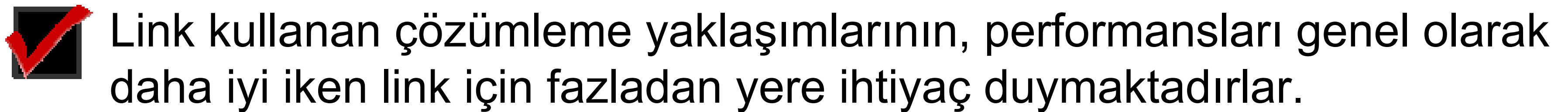
Hesaplanabilen Zincirler (Computed Chaining)

Bu kısma kadar, meydana gelen çakışmaları genel olarak 2 farklı yaklaşımla çözdük. Bunlar:

Link alanı kullanan çözümleme yaklaşımları (Colaesced hashing)
Link alanı kullanmayan çözümleme (Progressive Overflow, Linear Quotient, Brient's Method ve Binary Tree) yaklaşımlarıdır.



Link kullanan çözümleme yaklaşımlarının, performansları genel olarak daha iyi iken link için fazladan yere ihtiyaç duymaktadırlar.

Tam ters durumda ise link kullanmayan yaklaşımlarda görülmektedir. (Daha az yere ihtiyaç duymalarına rağmen performansları düşüktür.)

- Her iki yaklaşımı da tek bir çatı altında toplayarak hem performans artışı hem de yerden tasarruf sağlanabilir.

- Geliştirilecek olan bu yeni yaklaşım "Computed Chaining" dir. Bu metodda, gerçek link adresleri kayıt edilmeyip hesaplanarak bulunmaktadır.

- Az sayıda bit kullanarak gerçek adresleri elde etmek mümkündür.

- Link alanlarında, zincir içindeki elemanların tam adresleri yerine offset değerleri saklanılır. Bu offset değeri daha sonra hesaplamalarda kullanılarak zincir içerisindeki bir sonraki kayda ulaşmak mümkündür.

Computed Chaining

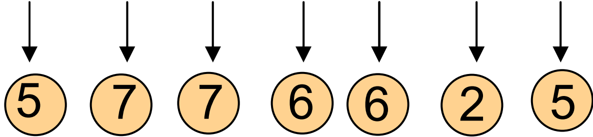
	Record	Pseudo-link
r	A	i(A)
s	B	i(B)
t	C	Λ

i(A) bir sonraki konuma ulaşmak için kullanılacak olan artım fonksiyonudur.

Computed Chaining (Örnek)

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

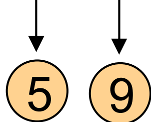
27, 18, 29, 28, 39, 13, 16



Kullandığımız bu anahtarlara ek olarak 2 anahtar değeri daha olsun. Bunlar :

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

38, 53

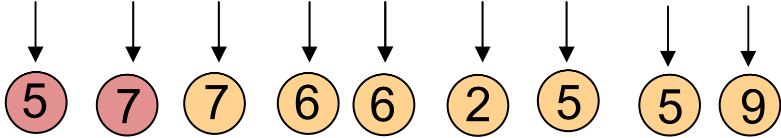


Artım fonksiyonu ise:

i(key)=Quotient(Key/11) mod11

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

27, 18, 29, 28, 39, 13, 16, 38, 53



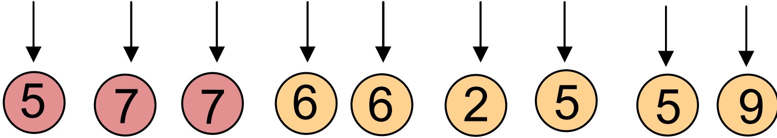
27 ve 18 herhangi bir çakışma olmadan doğrudan 5. ve 7. konumlara yerleştirilebilir.

	Konumlar	nof
0		
1		
2		
3		
4		
5	27	
6		
7	18	
8		
9		
10		

Computed Chaining

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

27, 18, 29, 28, 39, 13, 16, 38, 53



- 29 kaydı 7 numaralı konuma yerleşmek istiyor fakat bu noktada daha önceki 18 anahtarı ile çakışma meydana gelmektedir.

	Konumlar	nof
0		
1		
2		
3		
4		
5	27	
6		
7	18	1
8	29	
9		
10		

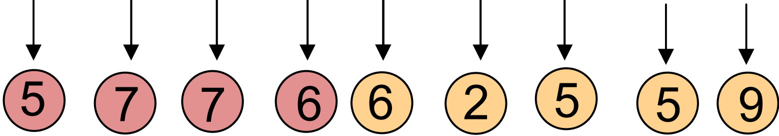
Computed Chaining

Bu anahtar için uygun konumun bulunmasında bazı işlemlerin yapılması gerekmektedir. Bunlar:

- Öncelikle eklenecek olan konumdaki anahtar kendi home adresine mi yerleşmiş? (18 kendi home adresinde olduğundan problem yok)
- Bu konumdaki anahtarın ardından gelen başka bir anahtar var mı? Bunu link alanına bakarak anlayabiliriz.(18 için null)
- Eklenecek olan konumdaki anahtarın artım değerini kullanarak yeni konum tayin edilir. (Artım değeri 18 için 1 ‘dir).

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

27, 18, 29, 28, 39, 13, 16, 38, 53

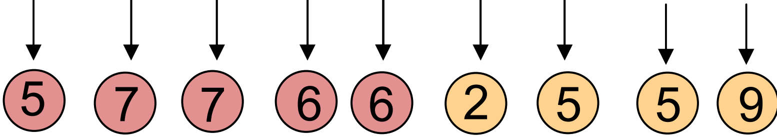


28 kaydı 6 numaralı konuma herhangi bir çakışma olmadan yerleşebilir.

	Konumlar	nof
0		
1		
2		
3		
4		
5	27	
6	28	
7	18	1
8	29	
9		
10		

Computed Chaining

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



- 39 kaydı da 6 numaralı konuma yerleşmek isteyeceğinden bir çakışma olacaktır.

