Algorithm Analysis And Design

With Infinity Teams

Eng: Abdulrahman Hamdi

INFINITY TEAM
Since 2020

Table of contents

Eng: Abdulrahman Hamdi

Table of contents

Eng: Abdulrahman Hamdi

09

Transform-and-Conquer

Eng: Abdulrahman Hamdi

Transform-and-Conquer Çerçevesi

- Transform and Conquer introduction

- Heaps and Heapsort:
  - Notion of the Heap
  - Heapsort

Eng: Abdulrahman Hamdi

Transform-and-Conquer Algoritmaları

Bu bölüm, dönüşüm fikrine dayanan bir grup tasarım yöntemini ele almaktadır. Bu genel tekniğe dönüştür ve yönet (transform-and-conquer) adını veriyoruz çünkü bu yöntemler iki aşamalı prosedürler olarak çalışır. İlk olarak, dönüşüm aşamasında, problemin örneği (instance), bir şekilde çözümü daha kolay hale getirmek için değiştirilir. Ardından, ikinci yani yönetme aşamasında, bu değiştirilmiş örnek çözülür. Bu fikrin, verilen bir örneğin neye dönüştürüldüğüne göre değişen üç ana çeşidi vardır (Şekilde):

1- Aynı problemin daha basit veya daha uygun bir örneğine dönüşüm — buna örnek basitleştirme (instance simplification) diyoruz.
2- Aynı örneğin farklı bir temsiline dönüşüm — buna temsil değişimi (representation change) diyoruz.
3- Halihazırda bir algoritmanın mevcut olduğu farklı bir problemin örneğine dönüşüm — buna problem indirgeme (problem reduction) diyoruz.

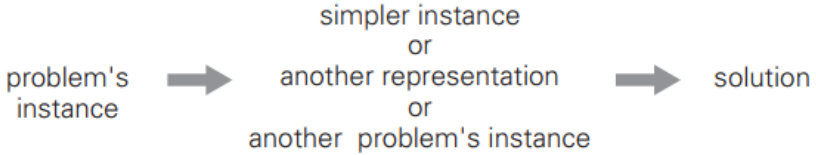


FIGURE Transform-and-conquer strategy.

Transform-and-Conquer Algoritmaları

Bu bölümün ilk üç kısmında, örnek basitleştirme türüne ait örneklerle karşılaşacağız. İlk bölümü, basit ama verimli bir fikir olan ön sıralama (presorting) ile ilgilidir. Birçok algoritmik problem, girdileri sıralı olduğunda daha kolay çözülür. Elbette, sıralamanın getirdiği faydalar, sıralamanın kendisinin maliyetinden fazla olmalıdır.

İkinci bölümü, heap veri yapısını ve heapsort sıralama algoritmasını sunar. Bu önemli veri yapısını ve sıralamadaki uygulamasını zaten biliyor olsanız bile, onları dönüştür ve yönet (transform-and-conquer) tasarımı bakış açısından incelemekten yine de fayda sağlayabilirsiniz.

Eng: Abdulrahman Hamdi

Transform-and-Conquer Algoritmaları

• Ön Sıralama (Presorting)

Ön sıralama, bilgisayar bilimlerinde eski bir fikirdir. Aslında, sıralama algoritmalarına duyulan ilginin önemli bir nedeni, bir listenin sıralı olduğunda birçok sorunun daha kolay yanıtlanabilmesidir. Elbette, sıralama içeren algoritmaların zaman verimliliği, kullanılan sıralama algoritmasının verimliliğine bağlı olabilir. Basitlik adına, bu bölüm boyunca listelerin dizi (array) olarak temsil edildiğini varsayıyoruz, çünkü bazı sıralama algoritmaları dizi yapısıyla daha kolay uygulanabilir.

Şu ana kadar üç temel sıralama algoritmasını ele aldık — seçmeli sıralama (selection sort), kabarcık sıralama (bubble sort) ve eklemeli sıralama (insertion sort) — ki bunlar en kötü ve ortalama durumlarda kare zaman (quadratic) karmaşıklığa sahiptir. Ayrıca iki gelişmiş algoritmadan da bahsettik: birleştirmeli sıralama (mergesort), ki bu her durumda Θ(n log n) karmaşıklığına sahiptir; ve hızlı sıralama (quicksort), ki ortalama durumda yine Θ(n log n) olsa da, en kötü durumda kare zaman karmaşıklığına sahiptir.Peki, daha hızlı sıralama algoritmaları var mı? Daha önce 3. slaytta belirttiğimiz gibi, karşılaştırmaya dayalı genel bir sıralama algoritması için... (devamı eksik).

