\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Kümeler ve Haritalar 5

Son bölümde, yinelenen değerlere izin verilen şeylerin listelerini takip etmek için kullanılan dizileri inceledik. Örneğin, bir dizide ya da tam sayılar listesinde iki tane altı olabilir. Bu bölümde, yinelenen değerlere izin verilmeyen kümelere bakacağız. Kümeleri inceledikten sonra haritalar hakkında konuşmaya geçeceğiz. Haritalar sözlükler veya karma tablolar olarak da adlandırılabilir. Hash tablosu terimi aslında bir küme ya da harita uygulaması anlamına gelir. Bu bölümün birincil odak noktası hashing'i anlamaktır. Hashing Bilgisayar Bilimlerinde çok önemli bir kavramdır çünkü bir değeri aramak için çok verimli bir yöntemdir. Bölüme başlamak için hashing'e olan ilgimizi motive edeceğiz, ardından bir kümedeki değerleri bulmak için bir hashing algoritması geliştireceğiz. Ayrıca hashing'i kümelerin ve haritaların oluşturulmasına da uygulayacağız. Daha sonra memoization adı verilen hashing kullanan önemli bir tekniğe bakacağız ve bu tekniği birkaç probleme uygulayacağız.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

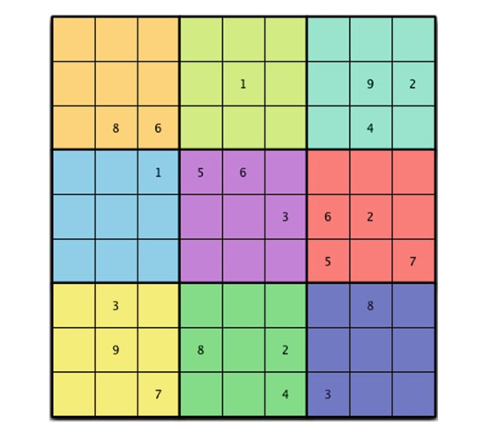
5.1 Bölüm Hedefleri

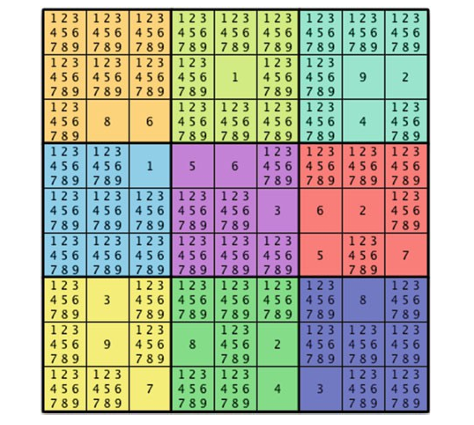
Bu bölümde birkaç soyut veri tipinin nasıl uygulanacağını öğreneceksiniz: kümeler ve eşlemeler. Bu veri türlerinin her ikisi için de hashing'in önemini okuyacaksınız. Ayrıca muta- ble ve immutable veriler arasındaki farkı anlamanın önemini de öğreneceksiniz. Bölümün sonunda şu sorulara cevap verebiliyor olmalısınız. • Bir kümede bir değer bulmanın karmaşıklığı nedir? • Yük faktörü nedir ve bir hash tablosunda aramanın genel verimliliğini nasıl etkiler? • Değişmez bir kümeyi ne zaman kullanırsınız? • Ne zaman değiştirilebilir bir set isteyebilirsiniz? • Bir problemde memoizasyon kullanmanın ne zaman bir avantajı vardır? • Hangi durumlarda bir harita veya sözlük kullanmak faydalı olabilir?

Yine bu bölümde, ilk olarak geçen bölümde sunulan tic tac toe oyununun optimizasyonu ve bir Sudoku bulmaca çözücüsü de dahil olmak üzere bazı ilginç programlama problemleri yer alacaktır. Bu ilginç problemleri çözmek için bilmeniz gerekenleri keşfetmek için okumaya devam edin.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_5.2 Sudoku Oynamak

Birçok insan Sudoku bulmacalarını çözmekten hoşlanır. Bir Sudoku bulmacasını çözmek için 9 × 9 matrisini dolduracak doğru sayıları bulmalısınız. Tüm sayılar 1- 9 olmalıdır. Her satırda 1-9'dan birer tane bulunmalıdır. Aynı şey her sütun için de geçerlidir. Son olarak, 9 × 9 matris içinde her birinde 1-9 olması gereken dokuz adet 3 3 kare vardır. Başlamak için, size Şekil 5.1'de gösterildiği gibi bazı konumları bilinen bir bulmaca verilir. Göreviniz, bulmacada zaten görünen sayılar göz önüne alındığında geri kalan sayıları bulmaktır. Bu bulmacaları çözmenin yaygın bir yolu eleme sürecidir. Bir bulmaca için olası değerleri yazmak ve ardından olası değerleri teker teker elemek yardımcı olur. Örneğin, yukarıdaki bulmaca Şekil 5.2'de gösterildiği gibi bilinmeyenler için olası değerlerle açıklanabilir. Bulmacayı çözmeye başlamak için, 8, 3 veya 9 içermeyen hücreler için bulmacanın ikinci sütunundan 8, 3 ve 9'u hemen eleyebiliriz. Bu sayıların hiçbiri ikinci sütundaki başka bir hücrede görünemez çünkü zaten bilinmektedirler. Aynı şekilde, 8, 6 ve 4 sayıları da bazı hücrelerden elenebilir

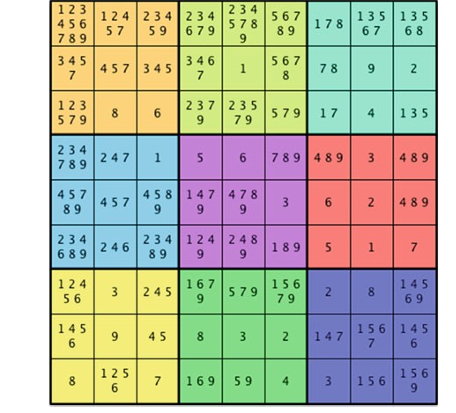




çünkü bu sayılar zaten diğer hücrelerde yer almaktadır. Bu gibi kuralların uygulanması bulmacadaki her bir hücre için olası değer sayısını azaltır. Şekil 5.3 bu kurallardan bazılarını uyguladıktan sonraki bulmacayı göstermektedir. Sudoku ve nasıl çözüleceği hakkında biraz düşünürsek, birçok Sudoku bulmacasını çözmek için kullanılabilecek iki kural türetebiliriz. Bu iki kural bulmaca içindeki herhangi bir gruba uygulanabilir. Bir grup, bulmacada bir satır, sütun veya kare içinde görünen dokuz hücreden oluşan bir koleksiyondur. Her hücrenin içinde, bulmacayı küçültme sürecinin bir noktasında hücre için olası değerleri temsil eden bir dizi sayı bulunur. İşte iki kural:

• KURAL 1: İlk kural, yukarıda hücrelerden bazı değerleri çıkarmak için kullandığımız işlemin bir genellemesidir. Bir grup içinde aynı olası değerler kümesini içeren hücreleri arayın. Kümenin kardinalitesi (yani kümedeki öğe sayısı) bulunan yinelenen kümelerin sayısıyla eşleşiyorsa, yinelenen kümelerin öğeleri gruptaki tüm yinelenmeyen kümelerden güvenli bir şekilde çıkarılabilir. Bu kural, yinelenen kümelerin sayısının 1 ve bu kümenin boyutunun 1 olduğu dejeneratif durumda bile geçerlidir. Dejeneratif durum, yukarıda tek öğeleri diğer kümelerden çıkarmak için kullandığımız durumdur. Bu kural, 2'nin tek başına göründüğü 7. sütun hariç bulmacanın 7. satırındaki tüm hücrelerden 2'yi çıkarmak için Şekil 5.3'e uygulanabilir.

• KURAL 2: İkinci kural, bir grup içindeki her hücreye bakar ve gruptaki diğer hücrelerde görünen tüm öğeleri atar. Seçilen hücrede yalnızca bir değer kaldıysa, bu değer bu hücrede görünmelidir ve hücre aşağıdakiler atılarak güncellenebilir.

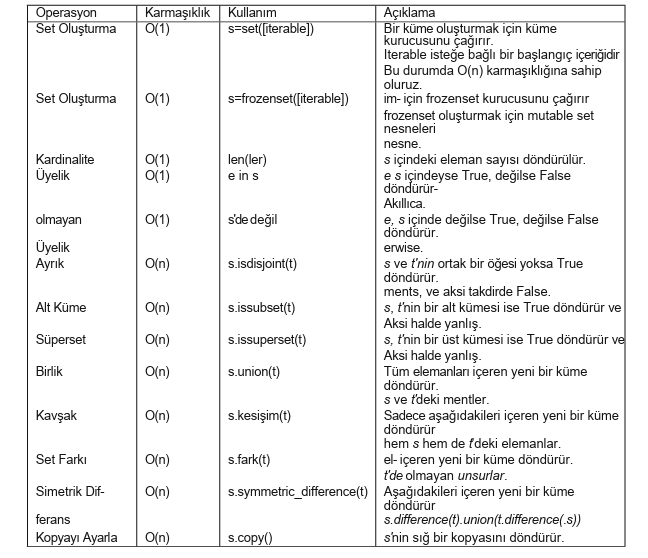


seçilen hücrede görünen diğer tüm değerleri uzaklaştırır. Bu kuralın Şekil 5.3'teki beşinci satıra uygulanması, dördüncü sütunun 1 içerecek şekilde indirgenmesiyle sonuçlanır çünkü 1, 5. satırdaki başka hiçbir hücrede görünmez. Bu kural bulmacanın son satırında da geçerlidir. 2 değeri ancak ikinci sütunda 1, 5 ve 6'yı o hücreden çıkardıktan sonra mümkündür çünkü bu değerler o satırdaki diğer hücrelerde yer almaktadır. Şekil 5.3'teki bulmacanın tam olarak indirgenmediğine dikkat edin. İndirgeme işlemi, daha fazla indirgeme mümkün olmayana kadar yinelemeli olarak uygulanabilir. Sudoku çözücü algoritması, bulmacadaki tüm gruplar üzerinde bir azaltma geçişi sırasında daha fazla değişiklik yapılmayana kadar bu azaltma işlemini uygulamaya devam eder. Bu iki kuralı bu şekilde uygulamak birçok Sudoku bulmacasını tamamen indirgeyecektir.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

5.3 Setler

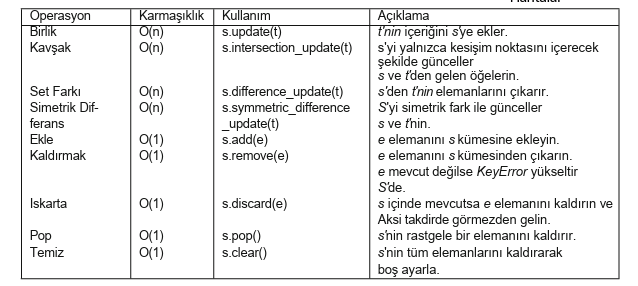
Sudoku bulmacaları için indirgeme algoritması sayı kümelerini manipüle eder ve indirgeme ilerledikçe bu kümelerdeki olası değerleri elimine eder. Küme, yinelenen değerlere izin vermeyen bir koleksiyondur. Kümeler herhangi bir değerden oluşabilir. Tamsayılar, çalışan nesneleri, karakterler, dizeler, kelimenin tam anlamıyla Python'daki herhangi bir nesne bir kümenin elemanı olabilir. Bir kümenin bir kardinalitesi vardır. Bir kümenin kardinalitesi, içindeki öğelerin sayısıdır.



