Çakışmalar ve Çakışmaların Statik Yaklaşımlarla Çözülmesi

Çakışma (Collesion)

- Belirtilen hash fonksiyonlarından bazıları diğerlerine göre daha düzgün bir dağıtım gerçekleştirir.

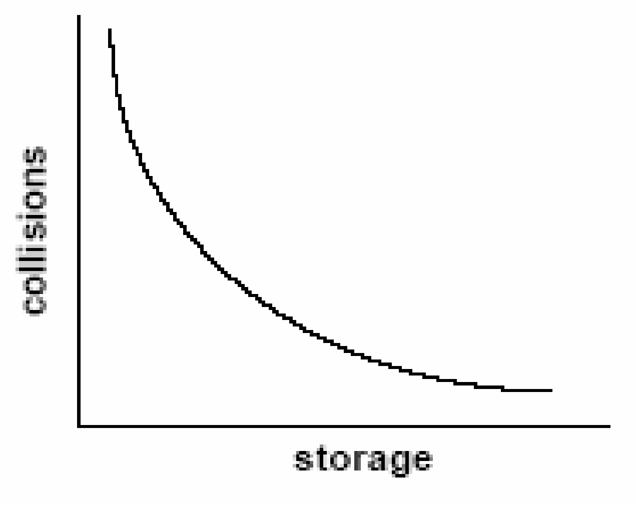
- Fakat aynı adrese çok fazla sayıda kayıt eklenmeye çalışılıyorsa çakışma miktarı artacaktır.

- İkincil belleklere erişim maliyeti çok fazla olduğu için bu çakışmaları daha karmaşık hash fonksiyonları kullanarak azaltılmalıdır.

Çakışma (Collesion-devam)

- Çakışmaları önlemek için kullanılan diğer bir yaklaşım ise **"Packing Factor"**'ün azaltılmasıdır.
- (Packing Factor=Packin Density=Load Factor)

$$\textbf{Packing Factor} = \frac{\textbf{Saklanılan Kayıt Sayısı (Number of Records Stored)}}{\textbf{Toplam Saklama Kapastesi (Total Number of Storage Locations)}}$$



Çakışmaların Çözülmesi (Collision Resolution)

Oluşan çakışmaların çözümlenmesinde aşağıdaki yaklaşımlar kullanılır :

Bağlantıları kullanarak çakışmaların çözümlenmesi
(Collesion Resolution with Links)

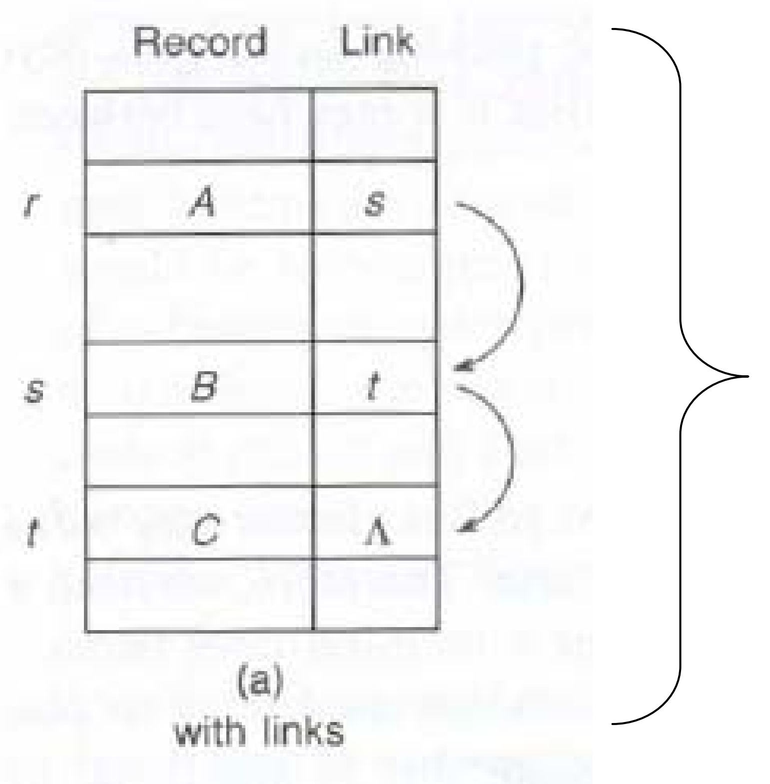
Bağlantıları kullanmadan çakışmaların çözümlenmesi
(Collesion Resolution without Links)

- Kayıtların statik olarak konumlandırılması (Static Positioning of records)
- Kayıtların dinamik olarak konumlandırılması (Dynamic Positioning of records)

Collesion Resolution with Links

Çakışmaya sebep olan kayıtlar bir link oluşturulur.

