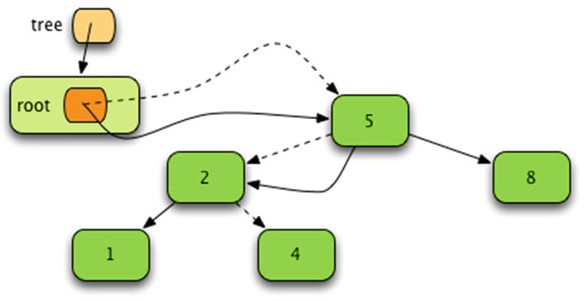
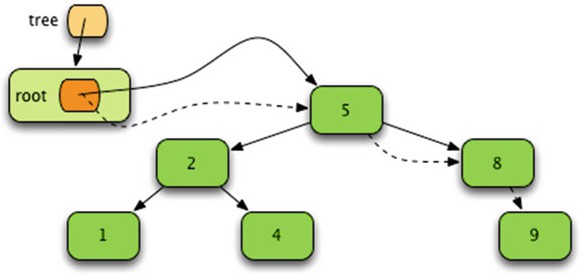


**Şekil 6.7 Eklemeden Sonra Ağaç**

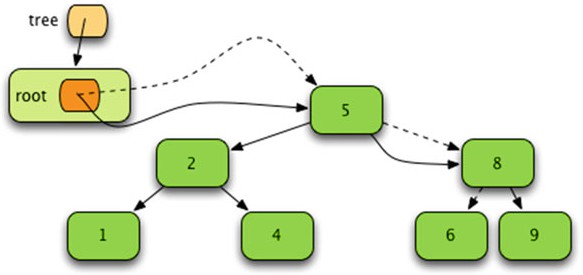


**Şekil 6.8 Eklemeden Sonra Ağaç**

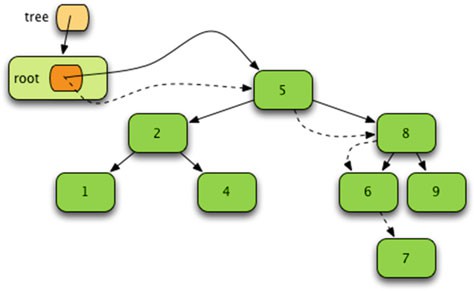
9'u eklemek için, şimdiye kadar eklenen tüm düğlerden daha büyük olduğu için ağacın sağ tarafına gitmesigerekir. Bu, Şekil 6.9'da gösterilmiştir.  
6,Şekil 6.10'da 5'in sağında ve 8'in solunda gider.  
7'nin gidebileceği tek yer, 5'in sağında, 8'in solunda ve 6'nın sağında, Şekil 6.11'de gösterildiği gibidir.  
Son ağaç, Şekil 6.12'de gösterilmiştir. Bu bir ikili arama ağacıdır çünkü alt ağaçları olan tüm düğümlerin değerleri, sol alt ağaçta düğümden küçük ve sağ alt ağaçta düğümden büyük ya da eşittir ve her iki alt ağaç da ikili arama ağacı özelliğine uyar.  
Programın 6.5.1. bölümündeki son kısmı, ana fonksiyonda ağaç üzerinde yineleme yapar. Bu, BinarySearchTree sınıfının iter yöntemini çağırır. Bu iter yöntemi, kök Node nesnesi üzerinde bir yineleyici döndürür. Node'un iter yöntemi ilginçtir çünkü bu, ağacın özyinelemeli bir gezintisidir. for elem in self.left yazıldığında, bu sol alt ağacın iter yöntemini çağırır. Sol alt ağaçtaki tüm öğeler döndürüldükten sonra, ağacın kökündeki değer döndürülür.



