



ÖZYİNELEME (RECURSION)

ÖZYİNELEME(RECURSION)

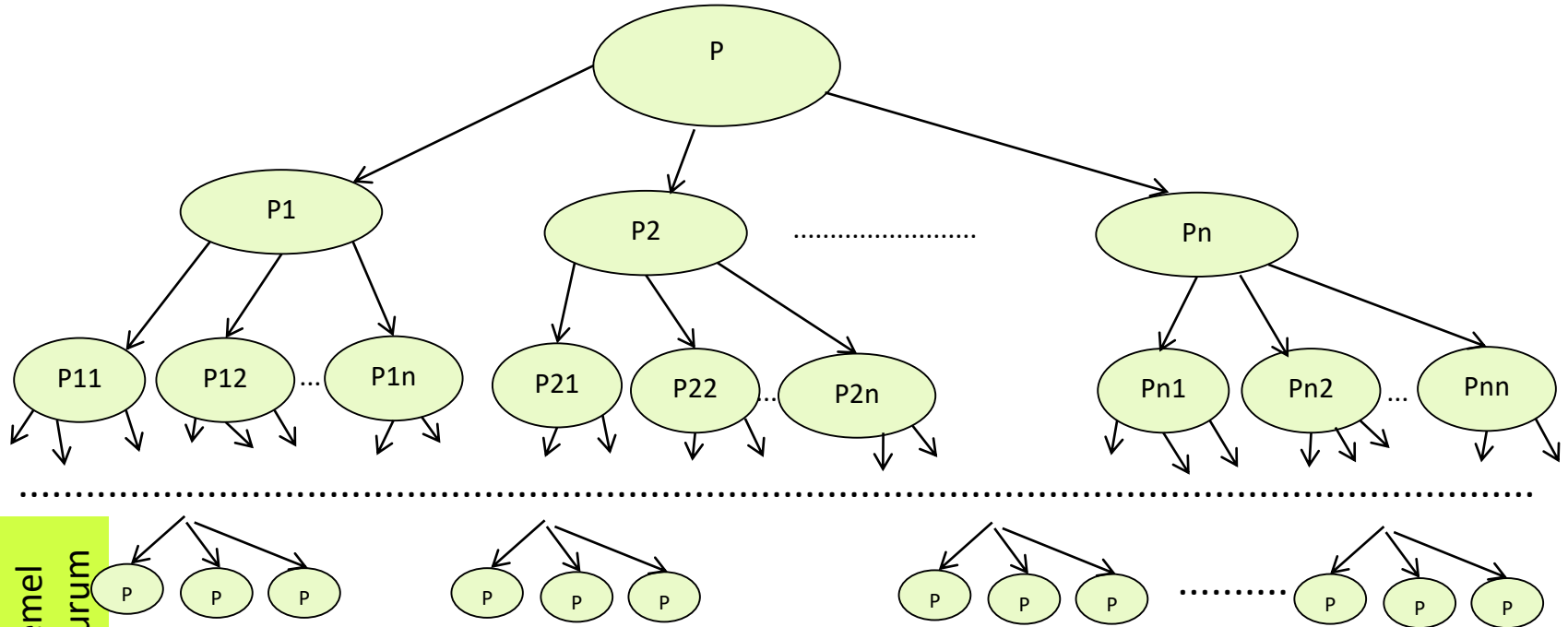
- Özyineleme, bir işlemin sonucu kendine önceki bir veya iki işlem sonucuna bağlı olarak bulunmasıdır.
- Kendi kendisini doğrudan veya dolaylı olarak çağıran fonksiyonlara **özyineli (recursive)** fonksiyonlar adı verilir.
- Özyineleme bir problemi benzer şekilde olan daha basit alt problemlere bölünerek çözülmesini sağlayan bir tekniktir.
- Alt problemler de kendi içlerinde başka alt problemlere bölünebilir.
- Alt problemler çözülebilecek kadar küçülünce bölme işlemi durur.
- Özyineleme, döngülere (iteration) alternatif olarak kullanılabilir.

ÖZYİNELEME

- Bir problem özyineli olmayan basit bir çözüme sahiptir. Problemin diğer durumları özyineleme ile durdurma durumuna (stopping case) indirgenebilir.
- Özyineleme işlemi durdurma durumu sağlanınca sonlandırılır.
- Özyineleme güçlü bir problem çözme mekanizmasıdır.
 - Çoğu algoritma kolayca özyinelemeli şekilde çözülebilir.
 - Fakat sonsuz döngü yapmamaya dikkat edilmeli.
- **Genel yazımı-kaba kod:**
 - *if (durdurma durumu sağlandıysa)*
 - *çözümü yap*
 - *else*
 - *problemi özyineleme kullanarak indirge*

Özyineleme- Böl & Yönet Stratejisi

- Bilgisayar birimlerinde önemli bir yere sahiptir:
 - Problemi küçük parçalara böl
 - Her bir parçayı bağımsız şekilde çöz
 - Parçaları birleştirerek ana problemin çözümüne ulaş



Temel
Durum

Özyineleme- Böl & Yönet Stratejisi

```
/* P problemini çöz */
Solve(P) {
    /* Temel durum(s) */
    if P problemi temel durumda ise
        return çözüm

    /* (n>=2) için P yi P1, P2, ..Pn şeklinde parçalara böl */
    /* Problemleri özyinelemeli şekilde çöz */
    S1 = Solve(P1); /* S1 için P1 problemini çöz */
    S2 = Solve(P2); /* S2 için P2 problemini çöz*/
    ...
    Sn = Solve(Pn); /* Sn için Pn problemini çöz */

    /* Çözüm için parçaları birleştir. */
    S = Merge(S1, S2, ..., Sn);

    /* Çözümü geri döndür */
    return S;
} //bitti-Solve
```

ÖZYİNELEME

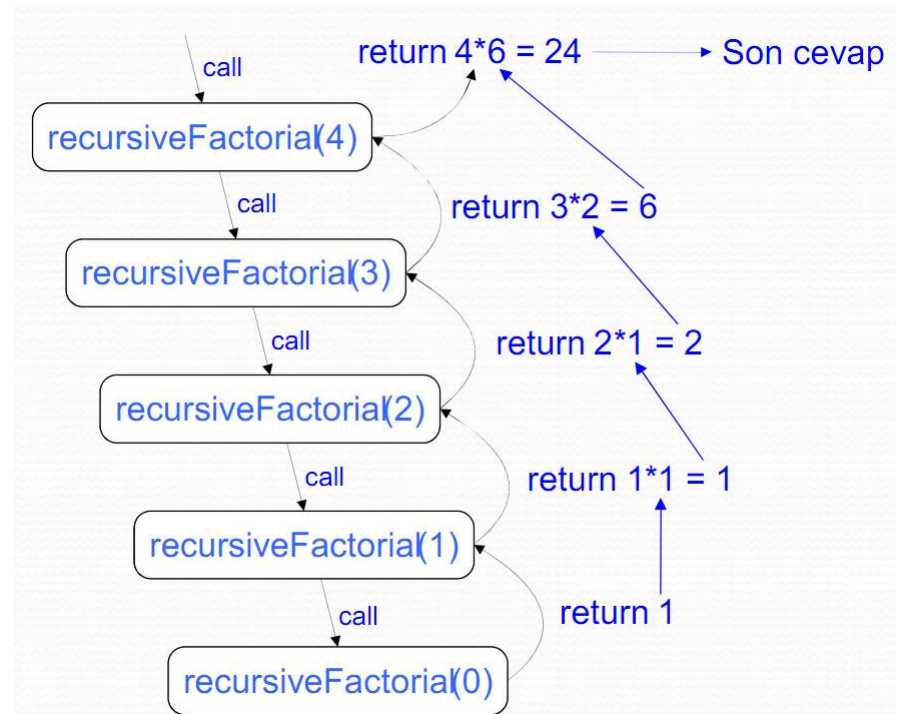
- Faktöryel fonksiyonu: Klasik bir özyineleme örneğidir:
 - $n! = 1 * 2 * 3 * \dots * (n - 1) * n$
- Faktoriyel işlemini özyineli tanımlamak için küçük sayıların faktöriyeli şeklinde tanımlamak gerekir.
 - $n! = n * (n - 1) !$
 - Durdurma durumu $0 ! = 1$ olarak alınır.
- Her çağırmada n değeri bir azaltılarak durdurma durumuna ulaşılır.
- Özyineli tanımlama:
 - $f(n) = \begin{cases} n = 1, & \text{if } n = 0 \\ n * f(n - 1), & \text{if } n > 0 \end{cases}$

ÖZYİNELEME

- Aşağıdaki kod çalıştığında n sayısının faktöriyel değerini hesaplar.
- ```
int recursiveFactorial(int n)
```
- ```
{
```
- ```
 if (n == 1) return(1);
```
- ```
    else return ( n * recursiveFactorial( n - 1 ));
```
- ```
}
```
- n! değerini hesaplar ve bulduğu değeri return ile gönderir.