Eng: Abdulrahman Hamdi

Transform-and-Conquer Algoritmaları

• Ön Sıralama (Presorting)

Ön sıralama, bilgisayar bilimlerinde eski bir fikirdir. Aslında, sıralama algoritmalarına duyulan ilginin önemli bir nedeni, bir listenin sıralı olduğunda birçok sorunun daha kolay yanıtlanabilmesidir. Elbette, sıralama içeren algoritmaların zaman verimliliği, kullanılan sıralama algoritmasının verimliliğine bağlı olabilir. Basitlik adına, bu bölüm boyunca listelerin dizi (array) olarak temsil edildiğini varsayıyoruz, çünkü bazı sıralama algoritmaları dizi yapısıyla daha kolay uygulanabilir.

Şu ana kadar üç temel sıralama algoritmasını ele aldık — seçmeli sıralama (selection sort), kabarcık sıralama (bubble sort) ve eklemeli sıralama (insertion sort) — ki bunlar en kötü ve ortalama durumlarda kare zaman (quadratic) karmaşıklığa sahiptir. Ayrıca iki gelişmiş algoritmadan da bahsettik: birleştirmeli sıralama (mergesort), ki bu her durumda Θ(n log n) karmaşıklığına sahiptir; ve hızlı sıralama (quicksort), ki ortalama durumda yine Θ(n log n) olsa da, en kötü durumda kare zaman karmaşıklığına sahiptir.Peki, daha hızlı sıralama algoritmaları var mı? Daha önce 3. slaytta belirttiğimiz gibi, karşılaştırmaya dayalı genel bir sıralama algoritması için... (devamı eksik).

Eng: Abdulrahman Hamdi

Transform-and-Conquer Algoritmaları

• Ön Sıralama (Presorting)

Hiçbir genel karşılaştırmaya dayalı sıralama algoritması, en kötü durumda $\Theta(n \log n)$ karmaşıklığından daha iyi bir verimliliğe sahip olamaz; aynı sonuç ortalama durum verimliliği için de geçerlidir.Aşağıda, ön sıralama (presorting) fikrini açıklayan üç örnek yer almaktadır. Bu bölümün alıştırmalarında daha fazla örnek bulabilirsiniz.

ÖRNEK 1 – Bir dizideki elemanların benzersizliğini kontrol etmeBu eleman benzersizliği problemi size tanıdık geliyorsa, gelmeli; bu problemi Slayt 6'da kaba kuvvet (brute-force) algoritması ile ele almıştık (bkz. Örnek 2). Kaba kuvvet algoritması, dizideki eleman çiftlerini karşılaştırarak iki eşit eleman bulunana kadar veya karşılaştıracak çift kalmayana kadar ilerliyordu.

Bu algoritmanın en kötü durumdaki verimliliği $\Theta(n^2)$ idi. Alternatif olarak, önce diziyi sıralayabiliriz ve ardından yalnızca ardışık (yani yan yana olan) elemanları kontrol edebiliriz: eğer dizide eşit elemanlar varsa, bu elemanların bir çifti mutlaka yan yana olacaktır — ve tam tersi de geçerlidir.

```
ALGORITHM PreSortElementUniqueness(A[0..n - 1])
//Eleman tekliği problemini önce diziyi
    sıralayarak çözer
//Girdi: Sıralanabilir elemanlardan oluşan bir
    dizi A[0..n-1]
//Çıktı: A'da eşit eleman yoksa "doğru" (true),
    aksi takdirde "yanlış" (false) döndürür
for i ← 0 to n - 2 do
  if A[i] = A[i + 1] return false
return true
```

Transform-and-Conquer Algoritmaları

• Ön Sıralama (Presorting)

Bu algoritmanın çalışma süresi, sıralama işlemi için harcanan süre ile ardışık elemanları kontrol etme süresinin toplamıdır. İlk aşama (sıralama) en az n log n karşılaştırma gerektirir, ikinci aşama (kontrol) ise en fazla n – 1 karşılaştırma gerektirir. Dolayısıyla, algoritmanın genel verimliliğini belirleyen kısım sıralama aşaması olacaktır.Bu nedenle, burada kare-zamanlı (quadratic) bir sıralama algoritması kullanırsak, algoritmanın tamamı kaba kuvvet yönteminden daha verimli olmayacaktır. Ancak, en kötü durumda θ(n log n) verimliliğe sahip mergesort gibi iyi bir sıralama algoritması kullanırsak, ön sıralamaya dayalı bu algoritmanın en kötü durumdaki verimliliği de θ(n log n) olur:

$$T(n) = T_{\text{sort}}(n) + T_{\text{scan}}(n) \in \Theta(n \log n) + \Theta(n) = \Theta(n \log n)$$