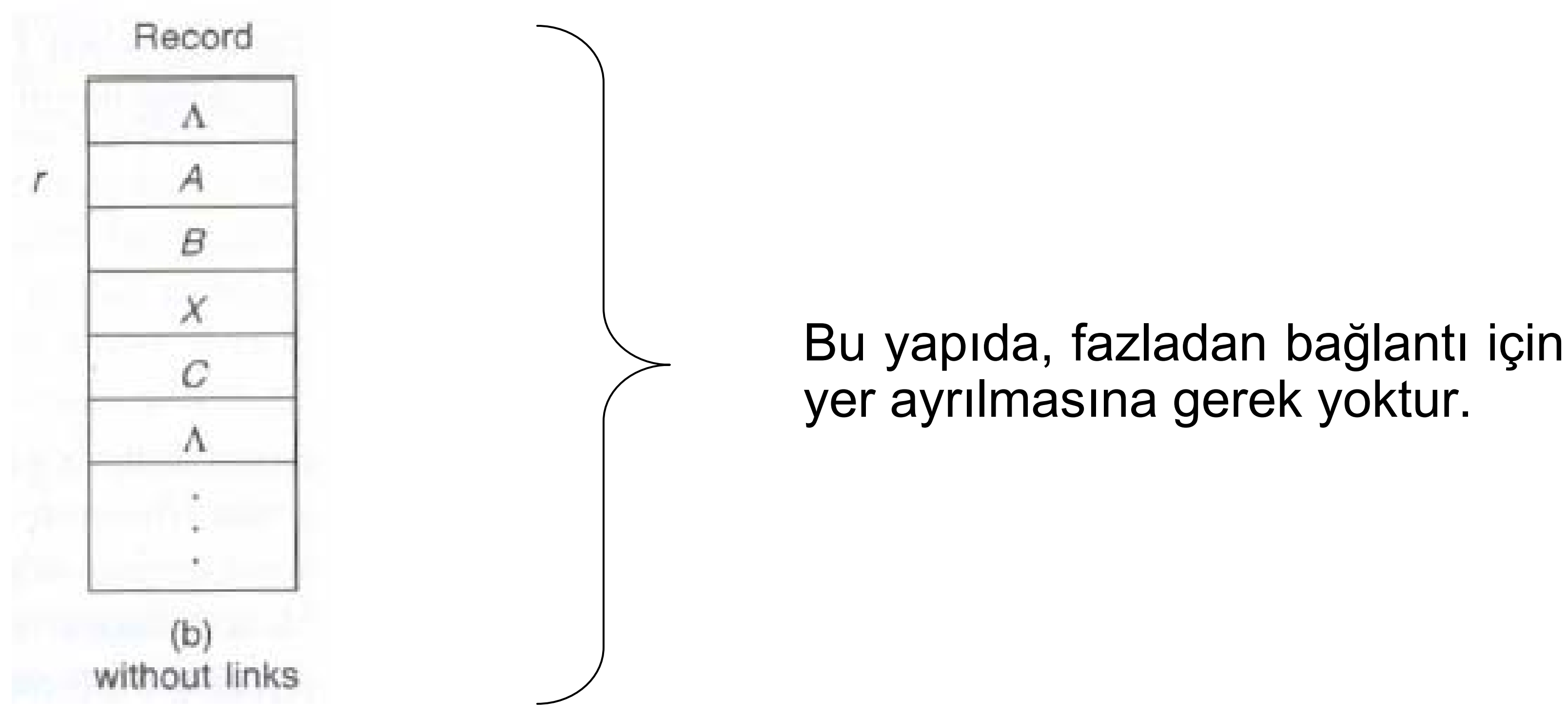
Örn:



Dezavantajı link bilgisi için yer gerektirir.

Collesion Resolution without Links

- Çakışmaya sebep olan kayıtlar ait oldukları adresten sonraki ilk boş yere yazılır.



Bu yapıda, fazladan bağlantı için yer ayrılmasına gerek yoktur.

