**6.4.2 APrefixExpressionParser**

1 **import** **queue**

2

1. **def** E(q):
2. **if** q.isEmpty():
3. **raise** ValueError("Invalid Prefix Expression")

6

7 token = q.dequeue()

8

6.4 Parsing Prefix Expressions

1. **if** token == "+":
2. **return** PlusNode(E(q),E(q))

11

1. **if** token == "\*":
2. **return** TimesNode(E(q),E(q))

14

15 **return** NumNode(float(token))

16

1. **def** main():
2. x = input("Please enter a prefix expression: ")

19

1. lst = x.split()
2. q = queue.Queue()

22

1. **for** token **in** lst:
2. q.enqueue(token)

25

26 root = E(q)

27

1. **print**(root.eval())
2. **print**(root.inorder())

30

1. **if** \_\_name\_\_ == "\_\_main\_\_":
2. main()

**Bölüm 6.4.2'de**, q parametresi, dosyadan veya bir metin dizisinden okunan belirteçlerin (token'ların) saklandığı bir kuyruğu ifade eder.Bu fonksiyonu çağırmak için gereken kod, **Bölüm 6.4.2'nin ana fonksiyonunda** verilmiştir. Ana fonksiyon, kullanıcıdan bir dize alır ve dizideki tüm belirteçleri (belirteçlerin boşluklarla ayrılması gerekir) bir belirteç kuyruğuna ekler. Daha sonra bu kuyruk, E fonksiyonuna geçirilir.

Bu fonksiyon, yukarıda verilen dil bilgisi kurallarına (gramer) dayanmaktadır. Fonksiyon, sıradaki belirtece bakarak hangi kuralın uygulanacağına karar verir. E fonksiyonuna yapılan her çağrı, soyut bir sözdizim ağacı (AST - Abstract Syntax Tree) döndürür. Ana fonksiyondan E fonksiyonu çağrıldığında, önek ifadeyi ayrıştırır (parse eder) ve ona karşılık gelen sözdizim ağacını oluşturur**.**Bu örnek, Python'un bir programı nasıl okuduğu ve onun için nasıl bir soyut sözdizim ağacı oluşturduğu hakkında küçük bir fikir verir. Bir Python programı, belirli bir dilbilgisine göre ayrıştırılır ve programdan bir soyut sözdizim ağacı oluşturulur. Ardından Python yorumlayıcısı, bu ağacı dolaşarak programı yürütür.

Bu ayrıştırıcı (parser), **Bölüm 6.4.2'de** bir üstten aşağıya (top-down) ayrıştırıcı olarak adlandırılır. Ancak tüm ayrıştırıcılar bu şekilde oluşturulmaz.Bu metinde sunulan önek (prefix) dil bilgisi, üstten aşağıya ayrıştırıcı oluşturmanın mümkün olduğu bir dil bilgisidir. Özellikle, bir dil bilgisi üstten aşağıya bir ayrıştırıcı oluşturmak için sol özyinelemeli (left-recursive) kurallara sahip olmamalıdır.Sol özyinelemeli kurallar, Bölüm 6.4.3'te verilen son ek (postfix) dil bilgisinde bulunmaktadır

**6.4.3 ThePostfixExpressionGrammar**

G=(N ,T,P,E)where

N ={E}

T ={identifier,number,+,∗}

P isdefined by the set of productions

E →EE+|EE∗|number

Bu dil bilgisinde, **i**lk ve ikinci üretim kuralları, bir ifadenin (expression), ardından başka birifadenin, ardından bir toplama (+) veya çarpma (\*) belirtecinin (token) gelmesiyle oluşur.Eğer bu dil bilgisi için özyinelemeli (recursive) bir fonksiyon yazmaya çalışsaydık, temel durum (base case) önce gelmezdi. Bunun yerine, özyinelemeli durum (recursive case) önce gelir ve bu yüzden fonksiyon doğru şekilde yazılamazdı. Çünkü özyinelemeli bir fonksiyonda temel durumun önce gelmesi gerekir.Bu tür üretim kurallarına sol özyinelemeli kural (left-recursive rule) denir. Sol özyinelemeli kurallara sahip dil bilgileri, üstten aşağıya (top-down) ayrıştırıcı (parser) oluşturmak için uygun değildir.Ayrıştırıcılar oluşturmanın başka yolları da vardır, ancak bunlar bu metnin kapsamı dışındadır. Ayrıştırıcı oluşturma hakkında daha fazla bilgi edinmek için derleyici (compiler) yapımı veya programlama dili uygulaması ile ilgili bir kitap inceleyebilirsiniz.