Kümeler, üzerlerinde yaygın olarak tanımlanmış birkaç işlem bulunan nesnelerdir. Bu işlemler bazen birden fazla kümeyi içeren ikili işlemlerdir, bazen de sadece bir küme hakkında bilgi alırlar. Şekil 5.4'teki tablo, kümeler üzerinde yaygın olarak tanımlanan işlemleri ve bunlarla ilişkili hesaplama karmaşıklıklarını açıklamaktadır. Python iki tür küme için yerleşik desteğe sahiptir: set ve frozenset class. Frozenset class değişmezdir. Küme sınıfının nesneleri değiştirilebilir. Şekil 5.4'te s değişkeni bir küme ve t değişkeni de kümeleri içeren yinelenebilir bir dizi olmalıdır.

Infix operatörleri de Şekil 5.4'te tanımlanan bazı işlemler için sözdizimsel şeker olarak tanımlanmıştır. Alt küme içerme için s <= 1 yazabilirsiniz. Uygun alt küme için s < t yazabilirsiniz. Uygun alt küme, üst kümesinden en az bir daha az elemanı olan bir alt kümedir. Üst küme için s >= t veya uygun üst küme için s > t yazabilirsiniz. Birleştirme işlemi için s❘t yazmak s.union(t) yazmakla eşdeğerdir. Ve kesişim için, s&t, s.intersection(t) yazmaya eşdeğerdir. s t yazmak - s.difference(t) yazmakla aynıdır ve s^t simetrik fark operatörüne eşdeğerdir. Şekil 5.5'teki işlemler, s kümesini değiştirdikleri için frozenset sınıfı üzerinde tanımlanmamıştır.

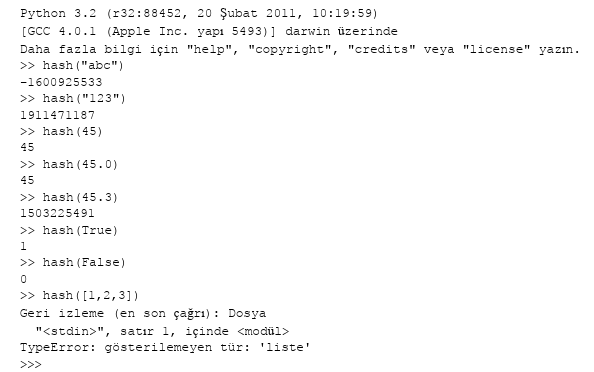
Yine, Şekil 5.5'te sunulan bazı yöntemler için operatörler vardır. Mutatör birleştirme yöntemi st olarak yazılabilir. Kesişim güncellemesi s&=t olarak yazılabilir. Son olarak, simetrik fark güncelleme operatörü s^=t olarak yazılır. Bu operatörler kullanışlı olmakla birlikte çok iyi bilinmemektedir ve yukarıdaki



Yukarıda sunulan hesaplama karmaşıklıkları şaşırtıcıdır! Küme üyeliği O(1) zamanda nasıl test edilebilir? Şimdiye kadar anlatılanlara bakılırsa, küme üyeliğini test etmek O(n) zaman almalıdır. Sonuçta, bir öğenin kümede olup olmadığını anlamak için kümedeki tüm öğelere ya da ortalama olarak en az yarısına bakmamız gerekir. Kümede yinelenen öğe olmadığından emin olmak istiyorsak, iki kümenin birleşimi O(n) zamanda nasıl hesaplanabilir? Küme bir şekilde sıralanmadığı sürece iki kümenin birleşiminin hesaplanması O(n²) zaman alacak gibi görünmektedir. Ancak bir kümenin elemanlarını sıralamak her zaman mümkün değildir çünkü kümelerin tüm elemanları bir sıralamaya sahip değildir.

5.4 Hashing

Bir küme üyelik testini O(1) zamanda uygulamak mümkünse, yukarıdaki diğer işlemleri belirttiğimiz karmaşıklıklarla uygulayabiliriz. O(1) üyelik testi olmadan, iki kümenin birleşimini almak yukarıda belirtildiği gibi çok daha uzun sürecektir. Küme üyeliğini O(1) zamanda test etmek hashing kullanılarak gerçekleştirilir. Hashing, Bilgisayar Biliminde son derece önemli bir kavramdır ve bir bilgisayardaki ran- dom erişimi ile ilgilidir. Bölüm 2'de gördüğümüz gibi, bir liste içindeki herhangi bir konuma erişim O(1) zamanda gerçekleştirilebilir. Bu rastgele erişim prensibidir. Rastgele erişilebilir bir liste, liste içindeki herhangi bir konuma O(1) zamanda erişilebileceği anlamına gelir. Listedeki bir konuma erişmek için erişmek istediğimiz konumun indeksine ihtiyacımız vardır. İndeks, listedeki bir öğenin adresi olarak işlev görür. Bir öğeyi listeye kaydettikten sonra, O(1) sürede geri getirmek istiyorsak, indeksini hatırlamamız gerekir. İndeks olmadan listedeki öğeyi aramak zorunda kalırız ki bu da O(1) zaman değil O(n) zaman alır. Dolayısıyla, üyeliği O(1) zamanda test edebileceğimiz bir küme uygulamak istersek, kümenin öğelerini bir listede saklamayı düşünebiliriz. O(1) zamanda tekrar bulmak için her bir öğenin saklandığı indeksi bir şekilde hatırlamamız gerekir. Bu ilk başta mümkün görünmüyor. Ancak, ya öğe adresini bulmak için kullanılabilseydi? Bu, hashing'e yol açan içgörüdür. Bilgisayardaki her nesne bilgisayarlar ikili sistemde çalıştığından sıfırlar ve birlerden oluşan bir dizi olarak saklanır. Bu sıfırlar ve birler, bir listenin indeksi olarak da dahil olmak üzere istediğimiz şekilde yorumlanabilir. Bu kavram o kadar önemlidir ki Python (ve diğer birçok modern dil) bir nesne için bir tamsayı değeri döndürmek üzere herhangi bir nesne üzerinde çağrılabilen hash adlı bir fonksiyon eklemiştir. Bu değere nesnenin hash kodu veya hash değeri diyeceğiz. Hash fonksiyonuna yapılan şu çağrıları düşünün.

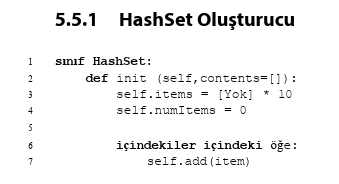


Çoğu nesne hashlenebilir olsa da, her nesne hashlenebilir değildir. Özellikle, listeler gibi değişebilir nesneler hashlenebilir olmayabilir, çünkü bir nesne değiştiğinde hash değeri de değişebilir. Bunun, bu bölümün ilerleyen kısımlarında göreceğimiz gibi, veri yapılarında hash değerleri kullanırken sonuçları vardır. Yerleşik türlere ek olarak, Python programcının bir sınıf üzerinde bir hash yöntemi uygulayarak hash kodları üzerinde biraz kontrol sahibi olmasına izin verir. Bir sınıf için bir hash metodu yazarsanız, o sınıfın örnekleri için istediğiniz hash değeri tamsayısını döndürebilirsiniz. "abc" dizesi üzerinde hash çağrısı negatif bir değer döndürürken, diğer hash çağrılarının son derece büyük tamsayılar döndürdüğüne dikkat edin. Bu hash tamsayısını bir listede kabul edilebilir bir indekse dönüştürmek için bazı çalışmalar yapılması gerektiği açıktır. Hash değerlerinin liste indekslerine nasıl dönüştürüldüğünü keşfetmek için okumaya devam edin.

5.5 HashSet

Sınıfı O(1) öğe arama karmaşıklığı elde etmek için bir listeye indeks hesaplamak üzere bir hash değeri kullanabiliriz. Listenin ayrıntılarını ve öğenin indeksini bulmak için hash fonksiyonunun çağrılmasını gizlemek için bir küme sınıfı yazılabilir. Python'un yerleşik küme sınıfıyla karıştırılmaması için küme sınıfımıza HashSet adını vereceğiz. Yerleşik küme sınıfı da hashing kullanır. Bu bölümde sunulan HashSet sınıfı size kümenin nasıl

sınıfı uygulanmaktadır. Başlangıçta, HashSet nesneleri bir liste ve listedeki öğelerin sayısını içerecektir. Başlangıçta liste bir grup None değeri içerecektir. Liste, içinde bir tür değer olacak şekilde oluşturulmalıdır. None, listede hiçbir değerin saklanmadığı yerler için null değer görevi görür. Liste, olası her hash değeri için bir konuma sahip olacak kadar büyük değildir. Yine de, listenin tüm olası hash değerleri için yeterince büyük olması mümkün değildir. Aslında, son bölümde gördüğümüz gibi, bazı hash değerleri negatiftir ve açıkça bir listedeki indisler negatif değildir. Bir hash değerinin bir liste indeksine dönüştürülmesi Bölüm 5.5.2'de daha ayrıntılı olarak açıklanmıştır. HashSet kurucusu Bölüm 5.5.1'de verilmiştir.



