Bölüm 14: Hash Tabloları

C ile Veri Yapıları ve Algoritmalar

Doğrudan adresleme | Hash fonksiyonları | Çakışma | Doğrusal / karesel yoklama | Çift hash | Zincirleme | Yük faktörü | Silindi işareti

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

GİRİŞ

Öğrenme Çıktıları

Fikri açıklamak

Anahtardan konumu hesaplayarak ortalama O(1) arama yapan hash tablosunu tanımlamak.

Fonksiyon seçmek

Bölme, çarpma, katlama ve dizgi hash'ini hesaplamak; iyi fonksiyonun özelliklerini saymak.

Çakışmayı çözmek

Doğrusal/karesel yoklama, çift hash ve zincirlemeyi adım adım uygulamak.

C ile yazmak

Ekleme, arama ve silme yordamlarını `malloc` / `free` ve boş durumlarıyla yazmak.

Silmeyi doğru yapmak

Açık adreslemede neden silindi işareti (tombstone) gerektiğini göstermek.

Analiz etmek

Yük faktörünün maliyete etkisini ve yeniden boyutlandırmanın amortize maliyetini açıklamak.

HASH FİKRİ

Arama maliyetini sabite indirmek

Yapı	Ara	Ekle	Sil	Not
Sırasız dizi	O(n)	O(1)	O(n)	sona eklenir, baştan taranır
Sıralı dizi	O(lg n)	O(n)	O(n)	ikili arama + kaydırma
Dengeli arama ağacı	O(lg n)	O(lg n)	O(lg n)	sıralı gezinme de verir
Hash tablosu	O(1)*	O(1)*	O(1)*	* ortalama; en kötü O(n)

Hash tablosu karşılaştırma yapmaz: anahtardan **konumu hesaplar**. Bedeli: sıralı gezinme, en küçük/en büyük ve aralık sorguları desteklenmez; en kötü durum O(n)'dir.

HASH FİKRİ

Önce en basiti: doğrudan adresleme tablosu

Anahtar evreni U = {0, …, 9}, kullanılan K = {2, 3, 5, 8}



T[k] doğrudan k'ye bakar: ara, ekle, sil O(1) — ama tablo |U| kadar büyük olmalı.

```
typedef struct { int var; char ad[16]; } kayit;
kayit T[10];                /* anahtar = indis */

kayit *da_ara(int k) {
  return T[k].var ? &T[k] : NULL;
}
```

Sorun

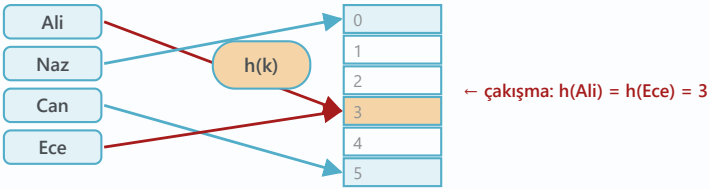
Öğrenci numarası gibi 9 haneli anahtarlarda $|U| = 10^9$: tablo sığmaz, çoğu boş kalır.

Çözüm

Anahtarı küçük bir aralığa eşleyen bir **hash fonksiyonu** h: U → {0, …, m−1}.

HASH FİKRİ

Anahtar → hash → kova



Terim	Anlamı
kova / yuva (bucket, slot)	tablonun bir konumu, 0 … m−1
ev adresi	h(k): anahtarın ilk bakılan yeri
çakışma (collision)	$k_1 \neq k_2$ ama $h(k_1) = h(k_2)$
eş anlamlılar (synonyms)	aynı ev adresine düşen anahtarlar
yük faktörü α	n / m (dolu oranı)

m < |U| olduğundan (güvercin yuvası ilkesi) çakışma **kaçınılmazdır**. İki soru kalır: iyi bir h nasıl seçilir, çakışma nasıl çözülür?

HASH FİKRİ

Soyut veri tipi: sözlük (dictionary)

```
typedef struct { int anahtar; /* + değer */ } oge;

bool hash_ekle(oge x);       /* varsa günceller */
oge *hash_ara(int anahtar); /* yoksa NULL      */
bool hash_sil(int anahtar); /* yoksa false     */
```

Sözlük

Anahtar → değer eşlemesi; her anahtar en çok bir kez bulunur.

Kullanım

Derleyicide sembol tablosu (Horowitz'in çıkış örneği), önbellek, tekrar denetimi, küme gerçeklemesi.

C standart kitaplığında hazır bir hash tablosu yoktur (POSIX'te sınırlı `hsearch` vardır); bu yüzden C programcısı kendi tablosunu yazmayı bilmelidir.

İyi bir hash fonksiyonunun özellikleri

Hızlı

Hesaplaması O(anahtar uzunluğu); birkaç aritmetik işlem.

Düzgün dağıtan

Anahtarlar m kovaya olabildiğince eşit dağılmalı (simple uniform hashing varsayımı).

Tüm bitleri kullanan

Yalnızca son basamaklara bakan fonksiyonlar örüntülü veride kümelenir.

Deterministik

Aynı anahtar her zaman aynı kovaya gider; rastgelelik yalnızca fonksiyon seçilirken olabilir.

Bölme yöntemi: h(k) = k mod m

k	10	20	30	40	50	60	17	27
k mod 10	0	0	0	0	0	0	7	7
k mod 11	10	9	8	7	6	5	6	5

- m = 10: son basamak dışında her şey yok sayılır; 10, 20, …, 60 aynı kovaya düşer.
- m için 2'nin kuvvetine yakın olmayan bir **asal sayı** seçmek iyi bir kuraldır (CLRS).
- m = 2^p ise h(k) yalnızca k'nin en düşük p bitidir: kaçınılmalı.

```
unsigned h_bolme(unsigned k, unsigned m) { return k % m; }
```

C'de `%` negatif sayıda negatif sonuç verebilir: anahtar `unsigned` tutulur ya da `((k % m) + m) % m` kullanılır.

Çarpma yöntemi: h(k) = ⌊m · (k·A mod 1)⌋

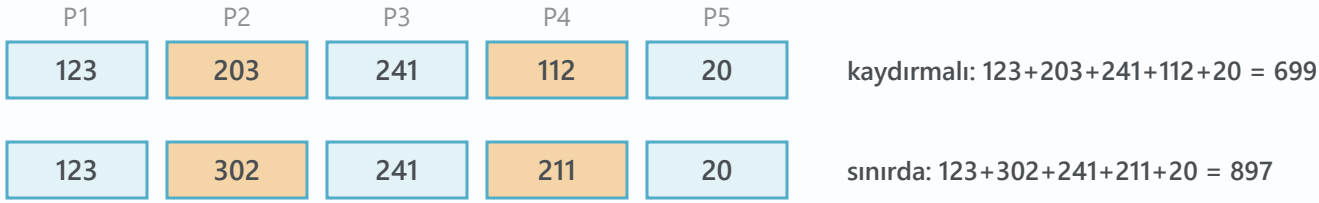
- 0 < A < 1 sabit; k·A'nın **kesir kısmı** m ile çarpılır.
- m'nin seçimi kritik değildir; m = 2^p seçilebilir.
- Knuth'un önerisi A ≈ (√5 − 1)/2 = 0,6180339887…
- Örnek (CLRS): k = 123456, p = 14 (m = 16384) → **h(k) = 67**.

```
/* w = 32 bit, m = 2^p; s = A * 2^32 */
uint32_t h_carpma(uint32_t k, int p) {
  const uint32_t s = 2654435769u;
  return (uint32_t)(k * s) >> (32 - p);
}
```

Tamsayı sürümünde k·s çarpımı 2^{32}'ye göre taşar; taşan kısım atılır (kesir kısmını almakla aynı). Kalan 32 bitin en üst p biti h(k)'dir. Kayan noktalı sürüm aynı sonucu verir: ⌊16384 · 0,…⌋ = 67. (`<stdint.h>` gerekir.)

Katlama (folding) ve orta-kare yöntemleri

Anahtar `x = 12320324111220` üç haneli parçalara bölünüp toplanır (Horowitz örneği):



Kaydırmalı katlama

Parçalar olduğu gibi toplanır: **699**.

Sınırda katlama

Çift sıradaki parçalar ters çevrilir (kâğıt katlar gibi): **897**.

Orta-kare

k^2 hesaplanır, ortadaki r bit alınır: $m = 2^r$. Orta bitler k'nin tüm bitlerine bağlıdır.

Sonuç tablo boyutundan büyükse ayrıca `mod m` alınır.

Dizgi anahtarlar: polinom (Horner) hash

```c
unsigned dizgi_hash(const char *s, unsigned m) {
  unsigned h = 0;
  for (; *s != '\0'; s++)
    h = (h * 31 + (unsigned char)*s) % m;
  return h;
}
```

```c
unsigned long djb2(const char *s) {  /* Bernstein */
  unsigned long h = 5381;
  int c;
  while ((c = (unsigned char)*s++) != 0)
    h = h * 33 + c;
  return h;       /* sonra: h % m */
}
```

Toplama kötüdür

Harf kodlarını toplamak `"abc"`, `"cba"` gibi anagramları aynı kovaya atar.

Polinom

$h = c_0 \cdot 31^{n-1} + \ldots + c_{n-1}$: Horner kuralıyla her karakterde bir çarpma. Konum bilgisi hash'e girer.

Taşma

Her adımda `% m` almak değerleri küçük tutar; `unsigned` taşması C'de tanımlıdır (mod 2^w).

Kod izleme: `dizgi_hash("kedi", 11)`

karakterler ve kodları

k	e	d	i	\0
107	101	100	105	

h = ? m = 11

`dizgi_hash("kedi", 11)` çağrılır.

Adım 1 / 12

```
▶ unsigned dizgi_hash(const char *s, unsigned m) {
2   unsigned h = 0;
3   for (; *s != '\0'; s++)
4     h = (h * 31 + (unsigned char)*s) % m;
5   return h;
6 }
```

DEĞİŞKENLER

`s = "kedi"` `m = 11`

ÇAĞRI YIĞINI

`dizgi_hash("kedi", 11)`

`main()`

Denetim: tam polinom 3287903 ve 3287903 mod 11 = 3; her adımda mod almak sonucu değiştirmez.

Kod izleme: `dizgi_hash("kedi", 11)`

karakterler ve kodları

k	e	d	i	\0
107	101	100	105	

h = 0 m = 11

h = 0 ile başlanır.

Adım 2 / 12

```
1 unsigned dizgi_hash(const char *s, unsigned m) {
▶   unsigned h = 0;
3   for (; *s != '\0'; s++)
4     h = (h * 31 + (unsigned char)*s) % m;
5   return h;
6 }
```

DEĞİŞKENLER

`s = "kedi"` `m = 11` `h = 0`

ÇAĞRI YIĞINI

`dizgi_hash("kedi", 11)`

`main()`

Denetim: tam polinom 3287903 ve 3287903 mod 11 = 3; her adımda mod almak sonucu değiştirmez.

Kod izleme: `dizgi_hash("kedi", 11)`

karakterler ve kodları

k	e	d	i	\0
107	101	100	105	

h = 0 m = 11

`*s` = 'k' ≠ '\0' → döngü sürer.

Adım 3 / 12

```
1 unsigned dizgi_hash(const char *s, unsigned m) {
2   unsigned h = 0;
▶   for (; *s != '\0'; s++)
4     h = (h * 31 + (unsigned char)*s) % m;
5   return h;
6 }
```

DEĞİŞKENLER

`s = "kedi"` `m = 11` `h = 0`

`*s = 'k' (107)`

ÇAĞRI YIĞINI

`dizgi_hash("kedi", 11)`

`main()`

Denetim: tam polinom 3287903 ve 3287903 mod 11 = 3; her adımda mod almak sonucu değiştirmez.

Kod izleme: `dizgi_hash("kedi", 11)`

karakterler ve kodları

k	e	d	i	\0
107	101	100	105	

h = 8 m = 11

h = (0·31 + 107) % 11 = 107 % 11 = **8**.

Adım 4 / 12

```
1 unsigned dizgi_hash(const char *s, unsigned m) {
2   unsigned h = 0;
3   for (; *s != '\0'; s++)
▶     h = (h * 31 + (unsigned char)*s) % m;
5   return h;
6 }
```

DEĞİŞKENLER

`s = "kedi"` `m = 11` `h = 8` `*s = 'k' (107)`

ÇAĞRI YIĞINI

`dizgi_hash("kedi", 11)`

`main()`

Denetim: tam polinom 3287903 ve 3287903 mod 11 = 3; her adımda mod almak sonucu değiştirmez.

Kod izleme: `dizgi_hash("kedi", 11)`

karakterler ve kodları

k	e	d	i	\0
107	101	100	105	

h = 8 m = 11

`*s` = 'e' ≠ '\0' → döngü sürer.

Adım 5 / 12

```
1 unsigned dizgi_hash(const char *s, unsigned m) {
2   unsigned h = 0;
▶   for (; *s != '\0'; s++)
4     h = (h * 31 + (unsigned char)*s) % m;
5   return h;
6 }
```

DEĞİŞKENLER

`s = "edi"` `m = 11` `h = 8`

`*s = 'e' (101)`

ÇAĞRI YIĞINI

`dizgi_hash("kedi", 11)`

`main()`

Denetim: tam polinom 3287903 ve 3287903 mod 11 = 3; her adımda mod almak sonucu değiştirmez.

Kod izleme: `dizgi_hash("kedi", 11)`

karakterler ve kodları

k	e	d	i	\0
107	101	100	105	

h = 8 m = 11

h = (8·31 + 101) % 11 = 349 % 11 = **8**.

Adım 6 / 12

```
1 unsigned dizgi_hash(const char *s, unsigned m) {
2   unsigned h = 0;
3   for (; *s != '\0'; s++)
▶     h = (h * 31 + (unsigned char)*s) % m;
5   return h;
6 }
```

DEĞİŞKENLER

`s = "edi"` `m = 11` `h = 8` `*s = 'e' (101)`

ÇAĞRI YIĞINI

`dizgi_hash("kedi", 11)`

`main()`

Denetim: tam polinom 3287903 ve 3287903 mod 11 = 3; her adımda mod almak sonucu değiştirmez.

Kod izleme: `dizgi_hash("kedi", 11)`

karakterler ve kodları

k	e	d	i	\0
107	101	100	105	

h = 8 m = 11

`*s` = 'd' ≠ '\0' → döngü sürer.

Adım 7 / 12

```
1 unsigned dizgi_hash(const char *s, unsigned m) {
2   unsigned h = 0;
▶   for (; *s != '\0'; s++)
4     h = (h * 31 + (unsigned char)*s) % m;
5   return h;
6 }
```

DEĞİŞKENLER

`s = "di"` `m = 11` `h = 8` `*s = 'd' (100)`

ÇAĞRI YIĞINI

`dizgi_hash("kedi", 11)`

`main()`

Denetim: tam polinom 3287903 ve 3287903 mod 11 = 3; her adımda mod almak sonucu değiştirmez.

Kod izleme: `dizgi_hash("kedi", 11)`

karakterler ve kodları

k	e	d	i	\0
107	101	100	105	

h = 7 m = 11

h = (8·31 + 100) % 11 = 348 % 11 = **7**.

Adım 8 / 12

```
1 unsigned dizgi_hash(const char *s, unsigned m) {
2   unsigned h = 0;
3   for (; *s != '\0'; s++)
▶     h = (h * 31 + (unsigned char)*s) % m;
5   return h;
6 }
```

DEĞİŞKENLER

`s = "di"` `m = 11` `h = 7` `*s = 'd' (100)`

ÇAĞRI YIĞINI

`dizgi_hash("kedi", 11)`

`main()`

Denetim: tam polinom 3287903 ve 3287903 mod 11 = 3; her adımda mod almak sonucu değiştirmez.

Kod izleme: `dizgi_hash("kedi", 11)`

karakterler ve kodları

k	e	d	i	\0
107	101	100	105	

h = 7 m = 11

`*s` = 'i' ≠ '\0' → döngü sürer.

Adım 9 / 12

```
1 unsigned dizgi_hash(const char *s, unsigned m) {
2   unsigned h = 0;
▶   for (; *s != '\0'; s++)
4     h = (h * 31 + (unsigned char)*s) % m;
5   return h;
6 }
```

DEĞİŞKENLER

`s = "i"` `m = 11` `h = 7` `*s = 'i' (105)`

ÇAĞRI YIĞINI

`dizgi_hash("kedi", 11)`

`main()`

Denetim: tam polinom 3287903 ve 3287903 mod 11 = 3; her adımda mod almak sonucu değiştirmez.

Kod izleme: `dizgi_hash("kedi", 11)`

karakterler ve kodları

k	e	d	i	\0
107	101	100	105	

h = 3 m = 11

h = (7·31 + 105) % 11 = 322 % 11 = **3**.

Adım 10 / 12

```
1 unsigned dizgi_hash(const char *s, unsigned m) {
2   unsigned h = 0;
3   for (; *s != '\0'; s++)
▶     h = (h * 31 + (unsigned char)*s) % m;
5   return h;
6 }
```

DEĞİŞKENLER

`s = "i"` `m = 11` `h = 3` `*s = 'i' (105)`

ÇAĞRI YIĞINI

`dizgi_hash("kedi", 11)`

`main()`

Denetim: tam polinom 3287903 ve 3287903 mod 11 = 3; her adımda mod almak sonucu değiştirmez.

Kod izleme: `dizgi_hash("kedi", 11)`

karakterler ve kodları

k	e	d	i	\0
107	101	100	105	

h = 3

m = 11

`*s == '\0'` : dizgi bitti.

Adım 11 / 12

```
1 unsigned dizgi_hash(const char *s, unsigned m) {
2   unsigned h = 0;
▶   for (; *s != '\0'; s++)
4     h = (h * 31 + (unsigned char)*s) % m;
5   return h;
6 }
```

DEĞİŞKENLER

`s = ""` `m = 11` `h = 3` `*s = '\0'`

ÇAĞRI YIĞINI

`dizgi_hash("kedi", 11)`

`main()`

Denetim: tam polinom 3287903 ve 3287903 mod 11 = 3; her adımda mod almak sonucu değiştirmez.

Kod izleme: `dizgi_hash("kedi", 11)`

karakterler ve kodları

k	e	d	i	\0
107	101	100	105	

h = 3 m = 11

return 3: "kedi" anahtarının ev adresi 3. Maliyet dizgi uzunluğuyla doğrusal.