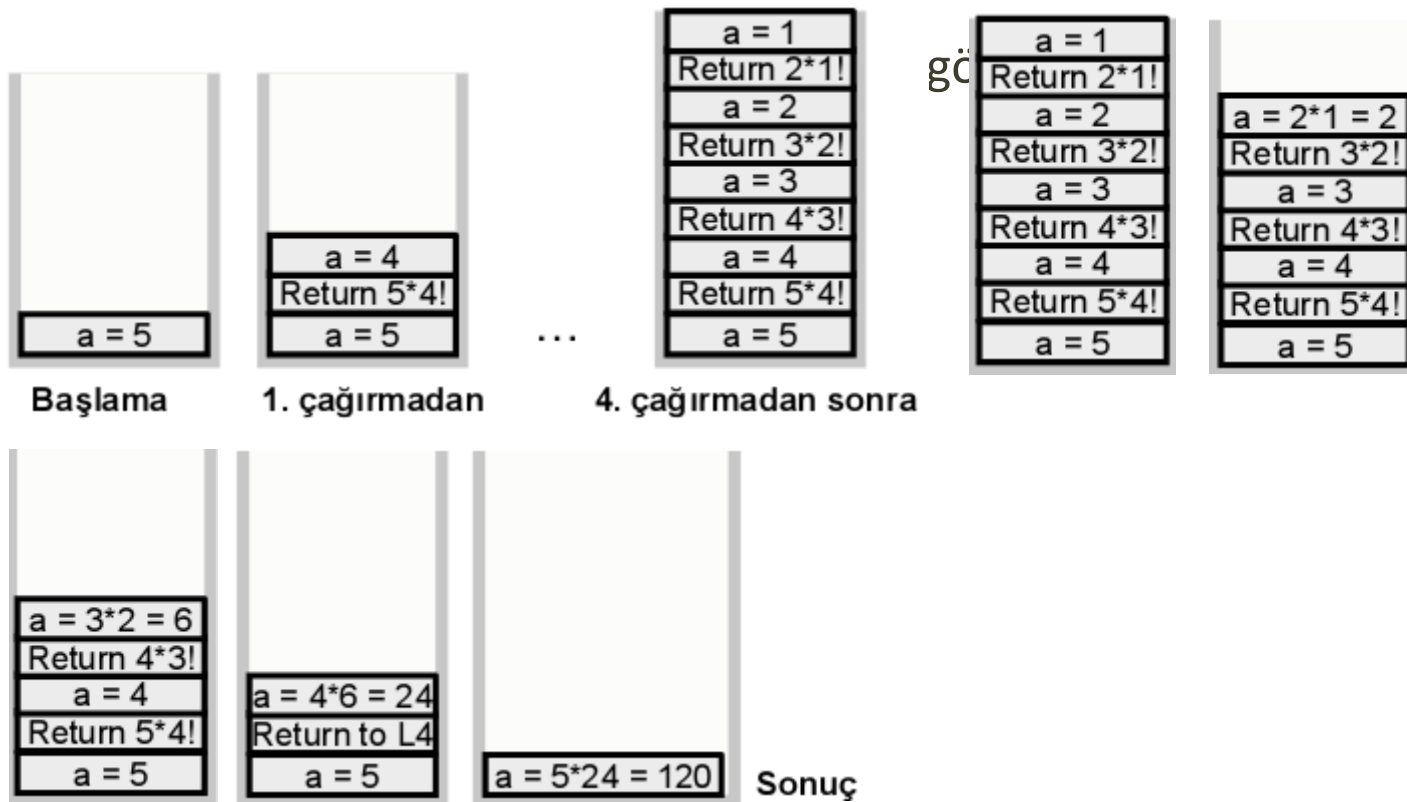
# ÖZYİNELEME

- Özyineleme izleme
- Her özyineleme çağrımı için bir kutu
- Her çağırandan çağrılana bir ok
- Her çağrılıandan çağırana çizilen ok geri dönüş değerini gösterir.





# ÖZYİNELEME



# ÖZYİNELEME

- Genellikle iterative fonksiyonlar zaman ve yer bakımından daha etkindirler.
- Iterative algoritma döngü yapısını kullanır.
- Özyineleme algoritması dallanma (branching) algoritmasını kullanır.
- Fonksiyon özyineli olarak her çağrılışında yerel değişkenler ve parametreler için bellekte yer ayrılır.
- Özyineleme problemin çözümünü basitleştirebilir, sonuç genellikle kısadır ve kod kolayca anlaşılabilir.
- Her özyinelemeli olarak tanımlanmış problemin iterative çözümüne geçiş yapılabilir.

# ÖZYİNELEME

## ○ Recursive

```
○ int recFact(int n)
○ {
○ if(n == 1) return(1);
○ else
○ return(n * recFact(n - 1));
○ }
```

## ■ Iterative

```
■ int iteFact(int n)
■ {
■ int araDeger = 1;
■ for (int i = n; i > 0; i--)
■ araDeger * = i;
■ return araDeger;
■ }
```

# FaktoriyelOrnek.java

```
import java.io.*;
class FaktoriyelOrnek
{ static int sayi;
 public static void main(String args[]) throws IOException
 {
 System.out.print("Sayi veriniz :"); System.out.flush();
 sayi=getInt(); int sonuc = factorial(sayi);
 System.out.println(sayi+"! =" +sonuc);
 }
 public static int factorial(int n)
 {
 if(n==0) return 1;
 else return(n*factorial(n-1));
 }
}
```

```
public static String getString() throws IOException
{
 InputStreamReader isr = new InputStreamReader(System.in);
 BufferedReader br = new BufferedReader(isr);
 String s = br.readLine();
 return s;
}
public static int getInt() throws IOException
{ String s = getString();
 return Integer.parseInt(s);
}
}
```

# N'ye Kadar Olan Sayıların Toplamı

- Problemimizin 1'den n'ye kadar sayıların toplamı olduğunu varsayalım.
- Bu problemi özyinelemeli nasıl düşüneceğiz:
  - $\text{Topla}(n) = 1+2+\dots+n$  ifadesini hesaplamak için
    - $\text{Topla}(n-1) = 1+2+\dots+n-1$  ifadesini hesapla (aynı türden daha küçük bir problem)
    - $\text{Topla}(n-1)$  ifadesine  $n$  ekleyerek  $\text{Topla}(n)$  ifadesi hesaplanır.
    - $\text{Topla}(n) = \text{Topla}(n-1) + n$ ;
  - Temel durumu belirlememiz gerekiyor.
    - Temel durum, (alt problem) problemi bölmeye gerek kalmadan kolayca çözülebilen problemdir.
    - $n = 1$  ise,  $\text{Topla}(1) = 1$ ;

# Topla(4) için Özyineleme Ağacı

```

/* Topla 1+2+3+...+n */
int Topla(int n){
 int araToplam = 0;

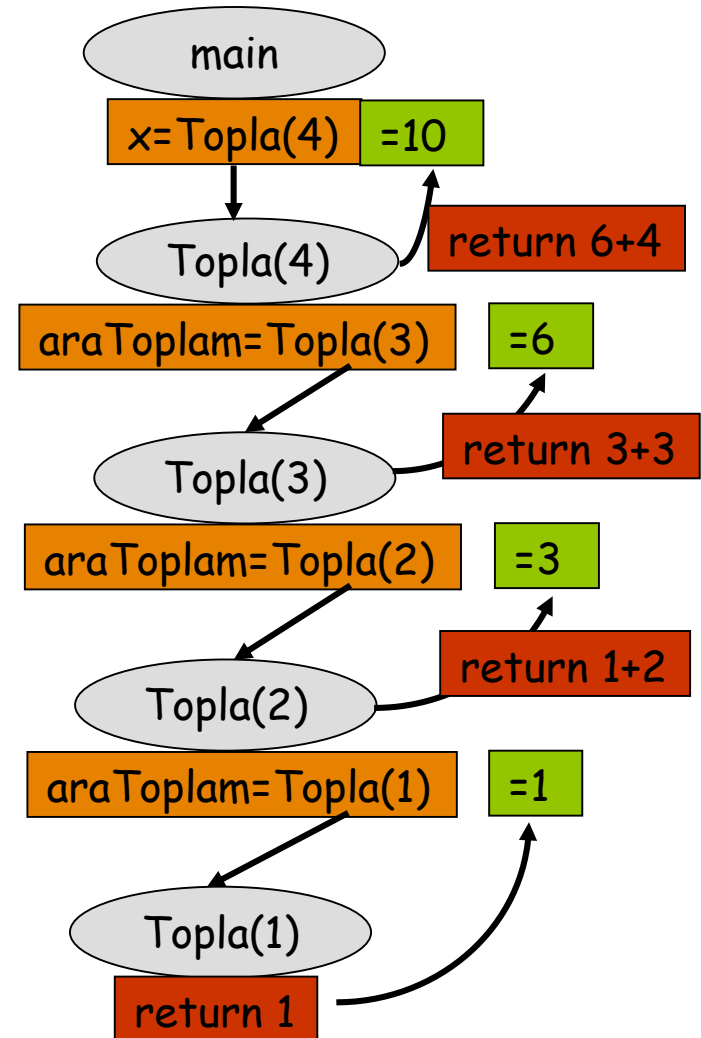
 /* Temel Durum */
 if (n == 1) return 1;

 /* Böl ve Yönet */
 araToplam = Topla(n-1);

 /* Birleştirir */
 return araToplam + n;
} /* bitti-Topla */

Public ... main(...){
print("Topla: " + Topla(4));
} /* bitti-main */

```



# Topla(n)'nin çalışma zamanı

```
/* Topla 1+2+3+...+n */
int Topla(int n){
 int araToplam = 0;

 /* Temel durum */
 if (n == 1) return 1;

 /* Böl ve yönet */
 araToplam = Topla(n-1);

 /* Birleştirir */
 return araToplam + n;
} /* bitti-araToplam */
```

$$T(n) = \begin{cases} n = 1 \rightarrow 1 \text{ (Temel durum)} \\ n > 1 \rightarrow T(n-1) + 1 \end{cases}$$

