Çakışmaların Dinamik Yaklaşımlarla Çözümlenmesi

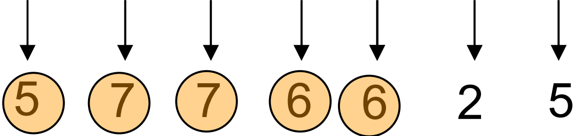
BRIENT'S YÖNTEMİ

- Şimdiye kadar ele alınan çakışma çözme yöntemleri statik yöntemlerdi. Bu yöntemlerde anahtar başlangıçta belirlenecek olan bir adrese yerleştirilmekte ve bir daha yeri değiştirilmemektedir.

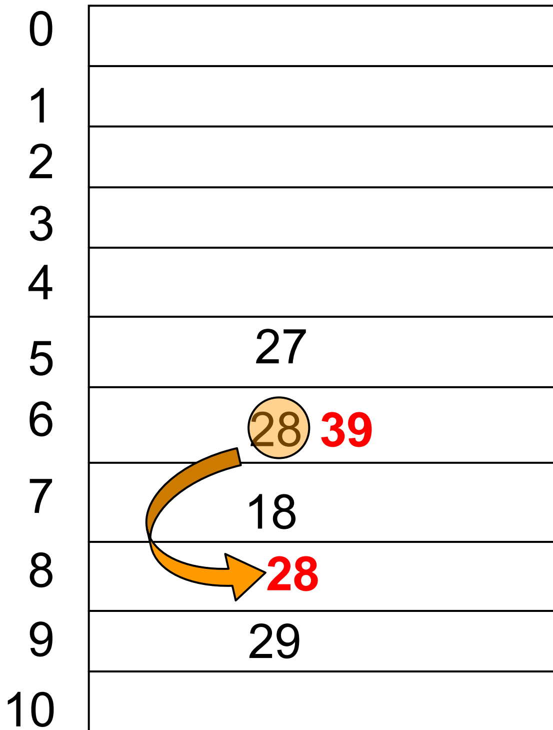
- BRINENT's yönteminde ise başlangıçta bir adrese yerleştirilen anahtarlar daha sonra dinamik olarak yer değiştirebilmektedir.

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

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



Konumlar



Brient’s Method

Bu örnekte 39’u yerleştirmek için mümkün olan 3 pozisyon vardır. Bunlar:

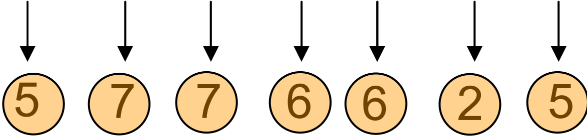
P1→6 (Dolu)
P2→9 (Dolu)
P3→1 (Dolu)

Bunun yerine 39 ilk olarak p1’e yerleştirilirse 39’a ulaşmak için sadece 1 probe gerekecektir. Fakat böyle bir durumda da p1 konumundaki 28 değerini de başka bir yere taşımak gerekecektir.

Linear Quotient metoduna göre 28’in artım değeri 2 olup, boş olan 8 nolu konuma yerleştirilebilir.

Diğer anahtar değerlerini de bu yönteme göre tabloya yerleştirirsek aşağıdaki görünümü elde etmiş oluruz.

