BLM267

Bölüm 15: Karma İşlem ve Çarpışma

C Kullanarak Veri Yapıları, İkinci Baskı

- giriiş
- Karma Tablolar
- Karma Fonksiyonlar
- Çarpışmalar

giriiş

- Doğrusal aramanın çalışma süresi O(n) ile orantılıyken, ikili aramanın çalışma süresi O(log n) ile orantılıdır; burada n dizideki eleman sayısıdır.
- İkili arama ve ikili arama ağaçları bir elemanı aramak için kullanılan etkili algoritmalardır.
- Peki ya arama işlemini O(1) ile orantılı bir sürede gerçekleştirmek istersek?
- Başka bir deyişle, bir dizinin boyutundan bağımsız olarak, sabit sürede arama yapmanın bir yolu var mıdır?

giriş

- Bu sorunun iki çözümü var.
- İlk çözümü açıklamak için bir örnek alalım. 100 çalışanı olan küçük bir şirkette, her çalışana 0–99 aralığında bir Emp_ID atanır.
- Kayıtları bir dizide depolamak için, her çalışanın Emp_ID'si, Şekil 15.1'de gösterildiği gibi, çalışanın kaydının depolanacağı dizinin bir indeksi görevi görür.
- Bu durumda, Emp_ID numarasını bildiğimiz herhangi bir çalışanın kaydına doğrudan erişebiliriz, çünkü dizi indeksi Emp_ID numarasıyla aynıdır.
- Ancak pratikte bu uygulamanın pek mümkün olduğu söylenemez.

Key	Array of Employees' Records
Key 0 → [0]	Employee record with Emp_ID 0
Key 1 → [1]	Employee record with Emp_ID 1
Key 2 → [2]	Employee record with Emp_ID 2
.....	.....
.....	.....
Key 98 → [98]	Employee record with Emp_ID 98
Key 99 → [99]	Employee record with Emp_ID 99

giriş

- Aynı şirketin birincil anahtar olarak beş haneli bir Emp_ID kullandığını varsayalım.
- Bu durumda anahtar değerleri 00000 ile 99999 arasında değişecektir. Yukarıdakiyle aynı tekniği kullanmak istersek, yalnızca 100 elemanı kullanılacak 100.000 boyutunda bir diziye ihtiyacımız olacak.
-

Key	Array of Employees' Records
Key 00000 → [0]	Employee record with Emp_ID 00000
.....	.....
Key n → [n]	Employee record with Emp_ID n
.....	.....
Key 99998 → [99998]	Employee record with Emp_ID 99998
Key 99999 → [99999]	Employee record with Emp_ID 99999

giriş

- Her çalışanın kaydının benzersiz ve öngörülebilir bir konumda olmasını sağlamak için bu kadar çok depolama alanını boşa harcamak pratik değildir.
- İster iki haneli birincil anahtar (Emp_ID) ister beş haneli anahtar kullanalım, şirkette sadece 100 çalışan var.
- Bu nedenle, dizide yalnızca 100 konum kullanacağız. Bu nedenle, dizi boyutunu gerçekte kullanacağımız boyuta (100 eleman) düşürmek için bir diğer iyi seçenek, her çalışanı tanımlamak için anahtarın yalnızca son iki basamağını kullanmaktır.
- Örneğin Emp_ID'si 79439 olan çalışan, dizinin 39 indeksli elemanında saklanacaktır.
- Benzer şekilde Emp_ID'si 12345 olan çalışanın kaydı dizide 45. konumda saklanacaktır.
- İkinci çözümde ise elemanlar anahtar değerine göre saklanmıyor.
- *Bu durumda beş basamaklı bir anahtar sayıyı iki basamaklı bir dizi indeksine dönüştürmenin bir yoluna ihtiyacımız var.*
- Dönüşümü yapacak bir fonksiyona ihtiyacımız var. Bu durumda, dizi için karma tablo terimini kullanacağız ve dönüşümü gerçekleştirecek fonksiyona karma fonksiyon adı verilecek.

Karma Tablolar

- Karma tablo, anahtarların bir karma fonksiyonu ile dizi konumlarına eşlendiği bir veri yapısıdır.
- Burada tartışılan örnekte anahtarın son iki basamağını çıkaran bir karma fonksiyonu kullanacağız.
- Bu nedenle anahtarları dizi konumlarına veya dizi indekslerine eşliyoruz.
- Bir karma tabloda saklanan bir değer, anahtardan bir adres üreten bir karma işlevi kullanılarak (değerin saklandığı dizinin dizinini üreterek) O(1) sürede aranabilir.
- Şekil 15.3, anahtarlar ve dizinin dizinleri arasında doğrudan bir ilişki olduğunu gösterir. Bu kavram, anahtarların toplam evreni küçük olduğunda ve anahtarların çoğu aslında tüm anahtar kümesinden kullanıldığında faydalıdır.
- Bu, 100 çalışan için 100 anahtarın bulunduğu ilk örneğimize eşdeğerdir.