# $a^n$ İfadesini Hesaplama-Pow(a,n)

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x, n-1) & \text{else} \end{cases}$$

```
/* a^n hesapla */
double Ust(double a, int n){
 double araSonuc;

 /* Temel durum */
 if (n == 0) return 1;
 else if (n == 1) return a;

 /* araSonuc = a^(n-1) */
 araSonuc = Ust(a, n-1);

 /* Birleştirir */
 return araSonuc*a;
} /* bitti-Ust */
```

- Böl, yönet & birleştir işlemleri bir ifade ile yapılabilir.

```
/* Hesapla a^n */
double Ust(double a, int n){
 /* Temel durum */
 if (n == 0) return 1;
 else if (n == 1) return a;

 return Ust(a, n-1)*a;
} /* bitti-Ust */
```



# Ust(3, 4) için Özyineleme ağacı

```

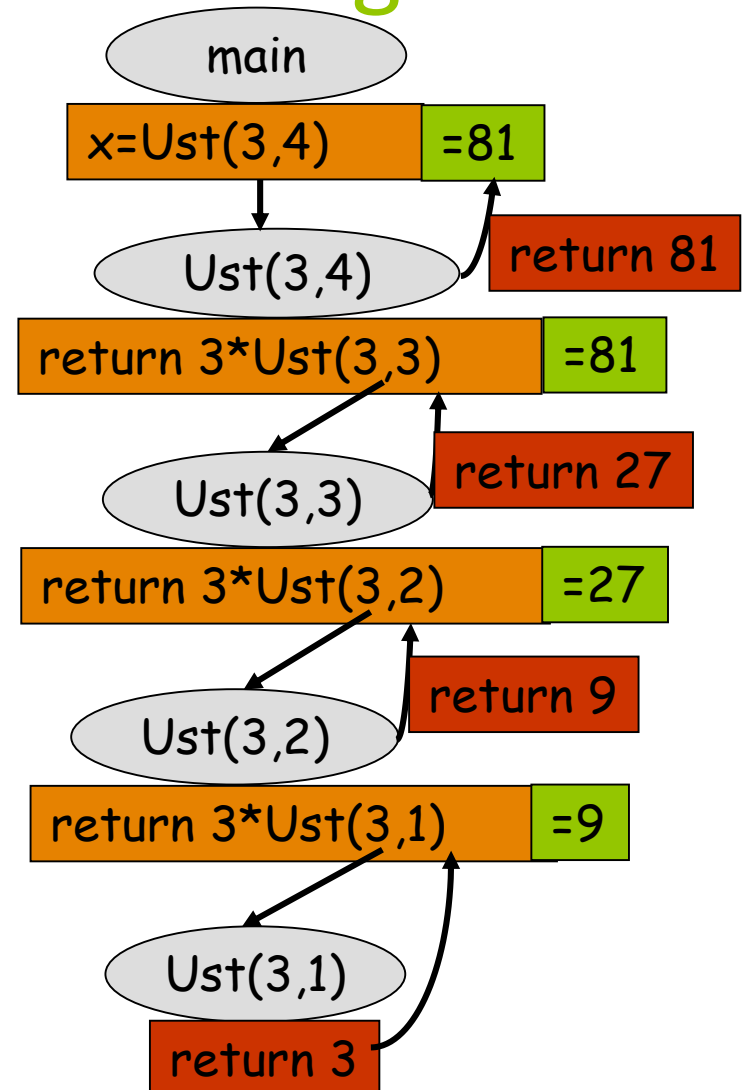
/* Hesapla a^n */
double Ust(double a, int n){
 /* Temel durum */
 if (n == 0) return 1;
 else if (n == 1) return a;

 return a * Ust(a, n-1);
} /* bitti-Ust */

Public ... main(...){
 double x;

 x = Ust(3, 4);
} /* bitti-main */

```



## Ust(a, n)'nin Çalışma Zamanı

```
/* Hesapla a^n */
double Ust(double a, int n){
 /* temel durum */
 if (n == 0) return 1;
 else if (n == 1) return a;

 return a * Ust(a, n-1);
} /* bitti-Ust */
```

$$T(n) = \begin{cases} n \leq 1 \rightarrow 1 \text{ (Temel durum)} \\ N > 1 \rightarrow T(n-1) + 1 \end{cases}$$

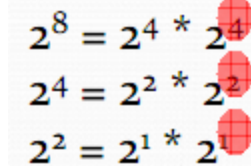
## Pow(x, n) 'nin Çalışma Zamanı

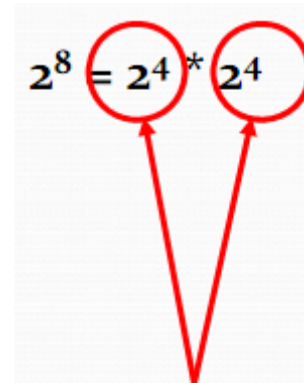
- Bu fonksiyon  $O(n)$  zamanında çalışır (n adet özyineli çağrı yapılır)
- Daha iyi bir çözüm var mı?
- Ara sonuçların karesini alarak daha etkin bir doğrusal özyineli algoritma yazabiliriz:

$$p(x, n) = \begin{cases} 1 & \text{if } x = 0 \\ x \cdot p(x, (n-1)/2)^2 & \text{if } x > 0 \text{ is odd} \\ p(x, n/2)^2 & \text{if } x > 0 \text{ is even} \end{cases}$$

## Power(2, 8)

- $2^8 = 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2$
- Ancak biz bu çözümü iki eşit parçaya bölerek ifade edebiliriz:
- $2^8 = 2^4 * 2^4$
- ve  $2^4 = 2^2 * 2^2$
- ve  $2^2 = 2^1 * 2^1$
- herhangi bir sayının 1. kuvveti kendisidir.
- Avantaj...
- İkisi de aynı olduğu için, her ikisini de hesaplamıyoruz! ve sadece 3 adet çarpma işlemi yapıyoruz.


$$\begin{aligned} 2^8 &= 2^4 * 2^4 \\ 2^4 &= 2^2 * 2^2 \\ 2^2 &= 2^1 * 2^1 \end{aligned}$$


$$2^8 = 2^4 * 2^4$$

## Kuvvet değeri tek sayı ise

- Tek olanları şu şekilde yaparız:
- $2^{\text{tek}} = 2 * 2^{(\text{tek}-1)}$
- Peki öyleyse,  $2^{21}$  'i hesaplayalım

- Görüldüğü gibi 20 defa çarpım yerine sadece 6 kere sonuca ulaşıyor

$$2^{21} = 2 * 2^{20} \text{ (Tek sayı hilesi)}$$

$$2^{20} = 2^{10} * 2^{10}$$

$$2^{10} = 2^5 * 2^5$$

$$2^5 = 2 * 2^4$$

$$2^4 = 2^2 * 2^2$$

$$2^2 = 2^1 * 2^1$$

$$2^{21} = 2 * 2^{20}$$

$$2^{20} = 2^{10} * 2^{10}$$

$$2^{10} = 2^5 * 2^5$$

$$2^5 = 2 * 2^4$$

$$2^4 = 2^2 * 2^2$$

$$2^2 = 2^1 * 2^1$$

# Özyinelemenin mantığı

- Eğer üst çift ise, iki parçaya ayırıp işleriz.
- Eğer üst tek ise, 1 çıkarır daha sonra kalan kısmı ikiye ayırıp işleriz. Ama üstten bir çıkardığımız için en sonunda 1 adet çarpma işlemi daha yapmayı unutmayın.
- İşlemi formülize etmeye başlayalım:
  - $e == 0$  ,  $\text{Pow}(x, e) = 1$
  - $e == 1$  ,  $\text{Pow}(x, e) = x$
  - $e$  çift ise ,  $\text{Pow}(x, e) = \text{Pow}(x, e/2) * \text{Pow}(x, e/2)$
  - $e > 1$  ve tek ise ,  $\text{Pow}(x, e) = x * \text{Pow}(x, e-1)$

# Power(x, n) 'nin Çalışma Zamanı

$$\begin{aligned}
 2^4 &= 2^{(4/2)^2} = (2^{4/2})^2 = (2^2)^2 = 4^2 = 16 \\
 2^5 &= 2^{1+(4/2)^2} = 2(2^{4/2})^2 = 2(2^2)^2 = 2(4^2) = 32 \\
 2^6 &= 2^{(6/2)^2} = (2^{6/2})^2 = (2^3)^2 = 8^2 = 64 \\
 2^7 &= 2^{1+(6/2)^2} = 2(2^{6/2})^2 = 2(2^3)^2 = 2(8^2) = 128.
 \end{aligned}$$

## Algorithm **Power**(x, n):

**Input:** x sayısı ve n tamsayısı,  $n \geq 0$

**Output:**  $x^n$  değeri

**if**  $n = 0$  **then**

**return** 1

**if** n is odd **then**

y = **Power**(x, (n - 1) / 2)

**return**  $x \cdot y \cdot y$

**else**

y = **Power**(x, n / 2)

**return**  $y \cdot y$

Her özyineli çağırmada n sayısını 2'ye bölüyoruz; dolayısıyla, log n özyineli çağrı yaparız. Bu metod  $O(\log n)$  zamanına sahiptir.