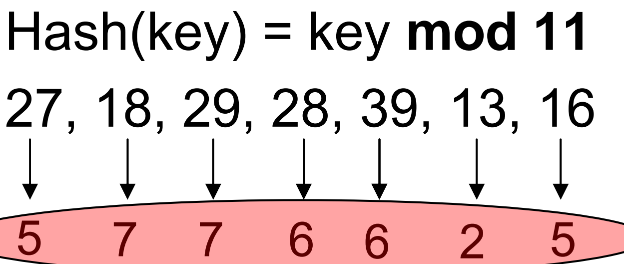
Coalesced Hashing (LISCH)

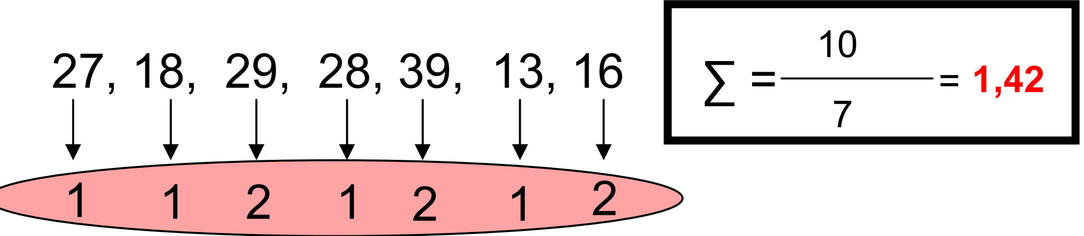
- Bu çözümleme tekniği, çakışan adreslerin birbirine bağlanmasında pointerları kullanır.
- Aranılan herhangi bir kaydın bulunmasında, zincir üzerinde ilgili kayda kadar arama gerçekleştirilir.
- Farklı home adreslerine sahip olan halkalar birlikte büyürler ve küçülürler.
- Zincir büyüdükçe aranılan kaydı getirmek için gerekli olan probe sayısı da artacaktır. Bu da performansı önemli ölçüde azaltacaktır.

Bu teknikte zincere olan eklemeler en alttaki en yüksek adresten itibaren yapılır. Boş olan yeri bulmak için, gösterici kademeli olarak 1’er 1’er azaltılır. (Boş bir yer bulununcaya kadar).

Örnek (LISCH)

Hash(key) = key **mod 11**





Her bir kayda erişmek için gerekli olan probe sayısı

	Konumlar	^(Link)
0		
1		
2	13	
3		
4		
5	27	8
6	28	9
7	18	10
8	16	
9	39	
10	29	

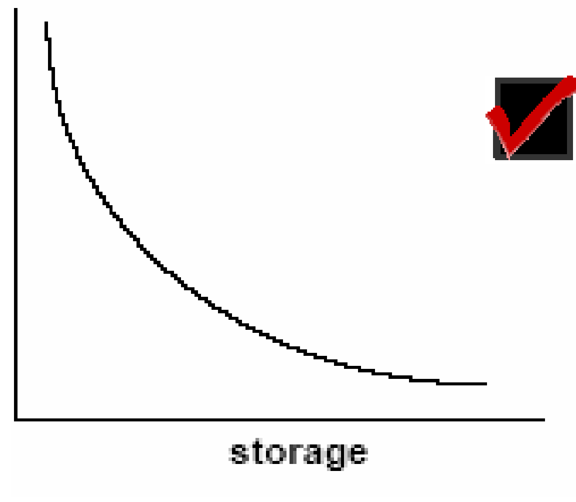
LISCH

Örnek (devam)

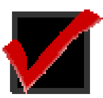
- Göstericinin 0’dan az bir değer göstermesi tablonun dolu olduğunu ifade eder.

Packing Factor = 7/11=0,63 = %63

Packing Faktor



Packing Faktör azaldıkça çakışmaların sayısı da azalacaktır.



Arama işlemi, kaydın home adresinden, tablo içerisinde ^ ile başlayan karaktere kadar gerçekleştirilir. Bu aramada ulaşılan kayıt ekrana getirilir. Fakat başarısız aramalar sırasında da tablo içerisinde baştan sonra bakmak gerekeceğinden zaman kaybı söz konusu olacaktır.

Eğer ki kayıtlara ulaşılmadaki sıklık derecesi biliniyorsa, bu kayıtlar home adresten sonra hemen yerleştirilirse performans kaybı bir ölçüde de giderilebilir.

Bu yapıda, çoklu halkalardan dolayı silme işlemi direkt olarak gerçekleştirilemez. Aksi taktirde kayıtlar arasındaki bağlantı kaybolur. Bunun için halkadaki en sonraki kayıt silinen kaydın yerine taşınır. En sondaki kayıt ile silinen kaydın aynı home adresine sahip olması gerekir.

Silinen kaydın yerine taşınan kaydın zinciri bozmayacak şekilde yeniden bağlantısı yapılır. Böylelikle zincir korunmuş olur.