Karma Tablolar

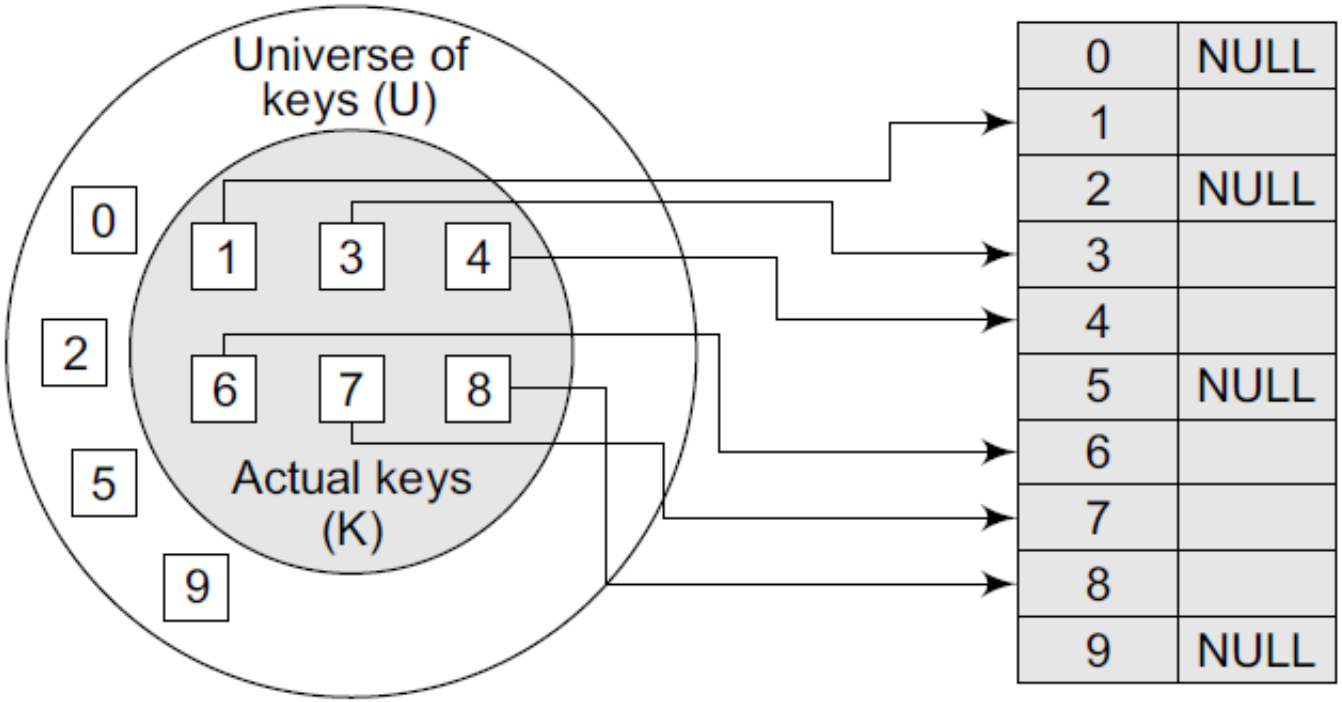


Figure 15.3 Direct relationship between key and index in the array

Karma Tablolar

- Ancak gerçekte kullanılan anahtarların K kümesi, anahtarların evreninden (U) daha küçük olduğunda, bir karma tablosu daha az depolama alanı tüketir.
- Bir karma tablosunun depolama gereksinimi O(k)'dır; burada k gerçekte kullanılan anahtar sayısıdır.
- Bir karma tabloda, k anahtarına sahip bir eleman k dizininde değil h(k) dizininde saklanır.
- Bu, h adlı bir karma fonksiyonunun, k anahtarına sahip elemanın saklanacağı indeksi hesaplamak için kullanıldığı anlamına gelir.
- Anahtarları bir karma tablosundaki uygun konumlara (veya endekslere) eşleme işlemine karma oluşturma denir.
- Şekil 15.4, K kümesindeki her anahtarın bir karma işlevi kullanılarak oluşturulan konumlara eşlendiği bir karma tablosunu göstermektedir.
- k2 ve k6 tuşlarının aynı bellek konumuna işaret ettiğini unutmayın.
- Buna çarpışma denir. Yani, iki veya daha fazla anahtar aynı bellek konumuna eşlendiğinde, bir çarpışmanın meydana geldiği söylenir.
- Benzer şekilde, k5 ve k7 anahtarları da çakışır. Bir karma işlevi kullanmanın temel amacı, işlenmesi gereken dizi dizinlerinin aralığını azaltmaktır.
- Dolayısıyla U değerleri yerine sadece K değerlerine ihtiyacımız

Karma Tablolar

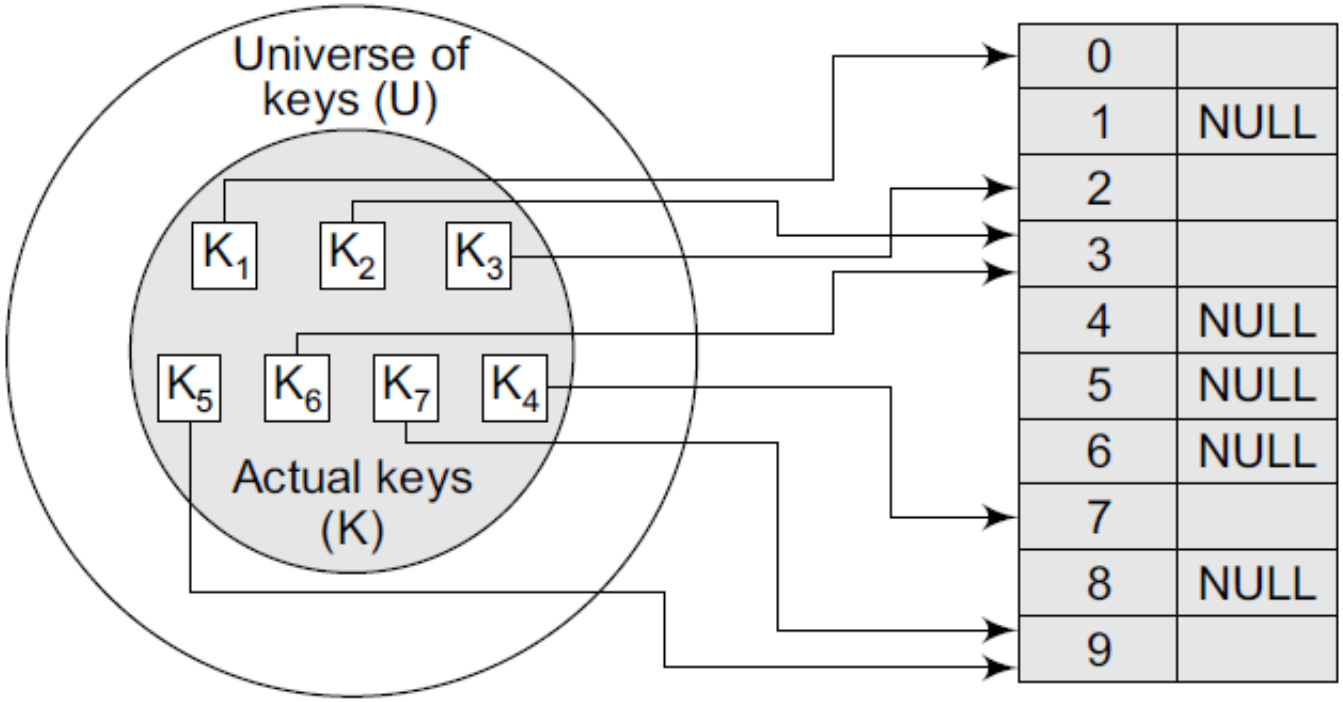


Figure 15.4 Relationship between keys and hash table index

Karma Fonksiyonlar

- Bir karma fonksiyonu, bir anahtara uygulandığında karma tablosundaki anahtar için bir indeks olarak kullanılabilecek bir tam sayı üreten matematiksel bir formüldür.
- Bir karma fonksiyonunun temel amacı, elemanların göreceli, rastgele ve düzgün bir şekilde dağıtılmasıdır.
- Çarpışma sayısını azaltmak için uygun bir aralıkta benzersiz bir tam sayı kümesi üretir.
- Pratikte, çarpışmaları tamamen ortadan kaldıran bir karma fonksiyonu bulunmamaktadır.
- İyi bir karma fonksiyonu, elemanları dizi boyunca eşit şekilde yayarak çarpışma sayısını en aza indirebilir.
- Bu bölümde, çarpışmaları en aza indirmeye yardımcı olan popüler karma fonksiyonlarını tartışacağız.
- Ancak ondan önce, iyi bir hash fonksiyonunun özelliklerine bakalım.

Karma Fonksiyonlar

- ***İyi Bir Karma Fonksiyonunun Özellikleri***
- ***Düşük maliyet Bir karma fonksiyonunu yürütmenin maliyeti düşük olmalıdır, böylece karma tekniğinin kullanılması diğer yaklaşımlara göre tercih edilir hale gelir.***
- Örneğin, ikili arama algoritması n öğeli sıralı bir tablodan bir elemanı $\log_2 n$ anahtar karşılaştırmasıyla arayabilseydi, o zaman karma işlevinin maliyeti $\log_2 n$ anahtar karşılaştırması yapmaktan daha az olurdu.
- ***Determinizm Bir karma prosedürü deterministik olmalıdır.***
- Bu, verilen bir giriş değeri için aynı karma değerinin üretilmesi gerektiği anlamına gelir.
- Ancak bu ölçüt, harici değişken parametrelerine (günün saati gibi) ve karması oluşturulan nesnenin bellek adresine (çünkü nesnenin adresi işleme sırasında değişebilir) bağlı olan karma işlevlerini hariç tutar.
- ***Tekdüzelik İyi bir karma fonksiyonu, anahtarları çıkış aralığı boyunca mümkün olduğunca eşit bir şekilde eşlemelidir.***
- Bu, çıktı aralığındaki her karma değerinin üretilme olasılığının kabaca aynı olması gerektiği anlamına gelir.
- Tekdüzelik özelliği aynı zamanda çarpışma sayısını da en aza indirir.

Farklı Karma Fonksiyonları

- Bu bölümde sayısal anahtarları kullanan karma fonksiyonlarını ele alacağız.
- Ancak gerçek dünya uygulamalarında basit sayısal anahtarlar yerine alfanümerik anahtarların da kullanılabileceği durumlar olabilir.
- Bu gibi durumlarda karakterin ASCII değeri kullanılarak eşdeğer sayısal anahtara dönüştürülebilir.
- Bu dönüşüm yapıldıktan sonra, aşağıda verilen hash fonksiyonlarından herhangi biri uygulanarak hash değeri üretilebilir.

Farklı Karma Fonksiyonları

- **Bölme Yöntemi**
- Bir tam sayı olan x'in karmasını hesaplamanın en basit yöntemidir.
- *Bu yöntemde x, M'ye bölünür ve kalan kullanılır.*
- Bu durumda, karma işlevi şu şekilde verilebilir:

  h(x) = xmod M
- Bölme yöntemi M'nin hemen hemen her değeri için oldukça iyidir ve yalnızca tek bir bölme işlemi gerektirdiğinden yöntem çok hızlı çalışır.
- Ancak M için uygun bir değerin seçilmesine ekstra özen gösterilmelidir.
- Örneğin, M çift sayı ise, x çiftse h(x) çifttir ve x tekse h(x) tektir.
- Eğer tüm olası anahtarlar eşit olasılıklı ise bu bir sorun değildir.
- Ancak çift anahtarların olasılığı tek anahtarlardan daha fazlaysa, bölme yöntemi karma değerlerini eşit olarak yaymayacaktır.

