    - PARENT(i)
      - return $\lfloor i/2 \rfloor$
    - LEFT(i)
      - return 2i
    - RIGHT(i)
      - return 2i + 1
  - Bu işlemler birçok bilgisayarda bir veya iki komut ile yapılabilir.
  - Yığın sıralama uygulamalarında bu işlemler macro veya in-line işlemlerle yapılır.

Yığın sıralama

- İki farklı heap tipi vardır. Heap tipine uygun olarak düğümler heap özelliğini taşırlar.
- Max-heap için heap özelliği şu şekilde tanımlanmıştır:
  - A[Parent(i)] ≥ A[i]
  - Yani bir düğümün değeri en fazla üst düğümünün değeri kadardır.
  - Bu durumda kök düğümün değeri ağaçtaki en yüksek değer olur.
  - Aynı şekilde herhangi bir altağacın en yüksek değeri bu altağacın kök düğümün değeridir.
- Min-heap için benzer şekilde heap özelliği şu şekilde tanımlanmıştır:
  - A[Parent(i)] ≤ A[i]
  - Bu durumda kök düğümün değeri ağaçtaki en düşük değerdir.
- Biz bu bölümde max-heap yapısını inceleyeceğiz.

Yığın sıralama

- Heap'in ağacını incelediğimizde herhangi bir düğümün yüksekliğini bu düğümden ağacın bir yaprağına doğru aşağı basit bir yol ile hareket ettiğimizde uğradığımız en fazla kenar sayısı olarak tanımlıyoruz.
- Ağacın yüksekliği ise ağacın kök düğümünün yüksekliğidir.
- Soru: h yüksekliğine sahip bir max-heap için minimum ve maksimum eleman sayısını bulalım.
  - Tahtada çözüm yapılacak.
- Tespit: n elemanlı bir max-heap için yükseklik $\lfloor lg\, n \rfloor$ olur.
  - Tahtada çözüm yapılacak.

Yığın sıralama

- n elemanlı bir heap ağacının yüksekliği Θ(lg n) olur.
- Birazdan göreceğimiz üzere heap ağacı üzerinde yapılan işlemler ağacın yüksekliği ile orantılı sürede yapılmaktadır. Bu nedenle bu işlemler O(lg n) süre alır.
- Yığın sıralama ile sıralama işlemi yapacağımızda aşağıdaki fonksiyonları kullanacağız:
  - MAX_HEAPIFY fonksiyonu O(lg n) süre alır ve max-heap özelliğinin korunmasını sağlar.
  - BUILD_MAX_HEAP fonksiyonu O(n) süre alır ve sıralanmamış bir eleman dizisinden max-heap oluşturur.
  - Yığın sıralama fonksiyonu O(n lg n) süre alır ve bir diziyi yerinde sıralar.

