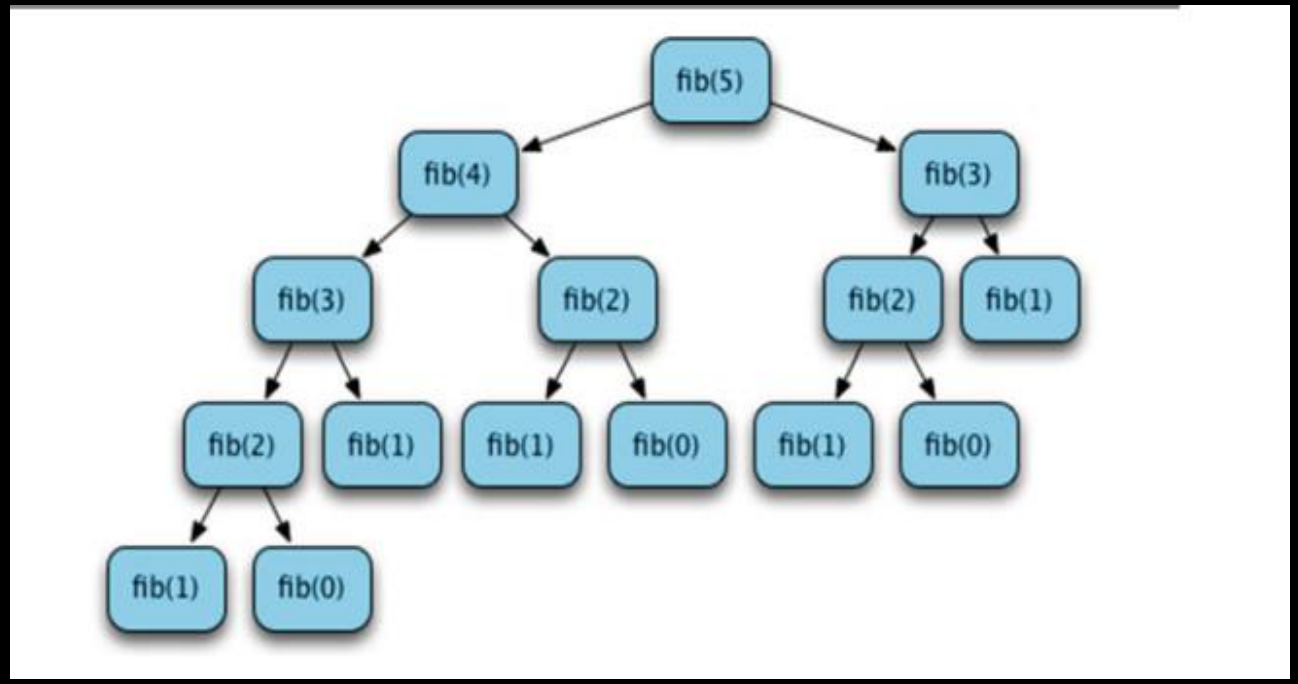
(TREES)AĞAÇLAR

İÇİNDEKİLER (CONTENTS)

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 Biliminde 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.

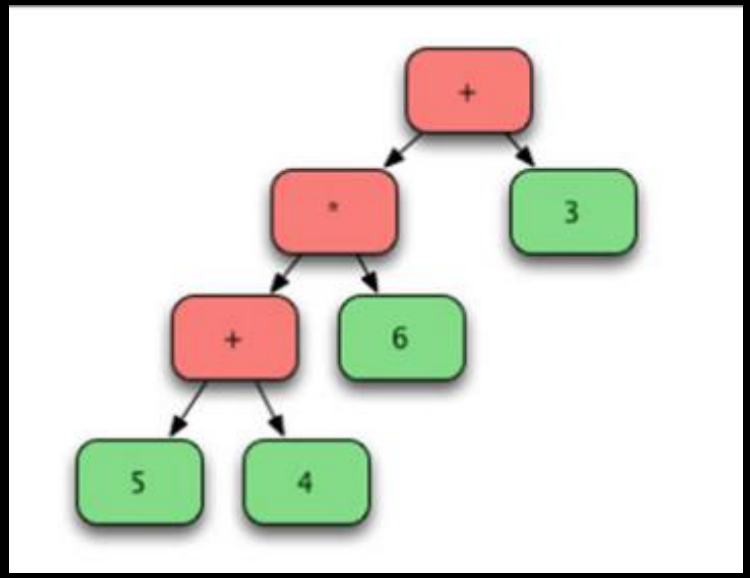


Şekil 6.1 fib(5) Hesaplama için Çağrı Ağacı

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.

6.2 Soyut Sözdizimi Ağaçları ve İfadeler (Abstract Syntax Trees and Expressions)

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.



