Ağaçlar

6

Günlük hayatımızda bir ağaç gördüğümüzde genellikle kökleri toprakta, yaprakları ise havadadır. Bir ağacın dalları köklerden az ya da çok düzenli bir şekilde yayılır. *Ağaç* kelimesi Bilgisayar Bilimi'nde verilerin organize edilebileceği bir yoldan bahsederken kullanılır. Ağaçlar, Bölüm 4'te bulunan bağlantılı liste organizasyonu ile bazı benzerliklere sahiptir. Bir ağaçta diğer düğümlere bağlantıları olan düğümler vardır. Bağlantılı listede her düğümün listedeki bir sonraki düğüme bir bağlantısı vardır. Bir ağaçta her düğümün diğer düğümlere iki ya da daha fazla bağlantısı olabilir. Ağaç, sıralı bir veri yapısı değildir. Ağaç veri yapılarında kökün en üstte ve yaprakların en altta olması dışında ağaç gibi düzenlenir. Bilgisayar bilimlerindeki bir ağaç, doğada gördüğümüz ağaçlarla karşılaştırıldığında genellikle ters çizilir. Bilgisayar bilimlerinde ağaçların birçok kullanım alanı vardır. Bazen Şekil 6.1'de gösterilen Fibonacci fonksiyonunu incelerken gördüğümüz gibi bir grup fonksiyon çağrısının yapısını gösterirler.

Şekil 6.1, fib(5)'i hesaplamak için fib fonksiyonunun bir çağrı ağacını göstermektedir. Gerçek ağaçlardan farklı olarak bir *kökü* (en üstte) ve en altta *yaprakları* vardır. Bu ağaçtaki düğümler arasında ilişkiler vardır. fib(5) çağrısının bir sol *alt ağacı* ve bir de sağ alt ağacı vardır. fib(4) düğümü fib(5) düğümünün bir *çocuğudur. fib(*4) düğümü, sağındaki fib(3) düğümünün *kardeşidir. Yaprak düğüm,* çocuğu olmayan bir düğümdür. Şekil 6.1'deki yaprak düğümler, *fib* fonksiyonuna yapılan ve fonksiyonun temel durumlarıyla eşleşen çağrıları temsil etmektedir.

Bu bölümde ağaçları ve bir programda ağaç oluşturmanın ya da kullanmanın ne zaman mantıklı olduğunu inceleyeceğiz. Her programın bir ağaç veri yapısına ihtiyacı olmayabilir. Bununla birlikte, ağaçlar birçok program türünde kullanılır. Ağaçlar hakkında bilgi sahibi olmak sadece bir gereklilik değil, aynı zamanda doğru kullanımları bazı program türlerini büyük ölçüde basitleştirebilir.

6.1 Bölüm Hedefleri

Bu bölümde ağaçlar ve ağaçları kullanan bazı algoritmalar tanıtılmaktadır. Bölümün sonunda aşağıdaki sorulara cevap verebiliyor olmalısınız.

- Ağaçlar nasıl inşa edilir?

Ağaçlar, düğümler (node) ve bu düğümleri birbirine bağlayan kenarlar (edge) üzerinden oluşur. Ağaçların inşa edilmesi ve yönetilmesi için birkaç temel yöntem ve veri yapıları vardır. İşte bazı temel ağaç inşa yöntemleri:

Elle Oluşturma: Ağaçlar, düğümleri ve kenarları belirli bir hiyerarşik yapıya göre elle oluşturularak inşa edilebilir. Bu yöntem, küçük ölçekli ağaçlar için uygun olabilir, ancak genellikle büyük ve karmaşık ağaç yapıları için pratik değildir.

www.it-ebooks.info

Dinamik Olarak Oluşturma: Programlama dillerinde ve bazı kütüphanelerde, ağaçlar dinamik olarak oluşturulabilir. Düğümler ve kenarlar, kod tarafından oluşturulup birbirine bağlanarak ağaç yapısı oluşturulur. Bu yöntem, veri yapılarının dinamik olarak büyüdüğü ve değiştiği durumlar için idealdir.

Dizin Ağaçları (Directory Trees): Dizin ağaçları, bilgisayar dosya sistemlerinde sıkça kullanılan bir ağaç türüdür. Burada, her düğüm bir dosya veya dizini temsil eder, kenarlar ise dizinler arasındaki ilişkiyi gösterir. Klasörler altında alt klasörler bulunabilir ve her dosya, son düğümler olarak hizmet eder.

İkili Ağaçlar (Binary Trees): İkili ağaçlar, her düğümün en fazla iki çocuğa sahip olduğu bir ağaç türüdür. Bu ağaç türü, genellikle sıralı verileri depolamak, sıralama algoritmalarını uygulamak veya hızlı arama işlemleri yapmak için kullanılır.

Ağaç Arama Algoritmaları: Ağaçlar, arama algoritmaları için temel veri yapılarıdır. Örneğin, derinlik öncelikli arama (DFS) ve genişlik öncelikli arama (BFS), ağaç yapısını taramak ve belirli bir düğümü bulmak için kullanılır.

Bu yöntemler ve veri yapıları, çeşitli bilgisayar bilimi uygulamalarında ağaçların nasıl inşa edilebileceğini göstermektedir. Hangi yöntemin tercih edileceği, ağacın kullanım senaryosuna, veri yapısının gereksinimlerine ve performans beklentilerine bağlıdır

www.it-ebooks.info

Bir Ağacı Nasıl Geçebiliriz?

Bir ağaç yapısını geçmek (traverse etmek), ağaçtaki düğümleri ziyaret etmek ve bu düğümleri belirli bir sıra veya algoritma doğrultusunda işlemek anlamına gelir. Bilgisayar biliminde, ağaçları geçmek için genellikle iki ana algoritma kullanılır: derinlik öncelikli arama (Depth-First Search - DFS) ve genişlik öncelikli arama (Breadth-First Search - BFS). İşte her iki algoritmanın da ağaç geçişi için kullanımı:

1.Derinlik Öncelikli Arama (DFS):

-DFS algoritması, bir ağaç yapısını derine inerek keşfeder. Bir düğüme geldiğinde, en son keşfedilen dalları (çocukları) keşfetmeye devam eder.

-DFS, özyinelemeli bir algoritma olarak uygulanabilir veya bir yığın (stack) veri yapısı kullanılarak iteratif olarak da gerçekleştirilebilir.

-DFS, pre-order, in-order ve post-order gibi farklı gezinme sıralamaları sağlayabilir. Bu sıralamalar, düğümleri ziyaret etme sırasını belirler.

2.Genişlik Öncelikli Arama (BFS):

-BFS algoritması, bir ağaç yapısını seviye seviye arar. İlk olarak kök düğümünden başlar ve daha sonra bir seviyedeki tüm düğümleri keşfeder.

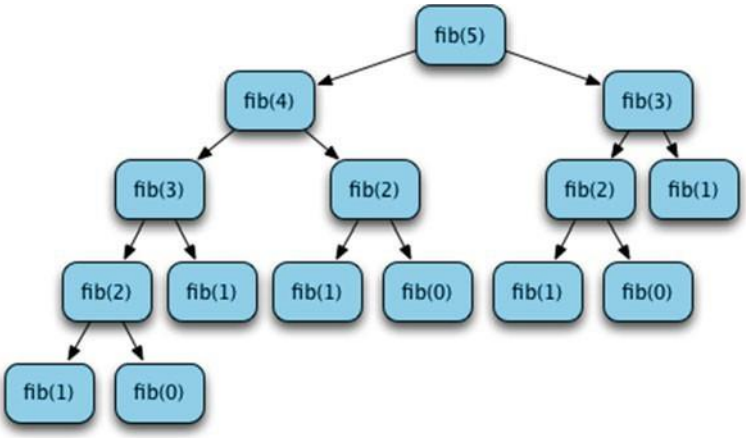
-BFS, genellikle bir kuyruk (queue) veri yapısı kullanılarak gerçekleştirilir. Bu, düğümlerin keşfedilme sırasını korumak için kullanılır.
-BFS, ağacı seviye seviye gezerek en kısa yolu bulmak veya ağaçtaki en kısa yol uzunluğunu bulmak için kullanışlıdır.

Ağaçların geçilmesi, birçok algoritmik sorunun çözümünde önemli bir adımdır. Örneğin, bir ağacın tüm düğümlerini gezerek belirli bir düğümü bulabilir, ağaçtaki düğümlerin toplam sayısını hesaplayabilir veya ağaçtaki en uzun veya en kısa yolun uzunluğunu bulabilirsiniz. Hangi algoritmanın kullanılacağı, problemin gereksinimlerine ve ağacın yapısına bağlıdır.

www.it-ebooks.info

-BFS, ağacı seviye seviye gezerek en kısa yolu bulmak veya ağaçtaki en kısa yol uzunluğunu bulmak için kullanışlıdır.

,



Şekil 6.1 fib(5) Hesaplama için Çağrı Ağacı

- İfadeler ve ağaçlar nasıl ilişkilidir?

İfadeler, genellikle ağaç yapılarında temsil edilir ve çeşitli işlemler için bu ağaç yapıları üzerinde çalışılır. İşte ifadeler ve ağaçlar arasındaki ilişkiyi daha ayrıntılı olarak açıklayan bazı noktalar:

Dilbilgisel Analiz (Parsing): Programlama dillerinde ve matematiksel ifadelerde, ifadeler genellikle sözdizimi ağaçları (syntax trees) olarak adlandırılan ağaç yapılarında temsil edilir. Bu ağaçlar, ifadeyi oluşturan semantik yapıyı yakalamak için kullanılır. Dilbilgisel analiz işlemi, bir ifadeyi ağaç yapısına dönüştürmek ve bu yapı üzerinde çeşitli işlemler gerçekleştirmek için kullanılır.

Derinlik Öncelikli Geçiş (Depth-First Traversal): İfade ağaçları genellikle derinlik öncelikli arama (DFS) gibi ağaç geçişi algoritmalarıyla işlenir. Bu algoritmalar, ağacın düğümlerini ziyaret etmenin ve ifadenin içeriğini işlemenin bir yolunu sağlar.

Değerlendirme ve Yürütme: İfadelerin çoğu, bir programın bir parçası olarak yürütülür. İfade ağaçları, ifadenin değerini hesaplamak ve yürütmek için kullanılır. Özellikle, derinlik öncelikli arama gibi ağaç geçişi algoritmaları, ifadelerin değerlerini hesaplamak için genellikle kullanılır.

Matematiksel İfadelerin Temsili: Matematiksel ifadeler de sıkça ağaç yapısı olarak temsil edilir. Örneğin, bir aritmetik ifadeyi temsil etmek için ağaç yapısı kullanılabilir. Bu ağaçta, operatörler ve operanlar düğümler olarak temsil edilir ve ifadenin yapısı ağaçtaki düğümler arasındaki

www.it-ebooks.info

ilişkilerle belirtilir.

İfadelerin ağaç yapısıyla temsil edilmesi, dilbilgisel analizden ifadelerin derlenmesine ve yürütülmesine kadar birçok alanda kullanılır. Ağaç yapıları, ifadelerin anlaşılması, değerlendirilmesi ve işlenmesi için güçlü bir araçtır ve bu nedenle bilgisayar biliminde ifadeler ve ağaçlar arasındaki ilişki önemlidir.

- İkili arama ağacı nedir?

İkili ağaçlar, birçok farklı uygulamada kullanılır. Örnek olarak:

Arama Ağaçları (Search Trees): İkili ağaçlar, arama algoritmaları için temel bir yapıdır. Özellikle sıralı ikili ağaçlar, arama işlemlerini hızlandırmak için kullanılır. Arama algoritmaları, bir düğümün ağaçta olup olmadığını kontrol etmek veya belirli bir düğümü bulmak için ikili ağaçları etkin bir şekilde kullanır.

Dizin Yapıları (Directory Structures): Bilgisayar dosya sistemlerinde, dizinler ve alt dizinlerin hiyerarşisini temsil etmek için ikili ağaçlar kullanılır. Her düğüm bir dizini veya bir dosyayı temsil ederken, düğümler arasındaki ilişkiler, dosya ve dizinlerin ilişkisini yansıtır.

Huffman Kodlaması: Veri sıkıştırma algoritmalarında, özellikle Huffman kodlaması gibi algoritmalar, metin veya veri akışlarını sıkıştırmak için ikili ağaçlar kullanır. Bu algoritmalar, veri örüntülerini analiz ederek sıkıştırılmış bir temsili oluşturur.

İfade Ağaçları (Expression Trees): Matematiksel ifadeleri temsil etmek ve işlemek için ikili ağaçlar kullanılır. Bu ağaçlar, bir matematiksel ifadenin yapısını ve operatörlerle operandları arasındaki ilişkiyi gösterir.

İkili ağaçlar, veri yapıları ve algoritmaların temel bir parçasıdır ve birçok bilgisayar bilimi probleminin çözümünde kullanılır.

- İkili arama ağacı hangi koşullar altında kullanışlıdır?