Burada ara sonucu y değişkeni ile göstermemiz önemli; şayet metod çağırma yazarsak metod 2 defa çağırılmış olur.

# Fibonacci Sayıları

- Fibonacci sayılarını tanımlayacak olursak:
- 1 1 2 3 5 8 13.....
  - $F(0) = 0$
  - $F(1) = 1$
  - $F(n) = F(n-1) + F(n-2)$

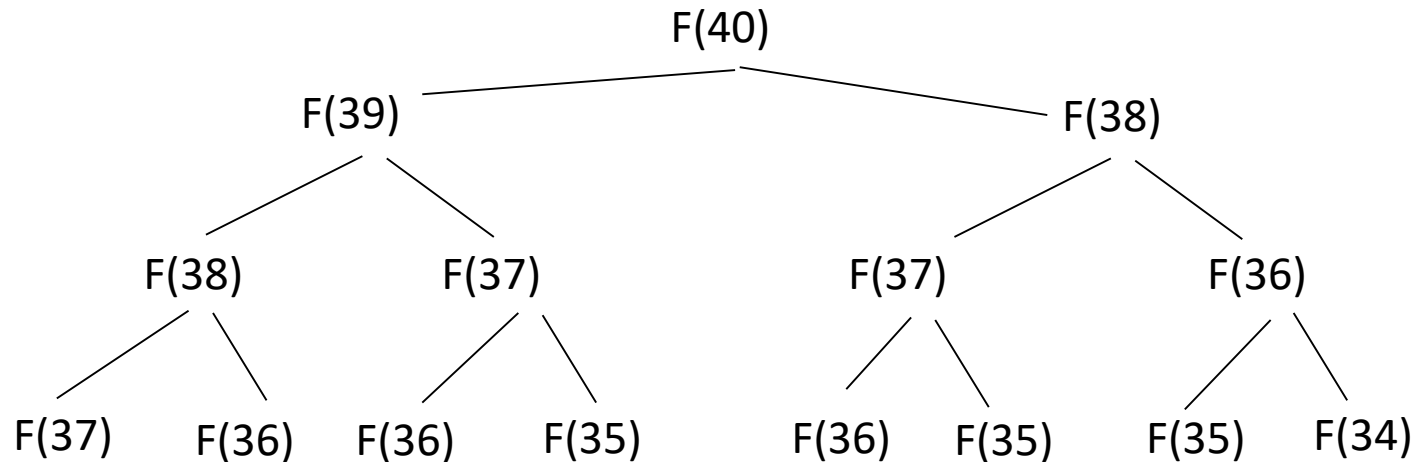
```
/* n. Fibonacci sayısını hesaplama*/
int Fibonacci(int n){
 /* Temel durum */
 if (n == 0) return 0;
 if (n == 1) return 1;

 return Fibonacci(n-1) + Fibonacci(n-2);
} /* bitti-Fibonacci */
```



# Fibonacci Sayıları

- Fibonacci sayılarının tanımı özyinelemelidir.
- Örneğin 40. fibonacci değerini bulmaya çalışalım.



- 
- F(40) için toplam kaç tane özyinelemeli çağrı yapılır.
  - **Cevap:** 300 000 000 den fazla yordam çağrılır.
  - Çalışma zamanı yaklaşık  $O(2^n)$  Bellek karmaşıklığı  $O(\text{derinlik} * n)$

# Fibonacci Sayıları

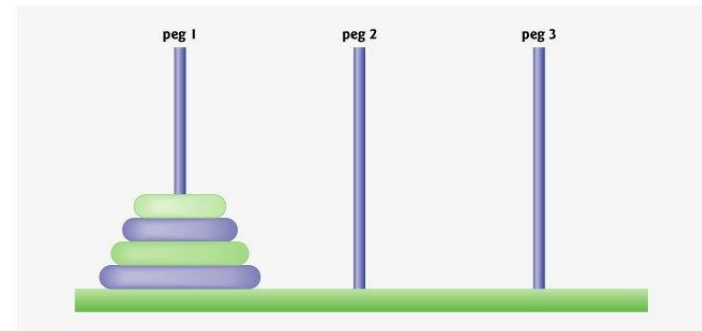
- Basit bir "for" ile çözülebilecek problemler için özyinelemeli algoritmalar kullanılmaz.

```
/* n. Fibonacci sayısını hesaplama*/
public static int fibonacci(int n){
 if(n == 1 || n == 2) return 1;

 int s1=1,s2=1,sonuc=0;
 for(int i=0; i<n; i++){
 sonuc = s1 + s2;
 s1 = s2;
 s2 = sonuc;
 }
 return sonuc;
}
```

# Hanoi Kuleleri

- **Verilen:** üç iğne
  - İlk iğnede en küçük disk en üstte olacak şekilde yerleştirilmiş farklı büyüklükte disk kümesi.
- **Amaç:** diskleri en soldan en sağa taşımak.
- **Şartlar:** aynı anda sadece tek disk taşınabilir.
- Bir disk boş bir iğneye veya daha büyük bir diskin üzerine taşınabilir.



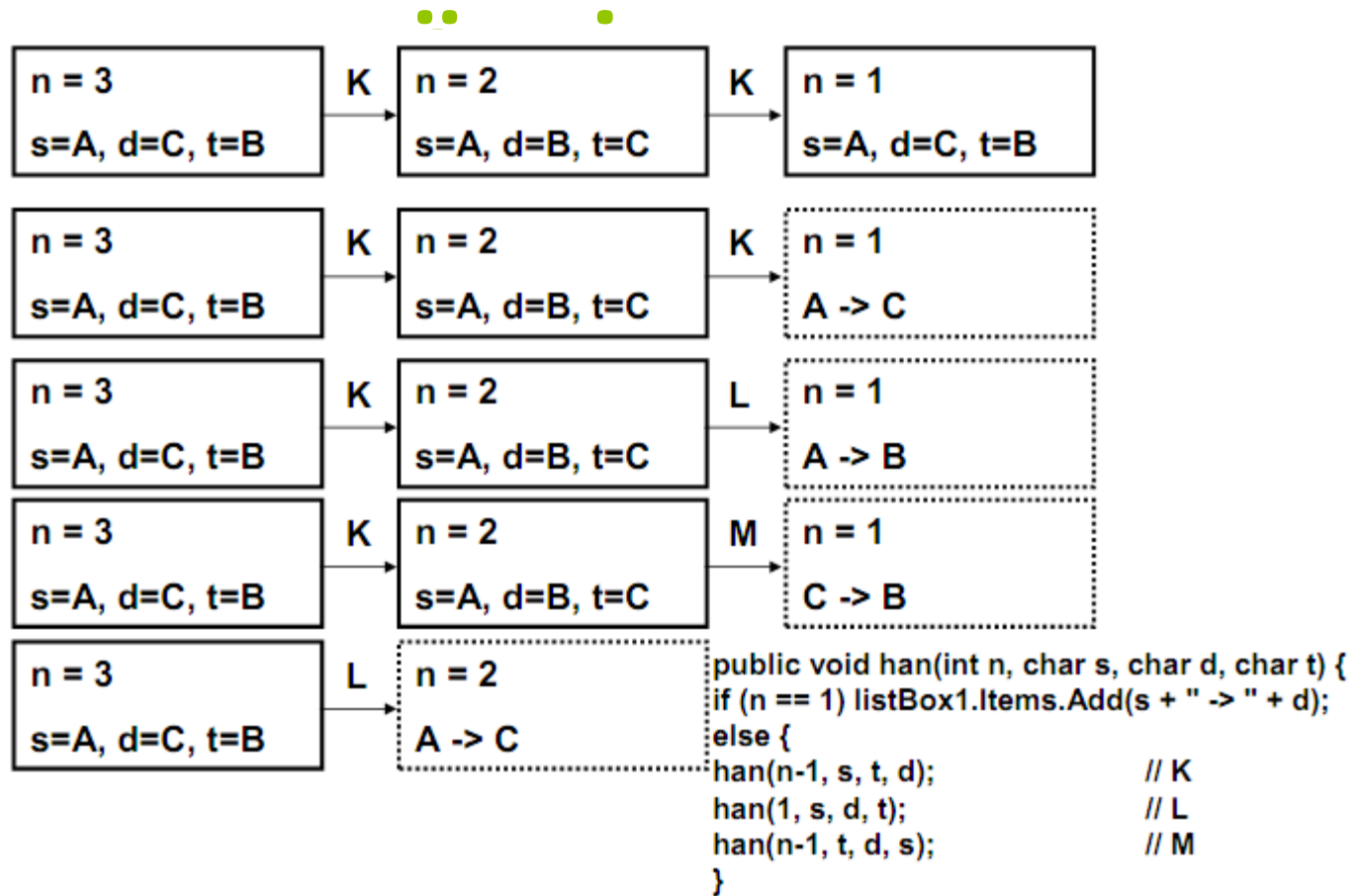
# Hanoi Kuleleri– Özyinelemeli Çözüm-Java

```

○ package hanoikuleleri;
○ import java.util.*;
○ public class Hanoikuleleri {

○ public static void main(String[] args)
○ {
○ System.out.print("n değerini giriniz : ");
○ Scanner klavye = new Scanner(System.in); int n = klavye.nextInt();
○ tasi(n, 'A', 'B', 'C');
○ }
○ public static void tasi(int n, char A, char B, char C)
○ {if(n==1) System.out.println(A + " --> " + B);
○ else
○ {
○ tasi(n-1, A, C, B); tasi(1, A, B, C); tasi(n-1, C, B, A); }
○ return;
○ }

```



# Kuyruk Özyineleme (Tail Recursion)

- Özyineleme metodunda, özyineli çağrı son adım ise bu durum oluşur. Yineleme çağrısı metodun en sonunda yapılır.
- `int recFact(int n)`
- `{`
- `if (n<=1) return 1;`
- `else return n * recFact(n-1);`
- `}`
- `void tail() {`
- `.....`
- `.....`
- `tail(); }`

## Kuyruk Olmayan Özyineleme (nonTail Recursion)

- Özyineleme metodunda, özyineli çağrı son adım değil ise bu durum oluşur. Yineleme çağrısından sonra başka işlemler yapılır (yazdırma v.b.) veya ikili özyineleme olabilir.
- Taban durumu haricinde iki özyinelemeli çağrı yapılıyorsa, ikili özyineleme oluşur.

```
int nontail(int n)
{
 if (n > 0) {

 nontail(n-1);
 printf(n);

 nontail(n-1); }
}
```

# Dolaylı Özyineleme (Indirect Recursion)

- Yineleme çağrısı başka bir fonksiyonun içinden yapılır.