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 AST'lerin Oluşturulması (Constructing ASTs)

```
class TimesNode:
    def __init__(self, left, right) :
        self.left = left
        self.right = right

    def eval(self) :
        return self.left.eval() * self.right.eval()

class PlusNode:
    def __init__(self, left, right) :
        self.left = left
        self.right = right

    def eval(self) :
        return self.left.eval() + self.right.eval()

class NumNode:
    def __init__(self, num):
        self.num = num

    def eval(self):
        return self.num

def main() :
    x = NumNode(5)
    y = NumNode(4)
    p = PlusNode(x,y)
    t = TimesNode(p, NumNode(6))
    root = PlusNode(t, NumNode(3) )

    print(root.eval())

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

Bölüm 6.2.1'de ağaç en alttan (yani yapraklardan) köke doğru oluşturulmuştur. Yandaki 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 ve Sonek İfadeleri (Prefix and Postfix Expressions)

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.

6.3.1 AST Ağaç Çaprazlama (AST Tree Traversal)

```
class TimesNode:
    def __init__(self, left, right):
        self.left = left
        self.right = right

        def eval(self) :
            return self.left.eval() * self.right.eval()

        def inorder(self) :
            return "(" + self.left.inorder() + " * " + self.right.inorder() + ")"

        class PlusNode:
            def __init__(self, left, right):
                self.left = left
                self.right = right


            def eval(self):
                return self.left.eval() + self.right.eval()

            def inorder(self) :
                return " ( " + self.left.inorder() + " + " + self.right.inorder() + ")"

        class NumNode:
            def __init__(self, num) :
                self.num = num

            def eval(self) :
                return self.num

            def inorder(self) :
                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 duyulmaz ifadeler.

6.4 Önek İfadelerini Ayrıştırma (Parsing Prefix Expressions)

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 kurulmadığını 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 (The Prefix Expression Grammar)

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

N = {E}

T = {tanımlayıcı, sayı, +, *}

P, üretimler kümesi tarafından tanımlanır

E → + E E |* E E | 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.

6.4.2 Bir Önek İfade Ayrıştırıcısı(A Prefix Expression Parser)

```
from queue import Queue

class PlusNode:
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def eval(self):
        return self.left.eval() + self.right.eval()

    def inorder(self):
        return "(" + self.left.inorder() + " + " + self.right.inorder() + ")"

class TimesNode:
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def eval(self):
        return self.left.eval() * self.right.eval()

    def inorder(self):
        return "(" + self.left.inorder() + " * " + self.right.inorder() + ")"

class NumNode:
    def __init__(self, num):
        self.num = num

    def eval(self):
        return self.num

    def inorder(self):
        return str(self.num)

def E(q):
    if q.empty():
        raise ValueError("Invalid Prefix Expression")

    token = q.get()

    if token == "+":
        return PlusNode(E(q), E(q))

    if token == "*":
        return TimesNode(E(q), E(q))

    return NumNode(float(token))

def main():
    x = input("Please enter a prefix expression: ")

    lst = x.split()
    q = Queue()

    for token in lst:
        q.put(token)

    root = E(q)

    print(root.eval())
    print(root.inorder())

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

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 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.

6.4.3 Postfix İfade Grameri (The Postfix Expression Grammar)

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

N = {E}

T = {tanımlayıcı, sayı, +, *}

P, üretimler kümesi tarafından tanımlanır

E → + E E |E E *| 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

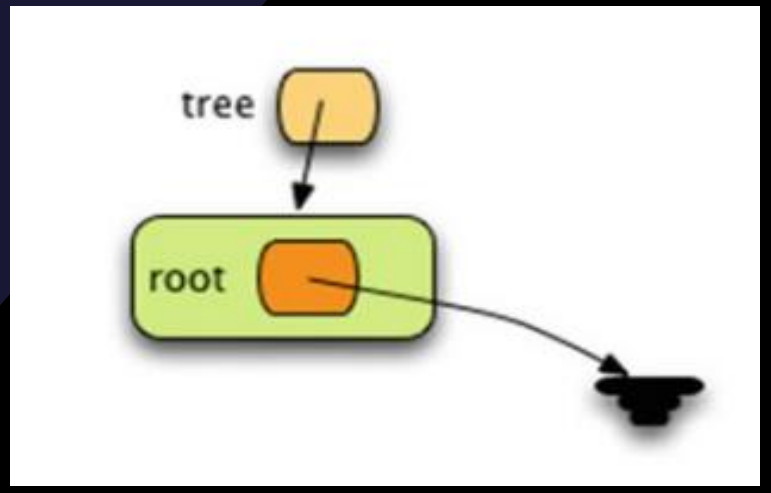
- İkili arama ağacı, her bir düğümün en fazla iki çocuğa sahip olduğu bir ağaçtır. Ayrıca, bir düğümün sol alt ağacındaki tüm değerler ağacın kökündeki değerden küçüktür ve bir düğümün sağ alt ağacındaki tüm değerler ağacın kökündeki değerden büyük veya ona eşittir. Son olarak, sol ve sağ alt ağaçlar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

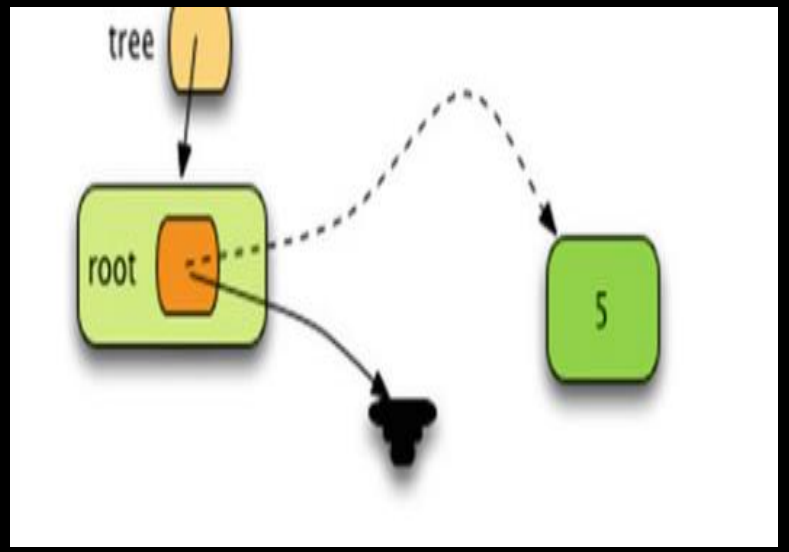
Bir sonraki slaytta bunu gerçekleştiren kodları göreceksiniz.