Farklı Karma Fonksiyonları

- **Bölme Yöntemi**
- Genellikle, M'yi asal sayı olarak seçmek en iyisidir çünkü M'yi asal sayı yapmak, anahtarların çıktı değer aralığında bir tekdüzelikle eşlenme olasılığını artırır.
- M aynı zamanda 2'nin tam kuvvetlerine çok yakın olmamalıdır.
- Eğer bizde varsa

  h(x) = x mod 2k
- o zaman fonksiyon x'in ikili gösteriminin en düşük k bitini çıkaracaktır.
- Bölme yönteminin uygulanması son derece basittir.
- Aşağıdaki kod parçası bunun nasıl yapılacağını göstermektedir:

```
int const M = 97; // bir asal sayı
int h (int x)
{ x % M'yi döndür; }
```

Farklı Karma Fonksiyonları

- **Bölme Yöntemi**
- Bölme yönteminin potansiyel bir dezavantajı, bu yöntemi kullanırken ardışık anahtarların ardışık karma değerlerine eşlenmesidir.
- Bu bir yandan ardışık anahtarların çakışmamasını sağladığı için iyi bir şey, ancak diğer yandan ardışık dizi konumlarının işgal edileceği anlamına da geliyor.
- Bu durum performansın düşmesine yol açabilir.

Example 15.1 Calculate the hash values of keys 1234 and 5462.

Solution Setting `M = 97`, hash values can be calculated as:

```
h(1234) = 1234 % 97 = 70
h(5642) = 5642 % 97 = 16
```

Farklı Karma Fonksiyonları

- **Çarpma Yöntemi**
- Çarpma yönteminde izlenen adımlar şu şekildedir:
- *Adım 1: 0 < A < 1 olacak şekilde bir A sabiti seçin.*
- *Adım 2: k anahtarını A ile çarpın.*
- *Adım 3: kA'nın kesirli kısmını çıkarın.*
- *Adım 4: Adım 3'ün sonucunu karma tablosunun boyutuyla (m) çarpın.*
- Dolayısıyla, karma fonksiyonu şu şekilde verilebilir:

  h(k) = m (kA mod 1)

  Burada (kA mod 1) kA'nın kesirli kısmını verir ve m ise karma tablodaki toplam endeks sayısını ifade eder.

Farklı Karma Fonksiyonları

- **Çarpma Yöntemi**
- Bu yöntemin en büyük avantajı, A'nın hemen hemen her değeriyle çalışabilmesidir.
- Algoritma bazı değerlerle daha iyi çalışsa da, en iyi seçim, karması alınan verinin özelliklerine bağlıdır.
- Knuth, A'nın en iyi seçiminin şu olduğunu ileri sürdü:
  (sqrt5 – 1) /2 = 0,6180339887

Example 15.2 Given a hash table of size 1000, map the key 12345 to an in the hash table.

Solution We will use A = 0.618033, m = 1000, and k = 12345

$$
\begin{aligned}
h(12345) &= \lfloor\ 1000\ (12345 \times 0.618033 \bmod 1)\ \rfloor \\
&= \lfloor\ 1000\ (7629.617385 \bmod 1)\ \rfloor \\
&= \lfloor\ 1000\ (0.617385)\ \rfloor \\
&= \lfloor\ 617.385\ \rfloor \\
&= 617
\end{aligned}
$$

Farklı Karma Fonksiyonları

- **Orta Kare Yöntemi**
- Orta kare yöntemi iki adımda çalışan iyi bir karma fonksiyonudur:
- *Adım 1: Anahtarın değerini kareleyin. Yani k^2'yi bulun.*
- *Adım 2: Adım 1'de elde edilen sonucun ortadaki r rakamını çıkarın.*
- Algoritma iyi çalışıyor çünkü anahtar değerinin çoğu veya tüm basamakları sonuca katkıda bulunuyor.
- Bunun nedeni, orijinal anahtar değerindeki tüm rakamların kare değerinin orta rakamlarının üretilmesine katkıda bulunmasıdır.
- Dolayısıyla sonuç, orijinal anahtar değerinin alt basamağının veya üst basamağının dağılımı tarafından belirlenmez.
- Orta kare yönteminde tüm anahtarlardan aynı r rakamının seçilmesi gerekir.
- Dolayısıyla, karma fonksiyonu şu şekilde verilebilir:

  h(k) = s

  Burada s, k^2'den r basamak seçilerek elde edilir.

Farklı Karma Fonksiyonları

- **Orta Kare Yöntemi**

Example 15.3 Calculate the hash value for keys 1234 and 5642 using the mid-squa
The hash table has 100 memory locations.

Solution Note that the hash table has 100 memory locations whose indices vary fro
This means that only two digits are needed to map the key to a location in the hash tab

When $k = 1234$, $k^2 = 1522756$, $h(1234) = 27$

When $k = 5642$, $k^2 = 31832164$, $h(5642) = 21$

Observe that the 3rd and 4th digits starting from the right are chosen.

Farklı Karma Fonksiyonları

- **Katlama Yöntemi**
- Katlama yöntemi aşağıdaki iki adımda çalışır:
- *Adım 1: Anahtar değerini birkaç parçaya bölün.*
- *Yani k'yi k_1, k_2, ..., kn parçalarına bölün, burada her parçanın rakam sayısı aynı olsun, son parça hariç; son parça diğer parçalardan daha az rakam içerebilir.*
- *Adım 2: Bireysel parçaları ekleyin. Yani, $k_1 + k_2 + ... + kn$ toplamını elde edin.*
- *Hash değeri, varsa son carry'nin göz ardı edilmesiyle üretilir.*
- Anahtarın her bir bölümündeki rakam sayısının, karma tablosunun boyutuna bağlı olarak değişeceğini unutmayın.
- Örneğin, hash tablosunun boyutu 1000 ise, hash tablosunda 1000 konum vardır.
- Bu 1000 lokasyonu adreslemek için en az üç rakama ihtiyacımız var; dolayısıyla anahtarın her bir parçası, son kısım hariç, üç rakamdan oluşmalıdır; son kısım daha az rakam içerebilir.

Farklı Karma Fonksiyonları

- **Katlama Yöntemi**

Example 15.4 Given a hash table of 100 locations, calculate the hash value using folding method for keys 5678, 321, and 34567.

Solution

Since there are 100 memory locations to address, we will break the key into parts where each part (except the last) will contain two digits. The hash values can be obtained as shown below:

key	5678	321	34567
Parts	56 and 78	32 and 1	34, 56 and 7
Sum	134	33	97
Hash value	34 (ignore the last carry)	33	97

Çarpışmalar

- Bu bölümde daha önce tartışıldığı gibi, çakışmalar, karma işlevi iki farklı anahtarı aynı konuma eşlediğinde meydana gelir.
- Elbette iki kayıt aynı yerde saklanamaz.
- Bu nedenle çarpışma problemini çözmede kullanılan, çarpışma çözümleme tekniği olarak da adlandırılan bir yöntem uygulanmaktadır.
- Çarpışmaları çözmenin en popüler iki yöntemi şunlardır:
    1. Açık adresleme
    2. Zincirleme
- Bu bölümde bu iki tekniği de detaylı olarak ele alacağız.