Sıralı Verilerin Depolanması: İkili arama ağacı, verilerin sıralı olarak depolanması için etkili bir veri yapısıdır. Özellikle verilerin sıralı bir şekilde eklenmesi veya sıralı bir şekilde alınması gereken durumlarda kullanışlıdır.

Ekleme ve Silme İşlemleri: İkili arama ağacı, veri eklemesi ve silmesi için oldukça hızlıdır. Verileri ekleme veya silme işlemleri, ağacın dengeli bir şekilde tutulmasına dikkat edilirse, genellikle logaritmik bir zaman karmaşıklığına sahiptir.

Arama İşlemleri: Veri koleksiyonundaki belirli bir değeri aramak için ikili arama ağacı kullanmak oldukça etkilidir. Veri koleksiyonu büyüdükçe bile, arama işlemleri genellikle logaritmik zaman karmaşıklığına sahiptir.

Sıralama İşlemleri: İkili arama ağacı, verileri sıralı bir şekilde alma işlemleri için de kullanışlıdır. Ağacın içindeki düğümleri in-order gezerek, verilerin sıralı bir şekilde alınması sağlanabilir.

www.it-ebooks.info

- Derinlik öncelikli arama nedir ve ağaçlar ve arama problemleriyle nasıl ilişkilidir?

Derinlik öncelikli arama (DFS), bir ağaç veya graf yapısında belirli bir hedef düğümü bulma veya bir yol bulma amacıyla kullanılan bir arama algoritmasıdır. Algoritma, bir başlangıç düğümünden başlayarak mümkün olduğunca derine iner ve dalları keşfetmeye devam eder. Bu şekilde, ağaç yapısını derinlemesine keşfeder ve arama problemlerinde belirli bir hedefi bulmak için kullanılır.

- İkili ağaçlar üzerinde yapabileceğimiz üç tür ağaç geçişi nedir?

Pre-order Geçiş (Pre-order Traversal):

Pre-order geçişinde, her düğüm önce kök düğüm olarak işaretlenir, ardından sol alt ağaç gezilir ve son olarak sağ alt ağaç gezilir.

Geçiş sırasında düğümler, önce kök düğümden başlayarak sıralanır, ardından sol alt ağaç gezilir ve son olarak sağ alt ağaç gezilir.

Pre-order geçiş, düğümleri kök-sol-sağ sırasında işlemek için kullanılır.

In-order Geçiş (In-order Traversal):

In-order geçişinde, sol alt ağaç önce gezilir, ardından kök düğüm işlenir ve en son olarak sağ alt ağaç gezilir.

Bu geçiş sırasında düğümler, sıralı bir şekilde işlenir. Örneğin, küçükten büyüğe sıralanmış bir ağaçta düğümler, küçükten büyüğe doğru sıralanarak işlenir.

In-order geçiş, ikili arama ağaçlarının sıralı listesini elde etmek için sıkça kullanılır.

Post-order Geçiş (Post-order Traversal):

Post-order geçişinde, sol alt ağaç gezilir, sonra sağ alt ağaç gezilir ve en sonunda kök düğüm işlenir.

Bu geçiş sırasında düğümler, alt ağaçlar önce gezilerek işlenir ve ardından kök düğüm işlenir.

Post-order geçiş, ağaç yapısının alt düğümleriyle işlemlerin tamamlanmasını bekleyen durumlarda kullanılır.

- Dilbilgisi nedir ve dilbilgisi ile ne yapabiliriz?

Metin Analizi ve İşleme: Bilgisayarlar, metin verilerini işlemek ve analiz etmek için dilbilgisi tekniklerini kullanabilirler. Metin analizi, metin verilerinden anlamlı bilgiler çıkarmak için

www.it-ebooks.info

kullanılır. Örneğin, duygu analizi, metin sınıflandırma, metin özetleme gibi görevler bu kapsamdadır.

Konuşma Tanıma ve Sentezleme: Konuşma tanıma, konuşma sesini yazılı metne dönüştürmek için kullanılırken, konuşma sentezi, yazılı metni konuşma sesine dönüştürmek için kullanılır. Sesli asistanlar ve konuşma tabanlı kullanıcı arayüzleri gibi uygulamalarda sıklıkla kullanılır.

Doğal Dil Anlayışı (NLU): Doğal Dil Anlama, metin verilerindeki dilbilgisi yapılarını anlamak için kullanılır. Bu, metinlerin içeriğini anlamak, metinler arasındaki ilişkileri belirlemek ve kullanıcıların metin tabanlı taleplerini anlamak gibi görevleri içerir.

Çeviri ve Uyarlama: Dilbilgisi teknikleri, metinlerin bir dilden diğerine çevrilmesi için kullanılabilir. Otomatik çeviri sistemleri, bir dilde yazılmış metni başka bir dile çevirmek için dilbilgisi algoritmalarını kullanır.

Dil Modelleme ve Sentetik Dil Üretimi: Dil modelleri, metin verilerindeki dilbilgisi yapılarını öğrenmek ve gelecek metinleri tahmin etmek için kullanılır. Sentetik dil üretimi, bilgisayarların kendi başlarına dil öğrenmelerini ve yeni metinler üretmelerini sağlar.

Bu görevler, dilbilgisinin çeşitli uygulamalarını ve bilgisayarların doğal dil üzerindeki yeteneklerini anlatmaktadır. Dilbilgisi, modern teknolojinin birçok yönünde önemli bir rol oynamakta ve insan-makine etkileşimini önemli ölçüde geliştirmektedir.

> Ağaçları ve Bilgisayar Bilimlerindeki kullanımlarını keşfetmek için okumaya devam edin.

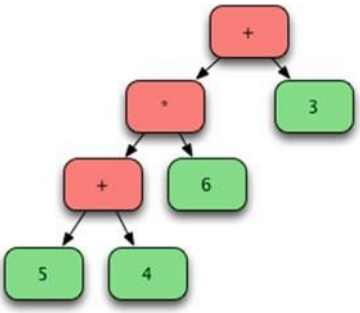
6.2 Soyut Sözdizimi Ağaçları ve İfadeler

Ağaçların Bilgisayar Biliminde birçok uygulaması vardır. Birçok farklı algoritma türünde kullanılırlar. Örneğin, yazdığınız her Python programı Python yorumlayıcısı tarafından çalıştırılmadan önce en azından bir süreliğine bir ağaca dönüştürülür. Dahili olarak, bir Python programı çalıştırılmadan önce *Soyut Sözdizimi Ağacı* adı verilen ve genellikle *AST* olarak kısaltılan ağaç benzeri bir yapıya dönüştürülür. İfadeler için kendi soyut sözdizimi ağaçlarımızı oluşturabiliriz, böylece bir ağacın nasıl değerlendirilebileceğini ve neden bir ağacı değerlendirmek isteyebileceğimizi görebiliriz.

Bölüm 4'te bağlı listeler bir listeyi organize etmenin bir yolu olarak sunulmuştu. Ağaçlar da benzer bir yapı kullanılarak saklanabilir. Bir ağaçtaki bir düğümün iki çocuğu varsa, sıradaki bir sonraki düğüme bir bağlantısı olan bağlantılı listenin aksine, bu düğümün çocuklarına iki bağlantısı olacaktır.

(5 + 4) * 6 + 3 ifadesini düşünün. Bu ifade için Şekil 6.2'de gösterildiği gibi soyut bir sözdizimi ağacı oluşturabiliriz. Bu fonksiyon değerlendirilirken gerçekleştirilen son işlem + işlemi olduğundan, + düğümü ağacın kökünde yer alacaktır. İki alt ağacı vardır: + düğümünün solundaki ifade ve + düğümünün sağındaki 3.

www.it-ebooks.info



Şekil 6.2 (5 + 4) * 6 + 3 için AST

Benzer şekilde, Şekil 6.2'de gösterilen ağacı elde etmek için diğer operatörler ve operandlar için düğümler oluşturulabilir.

Bunu bilgisayarda temsil etmek için, her düğüm türü için bir sınıf tanımlayabiliriz. Bir TimesNode, bir PlusNode ve bir NumNode sınıfı tanımlayacağız. Böylece soyut sözdizimi ağacını değerlendirebiliriz, ağaçtaki her düğümün üzerinde tanımlanmış bir eval yöntemi olacaktır. Bölüm 6.2.1'deki kod bu sınıfları, eval yöntemlerini ve Şekil 6.2'deki örnek ağacı oluşturan bir ana işlevi tanımlar.

6.2.1 6.2.1 AST'lerin Oluşturulması

```
6.2.2   1 class TimesNode:
6.2.3   2 def init (self, left, right):
6.2.4   3 self.left = left
6.2.5   4 self.right = right
6.2.6   5
6.2.7   6 def eval(self):
6.2.8   7 return self.left.eval() * self.right.eval()
6.2.9   8
6.2.10  9 class PlusNode:
6.2.11  10 def init (self, left, right):
6.2.12  11 self.left = left
6.2.13  12 self.right = right
6.2.14  13
6.2.15  14 def eval(self):
6.2.16  15 return self.left.eval() + self.right.eval()
6.2.17  16
6.2.18  17 class NumNode:
6.2.19  18 def init (self, num):
6.2.20  19 self.num = num
6.2.21  20
6.2.22  21 def eval(self):
6.2.23  22 return self.num
6.2.24  23
6.2.25  24 def main():
6.2.26  25 x = NumNode(5)
6.2.27  26 y = NumNode(4)
6.2.28  27 p = PlusNode(x,y)
6.2.29  28 t = TimesNode(p, NumNode(6))
6.2.30  29 root = PlusNode(t, NumNode(3))
6.2.31  30
```

www.it-ebooks.info

```
6.2.32 31 print(root.eval())
6.2.33  32
6.2.34 33 if name == " main ":
6.2.35 34 main()
```

Bölüm 6.2.1'de ağaç en alttan (yani yapraklardan) köke doğru oluşturulmuştur. Yukarıdaki kod her düğüm için bir *eval* fonksiyonu içerir. Kök düğümde *eval* çağrıldığında, ağaçtaki her düğümde özyinelemeli olarak eval çağrılır ve sonuç, 57, ekrana yazdırılır.

Bir AST oluşturulduktan sonra, böyle bir ağacın değerlendirilmesi, ağacın özyinelemeli bir şekilde çaprazlanmasıyla gerçekleştirilir. Bu örnekte *eval* yöntemleri birlikte özyinelemeli işlevdir. Tüm *eval yöntemleri* birlikte özyinelemeli işlevi oluşturduğundan, *eval yöntemlerinin karşılıklı olarak özyinelemeli* olduğunu söyleriz.

6.3 Önek (prefix) ve sönek (suffix) İfadeleri

Normalde yazdığımız ifadelerin infix formunda olduğu söylenir. Bir *infix ifade,* ikili operatörler operandları arasında olacak şekilde yazılmış bir ifadedir. Ancak ifadeler başka biçimlerde de yazılabilir. İfadeler için başka bir form postfix'tir. Bir *postfix ifadede* ikili operatörler operandlarından sonra yazılır.
(5 + 4)* 6 + 3 infix ifadesi postfix formunda 5 4 + 6 * 3 + şeklinde yazılabilir. Postfix ifadeler yığın ile değerlendirme için çok uygundur. Bir ifadeye geldiğimizde operandının değerini yığına iteriz. Bir operatöre geldiğimizde, operandları yığından alır, işlemi yapar ve sonucu iteriz. İfadeleri bu şekilde değerlendirmek insanlar için biraz pratikle oldukça kolaydır. Hewlett-Packard bu postfix değerlendirme yöntemini kullanan birçok hesap makinesi tasarlamıştır. Aslında, bilgi işlemin ilk yıllarında Hewlett-Packard, ifadeleri aynı şekilde değerlendirmek
için yığın kullanan bir dizi bilgisayar üretti. HP 2000 böyle bir bilgisayardı. Daha yakın zamanlarda, Java Sanal Makinesi veya JVM ve Python sanal makinesi de dahil olmak üzere birçok sanal makine yığın makineleri olarak uygulanmıştır.