```python
class BinarySearchTree:
    class __Node:
        def __init__(self, val, left=None, right=None):
            self.val = val
            self.left = left
            self.right = right

        def __iter__(self):
            if self.left is not None:
                for elem in self.left:
                    yield elem

            yield self.val

            if self.right is not None:
                for elem in self.right:
                    yield elem

    def __init__(self):
        self.root = None

    def insert(self, val):
        def __insert(root, val):
            if root is None:
                return BinarySearchTree.__Node(val)

            if val < root.val:
                root.left = __insert(root.left, val)
            else:
                root.right = __insert(root.right, val)

            return root

        self.root = __insert(self.root, val)

    def __iter__(self):
        if self.root is not None:
            return self.root.__iter__()
        else:
            return [].__iter__()

def main():
    s = input("Enter a list of numbers: ")
    lst = s.split()

    tree = BinarySearchTree()

    for x in lst:
        tree.insert(float(x))

    for x in tree:
        print(x)

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

Bu program bir değerler listesi ile çalıştırıldığında (bir sıralamaya sahip olmalıdırlar) değerleri artan sırada yazdıracaktır. Örneğin bu programa değerler olarak 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
```





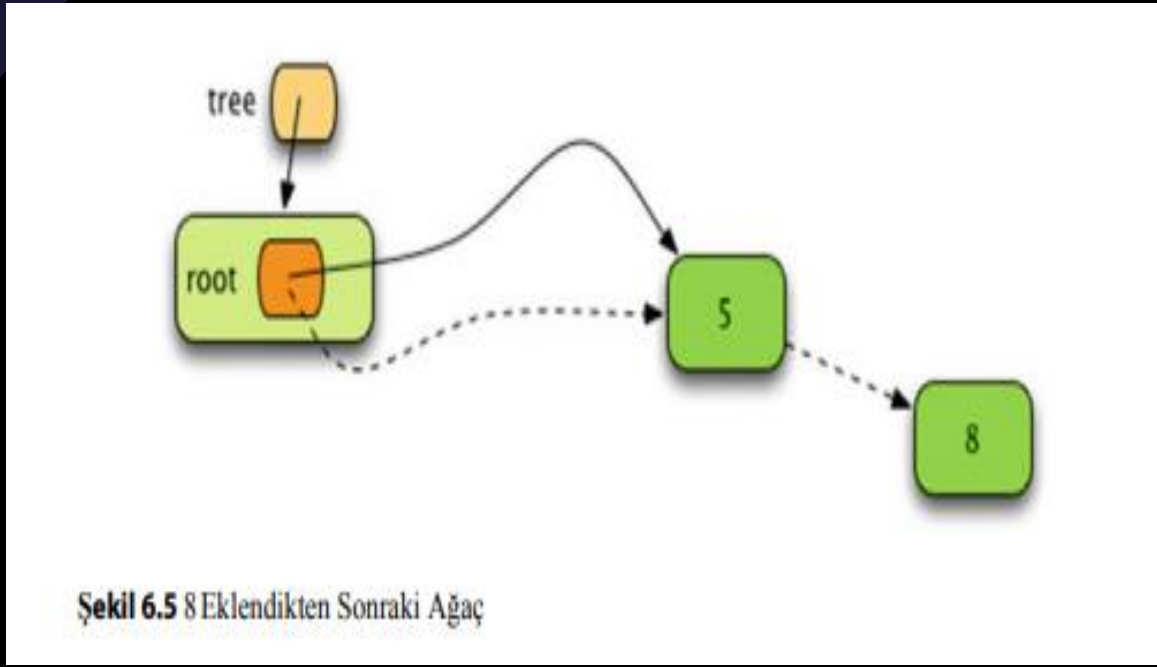
Peki bu program nasıl çalışır? Daha yakından inceleyelim.

Öncelikle sol üstteki resimde görüldüğü üzere boş bir "BinarySearchTree" nesnesi ile başlarız.

Ardından sol alttaki resimde görüldüğü üzere ağaca 5 değerini ekleriz ve insert metodu çağrılır hemen ardından da ağacın kökündeki insert fonksiyonu çağrılır.

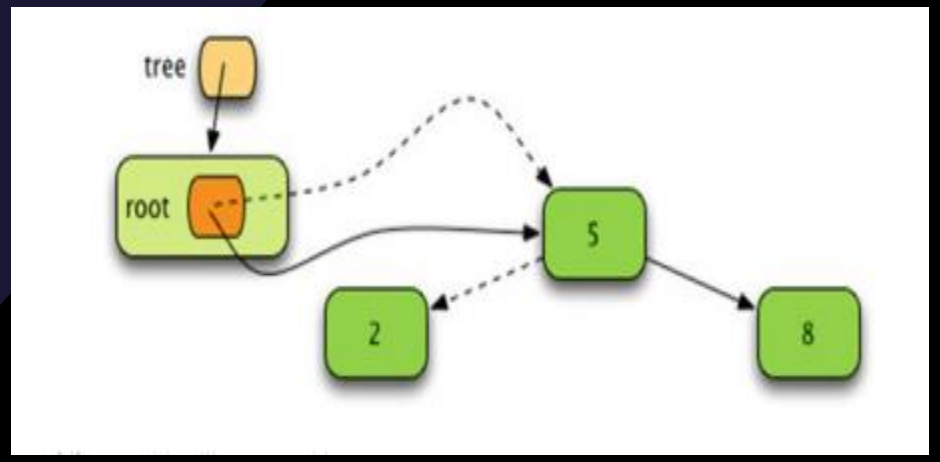
Bu insert fonksiyonuna boş bir ağaç verilir ve ardından eklenecek değerle bu fonksiyon yeni bir ağaca dönüşür.

Kök örnek değişkeni, kodun 62. satırının sonucu olan yeni ağaca eşit olarak ayarlanır. Insert fonksiyonu her çağrıldığında yeni bir ağaç döndürülür ve kök örnek değişkeni 62. satırda yeniden atanır. Çoğu zaman aynı düğümü gösterecek şekilde yeniden atanır.

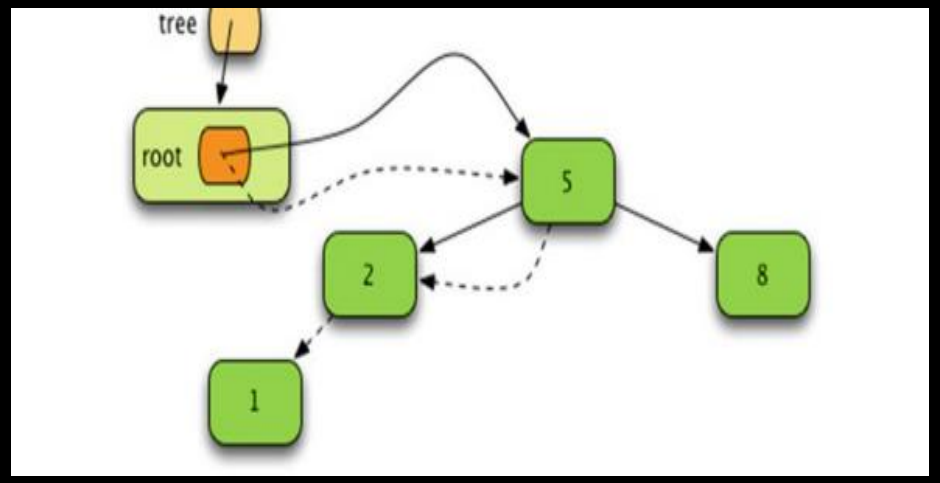


Şekil 6.5 8 Eklendikten Sonraki Ağaç

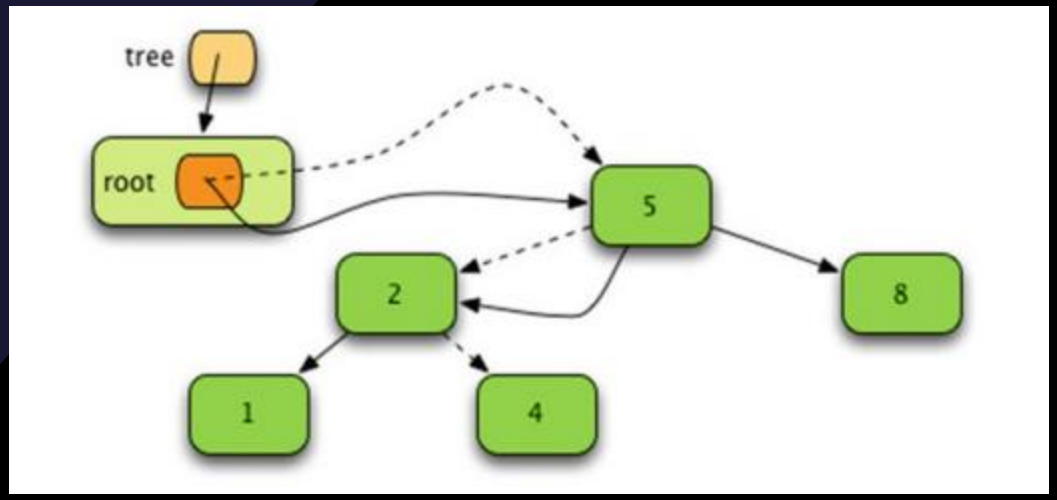
