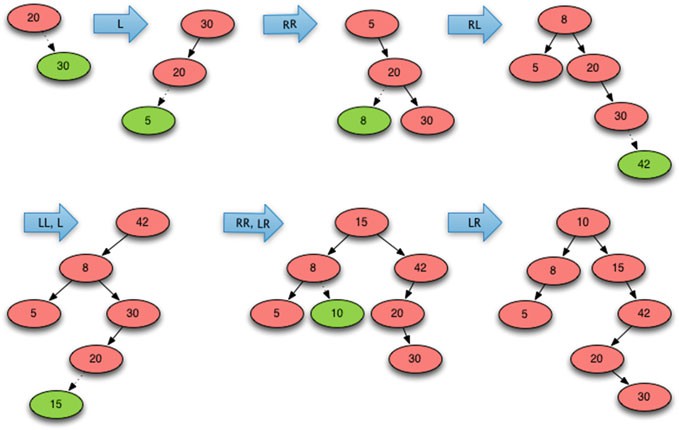
# Yinelemeli Yayma (Iterative Splaying)

Her yeni değer eklendiğinde veya arandığında, önceki bölümde açıklanan bir dizi rotasyon işlemiyle Splay Tree’nin köküne taşınır. Çift rotasyon işlemleri, değeri ya doğrudan ağacın köküne ya da kökün çocuğu konumuna getirir. Çift rotasyon işlemi yeni eklenen değerin ağacın kökünün çocuğunda olmasıyla sonuçlanırsa, Şekil 10.14'te 30 ve 15'in Splay Tree’ye eklendiğinde gösterildiği gibi yeni eklenen değeri köke taşımak için tek bir döndürme kullanılır.



**. 10.14** Splay Tree Örneği

Binary search tree’ye 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 Splay Tree’ nin köküne getirecektir.

Burada açıklanan şekilde yayılma 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 Tree’den bir düğümü silmenin bir yöntemi, tıpkı binary search tree’ de olduğu gibi silme işlemiyle gerçekleştirilir.Eğer silinecek düğümün hiç çocuğu yoksa veya yalnızca bir çocuğu varsa, düğümün silinmesi önemsizdir. Eğer 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 boşalır ve orijinal sağ alt ağaç buna eklenebilir. Orijinal sol alt ağaç, yeni oluşturulan Splay Tree’nin kökü olur.

# ( Özyinelemeli Yayma )Recursive Splaying

Yayılmanın özyinelemeli olarak uygulanması, binary search trees üzerindeki özyinelemeli ekleme işlemini takip eder. Yayılma bu recursive insert (özyinelemeli ekleme) fonksiyonu ile birleştirilir. Özyinelemeli ekleme işlemi, ağaç içinde aşağı doğru ilerlerken **"R" ve "L"** karakterlerinden oluşan bir döndürme dizisi oluşturur. Eğer yeni öğe, mevcut kök düğümün sağ tarafına eklenirse, yeni eklenen düğümü ağacın yukarısına taşımak için bir sola döndürme gereklidir ve döndürme dizisine bir **"L"** eklenir. Diğer bir durumda, sağa döndürme işlemi gerekli olacaktır ve döndürme dizisine bir 'R' eklenir..

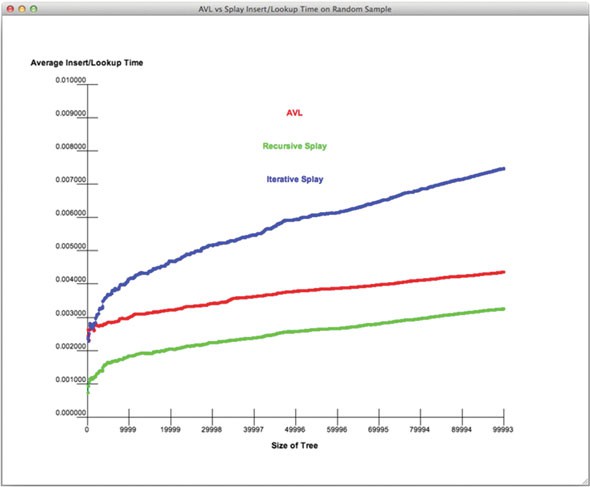
Özyinelemeli ekleme işlevi geri dönerken, yeni eklenen düğüme giden yol, geri dönen işlev tarafından izlenir. Döndürme dizisindeki son iki karakter, hangi çift döndürmenin gerektiğini belirler. Bir sözlük veya hash tablosu, "RR", "RL", "LR" ve "LL" değerlerini uygun döndürme işlevlerine eşlemeye yardımcı olur. Hash tablosu araması, uygun döndürmeyi çağırmak için kullanılır ve döndürme dizisi kısaltılır (veya "R" ve "L" döndürme dizisine eklendiğine bağlı olarak boş dizeye yeniden başlatılır). Özyinelemeli ekleme tamamlandığında, gerekli tekli döndürme rotate dizesine kaydedilir ve gerçekleştirilebilir.

Döndürme dizisi ve hash tablosu kullanarak yayılma uygulamak, yukarıda açıklanan yinelemeli algoritmaya kıyasla gerekli döndürmeleri belirlemek için yarısı kadar koşullu ifadeye ihtiyaç duyar. Yeni bir düğüm eklerken yol, üzerindeki her düğüme eklenecek değerin ağaçtaki konumuyla karşılaştırılmasıyla belirlenmelidir. Yukarıdaki yinelemeli tanımda, yol üzerindeki değerler yayılma işlemi sırasında tekrar karşılaştırılır. Bu özyinelemeli tanımda, yeni öğe yalnızca yol üzerindeki her öğe ile bir kez karşılaştırılır. Bu durumun performans üzerindeki etkisi, bölümün ilerleyen kısımlarında gösterilecektir.