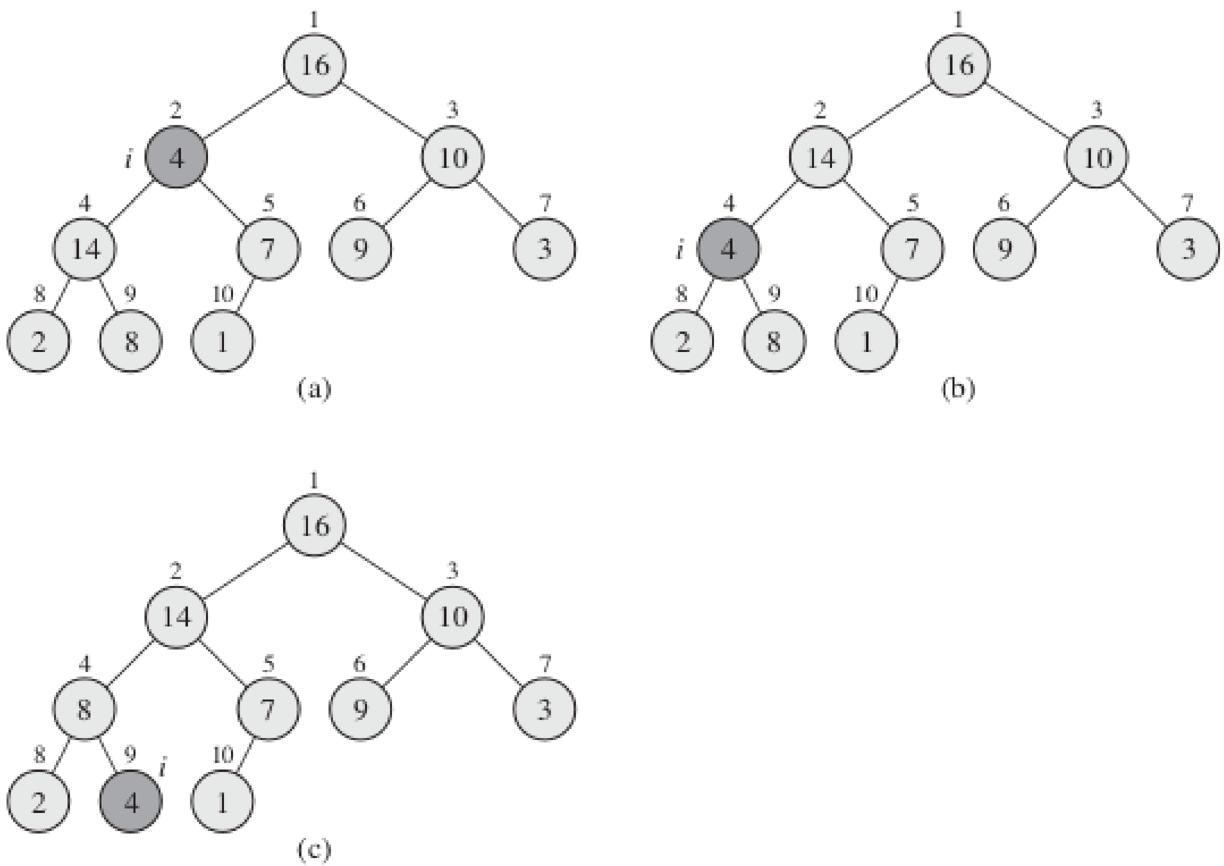
Yığın sıralama

- MAX_HEAPIFY fonksiyonu heap yapımıza yeni bir eleman eklendiğinde max-heap özelliğinin sağlanması için kullanılır.
- Girdi olarak bir A dizisi ve bu dizideki bir elemanın i indeks sayısını alır.
- Bu fonksiyon çağırıldığında LEFT(i) ve RIGHT(i) alt ağaçlarının max-heap özelliğine sahip olduğu kabul edilir.
  - Ancak A[i] elemanı alt düğümlerden küçük değere sahip olabilir. Bu durumda max-heap özelliği bozulacaktır.
- MAX_HEAPIFY fonksiyonu kullanılarak A[i] elemanının ağaçtaki doğru yere geçmesi sağlanır.

Yığın sıralama

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

Yığın sıralama



Yığın sıralama

- MAX_HEAPIFY fonksiyonun toplam çalışma süresi şu şekilde hesaplanır:
  - A[i], A[LEFT[i]] ve A[RIGHT[i]] arasındaki ilişkiyi düzeltmek $\Theta(1)$ zaman alır.
  - Alt ağaçtaki max-heap özelliğini düzeltmek için alt ağaçta MAX_HEAPIFY fonksiyonunun çalışma süresi kadar zaman alır.
- Alt ağacın büyüklüğü en fazla ağacın 2n/3 kadarıdır.
  - Bu durum en alt seviyenin tam olarak yarısının dolu olması durumunda görülür.
- Öyleyse MAX_HEAPIFY çalışma süresini şu şekilde tanımlayabiliriz:
  - $T(n) \leq T(2n/3) + \Theta(1)$
- Master metoduna göre bu durum 2'ye uygun olur.
  - $T(n) = O(\lg n)$
  - h yüksekliğinde bir düğüm için MAX_HEAPIFY $\Theta(h)$ sürede çalışır.

