Dengeli İkili Arama Ağaçları

10

Bölüm 6'da ikili arama ağaçları bir özyinelemeli ekleme algoritması ile birlikte tanımlanmıştır. İkili arama ağaçlarının tartışılması, bazı durumlarda sorunlara sahip olduklarına işaret etmiştir. İkili arama ağaçları dengesiz hale gelebilir, aslında oldukça sık. Bir ağaç dengesiz olduğunda ekleme, silme ve arama işlemlerinin karmaşıklığı O-(n) kadar kötü olabilir. Dengesiz ikili arama ağaçlarıyla ilgili bu sorun, 1962 yılında iki Sovyet bilgisayar bilimcisi olan G.M. Adelson-Velskii ve E.M. Landis tarafından yükseklik dengeli AVL ağaçlarının geliştirilmesi için motivasyon kaynağı olmuştur. AVL ağaçları adını bu iki mucitten almıştır. AVL ağaçları hakkındaki makaleleri [1], dengeli ikili arama ağaçlarını korumak için ilk algoritmayı tanımlamıştır.

Dengeli ikili arama ağaçları *O(log n)* ekleme, silme ve arama işlemleri sağlar. Buna ek olarak, dengeli bir ikili arama ağacı öğelerini sıralı düzende tutar. Bir ikili arama ağacının infix çaprazlaması, öğelerini artan sırada verecektir ve bu çaprazlama, ağacın zaten oluşturulmuş olduğu varsayılarak O-(n) zamanda gerçekleştirilebilir.

HashSet ve HashMap sınıfları da çok verimli ekleme, silme ve arama işlemleri sağlar, ilgili ikili arama ağacı işlemlerinden daha verimlidir. Yığınlar da O(log n) ekleme ve silme işlemleri sağlar. Ancak ne hash tabloları ne de yığınlar öğelerini sıralı bir dizi olarak tutmaz. Çok sayıda ekleme ve silme işlemi gerçekleştirmek istiyorsanız ve bir dizi üzerinde artan veya azalan sırada, belki de birçok kez yinelemeniz gerekiyorsa, dengeli bir ikili arama ağacı veri yapısı daha uygun olabilir.

10.1 Bölüm Hedefleri

Bu bölümde ikili arama ağaçlarının neden dengesiz olabileceği açıklanmaktadır. Daha sonra iki tip yükseklik dengeli ağacın, AVL ağaçlarının ve yayvan ağaçların çeşitli uygulamalarını tanımlamaya devam etmektedir. Bu bölümün sonunda, yinelemeli veya özyinelemeli olarak uygulanan işlemlerle kendi AVL veya splay ağaç veri türünüzü uygulayabilmelisiniz.

Springer International Publishing Switzerland 2015
K.D. Lee ve S. Hubbard, *Pvthon ile Veri Yapıları ve Al poritmalar,*
Undergraduat
e Topics i

10.2 İkili Arama Ağaçları

İkili arama ağacı, bir uygulama tarafından çok sayıda ekleme, silme ve arama işlemi gerektiğinde ve bazen öğeleri artan veya azalan sırada gezmek gerektiğinde kullanışlıdır. Wikipedia gibi geniş bir çevrimiçi materyal setine erişim sağlayan bir web sitesi düşünün. Web sitesinin tasarımcılarının son bir saat içinde web sitesine erişen tüm kullanıcıların kaydını tutmak istediğini düşünün. Web sitesi aşağıdaki gibi çalışabilir.

- Her ziyaretçi web sitesine benzersiz bir çerez ile erişir.
- Bir ziyaretçi siteye eriştiğinde, çerezleri tarih ve saatle birlikte sitenin sunucusundaki bir günlüğe kaydedilir.
- Siteye son iki saat içinde erişmişlerse, çerezleri ve erişim zamanları zaten kaydedilmiş olabilir. Bu durumda, son erişim tarih ve saati güncellenir.
- Her saat başı o anda siteye kimlerin eriştiğine dair bir anlık görüntü oluşturulur.
- Anlık görüntü, benzersiz çerez numaralarının artan sırasına göre oluşturulacaktır.
- Anlık görüntüye göre, bir kullanıcı en az bir saat boyunca aktif olmadıktan sonra, bilgileri web sitesi etkinlik günlüğü kayıtlarından silinir.

Site oldukça büyük olduğundan ve her saat binlerce, hatta on binlerce veya daha fazla kişi siteye eriştiğinden, bu bilgileri tutacak veri yapısının hızlı olması gerekir. Girdileri eklemek, aramak ve silmek hızlı olmalıdır. Ayrıca anlık görüntü almak da hızlı olmalıdır çünkü anlık görüntü alınırken web sitesi tüm istekleri bekletecektir.

Wikipedia gibi bir sitede bir saat içinde gelip giden kullanıcı sayısı genellikle uzun süre kalan kullanıcı sayısından fazlaysa, en az bir saat boyunca etkin olmayan her girdiyi silmek yerine etkinlik günlüğünden ağacı yeniden oluşturmak en verimli yöntem olabilir. Bu durum, sitede hala aktif olan kişi sayısı, günlüğün anlık görüntüsündeki aktif olmayan girişlerin sayısından çok daha azsa geçerli olacaktır. Bu durumda, etkin olmayan kullanıcıları sildikten sonra günlüğü yeniden oluşturmak da hızlı olmalıdır.

İkili arama ağacı, O-(log *n)* arama, ekleme ve silme ile O(n) anlık görüntü alma süresini garanti edebiliyorsak, bu günlüğün organizasyonu için mantıklı bir seçimdir. Ancak, ikili arama ağacının büyük bir sorunu vardır. Anlık görüntü alınırken günlüğün yalnızca son aktif kullanıcılarla yeniden oluşturulabileceğini ve ayrıca günlüğün yeniden oluşturulması sırasında çerezlere artan sırada erişileceğini hatırlayın.

Bölüm 10.2.1'de gösterilen ikili arama ağaçları üzerindeki ekleme işlemini düşünün. İkili arama ağacı yeniden oluşturulduğunda, yeni ağaca eklenecek öğeler artan sırada eklenecektir. Sonuç dengesiz bir ağaçtır.

10.2.1 İkili Arama Ağacı Ekleme

```
i  def    insert(root,val):
       zif kökü   Yok:
;          return BinarySearchTree. Node(val)
```

```
4       if val < root.getVal():
5           root.setLeft(PinarySearchTree. insert(root.getLeft(),val))
6       elee:
7           root.setRight(BinarySearchTree. insert(root.getRight(),val))
8       kök döndür
```

Öğeler bir ikili arama ağacına artan sırada eklenir, bunun etkisi yürütmenin her zaman satır 2'den 4, 6, 7 ve 8'e ilerlemesidir. Satır 7'deki sonuç, yeni değeri ikili arama ağacının en sağ konumuna yerleştirir, çünkü o ana kadar eklenen en büyük değerdir. Ortaya çıkan ağaç, aşağı ve sağa doğru uzanan bir çubuktur. Ağaçta herhangi bir denge olmadan, bir sonraki büyük değerin eklenmesi, yeni değerin konumunu bulmak için daha önce eklenmiş olan her bir değerin geçilmesine neden olacaktır. Bu, ilk değerin eklenmesi için sıfır karşılaştırma gerekirken, ikincisinin son konumunu bulmak için bir karşılaştırma gerektirdiği, üçüncü değerin iki karşılaştırma gerektirdiği ve bu şekilde devam ettiği anlamına gelir. Bölüm 2'de kanıtlandığı gibi, ağacı oluşturmak için toplam karşılaştırma sayısı $O(n^2)$'dir. Bu karmaşıklık, bir saat içinde makul miktarda etkinlik alan herhangi bir site için çok yavaş olacaktır. Buna ek olarak, ikili arama ağacının yüksekliği *n* olduğunda, burada *n* ağaçtaki değer sayısıdır, arama, ekleme ve silme süreleri hem en kötü hem de ortalama durumlar için O-*(n)*'dir. Ağaç bir çubuk olduğunda ya da çubuk olmaya yakın olduğunda ikili arama ağacının verimlilik özellikleri bağlı listeden daha iyi değildir.

10.3 AVL Ağaçları

Dengeli kalan bir ikili arama ağacı, son bölümde açıklanan web sitesi günlüğünün gerektirdiği her şeyi sağlayacaktır. AVL ağaçları, dengelerini korumak için ek bilgiler içeren ikili arama ağaçlarıdır. Bir AVL ağacının yüksekliğinin O(log *n)* olması garanti edilir, böylece arama, ekleme ve silme işlemlerinin hepsinin O(log n) zamanında tamamlanacağı garanti edilir. Bu garantilerle, bir AVL ağacı *n* öğeden oluşan bir diziden *O-(n log n)* zamanda oluşturulabilir. Dahası, AVL ağaçları, ikili arama ağaçları gibi, *O-(n)* zamanda artan sırada öğelerini veren bir sıralı geçiş kullanılarak çaprazlanabilir.

10.3.1 Tanımlar

AVL ağaçlarının nasıl çalıştığını anlamak için birkaç tanım yapmak gerekir.

Yükseklik(Ağaç): Bir ağacın yüksekliği bir artı alt ağaçlarının maksimum yüksekliğidir. Bir yaprak düğümün yüksekliği birdir.
Denge(Ağaç): İkili bir ağaçtaki bir düğümün dengesi yükseklik(sağ alt ağaç)-yükseklik(sol alt ağaç) şeklindedir.
AVL Ağacı: AVL ağacı, ağaçtaki her düğümün dengesinin -1, 0 veya 1 olduğu ikili bir ağaçtır.