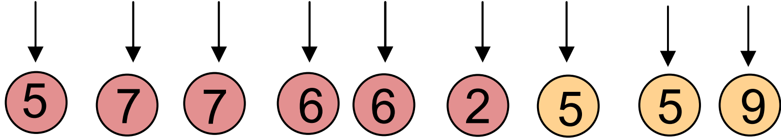
	Konumlar	nof
0		
1		
2		
3		
4		
5	27	
6	28	2
7	18	1
8	29	
9		
10	39	

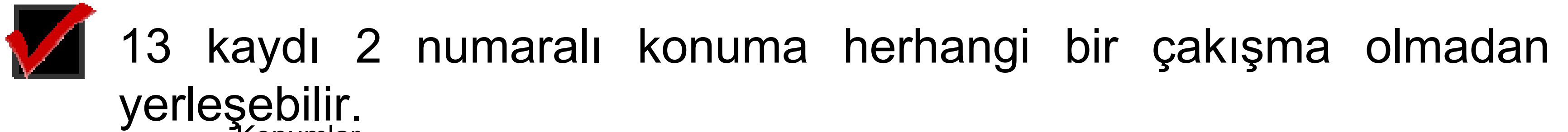
Computed Chaining

- Bu konumda önceden olan 28 anahtarı kendi home adresindedir ve ardından gelecek olan bir kayıt şu an için yoktur. (offset alanı boş)
- 39 anahtarının yerleşeceği bir sonraki konum için 28'in artım değeri hesaplanır. (28'in artım değeri 2)
- Artım değeri dikkate alınarak 39 için uygun olan konum, 10 numaralı konumdur.
- 39 anahtarı için gerekli olan probe sayısı 3 değil 2 dir.

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

27, 18, 29, 28, 39, 13, 16, 38, 53





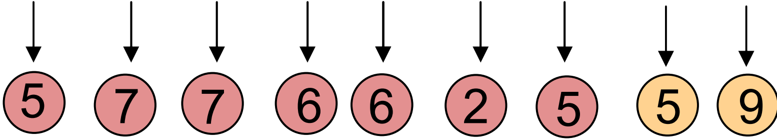
13 kaydı 2 numaralı konuma herhangi bir çakışma olmadan yerleşebilir.

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

Computed Chaining

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

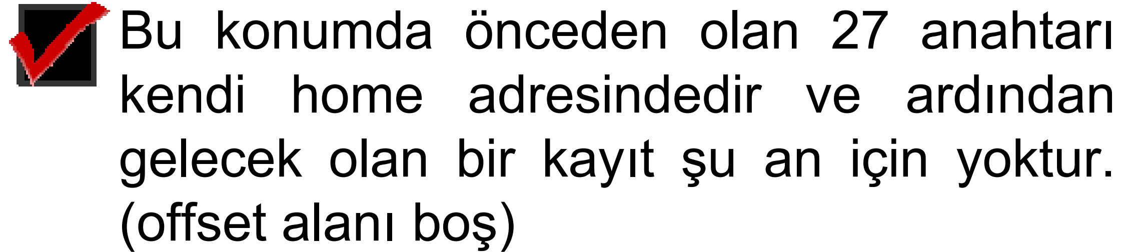
27, 18, 29, 28, 39, 13, 16, 38, 53



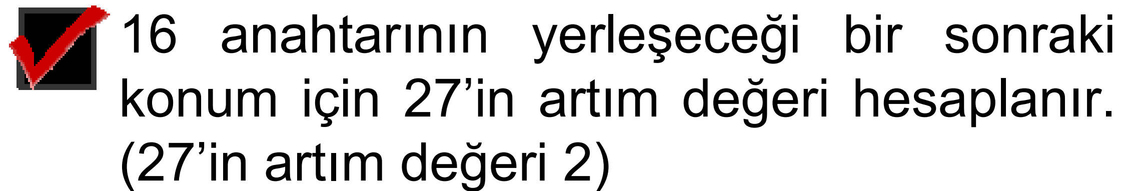
16 kaydı 5 numaralı konuma yerleşmek isteyeceğinden bir çakışma olacaktır.

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

Computed Chaining



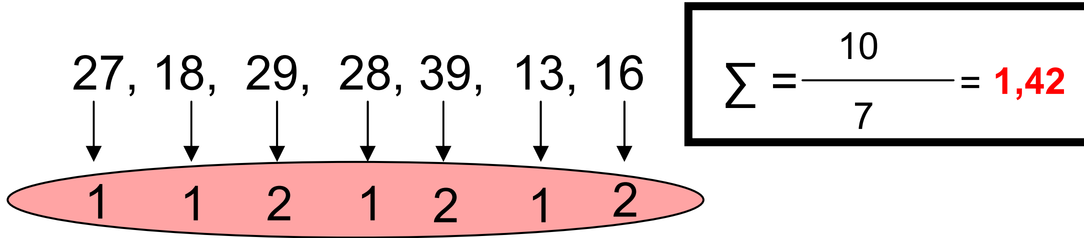
Bu konumda önceden olan 27 anahtarı kendi home adresindedir ve ardından gelecek olan bir kayıt şu an için yoktur. (offset alanı boş)



16 anahtarının yerleşeceği bir sonraki konum için 27'in artım değeri hesaplanır. (27'in artım değeri 2)

Artım değeri dikkate alınarak 16 için uygun olan konum, 9 numaralı konumdur.

Örneğe ekstra olarak eklenmiş olan 38 ve 53 anahtarlarına geçmeden önce genel olarak bu yaklaşımda her bir kayıda erişmek için gerekli olan probe sayısı hesaplayalım.