Yığın sıralama

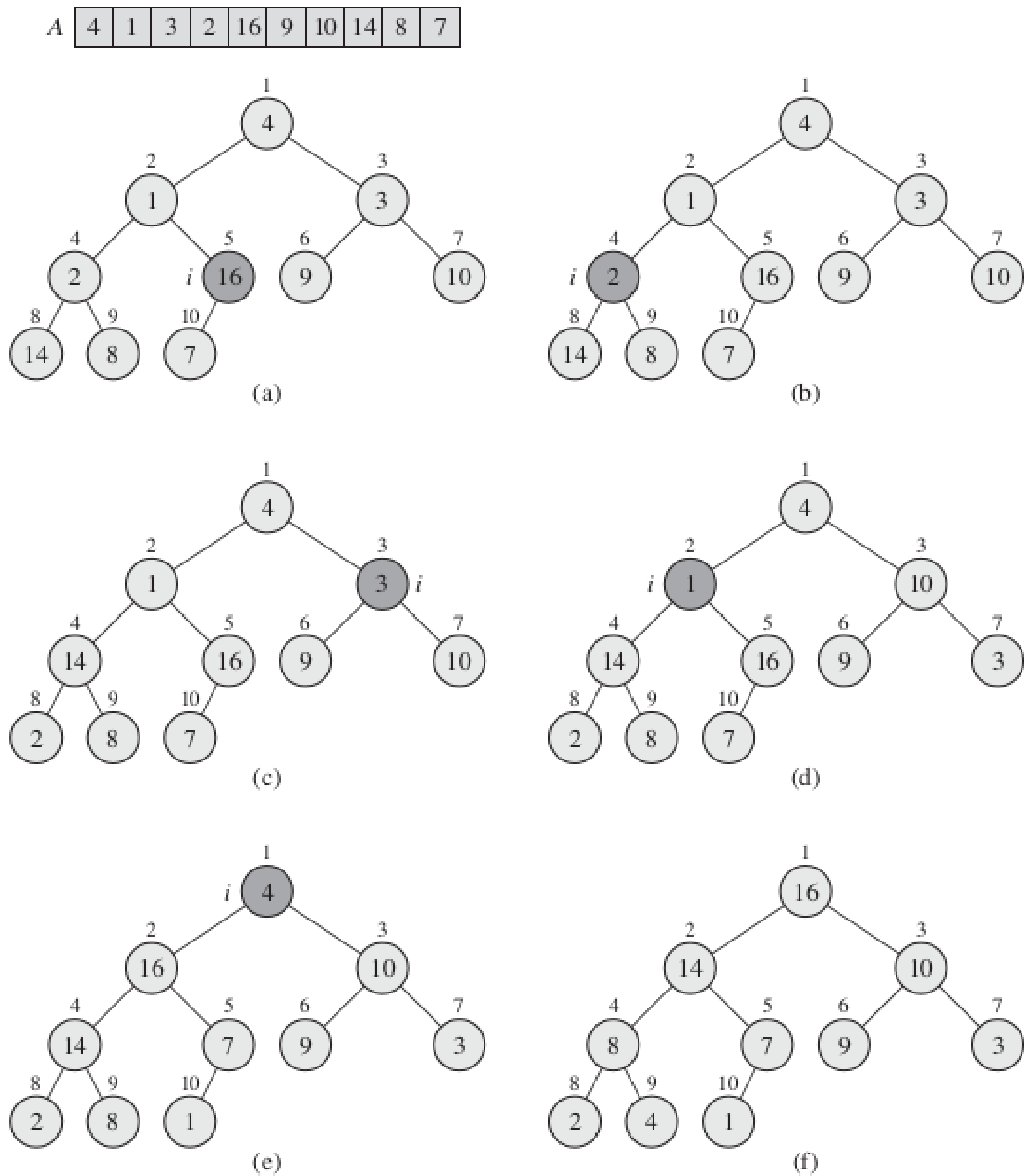
- MAX_HEAPIFY fonksiyonu kullanarak, aşağıdan yukarı şekilde verilmiş olan A[1..n] dizisinden, n = length[A], bir max-heap oluşturabiliriz.
- Soru: n elemanlı bir heap için, bu heap dizi olarak gösterildiğinde yaprak olan düğümler $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, .., n$ düğümlerdir. Bu durumun sebebi nedir?
  - Heap tanımımızda bir düğümün indeksi i ise, sol alt düğüm indeksi 2i, sağ alt düğüm indeksi ise 2i+1 olmalıdır. $\lfloor n/2 \rfloor$ Indeksli düğümden sonraki indekse sahip düğümlerin sol ve sağ yaprakları olamaz çünkü bu yaprakların indeksleri dizinin dışına çıkacaktır.
  - $A[\lfloor n/2 \rfloor + 1..n]$ Altdizisine ait bütün elemanlar ağacın yapraklarıdır.
    - Bu yaprakların her biri 1-elemanlı max-heap ağaçlarıdır.
  - Öyleyse geri kalan elemanlar için MAX_HEAPIFY fonksiyonunu çağırırsak max-heap oluşturabiliriz.