Ağaç çaprazlamasına başka bir örnek olarak, bir ifadenin dize gösterimini döndüren bir yöntem yazdığınızı düşünün. Dize, soyut sözdizimi ağacının bir çaprazlamasının sonucu olarak oluşturulur. İfadenin infix versiyonunu temsil eden bir dize elde etmek için AST'nin sıralı bir çaprazlamasını *gerçekleştirirsiniz.* Bir postfix ifade elde etmek için ağacın *postfix çaprazlamasını* yaparsınız. Bölüm 6.3.1 'deki inorder yöntemleri AST'nin inorder çaprazlamasını gerçekleştirir.

www.it-ebooks.info

6.3.1 AST Ağaç Çaprazlama

```
1  class TimesNode:
2      def init (self, left, right):
3          self.left = left
4          self.right = right
5
6           def eval(self):
7                 return self.left.eval() * self.right.eval()
8
9           def inorder(self):
10                return "(" + self.left.inorder()+"*"+self.right.inorder() + ")"
11
12    class PlusNode:
13          def init (self, left, right):
14                self.left = left
15                self.right = right
16
17          def eval(self):
18                return self.left.eval() + self.right.eval()
19
20
21          def inorder(self):
22                return "(" + self.left.inorder()+"+"+self.right.inorder() + ")"
23
24    class NumNode:
25          def init (self, num):
26                self.num = num
27
28          def eval(self):
29                return self.num
30
31          def inorder(self):
32                return str(self.num)
```

Bölüm 6.3.1'*deki inorder* metotları inorder traversal sağlar çünkü her bir ikili operatör iki operand *arasında* dizeye eklenir. Ağacın postorder traversalini yapmak için, iki operandın postorder traversinden *sonra* her bir ikili operatörü dizeye ekleyecek bir *postorder* metodu yazmamız gerekir. Postorder traversal'ın yazılma şekli nedeniyle, postfix ifadelerinde parantezlere asla ihtiyaç duyulmadığını unutmayın.

Ön sıralı çaprazlama olarak adlandırılan bir başka *çaprazlama daha* mümkündür. Ön sıralama geçişinde, her ikili işleç iki işleneninden önce dizeye eklenir. İnfix göz önüne alındığında
(5 + 4) * 6 + 3 ifadesinin önek eşdeğeri + * + 5 4 6 3 şeklindedir. Yine, bir önek ifadesinin yazılma şekli nedeniyle, önek ifadesinde parantezlere asla ihtiyaç

www.it-ebooks.info

duyulmaz
ifadeler.

www.it-ebooks.info

6.4 Önek İfadelerini Ayrıştırma

Soyut sözdizimi ağaçları neredeyse hiçbir zaman elle oluşturulmaz. Genellikle bir *yorumlayıcı* veya *derleyici* tarafından otomatik olarak oluşturulurlar. Bir Python programı çalıştırıldığında Python yorumlayıcısı onu tarar ve programın soyut sözdizimi ağacını oluşturur. Python yorumlayıcısının bu kısmına ayrıştırıcı denir. *Ayrıştırıcı,* bir dosyayı okuyan ve otomatik olarak ifadenin (yani bir kaynak programın) soyut sözdizimi ağacını oluşturan ve program veya ifade düzgün oluşturulmamışsa bir sözdizimi hatası bildiren bir program veya programın bir parçasıdır. Bunun nasıl gerçekleştirildiğinin tam ayrıntıları bu metnin kapsamı dışındadır. Ancak, önek ifadeleri gibi bazı basit ifadeler için kendimiz bir ayrıştırıcı oluşturmak nispeten kolaydır.

Ortaokulda bir cümlenin *düzgün kurulup kurul*madığını kontrol ederken İngilizce dilbilgisini kullanmamız gerektiğini öğrendik. Gramer, bir dildeki bir cümlenin nasıl bir araya getirilebileceğini belirleyen kurallar bütünüdür. Bilgisayar Biliminde birçok farklı dilimiz vardır ve her dilin kendi grameri vardır. Önek ifadeleri bir dil oluşturur. Bunlara önek ifadelerinin dili diyoruz ve bağlamdan bağımsız gramer olarak adlandırılan kendi gramerlerine sahipler. Önek ifadeleri için bağlamdan bağımsız bir gramer Bölüm 6.4.1'de verilmiştir.

6.4.1 Önek İfade Grameri

$G = (N, T, P, E)$ burada

$N = \{E\}$

$T = \{tanımlayıcı, sayı, +, *\}$

P, üretimler kümesi tarafından tanımlanır

$$E \rightarrow + E\ E \mid * E\ E \mid sayı$$

Bir gramer, *G*, üç kümeden oluşur: *N* ile gösterilen bir terminal olmayan semboller kümesi, *T olarak* adlandırılan bir terminaller veya belirteçler kümesi ve *P* olarak adlandırılan bir üretimler kümesi. Terminal olmayan sembollerden biri gramerin başlangıç sembolü olarak belirlenir. Bu gramer için, *E* özel sembolü gramerin başlangıç sembolü ve tek terminal olmayan sembolüdür. *E* sembolü herhangi bir önek ifadesi anlamına gelir. Bu gramerde üç *üretim* vardır
önek ifadelerinin nasıl oluşturulabileceğine ilişkin kuralları sağlar. Üretimler, herhangi bir önek ifadesinin bir artı işareti ve ardından iki önek ifadesi, bir çarpma sembolü ve ardından
iki önek ifadesi veya sadece bir sayı. Gramer özyinelemelidir, bu nedenle gramerde *E'yi* her gördüğünüzde, başka bir önek ifadesiyle değiştirilebilir. Bu grameri, bir belirteç kuyruğu verildiğinde bir önek ifadesinin soyut sözdizimi ağacını oluşturacak bir fonksiyona dönüştürmek çok kolaydır. Bölüm 6.4.2'deki *E* işlevi gibi, belirteçleri okuyan ve soyut bir sözdizimi ağacı döndüren bir işleve *ayrıştırıcı* denir. Gramer özyinelemeli olduğundan, ayrıştırma işlevi de özyinelemelidir. Önce bir temel durum, ardından da özyinelemeli durumlar vardır. Bölüm 6.4.2'deki kod bu işlevi sağlar.

www.it-ebooks.info

6.4.2 Bir Önek İfade Ayrıştırıcısı

```
1  import queue
2
3  def E(q):
4     if q.isEmpty():
5        raise ValueError("Invalid Prefix Expression")
6
7     token = q.dequeue()
8
9        if token == "+":
10          return PlusNode(E(q),E(q))
11
12       if token == "*":
13          return TimesNode(E(q),E(q))
14
15       return NumNode(float(token))
16
17    def main():
18       x = input("Lütfen bir önek ifadesi
         girin: ")
19
20       lst = x.split()
21       q = queue.Queue()
22
23       for token in lst:
24          q.enqueue(token)
25
26       root = E(q)
27
28       print(root.eval())
29       print(root.inorder())
30
31    if __name__ == " main ":
         main()
32
```

B Bölüm 6.4.2'de q parametresi dosyadan veya dizeden okunan belirteçlerin bir kuyruğudur. Bu fonksiyonu çağırmak için gerekli kod Bölüm 6.4.2'nin ana fonksiyonunda verilmiştir. Ana işlev kullanıcıdan bir dize alır ve dizedeki tüm belirteçleri (belirteçler boşluklarla ayrılmalıdır) bir belirteç kuyruğuna kaydeder. Daha sonra kuyruk E fonksiyonuna aktarılır. Bu fonksiyon yukarıda verilen gramere dayanır. İşlev bir sonraki belirtece bakar ve hangi kuralın uygulanacağına karar verir. E fonksiyonuna

www.it-ebooks.info

yapılan her çağrı soyut bir sözdizimi ağacı döndürür. Ana fonksiyondan E fonksiyonunun çağrılması, düzeltme öncesi ifadenin ayrıştırılması ve ilgili ağacın oluşturulmasıyla sonuçlanır. Bu örnek, Python'un bir programı nasıl okuduğu ve onun için nasıl soyut bir sözdizimi ağacı oluşturduğu hakkında size biraz fikir verir. Bir Python programı bir gramere göre ayrıştırılır ve programdan soyut bir sözdizimi ağacı oluşturulur. Python yorumlayıcısı daha sonra bu ağaç üzerinde gezinerek programı yorumlar.

Bölüm 6.4.2'deki bu ayrıştırıcı yukarıdan aşağıya ayrıştırıcı olarak adlandırılır. Tüm ayrıştırıcılar bu şekilde oluşturulmaz. Bu metinde sunulan önek grameri yukarıdan aşağıya ayrıştırıcı yapısının çalışacağı bir gramerdir. Özellikle, bir gramer için yukarıdan aşağıya bir ayrıştırıcı oluşturacaksak, herhangi bir sol özyinelemeli kurala sahip olamaz. Sol özyinelemeli kurallar Bölüm 6.4.3'te verilen postfix gramerinde ortaya çıkar.

www.it-ebooks.info

6.4.3 Postfix İfade Grameri

G = *(N, T, P,E)* burada

N = {E}

T = *{tanımlayıcı, sayı, +, ∗}*

P, üretimler kümesi tarafından tanımlanır

$$E \rightarrow E\ E + \mid E\ E * \mid sayı$$

Bu gramerde birinci ve ikinci üretimler, bir ifadeden oluşan bir ifadeye, ardından başka bir ifadeye ve ardından bir toplama veya çarpma belirtecine sahiptir. Bu gramer için özyinelemeli bir fonksiyon yazmaya çalışsaydık, temel durum önce gelmezdi. Özyinelemeli durum önce gelirdi ve dolayısıyla fonksiyon doğru yazılamazdı çünkü özyinelemeli bir fonksiyonda temel durum önce gelmelidir. Bu tür bir üretime sol-yinelemeli kural denir. Sol-yinelemeli kurallara sahip gramerler bir ayrıştırıcının yukarıdan aşağıya inşası için uygun değildir. Ayrıştırıcı oluşturmanın bu metnin kapsamı dışında kalan başka yolları da vardır. Derleyici yapımı veya programlama dili uygulaması üzerine bir kitap okuyarak ayrıştırıcı yapımı hakkında daha fazla bilgi edinebilirsiniz.

6.5 İkili Arama Ağaçları

İkili arama ağacı, her bir düğümün en fazla iki çocuğa sahip olduğu bir ağaçtır. Ayrıca, bir düğümün sol alt ağacındaki tüm değerler ağacın kökündeki değerden küçüktür ve bir düğümün sağ alt ağacındaki tüm değerler ağacın kökündeki değerden büyük veya ona eşittir. Son olarak, sol ve sağ alt ağaçlar da ikili arama ağaçları olmalıdır. Bu tanım, tanımı korurken değerlerin ağaca eklenebileceği bir sınıf yazmayı mümkün kılar. Bölüm 6.5.1'deki kod bunu gerçekleştirmektedir.

www.it-ebooks.info

6.5.1 BinarySearchTree Sınıfı

```
1  class BinarySearchTree:
2    # Bu, BinarySearchTree sınıfına dahili olan bir Node sınıfıdır.
3     Class __Node :
4      def_ __init__ (self,val,left=None,right=None):
5        self.val = val
6        self.left = left
7        self.right = right
8
9      def getVal(self):
10       return self.val
11
12     def setVal(self,newval):
13       self.val = newval
14
15     def getLeft(self):
16       return self.left
17
18     def getRight(self):
19       return self.right
20
21     def setLeft(self,newleft):
22       self.left = newleft
23
24     def setRight(self,newright):
25       self.right = newright
26
27     # Bu yöntem küçük bir açıklamayı hak ediyor. Sıralı bir
       çaprazlama yapar
28     # tüm değerleri veren ağacın düğümleri. Bu şekilde şunları
       elde ederiz

29     # değerler artan sırada.
30     def iter (self):
31       if self.left  = None:
32         for elem in self.left:
33           yield elem
34
35       yield self.val
36
37       if self.right != None:
```

www.it-ebooks.info

```
38            for elem in self.right:
39              yield elem
40
41    # Aşağıda BinarySearchTree sınıfının yöntemleri yer almaktadır.
42    def init (self):
43      self.root = None
44
45    def insert(self,val):
46
47          # insert işlevi özyinelemelidir ve bir self parametresi
aktarılmaz. Bu bir
48          # statik fonksiyon (sınıfın bir yöntemi değil) ancak
insert'in içinde gizlidir
49          # işlevini kullanın, böylece sınıfın kullanıcıları bunun
varlığından haberdar olmaz.
50
51          def insert(root,val):
52            if root == None:
53              return BinarySearchTree. Node(val)
54
55            if val < root.getVal():
56              root.setLeft(__insert(root.getLeft(),val))
57            else:
58              root.setRight(__insert(root.getRight(),val))
59
60            return root
61
62      self.root = insert(self.root,val)
63
64    def __iter__(self):
65      if self.root = None:
66        return self.root. __iter__ ()
67      else:
68        return [].__Iter__ ()
69
70  def main():
71    s = input("Enter a list of numbers: ")
72    lst = s.split()
73
74    tree = BinarySearchTree()
75
76    for x in lst:
```

www.it-ebooks.info

```
77      tree.insert(float(x))
78
79    for x in lst:
80      print(x)
81
82  if __name__ == " main ":
83    main()
```

Bölüm 6.5.1'deki program bir değerler listesi ile çalıştırıldığında (bir sıralamaya sahip olmalıdırlar) değerleri artan sırada yazdıracaktır. Örneğin, klavyeden 5 8 2 1 4 9 6 7 girilirse, program aşağıdaki gibi davranır.