- void A(int n)**

- {**

- if (n <= 0) return 1;

- n- -;

- B(n);

- }**

- void B(int n)**

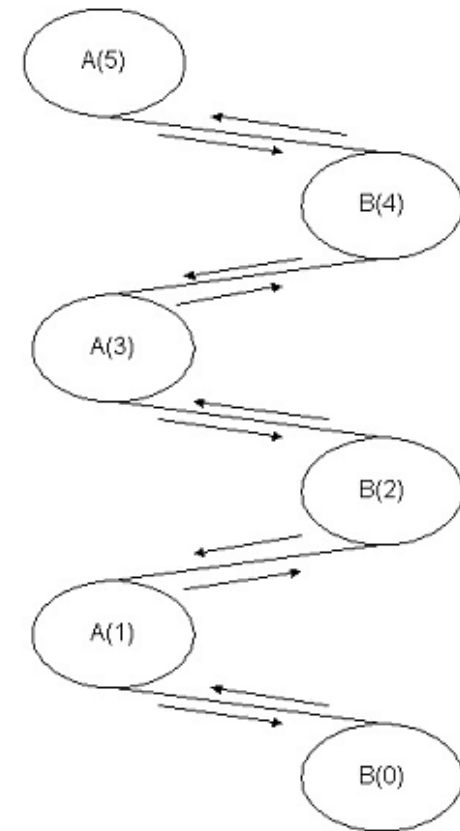
- {**

- if (n <= 0) return 1;

- n- -;

- A(n);

- }**





## İç içe Özyineleme (Nested Recursion)

- Yineleme çağrısı içindende yineleme çağrısı yapılır.
- `int A(int n, int m)`
- `{`
  - `if (n <= 0) return 1;`
  - `return A(n-1, A(n-1, m-1));`
- `}`

## Özyineleme mi İterasyon mu?

- Eğer iteratif çözümü tercih ederseniz algoritmanızın karmaşıklığı  **$O(n)$**  olur.
- Gerçek dünyada bu yöntemi kullanırsanız kovulursunuz.
- Eğer dinamik programlama tasarım yöntemi ile özyineleme kullanırsanız, algoritmanızın karmaşıklığı  $O(\log_2 n)$  olur.
- Bunun için terfi bile alabilirsiniz.

# ÖZET

- Özyineleme bütün doğada mevcuttur.
- Formül ezberlemekten daha kolaydır. Her problemin bir formülü olmayabilir, ancak her problem, küçük, tekrarlayan adımlar serisi olarak ifade edilebilir.
- Özyinelemeli bir işlemde her adım küçük ve hesaplanabilir olmalıdır.
- Kesinlikle sonlandırıcı bir şarta sahip olmalı.
- Özyineleme en çok, özyinelemeli olarak tanımlanmış binary tree dediğimiz veri yapılarında kullanılmak üzere yazılan algoritmalar için faydalıdır. Normal iterasyona göre çok daha kolaydır.
- Özyineleme, böl/yönet (divide & conquer) ve dinamik program türündeki algoritmalar için mükemmeldir.

# Ödev

- 1- n adet x değeri için standart sapmayı ( $\sigma$ ) bulan programı iterasyon ve recursive ile yapınız.

- $x_m = (1/n) \sum_k x_k$

$$\sigma = \sqrt{V}$$

- ( $\sigma$ ) = standart sapma

$$V = (1 / (n - 1)) \sum_k (x_k - x_m)^2$$

- V = varyans değeri

- $x_m$  = mean değeri

# Ödev

- 2- Ackerman fonksiyonu aşağıdaki şekilde tanımlanmıştır. Bu fonksiyonu öz yinelemeli olarak gerçekleştiriniz.

$$A(m,n) = \begin{cases} n + 1 & , m = 1 \\ A(m - 1, 1) & , m > 0 \text{ ve } n = 0 \\ A(m - 1, A(m, n - 1)) & , m > 0 \text{ ve } n > 0 \end{cases}$$



# Hashing

# HASHING (Kırpma veya Çırpma )

- Hashing, elimizdeki veriyi kullanarak o veriden elden geldiği kadar benzersiz bir tamsayı elde etme işlemidir.
- Bu elde edilen tamsayı, dizi şeklinde tutulan verilerin indisi gibi kullanılarak verilere tek seferde erişmemizi sağlar.
- Hasing konusunu alt başlıklara ayıracak olursak;
  - **Hashing Fonksiyonu**
  - **Hash Tablosu**
  - **Çakışma (collision)**

# HASHING (KIRPMA)

## ○ Hash Fonksiyonları

- Selecting Digits- Rakam seçme
- Folding (shift folding, boundary folding)-Katlama
- Division-Bölme
- Mid-Square-Orta-Kare
- Extraction-Çıkarım
- Radix Transformation- Taban Dönüşüm

## ○ Çakışma (Collision) ve çözümler

- Linear Probing- Doğrusal doldurma
- Double Hashing- Çift Kırpma
- Quadratic Probing-Kuadratik Doldurma
- Chaining-Zincirleme



# HASHING

- Arama metotlarında temel işlem anahtarları karşılaştırmaktır. Bir anahtarın tablo içerisinde bulunduğu pozisyona ulaşınca kadar arama işlemine devam edilir.
- Hash fonksiyonuyla aranan anahtar elemana doğrudan erişilebilmektedir. Hash fonksiyonu bir anahtar bilgisinin tabloda bulunduğu indeksi hesaplamaktadır.
- **Open hashing (Açık kırpma)**: Potansiyel olarak limitsiz alan kullanır.
- **Closed hashing (Kapalı kırpma)**: Bilgi kaydı için sabit alan kullanır.

# HASHING (Kırpma veya Çırpma )

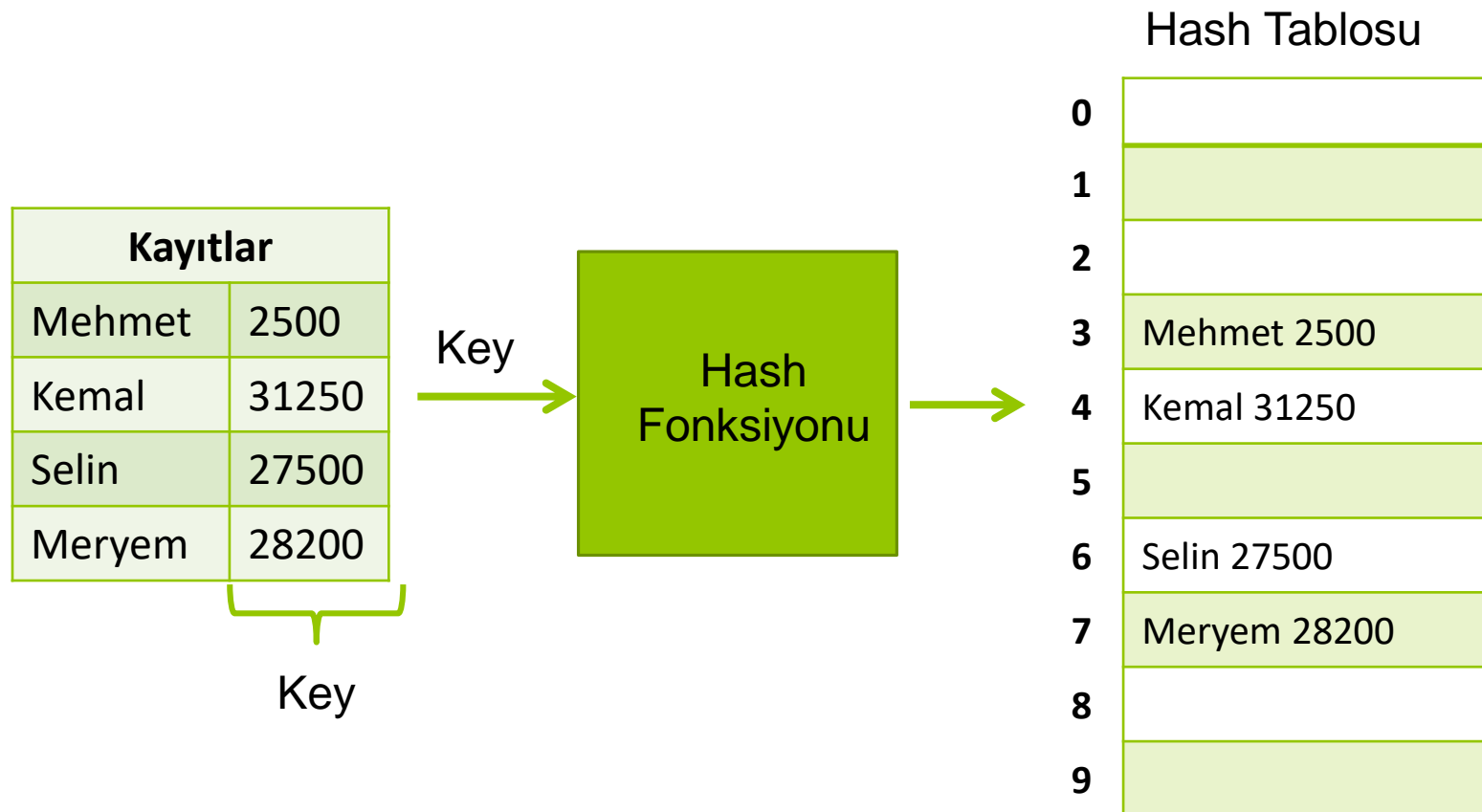
- Boyutu N olan bir tabloda, hash fonksiyonu ( $h(x)$ ) bir x anahtarını 0 ile N-1 arasında bir değerle eşleştirir.
- Örnek:
- N=15 olan bir tablo için  $h(x) = x \% 15$  (% - > modlu bölüm) olarak belirlenebilir. Eğer,