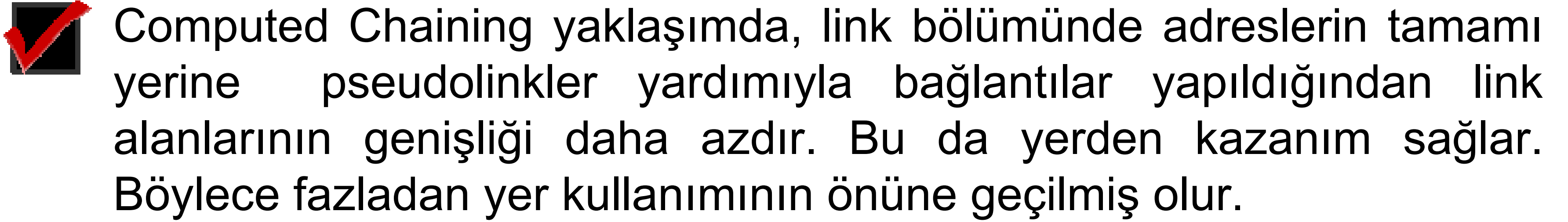


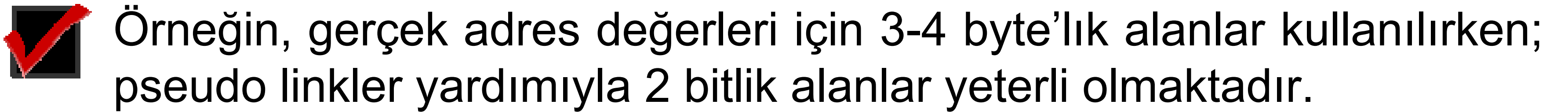
$$\sum = \frac{10}{7} = 1{,}42$$

Aynı işlem ;
Binary tree (Link alanları kullanılmıyor) ile gerekli olan probe sayısı =1,7'dır.
LISCH (Link alanları kullanılmakta) ile gerekli olan probe sayısı =1,4'dır.

Buradan da görüleceği gibi LISCH ile Computed Chaining yaklaşımları kayıtları okurken aynı probe sayısına ihtiyaç duymaktadırlar. (1,4)

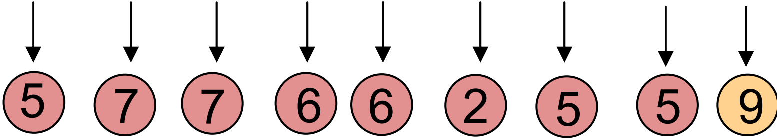
Peki, o zaman Computed Chaining Yöteminin avantajı nedir?


Computed Chaining yaklaşımda, link bölümünde adreslerin tamamı yerine pseudolinkler yardımıyla bağlantılar yapıldığından link alanlarının genişliği daha azdır. Bu da yerden kazanım sağlar. Böylece fazladan yer kullanımının önüne geçilmiş olur.


Örneğin, gerçek adres değerleri için 3-4 byte'lık alanlar kullanılırken; pseudo linkler yardımıyla 2 bitlik alanlar yeterli olmaktadır.

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

27, 18, 29, 28, 39, 13, 16, 38, 53



- 38 kaydı 5 numaralı konuma yerleşmek isteyeceğinden bir çakışma olacaktır.

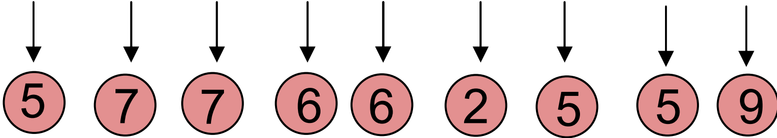
	Konumlar	nof
0	38	
1		
2	13	
3		
4		
5	27	2
6	28	2
7	18	1
8	29	
9	16	2
10	39	

Computed Chaining

- Bu konumda önceden olan 27 anahtarı kendi home adresindedir ve ardından gelecek olan bulunmaktadır. (Offset değeri 2 olduğundan ardından gelen kayıt 9 numaralı konumdadır)
- Bu konumda ise 16 bulunmaktadır ve ardından herhangi bir anahtar değeri gelmemektedir. (Offset alanı null)
- 16 anahtarının artım değeri olan 1 kullanılarak 38 kaydı için uygun konum belirlenir. (0. konum boştur ve buraya 2 artım değeri sonucunda varılabilir.)

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

27, 18, 29, 28, 39, 13, 16, 38, 53



- 53 kaydı 9 numaralı konuma yerleşmek isteyeceğinden bir çakışma olacaktır. Fakat burada karşımıza farklı bir durum çıkmaktadır.

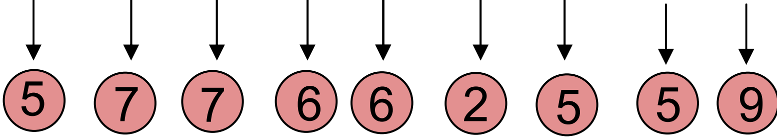
	Konumlar	nof
0		
1		
2	13	
3		
4		
5	27	2
6	28	2
7	18	1
8	29	
9	53	
10	39	

Computed Chaining

- 53 anahtarının konumu olan 9 numaralı konumda bulunan 16 anahtarının home adresi farklıdır.
- Bu durumda 16 anahtarı ve onun ardından gelen diğer anahtarların ötelenmesi gerekecektir.
- Bu işlem için öncelikle, 16 ve ardındaki anahtar değerleri sonra tekrar eklenmek üzere tablodan geçici olarak ayrılır.
- Böylelikle boşalan konuma 53 kaydı kolaylıkla yerleşebilir.

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

27, 18, 29, 28, 39, 13, 16, 38, 53



- Tablodan çıkarılan 16 ve 38 anahtarları tabloya yeniden eklenir.

	Konumlar	nof
0	16	1
1	38	
2	13	
3		
4		
5	27	3
6	28	2
7	18	1
8	29	
9	53	
10	39	

Computed Chaining

- Bunun için 16'nın home adresi olan 27'nin artım değeri tekrar kullanılarak ilk boş konum araştırılır. İlk boş konum 0. konumdur ve 3 artım değerinden sonra bu konuma varılmıştır.
- 16'dan sonra 38 anahtar değeri yerleştirilecektir ve bu sefer 16'nın artım değeri olan 1 kullanılarak boş konumlar araştırılır ve 1. konuma yerleştirilir.

Retrieval işleminde 16. Anahtarı ötelemekle performans kaybı yaşanmış olabilir mi ?