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

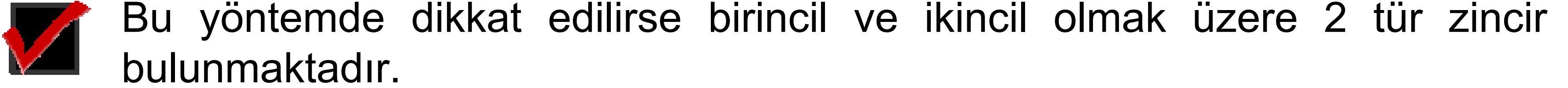


Konumlar

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

$$\sum = \frac{12}{7} = \mathbf{1{,}71}$$

Brient's Method



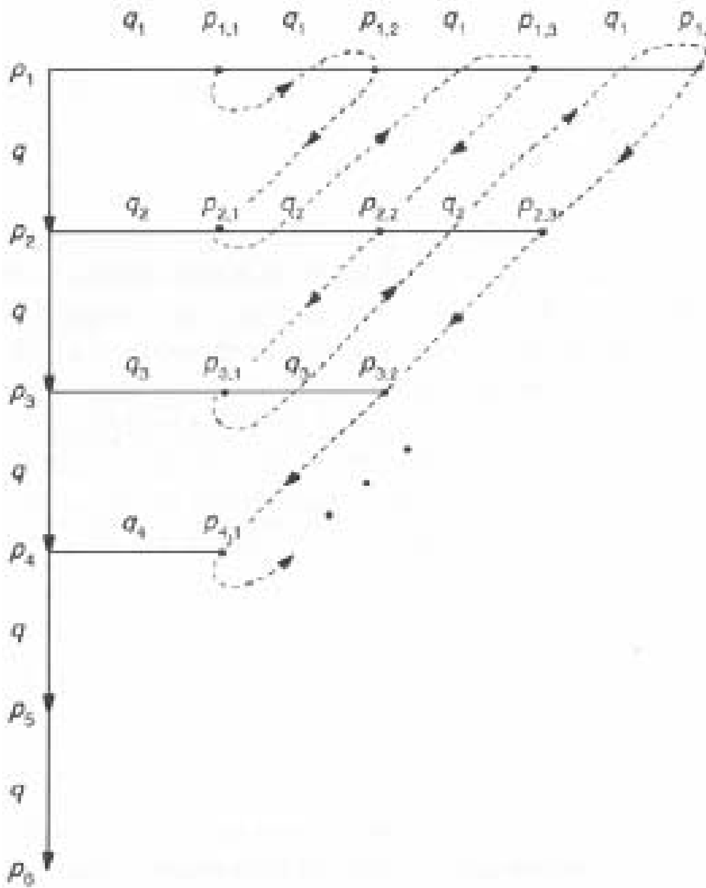
Bu yöntemde dikkat edilirse birincil ve ikincil olmak üzere 2 tür zincir bulunmaktadır.

- Birincil (primary) zincir, bir kaydın insert veya retrieval edilmesi sırasında oluşan halkadır.
- İkincil (secondary) zincir, bir kaydın kendi birincil zinciri içinde move edilmesi sonucu oluşan halkadır.

Brient’s yöntemi sadece kayıt eklemek için kullanılır. Retrieval için linear quotient kullanılır.

Bir kaydın birincil zincire eklenmesi ve diğer kaydın ikincil zincire taşınması arasında sürekli olarak işlem yapılır.

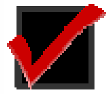
Kayıtların silinmesi sırasında tombstone(işaretçi) kullanılır.



Dikey çizgi birincil zinciri gösterir.

Yatay çizgiler, ikincil zinciri gösterir.

q değeri eklenmek istenilen kaydın birincil zincir içerisindeki artım miktarını gösterir.

 q_i değeri ise ikincil zincirdeki artım miktarını göstermektedir.

i indisi birincil zincir üzerindeki probe sayısını ifade eder.

j indisi taşınacak kaydın ek probe sayısını ifade eder.

Hiçbir şekilde taşıma yapmadan gerekli olan probe sayısı **s** olsun.

Bu durumda s değerini azaltmak için **$(i+j)<s$** olmalıdır.

$i = j$ olması durumunda keyfi olarak hareket edilebilir.

Probe sayısında hiç bir şekilde bir azalma olmuyorsa işlem sonlandırılır.

i+j<s şartını sağlamak için hesaplamalar yapmak yerine aşağıdaki grafik takip edilirse kolaylık sağlanmış olacaktır. Bu yapıda bulunan ilk boş yerde algoritma sonlandırılır.