Yığın sıralama

```
BUILD-MAX-HEAP(A)
    A.heap-size = A.length
    for i = ⌊A.length/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```

Soru: Neden algoritmada 1‘den $\lfloor A.length/2 \rfloor$‘ye hareket etmek yerine $\lfloor A.length/2 \rfloor$ ‘den 1‘e hareket ediyoruz?
MAX_HEAPIFY fonksiyonu sağ ve sol alt ağaçlarının max-heap olduğunu kabul ederek çalışıyor. $\lfloor A.length/2 \rfloor$‘deki eleman için bu durum doğrudur ancak 1 numaralı eleman için bu durum geçerli değildir.

Yığın sıralama



Yığın sıralama

- BUILD_MAX_HEAP fonksiyonun çalıştığını gösterebilmek için aşağıdaki döngü sabitini kullanacağız:
  - 2-3 satırlarında döngünün her çalışması öncesinde, i+1, i+2,…,n indeksli düğümlerin tamamı bir max-heap köküdür.
- Daha önce olduğu gibi bu döngü sabitinin döngü başlamadan doğru olduğunu, her döngü çalışmasından sonra doğru kaldığını ve döngü sonlandığında algoritmanın doğru çalıştığını gösterdiğini ispatlayacağız.
  - Başlangıç: Döngü ilk kez çalıştığında $i = \lfloor n/2 \rfloor$
  - Bu durumda $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ indeksli düğümlerin tamamı yaprak. Yaprak oldukları için her biri tek elemanlı bir max-heap.

Yığın sıralama

- Döngü sabiti
  - Sürdürme: i indeksli düğümün alt düğümleri i sayısından büyük indekse sahip. Döngü sabitine göre bu düğümler ilgili max-heap kökleri.
  - Bu durumda MAX_HEAPIFY(A,i) fonksiyonunu çağırabiliriz.
  - MAX_HEAPIFY fonksiyonu i+1, i+2,...,n indeksli düğümlerin max-heap kökü olma durumunu koruyor.
  - Fonksiyon çalıştığında i sayısı 1 azalacak.
  - Bu sayede bir sonraki döngü için döngü sabiti doğru oluyor.
  - Sonlanma: Döngü sonlandığında i=0. Döngü sabitine göre 1,2,...,n indeksli düğümlerin hepsi bir max-heap kökü. Ağacın tamamının kökü ise düğüm 1. Bu durumda elemanların hepsi max-heap ağacının parçası.

Yığın sıralama

- BUILD_MAX_HEAP işleminin analizi