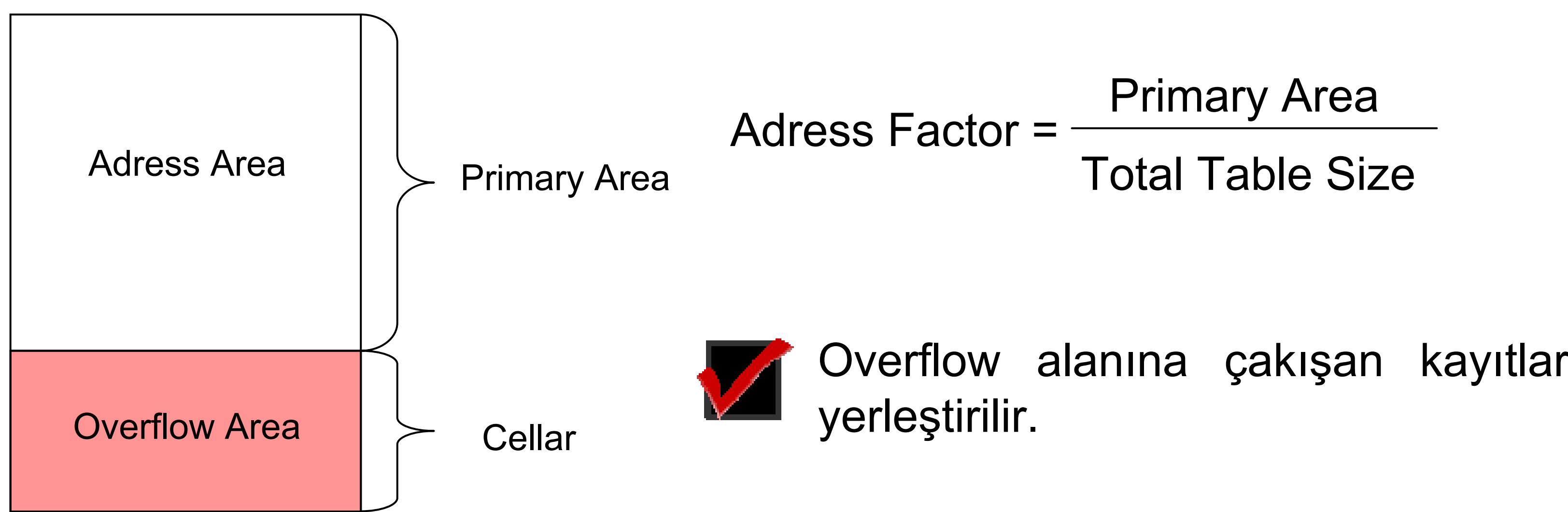
Zincirin Verimliliğini Arttırmak

Varolan yaklaşımları 3 kategori altında toplamak mümkündür.

i) Tabloların yeniden organizasyonu
ii) Çakışan halkaların birbirine bağlanması
iii) Boş olan konumların seçimi

Late Insertion Coalesced Hashing-LICH

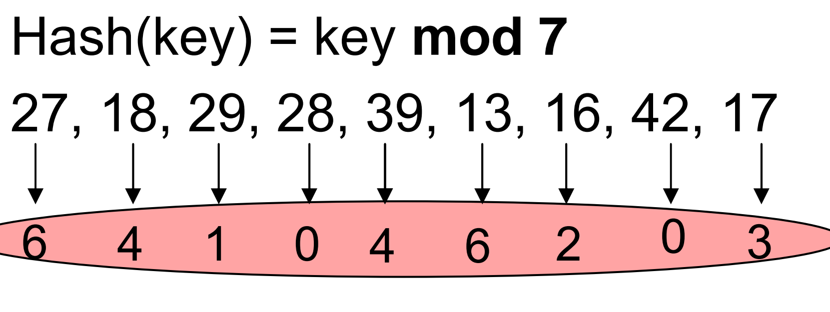
LISCH yapısındaki problemler tabloların yeniden organizasyonu sayesinde azaltılabilir. Tablo; adres ve taşma alanı olmak üzere 2 parçaya bölünebilir.



$$\text{Adress Factor} = \frac{\text{Primary Area}}{\text{Total Table Size}}$$

- Overflow alanına çakışan kayıtlar yerleştirilir.

Örnek-LICH



Konumlar

27, 18, 29, 28, 39, 13, 16, 42, 17

1 1 1 1 2 2 1 2 1

Gerekli Olan Probe Sayısı

		^
0	28	8
1	29	
2	16	
3	17	
4	18	10
5		
6	27	9
7		
8	42	
9	13	
10	39	

Overflow Area (7–10)