5.5.2 Bir Öğeyi Saklama

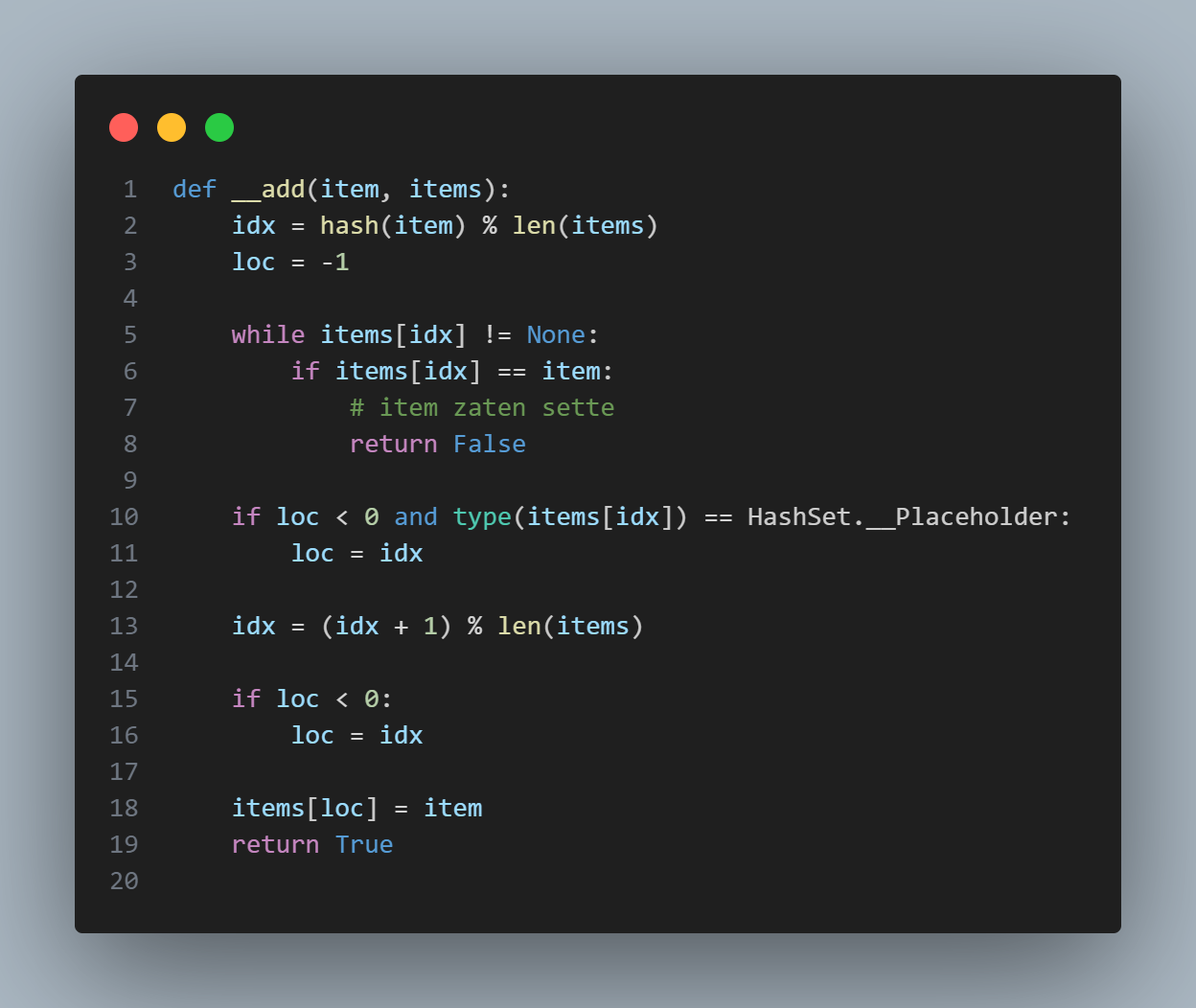
Bir öğeyi bir hash kümesinde saklamak için önce hash fonksiyonunu kullanarak indeksini hesaplarız. Ele alınması gereken iki sorun vardır. İlk olarak, öğelerin saklandığı liste sonlu uzunlukta olmalıdır ve kesinlikle hash fonksiyonunu çağırarak oluşturacağımız benzersiz hash değerleri kadar uzun olamaz. Listenin maksimum hash değerinden daha kısa olması gerektiğinden, listemiz için bir boyut seçeriz ve ardından hash değerlerini bu listenin uzunluğuna böleriz. Kalan (yani mod operatörü olarak adlandırılan % operatörünün sonucu) listenin indeksi olarak kullanılır. Listenin uzunluğuna bölündükten sonra kalan, hash değeri negatif bir tamsayı olsa bile her zaman 0 ile listenin uzunluğu eksi bir arasında olacaktır. Mod operatörünü kullanmak, seçtiğimiz boyuttaki bir listede bize geçerli indeksler verecektir. Başa çıkmamız gereken başka bir sorun daha var. Hash değerlerinin benzersiz olması gerekmez. Hash değerleri tam sayılardır ve bir bilgisayarda yalnızca sonlu sayıda tam sayı mümkündür. Buna ek olarak, hash değerlerini listenin uzunluğuna böldüğümüz için, kalanlar veya liste indeksleri orijinal hash değerlerinden daha az benzersiz olacaktır. Liste uzunluğu 10 ise, 44 ve -6 hash değerlerinin her ikisi de listede 4. indekste bir değer saklamaya çalışmakla sonuçlanacaktır. Bu elbette mümkün değildir.

5.5.3 Çarpışma

Çözünürlüğü Uzunluğu 10 olan bir listede hash kullanarak hem "Cow" hem de "Fox "u saklamaya çalıştığınızı düşünün. "Cow "un hash değeri -1432160826 ve "Fox "un hash değeri 1462539404'dur. Her iki değeri de 10 ile modladığımızda, kalan değer her iki hash değeri için de 4'tür ve bu da her ikisinin de listedeki beşinci konumda saklanması gerektiğini gösterir.

Hesaplanan indeksleri aynı olduğu için iki nesnenin karma küme listesinde aynı indekste saklanması gerekiyorsa, buna çakışma diyoruz. Bununla başa çıkmak için bir çarpışma çözümleme şeması tanımlamak gerekir. Mümkün olan birçok farklı şema vardır. Biz Lineer Problama adı verilen bir şemayı inceleyeceğiz. Doğrusal problama kullanırken bir çarpışma meydana geldiğinde, o konumun kullanılabilir olup olmadığını görmek için listedeki bir sonraki konuma ilerleriz. Bir konumun kullanılabilir olup olmadığını, listede o noktada bir None değeri bulursak anlayabiliriz. Listede bulabileceğimiz ve o konumun kullanılabilir olduğu anlamına gelen başka bir değer daha olduğu ortaya çıktı. Placeholder nesnesi adı verilen özel bir nesne türü de listede saklanabilir. Bu sınıfın nedeni bir sonraki bölümde ortaya çıkacaktır. Şimdilik, None ya da Placeholder nesnesi hash set listesi içinde açık bir konumu gösterir. Bölüm 5.5.4'teki kod, HashSet listesine bir öğe ekleme işlemini gerçekleştirir ve asıl add yöntemi için bir yardımcı işlevdir.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
**5.5.4 HashSet Add Yardımcı Fonksiyonu**



Bölüm 5.5.4'teki kod, zaten listede bulunan bir öğeyi eklemez. While döngüsü, kodun doğrusal adresleme (linear probing) kısmıdır. Öğe bulunana ya da bir None değeriyle karşılaşılana kadar idx indeksi, listenin uzunluğu modunda artırılır. None değerinin bulunması, doğrusal zincirin sonunu ve dolayısıyla öğenin zaten kümede olup olmadığını belirlemek için yapılan doğrusal aramanın tamamlandığını gösterir. Eğer öğe listede yoksa, öğe ya arama sırasında bulunan ilk \_\_Placeholder nesnesinin konumuna ya da zincirin sonundaki None değerinin konumuna eklenir.

Bir değer eklerken dikkate alınması gereken bir konu daha vardır. Hash küme listesinde yalnızca bir boş konum kaldığını düşünelim. Yukarıdaki kodda ne olurdu? Doğrusal arama, tüm listenin taranmasıyla sonuçlanırdı. Eğer liste dolu olsaydı, sonuç sonsuz bir döngü olurdu. Bu durumların hiçbirinin yaşanmasını istemiyoruz. Aslında, bir öğeyi amortize O(1) zamanda ekleyebilmek istiyoruz. Amortize edilmiş O(1) zaman karmaşıklığını sağlamak için listenin asla tamamen dolu veya neredeyse dolu olmaması gerekir.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**5.5.5 Yük Faktörü**