Şimdi, eklenecek bir sonraki değer 8'dir. 8'in eklenmesi, 5'i içeren kök düğüme insert çağrısı yapar. Bu yapıldığında, özyinelemeli olarak sağ alt ağaca insert çağrısı yapılır, ki bu ağaç boştur. Sonuç olarak yeni bir sağ alt ağaç oluşturulur ve 5'i içeren düğümün sağ alt ağaç bağlantısı, kodun 58. satırının sonucu olarak 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, kodun 62. satırında yeni değer eklendikten sonra kök referansı yeni ağaca yeniden atanır.



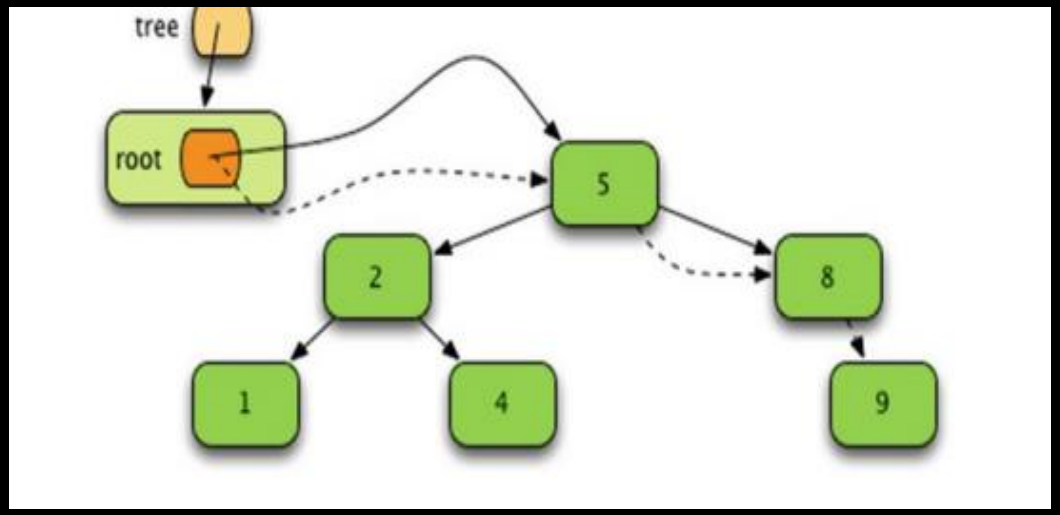
Daha sonra 2, sol üstteki resimde gösterildiği gibi ağaca eklenir. İkili arama ağacı özelliğini korumak için 8, 5'in sağında sona ermiştir. 2, 5'in sol alt ağacına eklenir çünkü 2, 5'ten küçüktür.



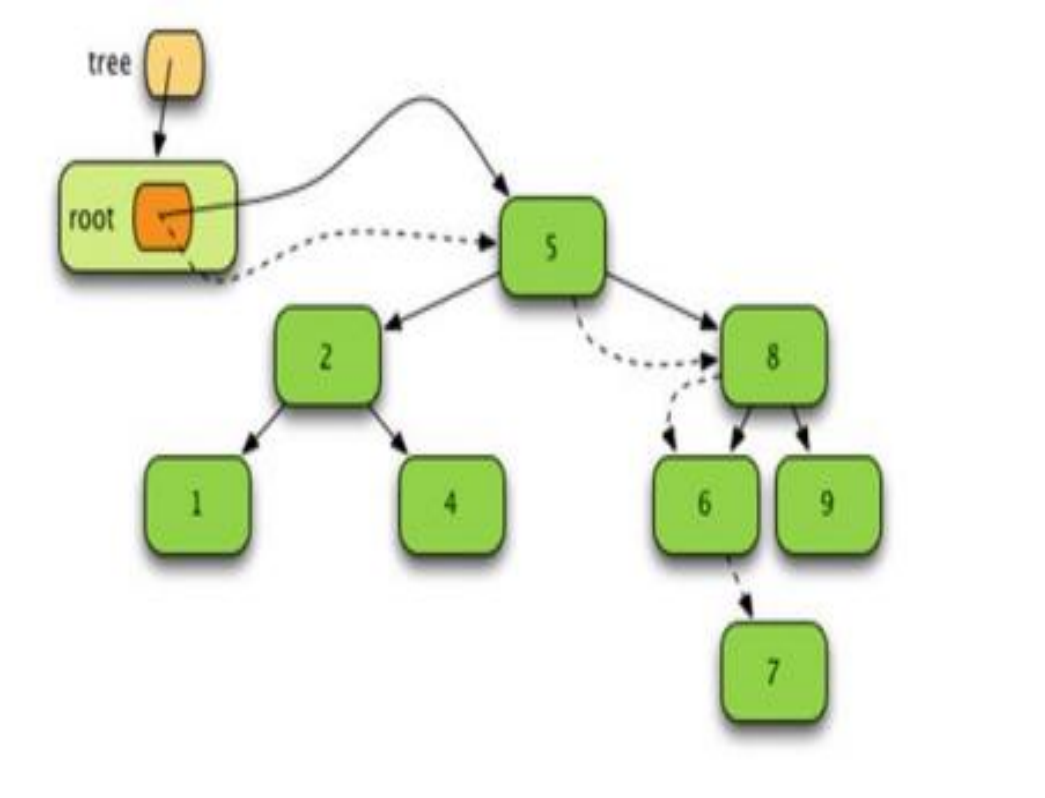
Bir sonraki 1 eklenir ve 5'ten küçük olduğu için 5'i içeren düğümün sol alt ağacına eklenir. Bu alt ağaç 2 içerdiği için 1, 2 içeren düğümün sol alt ağacına yerleştirilir. Bu durum sol alttaki resimde gösterilmektedir.



Ardından 4 değeri sol üstteki şekilde gösterildiği gibi, 5'in soluna ve 2'nin sağına konur. Bu durum ikili arama ağacı özelliğini korur.



9'u eklemek için ise sol alttaki şekilde gösterildiği gibi, ağaçtaki tüm düğümlerden daha büyük olduğu için şimdiye kadar eklenen tüm düğümlerin sağına gitmelidir.



Son olarak 6 ve 7 değerleri de kuralına uygun şekilde şekle yerleştirilir ve şeklin son hali soldaki gibidir.

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.