Herhangi bir performans kaybı sözkonusu değildir. Çünkü, 16. anahtarı ötelemeden önce retrieval için 2 probe söz konusu iken öteledikten sonra da aynı sayıda probe söz konusudur. Dolayısıyla herhangi bir performans kaybı yoktur. (Retrieval açısından)

	Konumlar	nof
0	38	
1		
2	13	
3		
4		
5	27	2
6	28	2
7	18	1
8	29	
9	16	2
10	39	

Computed Chaining

Ötelemeden Önce

	Konumlar	nof
0	16	1
1	38	
2	13	
3		
4		
5	27	3
6	28	2
7	18	1
8	29	
9	53	
10	39	

Computed Chaining

Ötelendikten Sonra

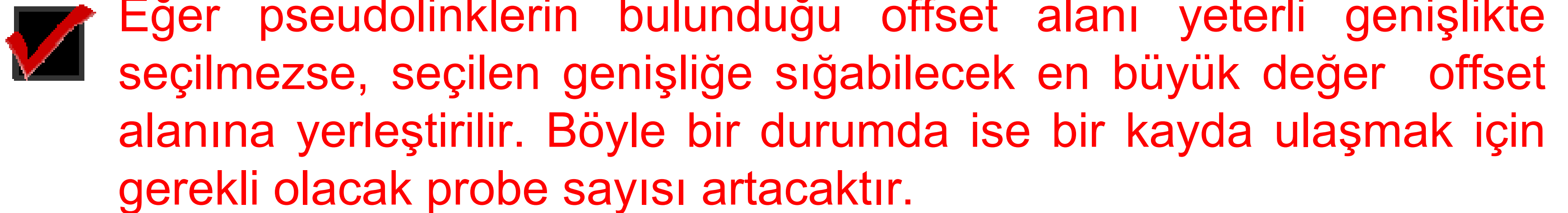
Avantajları

- Computed chaining yöntemi link kullanmayan yakşaımlara göre daha iyi performansa sahiptir.
- Bir kayda ulaşmak için gerekli olan probe sayısı, halkadaki sırasına bağımlıdır.
- Kayda ulaşmadaki performansı arttırmak için, anahtar değerlerini eşsiz olarak dağatan hash fonksiyonları tercih edilmelidir.
- Bir anahtar değeri silindiğinde, silinen elemandan sonraki tüm kayıtların yeniden yerleştirilmesi gerekmektedir.
- Bu yaklaşım kayıt ekleme ve silme işlemleri sırasında daha fazla hesaplamaya gereksinim duyduğundan, bu işlemler sırasında gereksinim duyulan zaman daha fazladır. Diğer taraftan ise okuma işlemini daha kısa sürede gerçekleştirirler.

Dezavantajları

Daha önceki link kullanan çakışma çözme yöntemlerinde link alanında adres bilgisi tutulduğundan link alanın adres bilgisini barındıracak genişlikte olması gerekiyordu.

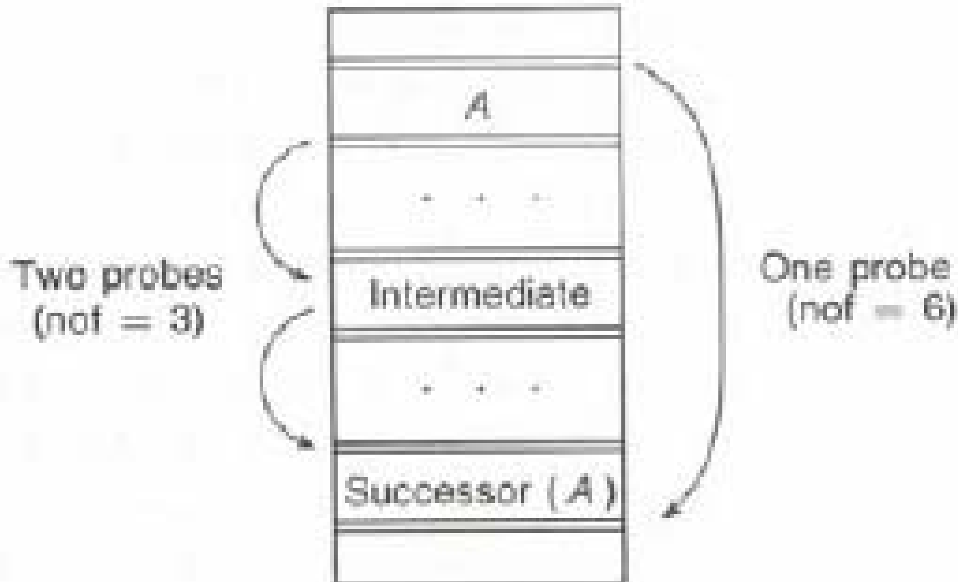
Computed chaining yönteminde ise; pseudo link alanı offset adresin en büyük değerini adresleyebilecek genişlikte seçilmelidir.



Eğer pseudolinklerin bulunduğu offset alanı yeterli genişlikte seçilmezse, seçilen genişliğe sığabilecek en büyük değer offset alanına yerleştirilir. Böyle bir durumda ise bir kayda ulaşmak için gerekli olacak probe sayısı artacaktır.

Sonuç olarak; offset alanı için yeterli miktarda bit ayrılmalıdır.

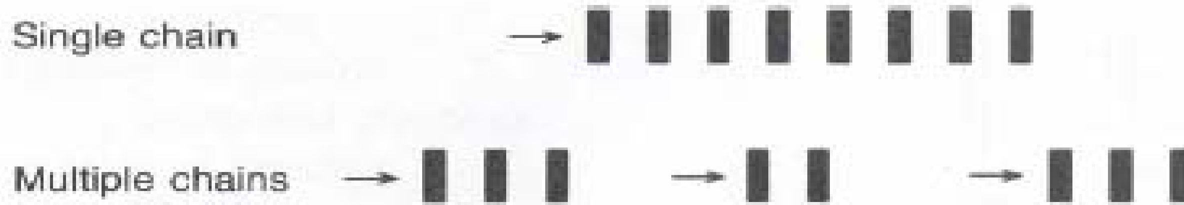
Örn: A kaydı için pseuodolink değeri 6 olsun ve fakat offset alanı için 2 bitlik bir alan ayrılsın. Bu durumda2 bit ile ifade edilebilecek en büyük sayı $(11)_2$ olup A kaydının ardından gelen herhangi bir kayda erişmek için 1 probe yeterli olabilecek iken artık 2 probe'un kullanılması gerekecektir.