Adım 12 / 12

```
1 unsigned dizgi_hash(const char *s, unsigned m) {
2   unsigned h = 0;
3   for (; *s != '\0'; s++)
4     h = (h * 31 + (unsigned char)*s) % m;
▶   return h;
6 }
```

DEĞİŞKENLER

m = 11 h = 3

ÇAĞRI YIĞINI

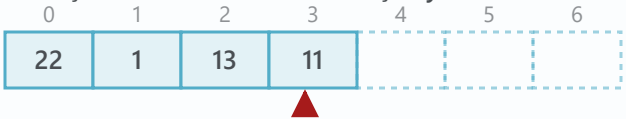
dizgi_hash("kedi", 11)

main()

Denetim: tam polinom 3287903 ve 3287903 mod 11 = 3; her adımda mod almak sonucu değiştirmez.

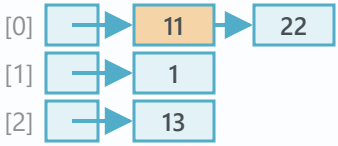
Çakışma: iki temel çözüm yolu

Açık adresleme: tabloda başka yuva aranır



11 → ev 0 dolu → 1 → 2 → 3

Zincirleme: kovada liste tutulur



Doğum günü paradoksu

23 kişilik bir sınıfta iki kişinin doğum gününün aynı olma olasılığı ≈ %50,7. m = 365 yuvaya 23 anahtar: çakışma şaşırtıcı derecede erken gelir.

Açık adresleme

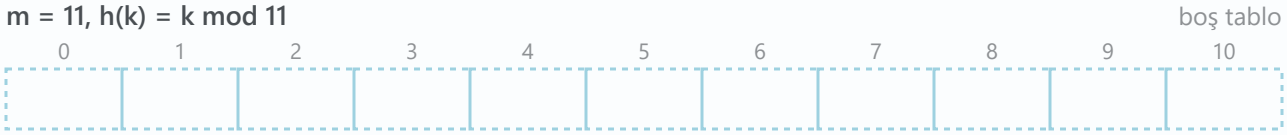
Tüm öğeler tablonun içinde; $\alpha \leq 1$. Yoklama (probe) dizisiyle boş yuva aranır.

Zincirleme

Her kova bir bağlı listenin başı; $\alpha > 1$ olabilir. Ekleme O(1).

AÇIK ADRESLEME

Doğrusal yoklama: h(k, i) = (h(k) + i) mod m

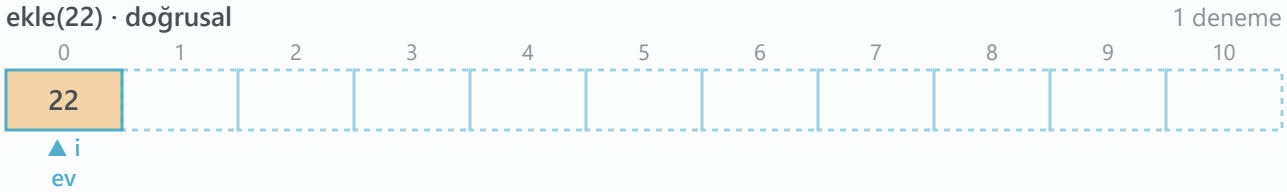


Sırasıyla **22, 1, 13, 11, 24, 33** eklenecek. Ev adresleri: 0, 1, 2, 0, 2, 0.

Adım 1 / 17

Birincil kümelenme (primary clustering): dolu yuvalar art arda bir blok oluşturur; bloğa düşen her yeni anahtar bloğu büyütür. 33 için 6 deneme gerekti.

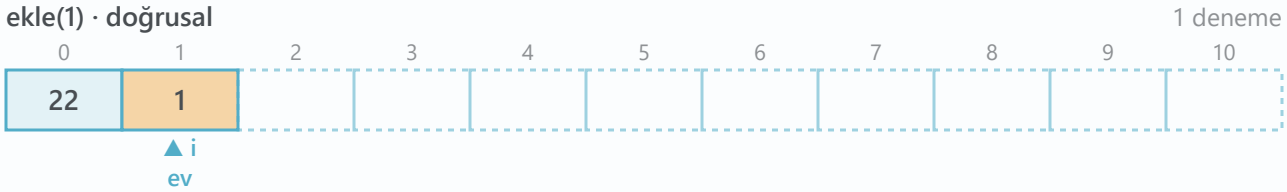
Doğrusal yoklama: h(k, i) = (h(k) + i) mod m



22 → ev 0: **yuva 0**.

Birincil kümelenme (primary clustering): dolu yuvalar art arda bir blok oluşturur; bloğa düşen her yeni anahtar bloğu büyütür. 33 için 6 deneme gerekti.

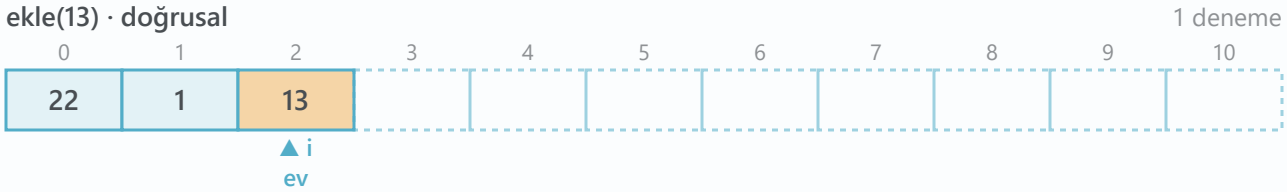
Doğrusal yoklama: h(k, i) = (h(k) + i) mod m



1 → ev 1: **yuva 1**.

Birincil kümelenme (primary clustering): dolu yuvalar art arda bir blok oluşturur; bloğa düşen her yeni anahtar bloğu büyütür. 33 için 6 deneme gerekti.

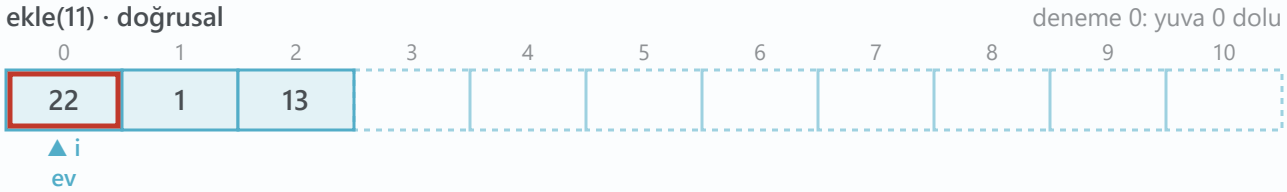
Doğrusal yoklama: h(k, i) = (h(k) + i) mod m



13 → ev 2: **yuva 2**.

Birincil kümelenme (primary clustering): dolu yuvalar art arda bir blok oluşturur; bloğa düşen her yeni anahtar bloğu büyütür. 33 için 6 deneme gerekti.

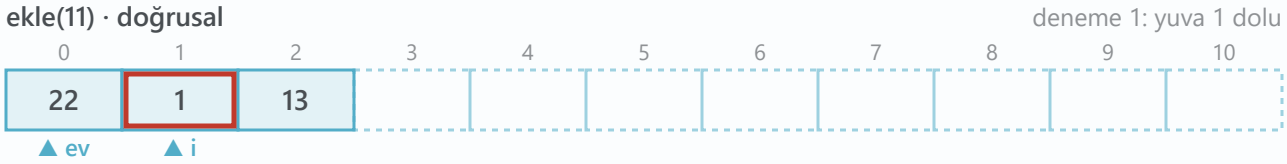
Doğrusal yoklama: h(k, i) = (h(k) + i) mod m



11: yuva **0** dolu (22) → sıradaki yuvaya bakılır.

Birincil kümelenme (primary clustering): dolu yuvalar art arda bir blok oluşturur; bloğa düşen her yeni anahtar bloğu büyütür. 33 için 6 deneme gerekti.

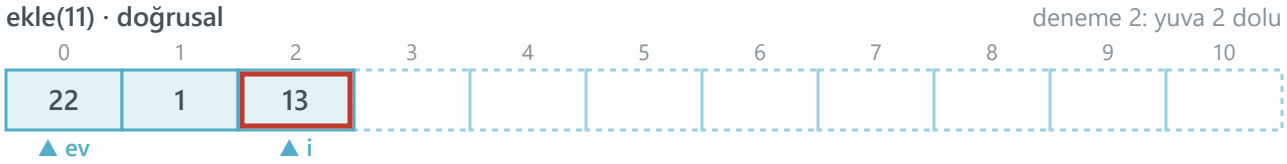
Doğrusal yoklama: h(k, i) = (h(k) + i) mod m



11: yuva **1** dolu (1) → sıradaki yuvaya bakılır.

Birincil kümelenme (primary clustering): dolu yuvalar art arda bir blok oluşturur; bloğa düşen her yeni anahtar bloğu büyütür. 33 için 6 deneme gerekti.

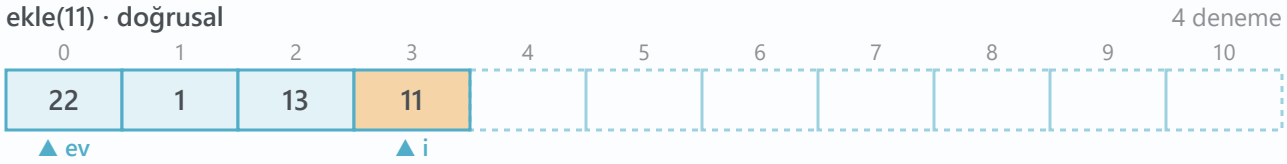
Doğrusal yoklama: h(k, i) = (h(k) + i) mod m



11: yuva **2** dolu (13) → sıradaki yuvaya bakılır.

Birincil kümelenme (primary clustering): dolu yuvalar art arda bir blok oluşturur; bloğa düşen her yeni anahtar bloğu büyütür. 33 için 6 deneme gerekti.

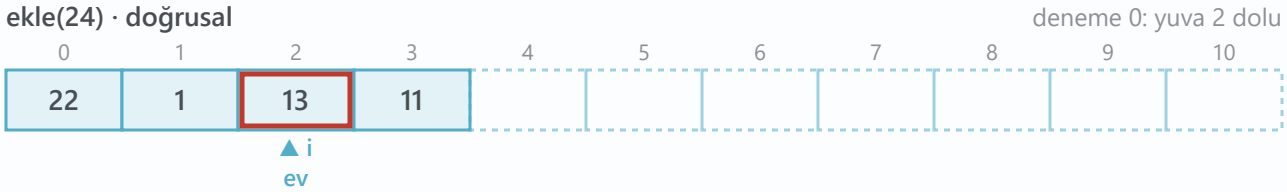
Doğrusal yoklama: h(k, i) = (h(k) + i) mod m



11 → ev 0, yoklama dizisi 0, 1, 2, 3: **yuva 3**.

Birincil kümelenme (primary clustering): dolu yuvalar art arda bir blok oluşturur; bloğa düşen her yeni anahtar bloğu büyütür. 33 için 6 deneme gerekti.

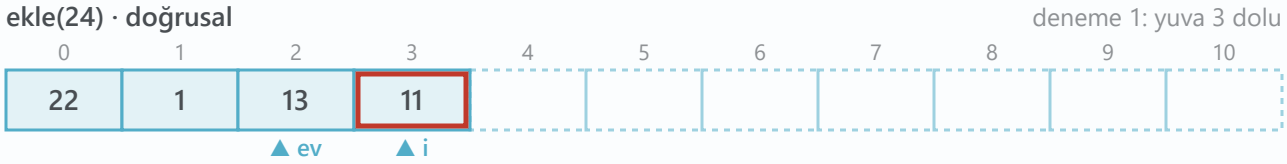
Doğrusal yoklama: h(k, i) = (h(k) + i) mod m



24: yuva **2** dolu (13) → sıradaki yuvaya bakılır.

Birincil kümelenme (primary clustering): dolu yuvalar art arda bir blok oluşturur; bloğa düşen her yeni anahtar bloğu büyütür. 33 için 6 deneme gerekti.

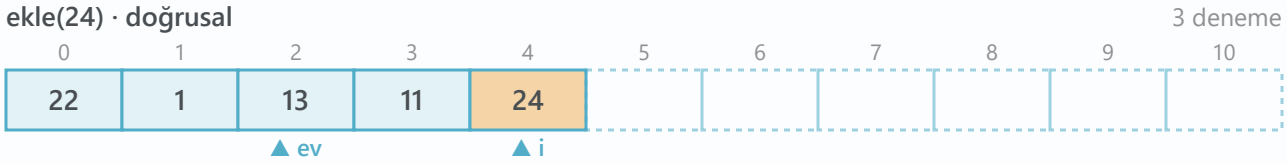
Doğrusal yoklama: h(k, i) = (h(k) + i) mod m



24: yuva **3** dolu (11) → sıradaki yuvaya bakılır.

Birincil kümelenme (primary clustering): dolu yuvalar art arda bir blok oluşturur; bloğa düşen her yeni anahtar bloğu büyütür. 33 için 6 deneme gerekti.

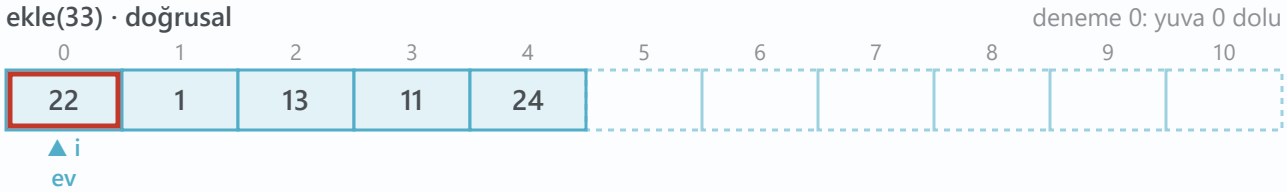
Doğrusal yoklama: h(k, i) = (h(k) + i) mod m



24 → ev 2, yoklama dizisi 2, 3, 4: **yuva 4**.

Birincil kümelenme (primary clustering): dolu yuvalar art arda bir blok oluşturur; bloğa düşen her yeni anahtar bloğu büyütür. 33 için 6 deneme gerekti.

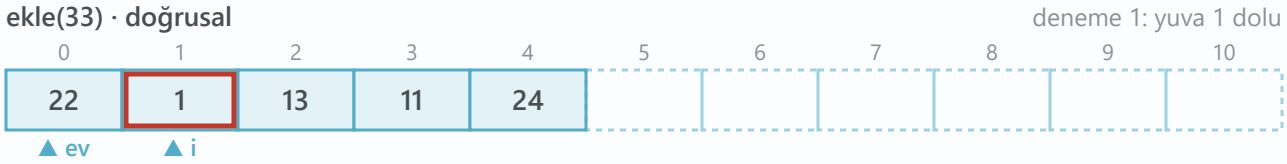
Doğrusal yoklama: h(k, i) = (h(k) + i) mod m



33: yuva **0** dolu (22) → sıradaki yuvaya bakılır.

Birincil kümelenme (primary clustering): dolu yuvalar art arda bir blok oluşturur; bloğa düşen her yeni anahtar bloğu büyütür. 33 için 6 deneme gerekti.

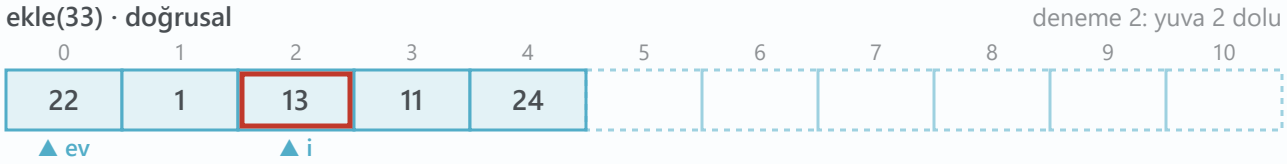
Doğrusal yoklama: h(k, i) = (h(k) + i) mod m



33: yuva **1** dolu (1) → sıradaki yuvaya bakılır.

Birincil kümelenme (primary clustering): dolu yuvalar art arda bir blok oluşturur; bloğa düşen her yeni anahtar bloğu büyütür. 33 için 6 deneme gerekti.

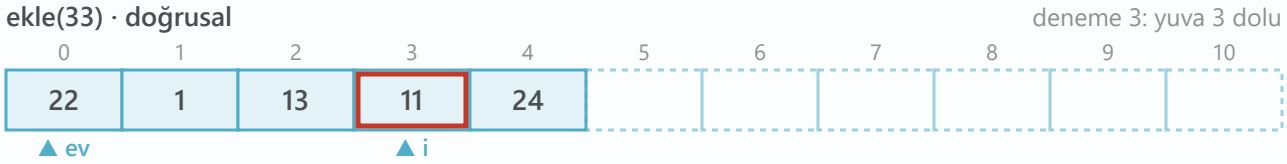
Doğrusal yoklama: h(k, i) = (h(k) + i) mod m



33: yuva **2** dolu (13) → sıradaki yuvaya bakılır.

Birincil kümelenme (primary clustering): dolu yuvalar art arda bir blok oluşturur; bloğa düşen her yeni anahtar bloğu büyütür. 33 için 6 deneme gerekti.

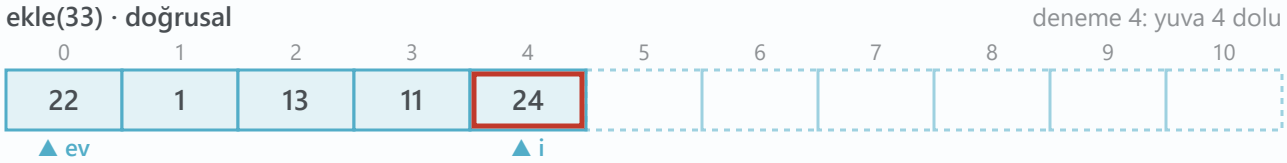
Doğrusal yoklama: h(k, i) = (h(k) + i) mod m



33: yuva **3** dolu (11) → sıradaki yuvaya bakılır.

Birincil kümelenme (primary clustering): dolu yuvalar art arda bir blok oluşturur; bloğa düşen her yeni anahtar bloğu büyütür. 33 için 6 deneme gerekti.

Doğrusal yoklama: h(k, i) = (h(k) + i) mod m

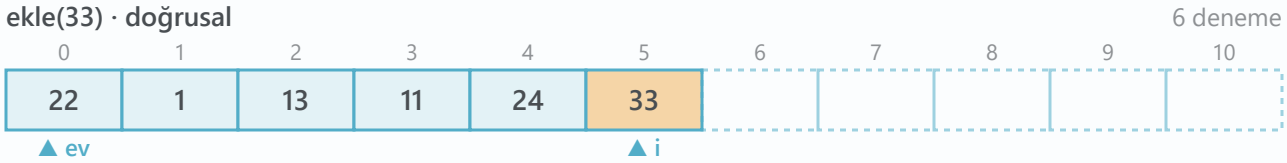


33: yuva **4** dolu (24) → sıradaki yuvaya bakılır.