10.3.2 Uygulama Alternatifleri

Bölüm 6'ya ve ikili arama ağaçlarının uygulanmasına geri dönersek, bir ağaca değer eklemek özyinelemeli olarak yazılabilir. Bir AVL ağacına değer eklemek de özyinelemeli olarak gerçekleştirilebilir. Bir AVL ağacına değer ekleme işlemini bir döngü ve bir yığın kullanarak yinelemeli olarak uygulamak da mümkündür. Bu bölüm her iki alternatifi de incelemektedir.

Ayrıca, bir AVL ağacının dengesi ağaçtaki her bir düğümün yüksekliği ya da ağaçtaki her bir düğümün dengesi kullanılarak korunabilir. AVL ağaç düğümlerinin uygulamaları ya dengelerini ya da yüksekliklerini saklar. Değerler ağaca eklendikçe, etkilenen düğümlerin denge veya yükseklik değerleri ağaca eklenen yeni öğeyi yansıtacak şekilde güncellenir.

10.3.3 Depolanmış Bakiye ile AVLNode

```
class AVLTree:
    class AVLNode:
        def __init__(self, item, balance=0, left=None, right=None):
            self.item = item
            self.left = left
            self.right = right
            self.balance = balance

        def __repr__(self):
            return "AVLTree.AVLNode(" + repr(self.item) + ", balance=" +
repr(self.balance) + ", left=" + repr(self.left) + ", right=" + repr(self.right) +
")"
```

İster özyinelemeli ister yinelemeli ekleme uygulansın, Bölüm 6'daki *Node* sınıfı, düğümün dengesine veya yüksekliğine uyum sağlamak için biraz genişletilmelidir. Bölüm 10.3.3'teki kod parçasını düşünün. AVLTree'nin inceleyeceğimiz ilk uygulaması, algoritmanın denge depolayan yinelemeli bir sürümüdür. AVLNode uygulamasının AVLTree sınıfının kullanıcılarından *gizlemek* için AVLTree sınıfının içine gömüldüğüne dikkat edin. Python aslında AVLTree sınıfının dışından AVLNode sınıfına erişimi engellemese de, AVLTree veri yapısının kullanıcıları kural olarak ağacın iç kısımlarını rahat bırakmayı bilmelidir. AVL ağaçları bu veri yapısının kullanıcıları tarafından oluşturulur, ancak AVL düğümleri oluşturulmaz. Düğümlerin oluşturulması ALVTree sınıfı tarafından gerçekleştirilir.

AVLNode kurucusunun denge, sol ve sağ için varsayılan değerleri vardır, bu da kodda hata ayıklarken AVLTrees oluşturmayı kolaylaştırır. *Therepr* fonksiyonu AVLNode'u böyle bir düğüm oluşturmak için kullanılabilecek bir biçimde yazdırır. *print(repr(node))* çağrısı bir düğümü yazdırır, böylece Python'a örnek bir ağaç oluşturması için sağlanabilir. repr*(self.left)* ve *repr(self.right), oradapr* fonksiyonuna özyinelemeli çağrılardır, bu nedenle tüm ağaç *self*'e köklenmiş olarak yazdırılır. Bölüm 6'dan itibaren aynı *liters* fonksiyonu bir AVLTree'yi

10.3 AVL
Ağaçları

AVL Ağaçları, ancak belki de ağaçtaki her bir düğümün yüksekliğini korumaktan biraz daha zordur. Bölümün ilerleyen kısımlarında bu algoritmaların her bir düğümün yüksekliğini koruyan modifikasyonları tartışılacaktır. AVL Ağaçlarında ister yükseklik ister denge saklansın, ağaç işlemlerinin karmaşıklığı etkilenmez.

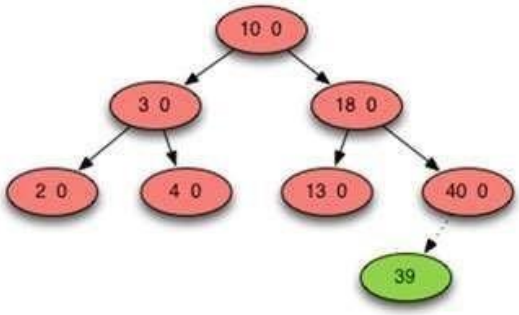
10.3.4 AVL Ağacı Yinelemeli Ekleme

Son bölümde açıklandığı gibi, yükseklik dengeli AVL ağaçları için ekleme algoritmasının iki çeşidi vardır. Ekleme yinelemeli veya özyinelemeli olarak gerçekleştirilebilir. Denge ayrıca açık bir şekilde saklanabilir veya her bir alt ağacın yüksekliğinden hesaplanabilir. Bu bölümde, her bir düğümün yüksekliğini muhafaza etmeden dengenin açık bir şekilde nasıl muhafaza edileceği açıklanmaktadır.

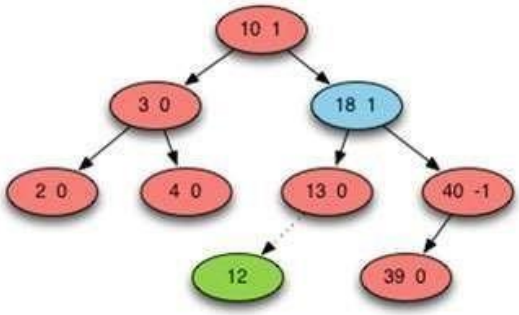
Yüksekliği dengelenmiş bir AVL ağacına yeni bir değerin yinelemeli olarak eklenmesi, yeni eklenen değere giden yolun takip edilmesini gerektirir. Bu yolu korumak için bir yığın kullanılır. Algoritmada bu yığına *yol yığını* diyeceğiz. Yeni bir düğüm eklemek için, kökten yeni düğümün konumuna giden benzersiz arama yolunu izleriz ve ilerledikçe her düğümü, tıpkı bir ikili arama ağacına ekliyormuşuz gibi, yol yığınına iteriz. Yeni düğümün hedefine giden yol boyunca ilerlerken, karşılaştığımız tüm düğümleri *yol yığınına* iteriz. Yeni öğeyi ikili arama ağacı özelliğine göre olması gereken yere yerleştiririz. Daha sonra algoritma, yol yığınından değerleri çıkarmaya ve ayarlanmadan önce sıfıra eşit olmayan bir dengeye sahip bir düğüm bulunana kadar dengelerini ayarlamaya devam eder. Sıfır olmayan bakiyeye sahip en yakın ata olan bu düğüme *pivot* adı verilir. Pivot ve yeni değerin konumuna bağlı olarak, aşağıda açıklanan birbirini dışlayan üç durum göz önünde bulundurulmalıdır. Aşağıdaki 3. durumda ayarlamalar yapıldıktan sonra, pivotta köklenen alt ağaç için yeni bir kök düğüm olabilir. Bu durumda, pivotun ebeveyni yol yığınındaki bir sonraki düğümdür ve yeni alt ağaca bağlanabilir. Pivot patlatıldıktan sonra yol yığını boşsa, ağacın kökü pivottur. Bu durumda, AVL ağacının kök düğümü, ağaçtaki yeni kök düğümü gösterecek şekilde yapılabilir. Belirtildiği gibi
Yukarıda, ağaca yeni bir değer eklerken üç durumdan biri ortaya çıkacaktır.

Durum 1: Pivot Yok Pivot düğüm yoktur. Bu durumda, arama yolundaki her bir düğümün dengesini, her bir düğümün anahtarına göre yeni anahtarın göreli değerine göre ayarlayın. Yeni düğüme giden yolu incelemek için *yol* yığınını kullanabilirsiniz.

Bu durum AVL ağacına 39'un ekleneceği Şekil 10.1'de gösterilmektedir. Her düğümde değer solda ve denge sağda verilmiştir. 10, 18 ve 40 içeren düğümlerin her biri *yol yığınına* itilir. 39'u içeren yeni düğümün bakiyesi 0 olarak ayarlanır. 40'ı içeren düğümün yeni bakiyesi
-1. 16 içeren düğümün yeni bakiyesi 1'dir. Eklemeden sonra kök düğümün bakiyesi 1'dir çünkü 39 sağına eklenmiştir ve bu nedenle bakiyesi bir artar. Yeni değer 40'ı içeren düğümün soluna eklenir, bu nedenle bakiyesi bir azalır. Şekil 10.2 yeni değerin eklenmesiyle değişen ağacı göstermektedir.



Şekil 10.1 AVL Ağacı Vaka 1-Pivot Düğüm Yok



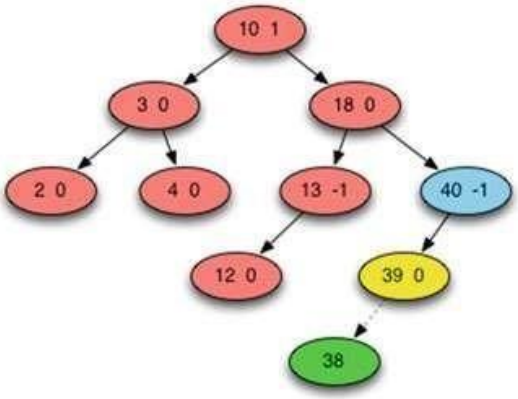
Şekil 10.2 AVL Ağacı Vaka 2-Döndürme Yok

Durum 2: Dengeleri **Ayarla** Pivot düğüm mevcuttur. Ayrıca, yeni düğümün eklendiği pivot düğümün alt ağacı daha küçük yüksekliğe sahiptir. Bu durumda, arama yolu boyunca yeni düğümden pivot düğüme kadar olan düğümlerin dengesini değiştirin. Pivot düğümün üzerindeki düğümlerin dengeleri etkilenmez. Bu doğrudur çünkü pivot düğümde köklenen alt ağacın yüksekliği yeni düğümün eklenmesiyle değişmez.