q_1 $p_{1,1}$ q_1 $p_{1,2}$ q_1 $p_{1,3}$ q_1 $p_{1,4}$

p_1 q q_2 $p_{2,1}$ q_2 $p_{2,2}$ q_2 $p_{2,3}$

p_2 q q_3 $p_{3,1}$ q_3 $p_{3,2}$

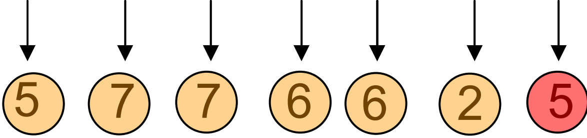
p_3 q q_4 $p_{4,1}$

p_4 q p_5 q p_6

Örnek

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

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



Konumlar

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

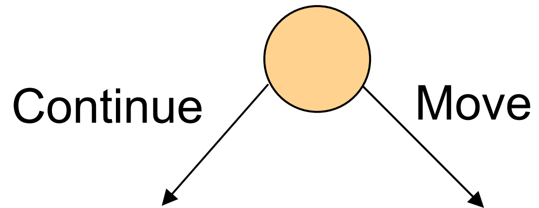
Brient's Method

Kütük

16 anahtarının grafiği kullanarak eklenmesi

BINARY TREE

Binary tree, her bir saklama adresi için 2 seçeneğin olduğu durumlarda uygundur.



Continue: Zincir üzerindeki bir sonraki adrese devam edilmesini belirtir.

Move: Bir adresteki nesenyi (kaydı) bir sonraki konuma taşı.

Binary tree, yukarıdan aşağıya ve soldan sağa doğru oluşturulur.

Bu yapı, nesnelerin hangi adreste saklanacağına karar vermek için kullanılır.

Her bir ekleme için yeni bir binary tree oluşturulur.

Boş bir yaprak veya algoritmanın dolu olması durumunda algoritma sonlanır.

Brient's yöntemine göre daha fazla ön-işlem gerektirir.

Brient's de olduğu gibi ekleme işlemi için kullanılır, arama işlemi için linear quotient tercih edilir.

Örnek

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

27, 18, 29, 28, 39, 13, 16, 41, 17, 19

↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓

5 7 7 6 6 2 5 8 6 8

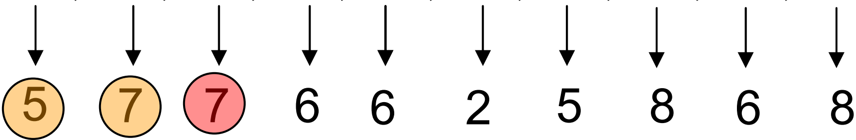
şeklinde verilen anahtarları sırasıyla binary tree yardımıyla çakışmalarını çözerek tablomuza kaydedelim.