Çarpışmalar

- **Açık Adresleme ile Çarpışma Çözümü**
- Bir çarpışma gerçekleştiğinde, açık adresleme veya kapalı karma, bir araştırma dizisi kullanarak yeni konumları hesaplar ve bir sonraki kayıt o konumda saklanır.
- Bu teknikte tüm değerler hash tablosunda saklanır.
- Karma tablo iki tür değer içerir: bekçi değerleri (örneğin –1) ve veri değerleri.
- *Bir bekçi değerinin varlığı, konumun şu anda herhangi bir veri değeri içermediğini ancak bir değeri tutmak için kullanılabileceğini gösterir.*
- Bir anahtar belirli bir bellek konumuna eşlendiğinde, tuttuğu değer kontrol edilir.
- Eğer bir sentinel değeri içeriyorsa, o zaman konum boştur ve veri değeri burada saklanabilir.
- Ancak eğer lokasyonda halihazırda depolanmış bir veri değeri varsa, boş bir slot bulmak için ileri yönde diğer slotlar sistematik olarak incelenir.
- Eğer tek bir boş yer bile bulunamazsa OVERFLOW durumu söz konusudur.
- Karma tablodaki bellek konumlarını inceleme sürecine probing denir.
- *Açık adresleme tekniği doğrusal araştırma, karesel araştırma, çift karma ve yeniden karma yöntemleri kullanılarak uygulanabilir.*

Çarpışmalar

- ***Doğrusal araştırma***
- Bir çarpışmayı çözmenin en basit yaklaşımı doğrusal araştırmadır.
- Bu teknikte, eğer bir değer h(k) tarafından oluşturulan bir konumda zaten depolanmışsa, çarpışmayı çözmek için aşağıdaki karma işlevi kullanılır:

  $$h(k, i) = [h'(k) + i] \bmod m$$

  burada m, karma tablonun boyutudur,
  $h'(k) = (k \bmod m)$ ve i, 0 ile m–1 arasında değişen prob numarasıdır.

Çarpışmalar

- ***Doğrusal araştırma***
- Bu nedenle, belirli bir k anahtarı için, ilk olarak [h′ (k) mod m] tarafından üretilen konum araştırılır, çünkü ilk kez i=0 olur.
- Eğer konum boş ise değer burada saklanır, aksi takdirde ikinci prob [h′ (k) + 1]mod m ile verilen konumun adresini üretir.
- Benzer şekilde, eğer konum işgal edilmişse, sonraki araştırmalar adresi [h′(k) + 2]mod m, [h′(k) + 3]mod m, [h′(k) + 4]mod m, [h′(k)+ 5]mod m şeklinde üretir, böylece boş bir konum bulunur.

Çarpışmalar

- ***Doğrusal araştırma***

Note Linear probing is known for its simplicity. When we have to store a value, we try the slots: [h′(k)] mod m, [h′(k) + 1]mod m, [h′(k) + 2]mod m, [h′(k) + 3]mod m, [h′(k) + 4]mod m, [h′(k) + 5]mod m, and so no, until a vacant location is found.

Example 15.5 Consider a hash table of size 10. Using linear probing, insert the keys 72, 27, 36, 24, 63, 81, 92, and 101 into the table.

Let h′(k) = k mod m, m = 10

Initially, the hash table can be given as:

0	1	2	3	4	5	6	7	8	9
–1	–1	–1	–1	–1	–1	–1	–1	–1	–1

Step 1

Key = 72

h(72, 0) = (72 mod 10 + 0) mod 10

= (2) mod 10

= 2

Since T[2] is vacant, insert key 72 at this location.

Çarpışmalar

- ***Doğrusal araştırma***

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	–1	–1	–1	–1

Step 2

```
Key = 27
h(27, 0) = (27 mod 10 + 0) mod 10
         = (7) mod 10
         = 7
```

Since `T[7]` is vacant, insert key 27 at this location.

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	–1	27	–1	–1

Step 3

```
Key = 36
h(36, 0) = (36 mod 10 + 0) mod 10
         = (6) mod 10
         = 6
```

Since `T[6]` is vacant, insert key 36 at this location.

Çarpışmalar

- ***Doğrusal araştırma***

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	36	27	–1	–1

Step 4 Key = 24

h(24, 0) = (24 mod 10 + 0) mod 10

= (4) mod 10

= 4

Since T[4] is vacant, insert key 24 at this location.

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	24	–1	36	27	–1	–1

Step 5 Key = 63

h(63, 0) = (63 mod 10 + 0) mod 10

= (3) mod 10

= 3

Since T[3] is vacant, insert key 63 at this location.

Çarpışmalar

- ***Doğrusal araştırma***

0	1	2	3	4	5	6	7	8	9
–1	–1	72	63	24	–1	36	27	–1	–1

Step 6 Key = 81

h(81, 0) = (81 mod 10 + 0) mod 10

= (1) mod 10

= 1

Since T[1] is vacant, insert key 81 at this location.

Çarpışmalar

- ***Doğrusal araştırma***

0	1	2	3	4	5	6	7	8	9
0	81	72	63	24	–1	36	27	–1	–1

Step 7

```
Key = 92
h(92, 0) = (92 mod 10 + 0) mod 10
         = (2) mod 10
         = 2
```

Now `T[2]` is occupied, so we cannot store the key 92 in `T[2]`. Therefore, try again for the next location. Thus probe, `i = 1`, this time.

```
Key = 92
h(92, 1) = (92 mod 10 + 1) mod 10
         = (2 + 1) mod 10
         = 3
```

Now `T[3]` is occupied, so we cannot store the key 92 in `T[3]`. Therefore, try again for the next location. Thus probe, `i = 2`, this time.

```
Key = 92
h(92, 2) = (92 mod 10 + 2) mod 10
         = (2 + 2) mod 10
         = 4
```

Now `T[4]` is occupied, so we cannot store the key 92 in `T[4]`. Therefore, try again for the next location. Thus probe, `i = 3`, this time.

```
Key = 92
h(92, 3) = (92 mod 10 + 3) mod 10
         = (2 + 3) mod 10
         = 5
```

Since `T[5]` is vacant, insert key 92 at this location.

Çarpışmalar

- ***Doğrusal araştırma***

0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	92	36	27	–1	–1

Step 8

```
Key = 101
h(101, 0) = (101 mod 10 + 0) mod 10
          = (1) mod 10
          = 1
```

Now `T[1]` is occupied, so we cannot store the key 101 in `T[1]`. Therefore, try again for location. Thus probe, `i = 1`, this time.

```
Key = 101
h(101, 1) = (101 mod 10 + 1) mod 10
          = (1 + 1) mod 10
          = 2
```

`T[2]` is also occupied, so we cannot store the key in this location. The procedure will be until the hash function generates the address of location 8 which is vacant and can b store the value in it.

Çarpışmalar

- ***Doğrusal Araştırma Kullanarak Bir Değer Arama***
- Bir karma tabloda bir değeri arama prosedürü, bir karma tabloda bir değeri depolama prosedürüyle aynıdır.
- Bir hash tablosunda bir değer aranırken, dizi indeksi yeniden hesaplanır ve o konumda saklanan elemanın anahtarı, aranacak değerle karşılaştırılır.
- Bir eşleşme bulunursa, arama işlemi başarılı olur. Bu durumda arama süresi O(1) olarak verilir.
- Anahtar eşleşmezse, arama fonksiyonu dizi içinde şu ana kadar devam eden ardışık bir arama başlatır:
  - değer bulundu veya
  - arama fonksiyonu dizide boş bir konumla karşılaşır, bu değerin mevcut olmadığını gösterir veya arama fonksiyonu tablonun sonuna ulaştığında değer mevcut olmadığı için sonlanır.

Çarpışmalar

- ***Doğrusal Araştırma Kullanarak Bir Değer Arama***
- En kötü durumda, arama işlemi n–1 karşılaştırma yapmak zorunda kalabilir ve arama algoritmasının çalışma süresi O(n) zaman alabilir.
- En kötü durum, n–1 elemanın tümü tarandıktan sonra değerin son konumda mevcut olması veya tabloda mevcut olmaması durumunda ortaya çıkar.
- Böylece, çarpışma sayısının artmasıyla birlikte, hash fonksiyonu tarafından hesaplanan dizi indeksi ile elemanın gerçek konumu arasındaki mesafenin arttığını, dolayısıyla arama süresinin arttığını görüyoruz.