**6.5 Binary SearchTrees**

İkili arama ağacı (binary search tree), her düğümün en fazla iki çocuğa sahip olduğu bir ağaçtır. Ayrıca, bir düğümün sol alt ağacındaki tüm değerler, ağacın kökündeki değerden küçük olmalıdır ve sağ alt ağacındaki tüm değerler, kökteki değerden büyük veya ona eşit olmalıdır.Son olarak, sol ve sağ alt ağaçlar da birer ikili arama ağacı olmalıdır.Bu tanım, ağaca yeni değerler eklerken bu yapıyı koruyacak şekilde bir sınıf yazmayı mümkün kılar. Bölüm **6.5.1'de** verilen kod bu işlemi gerçekleştirmektedir

**6.5.1 TheBinarySearchTreeClass**

**1** **class** **BinarySearchTree**:

2 # This is a Node class that is internal to the BinarySearchTree class.

**3** **class** **\_\_Node**:

**4** **def** \_\_init\_\_(self,val,left=None,right=None):

5 self.val = val

6 self.left = left

1. self.right = right

8

1. **def** getVal(self):

**11** **return** self.val

12

1. **def** setVal(self,newval):

14 self.val = newval

15

1. **def** getLeft(self):

**17** **return** self.left

18

1. **def** getRight(self):

**20** **return** self.right

21

1. **def** setLeft(self,newleft):

23 self.left = newleft

24

1. **def** setRight(self,newright):

26 self.right = newright

27

28 # This method deserves a little explanation. It does an inorder traversal

29 # of the nodes of the tree yielding all the values. In this way, we get

30 # the values in ascending order.

**31** **def** \_\_iter\_\_(self):

**32** **if** self.left != None:

1. **for** elem **in** self.left:

**34** **yield** elem

35

36 **yield** self.val

37

**38** **if** self.right != None:

1. **for** elem **in** self.right:

**40** **yield** elem

41

42 # Below are the methods of the BinarySearchTree class.

**43** **def** \_\_init\_\_(self):

44 self.root = None

45

45 46 **def** insert(self,val):

46

47 # The \_\_insert function is recursive and is not a passed a self parameter. It is a

1. # static function (not a method of the class) but is hidden inside the insert

49 # function so users of the class will not know it exists.

50

**51** **def** \_\_insert(root,val):

**52** **if** root == None:

**53** **return** BinarySearchTree.\_\_Node(val)

54

**55** **if** val < root.getVal():

1. root.setLeft(\_\_insert(root.getLeft(),val))57

**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:

1. **return** self.root.\_\_iter\_\_() 67

**67** **else**:

68 **return** [].\_\_iter\_\_()

69

**70** **def** main():

71 s = input("Enter a list of numbers: ")

1. lst = s.split()

73

74 tree = BinarySearchTree()

75

**76** **for** x **in** lst:

1. tree.insert(float(x))

78

**79** **for** x **in** tree:

1. **print**(x)

81

**82** **if** \_\_name\_\_ == "\_\_main\_\_":

1. main()

Bölüm 6.5.1'deki program, sıralı olması gereken bir değer listesiyle çalıştırıldığında, değerleri artan sırayla yazdıracaktır. Örneğin, klavyeden **"5 8 2 1 4 9 6 7"** girildiğinde, program aşağıdaki şekilde çalışacaktır.

Enter a list of numbers: 5 8 2 1 4 9 6 7

1.0

2.0

4.0

5.0

6.0

7.0

8.0

9.0

Bu örnekten, bir ikili arama ağacının dolaşıldığında sıralı bir değer listesi üretebileceği görülmektedir. Peki nasıl? Bu programın bu girişle nasıl davrandığını inceleyelim. Başlangıçta, ağaç referansı, kök işaretçisinin None'a işaret ettiği bir BinarySearchTree nesnesini gösterir. Bu durum Şekil **6.3'te** gösterilmiştir.