|        |    |     |    |      |    |    |
|--------|----|-----|----|------|----|----|
| x      | 25 | 129 | 35 | 2501 | 47 | 36 |
| $h(x)$ | 10 | 9   | 5  | 11   | 2  | 6  |

- Anahtarların tablo içerisindeki yerleri ise aşağıdaki gibidir:

| 0 | 1 | 2  | 3 | 4 | 5  | 6  | 7 | 8 | 9   | 10 | 11   | 12 | 13 | 14 |
|---|---|----|---|---|----|----|---|---|-----|----|------|----|----|----|
| - | - | 47 | - | - | 35 | 36 | - | - | 129 | 25 | 2501 | -  | -  | -  |

# HASHING (Kırpma veya Çırpma )



# Hash fonksiyonları

- Hash fonksiyonları, bir anahtarın tabloda bulunduğu indeks sırasını verir.
- **Perfect hash fonksiyonu:**
- Her bir anahtara sadece bir pozisyonu eşleştiren fonksiyona denir.
- **Simple perfect hash fonksiyonu:**
- Tablo boyutu ile toplam anahtar sayısı eşit olduğunda (tabloda boş yer yoksa) her bir anahtara sadece bir pozisyonu eşleştiren fonksiyona denir.
- **İyi bir hash fonksiyonu:**
  - kolay ve hızlı hesaplanabilir olmalıdır.
  - tablodaki her bir pozisyon için sadece bir anahtar atamalıdır.

# Hash fonksiyonları

- Hash fonksiyonları integer sayılarla işlem yaparlar.
- Integer olmayan anahtarlarda integer değere dönüştürme işlemi yapılır.
- Örneğin kişilere ait sağlık numarası 9635-8904 şeklinde ise aradaki tire işareti kaldırılarak 96358904 olarak alınır.
- Eğer anahtar karakterlerden oluşuyorsa karakterlerin ASCII kodları kullanılır.

## Hash fonksiyonları (Selecting Digits – Rakam Seçme)

- Anahtar üzerindeki belirlenmiş bazı haneleri seçip birleştirerek tablodaki pozisyon bulunur.
- 2. ve 5. hanelerin seçimiyle oluşturulan değer aşağıdaki gibidir.
  - $h(033475678) = 37$
  - $h(023455678) = 25$
- **Artıları ve Eksileri**
  - Yapısı basittir.
  - Anahtarları tablonun tamamına düzgün bir şekilde dağıtamaz.
  - Çakışma çok sık olabilir.

## Hash fonksiyonları (Folding-katlama)

- Anahtar birkaç parçaya bölünür ve bu parçalar kendi arasında toplanarak tablodaki pozisyon bulunur.
- **Shift folding** metodunda anahtarın her bir parçası değiştirilmeden tablo boyutuna göre mod ile toplanır.
  - Örnek: SSN = 123-45-6789 olarak verilsin. SSN numarası 123, 456, 789 olarak üç parçaya ayrılıp toplandığında  $123+456+789 = 1368$  olarak pozisyon numarası elde edilir.
- **Boundary folding** metodunda anahtarın parçalarının sırası değiştirilerek tablo boyutuna göre mod ile toplanır.
  - Örnek: SSN numarası 123, 456, 789 olarak üç parçaya ayrılır. Birinci parça aynı sırada kalır ve ikinci parça ters sıralanır. Daha sonra üçüncü parça aynı sırada alınır ve tablo boyutuna göre mod ile toplanır. ( $123+654+789 = 1566$ )

## Hash fonksiyonları (Division – Bölme)

- Anahtar değeri tablo boyutuna göre mod ile bölünür.
- Örnek: SSN = 123456789 olarak verilsin. Tablo boyutu 1000 olursa, hash fonksyonu sonucu aşağıdaki gibi bulur;
- $\text{hash}(h) = 123456789 \% 1000 = 789$
- Yapısı basittir ancak çakışma olur.



## Hash fonksiyonları (Mid-square, Orta kare)

- Anahtar değerin karesi alınır ve sonucun orta kısmı seçilerek tablodaki pozisyon değeri bulunur.
- Örnek: anahtar = 3121 olarak verilsin. Hash fonksyonu sonucu aşağıdaki gibi bulur;
- $3121^2 = 9740641$
- $\text{hash}(3121) = 406$
- Anahtarın karesi binary olarak gösterilebilir.
- $3121^2 = 100101001010000101100001$
- $\text{hash}(3121) = 0101000010 = 322$

## Hash fonksiyonları (Extraction)

- Anahtar değerinin sadece bazı kısımları seçilerek tablodaki pozisyon değeri bulunur.
- Örnek: anahtar = 123-45-6789 olarak verilsin. Hash fonksiyonu aşağıdakilerden herhangi birisi olabilir;
- $\text{hash}(123-45-6789) = 123456789 = 1234$
- $\text{hash}(123-45-6789) = 123456789 = 6789$
- $\text{hash}(123-45-6789) = 123456789 = 1289$

## Hash fonksiyonları (Radix Transformation)

- Anahtar değeri başka bir sayı tabanına dönüştürülür.
- Örnek: anahtar = 1238 olarak verilsin. Tablo boyutu 1000 olarak alındığında,
- $1238_{10} = 2326_8$
- Hesaplanan değer tablo boyutuna mod ile bölünerek pozisyon değeri bulunur.
- $\text{Hash}(1238) = 2326 \% 1000 = 326$

## Hash fonksiyonları -Çakışma

- $x = 65$  değerini aşağıdaki tabloya ekleyim.
- $x = 65$
- $h(x) = 65 \bmod 15 = 5$
- Aynı pozisyona birden fazla kayıt gelirse çakışma meydana gelir.

|    |      |
|----|------|
| 0  | -    |
| 1  | -    |
| 2  | 47   |
| 3  | -    |
| 4  | -    |
| 5  | 35   |
| 6  | 36   |
| 7  | -    |
| 8  | -    |
| 9  | 129  |
| 10 | 25   |
| 11 | 2501 |
| 12 | -    |
| 13 | -    |
| 14 | -    |

65

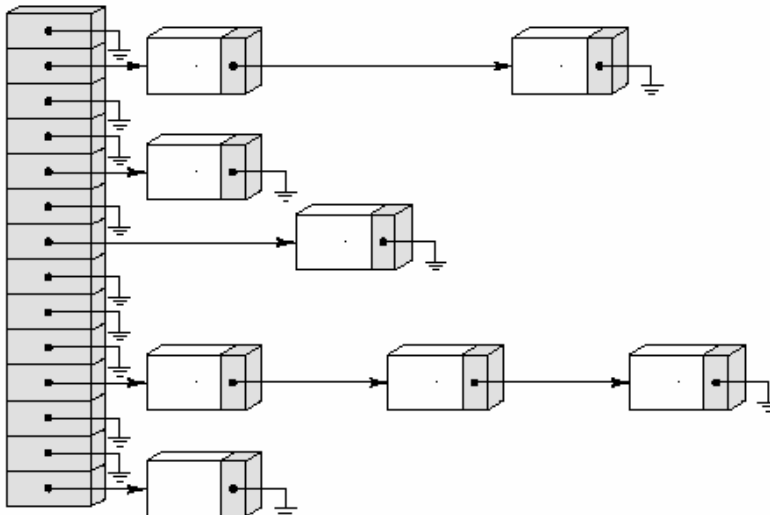
?