Birincil kümelenme (primary clustering): dolu yuvalar art arda bir blok oluşturur; bloğa düşen her yeni anahtar bloğu büyütür. 33 için 6 deneme gerekti.

AÇIK ADRESLEME

Doğrusal yoklama: h(k, i) = (h(k) + i) mod m



33 → ev 0, yoklama dizisi 0, 1, 2, 3, 4, 5: **yuva 5**.

Adım 17 / 17

Birincil kümelenme (primary clustering): dolu yuvalar art arda bir blok oluşturur; bloğa düşen her yeni anahtar bloğu büyütür. 33 için 6 deneme gerekti.

AÇIK ADRESLEME

C ile açık adresleme: yuva durumu ve ekleme

```c
#define M 11
typedef enum { BOS, DOLU, SILINDI } durum_t;
typedef struct {
  int anahtar;          /* anahtar >= 0 varsayılır */
  durum_t durum;
} yuva;
yuva tablo[M];          /* global: hepsi 0 = BOS */
```

Başlangıç

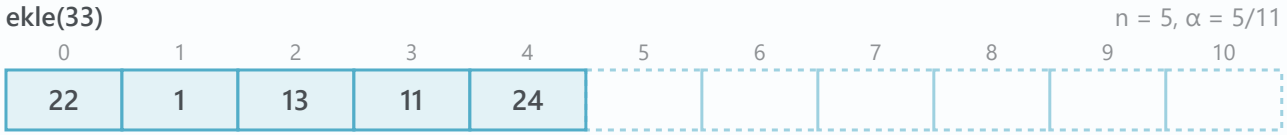
Global dizi sıfırla başlar; `BOS` = 0 olduğu için tüm yuvalar boştur.

Ekleme

Ev adresinden başlanır, `DOLU` olmayan ilk yuvaya yazılır; en çok m deneme.

Yuva üç durumdan birindedir: **BOS** (hiç kullanılmadı), **DOLU**, **SILINDI** (silindi işareti). Ekleme SILINDI yuvayı yeniden kullanabilir; ancak aynı anahtarın tabloda zaten olmadığı önce `ara` ile denetlenmelidir.

Kod izleme: `ekle(33)` — doğrusal yoklama



ekle(33): tabloda 22, 1, 13, 11, 24 var (yuva 0–4 dolu blok).

Adım 1 / 16

```
▶ int ekle(int k) {
2   int i = k % M;                    /* ev adresi */
3   for (int d = 0; d < M; d++, i = (i + 1) % M) {
4     if (tablo[i].durum != DOLU) {   /* BOS/SILINDI */
5       tablo[i].anahtar = k;
6       tablo[i].durum = DOLU;
7       return i;
8     }
9   }
10  return -1;                        /* tablo dolu */
11 }
```

DEĞİŞKENLER

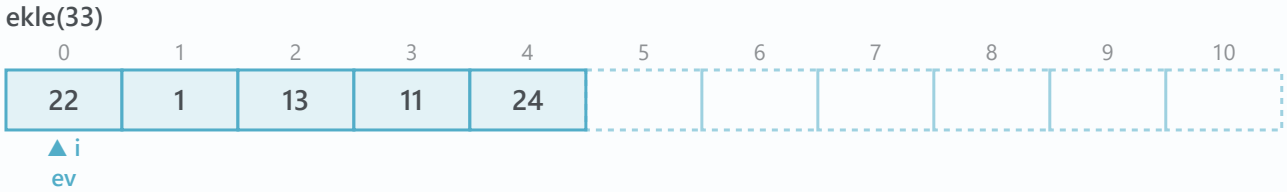
`k = 33`

ÇAĞRI YIĞINI

`ekle(33)`

`main()`

Kod izleme: `ekle(33)` — doğrusal yoklama



Ev adresi **i = 33 % 11 = 0**.

Adım 2 / 16

```
1  int ekle(int k) {
▶    int i = k % M;                    /* ev adresi */
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
4      if (tablo[i].durum != DOLU) {   /* BOS/SILINDI */
5        tablo[i].anahtar = k;
6        tablo[i].durum = DOLU;
7        return i;
8      }
9    }
10   return -1;                        /* tablo dolu */
11 }
```

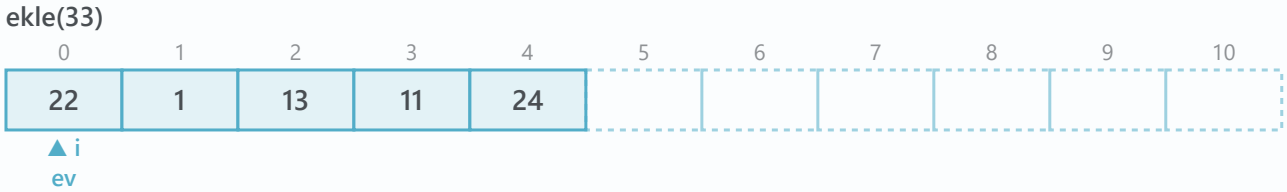
DEĞİŞKENLER

`k = 33` `i = 0`

ÇAĞRI YIĞINI

- **ekle(33)**
- main()

Kod izleme: `ekle(33)` — doğrusal yoklama



d = 0 < 11: 0. deneme, yuva 0.

Adım 3 / 16

```
1  int ekle(int k) {
2    int i = k % M;                    /* ev adresi */
▶    for (int d = 0; d < M; d++, i = (i + 1) % M) {
4      if (tablo[i].durum != DOLU) {   /* BOS/SILINDI */
5        tablo[i].anahtar = k;
6        tablo[i].durum = DOLU;
7        return i;
8      }
9    }
10   return -1;                        /* tablo dolu */
11 }
```

DEĞİŞKENLER

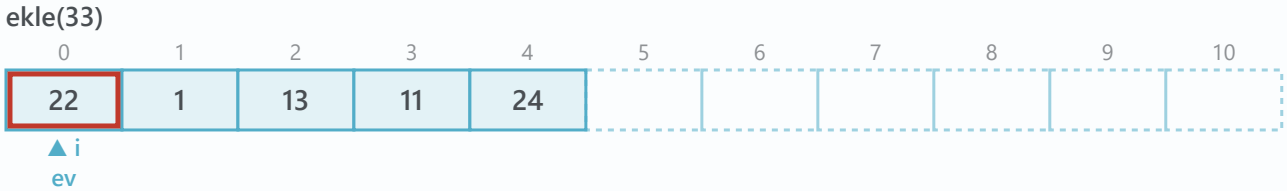
`k = 33` `i = 0` `d = 0`

ÇAĞRI YIĞINI

`ekle(33)`

`main()`

Kod izleme: `ekle(33)` — doğrusal yoklama



Yuva 0 DOLU (22) → sonraki: i = (0+1) % 11.

Adım 4 / 16

```
1  int ekle(int k) {
2    int i = k % M;                    /* ev adresi */
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (tablo[i].durum != DOLU) {   /* BOS/SILINDI */
5        tablo[i].anahtar = k;
6        tablo[i].durum = DOLU;
7        return i;
8      }
9    }
10   return -1;                        /* tablo dolu */
11 }
```

DEĞİŞKENLER

`k = 33` `i = 0` `d = 0`

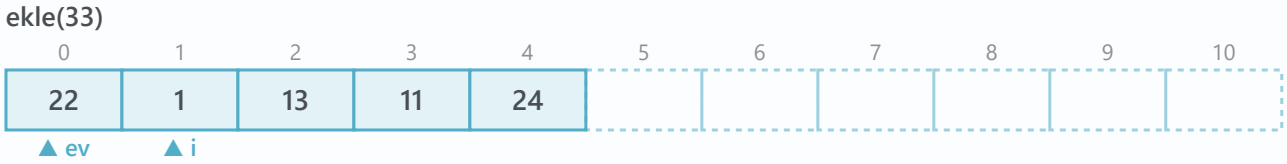
`tablo[i].durum = DOLU`

ÇAĞRI YIĞINI

`ekle(33)`

`main()`

Kod izleme: `ekle(33)` — doğrusal yoklama



d = 1 < 11: 1. deneme, yuva 1.

Adım 5 / 16

```
1  int ekle(int k) {
2    int i = k % M;                    /* ev adresi */
▶    for (int d = 0; d < M; d++, i = (i + 1) % M) {
4      if (tablo[i].durum != DOLU) {   /* BOS/SILINDI */
5        tablo[i].anahtar = k;
6        tablo[i].durum = DOLU;
7        return i;
8      }
9    }
10   return -1;                        /* tablo dolu */
11 }
```

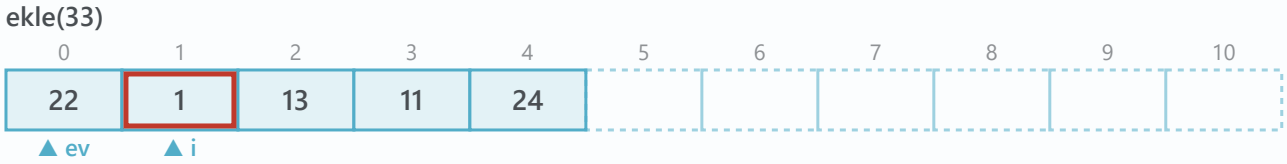
DEĞİŞKENLER

`k = 33` `i = 1` `d = 1`

ÇAĞRI YIĞINI

- `ekle(33)`
- `main()`

Kod izleme: `ekle(33)` — doğrusal yoklama



Yuva 1 DOLU (1) → sonraki: i = (1+1) % 11.

Adım 6 / 16

```
1  int ekle(int k) {
2    int i = k % M;                    /* ev adresi */
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (tablo[i].durum != DOLU) {   /* BOS/SILINDI */
5        tablo[i].anahtar = k;
6        tablo[i].durum = DOLU;
7        return i;
8      }
9    }
10   return -1;                        /* tablo dolu */
11 }
```

DEĞİŞKENLER

`k = 33` `i = 1` `d = 1`

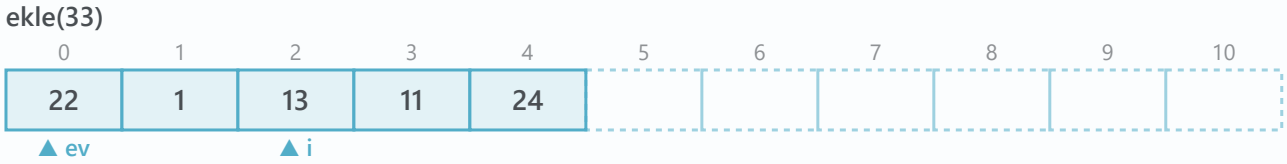
`tablo[i].durum = DOLU`

ÇAĞRI YIĞINI

`ekle(33)`

`main()`

Kod izleme: `ekle(33)` — doğrusal yoklama



d = 2 < 11: 2. deneme, yuva 2.

Adım 7 / 16

```
1  int ekle(int k) {
2    int i = k % M;                    /* ev adresi */
▶    for (int d = 0; d < M; d++, i = (i + 1) % M) {
4      if (tablo[i].durum != DOLU) {   /* BOS/SILINDI */
5        tablo[i].anahtar = k;
6        tablo[i].durum = DOLU;
7        return i;
8      }
9    }
10   return -1;                        /* tablo dolu */
11 }
```

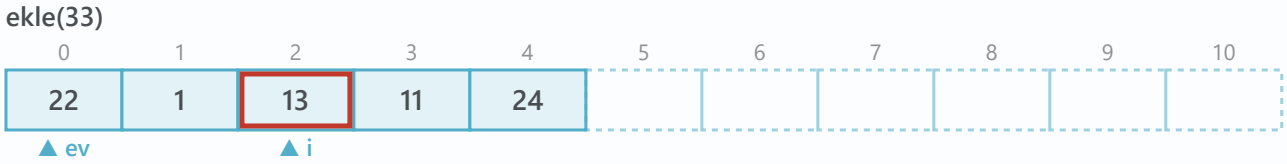
DEĞİŞKENLER

`k = 33` `i = 2` `d = 2`

ÇAĞRI YIĞINI

- **ekle(33)**
- main()

Kod izleme: `ekle(33)` — doğrusal yoklama



Yuva 2 DOLU (13) → sonraki: i = (2+1) % 11.

Adım 8 / 16

```
1  int ekle(int k) {
2    int i = k % M;                    /* ev adresi */
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (tablo[i].durum != DOLU) {   /* BOS/SILINDI */
5        tablo[i].anahtar = k;
6        tablo[i].durum = DOLU;
7        return i;
8      }
9    }
10   return -1;                        /* tablo dolu */
11 }
```

DEĞİŞKENLER

`k = 33` `i = 2` `d = 2`

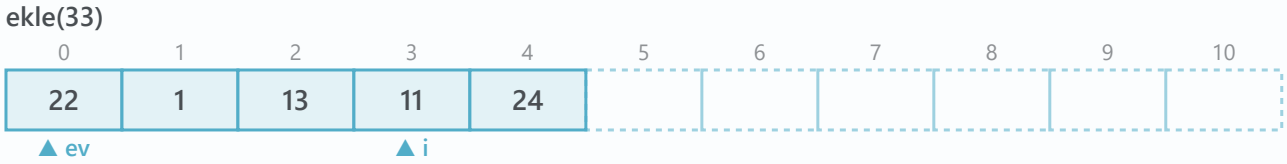
`tablo[i].durum = DOLU`

ÇAĞRI YIĞINI

`ekle(33)`

`main()`

Kod izleme: `ekle(33)` — doğrusal yoklama



d = 3 < 11: 3. deneme, yuva 3.

Adım 9 / 16

```
1  int ekle(int k) {
2    int i = k % M;                    /* ev adresi */
▶    for (int d = 0; d < M; d++, i = (i + 1) % M) {
4      if (tablo[i].durum != DOLU) {   /* BOS/SILINDI */
5        tablo[i].anahtar = k;
6        tablo[i].durum = DOLU;
7        return i;
8      }
9    }
10   return -1;                        /* tablo dolu */
11 }
```

DEĞİŞKENLER

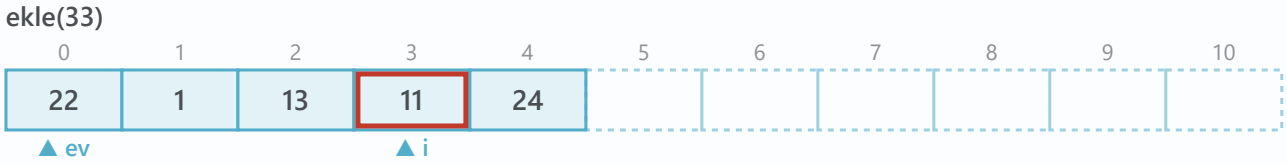
`k = 33` `i = 3` `d = 3`

ÇAĞRI YIĞINI

`ekle(33)`

`main()`

Kod izleme: `ekle(33)` — doğrusal yoklama



Yuva 3 DOLU (11) → sonraki: i = (3+1) % 11.

Adım 10 / 16

```
1  int ekle(int k) {
2    int i = k % M;                    /* ev adresi */
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (tablo[i].durum != DOLU) {   /* BOS/SILINDI */
5        tablo[i].anahtar = k;
6        tablo[i].durum = DOLU;
7        return i;
8      }
9    }
10   return -1;                        /* tablo dolu */
11 }
```

DEĞİŞKENLER

`k = 33` `i = 3` `d = 3`

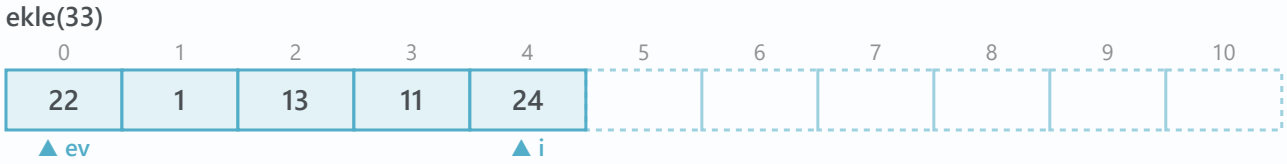
`tablo[i].durum = DOLU`

ÇAĞRI YIĞINI

`ekle(33)`

`main()`

Kod izleme: `ekle(33)` — doğrusal yoklama



d = 4 < 11: 4. deneme, yuva 4.

Adım 11 / 16

```
1  int ekle(int k) {
2    int i = k % M;                    /* ev adresi */
▶    for (int d = 0; d < M; d++, i = (i + 1) % M) {
4      if (tablo[i].durum != DOLU) {   /* BOS/SILINDI */
5        tablo[i].anahtar = k;
6        tablo[i].durum = DOLU;
7        return i;
8      }
9    }
10   return -1;                        /* tablo dolu */
11 }
```

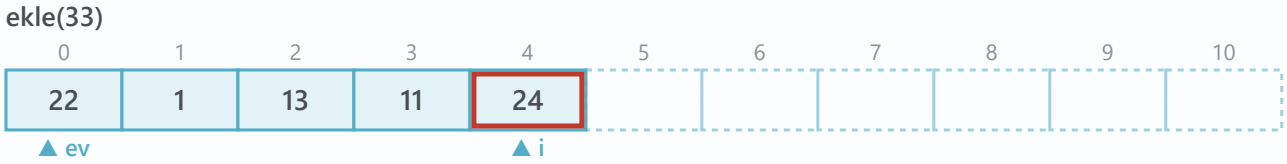
DEĞİŞKENLER

k = 33 | i = 4 | d = 4

ÇAĞRI YIĞINI

- ekle(33)
- main()

Kod izleme: `ekle(33)` — doğrusal yoklama



Yuva 4 DOLU (24) → sonraki: i = (4+1) % 11.

Adım 12 / 16

```
1  int ekle(int k) {
2    int i = k % M;                    /* ev adresi */
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (tablo[i].durum != DOLU) {   /* BOS/SILINDI */
5        tablo[i].anahtar = k;
6        tablo[i].durum = DOLU;
7        return i;
8      }
9    }
10   return -1;                        /* tablo dolu */
11 }
```

DEĞİŞKENLER

`k = 33` `i = 4` `d = 4`

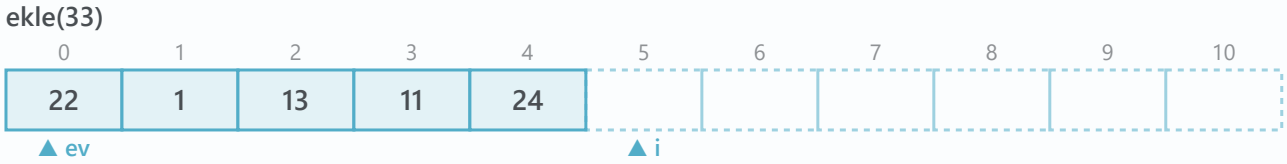
`tablo[i].durum = DOLU`

ÇAĞRI YIĞINI

`ekle(33)`

`main()`

Kod izleme: `ekle(33)` — doğrusal yoklama



d = 5 < 11: 5. deneme, yuva 5.