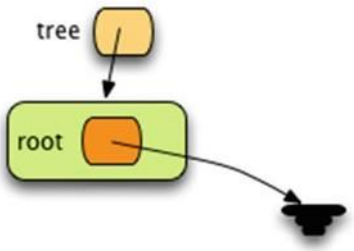
```
   Bir sayı listesi girin: 5 8 2 1 4 9 6 7
1.0
2.0
4.0
5.0
6.0
7.0
8.0
9.0
```

www.it-ebooks.info

Bu örnekten, bir ikili arama ağacının çaprazlandığında sıralanmış bir değer listesi üretebileceği anlaşılmaktadır. Peki nasıl? Bu programın bu girdi ile nasıl davrandığını inceleyelim. Başlangıçta ağaç referansı, Şekil 6.3'te gösterildiği gibi kök işaretçisinin *None'ı gösterdiği* bir BinarySearchTree nesnesine işaret eder.

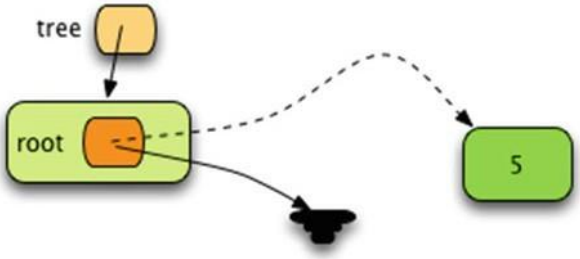
Şekil 6.3'teki ağaca 5 değerini ekleriz. insert metodu çağrılır ve hemen ardından ağacın kökündeki insert fonksiyonu çağrılır. *insert fonksiyonuna* bir ağaç verilir, bu durumda bu ağaç *None* (yani boş bir ağaç) olur ve *insert fonksiyonu* eklenen değerle birlikte yeni bir ağaç döndürür. *Kök* örnek değişkeni, Bölüm 6.5.1'deki kodun 62. satırının sonucu olan Şekil 6.4'te gösterildiği gibi bu yeni ağaca eşit olarak ayarlanır. Aşağıdaki şekillerde kesikli çizgi yeni düğüme işaret etmek üzere atanan yeni referansı göstermektedir. *Insert* fonksiyonu her çağrıldığında yeni bir ağaç döndürülür ve *kök* örnek değişkeni 62. satırda yeniden atanır. Çoğu zaman aynı düğümü gösterecek şekilde yeniden atanır.

Şimdi, eklenecek bir sonraki değer 8'dir. 8'in eklenmesi, 5'i içeren kök düğüme *insert* çağrısı yapar. Bu yapıldığında, özyinelemeli olarak sağ alt ağaca *insert* çağrısı yapılır, ki bu ağaç *None*'dır (ve resimde gösterilmemiştir). Sonuç olarak yeni bir sağ alt ağaç oluşturulur ve 5'i içeren düğümün sağ alt ağaç bağlantısı, Bölüm 6.5.1'deki 58. satırın sonucu olan Şekil 6.5'te gösterildiği gibi onu işaret edecek hale getirilir. Yine kesikli oklar ekleme sırasında atanan yeni referansları göstermektedir. Referansları yeniden atamak hiçbir şeye zarar vermez ve kod çok güzel çalışır. Özyinelemeli eklemede, ağaca yeni bir değer ekledikten sonra 56. ve 58. satırlarda referansı her zaman yeniden atarız. Aynı şekilde, yeni bir değer eklendikten sonra, Bölüm 6.5.1'deki kodun 62. satırında yeni değer eklendikten sonra *kök referansı* yeni ağaca yeniden atanır.

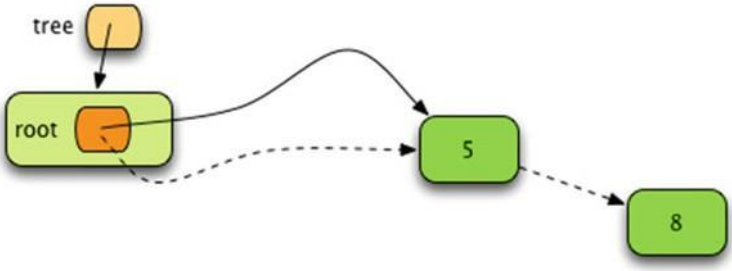


Şekil 6.3 Boş bir BinarySearchTree nesnesi

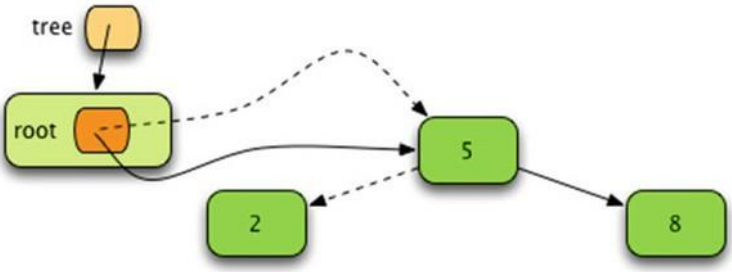
www.it-ebooks.info



Şekil 6.4 5 Eklendikten Sonraki Ağaç



Şekil 6.5 8 Eklendikten Sonraki Ağaç



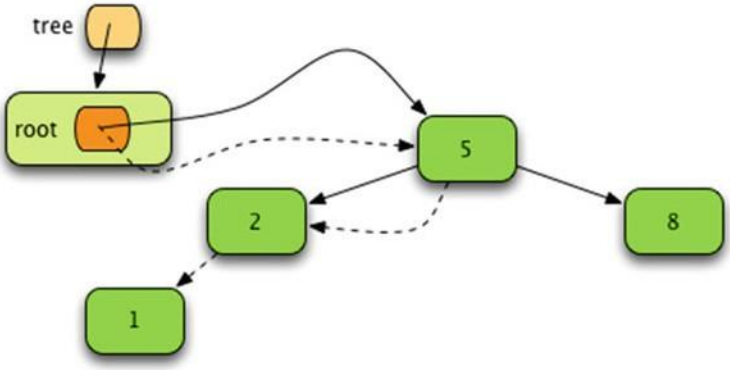
Şekil 6.6 2 Eklendikten Sonraki Ağaç

Daha sonra 2, Şekil 6.6'da gösterildiği gibi ağaca eklenir. İkili arama ağacı özelliğini korumak için 8, 5'in sağında sona ermiştir. 2, 5'in sol alt ağacına eklenir çünkü 2, 5'ten küçüktür.

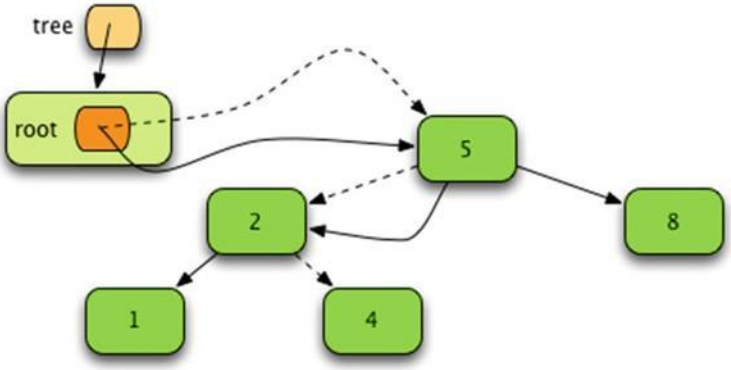
Bir sonraki 1 eklenir ve 5'ten küçük olduğu için 5'i içeren düğümün sol alt ağacına eklenir. Bu alt ağaç 2 içerdiği için 1, 2 içeren düğümün sol alt ağacına yerleştirilir. Bu durum Şekil 6.7'de gösterilmektedir.

Sonraki 4'ün eklenmesi, değerin 5'in soluna ve 2'nin sağına eklendiği anlamına gelir. Bu, Şekil 6.8'de gösterildiği gibi ikili arama ağacı özelliğini korur.

www.it-ebooks.info



Şekil 6.7 1 Eklendikten Sonraki Ağaç



Şekil 6.8 4 Eklendikten Sonraki Ağaç

9'u eklemek için, ağaçtaki tüm düğümlerden daha büyük olduğu için şimdiye kadar eklenen tüm düğümlerin sağına gitmelidir. Bu durum Şekil 6.9'da gösterilmektedir.

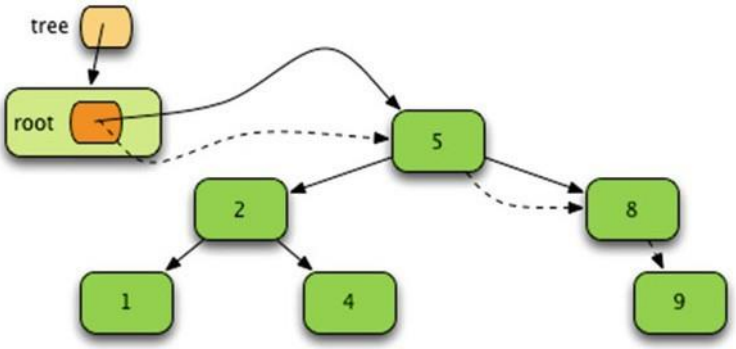
Şekil 6.10'da 6, 5'in sağında ve 8'in solunda yer almaktadır.

Şekil 6.11'de 7'nin gidebileceği tek yer 5'in sağında, 8'in solunda ve 6'nın sağındadır.

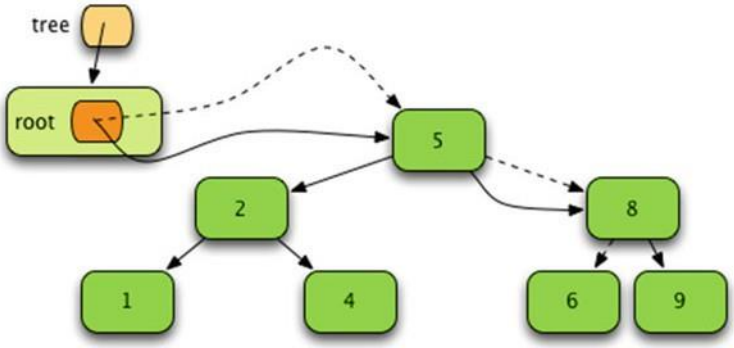
Nihai ağaç Şekil 6.12'de gösterilmektedir. Bu bir ikili arama ağacıdır çünkü alt ağaçlara sahip tüm düğümler sol alt ağaçtaki düğümden daha küçük değerlere ve sağ alt ağaçtaki düğümden daha büyük veya ona eşit değerlere sahiptir ve her iki alt ağaç da ikili arama ağacı özelliğine uygundur.

Bölüm 6.5.1'deki programın son kısmı main fonksiyonundaki *ağaç* üzerinde yineleme yapar. Bu, BinarySearchTree sınıfının *iter* yöntemini çağırır. Bu *iter* yöntemi kökün *Node* nesnesi üzerinde bir yineleyici döndürür. *Node'un iter* yöntemi ilginçtir çünkü ağacın özyinelemeli bir geçişidir. *for elem in self.left* yazıldığında, bu sol alt ağaçtaki *iter* yöntemini çağırır. Sol alt ağaçtaki tüm elemanlar elde edildikten sonra, ağacın kökündeki değer elde edilir,

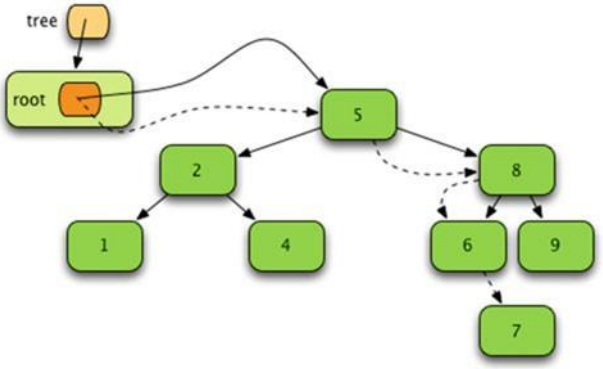
www.it-ebooks.info



Şekil 6.9 9 Eklendikten Sonraki Ağaç

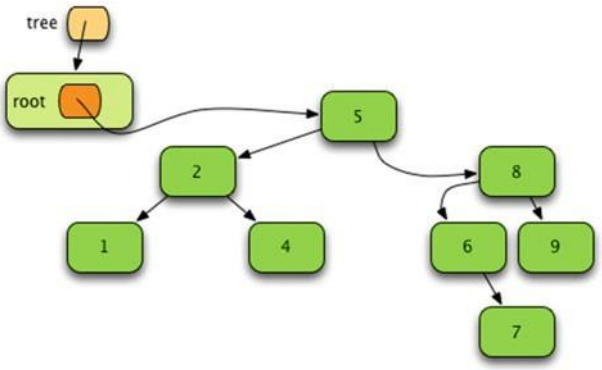


Şekil 6.10 6 Eklendikten Sonraki Ağaç



www.it-ebooks.info

Şekil 6.11 7 Eklendikten Sonraki Ağaç



Şekil 6.12 Nihai BinarySearchTree Nesne İçeriği

sonra sağ alt ağaçtaki değerler *for elem in self.right* yazılarak elde edilir. Bu özyinelemeli işlevin sonucu, ağacın *sıralı* bir şekilde çaprazlanmasıdır.

