***HEAPS*** ***9***  *215*

"Heap"(“YIĞIN”) kelimesi, Bilgisayar Bilimlerinde birkaç farklı bağlamda kullanılır. Bir heap bazen dinamik (yani çalışma zamanı) bellek tahsisi için kullanılan bir bellek alanını ifade eder. Diğer bir anlamı ise, bu bölümün konusu olan, kavramsal olarak tam bir binary tree (“İKİLİ AĞAÇ”) olan bir veri yapısıdır. Heaps, öncelikli kuyrukların, heapsort (“YIĞIN SIRALAMASI”) algoritmasının ve bazı grafik algoritmalarının uygulanmasında kullanılır. Heaps, binary tree aramaya benzer şekilde içindeki öğelerin bir sırasını korur. Ancak bir heap, öğelerinin tam bir sıralamasını korumaz. Bu durum, bir heap'in nasıl kullanılacağına dair bazı sonuçlar doğurur.

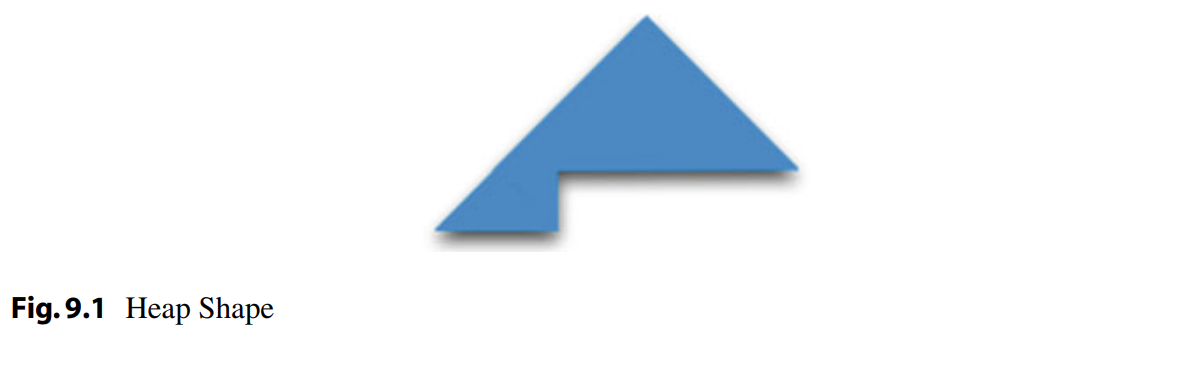
***9.1 Bölüm Hedefleri***

Bu bölümün sonunda aşağıdaki soruları yanıtlayabilecek durumda olmalısınız:  
• Heap nedir ve nasıl kullanılır?  
• Bir heap'e öğe ekleme ve silme işlemlerinin zaman karmaşıklığı nedir?  
• Bir heap kullanarak öğe araması yapar mıydınız, yapmaz mıydınız?  
• Ne zaman heap kullanırsınız?  
• Heapsort algoritmasında, neden en büyük öğenin üstte olduğu bir heap oluşturmak avantajlıdır?

***9.2*** Ana Fikirler

Heaps’i anlamak için önce bir tanımla başlayalım. En büyük öğe üstte olan bir heap, her düğümün (varsa) tüm çocuklarından büyük veya ona eşit olduğu tamamlanmış sıralı bir ağaçtır. Bu tanımı daha iyi anlamak için bir örnek yardımcı olacaktır. Kavramsal olarak, bir heap, her seviyede tam olan, ancak en alt seviye dışında kalan ve soldan sağa doğru doldurulan bir ağaçtır. Bu, Şekil **9.1**'de gösterilen genel şekli alır.

© Springer International Publishing Switzerland 2015 K.D. Lee and S. Hubbard, Data Structures and Algorithms with Python, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-13072-9\_9+(Bu, bir kitap veya akademik yayının alıntısıdır. )