```
36      def __iter__(self):
37          if self.root is not None:
38              return self.root.__iter__()
39          else:
40              return [].__iter__()
41
42  def main():
43      s = input("Enter a list of numbers: ")
44      lst = s.split()
45
46      tree = BinarySearchTree()
47
48      for x in lst:
49          tree.insert(float(x))
50
51      for x in tree:
52          print(x)
53
54  if __name__ == "__main__":
55      main()
```

- Bölüm 6.5'in başında gösterdiğimiz kodun 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. 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(n**2) 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.

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 kadarını çözdüğümüzü yansıtan bir durumu vardır. 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.

6.6.1 Derinlik Öncelikli Arama Algoritması

```python
def dfs(current, goal, graph, visited=None):
    if visited is None:
        visited = set()

    visited.add(current)

    if current == goal:
        return [current]

    for next_node in graph[current]:
        if next_node not in visited:
            result = dfs(next_node, goal, graph, visited)
            if result is not None:
                return [current] + result

    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.

- 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

```python
def is_valid_move(board, row, col, num):
    # Check if the move is valid in the row
    for i in range(9):
        if board[row][i] == num:
            return False

    # Check if the move is valid in the column
    for i in range(9):
        if board[i][col] == num:
            return False

    # Check if the move is valid in the 3x3 box
    start_row, start_col = 3 * (row // 3), 3 * (col // 3)
    for i in range(start_row, start_row + 3):
        for j in range(start_col, start_col + 3):
            if board[i][j] == num:
                return False

    return True

def solve_sudoku(board):
    for i in range(9):
        for j in range(9):
            if board[i][j] == 0:
                for num in range(1, 10):
                    if is_valid_move(board, i, j, num):
                        board[i][j] = num
                        if solve_sudoku(board):
                            return True
                        board[i][j] = 0  # Backtrack
                return False
    return True
```