İkili arama ağaçları bazı akademik ilgi alanlarıdır. Ancak pratikte çok fazla kullanılmazlar. Ortalama bir durumda, bir ikili arama ağacına ekleme yapmak O(log n) zaman alır. Bir ikili arama ağacına *n* öğe eklemek O(n log n) zaman alacaktır. Dolayısıyla, ortalama durumda, sıralı öğeler dizisini sıralamak için bir algoritmamız vardır. Bununla birlikte, bir listeden daha fazla yer kaplar ve quicksort algoritması bir listeyi aynı big-Oh karmaşıklığı ile sıralayabilir. En kötü durumda, ikili arama ağaçları quicksort ile aynı sorundan muzdariptir. Öğeler zaten sıralandığında, hem quicksort hem de ikili arama ağaçları kötü performans gösterir. İkili arama ağacına *n* öğe eklemenin karmaşıklığı en kötü durumda O(n2) olur. Değerler zaten sıralanmışsa ağaç bir çubuğa dönüşür ve esasen bağlantılı bir liste haline gelir.

İkili arama ağaçlarının rastgele erişim listelerinde olmayan birkaç güzel özelliği vardır. Bir listeye ekleme yapmak O(n) zaman alırken, bir ağaca ekleme yapmak ortalama durumda O(log n) zamanda yapılabilir. Bir ikili arama ağacından silme işlemi de ortalama durumda O(log n) zamanda yapılabilir. İkili arama ağacında bir değeri aramak da ortalama durumda O(log n) zamanda yapılabilir. Bir algoritma için çok sayıda ekleme, silme ve arama işlemimiz varsa, ağaç benzeri bir yapı yararlı olabilir. Ancak, ikili arama ağaçları O(log n) karmaşıklığını garanti edemez. Değer ekleme, silme ve arama için O(log n) karmaşıklığını veya daha iyisini garanti edebilen arama ağacı yapılarının uygulamaları olduğu ortaya çıktı. Bunlara örnek olarak, bu metnin ilerleyen bölümlerinde incelenecek olan Splay Ağaçları, AVL-Ağaçları ve B-Ağaçları verilebilir.

www.it-ebooks.info

6.6 Arama Alanları

Bazen birçok farklı durumdan oluşan bir problemimiz olabilir. Problemin *hedef olarak* adlandıracağımız belirli bir durumunu bulmak isteyebiliriz. Sudoku bulmacalarını düşünün. Bir Sudoku bulmacasının, ne kadarına sahip olduğumuzu yansıtan bir durumu vardır

çözüldü. Bulmacanın çözümü olan bir hedef arıyoruz. Bulmacanın bir hücresinde rastgele bir değer deneyebilir ve bu tahmini yaptıktan sonra bulmacayı çözmeye çalışabiliriz. Tahmin, bulmacanın yeni bir durumuna yol açacaktır. Ancak, tahmin yanlışsa geri dönüp tahminimizi geri almamız gerekebilir. Yanlış bir tahmin bizi çıkmaza sokabilir. Bu tahmin etme, bulmacayı bitirmeye çalışma ve kötü tahminleri geri alma sürecine *ilk derinlik araştırması* denir. Tahminler yaparak bir hedefi aramaya, bir problem uzayının ilk derinlik araması denir. Bir çıkmaz sokak bulunduğunda *geri* dönmemiz gerekebilir. Geri izleme, kötü tahminlerin geri alınmasını ve ardından yeni tahminde bulunarak sorunun çözülüp çözülemeyeceğini görmek için bir sonraki tahminin denenmesini içerir. Buradaki açıklama şu sonuca
Götürür.
Bölüm 12.2.1'deki ilk derinlik arama algoritması.

6.6.1 Derinlik Öncelikli Arama Algoritması

```
1  def dfs(current, goal):
2    if current == goal :
3      return [ current ]
4
5    for next in adjacent(current):
6      result = dfs(next)
7      if result != None:
8        return [ current ] + result
9
10   return None
```

Derinlik öncelikli arama algoritması özyinelemeli olarak yazılabilir. Bu kodda, derinlik öncelikli arama algoritması mevcut düğümden hedef düğüme giden yolu döndürür. Geri izleme, for döngüsü uygun bir komşu düğüm bulmadan tamamlanırsa gerçekleşir. Bu durumda, *None* döndürülür ve *dfs*'nin önceki özyinelemeli çağrısı, bu yol üzerindeki hedefi aramak için bir sonraki komşu düğüme gider.

www.it-ebooks.info

Son bölümde Sudoku bulmacalarını çözmek için birçok bulmacada işe yarayan ancak hepsinde işe yaramayan bir algoritma sunulmuştu. Bu durumlarda, problemi mümkün olduğunca azalttıktan *sonra bulmacaya* derinlik öncelikli arama uygulanabilir. Bulmacayı küçültmek için öncelikle son bölümdeki kuralları uygulamak önemlidir çünkü aksi takdirde arama uzayı makul bir sürede aranamayacak kadar büyük olur. Bölüm 6.6.2'deki çözme fonksiyonu, azaltma fonksiyonunun son bölümdeki kuralları bir bulmaca içindeki tüm gruplara uyguladığını varsayarak herhangi bir Sudoku bulmacasını çözecek bir derinlik ilk araması içerir. Bu kodun doğru çalışması için *copy* modülünün içe aktarılması gerekir.

6.6.2 Sudoku Derinlik Öncelikli Arama

```
1  def solutionViable(matrix):
2    # Hiçbir kümenin boş olmadığını kontrol edin
3    for i in range(9):
4      for j in range(9):
5        if len(matrix[i][j]) == 0:
6          return False
8    return True
9
10 def solve(matrix)
11
12     reduce(matrix)
13
14     if not solutionViable(matrix):
15       return None
16
17     if solutionOK(matrix):
18       return matrix
19
20     print("Searching...")
21
22     for i in range(9):
23       for j in range(9):
24         if len(matrix[i][j]) > 1:
25           for k in matrix[i][j]:
26             mcopy = copy.deepcopy(matrix)
27             mcopy[i][j] = set([k])
28
29             result = solve(mcopy)
30
31             if result != None:
```

www.it-ebooks.info

```
32                  return result
33
34      return None
```

www.it-ebooks.info

Bölüm 6.6.2'deki *solve* fonksiyonunda, bulmacayı son bölümdeki kurallarla çözmeye çalışmak için *reduce* çağrılır. *reduce çağrıldıktan* sonra bulmacanın hala çözülebilir olup olmadığı kontrol edilir (yani boş küme yoktur). Eğer değilse, *solve* fonksiyonu *None* döndürür. Arama, matris içindeki her konumu ve konumun alabileceği her olası değeri inceleyerek ilerler. *for k* döngüsü, birden fazla olasılığı olan bir hücre için tüm olası değerleri dener. Eğer *reduce* çağrısı bulmacayı çözerse, *solutionOK fonksiyonu True değerini* döndürür ve *solve* fonksiyonu matrisi döndürür. Aksi takdirde, *ilk derinlik* araması matriste birden fazla seçeneğe sahip bir hücre arayarak devam eder. Fonksiyon, matrisin *mcopy* adında bir kopyasını oluşturur ve *mcopy*'deki o konumdaki değere ilişkin bir tahminde bulunur. Daha sonra *mcopy* üzerinde özyinelemeli olarak *solve* çağrısı yapar.

Çöz fonksiyonu, çözüm bulunamazsa *None değerini*, bir çözüm bulunursa çözülmüş bulmacayı döndürür. Dolayısıyla, *solve* yinelemeli olarak çağrıldığında, *None* döndürülürse, işlev başka bir olası değeri deneyerek aramaya devam eder. Başlangıçta *solve* çağrıldığında
matrisin bir Sudoku bulmacasını temsil eden kümelerin 9 × 9 matrisi olduğu varsayılarak Bölüm 6.6.3'te gösterildiği gibi gerçekleştirilebilir.

6.6.3 Sudoku'nun Çözme İşlevini Çağırma

```
1   print("Begin Solving")
2
3   matrix = solve(matrix)
4
5   if matrix == None:
6     print("No Soution Found!!!")
7  return
```

Yok olmayan bir matris döndürülürse, bulmaca çözülür ve çözüm yazdırılabilir. Bu, hiçbir ağacın oluşturulmadığı, ancak arama uzayının bir ağaç gibi şekillendirildiği ve problem uzayını aramak için ilk derinlik aramasının kullanılabileceği bir örnektir.

6.7 Bölüm Özeti

Ağaç benzeri yapılar Bilgisayar Bilimlerindeki birçok problemde karşımıza çıkmaktadır. Bir ağaç veri tipi bilgileri tutabilir ve hızlı ekleme, silme ve arama sürelerine izin verebilir. İkili arama ağaçları pratikte kullanılmasa da, bunları yöneten ilkeler B-ağaçları, AVL-ağaçları ve Splay Ağaçları gibi birçok gelişmiş veri yapısında kullanılmaktadır. Referansların nesneleri nasıl işaret ettiğini ve bunun ağaç gibi bir veri türü oluşturmak için nasıl kullanılabileceğini anlamak, bilgisayar programcılarının anlaması gereken önemli bir kavramdır.

Birkaç seçenek arasında bir karar vermek başka bir karara yol açtığında arama uzayları genellikle ağaç benzeri olur. Bir arama uzayı bir veri türü değildir, dolayısıyla bu durumda bir ağaç oluşturulmaz. Ancak, aranan uzay ağaç benzeri bir

www.it-ebooks.info

yapıya sahiptir. Bir uzayda derinlemesine ilk arama yapmanın anahtarı, nerede olduğunuzu hatırlamaktır, böylece bir seçim çıkmaza yol açtığında geri dönebilirsiniz. Geri izleme genellikle özyineleme kullanılarak gerçekleştirilir.

Ağaçlarla ilgilenen birçok algoritma doğal olarak özyinelemelidir. İlk derinlik araması, ağaç geçişleri, ayrıştırma ve soyut sözdizimi değerlendirmesi özyinelemeli olarak uygulanabilir. Özyineleme, problemleri çözmek için alet kutunuzda bulunması gereken güçlü bir mekanizmadır.

6.8 İnceleme Soruları

Bu kısa cevaplı, çoktan seçmeli ve doğru/yanlış soruları yanıtlayarak bölüme hakimiyetinizi test edin.

1. **Bilgisayar Bilimlerinde bir ağacın kökü ağacın tepesinde mi yoksa dibinde midir?**

Cevap: Tepesindedir. Bilgisayar bilimlerinde ağaç, veri düğümleri ve düğümler arasındaki ilişkiyi yöneten, ters bir ağacı andıran veri yapısıdır.

2. Bir ağacın kaç kökü olabilir?

Cevap: Ağaçlardaki başlangıç düğümüne kök adı verilir ve başlangıç tek bir noktadan yapıldığından 1 adet kök bulunabilir.

3. Tam ikili ağaç, ağacın her seviyesinde dolu olan bir ağaçtır, yani yapraklar hariç ağacın herhangi bir seviyesinde başka bir düğüme yer yoktur. Üç seviyeli bir tam ikili ağaçta kaç düğüm vardır? Peki ya 4 seviye? Peki ya 5 seviye?

Cevap: 2 üssü n formulüyle herhangi bir seviyedeki düğüm sayısını hesaplayabiliriz.
Örneğin 2. seviyedeki düğüm sayısı 2 üssü 2 olarak 4 bulunur.

4. Tam ikili bir ağaçta, ağaçtaki yaprak sayısı ile ağaçtaki toplam düğüm sayısı arasında nasıl bir ilişki vardır?

Cevap: Ağaçtaki yaprak sayısının 2 katının 1 eksiği bize düğüm sayısını verir.

Yaprak sayısı = n Düğüm sayısı = 2n-1

www.it-ebooks.info

5. Bir ağaç oluştururken, kod yazmak hangisi için daha kolaydır, ağacın aşağıdan yukarıya mı yoksa yukarıdan aşağıya mı inşası?

Cevap: Bu projeye göre değişebilir. Genellikle, daha basit ve doğrudan bir yaklaşım olan "aşağıdan yukarıya" yöntemi, daha karmaşık ve soyut bir problemi ele almak istediğinizde "yukarıdan aşağıya" yöntemine göre daha iyidir.

6. Bir ağaçta bir değer aranırken yanlış bir seçim yapıldığında ve başka bir seçim denenmesi gerektiğinde hangi terim kullanılır?

Cevap : Bilgisayar bilimlerinde bir ağaçta bir değer aranırken yanlış bir seçim yapıldığında ve başka bir seçim denenmesi gerektiğinde kullanılan terim geri izleme'dir (backtracking).
Geri izleme, bir problem çözme algoritmasıdır. Bu algoritma, bir çözüm bulmak için tüm olası yolları denemek için kullanılır. Bir yol tıkanıklığına ulaşıldığında, algoritma geri döner ve başka bir yolu dener.