Şekil 10.2 bu durumu göstermektedir. *Anahtarı* 12 olan öğe AVL ağacına eklenmek üzeredir. 18'i içeren düğüm pivot düğümdür. Eklenecek değer 18'den küçük olduğundan ve 18'i içeren düğümün bakiyesi 1 olduğundan, yeni düğüm muhtemelen ağacın daha iyi dengelenmesine yardımcı olabilir. AVL ağacı AVL ağacı olarak kalır. Pivota kadar olan düğümlerin dengesi ayarlanmalıdır. Pivotun üzerindeki dengelerin ayarlanması gerekmez çünkü bunlar etkilenmez. Şekil 10.3, ağaca 12 eklendikten sonra ağacın nasıl göründüğünü göstermektedir.

Durum 3: Pivot düğüm mevcuttur. Ancak bu kez yeni düğüm, pivotun daha büyük yükseklikteki alt ağacına (dengesizlik yönündeki alt ağaç) eklenir. Bu, yeni düğüm eklendikten sonra pivot düğümün -2 veya 2 dengesine sahip olmasına neden olur, bu nedenle ağaç artık bir AVL ağacı olmayacaktır. Burada, ağacı AVL durumuna geri getirmek için *tek* bir *döndürme* veya *çift döndürme* gerektiren iki alt durum vardır. Dengesizlik yönündeki pivot düğümün çocuğunu *kötü çocuk* olarak adlandırın.

Alt Durum A: Tek **Rotasyon** Bu alt durum, yeni düğüm dengesizlik yönünde olan kötü çocuğun alt ağacına eklendiğinde ortaya çıkar. *Olasılık*



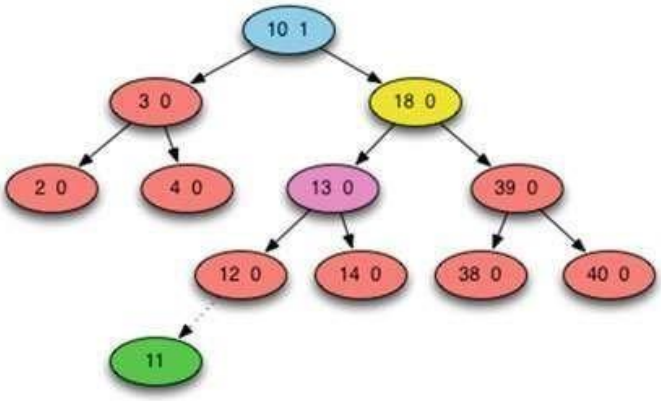
Şekil 10.3 AVL Ağacı Vaka 3A-Tek Rotasyon

pivot düğümde dengesizliğin tersi yönde bir döndürmedir. Döndürme işleminden sonra ağaç hala bir ikili arama ağacıdır. Buna ek olarak, pivotta köklenen alt ağaç bir kez daha dengelenir ve toplam yüksekliği bir azalır.

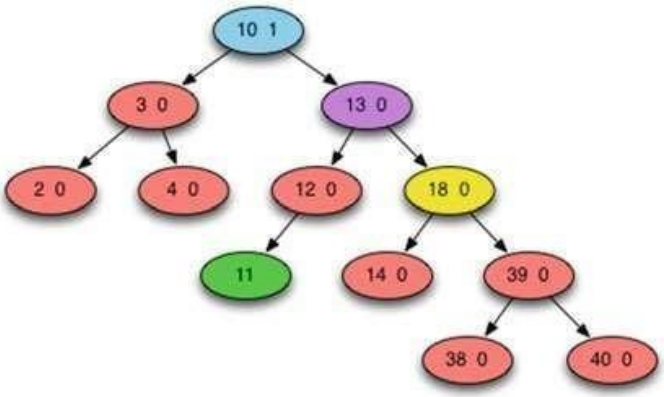
Şekil 10.3 bu alt durumu göstermektedir. 38 değeri, 39'u içeren düğümün solundaki ağaca eklenecektir. Ancak bunu yapmak, 40'ı içeren düğümün o1 dengesinin -2'ye düşmesine neden olur ki bu da uygunsuz dengeye sahip en yakın ata ve pivot düğümdür. Sarı düğüm *kötü çocuktur*. Buna ek olarak, 38 dengesizlikle aynı yönde yerleştirilmektedir. Dengesizlik sol taraftadır ve yeni değer sol tarafa eklenmektedir. Çözüm, 40'ta köklenen alt ağacı sağa döndürmektir, bu da Şekil 10.4'te gösterilen ağaçla sonuçlanır.

Alt Durum B: Çift Döndürme Bu alt durum, yeni düğüm dengesizliğin ters yönünde olan kötü çocuğun alt ağacına eklendiğinde ortaya çıkar. Bu alt durum için, kötü çocuğun arama yolu üzerinde bulunan çocuk düğümünü *kötü torun olarak* adlandırın. Bazı durumlarda, kötü torun olmayabilir. Şekil 10.4'te kötü torun mor düğümdür. Çözüm aşağıdaki gibidir:

1. Kötü çocukta dengesizlik yönünde tek bir rotasyon gerçekleştirin.
2. Pivotta dengesizlikten uzağa doğru tek bir dönüş gerçekleştirin.



Şekil 10.4 AVL Ağacı Vaka 3B-Çift Rotasyon

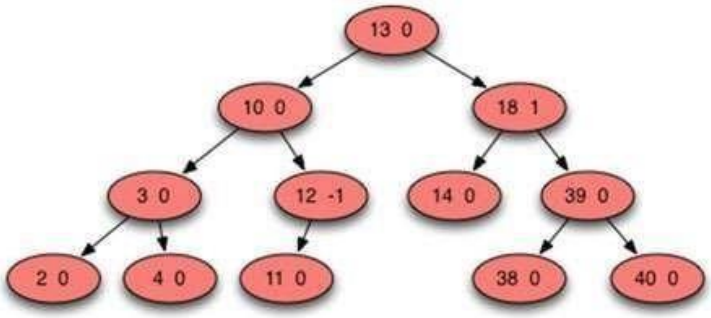


Şekil 10.5 AVL Ağacı Vaka 3B Adım 1 Döndürme Yönü

Yine, ağaç hala bir ikili arama ağacıdır ve orijinal pivot düğümün konumundaki alt ağacın yüksekliği çift döndürme ile değişmez. Şekil 10.4 bu durumu göstermektedir. Bu durumda pivot ağacın köküdür. 18'i içeren düğüm kötü çocuktur. Kötü torun ise 13'ü içeren düğümdür (Şekil 10.5).

Ağaçtaki dengesizlik pivotun sağındadır. Yine de 11 kötü çocuğun soluna yerleştirilmektedir. İlk adım, kötü çocukta sağa doğru bir rotasyondur. Bu, 11'*i yukarı getirerek* ağacın sağ tarafını dengelemeye bir *şekilde* yardımcı olur. Şekil 10.6'da gösterilen ikinci adım, pivotta sola dönerek tüm ağacı tekrar dengeye getirir.

Bu algoritmanın en zor kısmı bakiyeleri doğru şekilde güncellemektir. İlk olarak, pivot, kötü çocuk ve kötü torun değişebilecek bakiyeleri içerir. Eğer kötü torun yoksa, pivotun ve kötü çocuğun bakiyeleri sıfır olacaktır. Burada olduğu gibi kötü bir torun varsa, pivot ve kötü çocuğun bakiyelerini belirlemek için biraz daha fazla iş vardır. Kötü torun mevcut olduğunda, çift döndürmeden sonra bakiyesi 0 olur. Kötü çocuğun ve pivotun bakiyeleri dönüşün yönüne ve yeni öğenin ve kötü torunun öğesinin değerine bağlıdır. Bu durumlarda hem pivotun hem de kötü torunun bakiyelerini belirlemek için bu durum vaka bazında analiz edilebilir. Bir sonraki bölümde bakiyelerin nasıl hesaplandığını inceleyeceğiz.



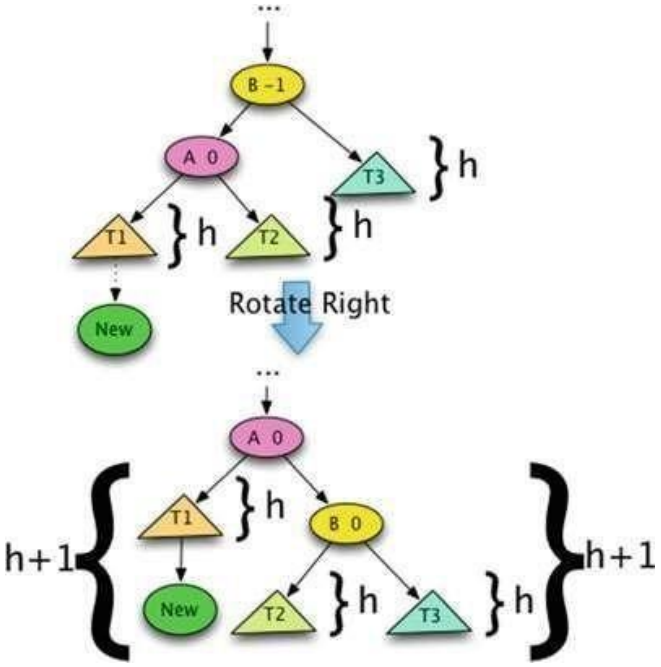
Şekil 10.6 AVL Ağacı Vaka 3B Adım 2 Uzağa Döndür