Çarpışmalar

- ***Artıları ve Eksileri***
- Doğrusal araştırma, h(k) konumundan başlayarak dizide doğrusal bir arama yaparak boş bir konum bulur.
- Algoritma, iyi referans yerelliği sayesinde iyi bir bellek önbelleğe alma sağlasa da, bu algoritmanın dezavantajı kümelemeye yol açması ve dolayısıyla bir çarpışmanın daha önce gerçekleştiği yerde daha fazla çarpışma riskinin daha yüksek olmasıdır.
- Doğrusal sondajın performansı giriş değerlerinin dağılımına duyarlıdır.
- Karma tablo doldukça, ardışık hücrelerden oluşan kümeler oluşur ve arama için gereken süre, kümenin büyüklüğüne bağlı olarak artar.
- Ayrıca tabloda halihazırda dolu olan bir pozisyona yeni bir değer girilmesi gerektiğinde, bu değer kümenin sonuna eklenir, bu da yine kümenin uzunluğunu artırır.

Çarpışmalar

- ***Artıları ve Eksileri***
- Genellikle, bir boş yerle ayrılmış iki küme arasına bir ekleme yapılır.
- Ancak doğrusal sondajla, sonraki eklemelerin de kümelerden birinde sonlanma olasılığı daha yüksektir; bu da küme uzunluğunu birden çok daha fazla miktarda artırabilir.
- Çarpışma sayısı arttıkça, boş bir yer bulmak için gereken prob sayısı artacak ve performans düşecektir.
- Bu olguya birincil kümelenme adı verilir.
- *Birincil kümelemeyi önlemek için, karesel araştırma ve çift karmalama gibi diğer teknikler kullanılır.*

Çarpışmalar

- ***İkinci Dereceden Araştırma***
- Bu teknikte, eğer bir değer h(k) tarafından oluşturulan bir konumda zaten depolanmışsa, çarpışmayı çözmek için aşağıdaki karma işlevi kullanılır:

  $$h(k, i) = [h'(k) + c_1 i + c_2 i^2] \bmod m$$

  burada m, karma tablonun boyutudur,

  $h'(k) = (k \bmod m)$, i 0 ile m–1 arasında değişen prob numarasıdır ve c_1 ve c_2, c_1 ve $c_2 \neq 0$ olacak şekilde sabitlerdir.

Çarpışmalar

- ***İkinci Dereceden Araştırma***
- Karesel araştırma, doğrusal araştırmanın birincil kümeleme olgusunu ortadan kaldırır, çünkü doğrusal arama yapmak yerine karesel arama yapar.
- Verilen bir k anahtarı için, öncelikle h′(k) mod m tarafından üretilen konum araştırılır.
- Eğer konum boş ise, değer burada saklanır, aksi takdirde daha sonra araştırılan konumlar, araştırma numarası i'ye ikinci dereceden bağlı faktörler tarafından dengelenir.
- Karesel araştırma doğrusal araştırmadan daha iyi performans gösterse de, karma tablonun kullanımını en üst düzeye çıkarmak için c_1, c_2 ve m değerlerinin sınırlandırılması gerekir.

Çarpışmalar

- ***İkinci Dereceden Araştırma***

Example 15.6 Consider a hash table of size 10. Using quadratic probing, insert the keys 72, 27, 36, 24, 63, 81, and 101 into the table. Take $c_1 = 1$ and $c_2 = 3$.

Solution

Let $h'(k) = k \bmod m$, $m = 10$

Initially, the hash table can be given as:

0	1	2	3	4	5	6	7	8	9
–1	–1	–1	–1	–1	–1	–1	–1	–1	–1

We have,

$$h(k, i) = [h'(k) + c_1 i + c_2 i^2] \bmod m$$

Step 1 Key = 72

$$h(72, 0) = [72 \bmod 10 + 1 \times 0 + 3 \times 0] \bmod 10$$
$$= [72 \bmod 10] \bmod 10$$
$$= 2 \bmod 10$$
$$= 2$$

Since T[2] is vacant, insert the key 72 in T[2]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	–1	–1	–1	–1

Çarpışmalar

- ***İkinci Dereceden Araştırma***

Step 2 Key = 27

$$h(27, 0) = [27 \bmod 10 + 1 \times 0 + 3 \times 0] \bmod 10$$
$$= [27 \bmod 10] \bmod 10$$
$$= 7 \bmod 10$$
$$= 7$$

Since T[7] is vacant, insert the key 27 in T[7]. The hash table now becor

0	1	2	3	4	5	6	7	
–1	–1	72	–1	–1	–1	–1	27	

Çarpışmalar

- ***İkinci Dereceden Araştırma***

Step 3 Key = 36

h(36, 0) = [36 mod 10 + 1 × 0 + 3 × 0] mod 10
= [36 mod 10] mod 10
= 6 mod 10
= 6

Since T[6] is vacant, insert the key 36 in T[6]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	36	27	–1	–1

Step 4 Key = 24

h(24, 0) = [24 mod 10 + 1 × 0 + 3 × 0] mod 10
= [24 mod 10] mod 10
= 4 mod 10
= 4

Since T[4] is vacant, insert the key 24 in T[4]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	24	–1	36	27	–1	–1

Çarpışmalar

- ***İkinci Dereceden Araştırma***

Step 5 Key = 63

h(63, 0) = [63 mod 10 + 1 × 0 + 3 × 0] mod 10
= [63 mod 10] mod 10
= 3 mod 10
= 3

Since T[3] is vacant, insert the key 63 in T[3]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	63	24	–1	36	27	–1	–1

Step 6 Key = 81

h(81,0) = [81 mod 10 + 1 × 0 + 3 × 0] mod 10
= [81 mod 10] mod 10
= 81 mod 10
= 1

Since T[1] is vacant, insert the key 81 in T[1]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	–1	36	27	–1	–1

Çarpışmalar

- ***İkinci Dereceden Araştırma***

Step 7 Key = 101

h(101,0) = [101 mod 10 + 1 × 0 + 3 × 0] mod 10

= [101 mod 10 + 0] mod 10

= 1 mod 10

= 1

Since T[1] is already occupied, the key 101 cannot be stored in T[1]. Therefore, try again for next location. Thus probe, i = 1, this time.

Key = 101

h(101,0) = [101 mod 10 + 1 × 1 + 3 × 1] mod 10

= [101 mod 10 + 1 + 3] mod 10

= [101 mod 10 + 4] mod 10

= [1 + 4] mod 10

= 5 mod 10

= 5

Since T[5] is vacant, insert the key 101 in T[5]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	101	36	27	–1	–1

Çarpışmalar

- ***İkinci Dereceden Araştırma Kullanarak Bir Değer Arama***
- Karesel sorgulama tekniği kullanılarak bir değer aranırken, dizi indeksi yeniden hesaplanır ve o konumda saklanan elemanın anahtarı, aranması gereken değerle karşılaştırılır.
- İstenen anahtar değeri o konumdaki anahtar değeriyle eşleşiyorsa, eleman hash tablosunda mevcut demektir ve aramanın başarılı olduğu söylenir.
- Bu durumda arama süresi O(1) olarak verilir.
- Ancak değer eşleşmezse, arama fonksiyonu dizi içinde şu ana kadar devam eden ardışık bir arama başlatır:
  - değer bulundu veya
  - arama fonksiyonu dizide boş bir konumla karşılaşır, bu da değerin mevcut olmadığını gösterir veya
  - tablo sonuna ulaşıldığı ve değerin mevcut olmadığı için arama fonksiyonu sonlanır.

Çarpışmalar

- ***İkinci Dereceden Araştırma Kullanarak Bir Değer Arama***
- En kötü durumda, arama işlemi n–1 karşılaştırma alabilir ve arama algoritmasının çalışma süresi O(n) olabilir.
- En kötü durum, n–1 elemanın tümü tarandıktan sonra değerin son konumda mevcut olması veya tabloda mevcut olmaması durumunda ortaya çıkar.
- Böylece, çarpışma sayısının artmasıyla birlikte, hash fonksiyonu tarafından hesaplanan dizi indeksi ile elemanın gerçek konumu arasındaki mesafenin arttığını, dolayısıyla arama süresinin arttığını görüyoruz.