Adım 13 / 16

```
1  int ekle(int k) {
2    int i = k % M;                    /* ev adresi */
▶    for (int d = 0; d < M; d++, i = (i + 1) % M) {
4      if (tablo[i].durum != DOLU) {   /* BOS/SILINDI */
5        tablo[i].anahtar = k;
6        tablo[i].durum = DOLU;
7        return i;
8      }
9    }
10   return -1;                        /* tablo dolu */
11 }
```

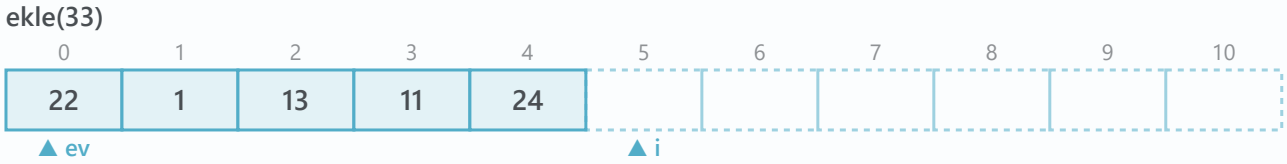
DEĞİŞKENLER

k = 33 | i = 5 | d = 5

ÇAĞRI YIĞINI

- ekle(33)
- main()

Kod izleme: `ekle(33)` — doğrusal yoklama



Yuva 5 **BOS** → buraya yazılacak.

Adım 14 / 16

```
1  int ekle(int k) {
2    int i = k % M;                    /* ev adresi */
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (tablo[i].durum != DOLU) {   /* BOS/SILINDI */
5        tablo[i].anahtar = k;
6        tablo[i].durum = DOLU;
7        return i;
8      }
9    }
10   return -1;                        /* tablo dolu */
11 }
```

DEĞİŞKENLER

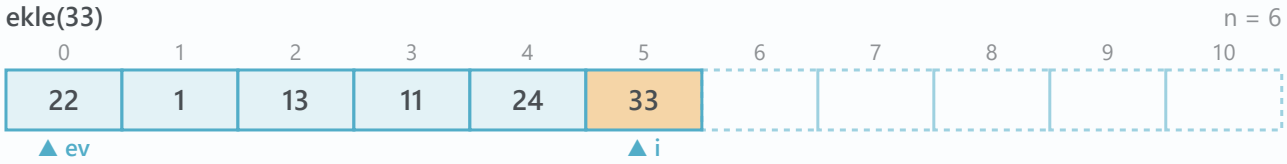
`k = 33` `i = 5` `d = 5`

`tablo[i].durum = BOS`

ÇAĞRI YIĞINI

- **`ekle(33)`**
- `main()`

Kod izleme: `ekle(33)` — doğrusal yoklama



`tablo[5] = {33, DOLU}` .

Adım 15 / 16

```
1  int ekle(int k) {
2    int i = k % M;                    /* ev adresi */
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
4      if (tablo[i].durum != DOLU) {   /* BOS/SILINDI */
▶        tablo[i].anahtar = k;
▶        tablo[i].durum = DOLU;
7        return i;
8      }
9    }
10   return -1;                        /* tablo dolu */
11 }
```

DEĞİŞKENLER

`k = 33` `i = 5` `d = 5`

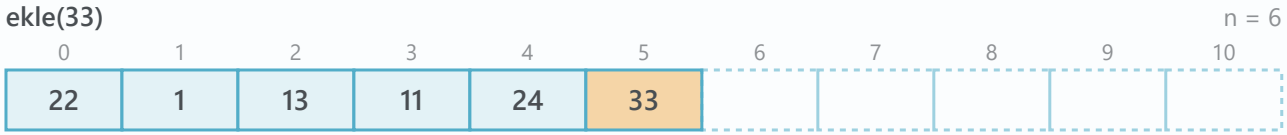
`tablo[i].durum = DOLU`

ÇAĞRI YIĞINI

`ekle(33)`

`main()`

Kod izleme: `ekle(33)` — doğrusal yoklama



return 5: 6 yoklama. Birincil kümelenme yüzünden ev adresinden 5 yuva uzağa düştü.

Adım 16 / 16

```
1  int ekle(int k) {
2    int i = k % M;                    /* ev adresi */
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
4      if (tablo[i].durum != DOLU) {   /* BOS/SILINDI */
5        tablo[i].anahtar = k;
6        tablo[i].durum = DOLU;
▶        return i;
8      }
9    }
10   return -1;                        /* tablo dolu */
11 }
```

DEĞİŞKENLER

`k = 33` `i = 5` `d = 5`

ÇAĞRI YIĞINI

`ekle(33)`

`main()`

Kod izleme: `ara(24)` ve `ara(44)`



ara(24) çağrılır.

Adım 1 / 14

```
▶ int ara(int k) {
2   int i = k % M;
3   for (int d = 0; d < M; d++, i = (i + 1) % M) {
4     if (tablo[i].durum == BOS) return -1;
5     if (tablo[i].durum == DOLU &&
6         tablo[i].anahtar == k) return i;
7   }
8   return -1;
9 }
10 bool sil(int k) {
11   int i = ara(k);
12   if (i < 0) return false;
13   tablo[i].durum = SILINDI;   /* silindi işareti */
14   return true;
15 }
```

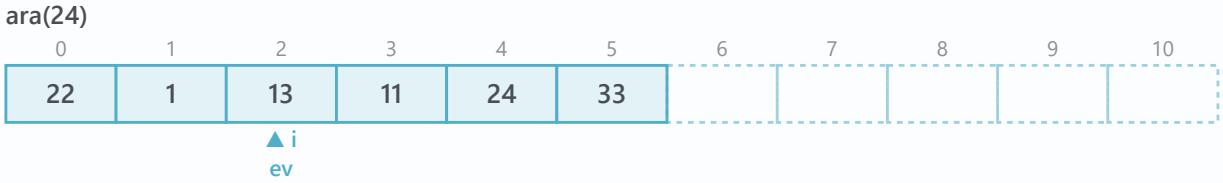
DEĞİŞKENLER

`k = 24`

ÇAĞRI YIĞINI

- **ara(24)**
- main()

Kod izleme: `ara(24)` ve `ara(44)`



Ev adresi i = 24 % 11 = **2**.

Adım 2 / 14

```
1  int ara(int k) {
▶    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
4      if (tablo[i].durum == BOS) return -1;
5      if (tablo[i].durum == DOLU &&
6          tablo[i].anahtar == k) return i;
7    }
8    return -1;
9  }
10 bool sil(int k) {
11   int i = ara(k);
12   if (i < 0) return false;
13   tablo[i].durum = SILINDI;   /* silindi işareti */
14   return true;
15 }
```

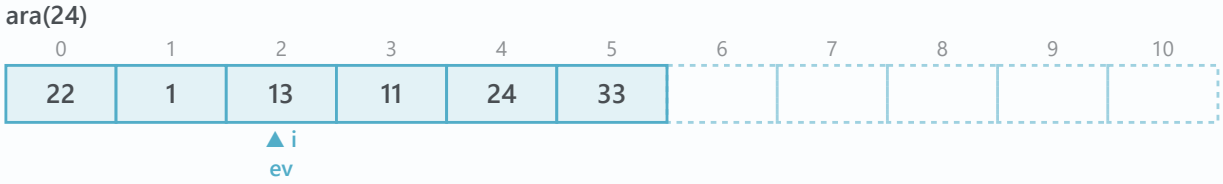
DEĞİŞKENLER

`k = 24` `i = 2`

ÇAĞRI YIĞINI

- **ara(24)**
- main()

Kod izleme: `ara(24)` ve `ara(44)`



Yuva 2: başka anahtar (13) → i = 3.

Adım 3 / 14

```
1  int ara(int k) {
2    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (tablo[i].durum == BOS) return -1;
▶      if (tablo[i].durum == DOLU &&
6          tablo[i].anahtar == k) return i;
7    }
8    return -1;
9  }
10 bool sil(int k) {
11   int i = ara(k);
12   if (i < 0) return false;
13   tablo[i].durum = SILINDI;   /* silindi işareti */
14   return true;
15 }
```

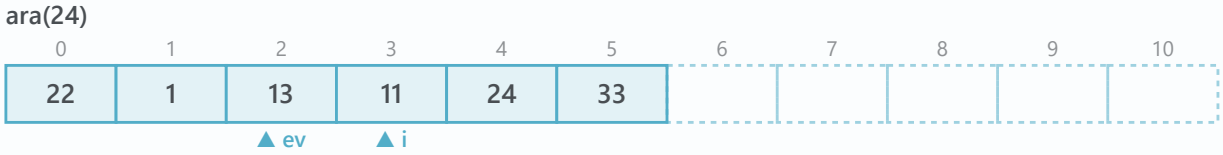
DEĞİŞKENLER

`k = 24` `i = 2` `d = 0` `durum = DOLU`

ÇAĞRI YIĞINI

- `ara(24)`
- `main()`

Kod izleme: `ara(24)` ve `ara(44)`



Yuva 3: başka anahtar (11) → i = 4.


```
1  int ara(int k) {
2    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (tablo[i].durum == BOS) return -1;
▶      if (tablo[i].durum == DOLU &&
6          tablo[i].anahtar == k) return i;
7    }
8    return -1;
9  }
10 bool sil(int k) {
11   int i = ara(k);
12   if (i < 0) return false;
13   tablo[i].durum = SILINDI;   /* silindi işareti */
14   return true;
15 }
```

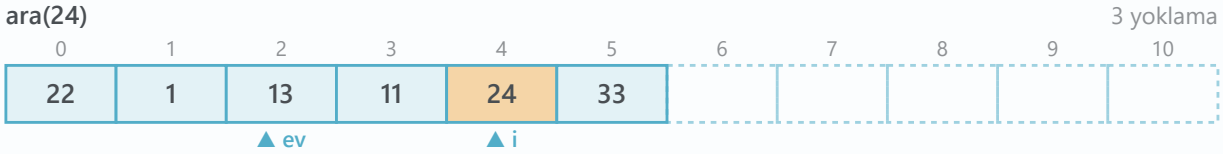
DEĞİŞKENLER

`k = 24` `i = 3` `d = 1` `durum = DOLU`

ÇAĞRI YIĞINI

- **ara(24)**
- main()

Kod izleme: `ara(24)` ve `ara(44)`



Yuva 4 DOLU ve anahtar 24 → **return 4** (3 yoklama).

Adım 5 / 14

```
 1 int ara(int k) {
 2   int i = k % M;
 3   for (int d = 0; d < M; d++, i = (i + 1) % M) {
 ▶     if (tablo[i].durum == BOS) return -1;
 ▶     if (tablo[i].durum == DOLU &&
 ▶         tablo[i].anahtar == k) return i;
 7   }
 8   return -1;
 9 }
10 bool sil(int k) {
11   int i = ara(k);
12   if (i < 0) return false;
13   tablo[i].durum = SILINDI;    /* silindi işareti */
14   return true;
15 }
```

DEĞİŞKENLER

`k = 24` `i = 4` `d = 2` `durum = DOLU`

ÇAĞRI YIĞINI

- **ara(24)**
- main()

Kod izleme: `ara(24)` ve `ara(44)`



ara(44) çağrılır.

Adım 6 / 14

```
▶ int ara(int k) {
2   int i = k % M;
3   for (int d = 0; d < M; d++, i = (i + 1) % M) {
4     if (tablo[i].durum == BOS) return -1;
5     if (tablo[i].durum == DOLU &&
6         tablo[i].anahtar == k) return i;
7   }
8   return -1;
9 }
10 bool sil(int k) {
11   int i = ara(k);
12   if (i < 0) return false;
13   tablo[i].durum = SILINDI;   /* silindi işareti */
14   return true;
15 }
```

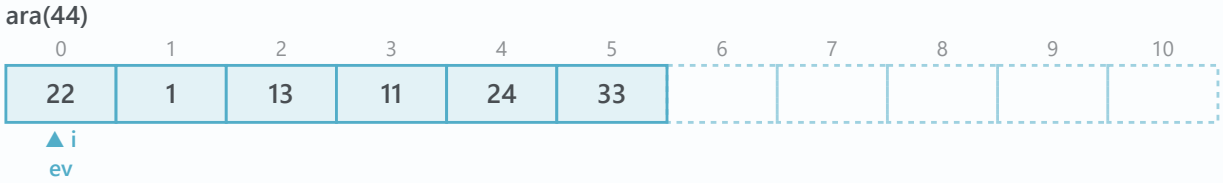
DEĞİŞKENLER

`k = 44`

ÇAĞRI YIĞINI

- **ara(44)**
- main()

Kod izleme: `ara(24)` ve `ara(44)`



Ev adresi i = 44 % 11 = **0**.

Adım 7 / 14

```
1  int ara(int k) {
▶    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
4      if (tablo[i].durum == BOS) return -1;
5      if (tablo[i].durum == DOLU &&
6          tablo[i].anahtar == k) return i;
7    }
8    return -1;
9  }
10 bool sil(int k) {
11   int i = ara(k);
12   if (i < 0) return false;
13   tablo[i].durum = SILINDI;   /* silindi işareti */
14   return true;
15 }
```

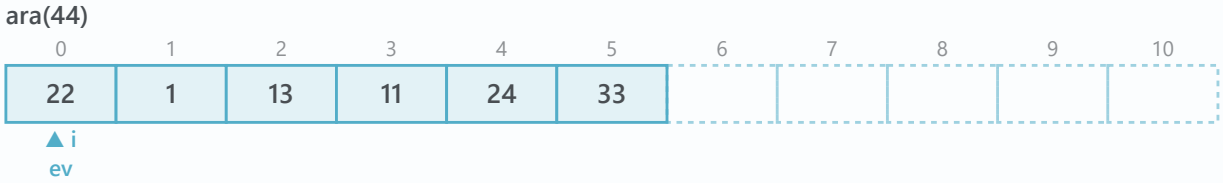
DEĞİŞKENLER

`k = 44` `i = 0`

ÇAĞRI YIĞINI

- **ara(44)**
- main()

Kod izleme: `ara(24)` ve `ara(44)`



Yuva 0: başka anahtar (22) → i = 1.


```
1  int ara(int k) {
2    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (tablo[i].durum == BOS) return -1;
▶      if (tablo[i].durum == DOLU &&
6          tablo[i].anahtar == k) return i;
7    }
8    return -1;
9  }
10 bool sil(int k) {
11   int i = ara(k);
12   if (i < 0) return false;
13   tablo[i].durum = SILINDI;   /* silindi işareti */
14   return true;
15 }
```

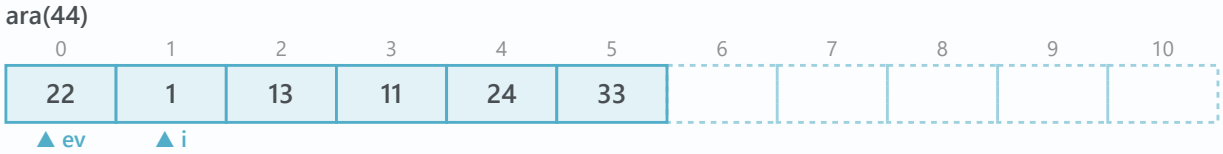
DEĞİŞKENLER

`k = 44` `i = 0` `d = 0` `durum = DOLU`

ÇAĞRI YIĞINI

- **ara(44)**
- main()

Kod izleme: `ara(24)` ve `ara(44)`



Yuva 1: başka anahtar (1) → i = 2.


```
1  int ara(int k) {
2    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (tablo[i].durum == BOS) return -1;
▶      if (tablo[i].durum == DOLU &&
6          tablo[i].anahtar == k) return i;
7    }
8    return -1;
9  }
10 bool sil(int k) {
11   int i = ara(k);
12   if (i < 0) return false;
13   tablo[i].durum = SILINDI;   /* silindi işareti */
14   return true;
15 }
```

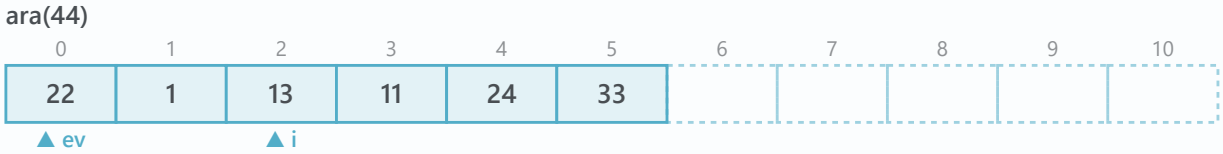
DEĞİŞKENLER

`k = 44` `i = 1` `d = 1` `durum = DOLU`

ÇAĞRI YIĞINI

- **ara(44)**
- main()

Kod izleme: `ara(24)` ve `ara(44)`



Yuva 2: başka anahtar (13) → i = 3.