10.3.5 Rotasyonlar

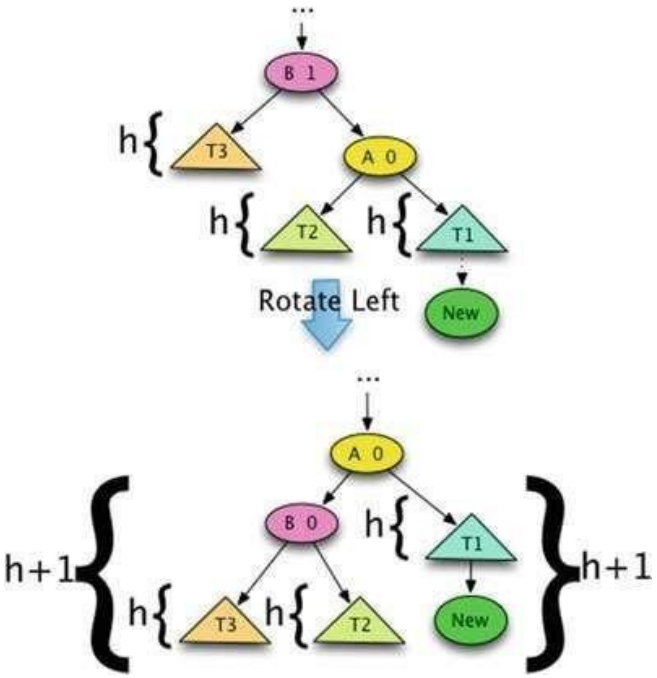
Her iki durum 1 ve 2, basitçe dengeleri ayarladıkları için uygulanması önemsizdir. Durum 3, uygulanması en zor olan durumdur. Bir alt ağacı döndürmek, ağaca yeni düğümler eklendikçe ağacın dengede kalmasını sağlayan işlemdir. Durum 3 A için ağaç, ağaca yeni bir düğümün ekleneceği bir durumdadır ve bu da ele alınması gereken bir dengesizliğe neden olur. İki olasılık vardır. Şekil 10.7 bu olası durumlardan ilkini göstermektedir. Yeni düğüm, pivot düğüme bağlı alt ağaç zaten sola ağırlıklıyken, kötü çocuk A'nın soluna eklenebilir. Pivot düğüm olan B, sıfır olmayan bir dengeye sahip en yakın atadır. B düğümünün yeni düğümü eklemeden önce -1 bakiyeye sahip olması için sağ alt ağacının yüksekliği ñ, sol alt ağacının yüksekliği ise *h* + 1 olmalıdır. Yeni düğümün kötü çocuğun alt ağacına eklenmesi, pivotun -2 bakiyeye sahip olmasına neden olur ki buna izin verilmez. Sağa döndürme sorunu çözer ve ikili arama ağacı özelliğini korur. *T2* alt ağacı rotasyonda hareket eder ancak rotasyondan önce *T2*'deki tüm değerler *B'den* küçük *ve A'dan* büyük olmalıdır.

Dengesizlik sağda olduğunda kötü çocuğun sağına bir değer eklemek, sola döndürme gerektiren benzer bir durumla sonuçlanır. Her iki rotasyonda da *A* ve *B* düğümlerinin dengesinin sıfır olduğuna dikkat edin. Bu yalnızca 3 A durumu için geçerlidir ve çift döndürme durumunda geçerli değildir (Şekil 10.8).

Yine, her iki düğümün, pivot ve kötü çocuğun dengesi, her iki yöndeki dönüşten sonra sıfır olur. Durum 3 A başka hiçbir koşul altında mümkün değildir. Durum 3B için sadece bir pivot ve kötü çocukla değil, aynı zamanda kötü bir torunla da ilgilenmeliyiz. Önceki bölümde açıklandığı gibi, bu durum bir



Şekil 10.7 AVL Ağacı Vaka 3A Sağa Döndürme

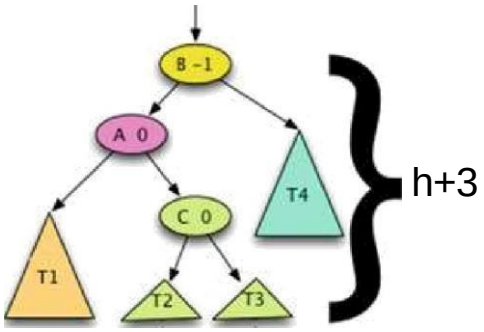


Şekil 10.8 AVL Ağacı Vaka 3A Sola Döndürme

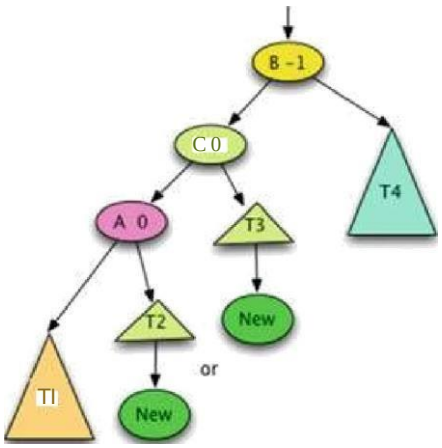
dengesizliğin tersi yönünde kötü bir çocuğun altına yeni bir değer ekler. Örneğin, Şekil 10.9'daki alt ağaç sola ağırlıklandırılmış ve yeni düğüm kötü çocuğun sağına eklenmiştir. Benzer bir durum, alt ağaç sağa doğru ağırlıklandırıldığında ve yeni düğüm kötü çocuğun sol alt ağacına yerleştirildiğinde meydana gelir. Her iki durum da gerçekleştiğinde, dengeyi yeniden sağlamak için çift döndürme gerekir.

Şekil 10.9 iki olası alt durum olduğunu göstermektedir. Aslında üç olası alt durum vardır. Kötü torun olmaması mümkündür. Bu durumda, yeni eklenen düğüm kötü torun tarafından işgal edilecek olan konuma yerleştirilecektir. Aksi takdirde yeni düğüm, Şekil 10.9'daki C düğümü olan kötü torunun soluna veya sağına yerleştirilebilir. Her iki durumda da Şekil 10.9'daki ilk adım kötü çocuk olan A düğümünü sola döndürmektir. Daha sonra pivot olan B düğümünü sağa döndürerek ağacın yeniden dengelenmesi tamamlanır.

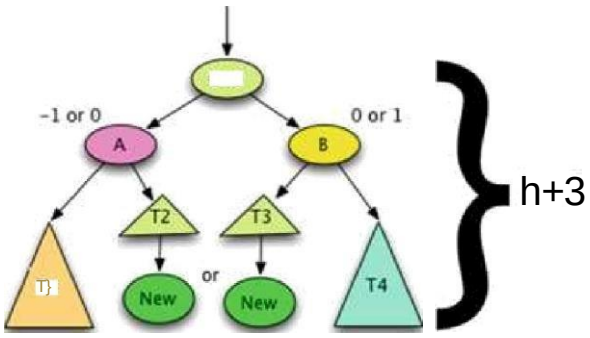
Yine, bu uygulamanın en zor kısmı her bir düğümün bakiyesinin hesaplanmasıdır. Kötü torun ve yeni pivot düğüm, Şekil 10.9'daki C düğümü, her zaman 0 bakiyeye sahiptir. Kötü torun yoksa, yeni pivot düğüm yeni eklenen değerdir. Kötü bir torun varsa ve yeni öğe kötü torunun öğesinden daha azsa, kötü çocuğun bakiyesi 0 ve eski pivotun bakiyesi 1'dir. Yeni öğe kötü torunun sağına eklenmişse, kötü çocuğun bakiyesi -1 ve eski pivotun bakiyesi 0'dır. Pivotun üzerindeki bakiyeler de dahil olmak üzere diğer tüm bakiyeler aynı kalır çünkü yeni değer eklenmeden önce ve yeni değer eklendikten sonra ağacın toplam yüksekliği değişmemiştir. Yine, Şekil 10.9'un ayna görüntüsünde benzer bir durum ortaya çıkar. Pivotun sağ alt ağacında bulunan ve halihazırda sağa doğru daha ağırlıklı olan kötü bir çocuğun sol alt ağacına yeni bir değer eklendiğinde, önce kötü çocukta sağa ve sonra pivotta sola dönerek çift döndürme gerekir.



Adım 1: A'da Sola Döndürün



5tep2: Raj Tam B'de



Şekil 10.9 AVL Ağacı Vaka 3B Adım 1 ve 2

10.3.6 AVL Ağacı Yinelemeli Ekleme

Özyinelemeli bir işlevi uygularken, bir sınıfın yöntemi yerine tek başına bir işlev olarak yazmak çok daha kolaydır. Bunun nedeni, tek başına bir yöntemin hiçbir şey (yani Python durumunda *None*) üzerinde çağrılabilmesiyken, bir yöntemin her zaman null olmayan bir *self* referansına sahip olması gerektiğidir. Özyinelemeli fonksiyonları metot olarak yazmak *self* için özel durumlara yol açar. Örneğin, *insert* yöntemi özyinelemeli olarak yazılırsa, özyinelemeli işlev olarak *insert*'i çağırırsa uygulaması daha kolaydır. Bölüm 10.2.1'deki *insert* fonksiyonu yüksekliği dengeli AVL ağaçları için yeterli olmayacaktır. Ekleme algoritması ağacın mevcut dengesini dikkate almalı ve önceki bölümde sunulan üç durumda tartıştığımız gibi dengeyi korumak için çalışmalıdır.