Çarpışmalar

- ***Artıları ve Eksileri***
- Karesel araştırma, doğrusal araştırma tekniğinde var olan birincil kümeleme sorununu çözer.
- İkinci dereceden araştırma, referansın belirli bir yerelliğini koruduğu için iyi bir bellek önbelleği sağlar.
- Ancak doğrusal araştırma bu görevi daha iyi yapar ve daha iyi bir önbellek performansı sağlar.
- İkinci dereceden araştırmanın en büyük dezavantajlarından biri, ardışık araştırmaların dizisinin tablonun yalnızca bir kısmını keşfedebilmesi ve bu kısmın oldukça küçük olabilmesidir.
- Eğer böyle olursa, tablonun hiçbir şekilde dolu olmamasına rağmen tabloda boş bir yer bulamayacağız.
- Örnek 15.6'da 92 anahtarını girmeyi denediğinizde bu sorunla karşılaşacaksınız.

Çarpışmalar

- ***Artıları ve Eksileri***
- İkinci dereceden araştırma birincil kümelemeden bağımsız olsa da, ikincil kümeleme olarak bilinen şeye maruz kalmaktadır.
- *Yani iki anahtar arasında çarpışma olması durumunda her ikisi için de aynı sorgulama sırası izlenecektir.*
- İkinci dereceden sorgulamada, tablo doldukça çoklu çarpışma olasılığı artar.
- Bu durumla genellikle hash tablosunun aşırı dolu olması durumunda karşılaşılır.
- Berkeley Hızlı Dosya Sistemi'nde boş blokları tahsis etmek için karesel araştırma yaygın olarak uygulanır.

Çarpışmalar

- ***Çift Karma***
- Çift karma, başlangıçta bir karma değeri kullanır ve ardından boş bir konuma ulaşana kadar tekrar tekrar bir aralık ileri doğru adım atar.
- Aralık, ikinci ve bağımsız bir karma fonksiyonu kullanılarak belirlenir; bu nedenle bu fonksiyona çift karma adı verilir.
- *Çift karma işleminde tek bir fonksiyon yerine iki karma fonksiyonu kullanırız.*
- Çift karma durumunda karma fonksiyonu şu şekilde verilebilir:

$$h(k, i) = [h_1(k) + ih_2(k)] \bmod m$$

burada m karma tablosunun boyutu, $h_1(k)$ ve $h_2(k)$, $h_1(k) = k \bmod m$ olarak verilen iki karma fonksiyonudur, $h_2(k) = k \bmod m'$, i 0 ile m–1 arasında değişen prob numarasıdır ve m', m'den küçük olacak şekilde seçilir. m' = m–1 veya m–2 seçebiliriz.

Çarpışmalar

- ***Çift Karma***
- Karma tabloya k anahtarını eklememiz gerektiğinde, öncelikle verilen konumu uygulayarak araştırırız.
- $[h_1(k) \bmod m]$ çünkü ilk araştırma sırasında i = 0.
- Eğer konum boşsa, anahtar buraya yerleştirilir, aksi takdirde sonraki araştırmalar, önceki konumdan $[h_2(k) \bmod m]$ uzaklıkta konumlar üretir.
- Ofset, üretilen değere bağlı olarak her prob ile değişebileceğinden
- İkinci karma fonksiyonunda, çift karmanın performansı, ideal düzgün karma şemasının performansına çok yakındır.
- ***Artıları ve Eksileri***
- Çift karma, tekrarlanan çarpışmaları ve kümelemenin etkilerini en aza indirir.
- Yani, çift karma, birincil kümelemenin yanı sıra ikincil kümelemeyle ilişkili sorunlardan da uzaktır.

Çarpışmalar

- ***Çift Karma***

Example 15.7 Consider a hash table of size = 10. Using double hashing, insert the keys 72, 27, 36, 24, 63, 81, 92, and 101 into the table. Take h_1 = (k mod 10) and h_2 = (k mod 8).

Solution

Let m = 10

Initially, the hash table can be given as:

0	1	2	3	4	5	6	7	8	9
–1	–1	–1	–1	–1	–1	–1	–1	–1	–1

We have,

$h(k, i) = [h_1(k) + ih_2(k)] \bmod m$

Step 1 Key = 72

$h(72, 0) = [72 \bmod 10 + (0 \times 72 \bmod 8)] \bmod 10$

$= [2 + (0 \times 0)] \bmod 10$

$= 2 \bmod 10$

$= 2$

Since T[2] is vacant, insert the key 72 in T[2]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	–1	–1	–1	–1

Çarpışmalar

- ***Çift Karma***

Step 2 Key = 27

h(27, 0) = [27 mod 10 + (0 × 27 mod 8)] mod 10
= [7 + (0 × 3)] mod 10
= 7 mod 10
= 7

Since T[7] is vacant, insert the key 27 in T[7]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	–1	27	–1	–1

Step 3 Key = 36

h(36, 0) = [36 mod 10 + (0 × 36 mod 8)] mod 10
= [6 + (0 × 4)] mod 10
= 6 mod 10
= 6

Since T[6] is vacant, insert the key 36 in T[6]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	36	27	–1	–1

Çarpışmalar

- ***Çift Karma***

Step 4 Key = 24

$h(24, 0) = [24 \bmod 10 + (0 \times 24 \bmod 8)] \bmod 10$

$= [4 + (0 \times 0)] \bmod 10$

$= 4 \bmod 10$

$= 4$

Since T[4] is vacant, insert the key 24 in T[4]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	24	–1	36	27	–1	–1

Step 5 Key = 63

$h(63, 0) = [63 \bmod 10 + (0 \times 63 \bmod 8)] \bmod 10$

$= [3 + (0 \times 7)] \bmod 10$

$= 3 \bmod 10$

$= 3$

Since T[3] is vacant, insert the key 63 in T[3]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	63	24	–1	36	27	–1	–1

Çarpışmalar

- ***Çift Karma***

Step 6 Key = 81

h(81, 0) = [81 mod 10 + (0 × 81 mod 8)] mod 10
= [1 + (0 × 1)] mod 10
= 1 mod 10
= 1

Since T[1] is vacant, insert the key 81 in T[1]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	–1	36	27	–1	–1

Step 7 Key = 92

h(92, 0) = [92 mod 10 + (0 × 92 mod 8)] mod 10
= [2 + (0 × 4)] mod 10
= 2 mod 10

= [2 + (1 × 4)] mod 10
= (2 + 4) mod 10
= 6 mod 10
= 6

Now T[6] is occupied, so we cannot store the key 92 in T[6]. Therefore, try again for the next location. Thus probe, i = 2, this time.

Key = 92

h(92, 2) = [92 mod 10 + (2 × 92 mod 8)] mod 10
= [2 + (2 × 4)] mod 10
= [2 + 8] mod 10
= 10 mod 10
= 0

Since T[0] is vacant, insert the key 92 in T[0]. The hash table now becomes:

Çarpışmalar

- ***Çift Karma***

Step 8 Key = 101

h(101, 0) = [101 mod 10 + (0 × 101 mod 8)] mod 10
= [1 + (0 × 5)] mod 10
= 1 mod 10
= 1

Now T[1] is occupied, so we cannot store the key 101 in T[1]. Therefore, try again f next location. Thus probe, i = 1, this time.

Key = 101

h(101, 1) = [101 mod 10 + (1 × 101 mod 8)] mod 10
= [1 + (1 × 5)] mod 10
= [1 + 5] mod 10
= 6

Now T[6] is occupied, so we cannot store the key 101 in T[6]. Therefore, try again for th location with probe i = 2. Repeat the entire process until a vacant location is found. You w that we have to probe many times to insert the key 101 in the hash table. Although double ha is a very efficient algorithm, it always requires m to be a prime number. In our case m=10, wh not a prime number, hence, the degradation in performance. Had m been equal to 11, the algo would have worked very efficiently. Thus, we can say that the performance of the techni sensitive to the value of m.

Çarpışmalar

- ***Tekrar ısıtma***
- Karma tablo neredeyse dolduğunda, çarpışma sayısı artar ve bu da ekleme ve arama işlemlerinin performansını düşürür.
- Böyle durumlarda daha iyi bir seçenek, orijinal hash tablosunun iki katı büyüklüğünde yeni bir hash tablosu oluşturmaktır.
- Daha sonra orijinal hash tablosundaki tüm girdilerin yeni hash tablosuna taşınması gerekecektir.
- Bu, her girdinin alınması, onun yeni karma değerinin hesaplanması ve daha sonra yeni karma tablosuna eklenmesiyle yapılır.
- Tekrarlama basit bir işlem gibi görünse de oldukça masraflıdır ve bu nedenle sık sık yapılmamalıdır.
- Aşağıda verilen 5 büyüklüğündeki hash tablosunu ele alalım.
- Kullanılan karma işlevi:

  h(x) = x % 5.
- Girişleri yeni bir karma tabloya yeniden işleyin.