←

## Hash fonksiyonları

### Çakışmanın giderilmesi (Chaining)

- Aynı pozisyona gelen kayıtlar bağlı listelerle gösterilir.
- Ekleme: Listenin başına eklenir
- Silme/Erişim: Uygun listede arama yapılır



|    |      |
|----|------|
| 0  | -    |
| 1  | -    |
| 2  | 47   |
| 3  | -    |
| 4  | -    |
| 5  | 65   |
| 6  | 36   |
| 7  | -    |
| 8  | -    |
| 9  | 129  |
| 10 | 25   |
| 11 | 2501 |
| 12 | -    |
| 13 | -    |
| 14 | -    |

35

## Hash fonksiyonları

### Çakışmanın giderilmesi (Chaining)

- Örnek:
- 29, 16, 14, 99, 127 sayılarının eklenmesi
- Aynı pozisyona gelen diğer anahtarlar bağlı listenin başına eklenmektedir.
- Chaining metodunun dezavantajları**
- Tablonun bazı kısımları hiç kullanılmamaktadır.
- Bağlı listeler uzadıkça arama ve silme işlemleri için gereken zaman uzamaktadır.

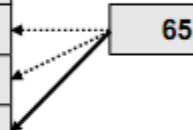
|    |      |       |
|----|------|-------|
| 0  | -    |       |
| 1  | 16   |       |
| 2  | 47   |       |
| 3  | -    |       |
| 4  | -    |       |
| 5  | 65   | → 35  |
| 6  | 36   |       |
| 7  | 127  |       |
| 8  | -    |       |
| 9  | 99   | → 129 |
| 10 | 25   |       |
| 11 | 2501 |       |
| 12 | -    |       |
| 13 | -    |       |
| 14 | 14   | → 29  |

## Hash fonksiyonları

### Çakışmanın giderilmesi (Linear Probing)

- Aynı pozisyona gelen ikinci kayıt ilgili pozisyondan sonraki ilk boş pozisyona yerleştirilir.
- Ekleme: Boş bir alan bulunarak yapılır.
- Silme/Erişim: İlk boş alan bulunana kadar devam edebilir.
- $h(k) = x \% m$
- $h(k,i) = (h(k) + i) \% m$
- x: anahtar, m: tablo boyutu

|    |      |
|----|------|
| 0  | -    |
| 1  | -    |
| 2  | 47   |
| 3  | -    |
| 4  | -    |
| 5  | 35   |
| 6  | 36   |
| 7  | 65   |
| 8  | -    |
| 9  | 129  |
| 10 | 25   |
| 11 | 2501 |
| 12 | -    |
| 13 | -    |
| 14 | -    |



## Hash fonksiyonları

### Çakışmanın giderilmesi (Linear Probing)

- Örnek :  $h(x) = x \bmod 13$
- 18, 41, 22, 44, 59, 32, 31, 73 değerlerini verilen sırada giriniz.

|   |   |   |   |   |   |   |   |   |   |    |    |    |
|---|---|---|---|---|---|---|---|---|---|----|----|----|
|   |   |   |   |   |   |   |   |   |   |    |    |    |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |



|   |   |    |   |   |    |    |    |    |    |    |    |    |
|---|---|----|---|---|----|----|----|----|----|----|----|----|
|   |   | 41 |   |   | 18 | 44 | 59 | 32 | 22 | 31 | 73 |    |
| 0 | 1 | 2  | 3 | 4 | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 |



## Hash fonksiyonları

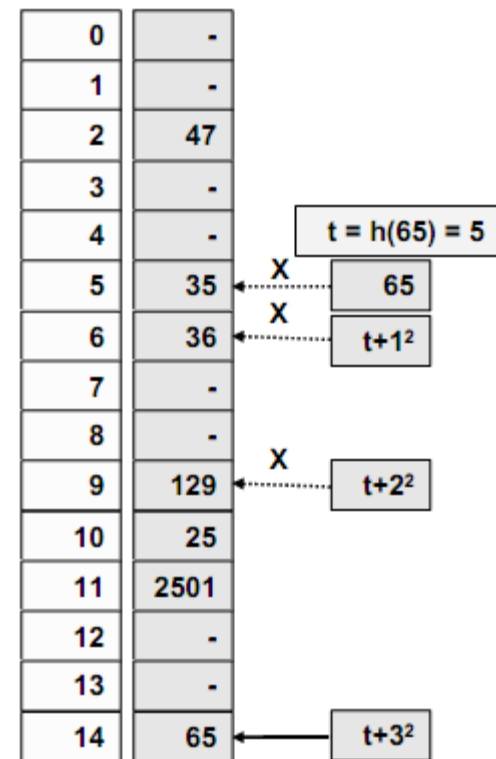
### Çakışmanın giderilmesi (Linear Probing)

- **Linear Probing metodunun avantajları / dezavantajları**
- Bağlı listeler gibi ayrı bir veri yapısına ihtiyaç duyulmaz.
- Kayıtların yığın şeklinde toplanmasına sebep olur.
- Silme ve arama işlemleri için gereken zaman aynı hash değeri sayısı arttıkça artar.

## Hash fonksiyonları

### Çakışmanın giderilmesi (Quadratic Probing)

- Aynı pozisyona gelen ikinci kayıt Quadratic Fonksiyonla yerleştirilir.
- En çok kullanılan fonksiyon
- ***x:anahtar, m:tablo boyutu***
- ***$h(x)=x \% m$***
- ***$h(x,i) = (h(k)'+i^2)\%m$***
- Yeni pozisyon için sırasıyla  $(t+1^2)$ ,  $(t+2^2)$ , ...,  $(t+n^2)$  değerlerine karşılık gelen pozisyonlara bakılır ve ilk boş olana yerleştirilir.



## Hash fonksiyonları

### Çakışmanın giderilmesi (Quadratic Probing)

- Örnek: 29, 16, 14, 99, 127 değerlerini hash tablosuna quadratic probing metoduyla sırayla yerleştiriniz.
- $h(x) = x \bmod m$
- $h(29) = 29 \% 15 = 14$
- $h(29,1) = (h(29)' + 1) \% 15 = (14 + 1) \% 15 = 0$

|    |      |           |
|----|------|-----------|
| 0  | 29   | ← $t+1^2$ |
| 1  | -    |           |
| 2  | 47   |           |
| 3  | -    |           |
| 4  | -    |           |
| 5  | 35   |           |
| 6  | 36   |           |
| 7  | -    |           |
| 8  | -    |           |
| 9  | 129  |           |
| 10 | 25   |           |
| 11 | 2501 |           |
| 12 | -    |           |
| 13 | -    |           |
| 14 | 65   | ← $t$     |

$t = h(29) = 14$

X

## Hash fonksiyonları

### Çakışmanın giderilmesi (Quadratic Probing)

- Örnek: 29, 16, 14, 99, 127 değerlerini hash tablosuna quadratic probing metoduyla sırayla yerleştiriniz.
- $h(16) = 16 \% 15 = 1$

|    |      |                 |
|----|------|-----------------|
| 0  | 29   | $t = h(16) = 1$ |
| 1  | 16   | $t$             |
| 2  | 47   |                 |
| 3  | -    |                 |
| 4  | -    |                 |
| 5  | 35   |                 |
| 6  | 36   |                 |
| 7  | -    |                 |
| 8  | -    |                 |
| 9  | 129  |                 |
| 10 | 25   |                 |
| 11 | 2501 |                 |
| 12 | -    |                 |
| 13 | -    |                 |
| 14 | 65   |                 |

## Hash fonksiyonları

### Çakışmanın giderilmesi (Quadratic Probing)

- Örnek: 29, 16, 14, 99, 127 değerlerini hash tablosuna quadratic probing metoduyla sırayla yerleştiriniz.
- $h(14) = 14 \% 15 = 14$
- $h(14,1) = (14+1)\%15 = 0$
- $h(14,2) = (14+2^2)\%15 = 18\%15 = 3$

|    |      |                                        |
|----|------|----------------------------------------|
| 0  | 29   | $\leftarrow$ X $t+1^2$                 |
| 1  | 16   |                                        |
| 2  | 47   |                                        |
| 3  | 14   | $\leftarrow$ $t+2^2$                   |
| 4  | -    |                                        |
| 5  | 35   |                                        |
| 6  | 36   |                                        |
| 7  | -    |                                        |
| 8  | -    |                                        |
| 9  | 129  |                                        |
| 10 | 25   |                                        |
| 11 | 2501 |                                        |
| 12 | -    |                                        |
| 13 | -    |                                        |
| 14 | 65   | $\leftarrow$ X $t = h(14) = 14$<br>$t$ |

## Hash fonksiyonları

### Çakışmanın giderilmesi (Quadratic Probing)

- Örnek: 29, 16, 14, 99, 127 değerlerini hash tablosuna quadratic probing metoduyla sırayla yerleştiriniz.
- $h(99) = 99 \% 15 = 9$
- $h(99,1) = (9+1)\%15 = 0$
- $h(99,2) = (9+2^2)\%15 = 13\%15 = 13$

|    |      |
|----|------|
| 0  | 29   |
| 1  | 16   |
| 2  | 47   |
| 3  | 14   |
| 4  | -    |
| 5  | 35   |
| 6  | 36   |
| 7  | -    |
| 8  | -    |
| 9  | 129  |
| 10 | 25   |
| 11 | 2501 |
| 12 | -    |
| 13 | 99   |
| 14 | 65   |

$t = h(99) = 9$

t

X

t

t+1<sup>2</sup>

X

t+1<sup>2</sup>

t+2<sup>2</sup>

X

t+2<sup>2</sup>

## Hash fonksiyonları

### Çakışmanın giderilmesi (Quadratic Probing)

- Örnek: 29, 16, 14, 99, 127 değerlerini hash tablosuna quadratic probing metoduyla sırayla yerleştiriniz.
- $h(127) = 127 \% 15 = 7$

|    |      |
|----|------|
| 0  | 29   |
| 1  | 16   |
| 2  | 47   |
| 3  | 14   |
| 4  | -    |
| 5  | 35   |
| 6  | 36   |
| 7  | 127  |
| 8  | -    |
| 9  | 129  |
| 10 | 25   |
| 11 | 2501 |
| 12 | -    |
| 13 | 99   |
| 14 | 65   |

$t = h(127) = 7$

$t + 2^2$

## Hash fonksiyonları

### Çakışmanın giderilmesi (Quadratic Probing)

- Quadratic Probing metodunun avantajları / dezavantajları
- Anahtar değerlerini linear probing metoduna göre daha düzgün dağıtır.
- Yeni eleman eklemede tablo boyutuna dikkat edilmezse sonsuza kadar çalışma riski vardır.
- Örn.: Boyutu 16 (0-15) olan bir tabloda 0, 1, 4 ve 9 pozisyanlarının dolu olduğu durumda 16 değerini eklemeye çalıştığımız zaman sonsuz döngüye girer.