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

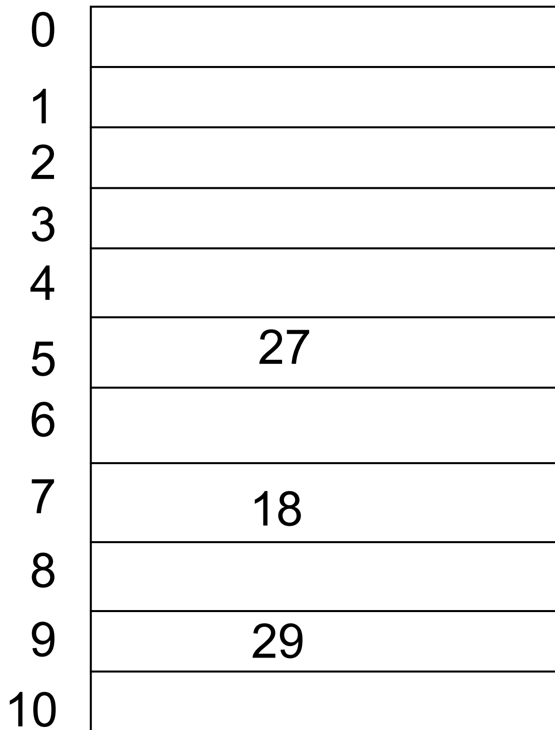
Binary Tree

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

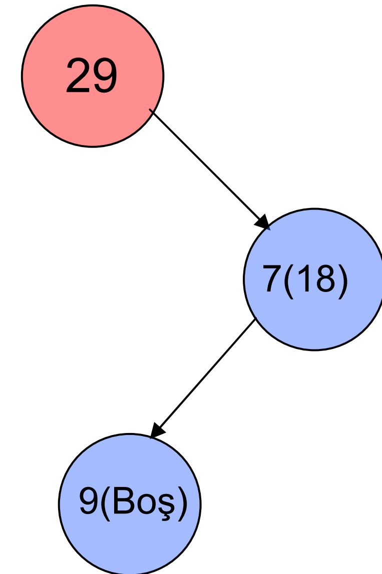
27, 18, 29, 28, 39, 13, 16, 41, 17, 19



29’da bir çakışma olacak ve 29 için binary tree oluşturlursa

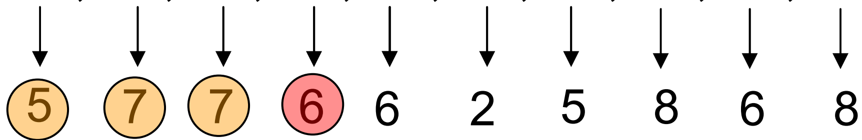


Binary Tree



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

27, 18, 29, 28, 39, 13, 16, 41, 17, 19

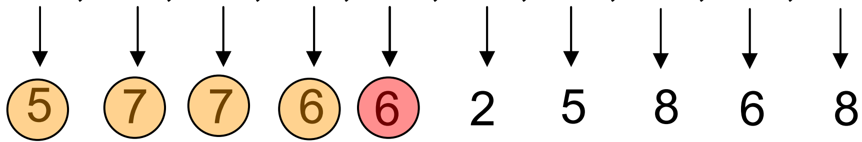


0	
1	
2	
3	
4	
5	27
6	28
7	18
8	
9	29
10	

Binary Tree

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

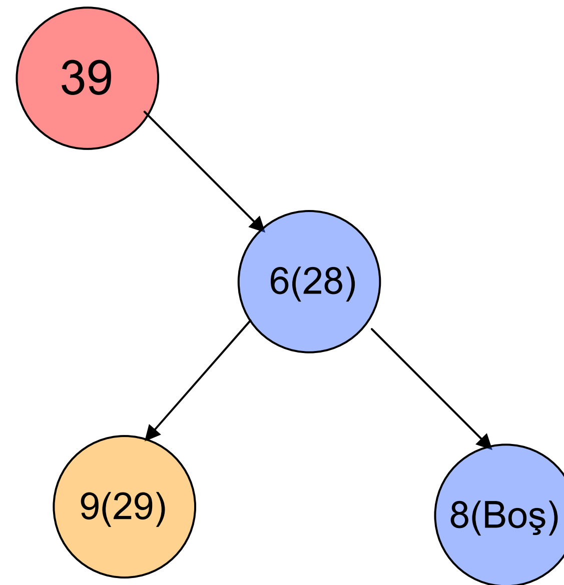
27, 18, 29, 28, 39, 13, 16, 41, 17, 19



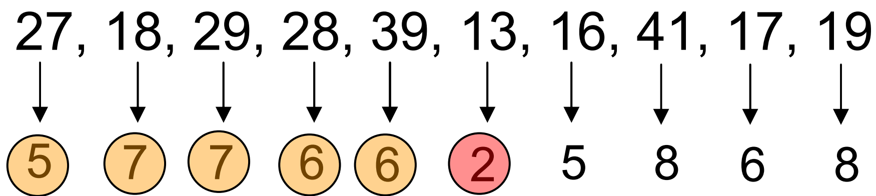
39’da bir çakışma olacak ve 39 için binary tree oluşturlursa