Çarpışmalar

- ***Tekrar ısıtma***

0	1	2	3	4
	26	31	43	17

Note that the new hash table is of 10 locations, double the size of the original table.

0	1	2	3	4	5	6	7	8	9

Now, rehash the key values from the old hash table into the new one using hash function—`h(x) = x % 10`.

0	1	2	3	4	5	6	7	8	9
	31		43			26	17		

Çarpışmalar

- **Zincirleme ile Çarpışma Çözümü**
- Zincirlemede, bir karma tablodaki her konum, o konuma karmalanmış tüm anahtar değerlerini içeren bağlı bir listeye bir işaretçi depolar.
- Yani, karma tablosundaki l konumu, l'ye karma işlemi uygulanan tüm anahtar değerlerinin bağlı listesinin başına işaret eder.
- Ancak, eğer hiçbir anahtar değer l'ye hashlenmezse, o zaman hash tablosundaki l konumu NULL içerir.
- Şekil 15.5, anahtar değerlerinin karma tablodaki bir konuma nasıl eşlendiğini ve o konuma karşılık gelen bir bağlı listede nasıl saklandığını göstermektedir.

Çarpışmalar

- **Zincirleme ile Çarpışma Çözümü**

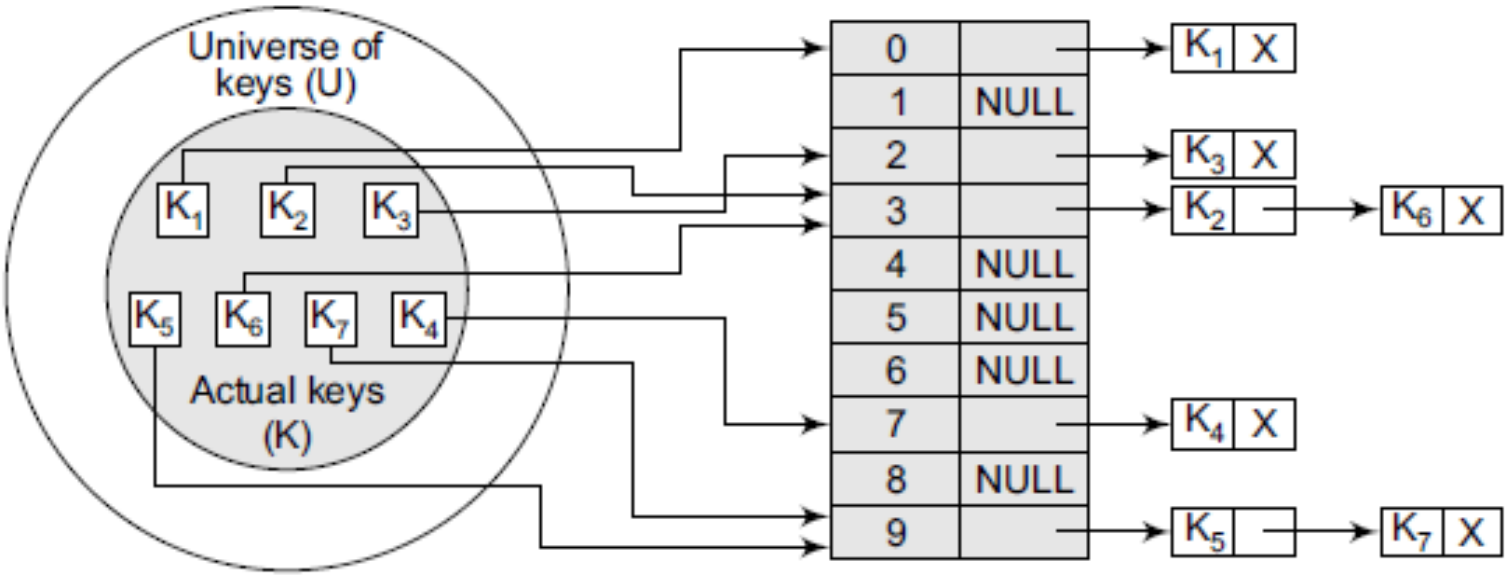


Figure 15.5 Keys being hashed to a chained hash table

Çarpışmalar

- ***Zincirlenmiş Karma Tablo Üzerindeki İşlemler***
- Zincirlenmiş bir karma tabloda bir değeri aramak, verilen anahtara sahip bir girişi bulmak için bağlı bir listeyi taramak kadar basittir.
- Ekleme işlemi, anahtarı, karma konumun işaret ettiği bağlı listenin sonuna ekler.
- Bir anahtarı silmek için listede arama yapılması ve öğenin kaldırılması gerekir.
- Bağlantılı listelerden oluşan zincirleme karma tablolar, bir anahtarın eklenmesi, silinmesi ve aranması için kullanılan algoritmaların basitliği nedeniyle yaygın olarak kullanılmaktadır.
- Bu algoritmaların kodu, 6. Bölümde incelediğimiz tek bir bağlı listeye değer ekleme, silme ve arama yapmak için kullanılan kodla tamamen aynıdır.

Çarpışmalar

- ***Zincirlenmiş Karma Tablo Üzerindeki İşlemler***
- Zincirlenmiş bir karma tabloya bir anahtar eklemenin maliyeti O(1) iken, bir değeri silmenin ve aramanın maliyeti O(m) olarak verilir; burada m, o konumdaki listedeki eleman sayısıdır.
- Arama ve silme işlemleri ise seçilen lokasyonun girdilerini istenilen anahtar için taradığından daha fazla zaman alır.
- En kötü durumda, bir değeri aramak O(n) çalışma süresi alabilir; burada n, zincirlenmiş karma tabloda depolanan anahtar değerlerinin sayısıdır.
- Bu durum, tüm anahtar değerlerinin aynı konumdaki (hash tablosunun) bağlı listeye eklenmesi durumunda ortaya çıkar.
- Bu durumda hash tablosunun bir etkisi olmayacaktır.
- Tablo 15.1, bir karma tablosunu başlatmak için kullanılan kodun yanı sıra zincirlenmiş bir karma tablosunda bir değeri eklemek, silmek ve aramak için

Çarpışmalar

- ***Zincirlenmiş Karma Tablo Üzerindeki İşlemler***

Table 15.1 Codes to initialize, insert, delete, and search a value in a chained hash table

Struture of the node

```
typedef struct node_HT
{
    int value;
    struct node *next;
}node;
```

Code to initialize a chained hash table

```
/* Initializes m location in the chained
hash table.
The operation takes a running time of
O(m) */
void initializeHashTable (node *hash_ta-
ble[], int m)
{
    int i;
    for(i=0i<=m;i++)
       hash_table[i]=NULL;
```

Code to insert a value

```
/* The element is inserted at the beginn
the linked list whose pointer to its he
stored in the location given by h(k). The
ning time of the insert operation is O(1
the new key value is always added as the
element of the list irrespective of the s
the linked list as well as that of the cl
hash table. */
node *insert_value( node *hash_table[]
val)
{
    node *new_node;
    new_node = (node *)malloc(sizeof(node
    new_node value = val; new_node next =
table[h(x)];
    hash_table[h(x)] = new_node;
}
```

Çarpışmalar

- ***Zincirlenmiş Karma Tablo Üzerindeki İşlemler***

Code to search a value

```
/* The element is searched in the linked
list whose pointer to its head is stored
in the location given by h(k). If search is
successful, the function returns a pointer
to the node in the linked list; otherwise
it returns NULL. The worst case running
time of the search operation is given as
order of size of the linked list. */
node *search_value(node *hash_table[],
int val)
{
    node *ptr;
    ptr = hash_table[h(x)];
    while ( (ptr!=NULL) && (ptr -> value
!= val))
        ptr = ptr -> next;
    if (ptr->value == val)
        return ptr;
else
        return NULL;
}
```

Code to delete a value

```
/* To delete a node from the linked list whose
head is stored at the location given by h(k)
in the hash table, we need to know the address
of the node's predecessor. We do this using a
pointer save. The running time complexity of
the delete operation is same as that of the
search operation because we need to search the
predecessor of the node so that the node can
be removed without affecting other nodes in the
list. */
void delete_value (node *hash_table[], int val)
{
    node *save, *ptr;
    save = NULL;
    ptr = hash_table[h(x)];
    while ((ptr != NULL) && (ptr value != val))
    {
        save = ptr;
        ptr = ptr next;
    }
    if (ptr != NULL)
    {
        save next = ptr next;
        free (ptr);
    }
    else
        printf("\n VALUE NOT FOUND");
}
```

Çarpışmalar

- ***Zincirlenmiş Karma Tablo Üzerindeki İşlemler***

Example 15.8 Insert the keys 7, 24, 18, 52, 36, 54, 11, and 23 in a chained hash table of 9 memory locations. Use `h(k) = k mod m`.