Farklılık (Variant)

- Computed Chaining yaklaşımının performansını arttırmak için bazı bir takım düzenlemeler yapılabilir.
- Bir kaydı aradığımızda, bu işlemi olabildiğince kısa sürede ve etkili bir şekilde gerçekleştirmek isteriz. Aynı durum günlük hayatımızda bir objeyi ararken de aynı şekilde gerçekleşir.
- Örn: Bir nesneyi ev içerisinde aramak istediğinizi düşünün. Eğer bu nesnenin evin hangi odasında olduğunu biliyorsanız, nesneyi direkt o oda içerisinde arayabilirsiniz. Diğer durumda ise arama işlemini evin tüm köşesinde yapmak zorunda kalabilirsiniz.
- Böyle bir yaklaşımdan genel olarak BÖL ve YÖNET olarak söz etmek mümkündür.
- Eğer kayıtlar daha küçük gruplara ayrılırsa ve aradığımız kaydın hangi grup içersinde yer aldığını biliyorsak arama işlemi daha basit olacaktır.

- Bir probe chain birden fazla parçaya bölünür ve bir kayıt için sadece ilgili olduğu gruba bakılır.



- Standart hash fonksiyonu ve arttırma fonksiyonun yanısıra bir de kaydın hangi gruba dahil olacağını belirten g(key) hash fonksiyonu bulunmaktadır.

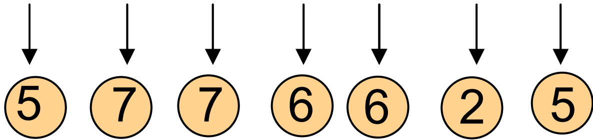
g(key) = 0, 1, . . . , R-1 olabilir. R alt grup sayısıdır

g(key) = key **mod** R

Multiple Chaining (Örnek)

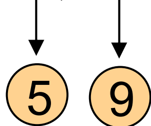
Hash(key) = key **mod 11**

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



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

38, 53



- Kullandığımız bu anahtarlara ek olarak 2 anahtar değeri daha olsun. Bunlar :
- Artım fonksiyonu ise:

  i(key)=Quotient(Key/11) mod11

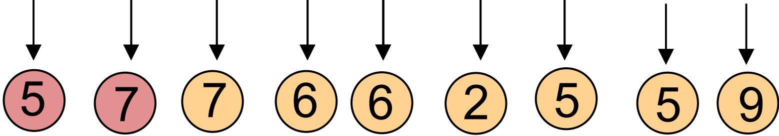
- Halka seçim fonksiyonu ise:

  g(key) = key mod 2

Bu durumda, eğer anahtar değeri çift ise 0. halkaya, tek ise 1. halkaya dahil olacaktır.

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

27, 18, 29, 28, 39, 13, 16, 38, 53



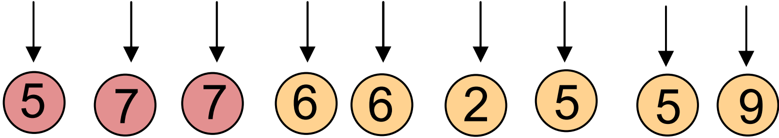
27 ve 18 herhangi bir çakışma olmadan doğrudan 5. ve 7. konumlara yerleştirilebilir.

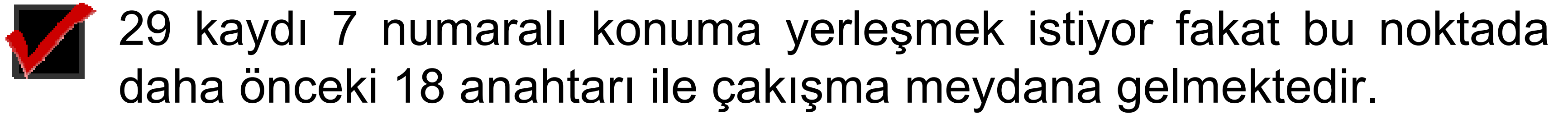
	Konumlar	0	1
0			
1			
2			
3			
4			
5	27		
6			
7	18		
8			
9			
10			

Computed Chaining

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

27, 18, 29, 28, 39, 13, 16, 38, 53



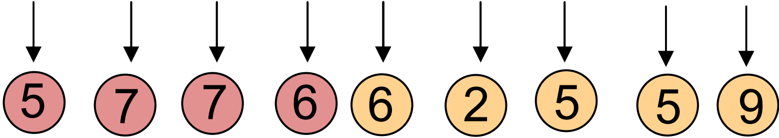


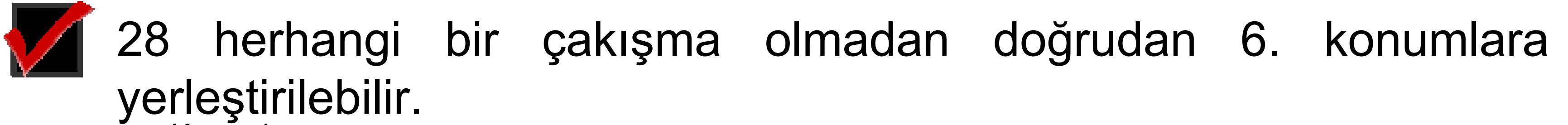
29 kaydı 7 numaralı konuma yerleşmek istiyor fakat bu noktada daha önceki 18 anahtarı ile çakışma meydana gelmektedir.