## Hash fonksiyonları

### Çakışmanın giderilmesi (Double Hashing)

- Aynı pozisyona gelen ikinci kayıt için ikinci bir hash fonksiyonu kullanılır.
- İkinci hash fonksiyonu 0 değerini alamaz.
- En çok kullanılan fonksiyon:
- $\text{hash}(x) = (\text{hash}_1(x) + i * \text{hash}_2(x)) \% m$
- $\text{hash}_2(x) = R - (x \% R)$ ,  $R < m$ ,  $R$ : asal (veya  $m$  ile aralarında asal)
- $\text{hash}(x) = \text{hash}_1(x)$
- $\text{hash}(x) = (\text{hash}_1(x) + 1 * \text{hash}_2(x)) \% m$
- $\text{hash}(x) = (\text{hash}_1(x) + 2 * \text{hash}_2(x)) \% m$
- $\text{hash}(x) = (\text{hash}_1(x) + 3 * \text{hash}_2(x)) \% m$

## Hash fonksiyonları

### Çakışmanın giderilmesi (Double Hashing)

- Örnek: 65 değerinin eklenmesi
- $\text{hash}_1(x) = x \% 15$
- $\text{hash}_2(x) = 11 - (x \% 11)$
- $\text{hash}(65) = \text{hash}_1(65)$
- $\text{hash}(65, 0) = 5$  dolu
- $\text{hash}(65, 1) = (\text{hash}_1(5) + 1 * \text{hash}_2(65)) \% 15$
- $\text{hash}(65, 1) = (5 + (11 - 10)) \% 15 = 6$
- $\text{hash}(65, 2) = (\text{hash}_1(5) + 2 * \text{hash}_2(65)) \% 15$
- $\text{hash}(65, 2) = (5 + 2) \% 15 = 7$

|    |      |
|----|------|
| 0  | -    |
| 1  | -    |
| 2  | 47   |
| 3  | -    |
| 4  | -    |
| 5  | 35   |
| 6  | 36   |
| 7  | 65   |
| 8  | -    |
| 9  | 129  |
| 10 | 25   |
| 11 | 2501 |
| 12 | -    |
| 13 | -    |
| 14 | -    |

$t = h_1(65) = 5$   

65

  
 $t+1 * h_2(65)$   
 $t+2 * h_2(65)$

X  
X

## Hash fonksiyonları

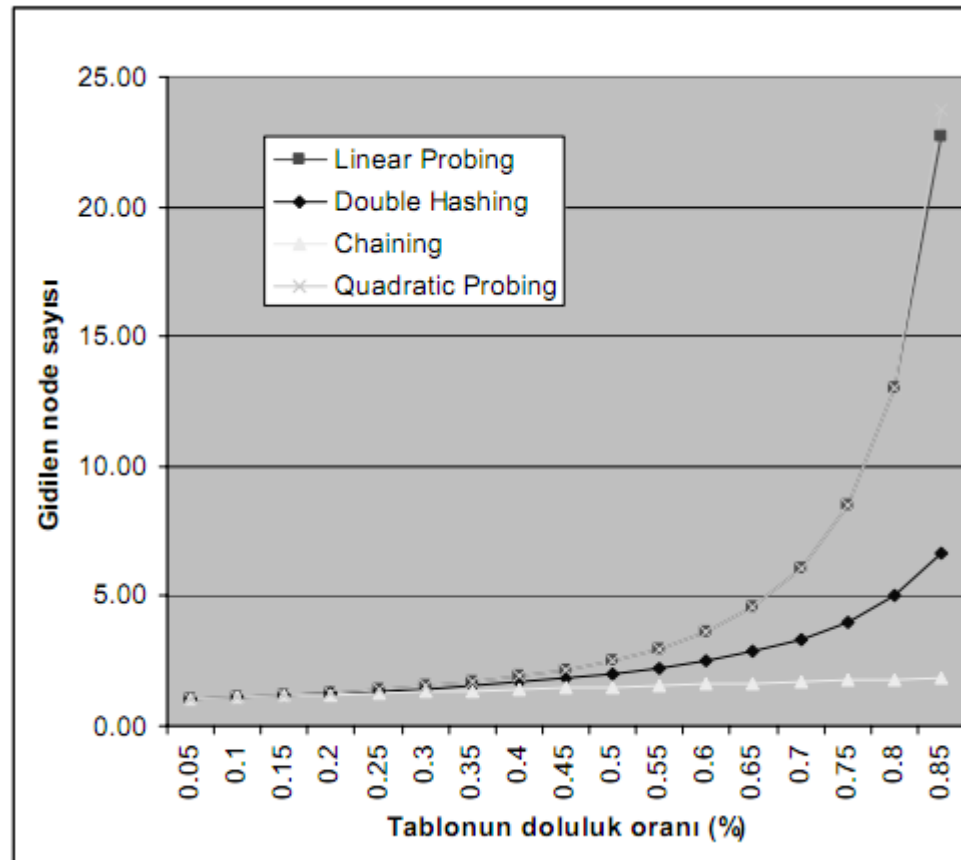
### Çakışmanın giderilmesi (Double Hashing)

- **Double Hashing metodunun avantajları / dezavantajları**
- Anahtar değerlerini linear probing metoduna göre daha düzgün dağıtır ve gruplar oluşmaz.
- Quadratic probing metoduna göre daha yavaştır çünkü ikinci bir hash fonksiyonu hesaplanır.

# Hash fonksiyonları

## Performans

yapılacak  
sondalama sayısı:  
 $1/(1-\alpha)$   
 $\alpha$ : tablo dolum oranı  
 $\alpha = n/m$   
 $n$ : girilen eleman  
sayısı  
 $m$ : tablo boyutu



## Ödev

- 100 tane anahtar değerini rastgele üreten ve bu değerleri boyutu 100 olan bir hash tablosuna yerleştiren programı yazınız.
- Programda hash fonksiyonu olarak division metodunu, çakışma çözümü için chaining, linear probing ile quadratic probing metodlarını ayrı ayrı kullanınız.
- Anahtar değerlerini integer olarak alınız.
- Hash tablosu için dizi yapısını kullanınız.

## Chaining java kd

- public class LinkedHashMap
- { private int key;
- private int value;
- private LinkedHashMap next;
- LinkedHashMap (int key, int value)
- { this.key = key;
- this.value = value;
- this.next = null;
- }
- public int getValue() { return value; }
- public void setValue(int value) { this.value = value; }
- public int getKey() { return key; }
- public LinkedHashMap getNext() { return next; }
- public void setNext(LinkedHashMap next) { this.next = next; }

## Chaining java kd

```

○ public class HashMap
○ { private static int TABLE_SIZE = 128;
○ LinkedHashMap[] table;
○ HashMap()
○ { table = new LinkedHashMap[TABLE_SIZE];
○ for (int i = 0; i < TABLE_SIZE; i++) table[i] = null;
○ }
○ public int get(int key) {
○ int hash = (key % TABLE_SIZE);
○ if (table[hash] == null) return -1;
○ else { LinkedHashMap entry = table[hash];
○ while (entry != null && entry.getKey() != key) entry = entry.getNext();
○ if (entry == null) return -1;
○ else return entry.getValue();
○ }
○ }
○ }

```

## Chaining java kd

- public void put(int key, int value) {
- int hash = (key % TABLE\_SIZE);
- if (table[hash] == null) table[hash] = new LinkedHashMapEntry(key, value);
- else { LinkedHashMapEntry entry = table[hash];
- while (entry.getNext() != null && entry.getKey() != key)
- entry = entry.getNext();
- if (entry.getKey() == key) entry.setValue(value);
- else entry.setNext(new LinkedHashMapEntry(key, value));
- }
- }
-



## Chaining java kd

```

○ public void remove(int key)
○ { int hash = (key % TABLE_SIZE);
○ if (table[hash] != null) {
○ LinkedHashMapEntry prevEntry = null;
○ LinkedHashMapEntry entry = table[hash];
○ while (entry.getNext() != null && entry.getKey() != key)
○ { prevEntry = entry; entry = entry.getNext(); }
○ if (entry.getKey() == key) {
○ if (prevEntry == null) table[hash] = entry.getNext();
○ else prevEntry.setNext(entry.getNext());
○ }
○ }
○ }
○ }
○ }

```