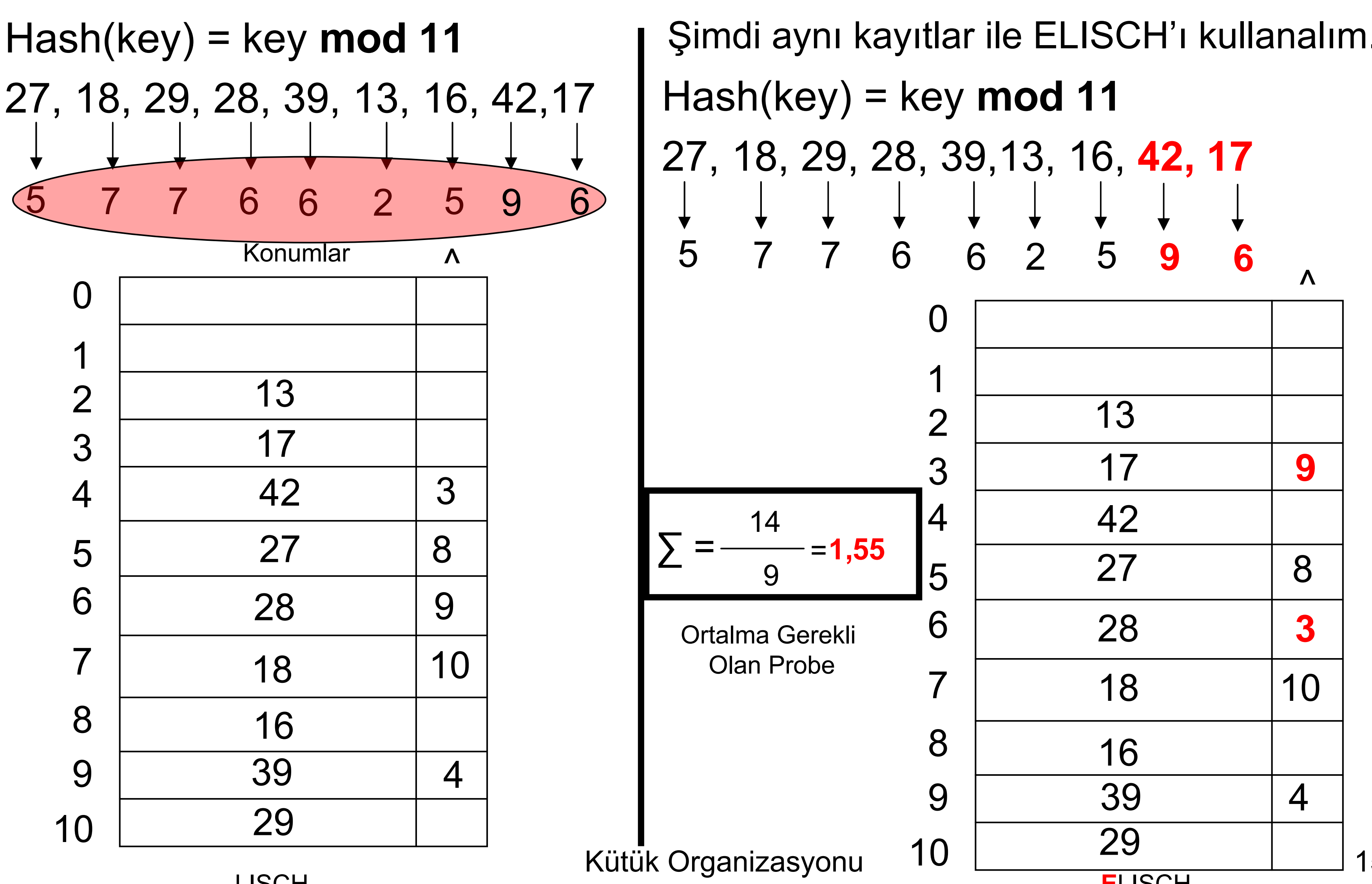
$\sum = \frac{12}{9} =$ **1,33**

Packing Factor=9 /11 = **%81**

Early Insertion Standart Coalesced Hashing-EISCH

Bu teknik yeni kaydı home adresten hemen sonra ekler. Home adresin bağlantı tutanağı yeni eklenen adresi gösterir.

Örnek-EISCH



LISCH

		^
0		
1		
2	13	
3	17	
4	42	3
5	27	8
6	28	9
7	18	10
8	16	
9	39	4
10	29	

ELISCH

		^
0		
1		
2	13	
3	17	**9**
4	42	
5	27	8
6	28	**3**
7	18	10
8	16	
9	39	4
10	29	

Farklı Alternatifler

Boş alanların seçilme tarzını değiştirmek ise farklı bir yaklaşımdır. Şimdiye kadar boş alanlar saklama biriminin en alt noktasından itibaren seçilmekteydi. Tabloda çakışan kayıtları tek bir bölgeye toplamak (overflow area) çakışmaların daha da artmasına neden olur. Bunun için Hsiao vd. boş alanların rastgele olarak seçilmesini önermişlerdir. Böylece çakışan kayıtlar daha tarafsız olarak tabloya dağıtılmış olacaktır.