Adım 10 / 14

```
1  int ara(int k) {
2    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (tablo[i].durum == BOS) return -1;
▶      if (tablo[i].durum == DOLU &&
6          tablo[i].anahtar == k) return i;
7    }
8    return -1;
9  }
10 bool sil(int k) {
11   int i = ara(k);
12   if (i < 0) return false;
13   tablo[i].durum = SILINDI;   /* silindi işareti */
14   return true;
15 }
```

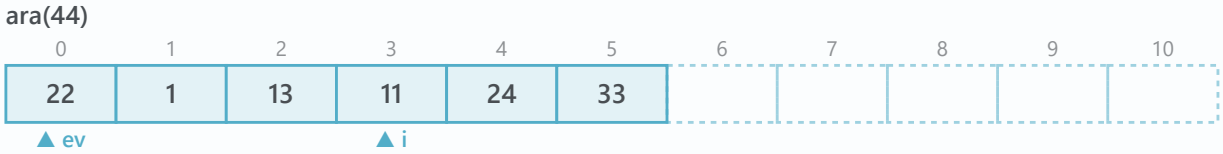
DEĞİŞKENLER

`k = 44` `i = 2` `d = 2` `durum = DOLU`

ÇAĞRI YIĞINI

- **ara(44)**
- main()

Kod izleme: `ara(24)` ve `ara(44)`



Yuva 3: başka anahtar (11) → i = 4.


```
1  int ara(int k) {
2    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (tablo[i].durum == BOS) return -1;
▶      if (tablo[i].durum == DOLU &&
6          tablo[i].anahtar == k) return i;
7    }
8    return -1;
9  }
10 bool sil(int k) {
11   int i = ara(k);
12   if (i < 0) return false;
13   tablo[i].durum = SILINDI;   /* silindi işareti */
14   return true;
15 }
```

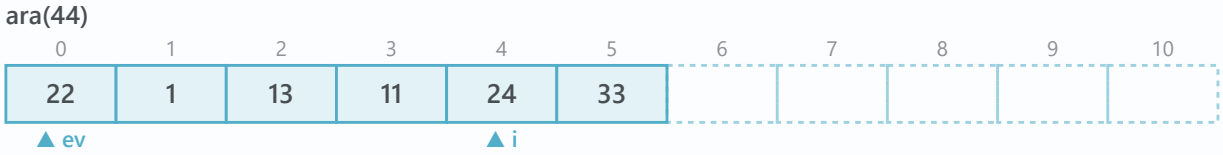
DEĞİŞKENLER

`k = 44` `i = 3` `d = 3` `durum = DOLU`

ÇAĞRI YIĞINI

- **ara(44)**
- main()

Kod izleme: `ara(24)` ve `ara(44)`



Yuva 4: başka anahtar (24) → i = 5.


```
1  int ara(int k) {
2    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (tablo[i].durum == BOS) return -1;
▶      if (tablo[i].durum == DOLU &&
6          tablo[i].anahtar == k) return i;
7    }
8    return -1;
9  }
10 bool sil(int k) {
11   int i = ara(k);
12   if (i < 0) return false;
13   tablo[i].durum = SILINDI;   /* silindi işareti */
14   return true;
15 }
```

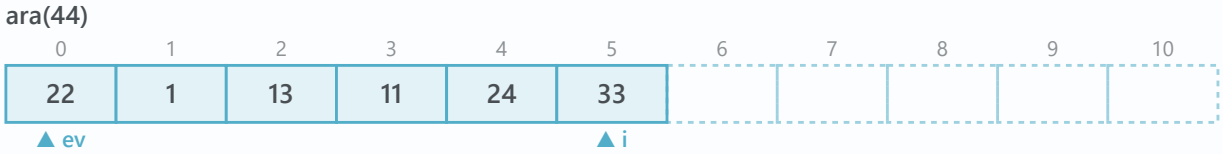
DEĞİŞKENLER

`k = 44` `i = 4` `d = 4` `durum = DOLU`

ÇAĞRI YIĞINI

- **ara(44)**
- main()

Kod izleme: `ara(24)` ve `ara(44)`



Yuva 5: başka anahtar (33) → i = 6.


```
1  int ara(int k) {
2    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (tablo[i].durum == BOS) return -1;
▶      if (tablo[i].durum == DOLU &&
6          tablo[i].anahtar == k) return i;
7    }
8    return -1;
9  }
10 bool sil(int k) {
11   int i = ara(k);
12   if (i < 0) return false;
13   tablo[i].durum = SILINDI;   /* silindi işareti */
14   return true;
15 }
```

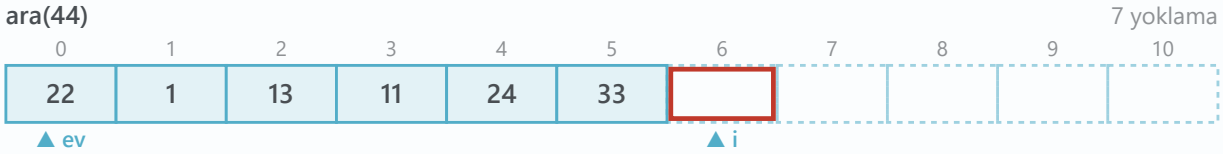
DEĞİŞKENLER

`k = 44` `i = 5` `d = 5` `durum = DOLU`

ÇAĞRI YIĞINI

- **ara(44)**
- main()

Kod izleme: `ara(24)` ve `ara(44)`



Yuva 6 **BOS**: 44 olsaydı en geç burada olurdu → **return −1** (yok).

Adım 14 / 14

```
1  int ara(int k) {
2    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (tablo[i].durum == BOS) return -1;
5      if (tablo[i].durum == DOLU &&
6          tablo[i].anahtar == k) return i;
7    }
8    return -1;
9  }
10 bool sil(int k) {
11   int i = ara(k);
12   if (i < 0) return false;
13   tablo[i].durum = SILINDI;   /* silindi işareti */
14   return true;
15 }
```

DEĞİŞKENLER

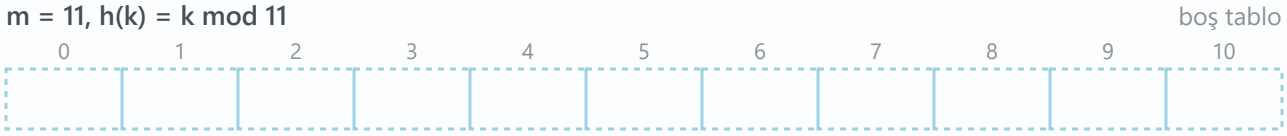
`k = 44` `i = 6` `d = 6` `durum = BOS`

ÇAĞRI YIĞINI

- `ara(44)`
- `main()`

AÇIK ADRESLEME

Karesel yoklama: $h(k, i) = (h(k) + i^2) \bmod m$

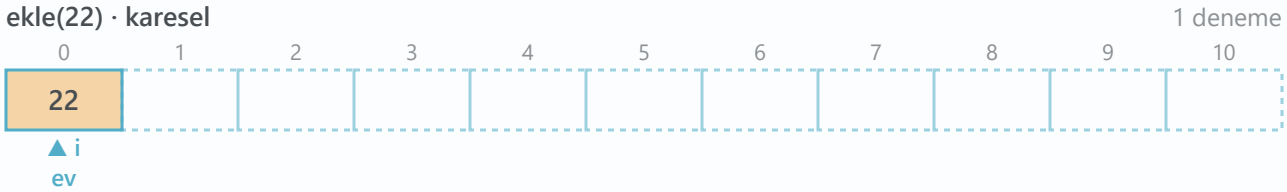


Aynı anahtarlar: 22, 1, 13, 11, 24, 33. Denemeler ev adresinden 1, 4, 9, 16… uzağa sıçrar.

Adım 1 / 13

Birincil kümelenme kalkar; ev adresi aynı olanlar yine aynı diziyi izler (**ikincil kümelenme**). m asal ve $\alpha \leq 1/2$ ise boş yuva bulunması garantidir.

Karesel yoklama: $h(k, i) = (h(k) + i^2) \bmod m$

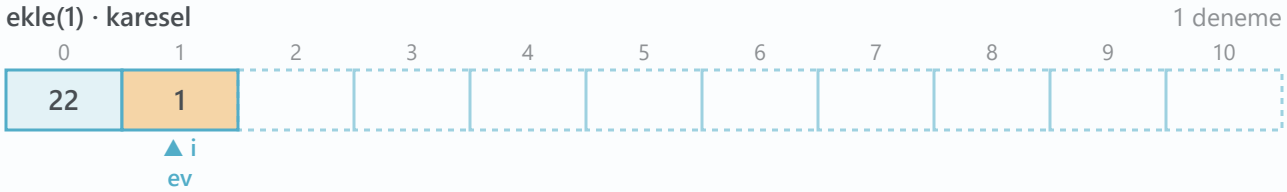


22 → ev 0: **yuva 0**.

Adım 2 / 13

Birincil kümelenme kalkar; ev adresi aynı olanlar yine aynı diziyi izler (**ikincil kümelenme**). m asal ve $\alpha \leq 1/2$ ise boş yuva bulunması garantidir.

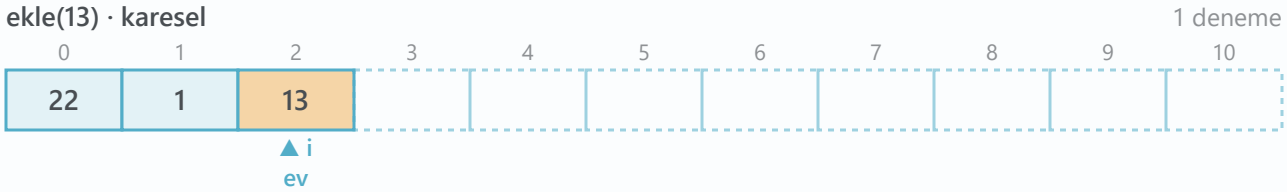
Karesel yoklama: $h(k, i) = (h(k) + i^2) \bmod m$



1 → ev 1: **yuva 1**.

Birincil kümelenme kalkar; ev adresi aynı olanlar yine aynı diziyi izler (**ikincil kümelenme**). m asal ve $\alpha \leq 1/2$ ise boş yuva bulunması garantidir.

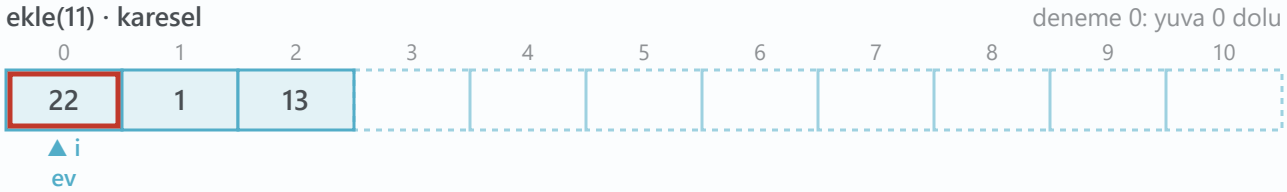
Karesel yoklama: $h(k, i) = (h(k) + i^2) \bmod m$



13 → ev 2: **yuva 2**.

Birincil kümelenme kalkar; ev adresi aynı olanlar yine aynı diziyi izler (**ikincil kümelenme**). m asal ve $\alpha \leq 1/2$ ise boş yuva bulunması garantidir.

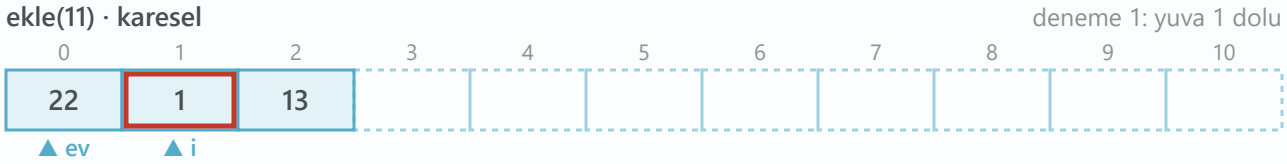
Karesel yoklama: $h(k, i) = (h(k) + i^2) \bmod m$



11: yuva **0** dolu (22) → sıradaki yuvaya bakılır.

Birincil kümelenme kalkar; ev adresi aynı olanlar yine aynı diziyi izler (**ikincil kümelenme**). m asal ve $\alpha \leq 1/2$ ise boş yuva bulunması garantidir.

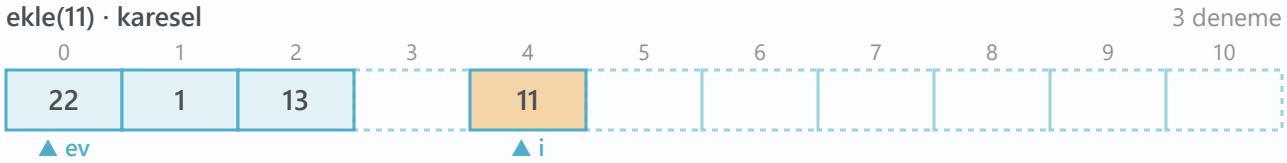
Karesel yoklama: $h(k, i) = (h(k) + i^2) \bmod m$



11: yuva **1** dolu (1) → sıradaki yuvaya bakılır.

Birincil kümelenme kalkar; ev adresi aynı olanlar yine aynı diziyi izler (**ikincil kümelenme**). m asal ve $\alpha \leq 1/2$ ise boş yuva bulunması garantidir.

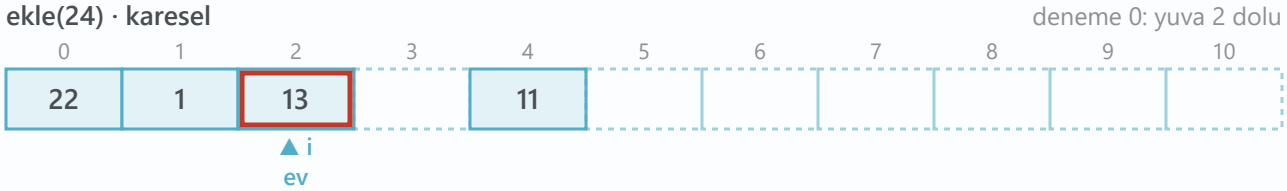
Karesel yoklama: $h(k, i) = (h(k) + i^2) \bmod m$



11 → ev 0, yoklama dizisi 0, 1, 4: **yuva 4**.

Birincil kümelenme kalkar; ev adresi aynı olanlar yine aynı diziyi izler (**ikincil kümelenme**). m asal ve $\alpha \leq 1/2$ ise boş yuva bulunması garantidir.

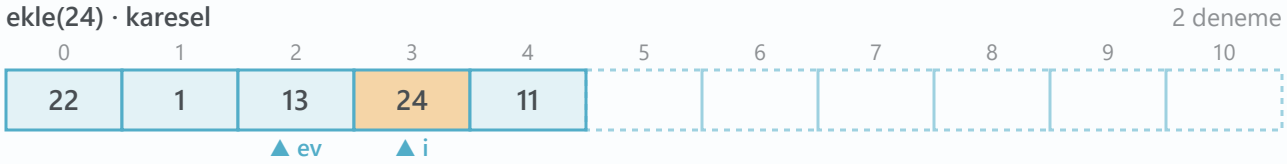
Karesel yoklama: $h(k, i) = (h(k) + i^2) \bmod m$



24: yuva **2** dolu (13) → sıradaki yuvaya bakılır.

Birincil kümelenme kalkar; ev adresi aynı olanlar yine aynı diziyi izler (**ikincil kümelenme**). m asal ve $\alpha \leq 1/2$ ise boş yuva bulunması garantidir.

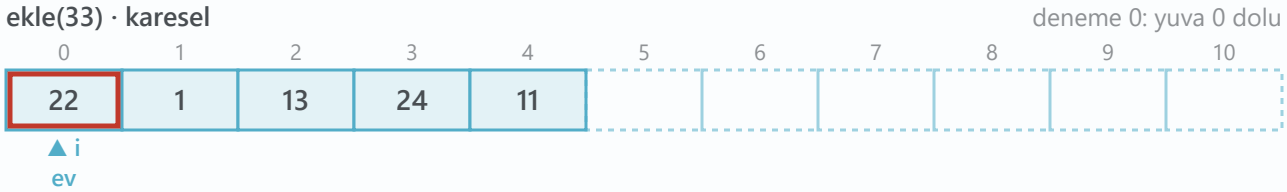
Karesel yoklama: $h(k, i) = (h(k) + i^2) \bmod m$



24 → ev 2, yoklama dizisi 2, 3: **yuva 3**.

Birincil kümelenme kalkar; ev adresi aynı olanlar yine aynı diziyi izler (**ikincil kümelenme**). m asal ve $\alpha \leq 1/2$ ise boş yuva bulunması garantidir.

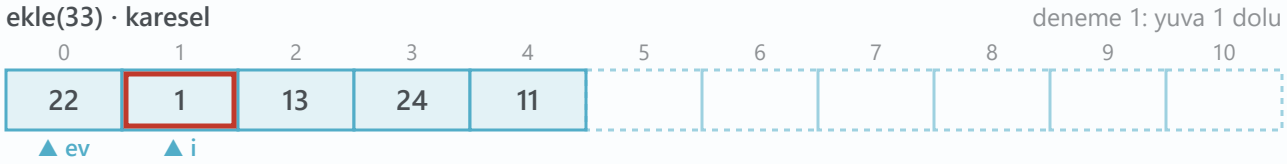
Karesel yoklama: $h(k, i) = (h(k) + i^2) \bmod m$



33: yuva **0** dolu (22) → sıradaki yuvaya bakılır.

Birincil kümelenme kalkar; ev adresi aynı olanlar yine aynı diziyi izler (**ikincil kümelenme**). m asal ve $\alpha \leq 1/2$ ise boş yuva bulunması garantidir.

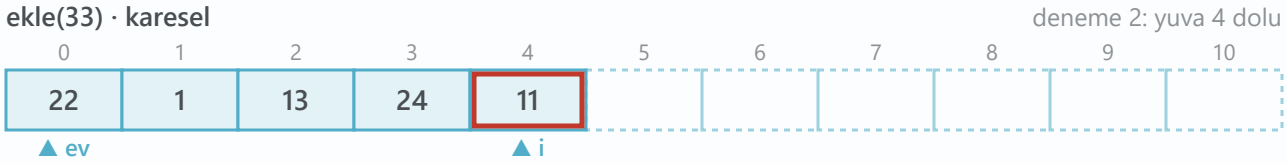
Karesel yoklama: $h(k, i) = (h(k) + i^2) \bmod m$



33: yuva **1** dolu (1) → sıradaki yuvaya bakılır.

Birincil kümelenme kalkar; ev adresi aynı olanlar yine aynı diziyi izler (**ikincil kümelenme**). m asal ve $\alpha \leq 1/2$ ise boş yuva bulunması garantidir.

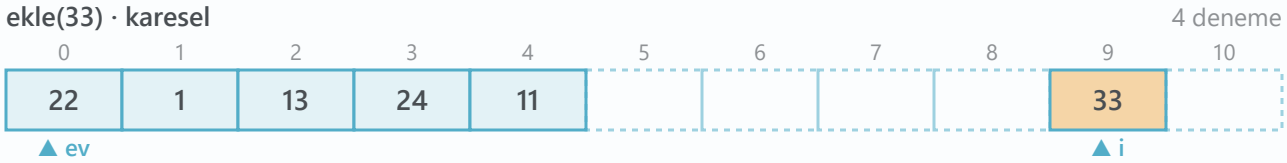
Karesel yoklama: $h(k, i) = (h(k) + i^2) \bmod m$



33: yuva **4** dolu (11) → sıradaki yuvaya bakılır.

Birincil kümelenme kalkar; ev adresi aynı olanlar yine aynı diziyi izler (**ikincil kümelenme**). m asal ve $\alpha \leq 1/2$ ise boş yuva bulunması garantidir.