ÖRNEK 2 – Moda (En Sık Görülen Değer) HesaplamaModa, bir sayı listesindeki en sık görülen değerdir. Örneğin, 5, 1, 5, 7, 6, 5, 7 listesi için moda değeri 5'tir. (Eğer birden fazla farklı değer aynı sıklıkla görülüyorsa, bunların herhangi biri moda olarak kabul edilebilir.)Kaba kuvvet yaklaşımı, listeyi tarayıp tüm farklı değerlerin frekanslarını hesaplayarak, ardından en yüksek frekansa sahip olan değeri bulur.Bu fikri uygulamak için, daha önce karşılaşılan değerleri ve bunların frekanslarını ayrı bir listede saklayabiliriz. Her yinelemede, orijinal listenin i. elemanı, bu yardımcı listede daha önce karşılaşılmış olan değerlerle karşılaştırılır. Eşleşen bir değer bulunursa, frekansı artırılır; aksi takdirde, mevcut eleman, frekansı 1 olarak yardımcı listeye eklenir.

Eng: Abdulrahman Hamdi

Transform-and-Conquer Algoritmaları

• **Ön Sıralama (Presorting)**

Sonuç olarak, bu algoritmanın frekans listesi oluştururken yaptığı en kötü durum karşılaştırma sayısı:

$$C(n) = \sum_{i=1}^{n} (i-1) = 0 + 1 + \cdots + (n-1) = \frac{(n-1)n}{2} \in \Theta(n^2).$$

Yardımcı listedeki en yüksek frekansı bulmak için gereken ek n-1 karşılaştırma, algoritmanın kare-zamanlı (quadratic) en kötü durum verimliliğini değiştirmez.Buna alternatif olarak, önce girdiyi sıralayalım. Böylece aynı değerlere sahip tüm elemanlar dizide yan yana gelmiş olacaktır.Modayı (en sık tekrar eden değeri) bulmak için yapmamız gereken tek şey, sıralanmış dizideki en uzun ardışık aynı değerlerden oluşan alt diziyi (run) bulmaktır.

```
ALGORITHM PresortMode(A[0..n - 1])
//Önce diziyi sıralayarak bir dizinin modunu
(en sık tekrar eden elemanını) hesaplar
//Girdi: Sıralanabilir elemanlardan oluşan bir dizi A[0..n - 1]
//Çıktı: Dizinin modu
dizi A'yı sırala
sort the array A
i ← 0 //current run begins at position i
modefrequency ← 0 //highest frequency seen so far
while i ≤ n - 1 do
  runlength ← 1; runvalue ← A[i]
  while i + runlength ≤ n - 1 and A[i + runlength]=runvalue
    runlength ← runlength + 1
  if runlength > modefrequency
    modefrequency ← runlength; modevalue ← runvalue
  i ← i + runlength
return modevalue
```

Eng: Abdulrahman Hamdi

Transform-and-Conquer Algoritmaları

• Heap ve Heapsort

"Heap" (yığın) adı verilen veri yapısı, standart bir sözlükteki anlamının çağrıştırdığı gibi düzensiz bir yığın değildir. Aksine, bu yapı kısmen sıralı, zekice tasarlanmış bir veri yapısıdır ve özellikle öncelik kuyruklarının (priority queues) uygulanması için oldukça uygundur.Hatırlayacak olursak, bir öncelik kuyruğu, her öğenin "öncelik" adı verilen sıralanabilir bir özelliğe sahip olduğu, çoklu öğelerden oluşan bir kümedir ve aşağıdaki işlemleri destekler:

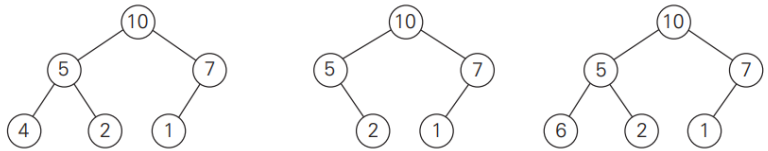


FIGURE Illustration of the definition of heap: only the leftmost tree is a heap.