Boş alanların seçiminde bir diğer yaklaşım ise, boş alanların saklama tablosunun bir altından bir üstünden sırayla çift yönlü olarak seçilmesidir(Bidirectional).

Progressive Overflow

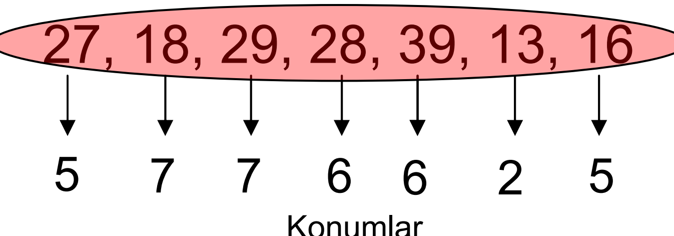
Coalesced yönteminin temel dezavantajı, bir sonraki kaydın yerini göstermesi için fazladan pointer bilgisi barındırmasıdır. Bunu için “Progressive Overflow” yöntemi geliştirilmiştir.

Bu teknikte çakışan kayıtların yerleştirilmesi için, çakışan adresten itibaren bir sonraki konumun boş olup olmadığına bakılır. Eğer boşsa çakışan kayıt buraya konumlandırılır. Değilse bir sonraki boş yer aranır.

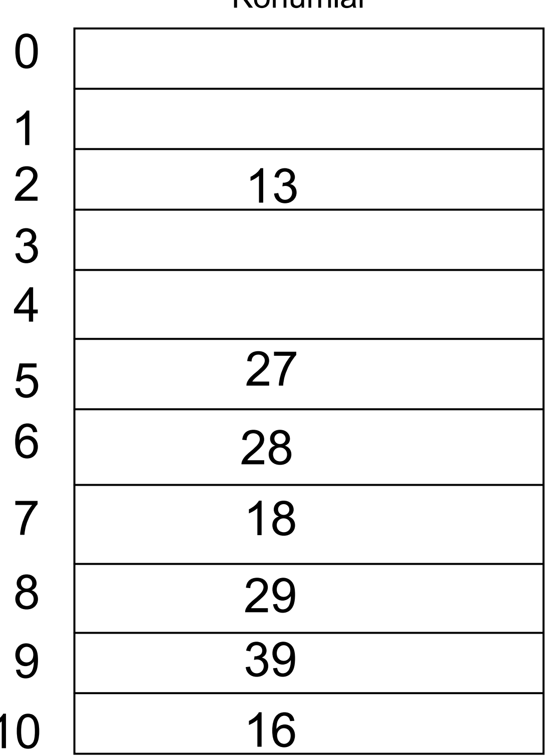
Tablo, dairesel bir bütün gibi düşünülür. Yine, başarısız kayıt aramalarında performans önemli ölçüde azalır. Bu yüzden bu algoritma fazla etkili değildir.

Örnek-Progressive Overflow

Hash(key) = key **mod 11**



Konumlar



0	
1	
2	13
3	
4	
5	27
6	28
7	18
8	29
9	39
10	16

Progressive Overflow

27, 18, 29, 28, 39, 13, 16

1 1 2 1 4 1 6

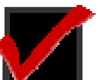
$$\sum = \frac{16}{7} = \mathbf{2,28}$$

Ortalma Gerekli Olan Probe

Görüldüğü ortalma olarak bir kayıt için gerekli olan probe sayısı oldukça yüksektir.

Örnek(devam)

0	
1	
2	13
3	
4	
5	27
6	28
7	18
8	29
9	39
10	16



16 anahtarına ulaşmak için 6 probe gerekmektedir. Bunun nedeni var olan ikincil kümelerdir. Diğer bir deyişle 16'ya erişebilmek için arada gereksiz olan 4 kayıda daha erişmek gerekmektedir. Bu ikincil kümler de farklı home adresine sahip olan kayıtlardan kaynaklanmaktadır.

İkincil kümelerin oluşmasında en büyük faktör artım sayısının "1" olmasıdır. Bunu ortadan kaldırmak için değişken artımlı teknik kullanılmalıdır(Linear Quatient).

Progressive Overflow’daki silme işleminde dikkatli olunması gerekmektedir. Bir kaydı direkt olarak silmek mümkün değildir. Çünkü böyle bir durumda arama sırasında boşluklar meydana gelecektir. Bu boşluklar meydana geldiğinde arama işlemi durur. Böyle bir durumda ise aranan kayıt tablo içerisinde olsa bile ulaşılamayabilir.