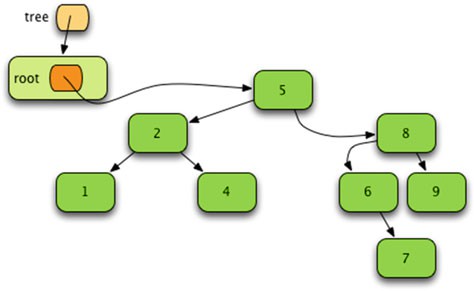
**Şekil 6.9 Eklemeden Sonra Ağaç**



**Şekil 6.10 Eklemeden Sonra Ağaç**



**Şekil 6.11 Eklemeden Sonra Ağaç**



**Şekil 6.12 Son İkili Arama Ağacı (BinarySearchTree) Nesnesi İçeriği**

O zaman, sağ alt ağaçtaki değerler for elem in self.right yazılarak döndürülür. Bu özyinelemeli fonksiyonun sonucu, ağacın sıralı gezintisi (inorder traversal) olur.  
İkili arama ağaçları bazı akademik ilgiye sahiptir. Ancak, pratikte çok fazla kullanılmazlar. Ortalama durumda, bir ikili arama ağacına eleman eklemek O(log n) zaman alır. Bir ikili arama ağacına n öğesi eklemek O(n log n) zaman alır. Yani, ortalama durumda, sıralı öğelerden oluşan bir diziyi sıralamak için bir algoritmamız vardır. Ancak, bir listeye göre daha fazla alan kullanır ve quicksort algoritması, aynı büyük-O karmaşıklığı ile bir listeyi sıralayabilir. En kötü durumda, ikili arama ağaçları, quicksort'un karşılaştığı aynı sorunu yaşar. Öğeler zaten sıralı olduğunda, hem quicksort hem de ikili arama ağaçları zayıf performans gösterir. Bir ikili arama ağacına n öğesi eklemenin karmaşıklığı, en kötü durumda O(n²) olur. Eğer değerler zaten sıralıysa, ağaç bir çubuk haline gelir ve temelde bir bağlı listeye dönüşür.  
İkili arama ağaçlarının, rastgele erişimli bir listenin sahip olmadığı birkaç hoş özelliği vardır. Bir ağaca ekleme işlemi, ortalama durumda O(log n) zaman alırken, bir listeye eklemek O(n) zaman alır. İkili arama ağacından silme işlemi de ortalama durumda O(log n) zaman alabilir. Bir ikili arama ağacında bir değeri aramak da ortalama durumda O(log n) zaman alabilir. Eğer bir algoritma için birçok ekleme, silme ve arama işlemi yapılıyorsa, ağaç benzeri bir yapı faydalı olabilir. Ancak, ikili arama ağaçları O(log n) karmaşıklığını garanti edemez. Gerçekten de, ekleme, silme ve değer arama işlemleri için O(log n) veya daha iyi karmaşıklık garantisi verebilen arama ağacı yapıları implementasyonları vardır. Bunlardan birkaç örnek, bu metnin ilerleyen bölümlerinde incelenecek olan Splay Ağaçları, AVL Ağaçları ve B-Ağaçları'dır.

**Arama Alanları**

Bazen birçok farklı durumdan oluşan bir problemle karşılaşabiliriz. Bu durumda, problemin belirli bir durumunu bulmak isteyebiliriz, bu durumu hedef (goal) olarak adlandıracağız. Sudoku bulmacalarını ele alalım. Bir Sudoku bulmacası, çözümünü ne kadar tamamladığımızı yansıtan bir duruma sahiptir. Burada amacımız, bulmacanın çözümü olan hedefi bulmaktır. Bir hücreye rastgele bir değer deneyip, bu tahminle bulmacayı çözmeye çalışabiliriz. Bu tahmin, bulmacanın yeni bir durumuna yol açacaktır. Ancak, tahmin yanlış olursa, tahminimizi geri alıp tekrar denememiz gerekebilir. Yanlış bir tahmin, bir çıkmaz sokağa (dead end) yol açabilir. Bu tahmin yapma, bulmacayı çözmeye çalışma ve kötü tahminleri geri alma süreci derinlik öncelikli arama (depth first search) olarak adlandırılır. Bir hedefi bulmak için tahminler yapmak, bir problemin uzayında derinlik öncelikli arama olarak adlandırılır. Eğer bir çıkmaz sokağa girilirse, geri izlememiz (backtracking) gerekebilir. Geri izleme, kötü tahminleri geri almayı ve sonra problemi çözebilmek için yeni bir tahminle tekrar denemeyi içerir. Burada yapılan açıklama, 12.2.1 bölümünde derinlik öncelikli arama algoritmasına yol açmaktadır.

**Derinlik-Önce 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 kadar olan yolu döndürür. Geri izleme (backtracking), for döngüsünün uygun bir komşu düğüm bulamadan tamamlanması durumunda gerçekleşir. Bu durumda, None döndürülür ve dfs'in önceki özyinelemeli çağrısı, o yolda hedefi aramak için bir sonraki komşu düğüme,geçer.  
Son bölümde, birçok bulmacada çalışmakla birlikte tüm bulmacalar için geçerli olmayan bir Sudoku çözme algoritması sunulmuştu. Bu tür durumlarda, problemi mümkün olduğunca küçültüp sonra derinlik öncelikli arama algoritması Sudoku bulmacasına uygulanabilir. Bulmacayı küçültmeden önce, geçen bölümdeki kuralların uygulanması önemlidir, çünkü aksi takdirde arama alanı, makul bir süre içinde aranamayacak kadar büyük olur. 6.6.2 bölümündeki solve fonksiyonu, geçen bölümdeki kuralları bulmacadaki tüm gruplara uyguladığı varsayılarak her türlü Sudoku bulmacasını çözecek şekilde derinlik öncelikli arama içerir. Bu kodun doğru çalışabilmesi için copy modülünün içe aktarılması gerekir.