- En yüksek (yani en büyük) önceliğe sahip öğeyi bulmak
- En yüksek önceliğe sahip öğeyi silmek
- Çoklu kümeye yeni bir öğe eklemek

Transform-and-Conquer Algoritmaları

• **Heap ve Heapsort**

Heap Sort Algoritması

Öncelikle diziyi heapify işlemiyle bir max heap (maksimum yığın) haline dönüştürün. Bu işlemin yerinde (in-place) yapıldığını unutmayın; yani dizi elemanları, yığın özelliklerine uygun olacak şekilde yeniden düzenlenir. Daha sonra, Max Heap'in kök düğümünü (en büyük eleman) teker teker silip son düğümle değiştirin ve yeniden heapify işlemi uygulayın. Bu işlemi, yığının boyutu 1'den büyük olduğu sürece tekrarlayın.

Adımlar:

1- Dizi elemanlarını yeniden düzenleyerek bir Max Heap oluşturun.

2- Aşağıdaki adımları yığında yalnızca bir eleman kalana kadar tekrarlayın:

- Yığının kök elemanını (şu anki yığındaki en büyük eleman) yığının son elemanıyla takas edin.
- Yığının son elemanını kaldırın (artık doğru konumundadır). Aslında diziden elemanı kaldırmak yerine sadece yığının boyutunu azaltırız.
- Geriye kalan yığın elemanları üzerinde heapify işlemini uygulayın.
- Sonuç olarak sıralı bir dizi elde edilir.

Eng: Abdulrahman Hamdi

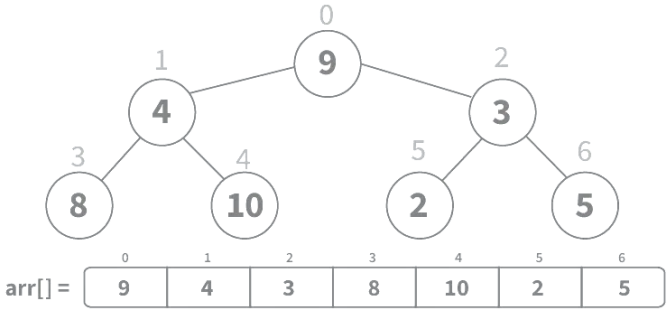
Transform-and-Conquer Algoritmaları

• **Heap ve Heapsort**

Heap Sort'un Ayrıntılı Çalışma Prensibi

Adım 1: Diziyi Tam Bir İkili Ağaç (Complete Binary Tree) Olarak Düşünmek

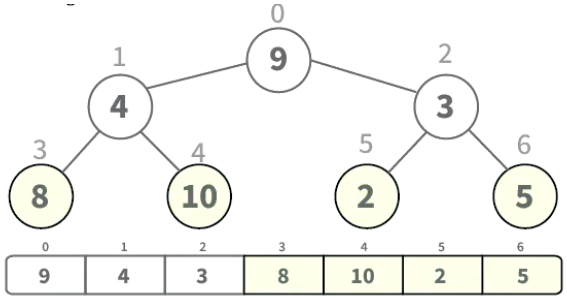
Öncelikle diziyi tam bir ikili ağaç olarak hayal etmemiz gerekir. Boyutu n olan bir dizi için, kök eleman 0. indekste yer alır. İndeksi i olan bir elemanın sol çocuğu 2i + 1, sağ çocuğu ise 2i + 2 indislerinde bulunur.



Adım 2: Max Heap Oluşturma

Adım 01

Her düğümü çocuklarıyla karşılaştırın ve ebeveyn düğümlerin daha büyük olduğundan emin olun.Bu işlem, küçük düğümlerin aşağıya inmesine ve büyük düğümlerin yukarıya çıkmasına neden olur.Yaprak düğümlerle başlayalım.