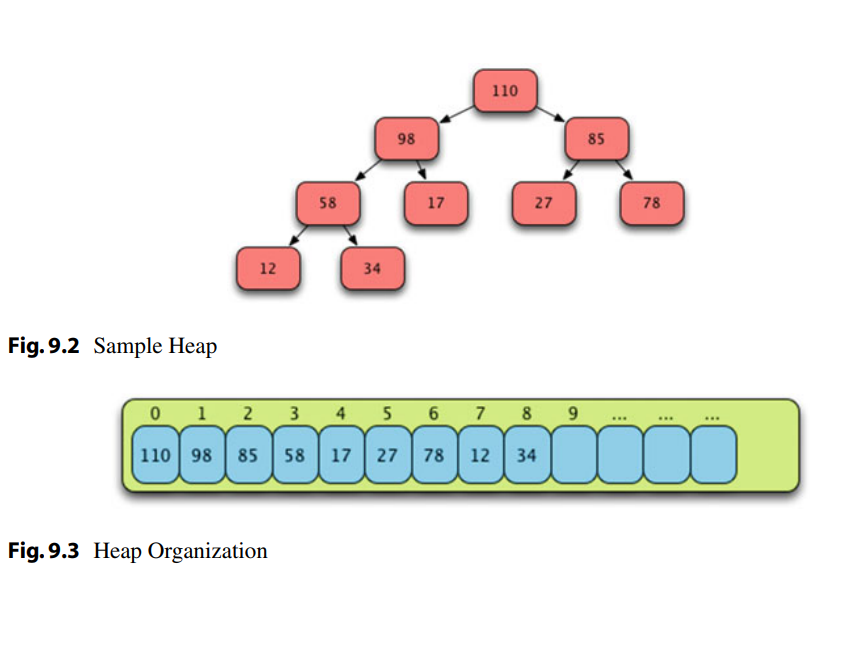


***216***

Kavramsal olarak bir heap, bir ağaçtır, ancak heap'ler genellikle ağaçlar olarak depolanmaz. Tam bir ağaç, her seviyede tam olan, ancak en alt seviyenin soldan sağa doğru doldurulduğu bir ağaçtır. Heap'ler tam ağaçlar olduğu için, bir array (DİZİ) içinde depolanabilirler. Bir örnek, heap'leri ve tamlık özelliğini daha iyi anlamaya yardımcı olacaktır. Üstte olan en büyük heap’i, root node (kök düğüm) bir dizide 0. indeksde saklandığını düşünün. Kavramsal olarak, Şekil **9.2**, tam sayılar içeren bir heap'tir. Bu kavramsal versiyondaki veriler, kök düğümden (root node) başlayarak ağacı seviyeler halinde gezerek bir array’da(dizide) saklar. Şekil 9.2'deki kavramsal heap, Şekil 9.3'te düzenlendiği gibi bir array’da(dizide) saklanır. Bir heap'in sergilediği iki özellik vardır. Bunlar şunlardır:

• **Heap Yapısı Özelliği(Heap Structure Property)**: Heap'in öğeleri, tam sıralı bir ağaç oluşturur.  
• ***Heap Sıra Özelliği(Heap Order Property)***: Her ebeveyn, tüm çocuklarından (ve tüm torunlardan) büyük veya onlara eşittir.

Şekil 9.2'deki heap, bu iki özelliği korur. Şekil 9.3'teki bu heap'in dizi(array) ile uygulanması da bu özellikleri korur. Özelliklerin dizi uygulamasında nasıl korunduğunu görmek için, çocukların ve ebeveynlerin konumlarını hesaplayabilmemiz gerekir. Dizideki herhangi bir öğenin çocukları,ebeveynin indeksinden hesaplanabilir.

leftChildIndex = 2 ∗ parentIndex + 1 / rightChildIndex = 2 ∗ parentIndex + 2

217 Bu formülleri Şekil 9.3 üzerinde kullandığımızda, kök düğümünün (root node) (yani 0. indeks) çocuklarının 98 (1. indeks) ve 85 (2. indeks) olduğunu görebiliriz. Aynı şekilde, 85'in çocukları 5. ve 6. indekslerde yer alır; bu indekslerdeki değerler 27 ve 78'dir, ve bunların kavramsal modeldeki aynı çocuklar olduğunu doğrulayabiliriz.

Tabii ki, her düğümün bir veya iki çocuğu olmayabilir. Eğer hesaplanan leftChildIndex (sol çocuk indeksi) veya rightChildIndex (sağ çocuk indeksi), heap'teki değerlerin sayısından büyük veya ona eşitse, o zaman söz konusu düğüm bir yaprak düğümdür(leaf node ) .

Ayrıca, diğer yönde de ilerlemek mümkündür. Bir çocuğun indeksine göre, ebeveynin nerede olduğunu keşfetmek mümkündür. Bu, şu formülle hesaplanabilir:

parentIndex = (childIndex − 1)//2

Önceki formüldeki // işareti, tam sayı bölmesini temsil eder. Bu, sonucun her zaman bir tamsayı olduğu anlamına gelir. Eğer kesirli bir kısım olursa (varsa), bir sonraki küçük tamsayıya yuvarlanır. Dolayısıyla, Şekil 9.3'teki 34'ün ebeveyninin indeksi şu şekilde hesaplanır:

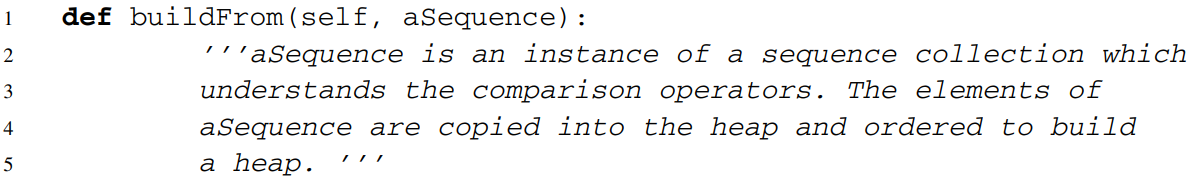
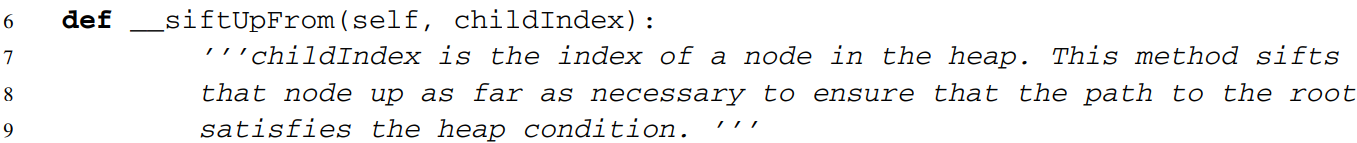
parentIndex = (8 − 1)//2 = 3

Şekil 9.2'deki kavramsal modele bakıldığında, dizideki 3. indeksdeki değerin, yani 58'in gerçekten 34'ün ebeveyni olduğu görülmektedir. Ayrıca, her düğümün bir ebeveyni olmadığını belirtmek önemlidir. Özellikle, 0. indeksteki kök düğümünün(root node) ebeveyni yoktur. Heap'teki diğer tüm düğümler ise ebeveyne sahiptir.

***9.3 Bir Heap İnşa Etmek (Building a Heap)***

Artık bir heap'in nasıl göründüğünü öğrendiğimize göre, bir heap'in nasıl inşa edileceğini inceleyeceğiz. Heap'ler, ya en büyük öğe üstte ya da en küçük öğe üstte olacak şekilde inşa edilebilir. Biz, en büyük öğe üstte olan bir heap inşa edeceğiz. Bir Heap sınıfı (class), bir heap inşa etmek için gereken veri ve yöntemleri kapsülleyecektir. Heap nesneleri (objects), içinde depolanan öğelerin listesini ve sayısını içerir. Bu sayıyı heap'in boyutu olarak adlandıracağız. Veriyi kapsüllemek için, bir dizi değer alıp bunlardan bir heap inşa edecek bir yönteme ihtiyacımız olacak. Bu yöntemi buildFrom olarak adlandıracağız. Ayrıca, özel bir yöntem de faydalı olacaktır. buildFrom yöntemi, sırasıyla her öğeyi heap içinde doğru pozisyona getirmek için \_siftUpFrom yöntemini çağıracaktır.

***9.3.1 buildFrom Yöntemi (9.3.1 The buildFrom Method)***

"aSequence, karşılaştırma operatörlerini anlayan bir dizi koleksiyon örneğidir. aSequence'in öğeleri heap'e kopyalanır ve bir heap oluşturacak şekilde sıralanır."

childIndex, heap'teki bir düğümün indeksidir. Bu yöntem, o düğümü köke kadar gerekli olduğu kadar yukarı kaydırır, böylece köke giden yol heap koşulunu sağlar.

***9.12 Gözden Geçirme Soruları***

1 : State the heap property for a largest on top heap.(En büyük üstte olan bir heap için heap özelliğini belirtin.)

***Cevap:* Heap Özelliği:** Bir largest heap, tam sıralı bir ağaç yapısıdır ve her düğüm, çocuklarından büyük veya ona eşittir. Yani, kök düğümündeki (root node) değer her zaman en büyük değeri taşır ve her ebeveyn düğümünün değeri, çocuk düğümlerinin değerlerinden büyük veya onlara eşittir.

2 : When removing a value from a heap, which value are you likely to remove? Why?(Bir heap'ten bir değer çıkartırken, hangi değeri çıkartmanız muhtemeldir? Neden?)

***Cevap:*** En büyük üstte olan bir heap'ten (largest on top heap) bir değer çıkarırken, **kök düğümündeki (root node) değeri** çıkartmanız olasıdır. Çünkü, heap özelliği gereği, kök düğümü (root node) her zaman en büyük değeri taşır. Bu, heap'in en önemli özelliğidir; kök düğümündeki (root node) değerin en büyük olması, heap'in sıralama düzenini korur. Bu yüzden, heap'ten bir değer çıkarılacaksa, çıkarılacak ilk değer her zaman kök düğümündeki (root node) en büyük değerdir.

3: After removing a value from a heap, what steps do you have to take to ensure you still have a heap?(Bir heap'ten bir değer çıkardıktan sonra, hala bir heap'iniz olduğundan emin olmak için hangi adımları atmanız gerekir?)

Cevap: 1. Son öğeyi kök ile değiştirin.

2. Yeni kök öğesini doğru konumuna yerleştirmek için aşağıya kaydırma (sift-down) işlemi yapın.

Bu adımlar, heap'in sıralama özelliğini korur ve geçerli bir heap yapısını sağlar.