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

Binary Tree



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

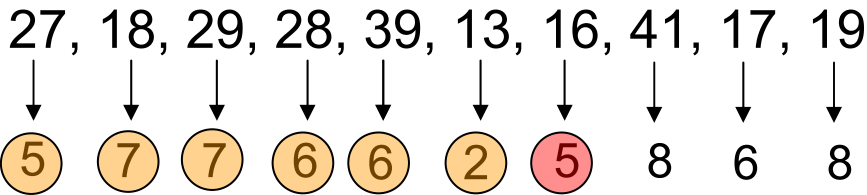


0	
1	
2	13
3	
4	
5	27
6	39
7	18
8	28
9	29
10	

Binary Tree

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

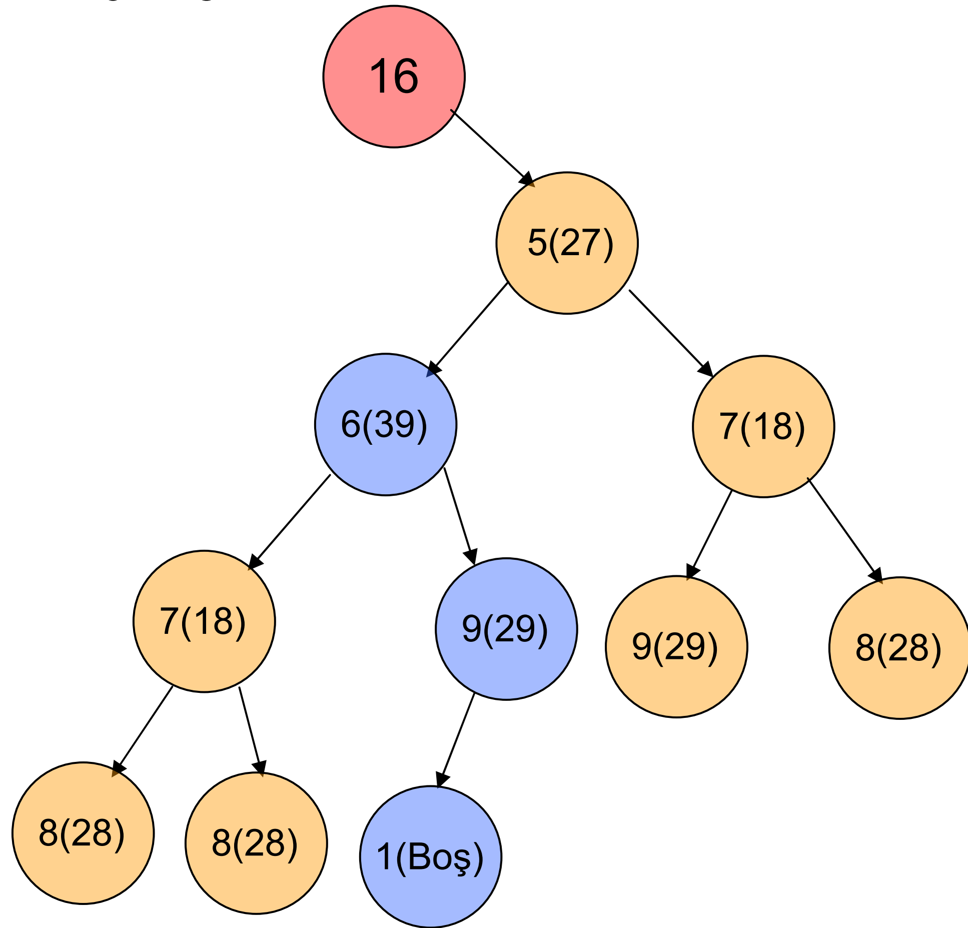
27, 18, 29, 28, 39, 13, 16, 41, 17, 19



16'da bir çakışma olacak ve 16 için binary tree oluşturlursa

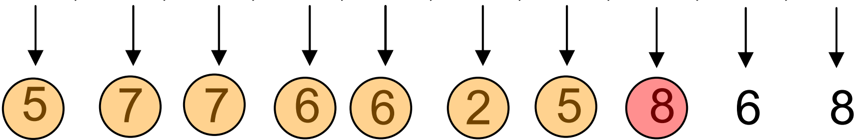
0	
1	39
2	13
3	
4	
5	27
6	16
7	18
8	28
9	29
10	