  - BUILD_MAX_HEAP fonksiyonunun çalışma süresini şu şekilde hesaplayabiliriz:
    - MAX_HEAPIFY fonksiyonun çağırısı O(lg n) zaman alıyor.
    - Toplam O(n) tane bu tür çağrı olacak.
      - Bu durumda O(n lg n) bir üst sınır olur.
      - Ancak bu sınır sıkı bir sınır değildir.
  - MAX_HEAPIFY bir düğüm için çağırılırken, geçen süre düğümün yüksekliğine bağlı olarak değişiyor ve birçok düğümün yüksekliği oldukça düşük.
  - Yapacağımız analiz aşağıdaki iki gözleme bağlıdır:
    - n elemanlı bir heap'in yüksekliği $\lfloor lg\, n \rfloor$
    - h yükseklikte en fazla $\lceil n/2^{h+1} \rceil$ düğüm bulunur.

Yığın sıralama

- BUILD_MAX_HEAP işleminin analizi
  - MAX_HEAPIFY h yüksekliğinde bir düğüm için çağırıldığında O(h) süre alıyor. Bu durumda BUILD_MAX_HEAP çalışma süresi şu şekilde hesaplanabilir.

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right)$$

  - Son formülde toplam formulünde değişiklik yaparsak

$$\begin{aligned} \sum_{h=0}^{\infty} \frac{h}{2^h} &= \frac{1/2}{(1-1/2)^2} \\ &= 2\,. \end{aligned}$$

  - Sonuç olarak BUILD_MAX_HEAP için sıkı sınır

$$\begin{aligned} O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) \\ &= O(n)\,. \end{aligned}$$

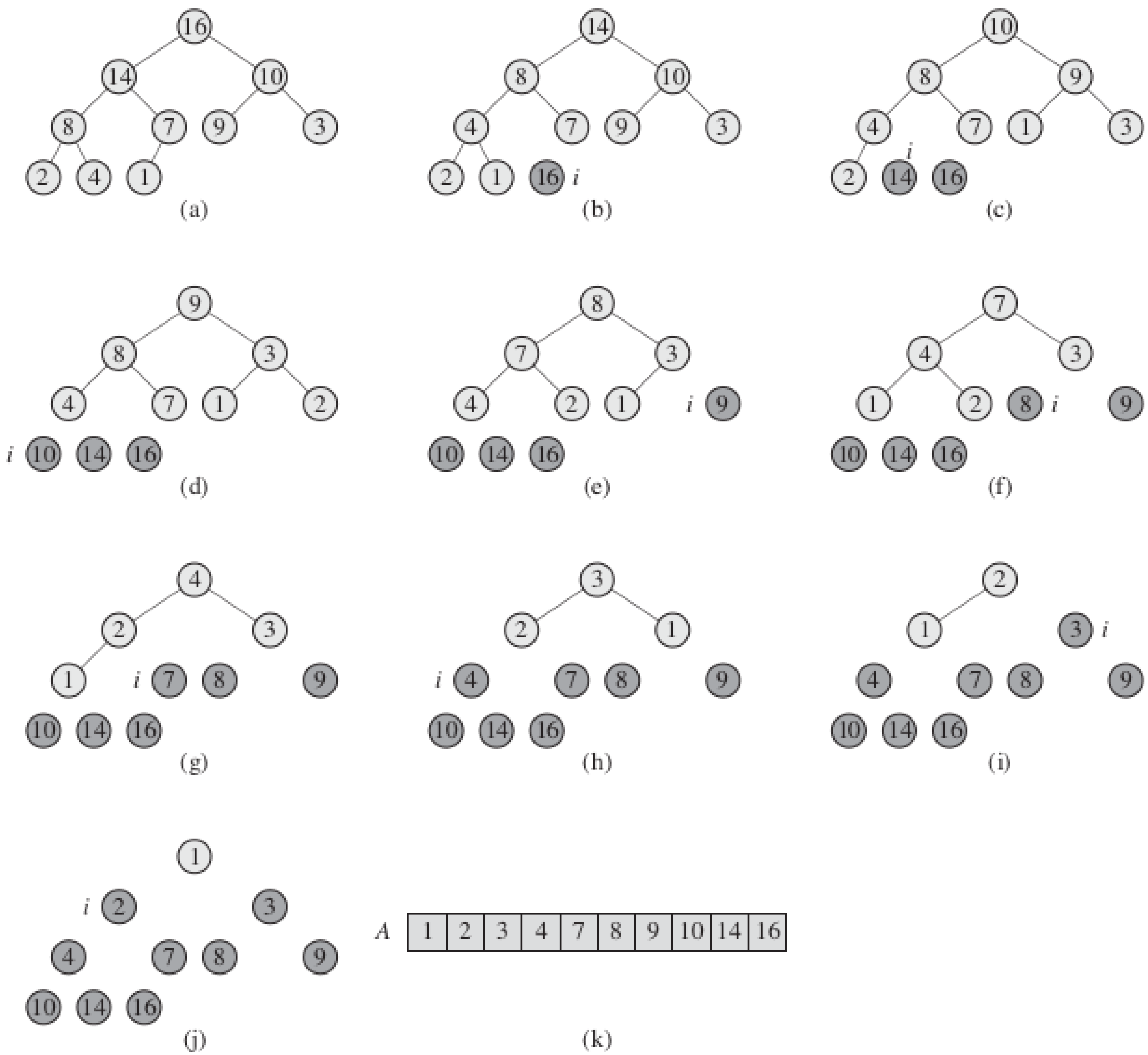
Yığın sıralama

- Yığın sıralama algoritması BUILD_MAX_HEAP fonksiyonu ile başlar ve dizideki elemanlardan oluşan bir max-heap oluşturulur.
  - Girdi olarak sıralanmamış A[1..n] dizisi alınır, n = length[A].
- Son durumda A[1] en yüksek değere sahip elemandır. Bu elemanı A[n] ile yer değiştirebiliriz.
- Eğer sona yerleştirdiğimiz elemanı dışarda bırakır ve A[1] için MAX_HEAPIFY fonksiyonunu çağırırsak, kalan A[1,..(n-1)] dizisi tekrar max-heap yapılabilir.
- Bu şekilde her eleman için aynı işemi tekrar ederek sıralı diziye ulaşabiliriz.

Yığın sıralama

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
```