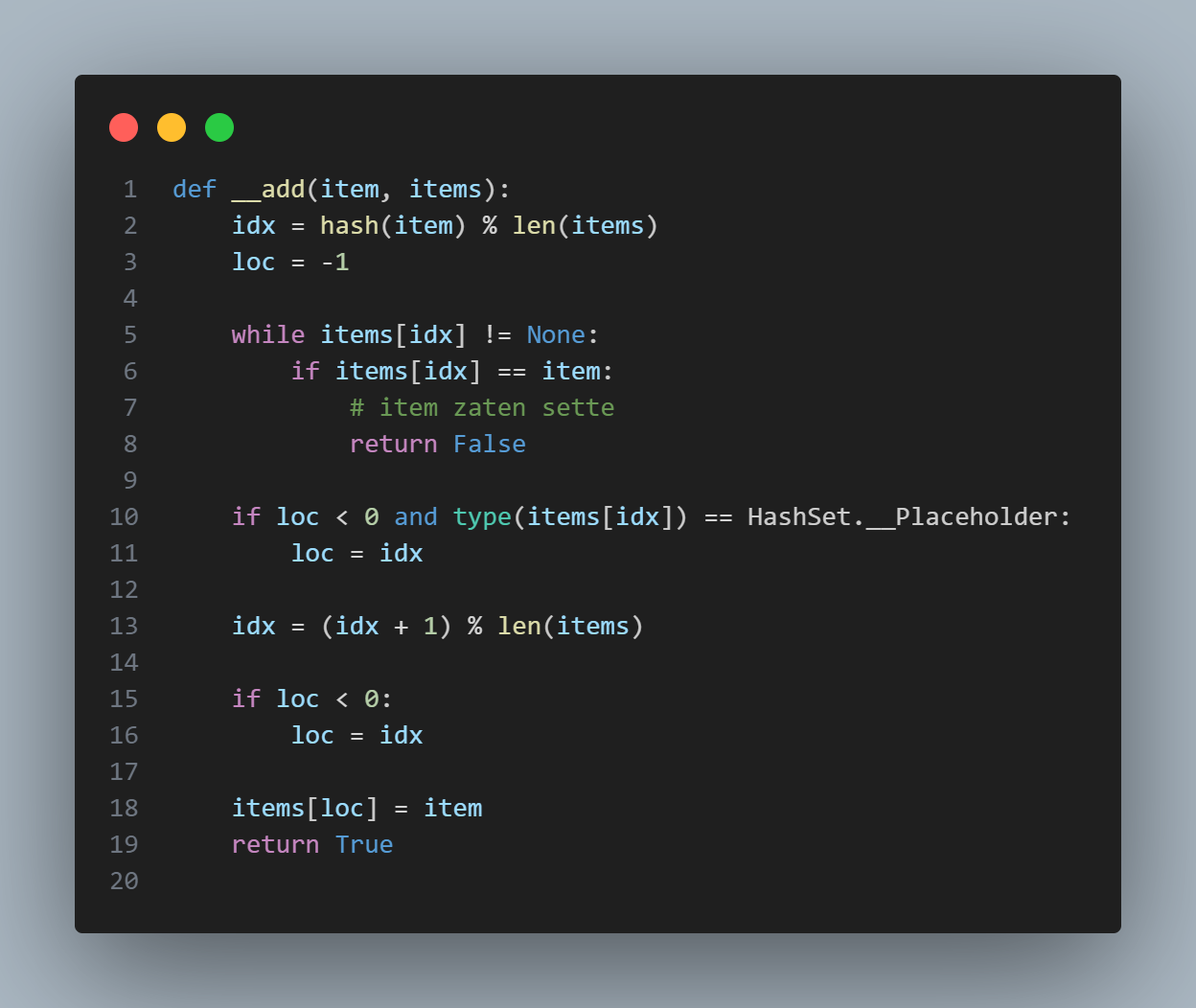
Karma küme listesinin doluluk oranına **yük faktörü** denir. Bir hash kümesinin yük faktörünü, listede saklanan öğe sayısını listenin uzunluğuna bölerek hesaplayabiliriz. Çok düşük bir yük faktörü, listenin içindeki öğe sayısına kıyasla çok daha büyük olduğu ve çarpışma olasılığının düşük olduğu anlamına gelir. Yüksek bir yük faktörü ise daha verimli alan kullanımı sağlarken çarpışma olasılığını artırır. Deneyler, en uygun yük faktörlerinin belirlenmesine yardımcı olabilir, ancak makul bir maksimum yük faktörü **%75** doluluktur.

Listeye bir değer eklerken, yük faktörü %75’i aşarsa, listedeki tüm değerler yeni bir listeye aktarılmalıdır. Değerleri yeni bir listeye aktarırken, yeni listenin uzunluğu farklı olacağından değerlerin tekrar **hashlenmesi** gerekir. Bu işleme **rehashing** denir. Hash kümesi uygulamasında, yeniden hashleme gerektiğinde listenin boyutunu **iki katına çıkarmayı** tercih ettik.

Bölüm 5.5.6'daki kod, Bölüm 5.5.4'teki \_\_add fonksiyonunu çağırır. Bu kod ve \_\_add yöntemi **HashSet** sınıfında yer almaktadır. \_\_add ve \_\_rehash fonksiyonları, genel erişime açık add metodunun kullandığı gizli yardımcı fonksiyonlardır.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**5.5.6 HashSet Ekle**



Yük faktörü kontrol altında tutulduğundan, listeye bir değer eklemenin amortize edilmiş karmaşıklığı **O(1)**'dir. Bu, liste içindeki herhangi bir zincirin uzunluğunun, hash kümesindeki öğe sayısından bağımsız olarak **sonlu** olacağı anlamına gelir.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

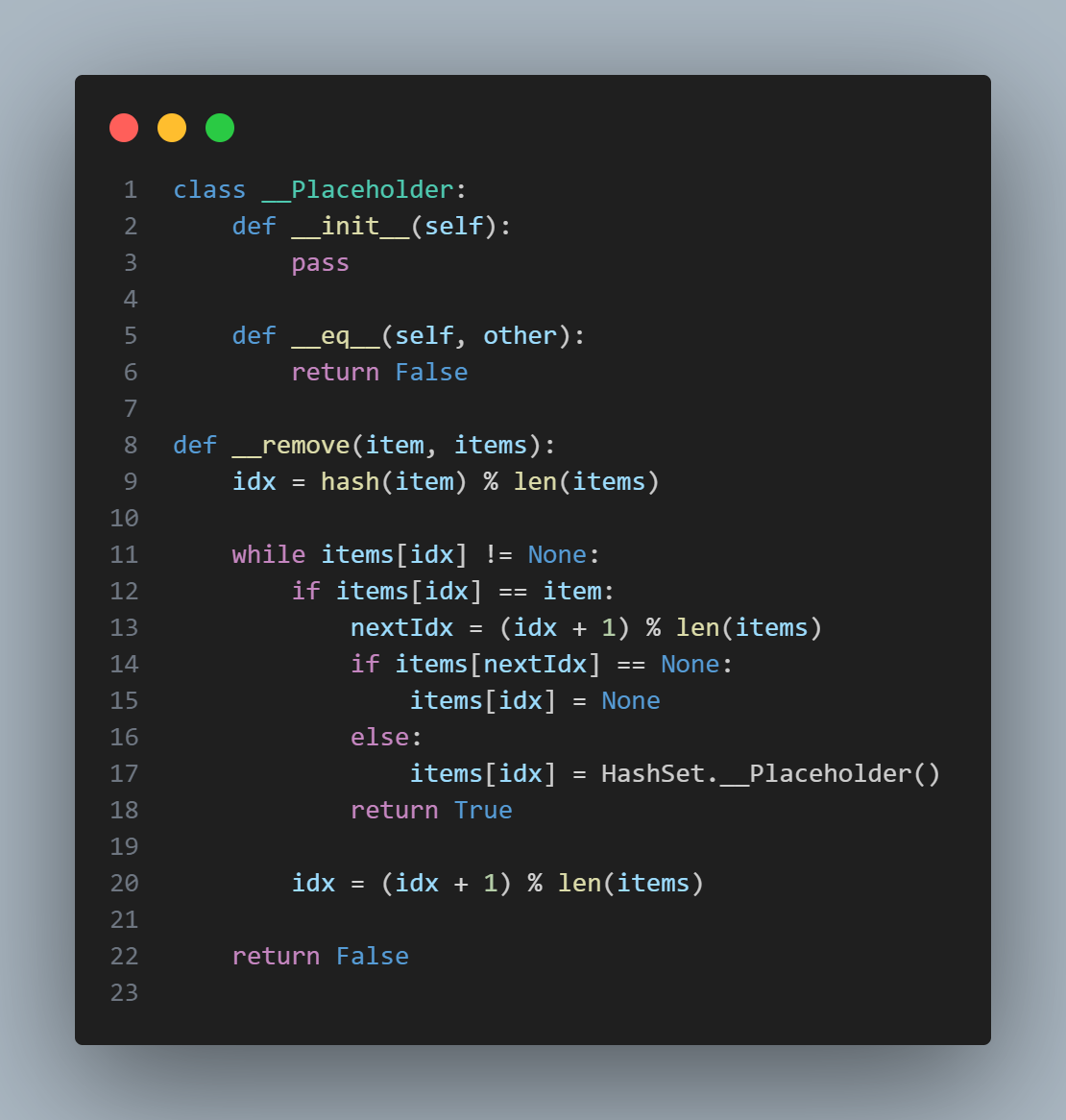
**5.5.7 Bir Öğeyi Silme**

Bir hash kümesinden bir değeri silmek, öncelikle öğeyi bulmayı gerektirir. Bu, listedeki belirli bir konumda bulunan değerler zincirinde **doğrusal bir arama** yapmayı içerebilir. Eğer silinecek değer zincirdeki **son öğeyse**, bir None ile değiştirilebilir. Ancak, öğe zincirin **ortasında** bulunuyorsa, onu None ile değiştiremeyiz çünkü bu, zinciri keserek bağlantıyı koparır. Bunun yerine, öğe bir \_\_Placeholder nesnesiyle değiştirilir.

Bir **yer tutucu** nesne zinciri bozmaz ve doğrusal adresleme gerektiğinde, arama işlemi **yer tutucu nesneleri atlayarak** devam eder. **remove** yardımcı fonksiyonu, Bölüm 5.5.8'de verilmiştir.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**5.5.8 HashSet Kaldırma Yardımcı Fonksiyonu**



Bir öğeyi kaldırırken, yük faktörü çok düşebilir ve bellekte alan kullanımının verimli olmamasına neden olabilir. **Yük faktörü %25’in altına düştüğünde**, yük faktörünü artırmak amacıyla **liste yeniden rehash edilir** ve boyutu **yarıya indirilir**.