Bunun üstesinden gelebilmek için silinecek olan kaydın başlangıcına bir işaretçi yerleştirilir (Tombstone).

Bu işaretçi arama işleminin durmak sızın devam etmesini sağlar.

Bu noktaya yeni bir kayıt ekleneceğinde de bu tombstone kaldırılır ve bu noktaya yeni bir kayıt eklenir.

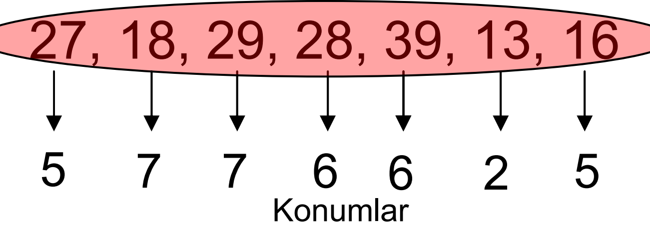
Use of Buckets

Şimdiye kadarki olan tekniklerde, saklama ünitelerinde tek bir kayıt saklanmaktaydı. İkincil ünitelerde bir adrese birden fazla kayıt saklayarak, bu ünitelere olarak erişim sayısını azaltabiliriz. Bunu kullanan teknik bucket'tır (Bloklama).

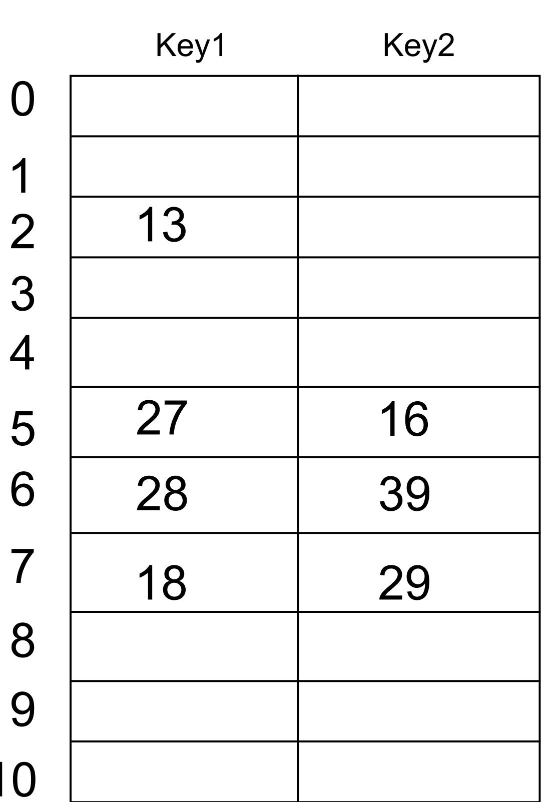
Bir blok içerisinde saklanılan kayıt sayısına blok faktörü denir. Blok faktörü arttıkça, ikincil ünitelere yapılacak erişimin sayısı da azalacaktır. Bunun anlamaı çakışan kayıtların aynı adrese yazılmasıdır.

Örnek

Hash(key) = key **mod 11**

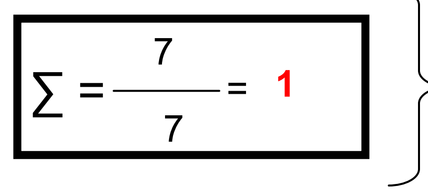


ve blok faktörü 2 olması durumunda



	Key1	Key2
0		
1		
2	13	
3		
4		
5	27	16
6	28	39
7	18	29
8		
9		
10		

Use of Buckets



$$\sum = \frac{7}{7} = 1$$

Herbir kayda erişmek için gerekli olan probe sayısı

Bloklama işlemi sırasında, her bir bloktaki kayıtların birbirinden ayırt edilmesi gerekmektedir. Bunun için fixed length record ya da kayıtların bir sınırlayıcı ile birbirinden ayrılması gerekir.

Linear Quotient (Doğrusal Oran)

- Linear Quotient metodu Progressive Overflow metoduna oldukça benzemektedir. Temel fark, linear quotient metodunda değişken artım kullanmaktır. Bunun amacı oluşan ikincil kümeleri azaltmaktır. İkincil kümeler azaltığı taktirde her bir kayda erişmek için gerekli olan probe sayısı da azalacaktır.

- Hatırlanacağı gibi ikincil kümeler, home adrese bağlı olarak artım değerinin sabit olduğu durumlarda ortaya çıkmaktadır.