1. `solve_sudoku` fonksiyonuna, bir Sudoku tahtası (9×9 boyutunda bir liste) verilir.
2. Fonksiyon, tahtadaki her hücreyi sırayla kontrol eder.
3. Eğer hücredeki değer 0 ise (yani boşsa), bu hücreye bir sayı atamak için tüm 1-9 arası sayıları dener.
4. Her bir sayı denenirken, `is_valid_move` fonksiyonu kullanılarak bu sayının geçerli olup olmadığı kontrol edilir.
5. `is_valid_move` fonksiyonu, verilen bir sayının, aynı satırda, sütunda veya 3×3'lük küçük karelerde tekrarlanıp tekrarlanmadığını kontrol eder.
6. Eğer bir sayı geçerli ise, bu sayı hücreye atanır ve `solve_sudoku` fonksiyonu bu yeni durumu ile çağrılır (rekürsif çağrı).
7. Eğer rekürsif çağrı sonucunda Sudoku bulmacası çözülürse (yani `True` döner), `solve_sudoku` fonksiyonu da `True` döner ve çözüm başarılı olur.
8. Eğer rekürsif çağrı sonucunda bir çözüm bulunamazsa (yani `False` döner), o atanan değer geri alınır (backtrack) ve başka bir sayı denemek için işlem devam eder.
9. Eğer tüm hücreler dolu ise veya çözüm başarısız olursa, fonksiyon sona erer.

6.6.3 Sudoku'nun Çözme İşlevini Çağırma

```python
def solve(matrix):
    pass  # solve fonksiyonunu buraya eklemelisiniz

def solve_and_print(matrix):
    print("Begin Solving")

    matrix = solve(matrix)

    if matrix is None:
        print("No Solution Found!!!")
    else:
        print("Solution Found:")
        for row in matrix:
            print(row)
```

1. `solve_and_print` fonksiyonu, verilen bir Sudoku matrisini çözmek için kullanılır. Bu fonksiyon, Sudoku matrisini parametre olarak alır.
2. İlk olarak, "Begin Solving" mesajını yazdırır. Bu, Sudoku çözümünün başladığını belirtir.
3. Ardından, `solve` fonksiyonunu çağırır ve Sudoku matrisini bu fonksiyona gönderir.
4. `solve` fonksiyonu, Sudoku matrisini çözmek için kullanılacak algoritmayı içerir. Ancak, bu kodda `solve` fonksiyonu tanımlanmamıştır (`pass` ifadesi ile yer tutucu olarak bırakılmıştır). Bu nedenle, Sudoku matrisini çözen algoritmanın buraya eklenmesi gerekmektedir.
5. Eğer `solve` fonksiyonu bir çözüm bulamazsa (yani `None` dönerse), "No Solution Found!!!" mesajını yazdırır.
6. Eğer `solve` fonksiyonu bir çözüm bulursa, çözüm matrisini yazdırır. Her bir satırı sırayla yazdırır ve Sudoku bulmacasının tam çözümünü gösterir.

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 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.

BÖLÜM SORULARI VE ÇÖZÜMLERİ

- 6.8 - İnceleme Soruları (1-10):

- 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.
- 3 seviyeli bir ağacın düğüm sayısını bulmak için sırasıyla 2 üssü 0,1,2 ve 3'ün cevaplarını bulmalıyız (ağacın kökü 0. seviyeyi belirtir), işlemleri sırasıyla yaptığımızda 1,2,4 ve 8 değerlerini buluruz, bunları toplarsak 3 seviyeli bir ağacın 15 düğüme sahip olduğunu bulabiliriz. Bu sayı 4 seviyeli ağaç için 31 ve 5 seviyeli ağaç için 63 olarak çıkar.

- 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

- 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 - Bilgisayar bilimlerinde 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:

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.

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)
```

9 - 5 * 4 + 3 * 2 infix ifadesi için bir ağaç oluşturmak üzere bir kod yazın. Ağacınızda operatörlerin önceliğini takip ettiğinizden emin olun.

Cevap:

- **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.

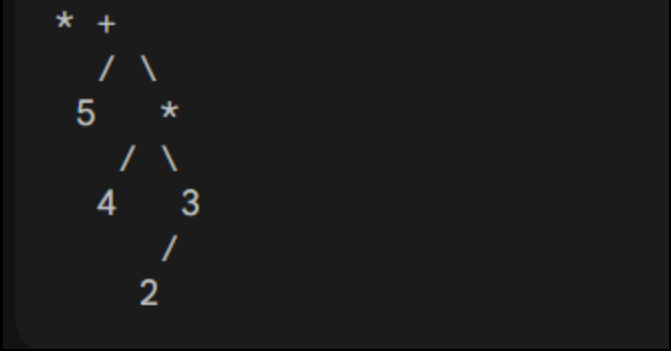
  Yeni düğümü ağaca ekler.

  Yığın boşalana kadar işlemi tekrarlar.
- **print_tree fonksiyonu:** Ağacı yazdırır.

- 10 - 5 * 4 + 3* 2'nin önek ve sonek biçimlerini veriniz.

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

5 * 4 + 3 * 2 ifadesinin önek biçimi:



5 * 4 + 3 * 2 ifadesinin sonek biçimi:

```
5 4 * 3 2 * +
```

- Önek biçiminde, operatörler operandlardan önce gelir.

- Sonek biçiminde, operatörler operandlardan sonra gelir.