	Konumlar	0	1
0			
1			
2			
3			
4			
5	27		
6			
7	18		1
8	29		
9			
10			

Computed Chaining

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





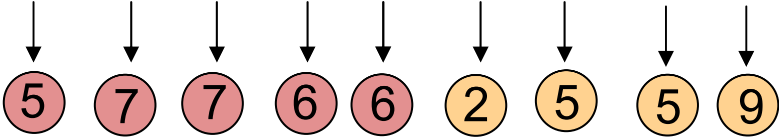
28 herhangi bir çakışma olmadan doğrudan 6. konumlara yerleştirilebilir.

	Konumlar	0	1
0			
1			
2			
3			
4			
5	27		
6	28		
7	18		1
8	29		
9			
10			

Computed Chaining

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

27, 18, 29, 28, 39, 13, 16, 38, 53



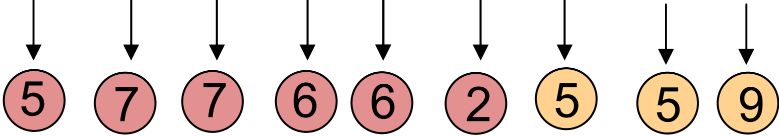
- 39 kaydı 6 numaralı konuma yerleşmek istiyor fakat bu noktada daha önceki 28 anahtarı ile çakışma meydana gelmektedir.

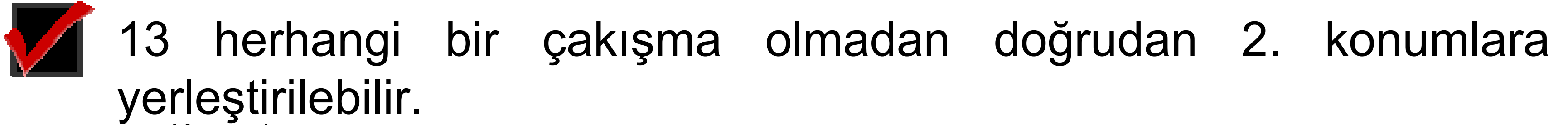
	Konumlar	0	1
0			
1			
2			
3			
4			
5	27		
6	28		2
7	18		1
8	29		
9			
10	39		

Computed Chaining

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

27, 18, 29, 28, 39, 13, 16, 38, 53





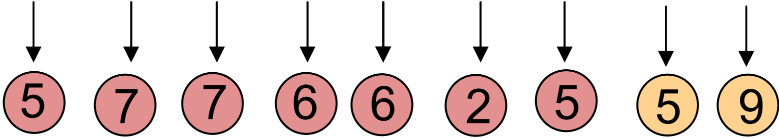
13 herhangi bir çakışma olmadan doğrudan 2. konumlara yerleştirilebilir.

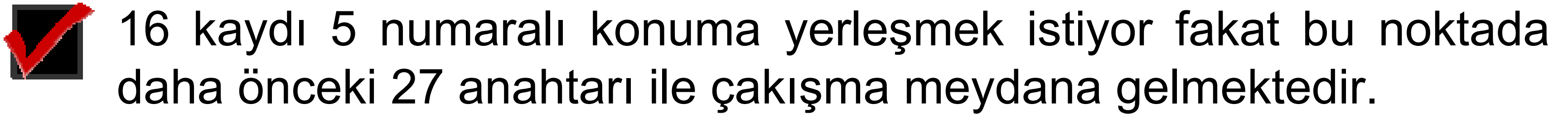
	Konumlar	0	1
0			
1			
2	13		
3			
4			
5	27		
6	28		2
7	18		1
8	29		
9			
10	39		

Computed Chaining

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

27, 18, 29, 28, 39, 13, 16, 38, 53





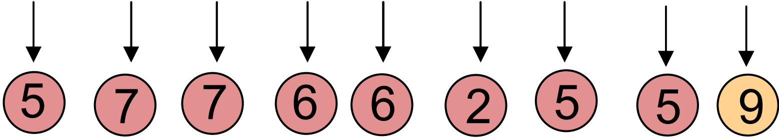
16 kaydı 5 numaralı konuma yerleşmek istiyor fakat bu noktada daha önceki 27 anahtarı ile çakışma meydana gelmektedir.

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

Computed Chaining

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

27, 18, 29, 28, 39, 13, 16, 38, 53



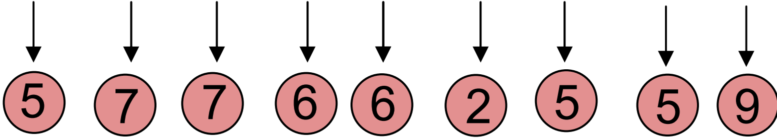
- 38 kaydı 5 numaralı konuma yerleşmek istiyor fakat bu noktada daha önceki 27 anahtarı ile çakışma meydana gelmektedir.

	Konumlar	0	1
0	38		
1			
2	13		
3			
4			
5	27	2	
6	28		2
7	18		1
8	29		
9	16	2	
10	39		

Computed Chaining

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

27, 18, 29, 28, 39, 13, 16, 38, 53



- 53 kaydı 9 numaralı konuma yerleşmek isteyeceğinden bir çakışma olacaktır. Fakat burada karşımıza farklı bir durum çıkmaktadır.

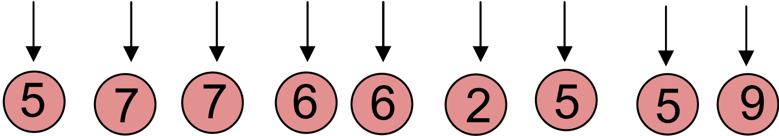
	Konumlar	0	1
0	38		
1			
2	13		
3			
4			
5	27	2	
6	28		2
7	18		1
8	29		
9	16	2	
10	39		

Computed Chaining

- 53 anahtarının konumu olan 9 numaralı konumda bulunan 16 anahtarının home adresi farklıdır.
- Bu durumda 16 anahtarı ve onun ardından gelen diğer anahtarların ötelenmesi gerekecektir.
- Bu işlem için öncelikle, 16 ve ardındaki anahtar değerleri sonra tekrar eklenmek üzere tablodan geçici olarak ayrılır.
- Böylelikle boşalan konuma 53 kaydı kolaylıkla yerleşebilir.