Bu özyinelemeli uygulama ile bir değeri aramak, bulunan değerin veya bulunamadığında onun ebeveyninin ağacın köküne yayılmasını sağlamak için ekleme işlemine benzer şekilde çalışır. Bir değeri silmek de yinelemeli olarak gerçekleştirilebilir; önce silinecek değeri arayıp bu değerin ağacın köküne yayılmasını sağladıktan sonra, önceki bölümde açıklanan kök kaldırma yöntemini uygulamak gerekir.

# Performans Analizi

En kötü durumda bir Splay Tree her bir arama, ekleme ve silme işlemi için G(n) karmaşıklığına neden olan bir çubuk haline gelebilirken AVL ağaçları arama, ekleme ve silme işlemleri için G(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 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 Tree, ekleme ve aramanın özyinelemeli uygulaması için Şekil 10.15'te kaydedilmiştir. Sonuçlar, özyinelemeli Splay Tree uygulamasının rastgele bir değer kümesinde AVL ağacı uygulamasından daha iyi performans gösterdiğini ortaya koymaktadır. Deney, Splay Tree’lerinin ekleme ve arama işlemleri için pratikte de G(log n) karmaşıklığı sergilediğini göstermektedir. Şekil 10.13 ve 10.12'de Splay Tree’lerinin 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 prensip, bir Splay Tree'de bir değer bulma veya ekleme masrafına da uygulanabilir.



**Şekil. 10.15** Ortalama Ekleme/Arama Süresi

Bunun analizi duruma göre yapılır ve sonuçları bu metinde yer almamakla birlikte çevrimiçi metinlerde bulunabilir. Bu kanıtlar, Splay Tree’lerinin rastgele erişilen veriler üzerinde gerçekten de AVL ağaçları kadar verimli çalıştığını göstermektedir. Buna ek olarak, bir değeri eklerken veya ararken 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, Splay Tree ekleme ve arama işlemlerinin genel hesaplama karmaşıklığını iyileştirmez.

Öte yandan, rastgele eklenen ve bakılan değerler üzerinde ortalama durumda bu gerçekleşmez. Aslında, önceki bölümde sunulan Splay Tree’lerinin özyinelemeli uygulaması, rastgele dağıtılmış bir değerler kümesinde G(log n) ortalama ekleme ve arama süresi sergiler, AVL ağaç uygulamasından rastgele bir örneklemde daha iyi performans gösterir.

Bir AVL ağacı üzerindeki ekleme, arama ve silme işlemleri G(log n) zamanında tamamlanabilir. Ortalama durumda bu durum Splay Tree için de geçerlidir. Bir AVL veya Splay Tree’lerinin çaprazlanması G(n) zamanında gerçekleşir ve öğelerini artan veya azalan sırada verir (yineleyicinin nasıl yazıldığına bağlı olarak). Quicksort algoritması bir listenin öğelerini verimli bir şekilde sıralayabilirken, AVL ve Splay Tree’leri öğelerinin sıralamasını koruyarak 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 Tree pratik bir seçim olabilir. AVL ağaçlarının avantajı, verimli arama, ekleme ve silme işlemlerinin karmaşıklığını garanti ederken öğelerin sıralanmasını koruyabilmelerinde yatmaktadır. Splay Tree neredeyse tüm durumlarda aynı şekilde çalışır ve bu bölümde açıklanan özyinelemeli Splay Tree uygulaması rastgele veri kümelerinde AVL Ağacı uygulamasından bile daha iyi performans gösterir. AVL ağacı ile özyinelemeli Splay Tree performans sayıları arasındaki performans farkı, AVL ağacında dengeyi açıkça korumak ile Splay Tree’de yeterince iyi denge elde etmek arasındaki farktır.

# Bölüm Özeti

Bu bölümde yüksekliği dengelenmiş AVL ağaçlarının ve Splay Tree’lerinin çeşitli uygulamaları sunulmuştur. Özyinelemeli ve yinelemeli ekleme algoritmaları sunulmuştur. Hem dengeyi koruyan hem de yüksekliği koruyan AVL düğümleri ele alınmıştır. Hem AVL hem de Splay Tree için özyinelemeli ekleme algoritmaları, çok fazla özel durum olmadan oldukça 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.

# Programlama Problemleri

1. 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.
2. 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.
3. Özyinelemeli ekleme ve arama işlevlerine sahip bir Splay Tree uygulaması yazınız. Her bir düğümün yüksekliğinin korunduğu bir AVL ağacını yinelemeli ya da özyinelemeli olarak uygulayınız. Ağaçların aynı değerler listesinden oluşturulduğu bir test gerçekleştirin. Değer listesini oluşturduğunuzda, yinelenen değerler bir arama olarak değerlendirilmelidir. Veri dosyasını bir L veya I ile yazın ardından bir arama veya ekleme işleminin gerçekleştirilmesi gerektiğini belirten bir değer ekleyin. Performans sonuçlarınızı karşılaştırmak için PlotData.py programı tarafından kullanılan formatta bir XML dosyası oluşturun.
4. Özyinelemeli ekleme ve arama işlevlerine sahip bir Splay tree uygulaması yazın. Bu bölümde ayrıntıları verilen diğer dengeli ikili ağaç uygulamalarından biriyle karşılaştırın. Ağaçların aynı değerler listesinden oluşturulduğu bir test yapın. Değer listesini oluşturduğunuzda, yinelenen değerler bir arama olarak değerlendirilmelidir. Veri dosyasını bir L veya I ile yazın ardından bir arama veya ekleme işleminin gerçekleştirilmesi gerektiğini belirten bir değer ekleyin. Performans sonuçlarınızı karşılaştırmak için PlotData.py programı tarafından kullanılan formatta bir XML dosyası oluşturun.