.

**Sudoku Derinliği-İlk Arama**

1 **def** solutionViable(matrix):

2 *# Check that no set is empty*

3 **for** i **in** range(9):

4 **for** j **in** range(9):

5 **if** len(matrix[i][j]) == 0:

6 **return** False

7

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:

32 **return** result

33

34 **return** None

6.6.2 bölümündeki solve fonksiyonunda, önce geçen bölümdeki kurallarla bulmacayı çözmeye çalışmak için reduce çağrılır. reduce çağrıldıktan sonra, bulmacanın hala çözülebilir olup olmadığı (yani, boş kümeler olup olmadığı) kontrol edilir. Eğer çözülemezse, solve fonksiyonu None döndürür. Arama, matris içindeki her bir konum ve o konumun alabileceği tüm olası değerler incelenerek devam eder. for k döngüsü, bir hücrede birden fazla olasılık varsa, tüm olasılıkları dener. Eğer reduce çağrısı bulmacayı çözerse, solutionOK fonksiyonu True döndürecek ve solve fonksiyonu matrisi döndürecektir. Aksi takdirde, derinlik öncelikli arama, matris içinde birden fazla seçeneği olan bir hücre arayarak devam eder. Fonksiyon, matrisin bir kopyasını mcopy olarak oluşturur ve bu konumda bir değer tahmini yapar. Ardından, solve fonksiyonunu mcopy üzerinde özyinelemeli olarak çağırır.  
solve fonksiyonu, çözüm bulunamazsa None döndürür, çözüm bulunursa çözülmüş bulmacayı döndürür. Yani, solve özyinelemeli olarak çağrıldığında, eğer None döndürülürse, fonksiyon başka bir olası değeri deneyerek aramaya devam eder. Başlangıçta solve fonksiyonunun çağrılması, 6.6.3 bölümünde gösterildiği şekilde, matrisin 9 × 9 boyutunda, bir Sudoku bulmacasını temsil eden kümelerden oluştuğu varsayılarak yapılabilir.

**Sudoku Çözme Fonksiyonunun Çağrılması**

1 **print**("Begin Solving")

2

3 matrix = solve(matrix)

4

5 **if** matrix == None:

6 **print**("No Solution Found!!!")

7 **return**

Eğer None olmayan bir matris döndürülürse, o zaman bulmaca çözülmüş demektir ve çözüm yazdırılabilir. Bu, hiçbir ağaç yapısının oluşturulmadığı bir örnektir, ancak arama alanı bir ağaç gibi şekillenir ve derinlik öncelikli arama, problem alanını aramak için kullanılabilir.

**Bölüm Özeti**

Ağaç benzeri yapılar, Bilgisayar Bilimleri'nde birçok problemde karşımıza çıkar. Bir ağaç veri tipi, bilgi tutabilir ve hızlı ekleme, silme ve arama işlemleri yapabilir. İkili arama ağaçları pratikte kullanılmasa da, bunları yöneten ilkeler, B-ağaçları, AVL-ağaçları ve Splay Ağaçları gibi birçok ileri düzey veri yapısında kullanılır. Nesnelere referansların nasıl işaret ettiğini ve bunun bir ağaç gibi bir veri tipi oluşturmak için nasıl kullanılabileceğini anlamak, bilgisayar programcılarının anlaması gereken önemli bir kavramdır.

Arama alanları, birkaç seçenek arasında karar vermek, başka bir karara yol açtığında genellikle ağaç benzeri bir yapı oluşturur. Arama alanı bir veri tipi değildir, bu nedenle bu durumda bir ağaç oluşturulmaz. Ancak, aranan alan ağaç benzeri bir yapıya sahiptir. Bir alanı derinlik öncelikli arama ile taramanın anahtarı, nereye gittiğinizi hatırlamaktır, böylece bir seçim çıkmaz sokağa yol açarsa geri izleme yapabilirsiniz. Geri izleme, genellikle özyineleme (recursion) kullanılarak yapılır.

Ağaçlarla ilgili birçok algoritma doğal olarak özyinelemeli (recursive) olabilir. Derinlik öncelikli arama, ağaç gezintileri (tree traversals), ayrıştırma (parsing) ve soyut sözdizimi değerlendirmesi (abstract syntax evaluation) tümü özyinelemeli olarak uygulanabilir. Özyineleme, problem çözme için araç kutunuzda sahip olmanız gereken güçlü bir mekanizmadır.