10.3.7 Yinelemeli Ekleme AVL Ağacı Sınıf Bildirimi

```
   sınıf AvLTree: sınıf
2      AVLNode:
3          def finite(se1f,item,ba1ance=0,1eft=None,right=None):
4              self.item   item
5              self.left   sol
6              self.sağ     doğru
7              self.balance   denge

9          # Other methods to be written here like __iter__ and
10         #    zepz     . flee Bölüm . 6

12     def   init (self,root=None):
13         self.root   kök
14
15     def insert(self, item):

17         def   insert(root,item):
IB             ... # Code to be written here

2£i            kök döndür
21
22         self.pivotFound   Yanlış
2?         self.root     insert(self.root,item)
24
25     def   repr (self):
2fi        return "AVLTree(" + repr(self.root) + ")"
27
2B      def   item(self):
20         return iter(self.root)
```

Özyinelemeli uygulamanın kabuğu Bölüm 10.3.7'de verilmiştir. Algoritma, Bölüm 10.2.1'de sunulan insert uygulaması ile birlikte yukarıda sunulan üç durumun bir kombinasyonu gibi ilerler. Özyinelemeli uygulamada *yol yığını* yoktur. Bunun yerine, çalışma zamanı yığını bu amaca hizmet eder. Bölüm 10.2.1'in 5. ve 6. satırları veya 7. ve 6. satırları arasında, kod geri dönüp özyinelemeli çağrılardan yukarı doğru ilerlerken ağacı yeniden dengeleme fırsatı vardır. Her çağrı geri döndüğünde, her düğümün dengeleri buna göre ayarlanabilir. Ayarlama

Dönmeden önceki bakiyeler, bölümde daha önce açıklandığı gibi birinci ve ikinci durumları uygular. Üçüncü durum, yeniden dengelemeden -2 veya 2 bakiye çıktığında tespit edilir. Bu durumda pivot bulunur ve durum 3'e göre yeniden dengeleme gerçekleşebilir.

Bir pivot bulunması halinde, pivotun üzerinde dengeleme yapılmasına gerek yoktur. Bu, Bölüm 10.3.7'deki kodun 22. satırında ilklendirilen *self.pivotFound* değişkeninin kullanımıdır. Bu bayrak, bulunması halinde pivot düğümün üzerinde herhangi bir dengeleme yapılmasını önlemek için *True* olarak ayarlanabilir. Dengeler, bölümün başlarında vaka bazında analizde açıklandığı gibi ayarlanır. En kötü durumda, pivot ve kötü çocuğun dengelerinin ayarlanması gerekecektir.

AVL ağaçlarına insert'in hem yinelemeli hem de özyinelemeli versiyonlarını uygulamak, yinelemeli versiyonda ele alınması gereken özel durumları göstermeye yardımcı olurken, özyinelemeli versiyon özel durumlara ihtiyaç duymayacaktır. Özyinelemeli versiyon, *insert*'in çalışma şekli nedeniyle özel durumların ele alınmasına ihtiyaç duymaz. Fonksiyona her zaman yeni öğenin ekleneceği bir ağacın kök düğümü verilir ve bu öğe eklendikten sonra ağacın kök düğümünü döndürür. Bu kadar düzenli bir şekilde çalıştığından, özel durum işleme gerekli değildir.

10.3.8 Yüksekliğe Karşı Dengenin Korunması

Bu bölümde sunulan iki uygulama, AVL ağaçları için özyinelemeli ve yinelemeli ekleme algoritmaları, her bir düğümün dengesini korumuştur. Alternatif olarak, her bir düğümün yüksekliği korunabilir. Bu durumda, bir yaprak düğümün yüksekliği 1'dir. Diğer herhangi bir düğümün yüksekliği 1 artı iki alt ağacının maksimum yüksekliğidir. Boş bir ağacın veya *Hiçbiri'nin yüksekliği* 0'dır.

10.3.9 Depolanmış Yükseklikli AVLNode

```
1  sınıf AVLNode:
2      def init (self,item,height=1,left=None,right=None):
3          self.item = item
4          self.left = sol
5          self.right = right
6          self.height = yükseklik
7
8      def balance(self):
9          return AVLTree.height(self.right) - AVLTree.height(self.left)
```

Dengeler yerine düğümlerin yüksekliği korunursa, yeni öğenin eklendiği konuma giden yoldaki tüm yükseklikler ağaca geri dönerken ayarlanmalıdır. Dengelerin aksine, pivot düğümde yüksekliklerin ayarlanmasını durdurmak mümkün değildir. Döndürmeden sonra pivot ve kötü çocuğun yüksekliği de yeniden hesaplanmalıdır çünkü döndürme onların yüksekliğini değiştirebilir. Yükseklikler aşağıdan yukarıya doğru hesaplandığından, pivot ve kötü çocuğun

yükseklikleri de dahil olmak üzere yol üzerindeki tüm yükseklikler aşağıdan yukarıya doğru yeniden hesaplanmalıdır. Bölüm 10.3.9'daki kod, düğümde köklenen ağacın yüksekliğini saklayan bir AVLNode'un kısmi bildirimini sağlar. Bu uygulamada, herhangi bir düğümün dengesi iki alt ağacın yüksekliklerinden hesaplanabilir.

10.3.10 AVL Ağacından Bir Öğeyi Silme

Bir AVL ağacından bir değerin silinmesi, Bölüm 6'daki programlama problemi 2'de açıklandığı gibi gerçekleştirilebilir. Ancak, son yaprak düğümün silinmesinden geri dönerken dengelerin ayarlanması gerekir. Bu, silme işlemi *yinelemeli olarak uygulanıyorsa bir yol yığını* tutularak ya da silme işleminin özyinelemeli bir uygulamasında özyinelemeli çağrılardan dönerken bakiyeleri veya yükseklikleri ayarlayarak yapılabilir.

Her iki durumda da, yol üzerindeki bir düğümün ayarlanmış dengesi 2'ye ulaştığında, ağacı yeniden dengelemek için sola döndürme gerekir. Yol üzerindeki bir düğümün ayarlanmış dengesi -2 ile sonuçlanırsa, sağa döndürme gerekir. Bu rotasyonlar yol boyunca ağacın köküne kadar kademeli olarak ilerleyebilir.

10.4 Yayvan Ağaçlar

AVL ağaçları her zaman dengelidir, çünkü her düğümün dengesi -1, 1 veya 0 olacak şekilde hesaplanır ve korunur. Dengeli oldukları için ①*(log n)* arama, ekleme ve silme süresini garanti ederler. AVL ağacı ikili bir arama ağacıdır, bu nedenle öğelerini sıralı olarak tutar ve ①*(n)* sürede en küçük öğeden en büyük öğeye doğru yinelemeye olanak tanır. Bu veri yapısının çok fazla dezavantajı yok gibi görünse de, splay ağaçları şeklinde olası bir iyileştirme vardır.

AVL ağaçlarına yöneltilen eleştirilerden biri de her düğümün kendi dengesini koruması gerektiğidir. Bu denge bakımı için gereken ekstra iş ve ekstra alan gereksiz olabilir. Peki ya bir ikili arama ağacı, dengesini her düğümde saklamadan da *yeterince iyi* koruyabilseydi? Her bir düğümün dengesini ya da yüksekliğini saklamak bellekteki verinin boyutunu artırır. Bellek boyutları daha küçükken bu daha büyük bir endişeydi. Ancak, fazladan bilgiyi korumak da fazladan zaman alır. Peki ya sadece toplam veri boyutunu azaltmakla kalmayıp ikili arama ağacının dengesini koruma işinin bir kısmını ortadan kaldırabilseydik?

AVL ağaçlarında yapılan iyileştirme, *uzamsal yerellik* kavramını içermektedir. Bu fikir, büyük veri setleriyle etkileşimin doğasını yansıtmaktadır. Büyük bir veri setine erişim genellikle yereldir, yani aynı veya birkaç veri parçasına kısa bir süre içinde birkaç kez erişilebilir ve daha sonra yeni değerler ekleyerek veya eski değerlere bakarak verilerin nispeten küçük bir alt kümesine erişilirken bir süre erişilmeyebilir. *Mekansal Yerellik,* verilerin nispeten küçük bir alt kümesine kısa bir süre içinde erişildiği anlamına gelir.

Bu bölümün başındaki örneğimiz açısından, çerezleri içeren bir ağaç, bir kullanıcı bir web sitesini ilk ziyaret ettiğinde atanan çerezlere sahip olabilir. Web sitesine gelen bir kullanıcı bir süre etkileşimde bulunacak ve daha sonra muhtemelen bir daha geri gelmemek üzere ayrılacaktır. Web sunucusuyla etkileşime giren kullanıcı kümesi zaman içinde değişecektir, ancak ağaçtaki toplam giriş sayısına kıyasla her zaman nispeten küçük bir alt kümedir. Son kullanıcıların çerezlerini ağacın tepesine daha yakın bir yerde saklayabilseydik, ağaca yeni bir değer ekleme ve arama süresini iyileştirebilirdik. Karmaşıklık artmayacaktır. Bir öğe eklemek hala

①*(log n)* zaman alır. Ancak bir öğeyi eklemek veya aramak için gereken toplam süre biraz iyileşebilir. Bu, bir yayılma ağacı için motivasyondur.

Bir yayma ağacında, her ekleme veya arama, eklenen veya bakılan değeri *yayma* adı verilen bir işlemle ağacın köküne taşır. Bir değer silinirken, üst değer ağacın köküne kaydırılabilir. Bir yayma ağacı hala bir ikili arama ağacıdır. Splay ağaçları genellikle dengeli kalır ancak AVL ağacının aksine, bir splay ağacı herhangi bir denge veya yükseklik bilgisi içermez. Bir düğümü köke yaymak, AVL ağaçlarının dönüşlerine çok benzeyen bir dizi dönüşü içerir, ancak küçük bir farkla.

Splay ağaçları verilerdeki *mekansal yerellikten* faydalanmak üzere tasarlanmış olsa da, iyi performans göstermek için mekansal yerellikten bağımsız olduklarını belirtmek ilginçtir. Splay ağaçları, tamamen rastgele veri setlerinde pratikte AVL ağaçları kadar iyi veya daha iyi çalışır.

Yayvan ağaçlarla ilgili ilginç olan birkaç şey vardır.

- İlk olarak, yayma işlemi alt ağaçların yüksekliği hakkında denge veya başka herhangi bir bilgi gerektirmez. İkili arama ağacı yapısı yeterince iyidir.
- Yayvan ağaçlar her zaman mükemmel şekilde dengeli kalmaz. Ancak nispeten dengeli kaldıkları için ekleme, arama ve silme işlemleri için ①*(log n)* ortalama durum karmaşıklığı elde etmeye yetecek kadar dengelidirler. Yeterince iyi oldukları fikri, Bölüm 2'nin ilerleyen kısımlarında ve bu bölümün ilerleyen kısımlarında tartışılan *amorti edilmiş karmaşıklık olarak* adlandırılan şeyin temelini oluşturur.
- Splaying'in uygulanması nispeten basittir.