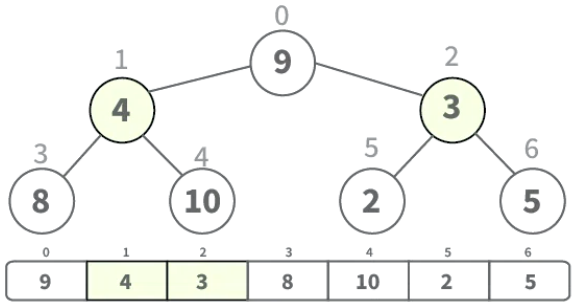
Eng: Abdulrahman Hamdi

Transform-and-Conquer Algoritmaları

• **Heap ve Heapsort**

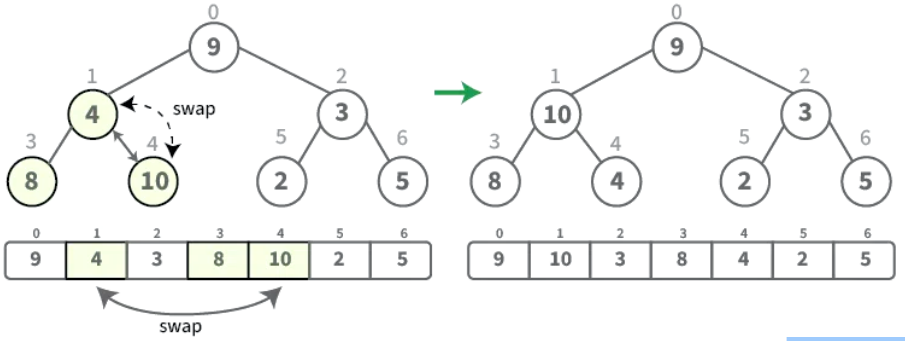
Adım 02

Şimdi bir üst seviyeye bakalım (düğüm 4 ve düğüm 3).



Adım 03

Düğüm 4, çocuk düğümünden (10) daha küçük,bu yüzden ebeveynin çocuklarından daha büyük olması gerektiği kuralını korumak içindaha büyük çocukla yer değiştirin.



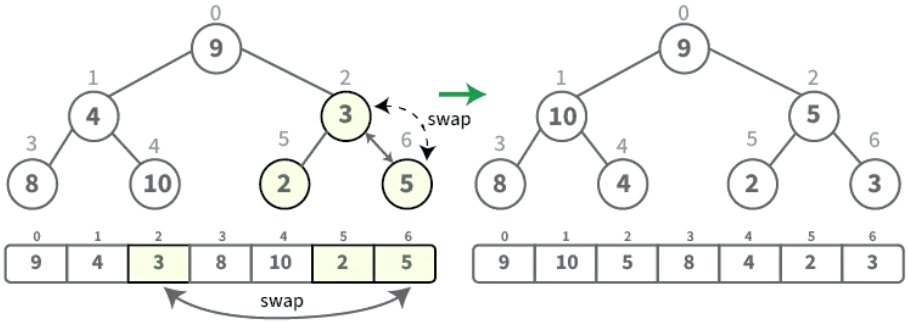
Eng: Abdulrahman Hamdi

Transform-and-Conquer Algoritmaları

• **Heap ve Heapsort**

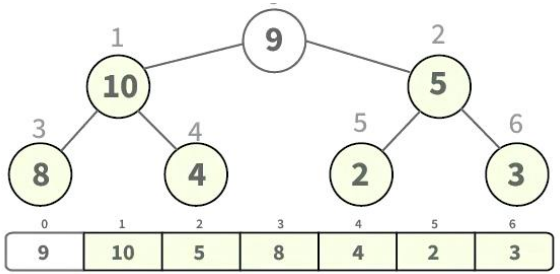
Adım 04

Geçerli seviyedeki bir sonraki düğüme (3) bakalım.Bu düğümün daha büyük bir çocuğu olduğundan,ebeveynin daha büyük bir değere sahip olması kuralını sağlamak için yer değiştirin.



Adım 05

Geçerli seviyeyi tamamladıktan sonra,artık elimizde iki küçük fakat geçerli max heap (maksimum yığın) var.