**Şekil 6.3'teki** ağaca 5 eklenir. Insert (ekleme) yöntemi çağrılır ve hemen ağacın köküne \_\_insert fonksiyonu uygulanır.\_\_insert fonksiyonuna bir ağaç verilir, ancak bu durumda ağaç None'dır (yani boş bir ağaçtır). Bunun sonucunda \_\_insert fonksiyonu, eklenen değerle birlikte yeni bir ağaç döndürür.Kök değişkeni, bu yeni ağaca eşitlenir ve Şekil **6.4'te** gösterildiği gibi güncellenir. Bu işlem, Bölüm **6.5.1'**deki kodun 62. satırının bir sonucudur.Aşağıdaki şekillerde, kesik çizgi, yeni bir düğüme işaret eden yeni referansı göstermektedir.Her \_\_insert fonksiyonu çağrıldığında yeni bir ağaç döndürülür ve 62. satırda kök değişkeni yeniden atanır. Çoğu zaman, bu değişken aynı düğüme işaret edecek şekilde kalır.

Şimdi, eklenecek bir sonraki değer 8'dir.8'in eklenmesi, 5 içeren kök düğümde \_\_insert fonksiyonunun çağrılmasına neden olur. Bu işlem yapıldığında, fonksiyon özyinelemeli olarak sağ alt ağaçta (None olan ve şekilde gösterilmeyen) tekrar çağrılır.Sonuç olarak, yeni bir sağ alt ağaç oluşturulur ve 5 içeren düğümün sağ alt bağlantısı, bu yeni düğüme işaret edecek şekilde güncellenir. Bu işlem, Bölüm **6.5.1'**deki kodun 58. satırının bir sonucudur ve Şekil **6.5'**te gösterilmektedir.Kesik oklar, ekleme işlemi sırasında oluşturulan yeni referansları göstermektedir.Referansları tekrar atamanın herhangi bir zararı yoktur ve kod sorunsuz bir şekilde çalışmaktadır.Özyinelemeli \_\_insert fonksiyonunda, yeni bir değer eklendikten sonra, referans her zaman 56. ve 58. satırlarda tekrar atanır. Benzer şekilde, yeni bir değer eklendikten sonra, Bölüm **6.5.1'deki** kodun 62. satırında kök referansı yeniden atanır.