Bu metinde iki aşağıdan yukarıya yayılan ağaç uygulamasını ele alıyoruz. Yayvan ağaçlar yinelemeli ya da özyinelemeli olarak uygulanabilir ve biz her iki uygulamayı da inceleyeceğiz. Bölüm 6'da ikili arama ağacı ekleme özyinelemeli olarak uygulanmıştır. Eğer yayma özyinelemeli olarak yapılacaksa, yayma insert fonksiyonunun bir parçası olabilir. Yinelemeli olarak yazılırsa, yayma işleminde bir yığın kullanılabilir. İlerleyen bölümler hem yinelemeli hem de özyinelemeli uygulamaları kapsamaktadır. Ancak önce splaying işleminde kullanılan rotasyonları inceleyeceğiz.

10.4.1 Yaylanma Dönüşleri

Bir değer her eklendiğinde veya arandığında, bu değeri içeren düğüm bir dizi döndürme işlemi ile en üste yayılır. AVL ağaçlarından farklı olarak, bir splay ağacı bir düğümü, eğer bir büyükbaba varsa, büyükbabasının seviyesine taşımak için çift döndürme kullanır. Bir dizi çift döndürme işlemi sayesinde düğüm ya köke ya da kökün çocuğuna ulaşacaktır. Eğer yayılan düğüm kökün çocuğuna ulaşırsa, onu köke getirmek için tek bir döndürme kullanılır.

Tekli döndürme fonksiyonları genellikle *zig* veya *zag olarak* etiketlenirken, çiftli döndürmeler yayılan düğümün hareket yönüne bağlı olarak *zig-zig* veya *zig-zag* işlemleri olarak adlandırılır. Bazen düğüm zig-zag hareketi ile hareket ederken bazen de zig-zig hareketi ile hareket eder.

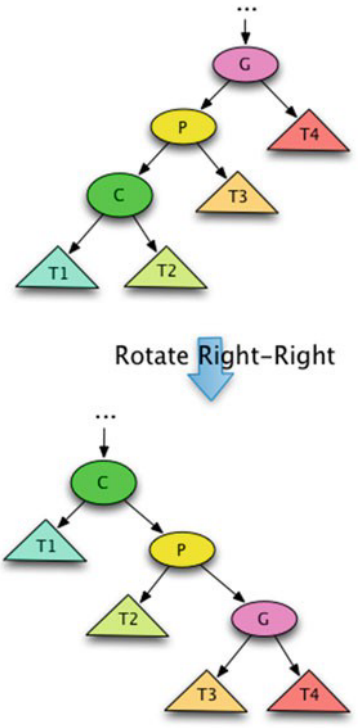
Yayılma, bir değer bir yayılma ağacına eklendiğinde, arandığında veya silindiğinde gerçekleşir. Bir değer arandığında ya aranan değer en üste yayılır ya

değer ağaçta bulunmazsa değerin ebeveyni olabilir. Ağaçtan silme işlemi, Bölüm 6'daki problem 2'de açıklandığı gibi diğer ikili arama ağaçlarından silme işlemi gibi uygulanabilir. İkili arama ağacından bir değer silindiğinde, silinen düğümün ebeveyni ağacın köküne yayılır.

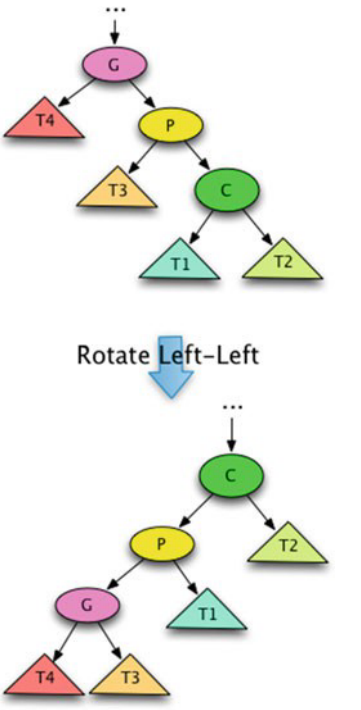
Şekil 10.14'teki örnek, yeşil düğümlerin bir yayma ağacına yerleştirilmesinden kaynaklanan yayma işlemlerini göstermektedir. 30 eklendiğinde, ağacın ikinci versiyonunda (kırmızı düğümler) görüldüğü gibi ağacın köküne doğru yayılır. 5 eklendiğinde, o da köke doğru yayılır. 5'in köke taşınması, çift sağa döndürme adı verilen bir zig-zig döndürme ile gerçekleştirilir. 8'in köke yayılması, sağa-sola döndürme adı verilen bir zig-zag döndürmenin sonucudur. 42 köke doğru yayıldığında, çift sol dönüş ve ardından tek bir sol dönüş gerçekleşir.

15'in köke doğru yayılması, çift sağ dönüş ve ardından sol-sağ dönüş yapılarak gerçekleştirilir. Çift sağa döndürme genellikle çift sola döndürme gibi zig-zig döndürme olarak adlandırılır. Sol-sağ ve sağ-sol rotasyonlar genellikle zig-zag rotasyonlar olarak adlandırılır. Her durumda sonuç, yeni eklenen düğümün veya bakılan düğümün ağacın köküne doğru yayılmasıdır.

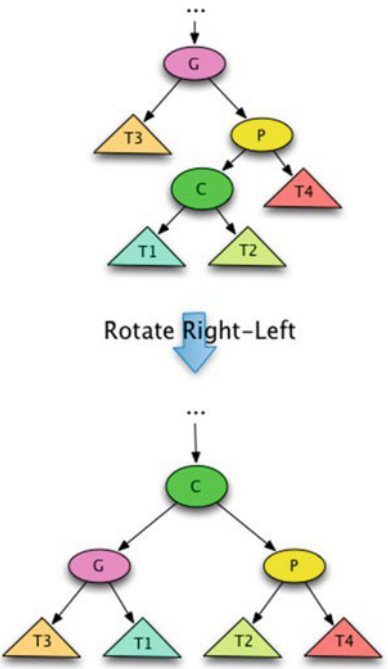
Şekil 10.10, 10.11, 10.12 ve 10.13 bu yayılma işlemlerini göstermektedir. Şekil 10.12 ve 10.13, yayılma ağaçlarının neden bu kadar iyi çalıştığına dair sezgisel bir anlayış sunmaktadır



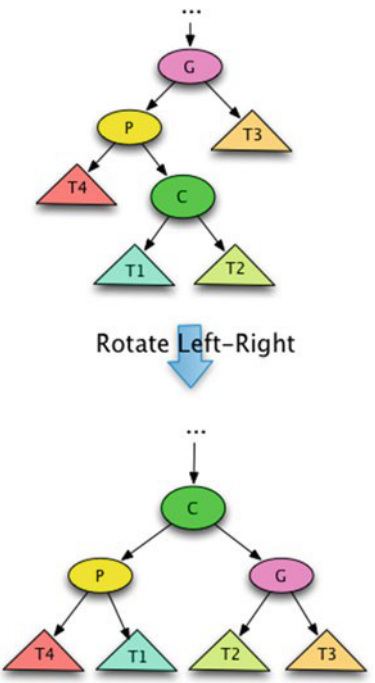
Şek. 10.10 Yayvan Ağaç Çift-Sağ Döndürme



Şek. 10.11 Yayvan Ağaç Çift-Sol Döndürme



Şek. 10.12 Yayvan Ağaç Sağ-Sol Döndürme



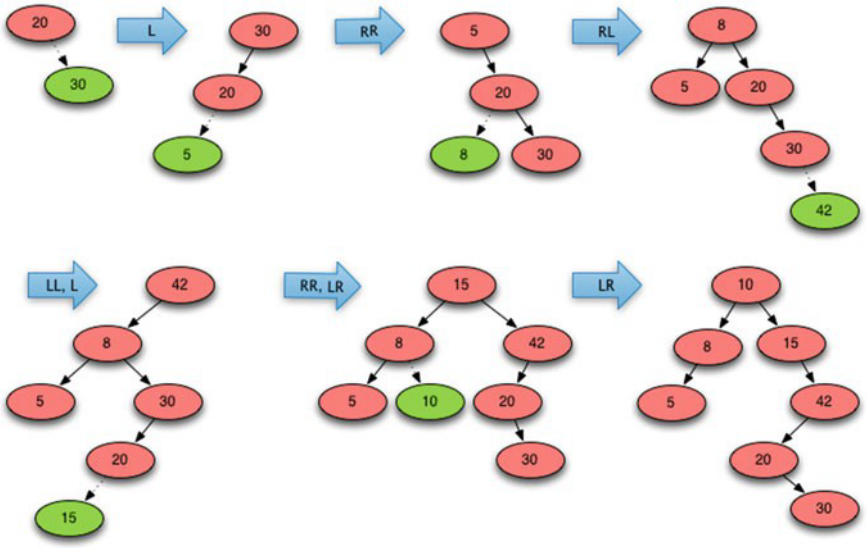
Şek. 10.13 Yayvan Ağaç Sol-Sağ Döndürme

yapın. Şekil 10.12 ve 10.13'te gösterilen döndürme işlemlerinden sonra, çocukta köklenen alt ağaç bu döndürmelerden öncesine göre daha dengeli görünür.