```python
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()
```

6.9 - Programlama Problemleri (1-5):

- 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. 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.

- 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.

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

def insert(root, value):
    if root is None:
        return Node(value)
    elif value < root.value:
        root.left = insert(root.left, value)
    else:
        root.right = insert(root.right, value)
    return root

def search(root, value):
    if root is None:
        return False
    elif value == root.value:
        return True
    elif value < root.value:
        return search(root.left, value)
    else:
        return search(root.right, value)

def delete(root, value):
    if root is None:
        return None
    elif value < root.value:
        root.left = delete(root.left, value)
    elif value > root.value:
        root.right = delete(root.right, value)
    else:
        if root.left is None:
            return root.right
        elif root.right is None:
            return root.left
        else:
            min_node = root.right
            while min_node.left is not None:
                min_node = min_node.left
            root.value = min_node.value
            root.right = delete(root.right, min_node.value)
    return root
```

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

def main():
    root = None
    while True:
        print("1. Ekle")
        print("2. Sil")
        print("3. Ara")
        print("4. Yazdır")
        print("5. Çıkış")
        choice = int(input("Seçiminizi girin: "))

        if choice == 1:
            value = int(input("Eklemek istediğiniz değeri girin: "))
            root = insert(root, value)
        elif choice == 2:
            value = int(input("Silmek istediğiniz değeri girin: "))
            root = delete(root, value)
        elif choice == 3:
            value = int(input("Aramak istediğiniz değeri girin: "))
            if search(root, value):
                print("Değer bulundu.")
            else:
                print("Değer bulunamadı.")
        elif choice == 4:
            print_tree(root)
        elif choice == 5:
            break
        else:
            print("Hatalı seçim!")

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

Kod Açıklaması:

- Node sınıfı, ikili arama ağacındaki bir düğümü temsil eder.
- insert 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 – Herhangi bir sudoku bulmacasını çözebilen bir sudoku çözme programını kodlayın. Bu bulmacaları neredeyse anında çözmelidir.
  (Bu alıştırmayı tamamlamak için iki fonksiyona ihtiyacınız olacak, solutionOK fonksiyonu ve solutionViable fonksiyonu.)

```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.
    """
```

```python
    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 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.

```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

def main():
    # Test your OrderedTreeSet class here
    tree_set = OrderedTreeSet()
    tree_set.add(5)
    tree_set.add(3)
    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.

İlk olarak, **orderedtreeset.py** modülünü oluştralım:

```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. '**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.

Bölüm Hedefi Sorularının Cevapları

Ağaçlar Nasıl İnşa 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:

I. **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.

II. **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.

III. **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.

IV. **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.

V. **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.

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.

İfadeler ve Ağaçlar Nasıl İlişkilidir?

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:

I. **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.

II. **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.

III. **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.

IV. **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 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.

I.

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.

Derinlik Öncelikli Arama Nedir ve Ağaçlar ve Arama Problemleriyle Nasıl İlişkilidir?

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?

I. **Pre-order Geçiş (Pre-order Traversal):**
   I. Pre-order geçişinde, her düğüm önce kök düğüm olarak işaretlenir, ardından sol alt ağaç gezilir ve son olarak sağ alt ağaç gezilir.
   II. Geçiş sırasında düğümler, önce kök düğümden başlayarak sıralanır, ardından sol alt ağaç gezilir ve son olarak sağ alt ağaç gezilir.
   III. Pre-order geçiş, düğümleri kök-sol-sağ sırasında işlemek için kullanılır.

II. **In-order Geçiş (In-order Traversal):**
   I. In-order geçişinde, sol alt ağaç önce gezilir, ardından kök düğüm işlenir ve en son olarak sağ alt ağaç gezilir.
   II. Bu geçiş sırasında düğümler, sıralı bir şekilde işlenir. Örneğin, küçükten büyüğe sıralanmış bir ağaçta düğümler, küçükten büyüğe doğru sıralanarak işlenir.
   III. In-order geçiş, ikili arama ağaçlarının sıralı listesini elde etmek için sıkça kullanılır.

III. **Post-order Geçiş (Post-order Traversal):**
   I. Post-order geçişinde, sol alt ağaç gezilir, sonra sağ alt ağaç gezilir ve en sonunda kök düğüm işlenir.
   II. Bu geçiş sırasında düğümler, alt ağaçlar önce gezilerek işlenir ve ardından kök düğüm işlenir.
   III. 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 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 Anlama (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.

•

- Elif Arşa Polat - 2023481028
- Hamza Çelik - 2023481030
- Cenk Darihan – 2023481032