Binary Tree



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

27, 18, 29, 28, 39, 13, 16, 41, 17, 19

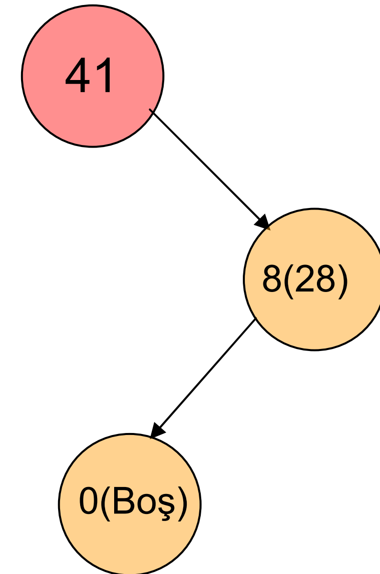


6 8

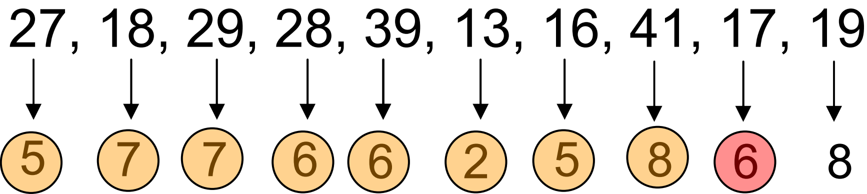
41’de bir çakışma olacak ve 41 için binary tree oluşturlursa