**remove** yöntemi, Bölüm 5.5.9’da verilmiştir.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
**5.5.9 HashSet Kaldır**

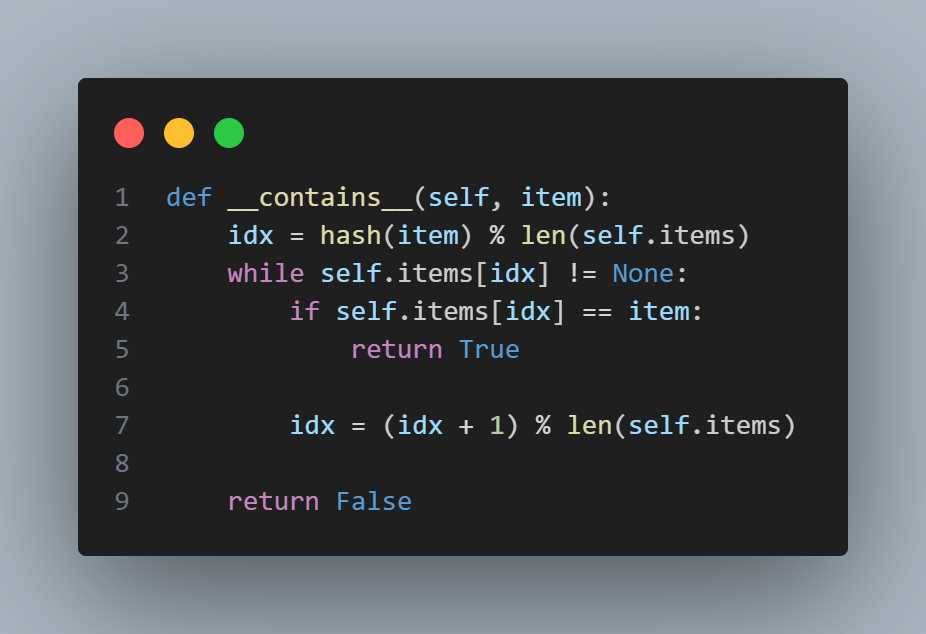


Bir değeri **O(1) zamanda** ekleyebilme sebebiyle, bir değerin silinmesi de **O(1) amortize karmaşıklık** ile gerçekleştirilebilir. **discard** yöntemi, Bölüm 5.5.9’da sunulan **remove** yöntemiyle neredeyse aynıdır, ancak **öğe kümede yoksa herhangi bir istisna fırlatılmaz**.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**5.5.10 Bir Oğe Bulma**

Bir hash kümesindeki bir öğeyi bulmak, öğenin adresini bulmak üzere **hashlenmesini** ve ardından olası değerler zincirinin **arama yapılmasını** gerektirir. Zincir, bir **None** ile sonlanır. Öğe zincirde bir yerdeyse, **contains** yöntemi **True** döndürecektir; aksi takdirde **False** döndürülür. Kümedeki öğe bir programa yazıldığında, Bölüm 5.5.11'deki yöntem çağrılır.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**5.5.11 HashSet Üyelik**

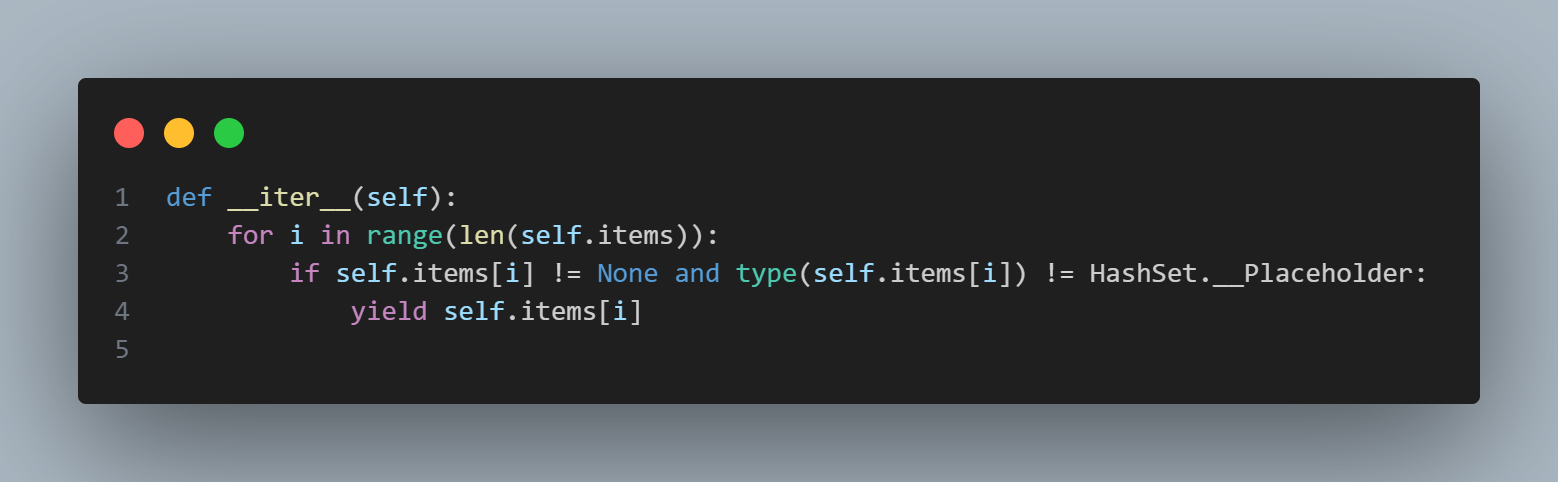


Bir öğeyi bulmak da O(1) amortize karmaşıklık ile sonuçlanır. Zincirler, çoğu hash değeri eşit olarak dağıtıldığı ve yük faktörü 1’e yaklaşmadığı sürece kısa tutulur.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**5.5.12 Bir Küme Üzerinde Yineleme**

Bir kümenin öğeleri üzerinde yineleme yapmak için, HashSet’in öğelerini **yield** ederek döndüren **iter** yöntemini tanımlamamız gerekir. Yöntem, listeyi gezerek yer tutucu öğeleri ve None referanslarını atlar. İşte   
yineleyici için kod.



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**5.5.13 Diğer Set İşlemleri**

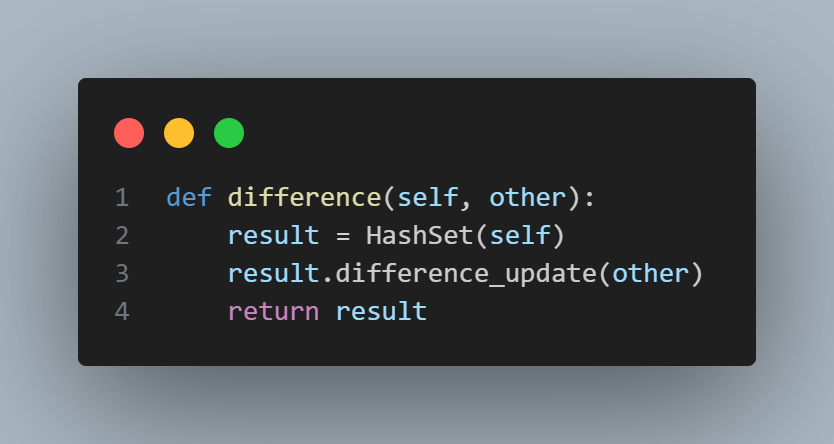
HashSet üzerindeki diğer küme işlemlerinin birçoğu okuyucu için bir alıştırma olarak bırakılmıştır. Ancak, bunların çoğu bu bölümde daha önce sunulan yöntemler açısından uygulanabilir. difference\_update yöntemini ele alalım. Yineleyici, üyelik testi ve ıskarta yöntemi kullanılarak uygulanabilir. Bölüm 5.5.14'teki kod, difference\_update yöntemi için bir uygulama sunmaktadır

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**5.5.14 HashSet Fark Güncellemesi**



Bölüm 5.5.14'te sunulan difference\_update yöntemi, self tarafından referans verilen diziyi değiştirdiği için bir mutatör yöntemidir. Bunu, self tarafından referans verilen nesneyi değiştirmeyen Bölüm 5.5.15'teki difference yöntemi ile karşılaştırın. Bunun yerine, fark yöntemi self ve diğer kümenin farkından oluşan yeni bir küme döndürür.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
**5.5.15 HashSet Fark**

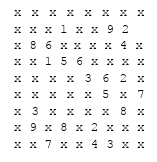


Fark yöntemi, sonuç HashSet üzerinde difference\_update yöntemi kullanılarak uygulanır. Yeni bir kümenin döndürüldüğüne dikkat edin. self tarafından başvurulan hash kümesi güncellenmez. Kod basittir ve difference\_update doğru yazılırsa bu yöntemin de doğru yazılacağı gibi bir avantajı vardır. Programcılar mümkün olduğunca yinelenen kodlar yazmaktan kaçınmalıdır. difference ve difference\_update yöntemleri, difference yönteminin self'in referans aldığı küme yerine yeni oluşturulan bir küme üzerinde fark işlemi gerçekleştirmesi dışında neredeyse aynıdır.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**5.6 Sudoku Çözme**

Bir küme veya HashSet veri türü kullanarak, artık çoğu Sudoku bulmacasını çözmek için gerekli araçlara sahibiz. Bir bulmaca, bilinen değerlerin rakamlarıyla temsil edildiği ve bilinmeyen değerlerin aşağıdaki bulmacada olduğu gibi X'ler olduğu bir dosyadan okunabilir.

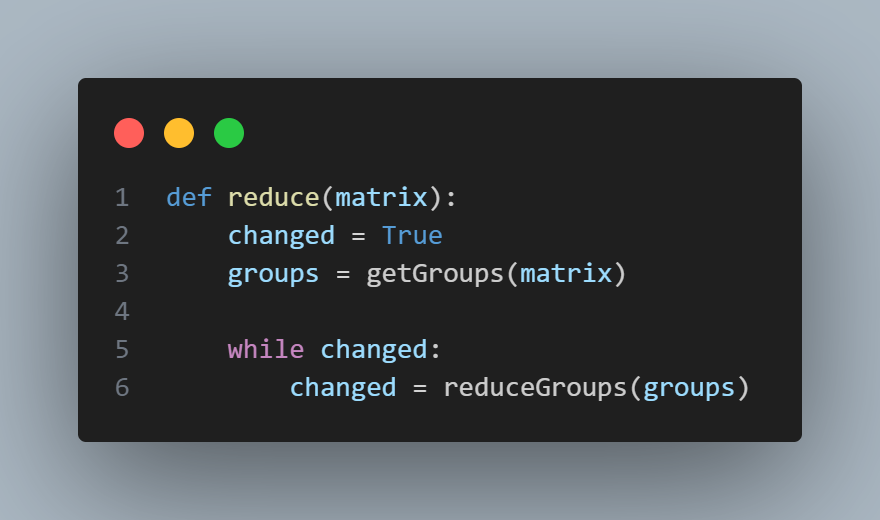


Dateable'ı okumak her seferinde bir satır yapılabilir. Satırı bölmek, bilinen ve bilinmeyen her değeri listede ayrı bir öğe olarak içeren bir dize sağlayacaktır. Bir X ile karşılaşıldığında, tıpkı Sudoku'yu elle çözerken yaptığınız gibi, 1-9 arası tüm değerleri içeren bir küme oluşturulabilir. Bilinen bir değer bulunduğunda, içinde bilinen sayının bulunduğu bir küme oluşturulabilir. Kümeler iki boyutlu bir matrise eklenir. Matris, listelerden oluşan bir listedir. Yani bir satırı okumak matrisin bir satırını okumaya karşılık gelir. Her satır bir kümeler listesi haline gelir. Her set listesi matris adını vereceğimiz bir listeye eklenir. Yani, matrix[row][col] Sudoku bulmacası içindeki bir settir. Bir Sudoku bulmacasında 81 küme vardır. Ancak her küme üç grubun üyesidir: satır, sütun ve içinde bulunduğu kare. Yukarıdaki Sudoku bulmacası açıklamasında sunulan kuralların her biri bu gruplardan biri içindeki bir kümeyi indirgemekle ilgilidir. Dosyadan girdiyi okuduktan ve 81 set ile matrisi oluşturduktan sonra, bir grup listesi oluşturularak 27 grup oluşturulur. Her satırın yüzeysel bir kopyası ilk olarak gruplar listesine eklenir. Sığ bir kopya, kümelerin her birinin bulmaca okunduğunda oluşturulan kümeyle aynı olduğu anlamına gelir.