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

27, 18, 29, 28, 39, 13, 16, 38, 53

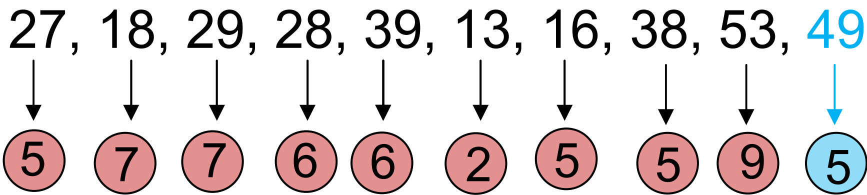


	Konumlar	0	1
0	16	1	
1	38		
2	13		
3			
4			
5	27	3	
6	28		2
7	18		1
8	29		
9	53		
10	39		

Computed Chaining

Burada kayıtların retrievalları için gerekli olan probe sayısında çok fazla bir değişim olmamıştır. Değişimin fark edilebilmesi için örneğimize 49 anahtarını ekleyelim.

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



	Konumlar	0	1
0	16	1	
1	38		
2	13		
3			
4	49		
5	27	3	5
6	28		2
7	18		1
8	29		
9	53		
10	39		

Computed Chaining

49 anahtarının elde edilebilmesi için sadece 2 proba gereksinim duyulur. Böyle bir durumda gerekli olan probe sayısı önemli ölçüde azaldığından önemli performans artışı elde edilir.

Çakışma Çözümleme Tekniklerinin Karşılaştırılması

TABLE 3.3 COMPARISON OF MEAN NUMBER OF PROBES FOR SUCCESSFUL LOOKUP (n = 997; α = packing factor)

α (percent)	DCWC (10-bit link)*	LISCH (10-bit link)	Progressive overflow†	Linear quotient	Brent's method	Binary tree	Computed chaining (6-bit link)	(2-bit link)
20	1.100	1.106	1.125	1.15	1.102	1.102	1.070	1.070
40	1.200	1.232	1.333	1.277	1.217	1.217	1.168	1.214
60	1.300	1.379	1.750	1.527	1.367	1.364	1.264	1.381
70	1.350	—	2.167	1.720	1.444	—	1.323	1.528
80	1.400	1.566	3.000	2.011	1.599	1.579	1.356	1.715
90	1.450	1.674	5.500	2.558	1.802	1.751	1.408	2.062
95	1.475	1.734	10.500	3.153	1.972	1.880	1.433	2.414
99	1.495	1.783	50.500	4.651	2.242	2.049	1.601	3.330
100	1.500	—	—	6.522	2.494	2.134	—	—

*Theoretical results; mean probes = $1 + \alpha/2$.

†Theoretical results; mean probes = $(1 - \alpha/2) / (1 - \alpha)$

TABLE 3.4 SEARCH, RELOCATION, AND STORAGE COMPARISONS

Criteria	LISCH	Progressive overflow	Linear quotient	Brent's method	Binary tree	Computed chaining
Successful search						
Best case	1	1	1	1	1	1
Worst case	n	n	n	n	n	n
Move an item	No	No	No	Yes	Yes	Yes
Extra storage	Yes	No	No	No	No	Yes

Yöntem	Avantajı	Dezavantajı	Ne Zaman Tercih?
LISCH, LICH, EISCH	Performansları oldukça yüksektir.	Link alanı için ekstra yer gereksinimi vardır.	Uygulama için yeterli yer söz konusu olduğunda
Progressive Overflow	Az yer gereksinmesi ve basit yapısı	Performansı oldukça düşüktür.	Tercih edilmemelidir.
Linear Quotient	Fazladan yer gerektirmemekte ve performansı ortalamaya yakındır.	Performansı ortalamanın altındadır.	Yer kısıtlaması olduğunda ve performans çok fazla önemli değilse
Brient's Method	Fazladan yer gerektirmemekte ve performansı iyidir.	Linkli yapıları kullanan yaklaşımlara göre performans düşüktür.	Yer kısıtlaması olduğunda tercih edilmelidir.
Binary Tree	Fazladan yer gerektirmemekte ve performansı iyidir.	Linkli yapıları kullanan yaklaşımlara göre performans düşüktür.	Yer kısıtlaması olduğunda tercih edilmelidir.
Computed Chaining	Çok az bir yer gerektirmekte ve performansı oldukça iyidir.	Performansı LISCH, LICH, EISCH'den birazcık düşüktür.	Fazladan yer olduğunda, performans önemli ise ve ekleme için ekstra zaman harcanabilecekse

Perfect Hashing

- Perfect hashing bir anahtar değerini eşisiz bir adresle eşleştirir.

  hash (key) -> unique address.

- Az sayıda kayda sahip olan dosyalar için tercih edilir.

- Eğer dosya potansiyel adres sayısı ile anahtar sayısı aynı ise bu fonksiyona minimum perfect hashing fonksiyonu denir.

Cichelli Algoritması

Bu algoritmada aşağıdaki fonksiyonlar kullanılır.

h_0 -> length(key)

h_1 -> first_karakter(key)

h_2 -> last_karakter(key)

g -> T(x)

T her bir x karakteri için hazırlanan tablodaki değeri gösterir.

Bu tablonun hazırlanması oldukça zaman alıcıdır.

TABLE 3.6 VALUES ASSOCIATED WITH THE CHARACTERS OF THE PASCAL RESERVED WORDS

A = 11	B = 15	C = 1	D = 0	E = 0	F = 15
G = 3	H = 15	I = 13	J = 0	K = 0	L = 15
M = 15	N = 13	O = 0	P = 15	Q = 0	R = 14
S = 6	T = 6	U = 14	V = 10	W = 6	X = 0
Y = 13	Z = 0				

Yukarıdaki tablo baz alınarak “ begin ” kelimesi için hash fonksyinunun oluşturacağı adres aşağıdaki gibi olacaktır.

$$
\begin{aligned}
\text{p.hash}(\textbf{begin}) &= 5 + T(h_1(\text{key})) + T(h_2(\text{key})) \\
&= 5 + T(\text{b}) + T(\text{n}) \\
&= 5 + 15 + 13 \\
&= 33
\end{aligned}
$$

Cichelli Algoritması (Tablonun Oluşturulması)

- Tablonun oluşturulması için önce bütün anahtarların ilk ve son karakterlerinin kullanım sıklığı belirlenir.
- Her bir anahtara ilk ve son karakterlerinin kullanım sıklıklarının toplam değeri atanır.
- Anahtarlar atanan değerlere göre büyükten küçüğe doğru sıralanır.
- Geridönüş yöntemi (backtracking) kullanılarak herbir başlangıç ve bitiş karakterine değer atanır.