Sola-sağa döndürme yapmanın, sola döndürme ve ardından sağa döndürme yapmakla aynı şey olmadığına dikkat edin. Sola-sağa döndürme farklı bir sonuç verir. Aynı şekilde, splay sağ-sol döndürme de sağ ve ardından sol döndürmeden farklı bir sonuç verir. Yayvan zig-zag döndürmeler, ağacı dengelemeye yardımcı olmak için bu şekilde tasarlanmıştır. Şekil 10.12 ve 10.13, döndürmeden önce biraz dengesiz olabilecek ağaçların sağ-sol döndürme veya sol-sağ döndürme ile çok daha iyi bir dengeye getirildiğini göstermektedir.

10.5 Yinelemeli Yayma

Bir değer her eklendiğinde veya arandığında, önceki bölümde açıklandığı gibi bir dizi döndürme işlemiyle splay ağacının köküne yayılır. Çift döndürme işlemleri değeri ya ağacın köküne ya da kökünün çocuğuna taşıyacaktır. Çift döndürme işlemi yeni eklenen değerin ağacın kökünün çocuğunda olmasıyla sonuçlanırsa, Şekil 10.14'te 30 ve 15'in yayma ağacına eklendiğinde gösterildiği gibi yeni eklenen değeri köke taşımak için tek bir döndürme kullanılır.



Şekil 10.14 Yayvan Ağaç Örneği

İkili arama ağacına özyineleme olmadan yeni bir değer eklemek bir while döngüsü kullanılarak mümkündür. While döngüsü ağacın kökünden yeni düğümün ebeveyni olacak yaprak düğüme doğru hareket eder, bu noktada döngü sonlanır, yeni düğüm oluşturulur ve ebeveyn yeni çocuğuna bağlanır.

Yeni düğüm eklendikten sonra, üste doğru yayılması gerekir. Yaymak için ağaç boyunca yeni eklenen düğüme giden yolun bilinmesi gerekir. Bu yol bir yığın kullanılarak kaydedilebilir. Ekleme döngüsü ağaçtaki başka bir düğümden geçerken, bu düğüm yığının üzerine itilir. Sonuç olarak, kökten yeni çocuğa giden yol üzerindeki tüm düğümler bu yol yığınına itilir.

Son olarak, bu yol yığını boşaltılarak yayılma gerçekleşebilir. Önce çocuk yığından çıkarılır. Ardından, yığının geri kalanı aşağıdaki gibi boşaltılır.

- Yığında iki düğüm daha varsa bunlar yeni eklenen düğümün ebeveyni ve büyük ebeveyni olur. Bu durumda, yeni döndürülen alt ağacın kökünün yeni eklenen düğüm olmasıyla sonuçlanan bir çift döndürme gerçekleştirilebilir. Hangi çift döndürmenin gerekli olduğu büyük ebeveyn, ebeveyn ve çocuk değerlerinden belirlenebilir.
- Yığında yalnızca bir düğüm kalırsa, bu yeni eklenen düğümün ebeveynidir. Tek bir döndürme yeni eklenen düğümü yayılma ağacının köküne getirecektir.

Burada açıklanan şekilde splay uygulamak, ağaçta bir değer ararken, bulunsa da bulunmasa da iyi çalışır. Bir değer bulunduğunda yol yığınına eklenecektir. Bir değer bulunamadığında, üst öğe en üste yayılmalıdır; bu, aranan değer bulunamadığında doğal olarak gerçekleşir çünkü yayma işlemi gerçekleştirildiğinde üst öğe yol yığınının en üstünde kalacaktır.

Bir splay ağacından bir düğümü silmenin bir yöntemi, tıpkı ikili arama ağacında olduğu gibi silme işlemiyle gerçekleştirilir. Silinecek düğümün sıfır veya bir çocuğu varsa, düğümü silmek önemsizdir. Silinecek düğümün iki çocuğu varsa, sağ alt ağacındaki en soldaki değer silinecek düğümdeki değerin yerini alabilir ve en soldaki değer sağ alt ağaçtan silinebilir. Silinen düğümün ebeveyni ağacın tepesine yayılır.

Başka bir silme yöntemi, silinen düğümün önce ağacın köküne yayılmasını gerektirir. Ardından sol alt ağacın en sağdaki değeri köke yayılır. Sol alt ağaç yayıldıktan sonra, kök düğümünün sağ alt ağacı boştur ve orijinal sağ alt ağaç buna eklenebilir. Orijinal sol alt ağaç, yeni oluşturulan yayma ağacının kökü olur.

10.6 Yinelemeli Yayma

Splaying'in özyinelemeli olarak uygulanması, ikili arama ağaçları üzerindeki özyinelemeli ekleme işlemini takip eder. Splaying bu özyinelemeli insert fonksiyonu ile birleştirilir. Özyinelemeli ekleme ağaçtaki yolu takip ederken "R" ve "L" şeklinde bir döndürme dizesi oluşturur. Yeni öğe mevcut kök düğümün sağına eklenirse, yeni eklenen düğümü ağaçta yukarı doğru yaymak için bir sola döndürme gerekir ve döndürme dizesine bir "L" eklenir. Aksi takdirde, sağa döndürme gerekir ve döndürme dizesine bir "R" eklenir.

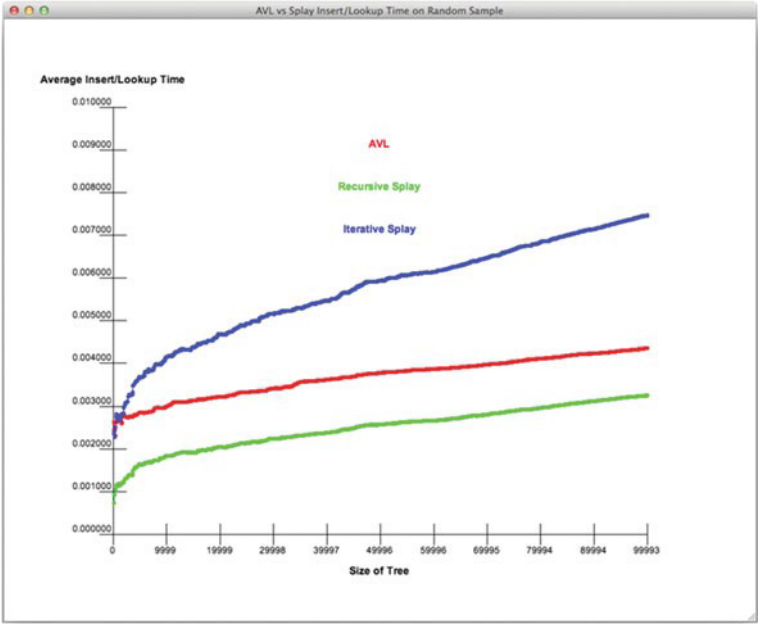
Özyinelemeli insert fonksiyonu geri döndüğünde, yeni eklenen düğüme giden yol geri dönen fonksiyon tarafından yeniden izlenir. Rotate dizesindeki son iki karakter hangi çift rotasyonun gerekli olduğunu belirler. Bir sözlük veya hash tablosu "RR", "RL", "LR" ve "LL "yi uygun döndürme fonksiyonlarıyla eşleştirir. Hash tablosu araması uygun döndürmeyi çağırmak için kullanılır ve döndürme dizesi kesilir (veya "R" ve "L "nin döndürme dizesine ne zaman eklendiğine bağlı olarak boş dizeye yeniden başlatılır). Özyinelemeli ekleme tamamlandığında, gerekli tekli döndürme rotate dizesine kaydedilir ve gerçekleştirilebilir.

Bu şekilde bir döndürme dizesi ve hash tablosu kullanarak kaydırma uygulamanın, yukarıda açıklanan yinelemeli algoritmaya kıyasla gerekli döndürmeleri belirlemek için koşullu ifadelerin yaklaşık yarısını gerektirdiğine dikkat edilmelidir. Yeni bir düğüm eklerken yol, yol üzerindeki her düğüme eklenecek değerin ağaçtaki konumuyla karşılaştırılmasıyla belirlenmelidir. Yukarıdaki yinelemeli tanımda, yol üzerindeki değerler kaydırma sırasında tekrar karşılaştırılır. Bu özyinelemeli tanımda, yeni öğe yoldaki her öğeyle yalnızca bir kez karşılaştırılır. Bu, bölümün ilerleyen kısımlarında gösterildiği gibi performans üzerinde bir etkiye sahiptir.

Bu özyinelemeli uygulamayı kullanarak bir değeri aramak, bulunan değeri ya da bulunamazsa üstünü ağacın köküne yayarak eklemekle benzer şekilde çalışır. Bir değerin silinmesi yine özyinelemeli olarak, önce silinecek değere bakılarak ağacın köküne yayılması ve ardından önceki bölümde açıklanan kök kaldırma yönteminin gerçekleştirilmesiyle yapılabilir.