Karesel yoklama: $h(k, i) = (h(k) + i^2) \bmod m$



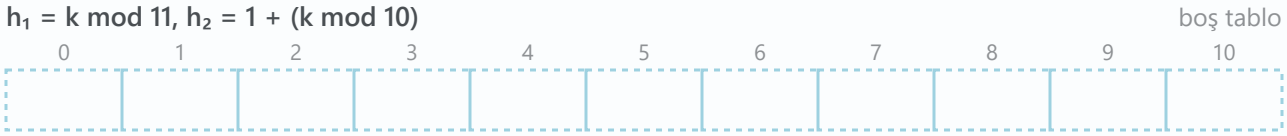
33 → ev 0, yoklama dizisi 0, 1, 4, 9: **yuva 9**.

Adım 13 / 13

Birincil kümelenme kalkar; ev adresi aynı olanlar yine aynı diziyi izler (**ikincil kümelenme**). m asal ve $\alpha \leq 1/2$ ise boş yuva bulunması garantidir.

AÇIK ADRESLEME

Çift hash: $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$

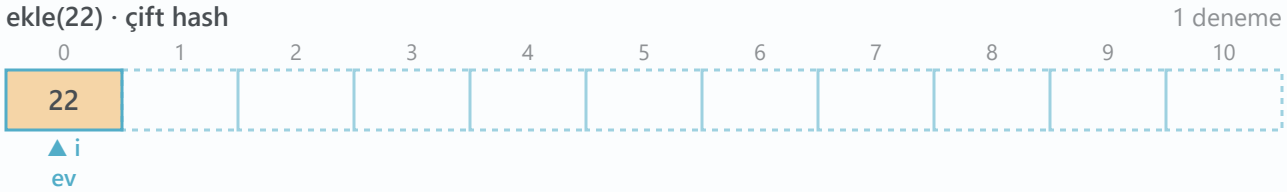


Adım boyu anahtara göre değişir: 11 → $h_2 = 2$, 24 → $h_2 = 5$, 33 → $h_2 = 4$.

Adım 1 / 12

$h_2(k)$ hiçbir zaman 0 olmamalı ve m ile aralarında asal olmalı (m asal seçilirse kendiliğinden sağlanır). Yoklama dizisi anahtara özgü olduğundan kümelenme en azdır.

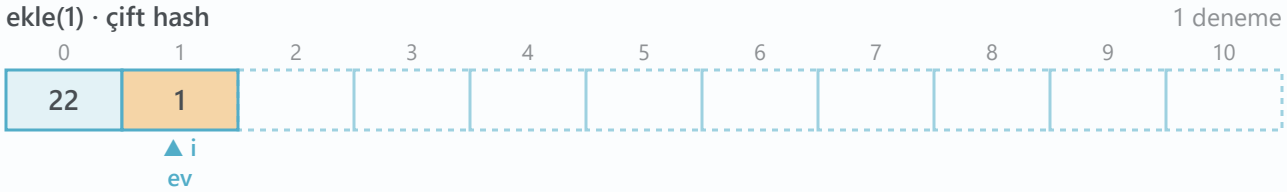
Çift hash: $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$



22 → ev 0: **yuva 0**.

$h_2(k)$ hiçbir zaman 0 olmamalı ve m ile aralarında asal olmalı (m asal seçilirse kendiliğinden sağlanır). Yoklama dizisi anahtara özgü olduğundan kümelenme en azdır.

Çift hash: $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$

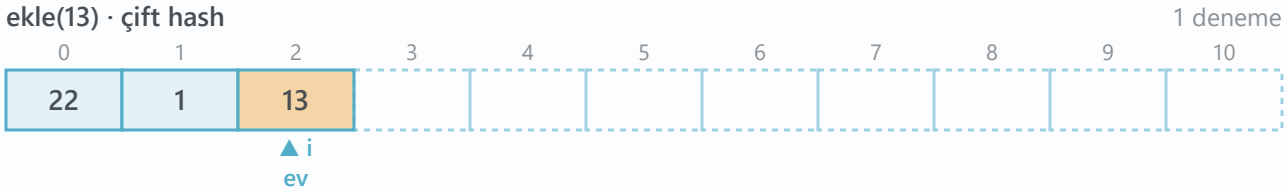


1 → ev 1: **yuva 1**.

Adım 3 / 12

$h_2(k)$ hiçbir zaman 0 olmamalı ve m ile aralarında asal olmalı (m asal seçilirse kendiliğinden sağlanır). Yoklama dizisi anahtara özgü olduğundan kümelenme en azdır.

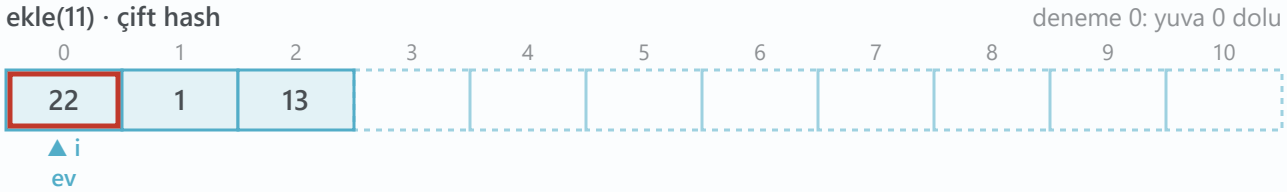
Çift hash: $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$



13 → ev 2: **yuva 2**.

$h_2(k)$ hiçbir zaman 0 olmamalı ve m ile aralarında asal olmalı (m asal seçilirse kendiliğinden sağlanır). Yoklama dizisi anahtara özgü olduğundan kümelenme en azdır.

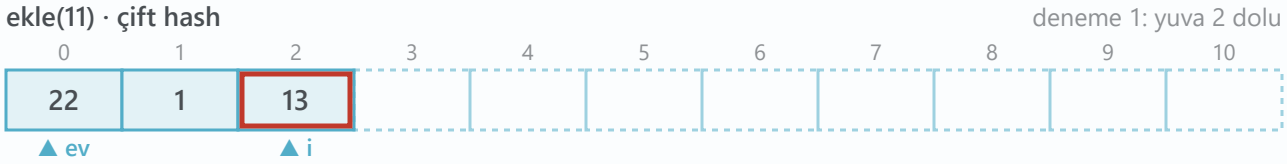
Çift hash: $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$



11: yuva **0** dolu (22) → sıradaki yuvaya bakılır.

$h_2(k)$ hiçbir zaman 0 olmamalı ve m ile aralarında asal olmalı (m asal seçilirse kendiliğinden sağlanır). Yoklama dizisi anahtara özgü olduğundan kümelenme en azdır.

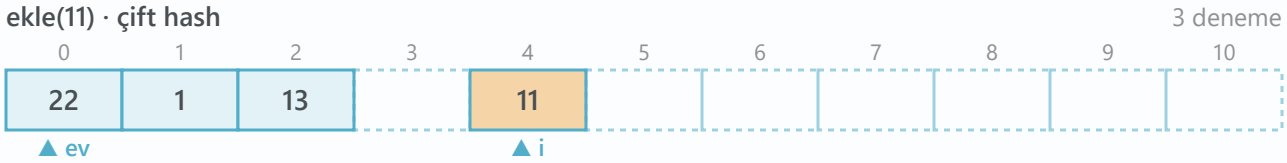
Çift hash: $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$



11: yuva **2** dolu (13) → sıradaki yuvaya bakılır.

$h_2(k)$ hiçbir zaman 0 olmamalı ve m ile aralarında asal olmalı (m asal seçilirse kendiliğinden sağlanır). Yoklama dizisi anahtara özgü olduğundan kümelenme en azdır.

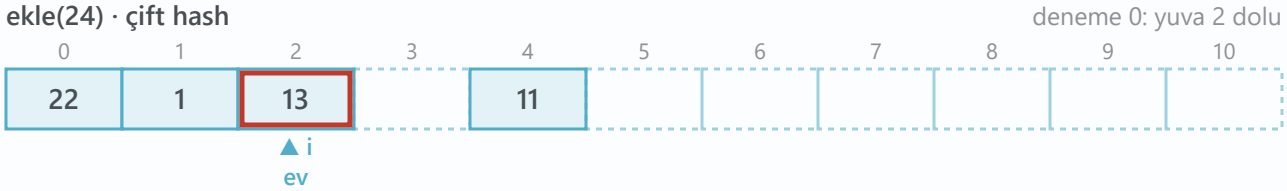
Çift hash: $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$



11 → ev 0, yoklama dizisi 0, 2, 4: **yuva 4**.

$h_2(k)$ hiçbir zaman 0 olmamalı ve m ile aralarında asal olmalı (m asal seçilirse kendiliğinden sağlanır). Yoklama dizisi anahtara özgü olduğundan kümelenme en azdır.

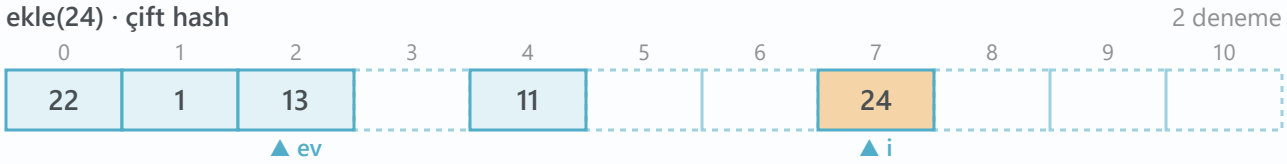
Çift hash: $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$



24: yuva **2** dolu (13) → sıradaki yuvaya bakılır.

$h_2(k)$ hiçbir zaman 0 olmamalı ve m ile aralarında asal olmalı (m asal seçilirse kendiliğinden sağlanır). Yoklama dizisi anahtara özgü olduğundan kümelenme en azdır.

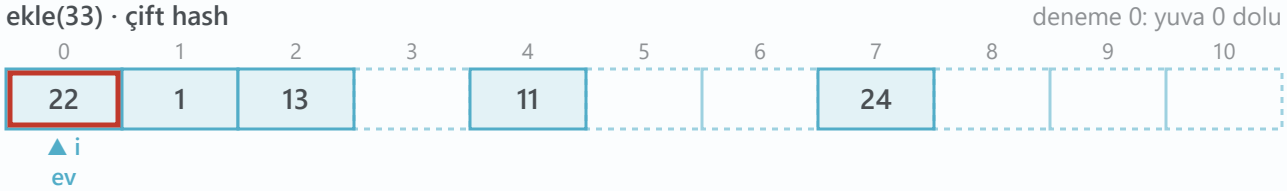
Çift hash: $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$



24 → ev 2, yoklama dizisi 2, 7: **yuva 7**.

$h_2(k)$ hiçbir zaman 0 olmamalı ve m ile aralarında asal olmalı (m asal seçilirse kendiliğinden sağlanır). Yoklama dizisi anahtara özgü olduğundan kümelenme en azdır.

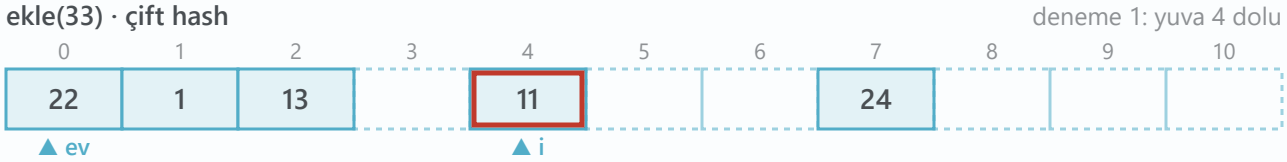
Çift hash: $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$



33: yuva **0** dolu (22) → sıradaki yuvaya bakılır.

$h_2(k)$ hiçbir zaman 0 olmamalı ve m ile aralarında asal olmalı (m asal seçilirse kendiliğinden sağlanır). Yoklama dizisi anahtara özgü olduğundan kümelenme en azdır.

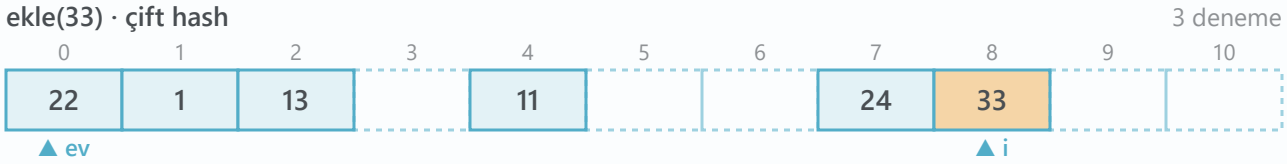
Çift hash: $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$



33: yuva **4** dolu (11) → sıradaki yuvaya bakılır.

$h_2(k)$ hiçbir zaman 0 olmamalı ve m ile aralarında asal olmalı (m asal seçilirse kendiliğinden sağlanır). Yoklama dizisi anahtara özgü olduğundan kümelenme en azdır.

Çift hash: $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$



33 → ev 0, yoklama dizisi 0, 4, 8: **yuva 8**.

Adım 12 / 12

$h_2(k)$ hiçbir zaman 0 olmamalı ve m ile aralarında asal olmalı (m asal seçilirse kendiliğinden sağlanır). Yoklama dizisi anahtara özgü olduğundan kümelenme en azdır.

Aynı tablo, üç yoklama: 33 nereye düşer?

Yöntem	i. deneme	33'ün yoklama dizisi	Deneme
Doğrusal	$h + i$	0, 1, 2, 3, 4, 5	6
Karesel	$h + i^2$	0, 1, 4, 9	4
Çift hash	$h_1 + i \cdot h_2$	0, 4, 8	3

Yöntem	Kümelenme	Farklı yoklama dizisi sayısı
Doğrusal	birincil (bloklar)	m
Karesel	ikincil (aynı ev adresi)	m
Çift hash	en az	$\approx m^2$

Doğrusal yoklama önbellek (cache) dostudur ve α küçükken çok hızlıdır; α büyüdükçe kümelenme yüzünden maliyet hızla artar.

Zincirleme (separate chaining): kova başına bir liste

```c
#define MZ 5
typedef struct dugum {
  int anahtar;
  struct dugum *sonraki;
} dugum;
dugum *kova[MZ];            /* hepsi NULL */
```

```c
bool zincir_ekle(int k) {
  int i = k % MZ;
  dugum *yeni = malloc(sizeof *yeni);
  if (yeni == NULL) return false;  /* bellek yok */
  yeni->anahtar = k;
  yeni->sonraki = kova[i];         /* başa ekle */
  kova[i] = yeni;
  return true;
}
```

Başa ekle

Liste sırası önemsiz: başa ekleme O(1). (Tekrar istenmiyorsa önce aranmalı.)

Arama

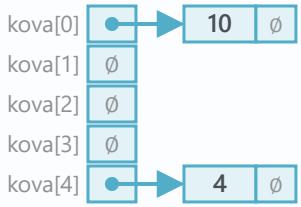
Yalnızca kendi kovasının listesinde gezinilir: ortalama 1 + α/2 düğüm.

Silme kolay

Bağlı listeden silmek yeter; silindi işareti gerekmez.

Kod izleme: `zincir_ekle` 12, 7, 22 (hepsi kova 2)

ekle(12)



zincir_ekle(12).

Adım 1 / 15

```
▶ bool zincir_ekle(int k) {
2   int i = k % MZ;
3   dugum *yeni = malloc(sizeof *yeni);
4   if (yeni == NULL) return false;  /* bellek yok */
5   yeni->anahtar = k;
6   yeni->sonraki = kova[i];         /* başa ekle */
7   kova[i] = yeni;
8   return true;
9 }
```

DEĞİŞKENLER

`k = 12`

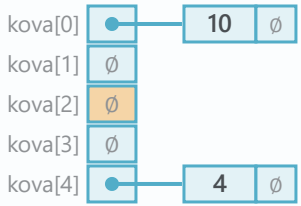
ÇAĞRI YIĞINI

`zincir_ekle(12)`

`main()`

Kod izleme: `zincir_ekle` 12, 7, 22 (hepsi kova 2)

ekle(12)



i = 12 % 5 = **2**.

Adım 2 / 15

```
1 bool zincir_ekle(int k) {
▶   int i = k % MZ;
3   dugum *yeni = malloc(sizeof *yeni);
4   if (yeni == NULL) return false;  /* bellek yok */
5   yeni->anahtar = k;
6   yeni->sonraki = kova[i];         /* başa ekle */
7   kova[i] = yeni;
8   return true;
9 }
```

DEĞİŞKENLER

`k = 12` `i = 2`

ÇAĞRI YIĞINI

`zincir_ekle(12)`

`main()`

Kod izleme: `zincir_ekle` 12, 7, 22 (hepsi kova 2)

ekle(12)



Yeni düğüm için bellek alınır; `NULL` değil, devam.

Adım 3 / 15

```
1 bool zincir_ekle(int k) {
2   int i = k % MZ;
▶   dugum *yeni = malloc(sizeof *yeni);
▶   if (yeni == NULL) return false;  /* bellek yok */
5   yeni->anahtar = k;
6   yeni->sonraki = kova[i];         /* başa ekle */
7   kova[i] = yeni;
8   return true;
9 }
```

DEĞİŞKENLER

`k = 12` `i = 2`

`yeni = &düğüm(?)`

ÇAĞRI YIĞINI

`zincir_ekle(12)`

`main()`

Kod izleme: `zincir_ekle` 12, 7, 22 (hepsi kova 2)



anahtar = 12; **sonraki = kova[2]** (NULL): yeni düğüm eski başı gösterir.

Adım 4 / 15

```
1 bool zincir_ekle(int k) {
2   int i = k % MZ;
3   dugum *yeni = malloc(sizeof *yeni);
4   if (yeni == NULL) return false;  /* bellek yok */
▶   yeni->anahtar = k;
▶   yeni->sonraki = kova[i];         /* başa ekle */
7   kova[i] = yeni;
8   return true;
9 }
```

DEĞİŞKENLER

`k = 12` `i = 2`

`yeni = &düğüm(12)`

`yeni→sonraki = NULL`

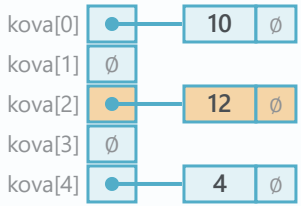
ÇAĞRI YIĞINI

`zincir_ekle(12)`

`main()`

Kod izleme: `zincir_ekle` 12, 7, 22 (hepsi kova 2)

ekle(12)



kova[2] = yeni: 12 listenin başında. O(1).