**İnceleme Soruları**

Bu kısa cevaplı, çoktan seçmeli ve doğru/yanlış soruları, bölümdeki bilginizi test etmek için cevaplayın.

1. Bir ağacın kökü bilgisayar bilimlerinde ağacın üst kısmında mı yoksa alt kısmında mı bulunur?

Cevap Kök ağacın **üst kısmında** bulunur.

1. Bir ağacın kaç kökü olabilir?

Cevap:Bir ağacın bir kökü olabilir

1. Dolu bir ikili ağaç, her seviyede tamamen dolu olan bir ağaçtır, yani ağacın hiçbir seviyesinde, yapraklar hariç başka bir düğüm eklenemez. Üç seviyeli bir dolu ikili ağaçta kaç düğüm vardır? Peki ya 4 seviyeli? 5 seviyeli?

Cevap:

Dolu bir ikili ağaçta:

* 3 seviye: **7 düğüm**
* 4 seviye: **15 düğüm**
* 5 seviye: **31 düğüm**

1. Dolu bir ikili ağacın yaprak sayısı ile toplam düğüm sayası arasındaki ilişki nedir?

Cevap: Dolu bir ikili açağta yaprak sayısı toplam düğüm sayısının yarısı kadardır, bir eksik olmak şartıyla.

1. Bir ağaç inşa ederken, hangi durumda kod yazmak daha kolaydır: alt-üst mü yoksa üst-alt mı?

Cevap:Bir ağacın üst-alt olarak inşa edilmesi genellikle daha kolaydır.

1. Bir ağaçta bir değeri ararken yanlış bir seçim yapıldığında ve başka bir seçim yapılması gerektiğinde hangi terim kullanılır?

Cevap:Bu terim geri izleme (backtracking) olarak bilinir.

1. Arama alanı (search space) ile ağaç veri tipi arasındaki fark nedir?

Cevap: **Arama alanı**, bir problemin tüm olası çözüm ya da durumlarının kümesini ifade ederken, **ağaç veri tipi** belirli bir veri yapısının, öğeler arasındaki hiyerarşik ilişkileri temsil eden bir yapıdır.

1. Bir ağacın sırasıyla gezilmesi (inorder traversal) için doğrusal olmayan bir algoritma tanımlayın. İpucu: Algoritmanızın bunu gerçekleştirebilmesi için bir yığına (stack) ihtiyacı olacak.

Cevap:

def inorder\_traversal(root):

stack = []

current = root

while current is not None or stack:

# Sol alt ağaç boyunca ilerle

while current:

stack.append(current)

current = current.left

# Yığıtın en üstündeki düğümü al

current = stack.pop()

print(current.value) # Düğümün değerini yazdır

# Sağ alt ağaçta ilerle

current = current.right

Bu algoritma, önce ağacın sol alt ağaçlarına iner, ardından düğümü işler ve son olarak sağ alt ağaçları gezmeye başlar.

1. 5 \* 4 + 3 \* 2 infiks ifadesi için bir ağaç oluşturacak kod yazın. Operatörlerin önceliğini ve ağacınızı takip edin. PlusNode ve TimesNode sınıflarının önceden tanımlandığını varsayabilirsiniz.

Cevap: Öncelikle, operatörlerin önceliğini dikkate alarak bir ağaç inşa etmemiz gerekir. Burada çarpma işlemi, toplama işleminden önce yapılır. Ağaç şöyle kurulur:

+

/ \

\* \*

/ \ / \

5 4 3 2

class PlusNode:

def \_\_init\_\_(self, left, right):

self.left = left

self.right = right

self.value = '+'

class TimesNode:

def \_\_init\_\_(self, left, right):

self.left = left

self.right = right

self.value = '\*'

class ValueNode:

def \_\_init\_\_(self, value):

self.value = value

# Düğümleri oluştur

left\_times = TimesNode(ValueNode(5), ValueNode(4))

right\_times = TimesNode(ValueNode(3), ValueNode(2))

root = PlusNode(left\_times, right\_times)

Burada, PlusNode ve TimesNode sınıfları ile işlemleri temsil ettik ve yapıyı kurduk.

1. 5 \* 4 + 3 \* 2 ifadesinin prefix ve postfix biçimlerini verin.

Cevap:

 **Prefix (ön ek) formu**: + \* 5 4 \* 3 2

 **Postfix (son ek) formu**: 5 4 \* 3 2 \* +

Prefix (ön ek) formunda, operatörler önce gelirken, postfix (son ek) formunda operatörler en son gelir.