Derin bir kopya 81 kümenin her birinin bir kopyasını oluşturur. Bir listenin sığ kopyası, liste içindeki kümeleri kopyalamaz. Bir liste üzerinde list çağrısı yapmak sığ bir kopya oluşturacaktır. Her sütun için başka bir grup oluşturulur ve bu gruplar gruplar listesine eklenir. Son olarak, her kare için bir grup oluşturulur ve bu gruplar gruplar listesine eklenir. Hepsi tamamlandığında, içinde 27 grup bulunan bir gruplar listesi oluşur. Bu grupları oluştururken, üç grubun her birinde aynı kümenin görünmesi çok önemlidir. Bunun nedeni, bir satır küçültüldüğünde, bu satırdaki değişikliklerin satırın elemanlarının göründüğü sütunlara ve karelere de yansımasını istememizdir. Bir Sudoku bulmacasını çözmek, Bölüm 5.2'de sunulan iki kurala göre bir grubun her bir kümesindeki öğe sayısını azaltmak anlamına gelir. İndirgenecek 9 kümelik bir liste verilen reduceGroup adında bir fonksiyon yazmak yardımcı olabilir. reduceGroup fonksiyonu, grubu azaltabildiyse True, azaltamadıysa False döndürmelidir. Bu reduceGroup fonksiyonu verildiğinde, reduce fonksiyonu Bölüm 5.6.1'de tanımlandığı gibidir.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**5.6.1 Sudoku Azaltma Fonksiyonu**



Bu algoritma oldukça basit ve yine çok güçlüdür. Şimdiye kadar bu metinde sunulan diğer algoritmalardan farklıdır. Kavram oldukça basittir: Daha fazla indirgeme mümkün olmayana kadar indirgemeye devam edin. Algoritmanın her iterasyonu, bulmacanın bazı kümelerindeki öğe sayısını azalttığı için, sonlanacağı garanti edilir. Bu kümelerin boyutunu asla artırmayız. Bu fonksiyondan döndüğümüzde, bir çözümümüz olup olmadığı belirsizdir. Bu Sudoku çözümleyicisiyle çözülemeyen bazı bulmacalar vardır çünkü yukarıda sunulan iki kural, tüm bulmacalar için yeterince güçlü değildir. Bazı durumlarda, bir kümedeki öğe sayısı sadece bir gruba bakarak indirgenemez. Öğelerin sayısını azaltmak için birden fazla gruba aynı anda bakmak gerekebilir. Ama durun, başka kurallar bulmanıza gerek yok. Bir sonraki bölüm, tüm Sudoku bulmacalarını, en zor olanları bile çözmek için bir algoritma sunacaktır.

Bu bölümde sunulan kurallar, bu bölümde verilen Sudoku bulmacasını ve daha birçok bulmacayı çözecektir. Metnin web sitesindeki birden altıya kadar olan Sudoku bulmacaları, bu Sudoku çözümleyicisiyle çözülebilir. Yukarıdaki reduceGroups fonksiyonu, muhtemelen listesinde bulunan her bir grup için reduceGroup fonksiyonunu çağırır ve gruplardan herhangi biri indirgenirse True, aksi takdirde False döndürür.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**5.7 Haritalar**

Bilgisayar bilimlerinde harita, arabanızla bir yere giderken kullandığınız harita gibi değildir. Harita terimi, bir alanı bir aralığa eşleyen bir fonksiyona atıfta bulunan daha matematiksel bir terimdir. Python'da zaten bir harita kullanmış olabilirsiniz. Haritalar, sözlükler, hash tabloları ve hash haritaları gibi birçok isimle anılır. Hepsi aynı veri yapısıdır.

Bir harita veya sözlük, bir fonksiyonun etki alanındaki bir değeri aralığa eşlemesi gibi, bir dizi benzersiz anahtarı ilişkili değerlerle eşler. Anahtar, anahtar/değer çiftini aramak istediğimizde bir haritaya sağladığımız şeydir. Bir haritanın anahtarları benzersizdir. Bir sözlükte, belirli bir anahtarın sadece bir kopyası olabilir. Birinci bölümde gördüğümüz gibi, Python, sözlükler veya haritalar için yerleşik destek sunar. İşte Python kabuğunda bir sözlükle yapılan bazı örnek etkileşimler.



Bir harita veya sözlük, bir kümeye çok benzer. Küme ve sözlüğün her ikisi de benzersiz değerler içerir. Küme veri türü, bir grup benzersiz değer içerir. Bir harita ise ilişkili değerlerle eşlenen benzersiz anahtarlar kümesi içerir. Kümelerde olduğu gibi, haritadaki bir anahtarı ve ilişkili değerini O(1) sürede arayabiliriz. Tahmin edebileceğiniz gibi, kümeler gibi haritalar da hashing kullanılarak uygulanır. Temel uygulama aynı olsa da, haritalar ve kümeler farklı şekilde kullanılır. Aşağıdaki tabloda, haritaların veya sözlüklerin yöntemleri ve operatörleri ile bunlara karşılık gelen karmaşıklıkları verilmiştir. Yukarıdaki tabloda yer alan işlemler, Bölüm 5.5'te sunulduğu gibi bir hash uygulaması göz önüne alındığında beklenen karmaşıklıklara sahiptir. İlginç olan fark, sözlükte sadece bir kümenin öğeleri yerine anahtar/değer çiftlerinin saklanmasıdır. Anahtar/değer çiftinin anahtar kısmı, beklediğiniz gibi bir anahtarın sözlükte olup olmadığını belirlemek için kullanılır. Uygun olduğunda değer döndürülür.

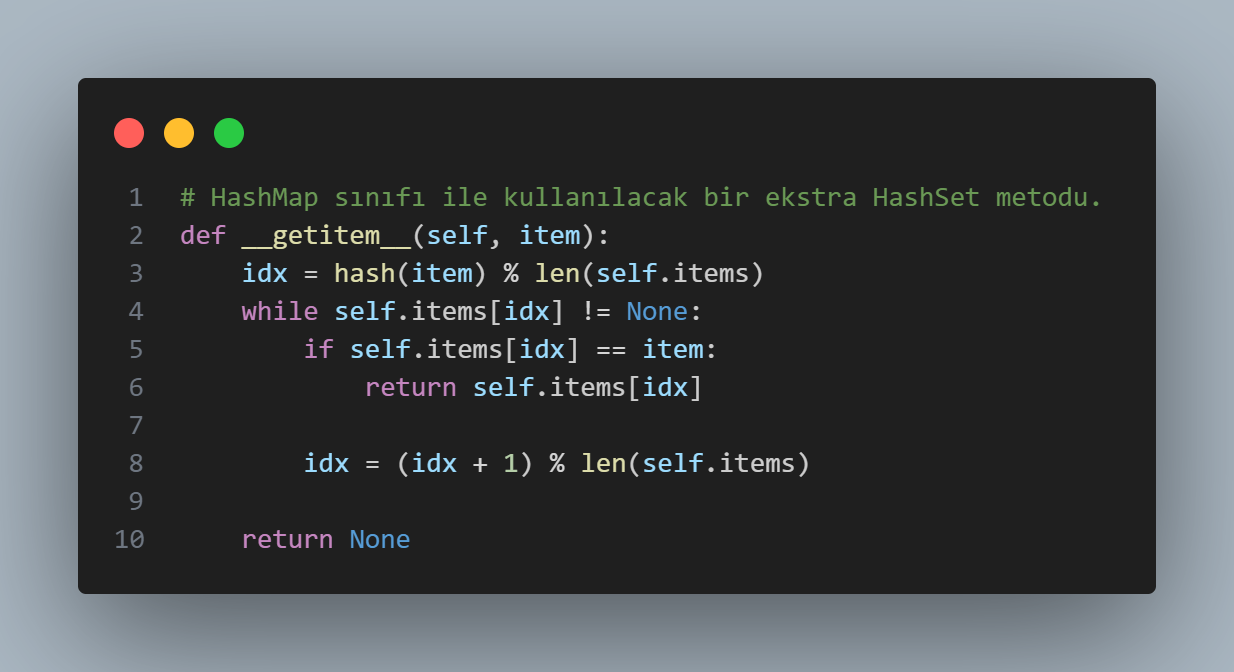
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