Adım 5 / 15

```
1 bool zincir_ekle(int k) {
2   int i = k % MZ;
3   dugum *yeni = malloc(sizeof *yeni);
4   if (yeni == NULL) return false;  /* bellek yok */
5   yeni->anahtar = k;
6   yeni->sonraki = kova[i];         /* başa ekle */
▶   kova[i] = yeni;
▶   return true;
9 }
```

DEĞİŞKENLER

`k = 12` `i = 2`

`yeni = &düğüm(12)`

`kova[i] = &düğüm(12)`

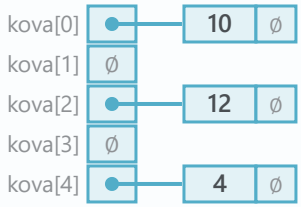
ÇAĞRI YIĞINI

`zincir_ekle(12)`

`main()`

Kod izleme: `zincir_ekle` 12, 7, 22 (hepsi kova 2)

ekle(7)



zincir_ekle(7).

Adım 6 / 15

```
▶ bool zincir_ekle(int k) {
2   int i = k % MZ;
3   dugum *yeni = malloc(sizeof *yeni);
4   if (yeni == NULL) return false;  /* bellek yok */
5   yeni->anahtar = k;
6   yeni->sonraki = kova[i];         /* başa ekle */
7   kova[i] = yeni;
8   return true;
9 }
```

DEĞİŞKENLER

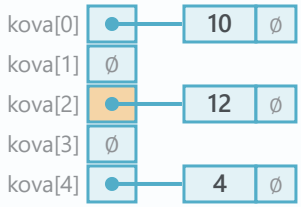
`k = 7`

ÇAĞRI YIĞINI

- `zincir_ekle(7)`
- `main()`

Kod izleme: `zincir_ekle` 12, 7, 22 (hepsi kova 2)

ekle(7)



i = 7 % 5 = **2**.

Adım 7 / 15

```
1 bool zincir_ekle(int k) {
▶   int i = k % MZ;
3   dugum *yeni = malloc(sizeof *yeni);
4   if (yeni == NULL) return false;  /* bellek yok */
5   yeni->anahtar = k;
6   yeni->sonraki = kova[i];         /* başa ekle */
7   kova[i] = yeni;
8   return true;
9 }
```

DEĞİŞKENLER

`k = 7` `i = 2`

ÇAĞRI YIĞINI

`zincir_ekle(7)`

`main()`

Kod izleme: `zincir_ekle` 12, 7, 22 (hepsi kova 2)

ekle(7)



Yeni düğüm için bellek alınır; `NULL` değil, devam.

Adım 8 / 15

```
1 bool zincir_ekle(int k) {
2   int i = k % MZ;
▶   dugum *yeni = malloc(sizeof *yeni);
▶   if (yeni == NULL) return false;  /* bellek yok */
5   yeni->anahtar = k;
6   yeni->sonraki = kova[i];         /* başa ekle */
7   kova[i] = yeni;
8   return true;
9 }
```

DEĞİŞKENLER

`k = 7` `i = 2` `yeni = &düğüm(?)`

ÇAĞRI YIĞINI

- `zincir_ekle(7)`
- `main()`

Kod izleme: `zincir_ekle` 12, 7, 22 (hepsi kova 2)



anahtar = 7; **sonraki = kova[2]** (&düğüm(12)): yeni düğüm eski başı gösterir.

Adım 9 / 15

```
1 bool zincir_ekle(int k) {
2   int i = k % MZ;
3   dugum *yeni = malloc(sizeof *yeni);
4   if (yeni == NULL) return false;  /* bellek yok */
▶   yeni->anahtar = k;
▶   yeni->sonraki = kova[i];         /* başa ekle */
7   kova[i] = yeni;
8   return true;
9 }
```

DEĞİŞKENLER

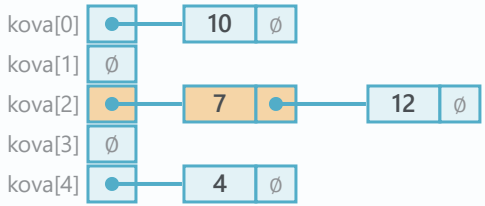
`k = 7` `i = 2` `yeni = &düğüm(7)` `yeni→sonraki = &düğüm(12)`

ÇAĞRI YIĞINI

- `zincir_ekle(7)`
- `main()`

Kod izleme: `zincir_ekle` 12, 7, 22 (hepsi kova 2)

ekle(7)



kova[2] = yeni: 7 listenin başında. O(1).

Adım 10 / 15

```
1 bool zincir_ekle(int k) {
2   int i = k % MZ;
3   dugum *yeni = malloc(sizeof *yeni);
4   if (yeni == NULL) return false;  /* bellek yok */
5   yeni->anahtar = k;
6   yeni->sonraki = kova[i];         /* başa ekle */
▶   kova[i] = yeni;
▶   return true;
9 }
```

DEĞİŞKENLER

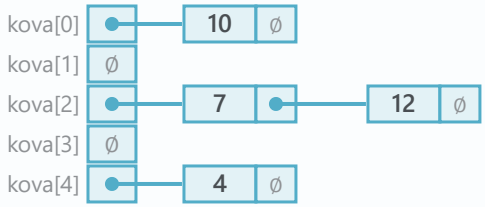
`k = 7` `i = 2` `yeni = &düğüm(7)` `kova[i] = &düğüm(7)`

ÇAĞRI YIĞINI

- `zincir_ekle(7)`
- `main()`

Kod izleme: `zincir_ekle` 12, 7, 22 (hepsi kova 2)

ekle(22)



zincir_ekle(22).

Adım 11 / 15

```
▶ bool zincir_ekle(int k) {
2   int i = k % MZ;
3   dugum *yeni = malloc(sizeof *yeni);
4   if (yeni == NULL) return false;  /* bellek yok */
5   yeni->anahtar = k;
6   yeni->sonraki = kova[i];         /* başa ekle */
7   kova[i] = yeni;
8   return true;
9 }
```

DEĞİŞKENLER

`k = 22`

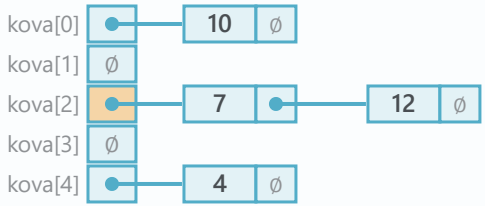
ÇAĞRI YIĞINI

`zincir_ekle(22)`

`main()`

Kod izleme: `zincir_ekle` 12, 7, 22 (hepsi kova 2)

ekle(22)



i = 22 % 5 = **2**.

Adım 12 / 15

```
1 bool zincir_ekle(int k) {
▶   int i = k % MZ;
3   dugum *yeni = malloc(sizeof *yeni);
4   if (yeni == NULL) return false;  /* bellek yok */
5   yeni->anahtar = k;
6   yeni->sonraki = kova[i];         /* başa ekle */
7   kova[i] = yeni;
8   return true;
9 }
```

DEĞİŞKENLER

`k = 22` `i = 2`

ÇAĞRI YIĞINI

`zincir_ekle(22)`

`main()`

Kod izleme: `zincir_ekle` 12, 7, 22 (hepsi kova 2)



Yeni düğüm için bellek alınır; `NULL` değil, devam.


```
1 bool zincir_ekle(int k) {
2   int i = k % MZ;
▶   dugum *yeni = malloc(sizeof *yeni);
▶   if (yeni == NULL) return false;  /* bellek yok */
5   yeni->anahtar = k;
6   yeni->sonraki = kova[i];         /* başa ekle */
7   kova[i] = yeni;
8   return true;
9 }
```

DEĞİŞKENLER

`k = 22` `i = 2`

`yeni = &düğüm(?)`

ÇAĞRI YIĞINI

`zincir_ekle(22)`

`main()`

Kod izleme: `zincir_ekle` 12, 7, 22 (hepsi kova 2)



anahtar = 22; **sonraki = kova[2]** (&düğüm(7)): yeni düğüm eski başı gösterir.


```
1 bool zincir_ekle(int k) {
2   int i = k % MZ;
3   dugum *yeni = malloc(sizeof *yeni);
4   if (yeni == NULL) return false;  /* bellek yok */
▶   yeni->anahtar = k;
▶   yeni->sonraki = kova[i];          /* başa ekle */
7   kova[i] = yeni;
8   return true;
9 }
```

DEĞİŞKENLER

`k = 22` `i = 2`

`yeni = &düğüm(22)`

`yeni→sonraki = &düğüm(7)`

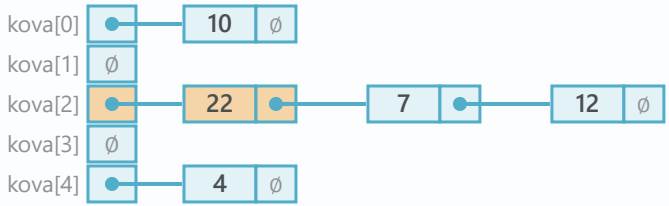
ÇAĞRI YIĞINI

`zincir_ekle(22)`

`main()`

Kod izleme: `zincir_ekle` 12, 7, 22 (hepsi kova 2)

ekle(22)



kova[2] = yeni: 22 listenin başında. O(1).

Adım 15 / 15

```
1 bool zincir_ekle(int k) {
2   int i = k % MZ;
3   dugum *yeni = malloc(sizeof *yeni);
4   if (yeni == NULL) return false;  /* bellek yok */
5   yeni->anahtar = k;
6   yeni->sonraki = kova[i];         /* başa ekle */
▶   kova[i] = yeni;
▶   return true;
9 }
```

DEĞİŞKENLER

`k = 22` `i = 2`

`yeni = &düğüm(22)`

`kova[i] = &düğüm(22)`

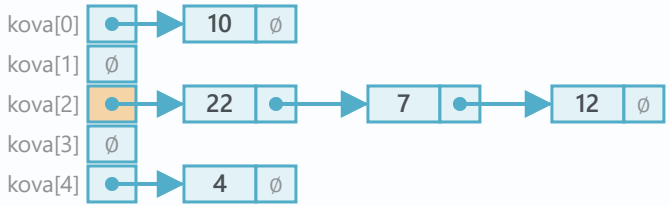
ÇAĞRI YIĞINI

`zincir_ekle(22)`

`main()`

Kod izleme: `zincir_ara(12)` ve `zincir_ara(17)`

ara(12)



zincir_ara(12): kova 12 % 5 = 2.

Adım 1 / 18

```
▶ dugum *zincir_ara(int k) {
2   dugum *p = kova[k % MZ];
3   while (p != NULL && p->anahtar != k)
4     p = p->sonraki;
5   return p;                  /* NULL: yok */
6 }
```

DEĞİŞKENLER

`k = 12`

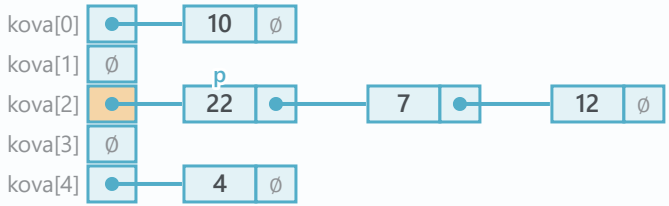
ÇAĞRI YIĞINI

`zincir_ara(12)`

`main()`

Kod izleme: `zincir_ara(12)` ve `zincir_ara(17)`

ara(12)



p = kova[2]: listenin başı (22).

Adım 2 / 18

```
1 dugum *zincir_ara(int k) {
▶   dugum *p = kova[k % MZ];
3   while (p != NULL && p->anahtar != k)
4     p = p->sonraki;
5   return p;                  /* NULL: yok */
6 }
```

DEĞİŞKENLER

`k = 12` `p = &düğüm(22)`

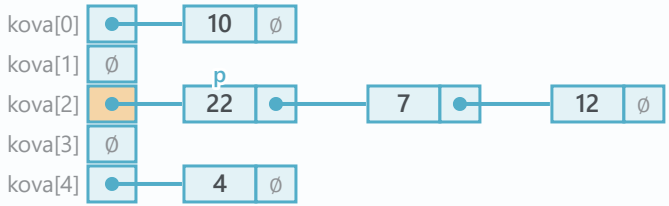
ÇAĞRI YIĞINI

`zincir_ara(12)`

`main()`

Kod izleme: `zincir_ara(12)` ve `zincir_ara(17)`

ara(12)



p->anahtar = 22 ≠ 12 → ilerlenir.

Adım 3 / 18

```
1 dugum *zincir_ara(int k) {
2   dugum *p = kova[k % MZ];
▶   while (p != NULL && p->anahtar != k)
4     p = p->sonraki;
5   return p;                  /* NULL: yok */
6 }
```

DEĞİŞKENLER

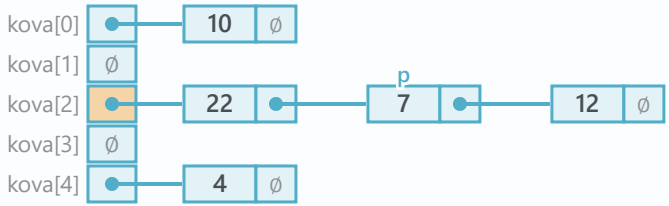
`k = 12` `p = &düğüm(22)`

ÇAĞRI YIĞINI

- `zincir_ara(12)`
- `main()`

Kod izleme: `zincir_ara(12)` ve `zincir_ara(17)`

ara(12)



p = p->sonraki → düğüm(7).

Adım 4 / 18

```
1 dugum *zincir_ara(int k) {
2   dugum *p = kova[k % MZ];
3   while (p != NULL && p->anahtar != k)
▶     p = p->sonraki;
5   return p;                  /* NULL: yok */
6 }
```

DEĞİŞKENLER

`k = 12` `p = &düğüm(7)`

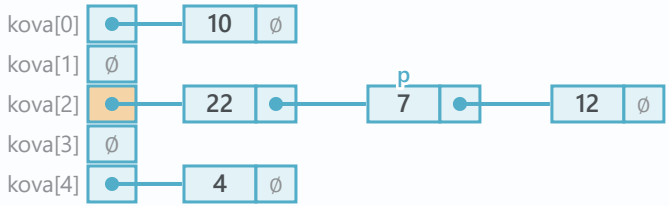
ÇAĞRI YIĞINI

`zincir_ara(12)`

`main()`

Kod izleme: `zincir_ara(12)` ve `zincir_ara(17)`

ara(12)



p->anahtar = 7 ≠ 12 → ilerlenir.

Adım 5 / 18

```
1 dugum *zincir_ara(int k) {
2   dugum *p = kova[k % MZ];
▶   while (p != NULL && p->anahtar != k)
4     p = p->sonraki;
5   return p;                  /* NULL: yok */
6 }
```

DEĞİŞKENLER

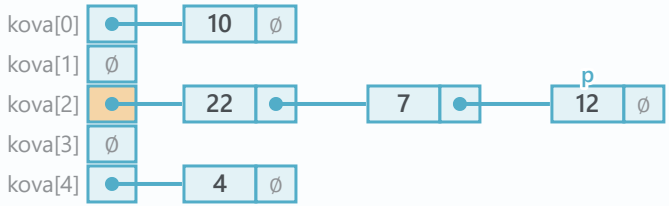
`k = 12` `p = &düğüm(7)`

ÇAĞRI YIĞINI

- `zincir_ara(12)`
- `main()`

Kod izleme: `zincir_ara(12)` ve `zincir_ara(17)`

ara(12)



p = p->sonraki → düğüm(12).

Adım 6 / 18

```
1 dugum *zincir_ara(int k) {
2   dugum *p = kova[k % MZ];
3   while (p != NULL && p->anahtar != k)
▶     p = p->sonraki;
5   return p;                  /* NULL: yok */
6 }
```

DEĞİŞKENLER

`k = 12` `p = &düğüm(12)`

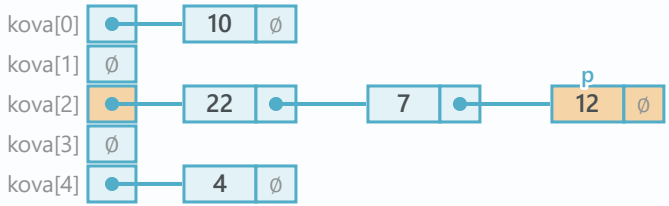
ÇAĞRI YIĞINI

`zincir_ara(12)`

`main()`

Kod izleme: `zincir_ara(12)` ve `zincir_ara(17)`

ara(12)



p->anahtar == 12: döngü koşulu yanlış, durulur.

Adım 7 / 18

```
1 dugum *zincir_ara(int k) {
2   dugum *p = kova[k % MZ];
▶   while (p != NULL && p->anahtar != k)
4     p = p->sonraki;
5   return p;                  /* NULL: yok */
6 }
```

DEĞİŞKENLER

`k = 12` `p = &düğüm(12)`

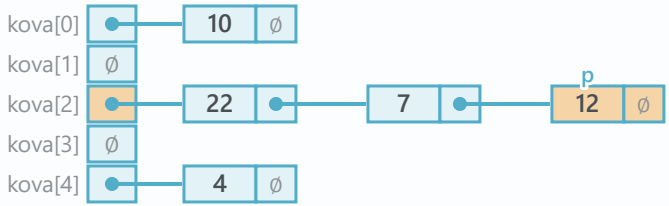
ÇAĞRI YIĞINI

`zincir_ara(12)`

`main()`

Kod izleme: `zincir_ara(12)` ve `zincir_ara(17)`

ara(12)



return p: bulundu (3 düğüm gezildi).

Adım 8 / 18

```
1 dugum *zincir_ara(int k) {
2   dugum *p = kova[k % MZ];
3   while (p != NULL && p->anahtar != k)
4     p = p->sonraki;
▶   return p;                  /* NULL: yok */
6 }
```

DEĞİŞKENLER

`k = 12` `p = &düğüm(12)`

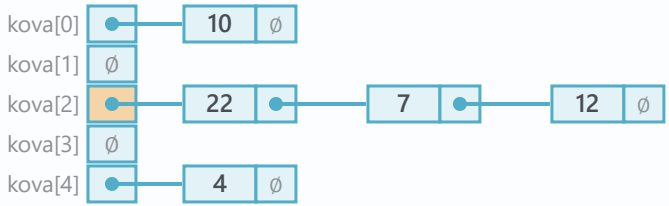
ÇAĞRI YIĞINI

`zincir_ara(12)`

`main()`

Kod izleme: `zincir_ara(12)` ve `zincir_ara(17)`

ara(17)



zincir_ara(17): kova 17 % 5 = 2.