7. Bir arama alanının ağaç veri türünden farkı nedir?

Cevap: Arama alanları ve ağaç veri türleri, her ikisi de verileri organize etmek için kullanılan yapılardır. Aradaki farklar şunlardır:

Arama alanları, anahtar-değer çiftleri saklayan ve hızlı veri arama için optimize edilmiş yapılardır. Veriler, bir hash fonksiyonu kullanılarak bir diziye yerleştirilir.

Ağaçlar ise hiyerarşik yapılardır ve verileri organize etmek ve ilişkileri göstermek için kullanılırlar. Her düğüm, bir veya daha fazla alt düğüme sahip olabilir.

Özetle: Arama alanları hızlı arama için idealken, ağaçlar veri organizasyonu ve ilişki gösterimi için daha uygundur.

8. Bir ağacın sıralı geçişini yapmak için özyinelemesiz bir algoritma tanımlayın.
İPUCU: Algoritmanızın çalışması için bir yığına ihtiyacı olacaktır.

Cevap:

1- Başlangıçta, ağacın boş olup olmadığını kontrol edin. Boşsa, işlemi sonlandırın.

2- Bir yığın (stack) oluşturun ve bir liste oluşturun, bu liste gezilen düğümleri tutacak.

3- Mevcut düğümü kök düğüm olarak ata.

4- Bir döngü başlatın, bu döngü mevcut düğüm null olana veya yığın boş olana kadar devam edecektir.

5- Mevcut düğüm null değilse:

www.it-ebooks.info

Mevcut düğümü ziyaret edin.

Ziyaret edilen düğümü gezilen düğümler listesine ekleyin.

Eğer mevcut düğümün sağ çocuğu varsa, sağ çocuğunu yığına ekleyin.

Mevcut düğümü sol çocuğuna ata.

6- Eğer yığın boş değilse:

Yığının üstündeki düğümü mevcut düğüm olarak belirleyin.

Yığının üstündeki düğümü yığından çıkarın.

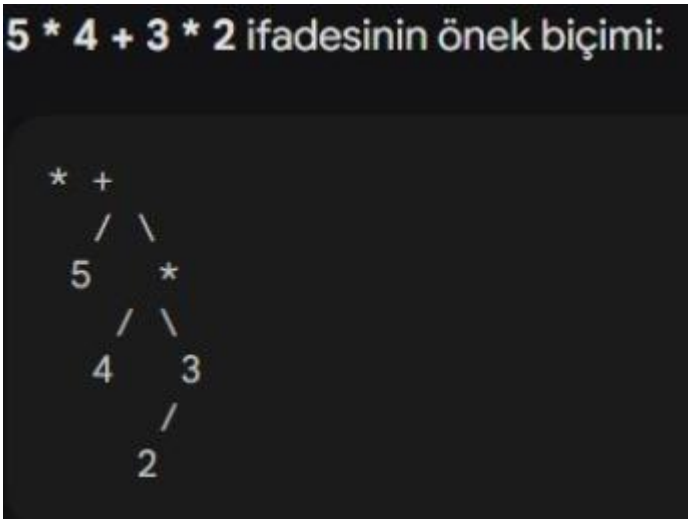
7- Mevcut düğümü güncelledikten sonra, adım 5'e geri dönün ve işlemi tekrarlayın.

8- Döngü, ne mevcut düğüm null olana ne de yığın boş olana kadar devam eder.

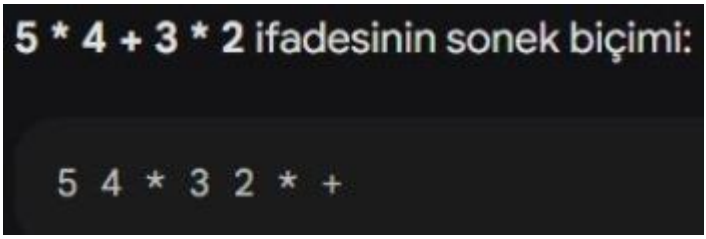
Bu algoritma, ağacın düğümlerini sırayla ziyaret eder ve sağ çocukları işlemek üzere yığına ekler, böylece sıralı geçiş sağlanır.

9. 5 * 4 + 3 * 2'nin önek ve sonek biçimlerini veriniz.

Cevap: Önek ve sonek biçimler, bilgisayarlarda ifadeleri değerlendirmek için kullanılır.



www.it-ebooks.info



6.9 İnceleme Soruları

1. Kullanıcıdan bir önek ifadesi girmesini isteyen bir program yazın. Ardından, program bu ifadenin infix ve postfix formlarını yazdırmalıdır. Son olarak, ifadenin değerlendirilmesinin sonucunu yazdırmalıdır. Programla etkileşim şu şekilde olmalıdır.

```
Lütfen bir önek ifadesi girin: + + * 4 5 6 7
İnfix formu şöyledir: (((4 * 5) + 6) + 7)
Postfix formu: 4 5 * 6 + 7 + Sonuç
şudur: 33
```

CEVAP:

İnfix İfadeye Göre Ağaç Oluşturma Kodu (Python)

Python

```python
class Node:
    def __init__(self, value, operator=None):
        self.value = value
        self.operator = operator
        self.left = None
        self.right = None

def build_tree(expression):
    tokens = expression.split()
    stack = []
    for token in tokens:
        if token.isdigit():
            stack.append(Node(int(token)))
        else:
            node = Node(token, operator=token)
            node.right = stack.pop()
            node.left = stack.pop()
            stack.append(node)
    return stack.pop()

expression = "5 * 4 + 3 * 2"
root = build_tree(expression)

def print_tree(root):
    if root is None:
        return
    print(root.value)
    print_tree(root.left)
    print_tree(root.right)

print_tree(root)
```

www.it-ebooks.info

Kod Açıklaması:

Node sınıfı: Bir düğümün değerini ve operatörünü (varsa) saklayan bir sınıf tanımlar.

build_tree fonksiyonu: İnfix ifadeyi alır ve bir ağaç oluşturur.

İfadeyi tokenlara (operatörler ve operandlar) ayırır.

Operatörler için bir yığın ve operandlar için bir ağaç oluşturur.

Her operatör için, yığından iki operand alır ve operatörü kökü olarak kullanarak yeni bir düğüm oluşturur.

Yeni düğümü ağaca ekler.

Yığın boşalana kadar işlemi tekrarlar.

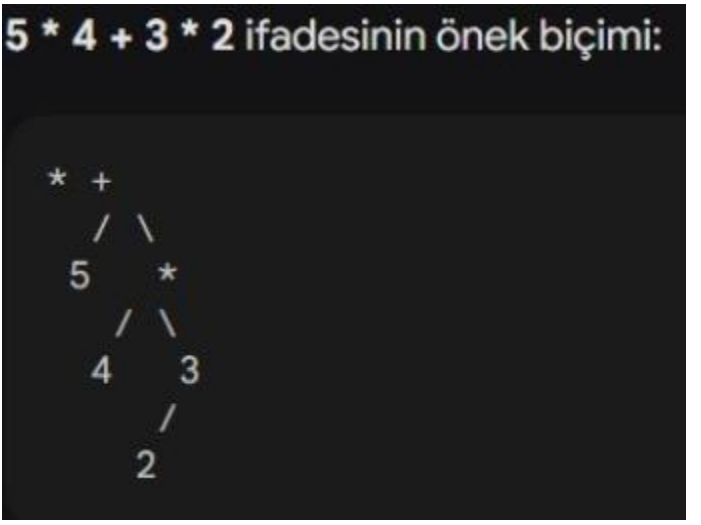
www.it-ebooks.info

print_tree fonksiyonu: Ağacı yazdırır.

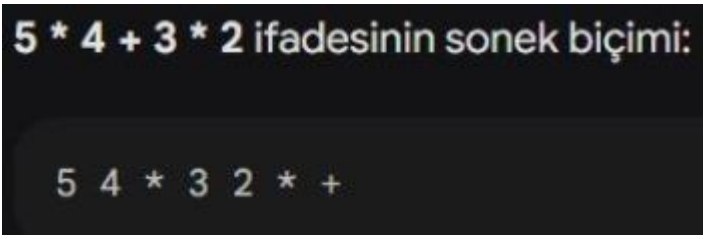
9. 5 * 4 + 3 * 2'nin önek ve sonek biçimlerini veriniz.

Cevap: Önek ve sonek biçimler, bilgisayarlarda ifadeleri değerlendirmek için kullanılır.

Önek biçiminde, operatörler operandlardan önce gelir.



Sonek biçiminde, operatörler operandlardan sonra gelir



6.10 Programlama Problemleri

1. Kullanıcıdan bir önek ifadesi girmesini isteyen bir program yazın. Ardından, program bu ifadenin infix ve postfix formlarını yazdırmalıdır. Son olarak,

www.it-ebooks.info

ifadenin değerlendirilmesinin sonucunu yazdırmalıdır. Programla etkileşim şu şekilde olmalıdır.

```
Lütfen bir önek ifadesi girin: + + * 4 5 6 7
İnfix formu şöyledir: (((4 * 5) + 6) + 7)
Postfix formu: 4 5 * 6 + 7 + Sonuç
şudur: 33
```

Önek ifadesi hatalı biçimlendirilmişse, program ifadenin hatalı biçimlendirilmiş olduğunu yazdırmalı ve çıkmalıdır. Bu durumda, ifadenin infix veya postfix biçimlerini yazdırmaya çalışmamalıdır.

CEVAP:

1. Önek İfadeyi Postfix'e Dönüştürme:

infix_to_postfix fonksiyonu, önek ifadeyi postfix'e dönüştürmek için kullanılır. Bu fonksiyon, önek ifadeyi tokenlere ayırır ve her tokeni işler. Token bir operatörse, fonksiyon yığından iki operandı alır ve operatörü operandlar arasına yerleştirir. Token bir operandsa, fonksiyon operandı yığına iter. Yığın işlemi bittiğinde, yığındaki son token postfix ifadeyi oluşturur.

2. Postfix İfadeyi Değerlendirme:

postfix_eval fonksiyonu, postfix ifadeyi değerlendirmek için kullanılır. Bu fonksiyon, postfix ifadeyi tokenlere ayırır ve her tokeni işler. Token bir operatörse, fonksiyon yığından iki operandı alır ve operatörü kullanarak operandları hesaplar. Sonucu yığına iter. Token bir operandsa, fonksiyon operandı yığına iter. Yığın işlemi bittiğinde, yığındaki son token ifadenin değerini temsil eder

3. Ana İşlev:

main fonksiyonu programın ana işlevini gerçekleştirir. Bu fonksiyon kullanıcıdan önek ifadeyi alır ve infix_to_postfix fonksiyonunu kullanarak postfix'e dönüştürür. Daha sonra postfix_eval fonksiyonunu kullanarak postfix ifadenin değerini hesaplar. Son olarak, infix, postfix ve değerini yazdırır.

4.Hata İşleme:

Kod, önek ifadenin hatalı biçimlendirilmiş olup olmadığını kontrol eder. İfade hatalıysa, bir hata mesajı yazdırır ve programdan çıkar.

www.it-ebooks.info

```
def infix_to_postfix(expr):
    """
    Önek ifadeyi postfix'e dönüştürür.

    Parametreler:
        expr: Önek ifade.

    Döndürülen değer:
        Postfix ifade.
    """
    stack = []
    for token in expr.split():
        if token.isalpha():
            stack.append(token)
        else:
            operand2 = stack.pop()
            operand1 = stack.pop()
            stack.append(operand1 + token + operand2)
    return stack.pop()

def postfix_eval(expr):
    """
    Postfix ifadeyi değerlendirir.

    Parametreler:
        expr: Postfix ifade.

    Döndürülen değer:
        İfadenin değeri.
    """
    stack = []
    for token in expr.split():
        if token.isalpha():
            stack.append(float(input(f"{token} değerini girin: ")))
        else:
            operand2 = stack.pop()
            operand1 = stack.pop()
            if token == "+":
                result = operand1 + operand2
            elif token == "-":
                result = operand1 - operand2
            elif token == "*":
                result = operand1 * operand2
            elif token == "/":
                result = operand1 / operand2
            stack.append(result)
    return stack.pop()

def main():
    """
    Programın ana işlevi.
    """
    expr = input("Önek ifadeyi girin: ")
    try:
        infix = infix_to_postfix(expr)
        postfix = postfix_eval(infix)
        print(f"Infix: {infix}")
        print(f"Postfix: {postfix}")
        print(f"Değer: {postfix}")
    except ValueError:
        print("Hatalı biçimlendirilmiş ifade!")

if __name__ == "__main__":
    main()
```

www.it-ebooks.info

2. Kullanıcıdan bir sayı listesi okuyan ve kullanıcının ağaçtaki değerleri eklemesine, silmesine ve aramasına izin veren bir program yazın. Program, ikili arama ağacına değer ekleme, arama ve silme işlemlerine olanak tanıyan menü odaklı bir program olmalıdır. Ağaca ekleme aşağıdaki gibi birden fazla eklemeye izin vermelidir.

```
İkili Arama Ağacı Programı


Bir seçim yap.
1. Ağacın içine yerleştirin.
2. Ağaçtan sil.
3. Arama Değeri.
 Seçim
  1 mi?
Eklemek
  5 mi?
Eklemek
  2 mi?
Eklemek
  8 mi?
Eklemek
  6 mi?
Eklemek
  7 mi?
Eklemek
  9 mi?
Eklemek
  4 mi?
Eklemek
  1 mi?
Eklem
  ek
  mi?

Bir seçim yap.
1. Ağacın içine yerleştirin.
2. Ağaçtan sil.
3. Arama değeri
   Seçim mi? 3
   Değer mi? 8
   Evet, 8 ağaçta

   Bir seçim yap.
   1. Ağacın içine yerleştirin.
   2. Ağaçtan sil.
   3.Arama değeri.
   Seçim mi? 2
   Değer mi? 5
   5 ağaçtan silinmiştir.
```