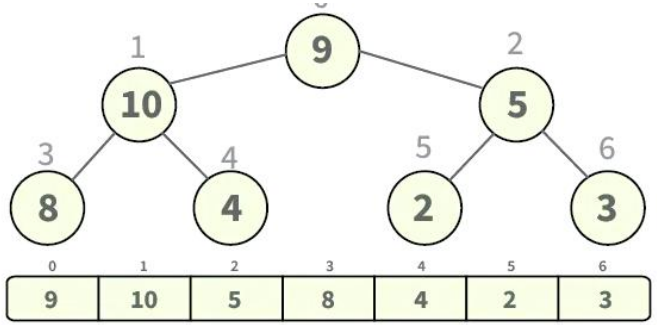
Eng: Abdulrahman Hamdi

Transform-and-Conquer Algoritmaları

• **Heap ve Heapsort**

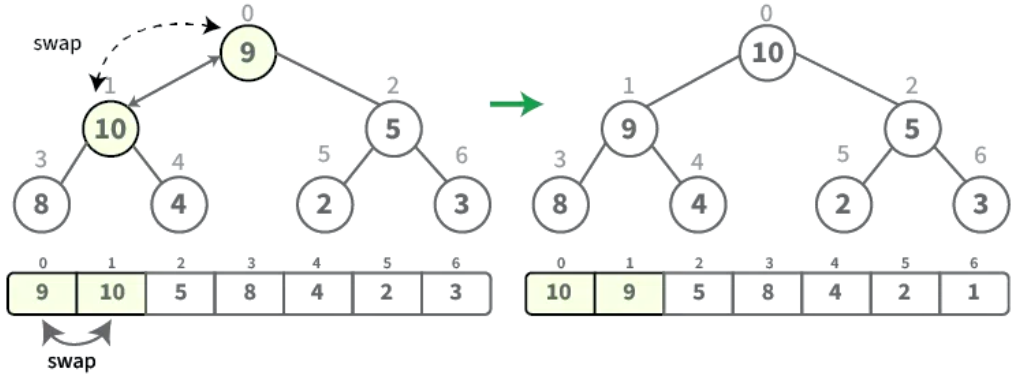
Adım 06

Şimdi bir üst seviyeye geçelim.
Burada kök düğüm olarak 9 var.



Adım 07

Düğüm 9, çocuk düğümünden (10) daha küçük,bu yüzden ebeveynin çocuklarından daha büyük olması gerektiği kuralını sağlamak içindaha büyük çocukla yer değiştirin.



Eng: Abdulrahman Hamdi

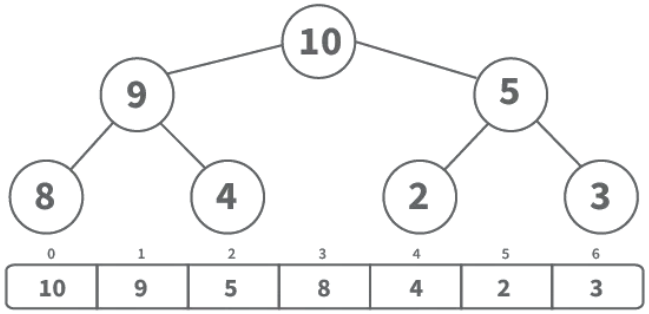
Transform-and-Conquer Algoritmaları

• **Heap ve Heapsort**

Adım 3: Diziyi Sırala – En Büyük Elemanı Sıralanmamış Dizinin Sonuna Yerleştir

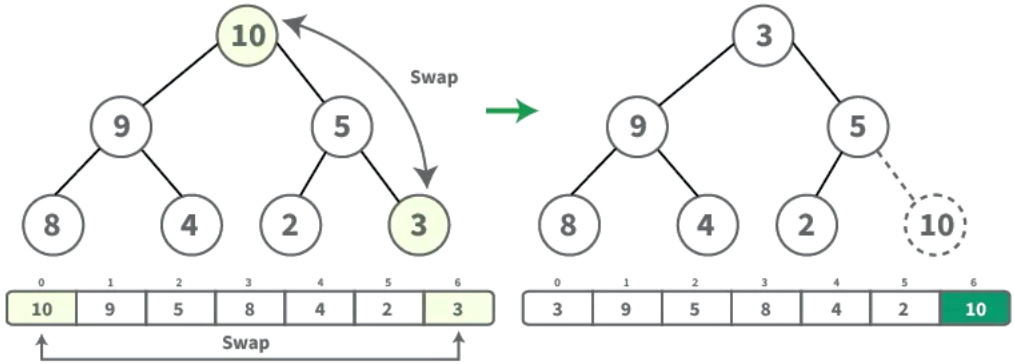
Adım 01

Diyelim ki elimizdeki dizi, max heap (maksimum yığın) özelliğine uygun hale getirildi. İşte dizimizin max heap biçimindeki görünümü:



Adım 02

Maksimum eleman olan 10 ile sıralanmamış dizideki son eleman (3) yer değiştirilir. Yığının boyutu bir azaltılır (son eleman artık sıralandığı için dikkate alınmaz).