Yığın sıralama



Yığın sıralama

- Yığın sıralama algoritmasının analizi
  - BUILD_MAX_HEAP fonksiyonu daha önce belirtildiği üzere O(n) süre alıyor.
  - MAX_HEAPIFY O(lg n) süre alıyor ve bu fonksiyon n-1 kere çağırılıyor.
  - Bu durumda Yığın sıralama algoritması O(n lg n) süre alır.

Hızlı sıralama

- Hızlı sıralama algoritması en kötü çalışma süresi $O(n^2)$ olan bir sıralama algoritmasıdır.
- En kötü çalışma süresi yavaş olmasına rağmen, birçok uygulamada en verimli sıralama algoritması olarak kullanılır. Bunun nedeni:
  - Ortalamaya bakıldığında, ortalama çalışma süresi $\Theta(n \lg n)$ olarak hesaplanır.
  - $\Theta(n \lg n)$ çalışma süresindeki gizli sabitler oldukça küçüktür.
  - Ayrıca Yığın sıralama ve insertion sort için olduğu gibi yerinde sıralama yapar.
- Bu bölümde öncelikle Hızlı sıralama algoritmasını tanımlayacağız.
- Daha sonra algoritmanın çalışma süresini analiz edeceğiz.

Hızlı sıralama

- Hızlı sıralama, merge sort gibi böl ve yönet yaklaşımı ile çalışır.