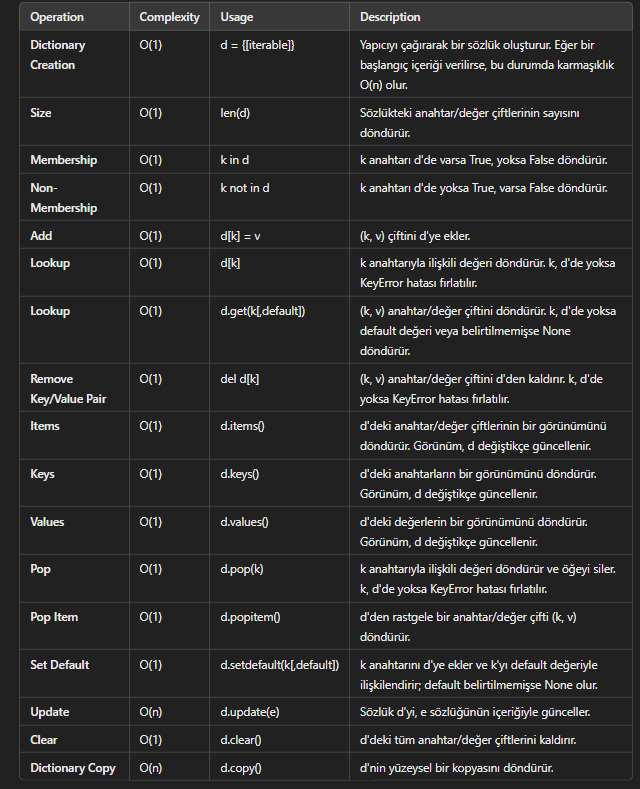
**5.7.1 HashMap Sınıfı**

Python'daki dict sınıfı gibi bir HashMap sınıfı, Şekil 5.6'daki tabloda özetlenen karmaşıklıkları elde etmek için hashing kullanır. Özel bir \_\_KVPair sınıfı tanımlanmıştır. \_\_KVPair örnekleri, HashMap nesnesine eklendikçe anahtar/değer çiftlerini tutar. HashSet sınıfına bir **getitem** yöntemi eklenerek, HashSet sınıfı HashMap sınıfı uygulaması için kullanılabilir. HashSet için ek **getitem** yöntemi Bölüm 5.7.3'te verilmiştir.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

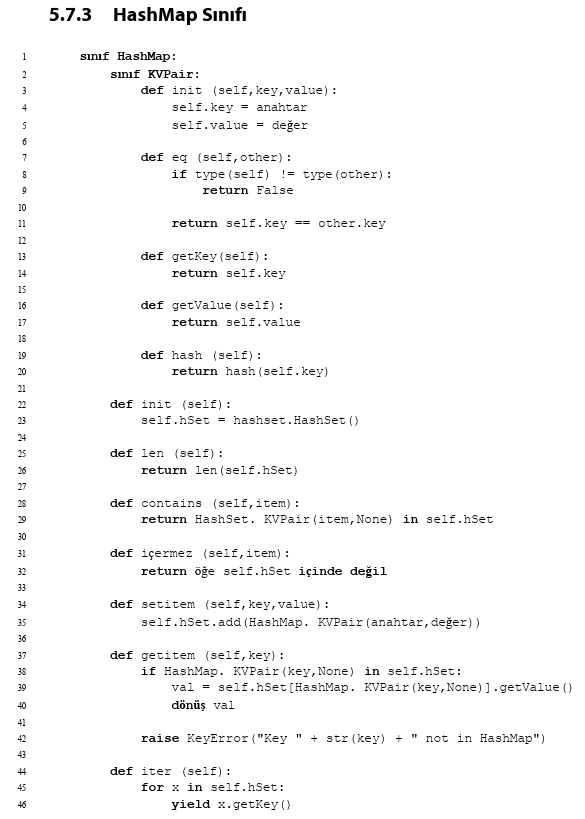
**5.7.2 HashSet Öğeyi Al**





Daha sonra, HashMap'i uygulamak için Bölüm 5.7.3'te gösterildiği gibi bir HashSet kullanabiliriz. \_\_KVPair sınıf tanımında, hash haritasındaki iki öğeyi karşılaştırırken anahtarların karşılaştırılması için **eq** yöntemini tanımlamak gerekir. \_\_KVPair'in **hash** yöntemi yalnızca anahtar değerini hash eder, çünkü anahtarlar hash haritasında anahtar/değer çiftlerini aramak için kullanılır. Bölüm 5.7.3'te verilen uygulama kısmi bir uygulamadır. Diğer yöntemler okuyucu için bir alıştırma olarak bırakılmıştır.

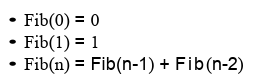
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_



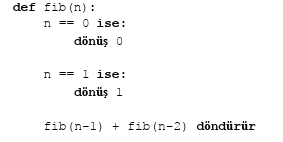
Bölüm 5.7.3'te verilen uygulama, HashSet sınıfı ile HashMap sınıfı veya Python'daki set ve dict sınıfları arasındaki benzerlikleri göstermeye yardımcı olur. Her iki veri yapısı türü de hashing kullanılarak uygulanmaktadır. Her ikisi de büyük ölçüde O(1) üyelik testine dayanır. HashMap sınıfının nasıl uygulandığını anlamak önemli olsa da, çoğu programlama dili Python gibi yerleşik türler kütüphanesinde bir çeşit hash haritası içerir. Bir hash haritasındaki yöntemlerin karmaşıklığını anlamak önemlidir, ancak bir hash haritasının ne zaman kullanılacağını ve nasıl kullanılabileceğini anlamak da bir o kadar önemlidir. Yazdığınız kodu daha verimli hale getirmek için bir hash haritasını nasıl kullanabileceğinizi görmek için okumaya devam edin.

5.8 Memoizasyon

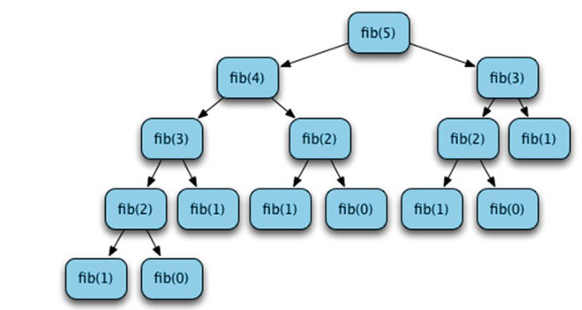
Memoization, aynı argümanlarla birden fazla kez çağrılabilecek fonksiyonlar yazarken kullanılabilecek ilginç bir programlama tekniğidir. Memoizasyonun arkasındaki fikir, bir fonksiyonda bir değer hesaplama işini bir kez yapmaktır. Daha sonra, fonksiyon aynı argümanlarla tekrar çağrıldığında, az önce hesapladığımız değeri tekrar döndürmek için kendimize bir not alırız. Bu, değeri yeniden hesaplama işine girmeyi önler. Bunun güçlü bir örneği özyinelemeli Fibonacci fonksiyonudur. Fibonacci dizisi aşağıdaki gibi tanımlanır.



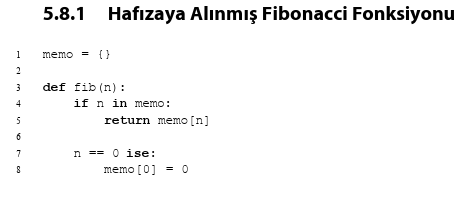
Bu dizi, aşağıdaki gibi bir Python fonksiyonu yazılarak özyinelemeli olarak hesaplanabilir.

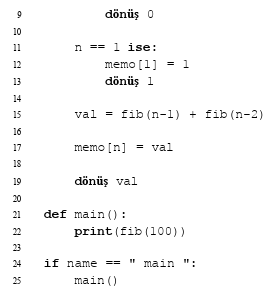


Ancak, bu fonksiyonu küçük bir Fibonacci sayısının basit bir gösteriminden başka bir şey için asla kullanmak istemeyiz. Bu fonksiyon fib(100) kadar büyük bir sayıyı hesaplamak için bile kullanılamaz. Fonksiyonu 100'lük bir argümanla çalıştırmak en hızlı bilgisayarlarda bile çok uzun zaman alacaktır. fib(5)'i hesaplamak için neler yapıldığını düşünün. Bunu yapmak için önce fib(4) ve fib(3) hesaplanmalıdır. Daha sonra bu iki sonuç fib(5)'i bulmak için toplanabilir. Ancak, fib(4)'ü hesaplamak için fib(3) ve fib(2) değerleri hesaplanmalıdır. Şimdi fib(5)'i hesaplamak için fib(3)'ü iki kez hesaplıyoruz, Ancak fib(3)'ü hesaplamak için fib(2) ve fib(1)'i hesaplamalıyız. Ayrıca, fib(4)'ü bulmak için de fib(2) hesaplanmalıdır. Şekil 5.7, fib(5)'i hesaplamak için fib'e yapılan tüm çağrıları göstermektedir.



Şekil 5.7'den de görebileceğiniz gibi, fib(5)'i hesaplamak için fib fonksiyonuna çok sayıda çağrı yapılması gerekir. Şimdi fib(6)'yı hesaplamak için kaç çağrı gerektiğini hayal edin. fib(6)'yı hesaplamak için önce fib(5)'i hesaplamamız ve ardından fib(4)'ü hesaplamamız gerekir. Fib(5)'i hesaplamak için fib'e 15 çağrı yapıldı ve şekilden fib(4)'ü hesaplamak için 9 çağrı yapıldığını görebiliriz. fib(6) çağrısı da dahil olmak üzere, fib(6)'yı hesaplamak için fib'e 25 çağrı yapılması gerekecektir. fib(7)'nin hesaplanması 15+ 25 + 1 çağrı veya 41 çağrı alacaktır. Bu şekilde fib(n)'yi hesaplamak, fib(n-2)'yi hesaplamak için yapılan çağrı sayısını iki katından fazla artırır. Buna üstel büyüme denir. fib fonksiyonunun karmaşıklığı O(2)'dir. Üstel karmaşıklığa sahip bir fonksiyon, n'nin çok küçük değerleri dışında değersizdir. Her şey bitmiş değil. Fibonacci dizisini hesaplamanın daha iyi yolları var. Verimliliği artırmanın yolu, tüm bu gereksiz işlerden kaçınmaktır. Fib(2) bir kez hesaplandıktan sonra, onu tekrar hesaplamamalıyız. Bu işi zaten yaptık. Verimliliği artırmanın en azından birkaç yolu vardır. Bir yöntem, özyinelemeyi kaldırmayı ve fib(n)'yi bir döngü ile hesaplamayı içerir ki bu muhtemelen en iyi seçenektir. Ancak, özyinelemeli fonksiyon orijinal tanıma daha yakındır. Fonksiyonun özyinelemeli versiyonunu memoizasyon ile geliştirebiliriz. Bölüm 5.8.1'de, memo sözlüğü n değerlerinden fib(n) sonuçlarına eşlememiz olarak hizmet eder.