Eng: Abdulrahman Hamdi

Transform-and-Conquer Algoritmaları

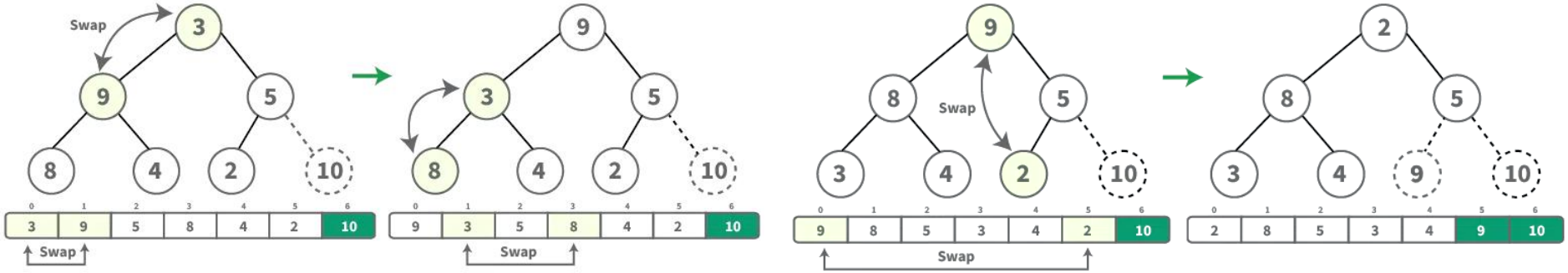
• **Heap ve Heapsort**

Adım 03

Şimdi, kök düğüm max-heap özelliğini ihlal ediyor, bu yüzden heapify işlemi uygulanır. Düğüm 3, en büyük çocuğu olan 9 ile yer değiştirilir. Ancak hâlâ düğüm 3'ün kendisinden büyük çocukları var, bu yüzden en büyük olanı (düğüm 8) ile tekrar yer değiştirilir.

Adım 04

Şimdi elimizde geçerli bir max heap var. Maksimum eleman olan 9, sıralanmamış dizideki son eleman (2) ile yer değiştirilir. Ardından yığının boyutu bir azaltılır (sondan bir önceki eleman artık sıralandığı için dikkate alınmaz).

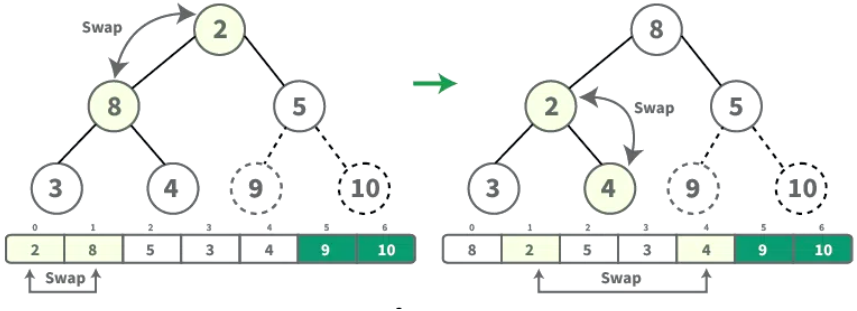


Transform-and-Conquer Algoritmaları

• **Heap ve Heapsort**

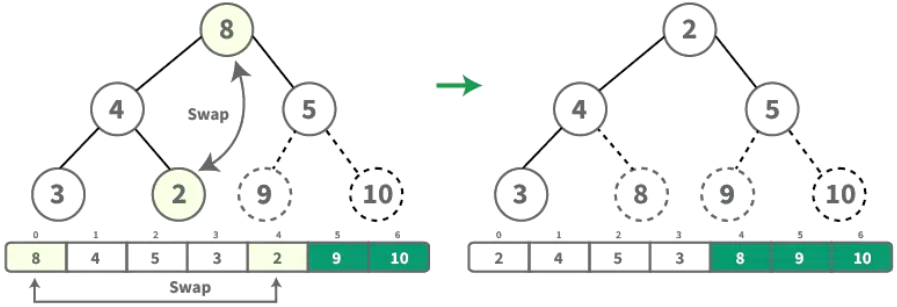
Adım 05

Şimdi kök düğüm max-heap özelliğini ihlal ediyor, bu yüzden heapify işlemi uygulanır. Düğüm 2, en büyük çocuğu olan 8 ile yer değiştirilir. Ancak düğüm 2'nin hâlâ kendisinden büyük çocukları var, bu yüzden en büyük olanıyla (düğüm 4) tekrar yer değiştirilir.



Adım 06

Şimdi elimizde geçerli bir max heap var. Maksimum eleman olan 8, sıralanmamış dizideki son eleman (2) ile yer değiştirilir. Ardından yığının boyutu bir azaltılır (üçüncü sondaki eleman artık sıralandığı için dikkate alınmaz).