- Yapılan işlemler, sıralanmamış A[1..r] dizisi için şunlardır:
  - Böl: Verilen A[p..r] dizisini şu şekide parçala (veya yeniden düzenle)
    - Verilen dizi iki farklı altdiziye ayrılır. Bunlar A[p..q-1] ve A[q+1..r].
    - A[p..q-1] altdizisindeki elemanların değeri A[q]‘den azdır veya eşittir.
    - A[q+1..r] altdisizindeki elemanların değeri A[q]‘den fazladır veya eşittir.
  - Fethet: Oluşturulan iki altdiziyi yinelemeli Hızlı sıralama çağrısı ile sırala.
  - Birleştir: Altdiziler yerinde sıralandığı için son birleşme işlemine gerek yoktur. Dizinin tamamı sıralanmıştır.

Hızlı sıralama

- Hızlı sıralama algoritması aşağıdaki şekilde uygulanabilir:

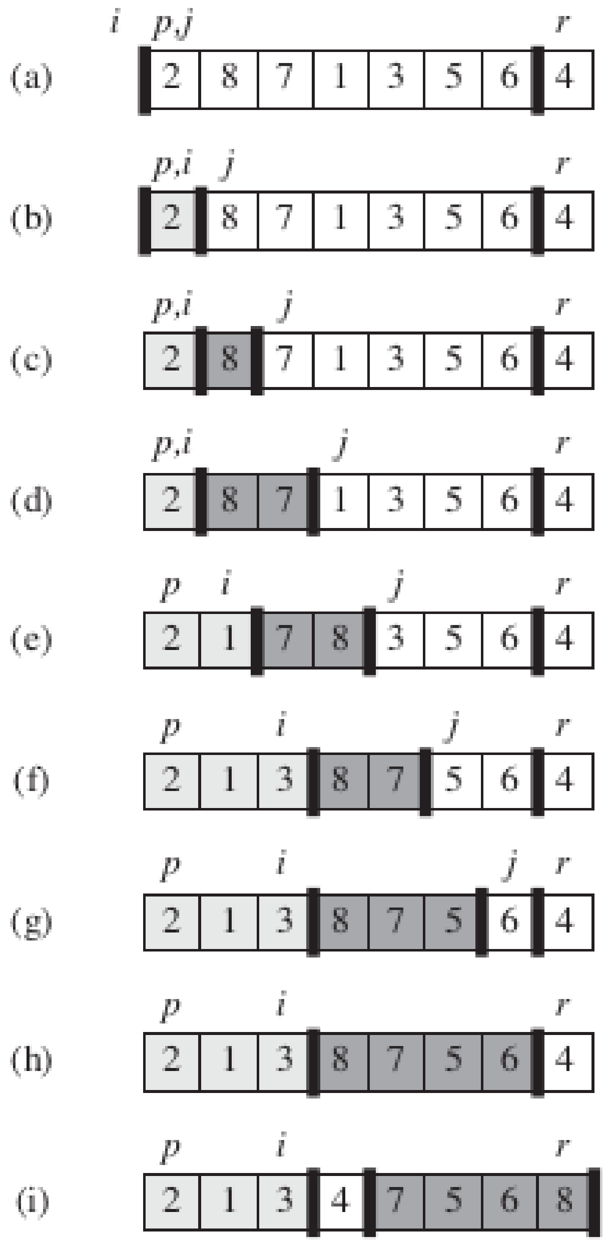
```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

Hızlı sıralama

- Verilen diziyi parçalama işlemi şu şekilde yapılır:

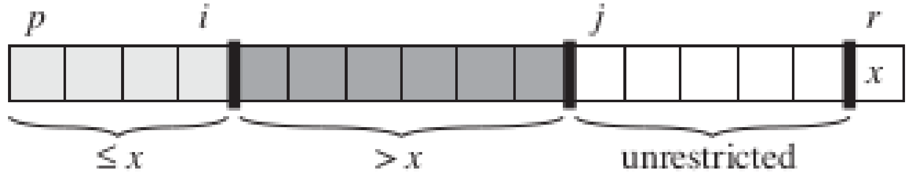
```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Hızlı sıralama