Bölüm 5.8.1'deki memolaştırılmış fib fonksiyonu, fonksiyon tarafından döndürülen herhangi bir değeri memosuna kaydeder. Memo değişkenine çevreleyen kapsamdan erişilir. Memo yerel olarak oluşturulmaz çünkü fib'in bir çağrısından diğerine kadar kalıcı olmasını isteriz. fib yeni bir n değeri ile her çağrıldığında cevap memo'ya kaydedilir. Bir n değeri için fib(n) daha sonra çağrıldığında, memo haline getirilmiş sonuç aranır ve döndürülür. Sonuç: memolaştırılmış fib fonksiyonu artık O(n) karmaşıklığına sahiptir ve fib(100)'ü neredeyse anında hesaplayabilir. Memolaştırma olmasaydı, fib(100)'ü hesaplamak için fib fonksiyonuna 1,146,295,688,027,634,168,201 çağrı yapılması gerekirdi. Her bir fonksiyon çağrısının 10 mikrosaniyede tamamlandığını varsayarsak, fib(100)'ü hesaplamak yaklaşık 363 milyon yıl sürerdi. Memoizasyon ile fib'e 100 çağrı yapılır ve çağrı başına 10 mikrosaniye olduğu varsayılırsa, bu 1000 mikrosaniye veya saniyenin 1/1000'idir. Bu, hafızaya almanın faydasına ilişkin uç bir örnektir, ancak birçok durumda kullanışlı olabilir. Örneğin, Bölüm 4'teki tic tac toe probleminde minimax fonksiyonu birbirinin aynısı olan birçok tahta üzerinde çağrılır. Minimax fonksiyonu bir X'in önce sağ üst köşeye sonra sol alt köşeye ya da tam tersi şekilde yerleştirilmesini önemsemez. Yine de, minimax'ın yazıldığı şekilde, aynı tahtanın değerini hesaplamak için birden çok kez çağrılacaktır. Minimax'ın hatırlanması tic tac toe oyununu hızlandırır.

5.9 İki Bilgi Kaynağının İlişkilendirilmesi

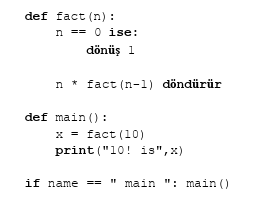
Harita veya sözlüğün bir başka kullanımı da farklı kaynaklardan gelen verilerin ilişkilendirilmesidir. Size şehirlerin ve bu şehirlerdeki posta kodu veya kodlarının bir listesinin verildiğini varsayalım. İnsanların ABD'deki bir şehrin posta koduna bakabilecekleri bir hizmet sunmak istiyorsunuz. Dolayısıyla, size bilgi sağlayan web sayfası tarafından size bir şehir verilecektir. Olası posta kodlarının bir listesini bulmak için bu şehri kullanmanız gerekiyor. Karşılık gelen posta kodları listesini bulmak için şehir listesinde arama yapabilirsiniz. Ya da şehir adından posta kodu listesine bir sözlük oluşturabilirsiniz. Daha sonra bir şehir adı verildiğinde, sözlükte olup olmadığını kontrol edersiniz ve eğer varsa, O(1) sürede karşılık gelen posta kodları listesine bakabilirsiniz.

5.10 Bölüm Özeti

Bu bölümde Python'da kümelerin ve eşlemelerin uygulamalarını ve bazı kullanımlarını inceledik. Hashing önemli bir kavramdır. Hashing veri yapıları, bir çarpışma çözümleme stratejisi ile hash tablosu içindeki çarpışmaları engelleyebilmelidir. Bu bölümde incelenen çözümleme stratejisi doğrusal problama idi. Başka çarpışma çözümleme stratejileri de mümkündür. Herhangi bir çarpışma çözümleme stratejisi, bir zincire eklenen yeni değerleri ve bir zincirden silinen mevcut değerleri ele almanın bir yoluna sahip olmalıdır. Hashing'in en önemli özelliği, üyelik testi ve tablo içinde arama için amorti edilmiş O(1) karmaşıklığıdır. Üyeliği test etme veya bir değeri O(1) zamanında arama yeteneği, aksi takdirde büyük veri kümelerinde verimli çalışmayabilecek birçok algoritmayı verimli hale getirir. Hafızaya alma, bir sözlüğün veya haritanın önemli bir kullanımıdır. Bir fonksiyonu hafızaya alarak gereksiz iş yapmaktan kaçınırız. Harita veya sözlüklerin bir başka önemli kullanımı da bilgi kaynaklarını ilişkilendirmektir. Bize iki farklı kaynaktan bilgi verildiğinde ve bu iki kaynağı eşleştirmemiz gerektiğinde, bir harita veya sözlük bu korelasyonu verimli hale getirecektir.

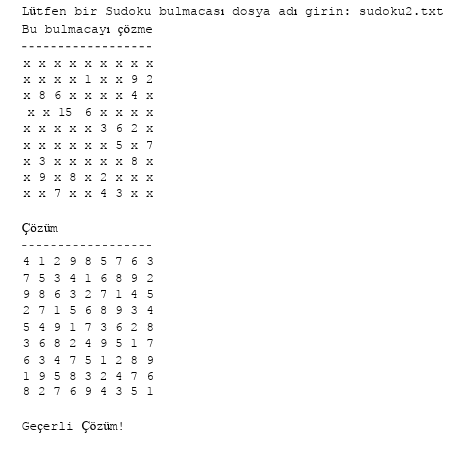
5.11 İnceleme Soruları

Bu kısa cevaplı, çoktan seçmeli ve doğru/yanlış soruları yanıtlayarak bölüme hakimiyetinizi test edin. 1. Hash kodu ne tür bir değerdir? 2. Hash kodları hem pozitif hem de negatif olabilir. Bir hash kodu, bir hash tablosunda kullanılabilecek bir değere nasıl dönüştürülür? 3. Bir hash tablosu ile uygun konumu bulduğunuzda, aradığınız öğenin tabloda olup olmadığını nasıl anlarsınız? Buna tam olarak cevap vermeye dikkat edin. 4. Hash tablosu ile çalışırken neden bir çarpışma çözümleme stratejisine ihtiyaç duyulur? 5. Harita ile set arasındaki fark nedir? 6. Bu bölümde HashMap sınıfını uygulamak için HashSet kullanıldı. Peki ya işleri tersine çevirirsek? Python'da bir sözlük bir kümeyi gerçekleştirmek için nasıl kullanılabilir? Bunun nasıl yapılabileceğini, bir kümenin add ve membership yöntemlerini ve dahili olarak küme bir sözlük kullanırsa bunların nasıl uygulanacağını açıklayarak anlatın. 7. Yük faktörü, küme veri türünde üyelik testinin karmaşıklığını nasıl etkiler? 8. Tekrarlamak nedir? 9. Memoizasyon ne zaman etkili bir programlama tekniğidir? 10. Doğru ya da yanlış: Memoizasyon faktöriyel fonksiyonunun daha hızlı çalışmasına yardımcı olur mu? Cevabınızı gerekçelendirin.



5.12 Programlama Problemleri

1. Bölümde anlatıldığı gibi Sudoku bulmacasını tamamlayın. Program bir metin dosyasını okumalıdır. Kullanıcıdan metin dosyasının adını isteyin. Metin dosyası programınız tarafından kolayca bulunabilmesi için programla aynı dizine veya klasöre yerleştirilmelidir. Metnin web sitesinde çözebileceğiniz altı örnek Sudoku bulmacası bulunmaktadır. Programı, metnin web sitesinde bulduklarınıza benzer bir metin dosyasını okuyacak şekilde yazın. Hem çözülmemiş hem de çözülmüş problemi aşağıda gösterildiği gibi ekrana yazdırın.



2. Bölümde bulunan HashSet sınıfını, iki küme işlemi tablosunda açıklanan yöntemleri uygulayarak tamamlayın. Ardından, bu işlemleri test etmek için bir ana fonksiyon yazın. Sınıfı hashset.py adlı bir dosyaya kaydedin, böylece diğer programlara aktarılabilir. Ana fonksiyonunuzu hashset.py dosyasında if ismiyle çağırırsanız == " main " deyiminden sonra başka bir programa aktardığınızda hashset.py ana fonksiyonunuz çalıştırılmayacak, ancak hashset.py'yi kendi başına çalıştırdığınızda ana fonksiyonu HashSet sınıfınızı test etmek için çalışacaktır.

3. Bölüm 3'teki tic tac toe programının performansını artırmak için ezberleyin. Bunu yapmak için her tahtanın bir hash değeri olmalıdır. Board sınıfı için bir hash yöntemi uygulamalısınız. Hash değeri bir tahtanın yapılandırması için benzersiz olmalıdır. Başka bir deyişle, X'ler, O'lar ve Kukla nesneler tahtanın hash değerine dahil edilmelidir, böylece her tahtanın kendine özgü bir hash değeri olur. Ardından, belirli bir kartın yapılandırması için bulunan değeri hatırlamak üzere minimax fonksiyonunu hafızaya alın. Minimax fonksiyonu, bu pano için değerin daha önce hesaplanıp hesaplanmadığını kontrol ederek başlamalı ve hesaplanmışsa fonksiyon değeri döndürmelidir.

4. HashSet sınıfının izin verilen maksimum ve minimum yük faktörünü belirlemenize olanak tanıyan bir versiyonunu yazın. Ardından, farklı maksimum yük faktörleri verilen bir öğeyi kümeye eklemek için geçen ortalama süreyi çizdiğiniz bir dizi test yapın. Ayrıca, farklı maksimum yük faktörleri için bir öğenin bir kümeye üyeliğini test etmek için geçen ortalama süre hakkında bilgi toplayın. Bu bilgilerden hash tablolarındaki alan/zaman dengesinin bir kısmını görebilmeniz gerekir. Bu deneysel sonuçlardan çizim formatında XML verileri oluşturun ve size ne anlattığını görmek için verileri çizin. Toplanan bilgilerden yola çıkarak, HashSet sınıfı için en uygun yük faktörü hakkındaki görüşünüzü ifade edin. Testlerinizi gerçekleştiren programın üst kısmında optimum maksimum yük faktörü hakkında yorum yapım.