Eng: Abdulrahman Hamdi

Transform-and-Conquer Algoritmaları

- **Heap ve Heapsort**

Yukarıdaki görselde, diziyi sıralamak için bazı adımları gösterdik. Bu adımları, yığında yalnızca bir eleman kalana kadar tekrarlamamız gerekir.

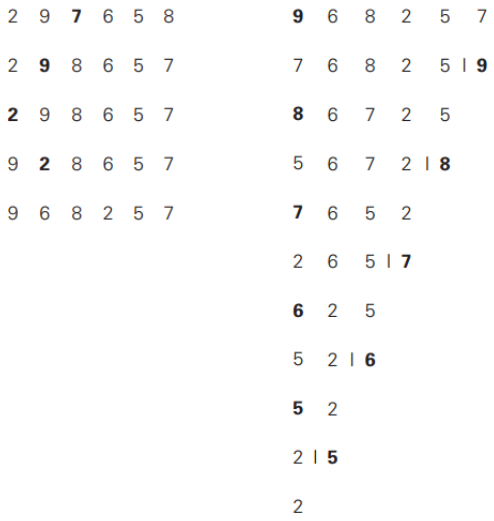


FIGURE Sorting the array 2, 9, 7, 6, 5, 8 by heapsort.

Şimdi Pseudo kodu geçeriz:

```
HEAPIFY(H, n, i)
    largest ← i
    left ← 2 * i
    right ← 2 * i + 1

        if left ≤ n and H[left] > H[largest]
        largest ← left

        if right ≤ n and H[right] > H[largest]
        largest ← right

        if largest ≠ i
        swap H[i] and H[largest]
        HEAPIFY(H, n, largest)
```

Transform-and-Conquer Algoritmaları

- **Heap ve Heapsort**

```
BUILD_MAX_HEAP(H, n)
// Build max-heap from unsorted array H[1..n]
for i ← ⌊n / 2⌋ downto 1
       HEAPIFY(H, n, i)
```

```
HEAP_SORT(H, n)
// Sort array H[1..n]
BUILD_MAX_HEAP(H, n)

    for i ← n downto 2
       swap H[1] and H[i]      // move max to end
       n ← n - 1               // reduce heap
    size
       HEAPIFY(H, n, 1)        // fix the heap
```

Eng: Abdulrahman Hamdi

Transform-and-Conquer Algoritmaları

• **Heap ve Heapsort**

Zaman Karmaşıklığı

Aşama 1: Max-Heap Oluşturma Dizi, BUILD_MAX_HEAP fonksiyonu kullanılarak bir max-heap'e dönüştürülür. Bu işlem, her yaprak olmayan düğüm için HEAPIFY fonksiyonunu çağırır; yani n/2 indeksinden 1'e kadar olan düğümler üzerinde çalışır.

Matematiksel Analiz: HEAPIFY fonksiyonunun her çağrısı, en fazla O(h) zaman alır, burada h, düğümün yüksekliğidir. n boyutundaki bir ikili yığının yüksekliği yaklaşık log n'dir. Ancak düğümlerin çoğu yapraklara yakındır, bu yüzden yükseklikleri küçüktür. Yığını oluşturmanın toplam maliyeti: En kötü durumda bile O(n) zaman alır.

$$T(n) = \sum_{i=1}^{\lfloor n/2 \rfloor} O(\log n) = O(n)$$

Eng: Abdulrahman Hamdi

Aşama 2: Sıralama (En Büyük Elemanı Çıkarma – Extract Max)En büyük elemanı çıkarmak ve yığını düzeltmek için n-1 işlem gerçekleştiririz. Her çıkarma işlemi şu adımlardan oluşur:1 adet yer değiştirme (swap) işlemi: O(1)1 adet HEAPIFY çağrısı: O(log n) zaman alır (en kötü durumda heapify, kökten yaprağa kadar iner)

Dolayısıyla her çıkarma işleminin maliyeti:

$$T(n) = (n-1) \cdot O(\log n) = O(n \log n)$$

Toplam Zaman Karmaşıklığı:

O(n) + O(n log n) = O(n log n)

Bu karmaşıklık, girdi sırasından bağımsızdır; yani en iyi, ortalama ve en kötü durumların hepsi için zaman karmaşıklığı O(n log n) olur.

Algorithm Analysis And Design

Thanks for listening!

Eng: Abdulrahman Hamdi