0	41
1	39
2	13
3	
4	
5	27
6	16
7	18
8	28
9	29
10	

Binary Tree



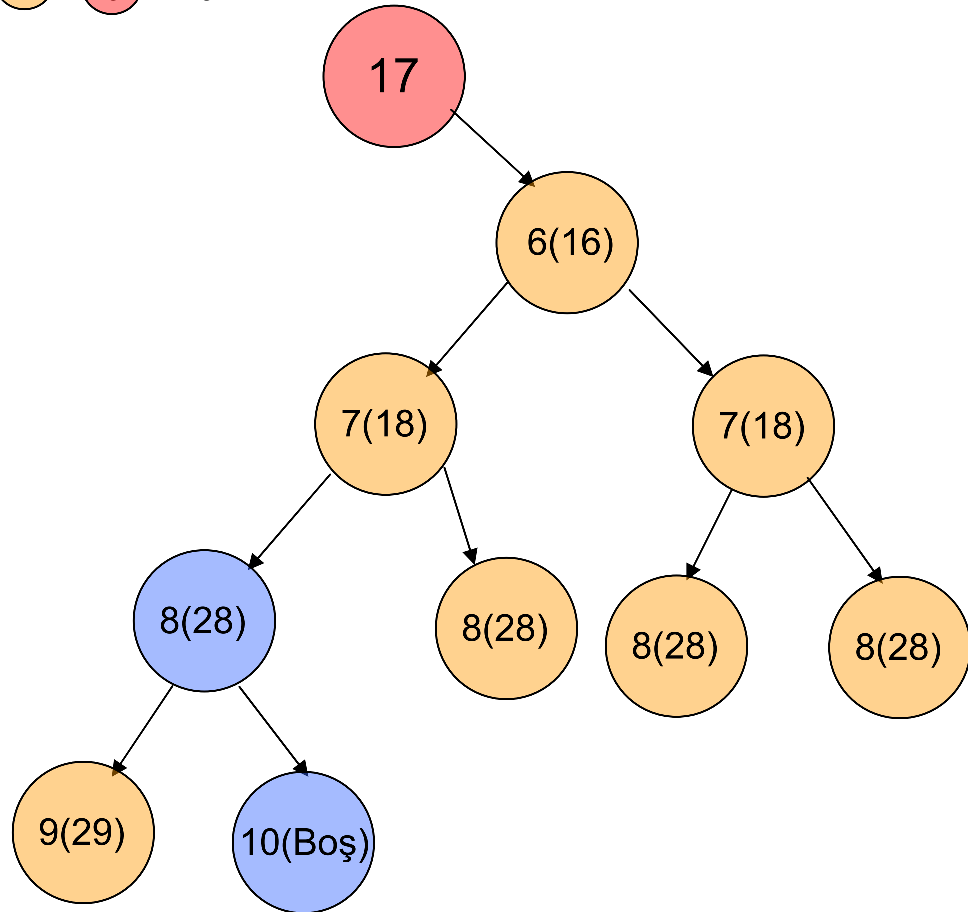
Hash(key) = key **mod 11**



17’de bir çakışma olacak ve 17 için binary tree oluşturlursa

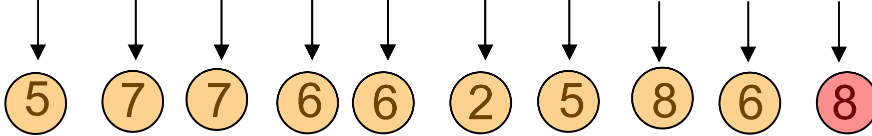
0	41
1	39
2	13
3	
4	
5	27
6	16
7	18
8	17
9	29
10	28

Binary Tree

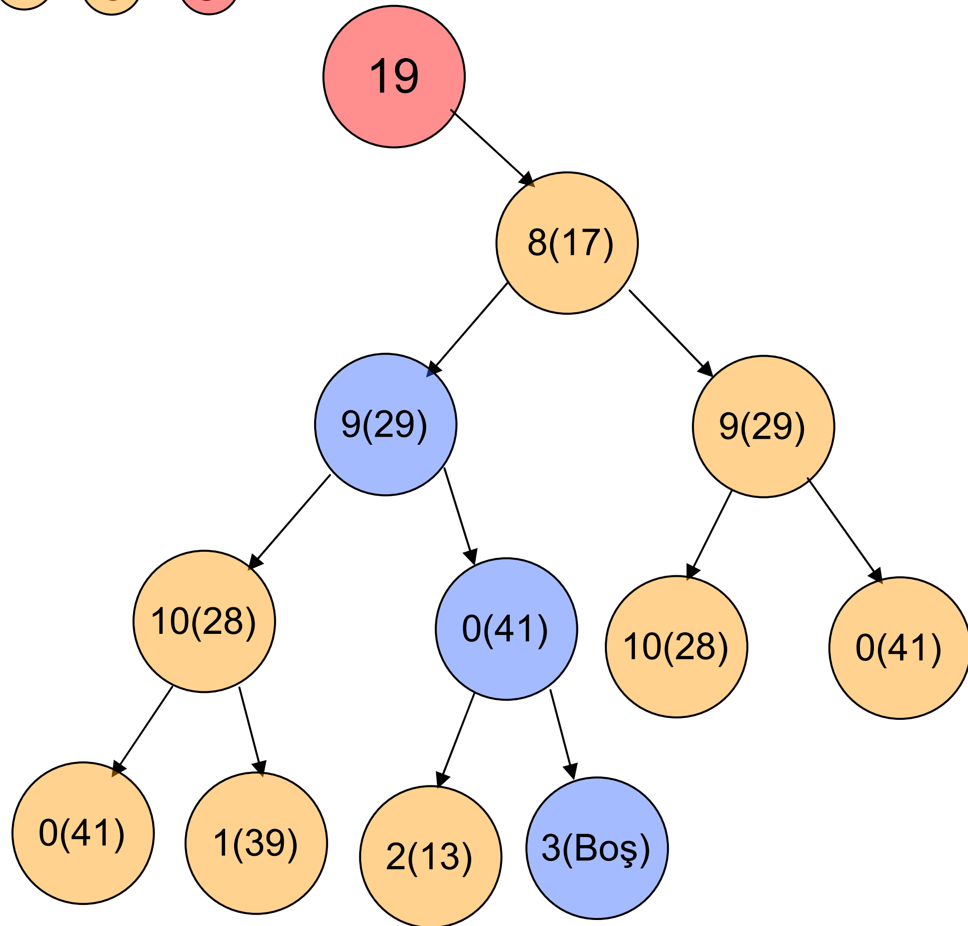


Hash(key) = key **mod 11**

27, 18, 29, 28, 39, 13, 16, 41, 17, 19



19’da bir çakışma olacak ve 19 için binary tree oluşturlursa



0	29
1	39
2	13
3	41
4	
5	27
6	16
7	18
8	17
9	19
10	28

Binary Tree