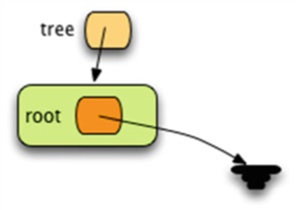


Fig.6.3 Boş bir BinarySearchTree (İkili Arama Ağacı) nesnesi

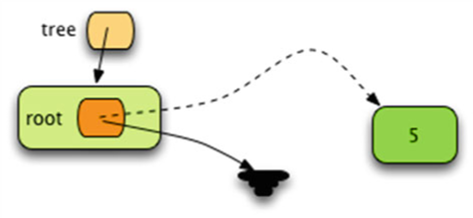


Fig.6.4 5 Ekleme Sonrası Ağaç

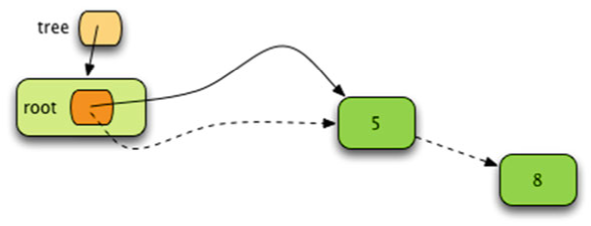


Fig.6.5 8 Ekleme Sonrası Ağaç

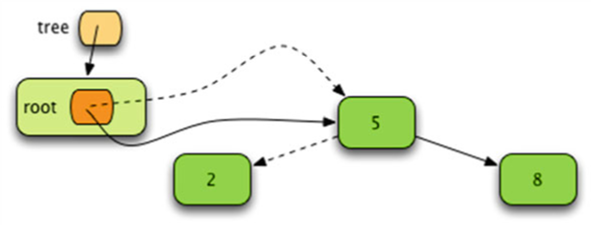


Fig.6.6 2 Ekleme Sonrası Ağaç

Sonraki adımda, 2 ağaçta Fig**.6.6'da** gösterildiği şekilde eklenir. 8, ikili arama ağacı özelliğini korumak için 5'in sağına yerleştirilmiştir. 2 ise 5'ten küçük olduğu için 5'in sol alt ağacına eklenir.

Sonraki adımda, 1 eklenir ve 5'ten küçük olduğu için 5'i içeren düğümün sol alt ağacına eklenir. Bu alt ağaçta 2 bulunduğu için 1, 2'yi içeren düğümün sol alt ağacına eklenir. Bu durum, Fig.**6.7'de** gösterilmektedir.

Sonraki adımda 4 eklenir, bu da değerin 5'in soluna ve 2'nin sağına yerleştirilmesi anlamına gelir. Bu, ikili arama ağacı özelliğini korur ve Fig.**6.8'de** gösterilmektedir.

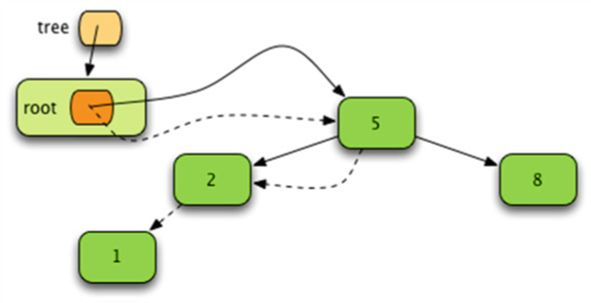


Fig.6.7 1 Ekleme Sonrası Ağaç

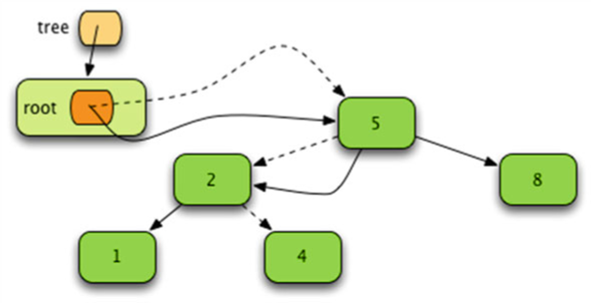


Fig.6.8 4 Ekleme Sonrası Ağaç

9'u eklemek için, ağaçta şimdiye kadar eklenen tüm düğümlerin sağına gitmesi gerekir, çünkü 9, ağaçtaki tüm düğümlerden büyüktür. Bu durum, Fig**.6.9'da** gösterilmektedir.6, 5'in sağına ve 8'in soluna gider, bu Fig**.6.10'da** gösterilmektedir.7'nin gidebileceği tek yer ise 5'in sağına, 8'in soluna ve 6'nın sağına yerleşmesidir, bu Fig.**6.11'de** gösterilmektedir.

Son ağaç, Fig.**6.12'de** gösterilmektedir. Bu bir ikili arama ağacıdır çünkü alt ağaçları olan tüm düğümler, sol alt ağaçtaki düğümden küçük, sağ alt ağaçtaki düğümden ise büyük veya eşit değerlere sahiptir ve her iki alt ağaç da ikili arama ağacı özelliğini korur.

Programın **6.5.1** bölümündeki son kısmı, ana fonksiyonda ağacı yinelemektedir. Bu, BinarySearchTree sınıfının iter metodunu çağırır. Bu iter metodu, kökün \_\_Node nesnesi üzerinde bir yineleyici döndürür. \_\_Node'nun iter metodu ilginçtir çünkü bu, ağacın özyinelemeli bir gezintisidir. "for elem in self.left" yazıldığında, bu sol alt ağacın iter metodunu çağırır. Sol alt ağacın tüm elemanları döndürüldükten sonra, ağacın kökündeki değer döndürülür.