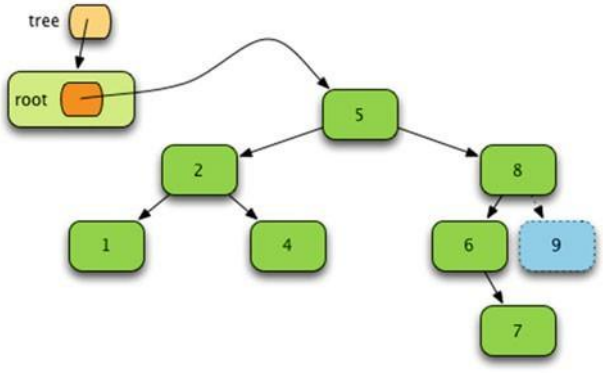
www.it-ebooks.info

```
Bir seçim yap.
1. Ağacın içine yerleştirin.
2. Ağaçtan sil.
3.Arama değeri.
Seçim mi? 2
Değer mi? 3
3 ağaçtan değildi.
```

Problemlerler

Bu programın en zor kısmı ağaçtan silme işlemidir. Bir değeri silmek için özyinelemeli bir fonksiyon yazabilirsiniz. Ağaçtan silme fonksiyonu bazı yönlerden bölümde verilen insert fonksiyonuna benzer. İki fonksiyon yazmak isteyeceksiniz, biri ikili arama ağacında bir değeri silmek için çağrılacak bir yöntem, diğeri ise gizli bir ağaçtan özyinelemeli silme fonksiyonu olacaktır. Özyinelemeli fonksiyona bir ağaç ve silinecek bir değer verilmelidir. Değeri ağaçtan sildikten sonra ağacı döndürmelidir. Özyinelemeli silme işlevi aşağıdaki gibi üç durumda ele alınmalıdır.

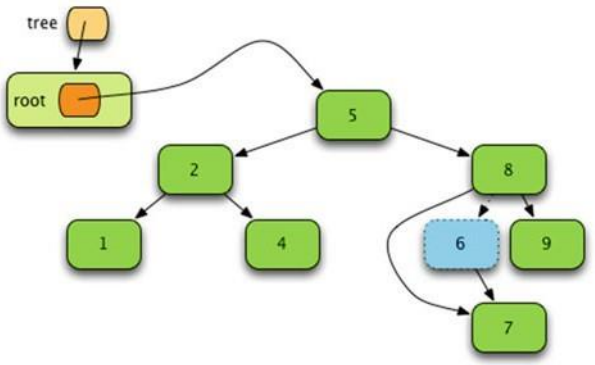
- **Durum 1**. Silinecek değer, çocuğu olmayan bir düğümdedir. Bu durumda, özyinelemeli fonksiyon boş bir ağaç (yani None) döndürebilir çünkü ağaç budur
  değerini sildikten sonra. Şekil 6.12'deki ikili arama ağacından 9 silindiğinde durum böyle olacaktır. Şekil 6.13'te 8'i içeren düğümün sağ alt ağacı artık *Yok'tur ve bu* nedenle 9'u içeren düğüm ağaçtan silinmiştir.
- **Durum 2**. Silinecek değer bir çocuğu olan bir düğümdedir. Bu durumda, özyinelemeli işlev değeri sildikten sonra çocuğu ağaç olarak döndürebilir. Bu
  Şekil 6.13'teki ağaçtan 6'nın silinmesi durumunda durum böyle olacaktır. Bu durumda, silmek için



Şekil 6.13 Silme İşleminden Sonra Ağaç 9

www.it-ebooks.info

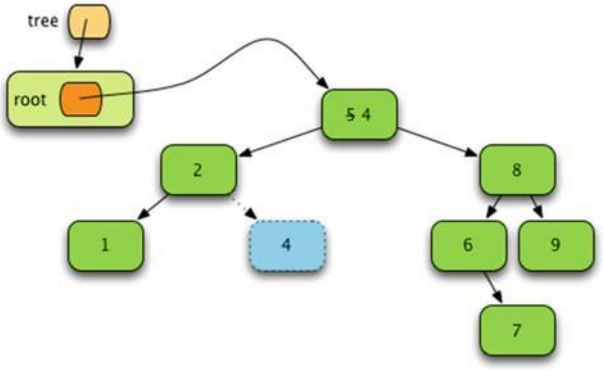
www.it-ebooks.info



Şekil 6.14 6 Silindikten Sonra Ağaç

6'yı içeren düğümü ağaçtan çıkardığınızda, 7'yi içeren düğümün ağacını döndürürsünüz, böylece 8'i içeren düğüme bağlanmış olur. Şekil 6.14'te 6'yı içeren düğüm, 8'i içeren düğümün sol alt ağacının 6'yı içeren düğümün sağ alt ağacını göstermesiyle ortadan kaldırılmıştır.

- **Vaka 3**. Bu, uygulanması en zor durumdur. Silinecek değer iki çocuğu olan bir düğümde olduğunda, düğümü silmek için başka bir fonksiyonuna *getRightMost* adını vererek bir ağacın en sağdaki değerini elde edin. Daha sonra bu fonksiyonu, silinecek düğümün sol alt ağacının en sağ değerini almak için kullanırsınız. Düğümü silmek yerine, düğümün değerini sol alt ağacın en sağdaki değeriyle değiştirirsiniz. Ardından sol alt ağacın en sağındaki değeri sol alt ağaçtan silersiniz. Şekil 6.15'te 5'i içeren düğüm, sol alt ağacın en sağ değeri olan 4'e ayarlanarak 5 ortadan kaldırılır. Ardından 4 sol alt ağaçtan silinir.

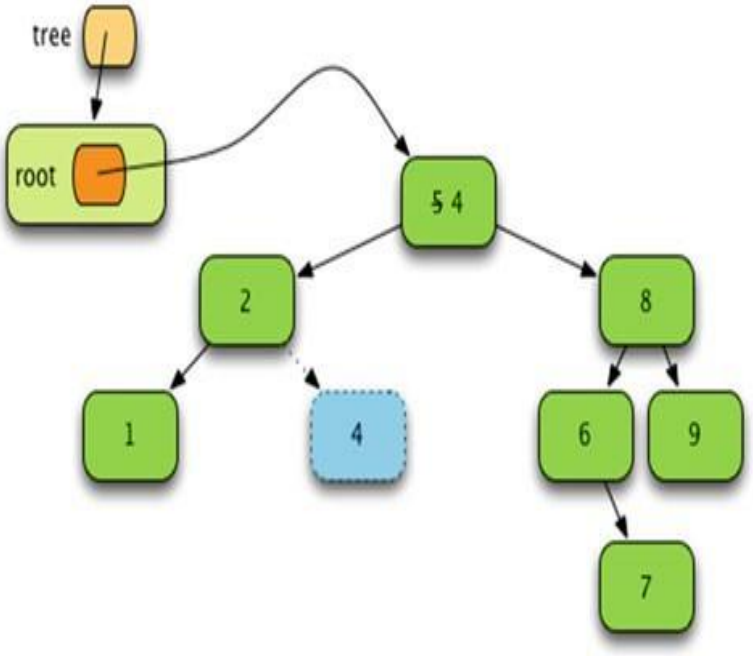


www.it-ebooks.info

Şekil 6.15 5 Silindikten Sonra Ağaç

6'yı içeren düğümü ağaçtan çıkardığınızda, 7'yi içeren düğümün ağacını döndürürsünüz, böylece 8'i içeren düğüme bağlanmış olur. Şekil 6.14'te 6'yı içeren düğüm, 8'i içeren düğümün sol alt ağacının 6'yı içeren düğümün sağ alt ağacını göstermesiyle ortadan kaldırılmıştır.

- **Vaka 3**. Bu, uygulanması en zor durumdur. Silinecek değer iki çocuğu olan bir düğümde olduğunda, düğümü silmek için başka bir fonksiyonuna *getRightMost* adını vererek bir ağacın en sağdaki değerini elde edin. Daha sonra bu fonksiyonu, silinecek düğümün sol alt ağacının en sağ değerini almak için kullanırsınız. Düğümü silmek yerine, düğümün değerini sol alt ağacın en sağdaki değeriyle değiştirirsiniz. Ardından sol alt ağacın en sağındaki değeri sol alt ağaçtan silersiniz. Şekil 6.15'te 5'i içeren düğüm, sol alt ağacın en sağ değeri olan 4'e ayarlanarak 5 ortadan kaldırılır. Ardından 4 sol alt ağaçtan silinir.



Şekil 6.15 5 Silindikten Sonra Ağaç

www.it-ebooks.info

Kod Açıklaması:

Node sınıfı, ikili arama ağacındaki bir düğümü temsil eder.

İnsert fonksiyonu, ağaca yeni bir değer ekler.

Search fonksiyonu, ağacın belirli bir değeri içerip içermediğini kontrol eder.

Delete fonksiyonu, ağacı belirli bir değerden siler.

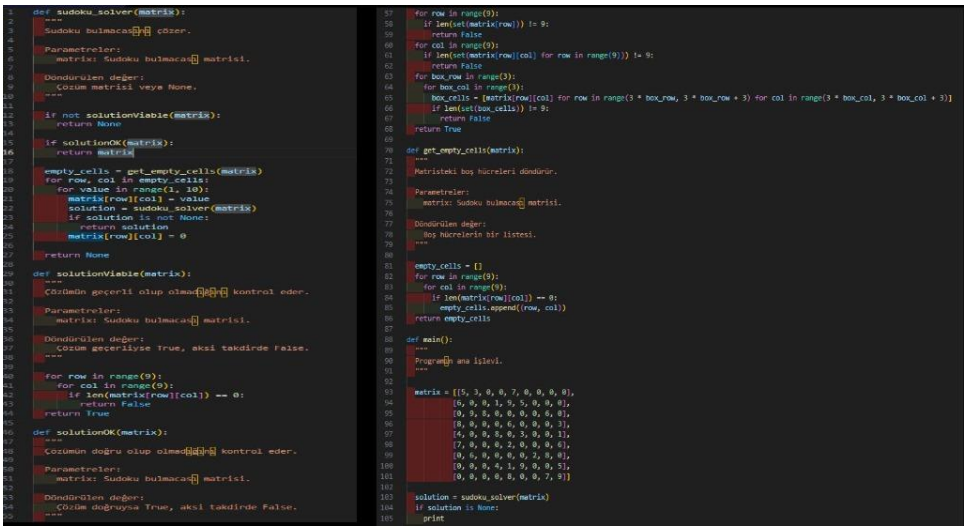
Print_tree fonksiyonu, ağacı yazdırır.

Main fonksiyonu, programın ana işlevini gerçekleştirir.