Linear Quotient (devam)

- Bu metodda artım, eklenen anahtar değerine bağlı olan fonksiyonun sonucudur. Diğer bir değişle bu metodda 2 farklı hash'leme gerçekleştirilir.
  - Birincisi kaydın yerleştirileceği adresin tespiti için yapılan hash'leme.
  - Artım miktarının tespiti için yapılan hash'lemedir.
- Bu yüzden bu yöntem “double hashing”(çifli hash'leme) olarak da yorumlanabilir. Yani:

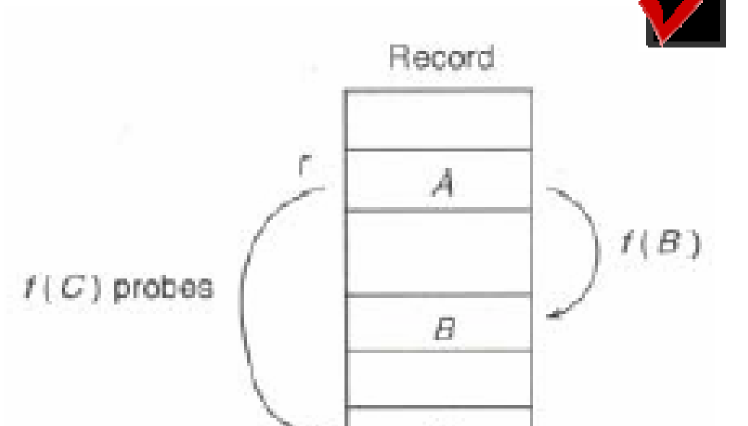
h1 ve h2

Adres için — Artım

$$H_2 = \text{Quotient}(Key / P) \bmod P$$

$$H_2' = (Key \bmod (P-2)) + 1$$

P tablo boyutunu göstermektedir.

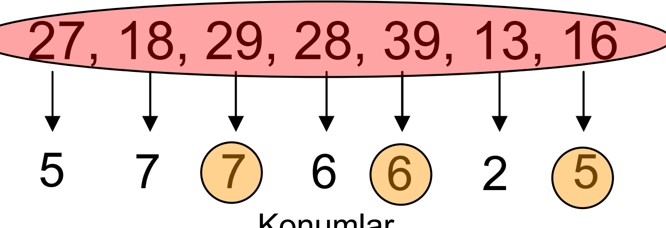


- Yandaki şekilde A,B ve C kayıtları çakışan kayıtlardır ve aynı home adreslerini paylaşmaktadırlar. Fakat B ve C'nin konumları farklıdır. Çünkü her komut için gerekli olan artım miktarı f() fonksiyonu tarafından belirlenmektedir.

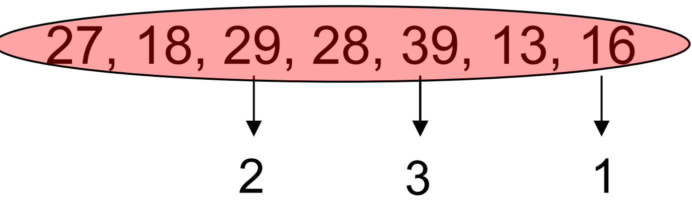
- Burada dikkat edilmesi gereken en önemli nokta P'nin asal sayı olmasıdır.

Örnek (Linear Quotient)

Hash(key) = key **mod 11**



Artım Miktarı



0	
1	39
2	13
3	
4	
5	27
6	28
7	18
8	16
9	29
10	

$$\sum = \frac{13}{7} = \mathbf{1,85}$$

Aynı işlem için Progressive Overflow'da 2,28 gerekmekteydi.

Artım oranını değişken olarak ayarlamamız Progressive Overflow'da karşımıza çıkan ikincil kümeleri kırmamıza ve böylece de performansta artışa neden olmaktadır.

Örnek (devam)

0	
1	39
2	13
3	**67**
4	
5	27
6	28
7	18
8	16
9	29
10	

Linear Quotient

- Yandaki tabloya 67 anahtarını 3 numaralı konuma yerleşir ve bu durumda bu anahtara ulaşmak için 5 probe'a gereksinim duyulur.
- Eğer 1 numaralı konumda 39 anahtarı olmasaydı buraya 67 direkt olarak yerleştirilebilecekti.
- Ele alacağımız bir sonraki yöntem yukarıda bahsedilen işlemi yapmadan önce genel olarak bütün kayıtları bulmak için gerekli probe sayısını azaltıp azaltamayacağına bakar. Eğer ki azaltabiliyorsa bu işlemi gerçekleştirir.

Silme işleminde de yine tombstone kullanmak gerekir.

Arama işlemi yine Progressive Overflow'da olduğu gibi boş bir adresle karşılaşınca sonlandırılmaktadır.

İkincil kümeleri azaltmeda Linear Quotient kendi başına tek bir yöntem olmayıp bir çok yöntem vardır.