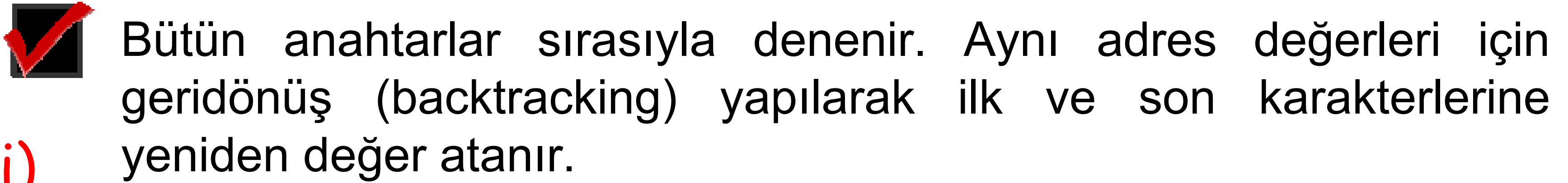
Örn: Kullanılacak kelimeler aşağıdakiler olsun.

cat, ant, dog, gnat, chimp, rat, toad

İlk ve son karakterlerin sıklığı:
a = 1 c = 2 d = 2 g = 2 p = 1 r = 1 t = 5

cat	7	toad	7	toad	7
ant	6	gnat	7	gnat	7
dog	4	cat	7	dog	4
gnat	7	rat	6	cat	7
chimp	3	ant	6	rat	6
rat	6	dog	4	ant	6
toad	7	chimp	3	chimp	3

İlk ve son karakteri daha önceden geçen anahtar kelimeler, en son geçtikleri yerden sonra tekrar sıraya konulur.



i) Bütün anahtarlar sırasıyla denenir. Aynı adres değerleri için geridönüş (backtracking) yapılarak ilk ve son karakterlerine yeniden değer atanır.

$t = 0 \quad d = 0$

p.hash(toad) = 4

ii) g=0 değeri atandığında , p.hash(gnat) = 4 sonucunu üretir ki bu da çakışmaya neden olur. g=1 değerini vermek çakışmayı çözecektir.

$t = 0 \quad d = 0 \quad g = 1$

p.hash(toad) = 4
p.hash(gnat) = 5

iii) “dog” kelimesinde bulunan d ve g harflerinin değerleri daha önce atanmış idi (d=0 ve g=1). Bu durum çakışmaya neden olacaktır. Bunun önüne geçmek için backtracking yapılarak bir önceki aşamada “goat” kelimesi için belirlenmiş oldunan g değeri 2 yapılır.

$t = 0 \quad d = 0 \quad g = 2$

p.hash(toad) = 4
p.hash(gnat) = 6

iv) Böylelikle dog kelimesi için olan çakışma problemi de ortadan kalkacaktır.

$t = 0 \quad d = 0 \quad g = 2$

p.hash(toad) = 4
p.hash(gnat) = 6
p.hash(dog) = 5

v) Aynı işlemler "cat" anahtarı için de tekrarlanır. İlk etapda c=0 verilir ve herhangi bir çakışma olmadığı görülür.

$t = 0 \quad d = 0 \quad g = 2 \quad c = 0$

p.hash(toad) = 4
p.hash(gnat) = 6
p.hash(dog) = 5
p.hash(cat) = 3

vi) Bir sonraki kelime olan "rat" kelimesi için bir çakışmaya sebep olmamak için r=4 değeri atanır.

$t = 0 \quad d = 0 \quad g = 2 \quad c = 0 \quad r = 4$

p.hash(toad) = 4
p.hash(gnat) = 6
p.hash(dog) = 5
p.hash(cat) = 3
p.hash(rat) = 7

vii) "ant" kelimesi için çakışmaya neden olur. Yine bu çakışmayı çözebilmek için backtracking uygulanır ve g=3 olarak değiştirilir. Ve a için de 0'dan başlayarak çakışma yaratmayacak değerler denenir ve en sonunda a için de 3 değeri bulunur.

$t = 0 \quad d = 0 \quad g = 3 \quad c = 0 \quad r = 2$

p.hash(toad) = 4
p.hash(gnat) = 7
p.hash(dog) = 6
p.hash(cat) = 3
p.hash(rat) = 5

a)

$t = 0 \quad d = 0 \quad g = 4 \quad c = 0 \quad r = 2 \quad a = 3$

p.hash(toad) = 4
p.hash(gnat) = 8
p.hash(dog) = 7
p.hash(cat) = 3
p.hash(rat) = 5
p.hash(ant) = 6

b)

viii) "chimp" kelimesi için üretilecek eşsiz adresi sağlayan p değeri ise 4'tü

$t = 0 \quad d = 0 \quad g = 4 \quad c = 0 \quad r = 2 \quad a = 3 \quad p = 4$

p.hash(toad) = 4
p.hash(gnat) = 8
p.hash(dog) = 7
p.hash(cat) = 3
p.hash(rat) = 5
p.hash(ant) = 6
p.hash(chimp) = 9

En son durum

Avantaj / Dezavantaj

- Geri dönüşlerin (Backtrackings) toplam sayısı hesaplama süresini oldukça etkilemektedir.

- Bu algoritmadayı kullanan yapılarda;

  • Aynı uzunlukta ve aynı ilk ve son karaktere sahip 2 anahtar bulunamaz.

  • Eleman sayısı fazla olmaya başladığında listeleri alt gruplara bölmek gerekebilir.