3. Sudoku programını Bölüm 5'te açıklandığı şekilde tamamlayın ve herhangi bir Sudoku bulmacasını çözebilen bir Sudoku programını tamamlamak için Bölüm 6.6.2'de açıklanan derinlik öncelikli arama ile güçlendirin. Bu bulmacaları neredeyse anında çözmelidir. Eğer bir bulmacayı çözmek uzun zaman alıyorsa, bunun nedeni muhtemelen indirgeme fonksiyonunuzun bulmacayı Bölüm 5'te anlatıldığı gibi indirgememesi olabilir.
   Bu alıştırmayı tamamlamak için iki fonksiyona ihtiyacınız olacak: *solutionOK* fonksiyonu ve *solutionViable* fonksiyonu. *solutionViable* fonksiyonu bu bölümde verilmiştir ve matristeki kümelerden hiçbiri boş değilse *True değerini* döndürür. *solutionOK* fonksiyonu, çözüm geçerli bir çözümse *True* değerini döndürür. Bu çok kolay bir şekilde kontrol edilebilir. Matristeki kümelerden herhangi biri tam olarak 1 eleman içermiyorsa, çözüm uygun değildir ve *False* döndürülmelidir. Bir Sudoku bulmacasındaki herhangi bir grubun birleşimi 9 eleman içermiyorsa, çözüm uygun değildir ve *False* döndürülmelidir. Aksi takdirde, çözüm tamamdır ve *True* döndürülmelidir.
   Bu programı tamamladıktan sonra, metnin web sitesinden indirilebilen sudoku7.txt veya sudoku8.txt gibi Sudoku problemlerini çözebilmeniz gerekir.

www.it-ebooks.info

Cevap:



```python
def sudoku_solver(matrix):
    """
    Sudoku bulmacasını çözer.

    Parametreler:
        matrix: Sudoku bulmacası matrisi.

    Döndürülen değer:
        Çözüm matrisi veya None.
    """

    if not solutionViable(matrix):
        return None

    if solutionOK(matrix):
        return matrix

    empty_cells = get_empty_cells(matrix)
    for row, col in empty_cells:
        for value in range(1, 10):
            matrix[row][col] = value
            solution = sudoku_solver(matrix)
            if solution is not None:
                return solution
            matrix[row][col] = 0

    return None

def solutionViable(matrix):
    """
    Çözümün geçerli olup olmadığını kontrol eder.

    Parametreler:
        matrix: Sudoku bulmacası matrisi.

    Döndürülen değer:
        Çözüm geçerliyse True, aksi takdirde False.
    """

    for row in range(9):
        for col in range(9):
            if len(matrix[row][col]) == 0:
                return False
    return True

def solutionOK(matrix):
    """
    Çözümün doğru olup olmadığını kontrol eder.

    Parametreler:
        matrix: Sudoku bulmacası matrisi.

    Döndürülen değer:
        Çözüm doğruysa True, aksi takdirde False.
    """

    for row in range(9):
        if len(set(matrix[row])) != 9:
            return False
    for col in range(9):
        if len(set(matrix[row][col] for row in range(9))) != 9:
            return False
    for box_row in range(3):
        for box_col in range(3):
            box_cells = [matrix[row][col] for row in range(3 * box_row, 3 * box_row + 3) for col in range(3 * box_col, 3 * box_col + 3)]
            if len(set(box_cells)) != 9:
                return False
    return True

def get_empty_cells(matrix):
    """
    Matristeki boş hücreleri döndürür.

    Parametreler:
        matrix: Sudoku bulmacası matrisi.

    Döndürülen değer:
        Boş hücrelerin bir listesi.
    """

    empty_cells = []
    for row in range(9):
        for col in range(9):
            if len(matrix[row][col]) == 0:
                empty_cells.append((row, col))
    return empty_cells

def main():
    """
    Programın ana işlevi.
    """

    matrix = [[5, 3, 0, 0, 7, 0, 0, 0, 0],
              [6, 0, 0, 1, 9, 5, 0, 0, 0],
              [0, 9, 8, 0, 0, 0, 0, 6, 0],
              [8, 0, 0, 0, 6, 0, 0, 0, 3],
              [4, 0, 0, 8, 0, 3, 0, 0, 1],
              [7, 0, 0, 0, 2, 0, 0, 0, 6],
              [0, 6, 0, 0, 0, 0, 2, 8, 0],
              [0, 0, 0, 4, 1, 9, 0, 0, 5],
              [0, 0, 0, 0, 8, 0, 0, 7, 9]]

    solution = sudoku_solver(matrix)
    if solution is None:
        print
```

Kodun Çalışması:

main fonksiyonu, başlangıç matrisini tanımlar.

sudoku_solver fonksiyonu, matrisi çözmek için tekrarlı olarak çağrılır.

sudoku_solver fonksiyonu, ilk önce çözümün geçerli olup olmadığını kontrol eder.

Çözüm geçerliyse, fonksiyon her bir boş hücre için 1'den 9'a kadar her sayıyı dener.

Bir sayı denendiğinde, fonksiyon tekrarlı olarak sudoku_solver fonksiyonunu çağırır.

Bir çözüm bulunursa, fonksiyon çözümü döndürür.

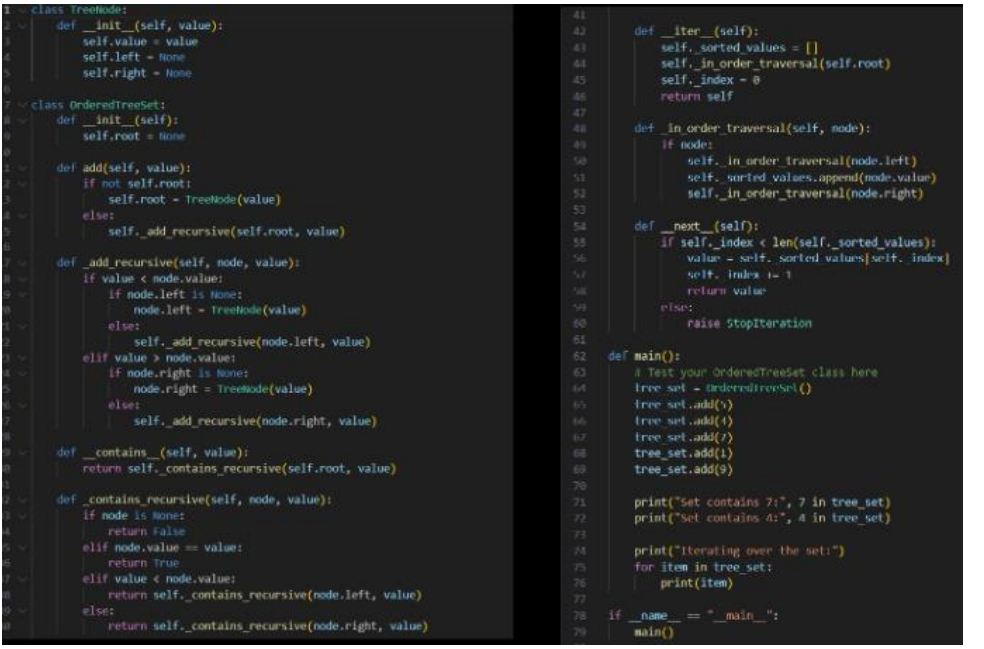
Bir çözüm bulunamazsa, fonksiyon None döndürür.

4. Ortalama O(log n) zamanda öğe eklemek, öğe silmek ve öğeleri aramak için kullanılabilecek bir OrderedTreeSet sınıfı tasarlayın. Kümeyi içermek için bu sınıf üzerinde *in* operatörünü gerçekleyin. Ayrıca kümenin öğelerini artan sırada döndüren bir yineleyici uygulayın. Bu kümenin tasarımı, lt operatörünü gerçekledikleri sürece herhangi bir türdeki öğelerin kümeye eklenmesine izin vermelidir. Bu OrderedTreeSet sınıfı orderedtreeset.py adlı bir dosyaya yazılmalıdır. Bu modülün ana işlevi, OrderedTreeSet sınıfınız için kodunuzu

www.it-ebooks.info

kapsamlı bir şekilde test eden bir test programından oluşmalıdır. *Ana fonksiyon*, modülün içe aktarılması veya kendisinin çalıştırılması arasında ayrım yapan standart *if* deyimi kullanılarak çağrılmalıdır.

CEVAP:



```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

class OrderedTreeSet:
    def __init__(self):
        self.root = None

    def add(self, value):
        if not self.root:
            self.root = TreeNode(value)
        else:
            self._add_recursive(self.root, value)

    def _add_recursive(self, node, value):
        if value < node.value:
            if node.left is None:
                node.left = TreeNode(value)
            else:
                self._add_recursive(node.left, value)
        elif value > node.value:
            if node.right is None:
                node.right = TreeNode(value)
            else:
                self._add_recursive(node.right, value)

    def __contains__(self, value):
        return self._contains_recursive(self.root, value)

    def _contains_recursive(self, node, value):
        if node is None:
            return False
        elif node.value == value:
            return True
        elif value < node.value:
            return self._contains_recursive(node.left, value)
        else:
            return self._contains_recursive(node.right, value)

    def __iter__(self):
        self._sorted_values = []
        self._in_order_traversal(self.root)
        self._index = 0
        return self

    def _in_order_traversal(self, node):
        if node:
            self._in_order_traversal(node.left)
            self._sorted_values.append(node.value)
            self._in_order_traversal(node.right)

    def __next__(self):
        if self._index < len(self._sorted_values):
            value = self._sorted_values[self._index]
            self._index += 1
            return value
        else:
            raise StopIteration

def main():
    # Test your OrderedTreeSet class here
    tree_set = OrderedTreeSet()
    tree_set.add(5)
    tree_set.add(4)
    tree_set.add(7)
    tree_set.add(1)
    tree_set.add(9)

    print("Set contains 7:", 7 in tree_set)
    print("Set contains 4:", 4 in tree_set)

    print("Iterating over the set:")
    for item in tree_set:
        print(item)

if __name__ == "__main__":
    main()
```

Bu kod, bir 'OrderedTreeSet' sınıfını tanımlar. Bu sınıf, verilen öğeleri sıralı bir şekilde saklamak için bir ağaç kullanır. 'add()' yöntemi, logaritmik zamanda çalışır. '_contains()'yöntemi, bir öğenin kümede olup olmadığını kontrol eder. 'iter_()' yöntemi, kümedeki öğeleri artan sırada döndürmek için kullanılır.

Ayrıca, bir test programı, 'main()' fonksiyonuyla, sınıfın işlevselliğini test eder. Bu test programı, bu dosya doğrudan çalıştırıldığında yürütülür, ancak başka bir dosyaya içe aktarıldığında çalıştırılmaz.

5. Uygulamasında bir OrderedTreeSet sınıfı kullanan bir OrderedTreeMap sınıfı tasarlayın. Bunu doğru bir şekilde düzenlemek için iki modül oluşturmalısınız: bir orderedtreeset.py modülü ve bir orderedtreemap.py modülü. Ordered TreeMap sınıfının uygulamasında, Bölüm 5'te *Hash- Set* ve *HashMap*'in uygulandığı şekilde OrderedTreeSet sınıfını kullanmasını sağlayın. OrderedTreeMap sınıfınızı kapsamlı bir şekilde test etmek için test senaryoları tasarlayın.

CEVAP:

İlk olarak, orderedtreeset.py modülünü oluşturalım:

www.it-ebooks.info

```python
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

class OrderedTreeSet:
    def __init__(self):
        self.root = None

    def add(self, value):
        if not self.root:
            self.root = TreeNode(value)
        else:
            self._add_recursive(self.root, value)

    def _add_recursive(self, node, value):
        if value < node.value:
            if node.left is None:
                node.left = TreeNode(value)
            else:
                self._add_recursive(node.left, value)
        elif value > node.value:
            if node.right is None:
                node.right = TreeNode(value)
            else:
                self._add_recursive(node.right, value)

    def __contains__(self, value):
        return self._contains_recursive(self.root, value)

    def _contains_recursive(self, node, value):
        if node is None:
            return False
        elif node.value == value:
            return True
        elif value < node.value:
            return self._contains_recursive(node.left, value)
        else:
            return self._contains_recursive(node.right, value)
```

```python
    def __iter__(self):
        self._sorted_values = []
        self._in_order_traversal(self.root)
        self._index = 0
        return self

    def _in_order_traversal(self, node):
        if node:
            self._in_order_traversal(node.left)
            self._sorted_values.append(node.value)
            self._in_order_traversal(node.right)

    def __next__(self):
        if self._index < len(self._sorted_values):
            value = self._sorted_values[self._index]
            self._index += 1
            return value
        else:
            raise StopIteration
```

Şimdi de 'orderedtreemap.py' modülünü oluşturalım:

```python
from orderedtreeset import OrderedTreeSet

class OrderedTreeMap:
    def __init__(self):
        self.keys = OrderedTreeSet()
        self.map = {}

    def put(self, key, value):
        self.keys.add(key)
        self.map[key] = value

    def get(self, key):
        return self.map.get(key, None)

    def contains_key(self, key):
        return key in self.keys

    def items(self):
        return [(key, self.map[key]) for key in self.keys]

def test_ordered_tree_map():
    tree_map = OrderedTreeMap()

    tree_map.put(3, "Three")
    tree_map.put(1, "One")
    tree_map.put(5, "Five")
    tree_map.put(2, "Two")

    print("Map contains key 3:", tree_map.contains_key(3))
    print("Map contains key 4:", tree_map.contains_key(4))

    print("Items in the map:")
    for key, value in tree_map.items():
        print(key, ":", value)

if __name__ == "__main__":
    test_ordered_tree_map()
```

Bu kod, 'OrderedTreeMap' sınıfını tanımlar. Bu sınıf, bir anahtar-değer eşlemesi sağlar ve anahtarları sıralı bir şekilde tutar. Anahtarlar, 'OrderedTreeSet' sınıfını kullanılarak saklanır ve anahtar-değer çiftleri bir sözlükte depolanır.

www.it-ebooks.info

'test_ordered_tree_map()' fonksiyonu, 'OrderedTreeMap' sınıfını test eder ve işlevselliğini gösterir.

Her iki modülü de test etmek için, her dosyayı doğrudan çalıştırabilirsiniz. 'orderedtreemap.py' dosyasını çalıştırdığınızda, test senaryoları çalıştırılacaktır.

www.it-ebooks.info