Hızlı sıralama

- Döngü sabiti:
  - 3-6 satırlarındaki döngünün her çalışmasından önce, her k indeks değeri için aşağıdaki durumlardan biri söz konusudur:
    - Eğer p ≤ k ≤ i ise A[k] ≤ x
    - Eğer i+1 ≤ k ≤ j – 1 ise A[k] > x
    - Eğer k = r ise, A[k] = x



Hızlı sıralama

- Döngü sabiti:
  - Başlangıç: Döngünün ilk çalışmasından önce i = p – 1 ve j = p. Bu durumda p ile i arasında ve i + 1 ile j – 1 arasında bir değer bulunmuyor. Öyleyse döngü sabiti doğru.
  - Sürdürme: Bir sonraki şekilde görüldüğü üzere her döngü çalışmasında iki farklı durum söz konusudur:
    - A[j] > x ise, j değeri bir artırılır. Bu durumda A[j – 1] için 2 numaralı durum söz konusudur. Diğer indekslerin durumu etkilenmemiştir.
    - A[j] ≤ x ise, i değeri bir artırılır, A[i] ile A[j]‘nin yeri değiştirilir ve j değeri bir artırılır. Sonuç olarak yer değiştirmeden dolayı A[i]<x ise 1 numaralı durum söz konusudur. Ayrıca A[j – 1] > x durumu devam etmektedir çünkü değiştirdiğimiz değer döngü sabitine göre x değerinden büyüktür.

Hızlı sıralama

- Döngü sabiti:
  - Sonlanma: Döngü sonlandığında $j = r$. Bu durumda dizideki her eleman döngü sabitinde belirtilen üç durumdan birindedir. Böylece diziyi üç ayrı parçaya ayırdık: değeri x'den küçük veya x'e eşit olanlar, değeri x'den büyük olanlar ve değeri x olan eleman.
- Algortmanın son iki satırı pivot olarak kullanılan elemanı doğru yerine yerleştirmektedir. Böylece parçalama işleminde belirtilen dizi oluşturulmuştur.
- Verilen A[p..r] dizisi için PARTITION fonksiyonu $\Theta(n)$ sürede çalışmaktadır, şöyle ki $n = r - p + 1$.

Hızlı sıralama

- Hızlı sıralama algoritmasının performans analizi
  - Hızlı sıralama algoritmasının çalışma hızı parçalama fonksiyonun oluşturduğu parçaların dengeli veya dengesiz olmasına bağlıdır.
    - Oluşan parçalar benzer büyüklükte ise algoritma asimtotik olarak merge sort kadar hızlı çalışabilmektedir.
    - Oluşan parçalar dengesiz şekilde dağılmış ise algoritma asimtotik olarak insertion sort kadar yavaş çalışabilmektedir.
  - Ön kötü parçalama durumu
    - Bu durumda parçalama mevcut problemi n – 1 elemanlı bir altproblem ve 0 elemanlı bir altprobleme ayırmaktadır.
    - Bu durumun her yinelemeli çağrıda oluştuğunu düşünelim
      - Parçalama $\Theta(n)$ kadar zaman almakta.
      - $T(n) = T(n-1) + \Theta(0) + \Theta(n)$

        $= T(n-1) + \Theta(n)$

        $= \Theta(n^2)$
    - Bu durum ne zaman oluşur?

Hızlı sıralama

- Hızlı sıralama algoritmasının performans analizi:
  - En iyi çalışma zamanı:
    - En iyi durumda parçalıyıcı problemi yaklaşık olarak iki eşit parçaya böler
      - Parçalardan biri $\lfloor n/2 \rfloor$ diğeri ise $\lceil n/2 \rceil - 1$ büyüklüğünde olacaktır.
      - Bu durumda her iki problemin büyüklüğüde n/2 ile sınırlıdır.
    - T(n) ≤ 2T(n/2) + Θ(n)
    - Master metodu kullanırsak
      - O(n lg n)