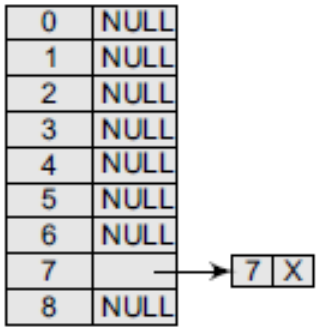
In this case, `m=9`. Initially, the hash table can be given as:

0	NULL
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	NULL
7	NULL
8	NULL

Step 1

```
Key = 7
h(k) = 7 mod 9
     = 7
```

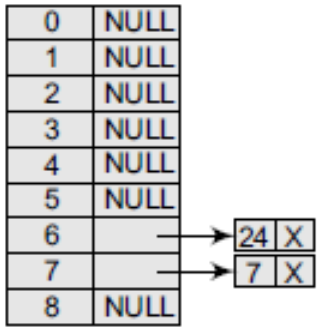
Create a linked list for location 7 and store the key value 7 in it as its only node.



Step 2

```
Key = 24
h(k) = 24 mod 9
     = 6
```

Create a linked list for location 6 and store the key value 24 in it as its only node.



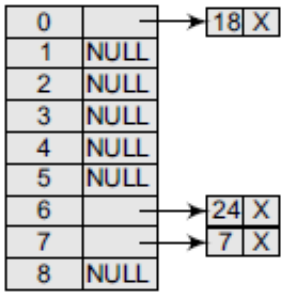
Çarpışmalar

- ***Zincirlenmiş Karma Tablo Üzerindeki İşlemler***

Step 3 `Key = 18`

`h(k)= 18 mod 9 = 0`

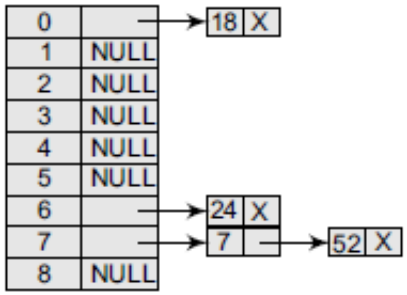
Create a linked list for location 0 and store the key value 18 in it as its only node.



Step 4 `Key = 52`

`h(k) = 52 mod 9 = 7`

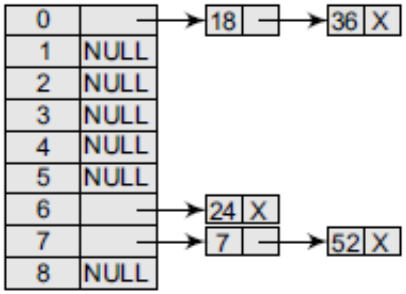
Insert 52 at the end of the linked list of location 7.



Step 5: `Key = 36`

`h(k)= 36 mod 9 = 0`

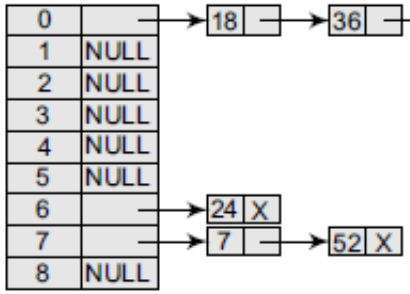
Insert 36 at the end of the linked list of location 0.



Step 6: `Key = 54`

`h(k) = 54 mod 9 = 0`

Insert 54 at the end of the linked list of location 0.



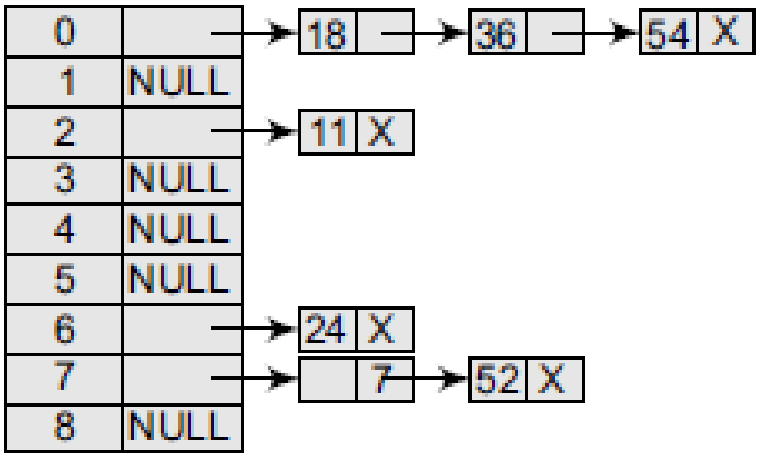
Çarpışmalar

- ***Zincirlenmiş Karma Tablo Üzerindeki İşlemler***

Step 7: Key = 11

h(k)= 11 mod 9 = 2

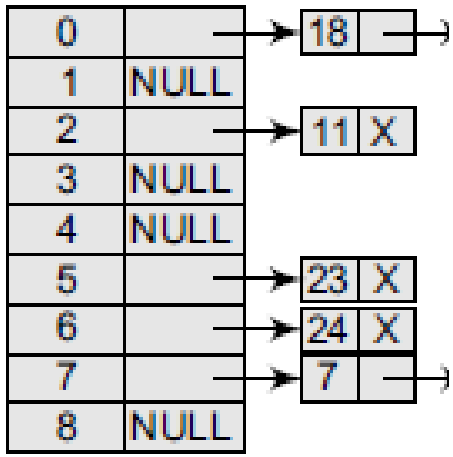
Create a linked list for location 2 and store the key value 11 in it as its only node.



Step 8: Key = 23

h(k) = 23 mo

Create a linked list fo key value 23 in it as



Çarpışmalar

- ***Artıları ve Eksileri***
- Zincirlenmiş karma tablo kullanmanın en büyük avantajı, saklanacak anahtar değerlerinin sayısı karma tablodaki konum sayısından çok daha fazla olduğunda bile etkililiğini korumasıdır.
- Ancak saklanacak anahtar sayısının artmasıyla zincirlenmiş karma tablonun performansı giderek (doğrusal olarak) azalır.
- Örneğin, 1000 bellek konumuna ve 10.000 saklanan anahtara sahip zincirlenmiş bir karma tablosu, 10.000 konuma sahip zincirlenmiş bir karma tablosuna kıyasla 5 ila 10 kat daha az performans sağlayacaktır.
- Ama zincirlenmiş bir hash tablosu hala basit bir hash tablosundan 1000 kat daha hızlıdır.
- Çarpışma çözümü için zincirlemenin kullanılmasının bir diğer avantajı da, ikinci dereceden sondajın aksine, tablonun yarısından fazlası dolu olduğunda performansının düşmemesidir.
- Bu teknik kümeleme sorunlarından tamamen uzaktır ve dolayısıyla çarpışmaları ele almak için etkili bir mekanizma sağlar.
- Ancak, zincirlenmiş karma tablolar bağlı listelerin dezavantajlarını devralır. İlk olarak, bir anahtar değerini depolamak için, her girdideki bir sonraki işaretçinin alan yükü önemli olabilir.
- İkincisi, bağlı listeyi dolaşmak düşük önbellek performansına sahiptir ve bu da işlemci önbelleğini etkisiz hale getirir.