10.7 Performans Analizi

En kötü durumda bir yayvan ağaç her bir arama, ekleme ve silme işlemi için ①*(n)* karmaşıklığa neden olan bir çubuk haline gelebilirken AVL ağaçları arama, ekleme ve silme işlemleri için ①*(log n)* zaman garanti eder. AVL ağaçlarının daha iyi performansa sahip olabileceği görülmektedir. Ancak, pratikte durum böyle görünmemektedir. Önceden oluşturulmuş bir veri kümesi kullanılarak yapılan bir deneyde 100.000'e yakın ekleme ve 900.000 rastgele arama işlemi gerçekleştirilmiştir. Ekleme ve arama işlemleri veri kümesinde tanımlanmış ve aranan tüm değerler ağaçta bulunmuştur. Ortalama birleşik ekleme ve arama süreleri bir AVL ağacı, yinelemeli olarak uygulanan bir splay ağacı ve splay ağacı ekleme ve aramanın özyinelemeli uygulaması için Şekil 10.15'te kaydedilmiştir. Sonuçlar, özyinelemeli splay tree uygulamasının rastgele bir değer kümesinde AVL tree uygulamasından daha iyi performans gösterdiğini ortaya koymaktadır. Deneyler, splay ağaçlarının ekleme ve arama işlemleri için pratikte ①*(log n)* karmaşıklık sergilediğini göstermektedir. Şekil 10.13 ve 10.12'de, yayılan ağaçların özel çift dönüşleri sayesinde dengeyi nasıl koruduğuna dair sezgisel bir anlayış elde ettik. Ancak, çift dönüşlerin ağacı daha dengeli hale getirdiğini söylemek çok ikna edici bir argüman değildir. Bu fikir, *amorti edilmiş karmaşıklık* kullanılarak resmileştirilmiştir. İlk olarak Bölüm 2'de karşılaşılan amortisman, bir giderin tamamının bir yıl içinde giderleştirilmesi yerine birkaç yıla yayılması durumunda kullanılan bir muhasebe terimidir. Aynı ilke, bir Splay Tree'de bir değer bulma veya ekleme masrafına da uygulanabilir. Eksiksiz



Şekil 10.15 Ortalama Ekleme/Arama Süresi

Bunun analizi duruma göre yapılır ve bu metinde yer almaz ancak çevrimiçi metinlerde bulunabilir. Bu kanıtlar, splay ağaçlarının rastgele erişilen veriler üzerinde gerçekten de AVL ağaçları kadar verimli çalıştığını göstermektedir. Buna ek olarak, bir değer eklenirken veya aranırken kullanılan yayma işlemi, verideki *uzamsal yerellikten* faydalanır. Sık sık aranan veri değerleri, gelecekte daha verimli bir şekilde aranmak üzere ağacın tepesine doğru ilerleyecektir. Verilerde mevcutsa uzamsal yerellikten yararlanmak kesinlikle arzu edilir olsa da, yayvan ağaç ekleme ve arama işlemlerinin genel hesaplama karmaşıklığını iyileştirmez.

Ancak, bu durum rastgele eklenen ve bakılan değerler üzerinde ortalama durumda gerçekleşmez. Aslında, önceki bölümde sunulan splay ağaçlarının özyinelemeli uygulaması, rastgele dağıtılmış bir değerler kümesinde ①*(log n)* ortalama ekleme ve arama süresi sergiler ve rastgele bir örneklemde AVL ağacı uygulamasından daha iyi performans gösterir.

Bir AVL ağacı üzerindeki ekleme, arama ve silme işlemleri ①*(log n)* zamanda tamamlanabilir. Ortalama durumda bu durum splay ağaçları için de geçerlidir. Bir AVL veya splay ağacının çaprazlanması ①*(n) zamanda* çalışır ve öğelerini artan veya azalan sırada verir (yineleyicinin nasıl yazıldığına bağlı olarak). Quicksort algoritması bir listenin öğelerini verimli bir şekilde sıralayabilirken, AVL ve splay ağaçları, öğelerinin sıralamasını korurken birçok ekleme ve silme işlemine izin veren veri yapılarıdır. Arama, silme ve ekleme işlemlerini verimli bir şekilde uygulayan ve aynı zamanda değer dizisinin artan veya azalan sırada yinelenmesine izin veren bir veri yapısı gerekiyorsa, AVL veya splay ağacı pratik bir seçim olabilir. AVL ağaçlarının avantajı, verimli arama, ekleme ve silme karmaşıklığını garanti ederken öğelerin sıralamasını koruma yeteneklerinde yatmaktadır. Splay ağaçları neredeyse tüm durumlarda aynı şekilde çalışır ve bu bölümde açıklanan özyinelemeli splay ağacı uygulaması rastgele veri kümelerinde AVL Ağacı uygulamasından bile daha iyi performans gösterir. AVL ağacı ile özyinelemeli splay ağacı performans sayıları arasındaki performans farkı, AVL ağacında dengeyi açıkça korumak ile splay ağacında *yeterince iyi denge* elde etmek arasındaki farktır.

10.8 Bölüm Özeti

Bu bölümde yükseklik dengeli AVL ağaçlarının ve yayvan ağaçların çeşitli uygulamaları sunulmuştur. Özyinelemeli ve yinelemeli ekleme algoritmaları sunulmuştur. Hem dengeyi koruyan hem de yüksekliği koruyan AVL düğümleri ele alınmıştır. Hem AVL hem de yayvan ağaçlar için özyinelemeli ekleme algoritmaları, çok fazla özel durum olmadan çok temiz bir kodla sonuçlanırken, yinelemeli sürümler bazı koşulları ele almak için birkaç if deyimine daha ihtiyaç duyar. Bazı durumlarda yinelemeli versiyon özyinelemeli versiyondan biraz daha verimli olabilir, çünkü herhangi bir dilde fonksiyon çağrıları ile ilişkili bir maliyet vardır, ancak bu bölümde yapılan deneylerden elde edilen deneysel sonuçlar, Python'da yazıldığında özyinelemeli uygulamaların çok verimli çalıştığını göstermektedir.

10.9 İnceleme Soruları

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

1. AVL ağacındaki bir düğümün dengesi nedir?
2. Bir düğümün dengesi yüksekliği ile nasıl ilişkilidir?
3. Bir AVL ağacı bir düğümün dengesini nasıl kullanır?
4. Pivot düğüm nedir?
5. AVL ağaçlarıyla ilişkili olarak kötü çocuk nedir?
6. Yol yığını nedir ve ne zaman gereklidir?
7. Sağa döndürme yaptıktan sonra, başlangıçta pivotta köklenmiş olan alt ağaçta pivot düğüm ve kötü çocuk nerede?
8. Durum 3 için kod yürütüldükten sonra bir alt ağacın kökünün bakiyesi neden her zaman 0 olur?
9. Vaka 3 için iki alt durumda, her bir alt durumda algoritma yürütüldükten sonra hangi düğüm pivotta köklenen alt ağacın kök düğümü olur?
10. AVL ağaç ekleme algoritması neden her zaman ①*(log n)* zamanda tamamlanır? Bir değerin eklenmesi ile ilgili üç durumun her biri için cevabınızı gerekçelendirmek için durum analizi yapın.
11. Özyinelemeli ekleme yayma ağacı uygulamasında döndürme dizesinin amacı nedir?
12. Neden özyinelemeli yayvan ağaç ekleme ve arama uygulaması AVL ağaç uygulamasından daha hızlı çalışıyor gibi görünüyor?

10.10 Programlama Problemleri

1. Her düğümde dengeleri koruyan ve eklemeyi yinelemeli olarak uygulayan bir AVL ağacı uygulaması yazın. Programınızı rastgele oluşturulmuş bazı veriler üzerinde kapsamlı bir şekilde test etmek için bir test programı yazın.
2. Her düğümde dengeleri koruyan ve eklemeyi özyinelemeli olarak uygulayan bir AVL ağacı uygulaması yazın. Programınızı rastgele oluşturulmuş bazı veriler üzerinde kapsamlı bir şekilde test etmek için bir test programı yazın.
3. Her düğümde yükseklikleri koruyan ve özyinelemeli olarak ekleme yapan bir AVL ağacı uygulaması yazın. Programınızı rastgele oluşturulmuş bazı veriler üzerinde kapsamlı bir şekilde test etmek için bir test programı yazın.
4. Her düğümde yükseklikleri koruyan ve yinelemeli olarak ekleme yapan bir AVL ağacı uygulaması yazın. Programınızı rastgele oluşturulmuş bazı veriler üzerinde kapsamlı bir şekilde test etmek için bir test programı yazın.
5. Programlama problemi 3'ü tamamlayın. Ardından AVL Ağaçları için silme işlemini uygulayın. Son olarak, veri yapınızı kapsamlı bir şekilde test etmek için bir test programı yazın. Değerler ağacınıza eklendikçe ve silindikçe, tüm yükseklikleri ve ağaçtaki tüm değerlerin sıralamasını doğru şekilde koruduğundan emin olmak için kodunuzu test etmelisiniz.

6. Bu bölümdeki 1-4 programlama problemlerinden ikisini uygulayın ve ardından rastgele bir tamsayı listesi oluşturan bir test programı yazın. Değerleri ilk uygulamaya ekleme zamanı ve ardından her bir değeri ikinci uygulamaya ekleme zamanı. Tüm zamanları ikinci bölümdeki PlotData.py programının ihtiyaç duyduğu XML formatında kaydedin. Göreceli verimliliklerini karşılaştırmak için iki algoritmanın zamanlamasını çizin.
7. Özyinelemeli ekleme ve arama işlevlerine sahip bir splay ağacı uygulaması yazınız. Her bir düğümün yüksekliğinin korunduğu yinelemeli veya özyinelemeli bir AVL ağacı uygulayın. Ağaçların aynı değerler listesinden oluşturulduğu bir test gerçekleştirin. Değer listesini oluşturduğunuzda, yinelenen değerler bir arama olarak değerlendirilmelidir. Veri dosyasını bir *L* veya *I* ile ve ardından bir *arama* veya *ekleme* işleminin gerçekleştirilmesi gerektiğini belirten bir değerle yazın. Performans sonuçlarınızı karşılaştırmak için PlotData.py programı tarafından kullanılan formatta bir XML dosyası oluşturun.
8. Özyinelemeli ekleme ve arama işlevlerine sahip bir yayvan ağaç uygulaması yazın. Bu bölümde ayrıntıları verilen diğer dengeli ikili ağaç uygulamalarından biriyle karşılaştırın. Ağaçların aynı değer listesinden oluşturulduğu bir test çalıştırın. Değer listesini oluşturduğunuzda, yinelenen değerler bir arama olarak değerlendirilmelidir. Veri dosyasını bir *L* veya *I ile ve* ardından bir *arama* veya *ekleme* işleminin gerçekleştirilmesi gerektiğini belirten bir değerle yazın. Performans sonuçlarınızı karşılaştırmak için PlotData.py programı tarafından kullanılan formatta bir XML dosyası oluşturun.