Adım 9 / 18

```
▶ dugum *zincir_ara(int k) {
2   dugum *p = kova[k % MZ];
3   while (p != NULL && p->anahtar != k)
4     p = p->sonraki;
5   return p;                  /* NULL: yok */
6 }
```

DEĞİŞKENLER

`k = 17`

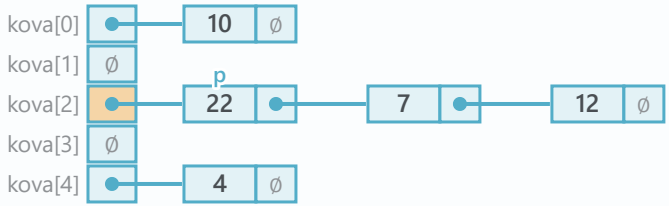
ÇAĞRI YIĞINI

`zincir_ara(17)`

`main()`

Kod izleme: `zincir_ara(12)` ve `zincir_ara(17)`

ara(17)



p = kova[2]: listenin başı (22).

Adım 10 / 18

```
1 dugum *zincir_ara(int k) {
▶   dugum *p = kova[k % MZ];
3   while (p != NULL && p->anahtar != k)
4     p = p->sonraki;
5   return p;                  /* NULL: yok */
6 }
```

DEĞİŞKENLER

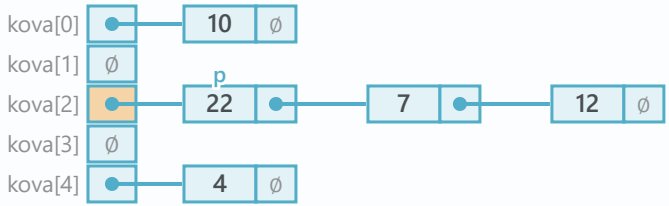
`k = 17` `p = &düğüm(22)`

ÇAĞRI YIĞINI

- `zincir_ara(17)`
- `main()`

Kod izleme: `zincir_ara(12)` ve `zincir_ara(17)`

ara(17)



p->anahtar = 22 ≠ 17 → ilerlenir.

Adım 11 / 18

```
1 dugum *zincir_ara(int k) {
2   dugum *p = kova[k % MZ];
▶   while (p != NULL && p->anahtar != k)
4     p = p->sonraki;
5   return p;                  /* NULL: yok */
6 }
```

DEĞİŞKENLER

`k = 17` `p = &düğüm(22)`

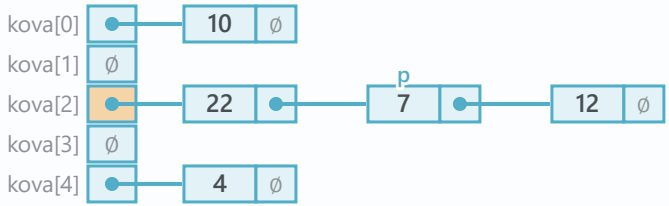
ÇAĞRI YIĞINI

`zincir_ara(17)`

`main()`

Kod izleme: `zincir_ara(12)` ve `zincir_ara(17)`

ara(17)



p = p->sonraki → düğüm(7).

Adım 12 / 18

```
1 dugum *zincir_ara(int k) {
2   dugum *p = kova[k % MZ];
3   while (p != NULL && p->anahtar != k)
▶     p = p->sonraki;
5   return p;                  /* NULL: yok */
6 }
```

DEĞİŞKENLER

`k = 17` `p = &düğüm(7)`

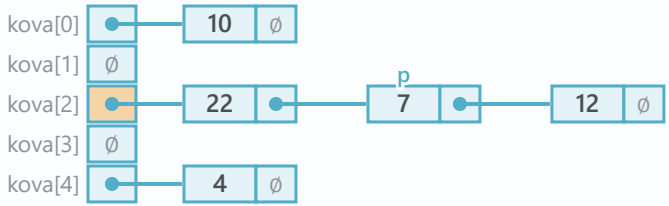
ÇAĞRI YIĞINI

`zincir_ara(17)`

`main()`

Kod izleme: `zincir_ara(12)` ve `zincir_ara(17)`

ara(17)



p->anahtar = 7 ≠ 17 → ilerlenir.

Adım 13 / 18

```
1 dugum *zincir_ara(int k) {
2   dugum *p = kova[k % MZ];
▶   while (p != NULL && p->anahtar != k)
4     p = p->sonraki;
5   return p;                  /* NULL: yok */
6 }
```

DEĞİŞKENLER

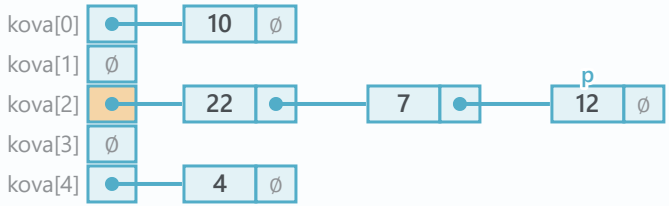
`k = 17` `p = &düğüm(7)`

ÇAĞRI YIĞINI

- `zincir_ara(17)`
- `main()`

Kod izleme: `zincir_ara(12)` ve `zincir_ara(17)`

ara(17)



p = p->sonraki → düğüm(12).

Adım 14 / 18

```
1 dugum *zincir_ara(int k) {
2   dugum *p = kova[k % MZ];
3   while (p != NULL && p->anahtar != k)
▶     p = p->sonraki;
5   return p;                  /* NULL: yok */
6 }
```

DEĞİŞKENLER

`k = 17` `p = &düğüm(12)`

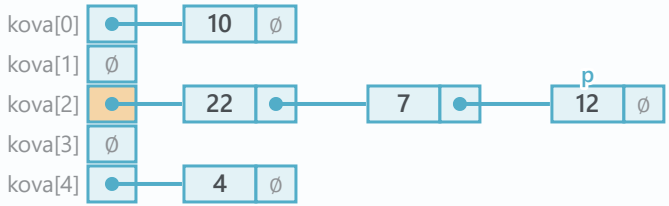
ÇAĞRI YIĞINI

`zincir_ara(17)`

`main()`

Kod izleme: `zincir_ara(12)` ve `zincir_ara(17)`

ara(17)



p->anahtar = 12 ≠ 17 → ilerlenir.

Adım 15 / 18

```
1 dugum *zincir_ara(int k) {
2   dugum *p = kova[k % MZ];
▶   while (p != NULL && p->anahtar != k)
4     p = p->sonraki;
5   return p;                  /* NULL: yok */
6 }
```

DEĞİŞKENLER

`k = 17` `p = &düğüm(12)`

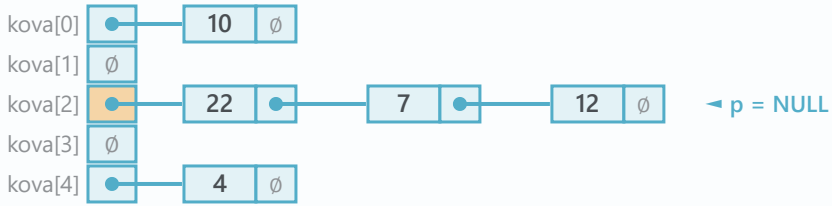
ÇAĞRI YIĞINI

`zincir_ara(17)`

`main()`

Kod izleme: `zincir_ara(12)` ve `zincir_ara(17)`

ara(17)



p = p->sonraki → NULL.

Adım 16 / 18

```
1 dugum *zincir_ara(int k) {
2   dugum *p = kova[k % MZ];
3   while (p != NULL && p->anahtar != k)
▶     p = p->sonraki;
5   return p;                  /* NULL: yok */
6 }
```

DEĞİŞKENLER

`k = 17` `p = NULL`

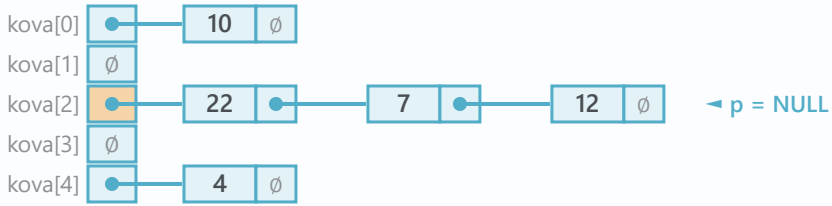
ÇAĞRI YIĞINI

`zincir_ara(17)`

`main()`

Kod izleme: `zincir_ara(12)` ve `zincir_ara(17)`

ara(17)



p == NULL: liste bitti, döngüden çıkılır.

Adım 17 / 18

```
1 dugum *zincir_ara(int k) {
2   dugum *p = kova[k % MZ];
▶   while (p != NULL && p->anahtar != k)
4     p = p->sonraki;
5   return p;                  /* NULL: yok */
6 }
```

DEĞİŞKENLER

`k = 17` `p = NULL`

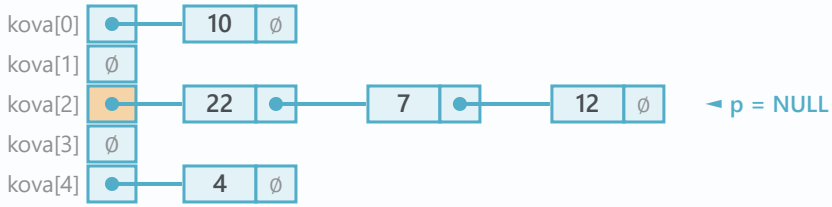
ÇAĞRI YIĞINI

`zincir_ara(17)`

`main()`

Kod izleme: `zincir_ara(12)` ve `zincir_ara(17)`

ara(17)



return NULL: 17 tabloda yok. Tüm zincir (3 düğüm) tarandı.

Adım 18 / 18

```
1 dugum *zincir_ara(int k) {
2   dugum *p = kova[k % MZ];
3   while (p != NULL && p->anahtar != k)
4     p = p->sonraki;
▶   return p;                  /* NULL: yok */
6 }
```

DEĞİŞKENLER

`k = 17` `p = NULL`

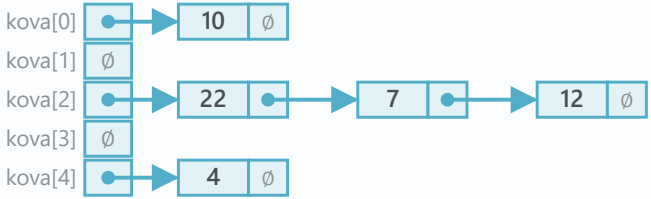
ÇAĞRI YIĞINI

`zincir_ara(17)`

`main()`

Kod izleme: `zincir_sil(7)`

sil(7)



zincir_sil(7): 7 zincirin ortasında (22 → 7 → 12).

Adım 1 / 10

```
▶ bool zincir_sil(int k) {
2   int i = k % MZ;
3   dugum *p = kova[i], *onceki = NULL;
4   while (p != NULL && p->anahtar != k) {
5     onceki = p;
6     p = p->sonraki;
7   }
8   if (p == NULL) return false;          /* yok */
9   if (onceki == NULL) kova[i] = p->sonraki;
10  else onceki->sonraki = p->sonraki;
11  free(p);
12  return true;
13 }
```

DEĞİŞKENLER

`k = 7`

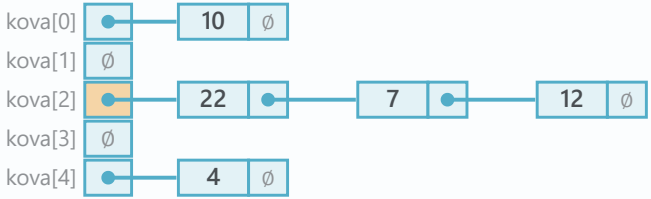
ÇAĞRI YIĞINI

`zincir_sil(7)`

`main()`

Kod izleme: `zincir_sil(7)`

sil(7)



i = 7 % 5 = **2**.

Adım 2 / 10

```
1  bool zincir_sil(int k) {
▶    int i = k % MZ;
3    dugum *p = kova[i], *onceki = NULL;
4    while (p != NULL && p->anahtar != k) {
5      onceki = p;
6      p = p->sonraki;
7    }
8    if (p == NULL) return false;           /* yok */
9    if (onceki == NULL) kova[i] = p->sonraki;
10   else onceki->sonraki = p->sonraki;
11   free(p);
12   return true;
13 }
```

DEĞİŞKENLER

`k = 7` `i = 2`

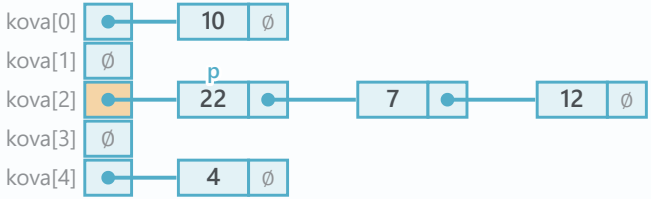
ÇAĞRI YIĞINI

`zincir_sil(7)`

`main()`

Kod izleme: `zincir_sil(7)`

sil(7)



p listenin başında, onceki = NULL.

Adım 3 / 10

```
1  bool zincir_sil(int k) {
2    int i = k % MZ;
▶    dugum *p = kova[i], *onceki = NULL;
4    while (p != NULL && p->anahtar != k) {
5      onceki = p;
6      p = p->sonraki;
7    }
8    if (p == NULL) return false;          /* yok */
9    if (onceki == NULL) kova[i] = p->sonraki;
10   else onceki->sonraki = p->sonraki;
11   free(p);
12   return true;
13 }
```

DEĞİŞKENLER

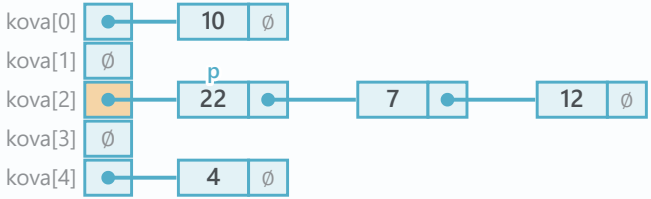
`k = 7` `i = 2` `p = &düğüm(22)` `onceki = NULL`

ÇAĞRI YIĞINI

- `zincir_sil(7)`
- `main()`

Kod izleme: `zincir_sil(7)`

sil(7)



22 ≠ 7 → döngüye girilir.

Adım 4 / 10

```
1  bool zincir_sil(int k) {
2    int i = k % MZ;
3    dugum *p = kova[i], *onceki = NULL;
▶    while (p != NULL && p->anahtar != k) {
5      onceki = p;
6      p = p->sonraki;
7    }
8    if (p == NULL) return false;           /* yok */
9    if (onceki == NULL) kova[i] = p->sonraki;
10   else onceki->sonraki = p->sonraki;
11   free(p);
12   return true;
13 }
```

DEĞİŞKENLER

`k = 7` `i = 2` `p = &düğüm(22)` `onceki = NULL`

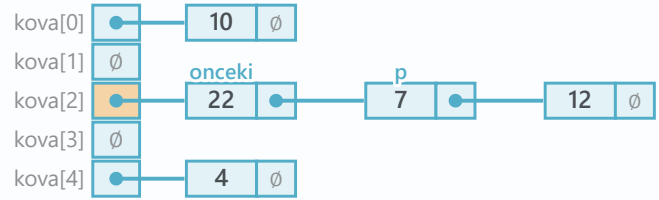
ÇAĞRI YIĞINI

`zincir_sil(7)`

`main()`

Kod izleme: `zincir_sil(7)`

sil(7)



onceki = 22'nin düğümü, p = 7'nin düğümü.

Adım 5 / 10

```
1  bool zincir_sil(int k) {
2    int i = k % MZ;
3    dugum *p = kova[i], *onceki = NULL;
4    while (p != NULL && p->anahtar != k) {
▶      onceki = p;
▶      p = p->sonraki;
7    }
8    if (p == NULL) return false;          /* yok */
9    if (onceki == NULL) kova[i] = p->sonraki;
10   else onceki->sonraki = p->sonraki;
11   free(p);
12   return true;
13 }
```

DEĞİŞKENLER

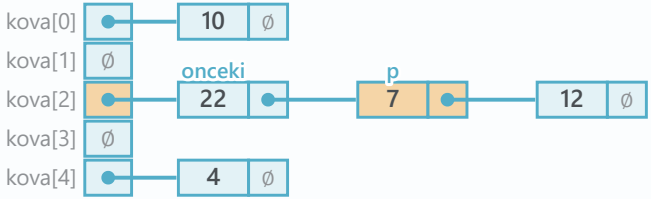
`k = 7` `i = 2` `p = &düğüm(7)` `onceki = &düğüm(22)`

ÇAĞRI YIĞINI

- `zincir_sil(7)`
- `main()`

Kod izleme: `zincir_sil(7)`

sil(7)



p->anahtar == 7 → döngü biter.

Adım 6 / 10

```
1  bool zincir_sil(int k) {
2    int i = k % MZ;
3    dugum *p = kova[i], *onceki = NULL;
▶5   while (p != NULL && p->anahtar != k) {
5      onceki = p;
6      p = p->sonraki;
7    }
8    if (p == NULL) return false;          /* yok */
9    if (onceki == NULL) kova[i] = p->sonraki;
10   else onceki->sonraki = p->sonraki;
11   free(p);
12   return true;
13 }
```

DEĞİŞKENLER

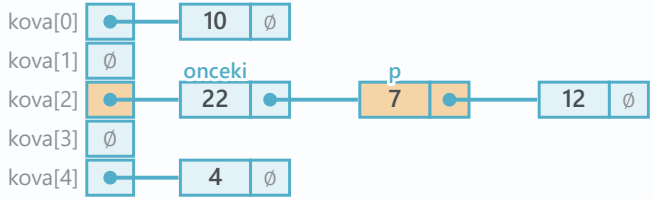
`k = 7` `i = 2` `p = &düğüm(7)` `onceki = &düğüm(22)`

ÇAĞRI YIĞINI

- `zincir_sil(7)`
- `main()`

Kod izleme: `zincir_sil(7)`

sil(7)



p NULL değil: anahtar bulundu.

Adım 7 / 10

```
 1 bool zincir_sil(int k) {
 2   int i = k % MZ;
 3   dugum *p = kova[i], *onceki = NULL;
 4   while (p != NULL && p->anahtar != k) {
 5     onceki = p;
 6     p = p->sonraki;
 7   }
▶8   if (p == NULL) return false;          /* yok */
 9   if (onceki == NULL) kova[i] = p->sonraki;
10   else onceki->sonraki = p->sonraki;
11   free(p);
12   return true;
13 }
```

DEĞİŞKENLER

`k = 7` `i = 2` `p = &düğüm(7)` `onceki = &düğüm(22)`

ÇAĞRI YIĞINI

- `zincir_sil(7)`
- `main()`

Kod izleme: `zincir_sil(7)`



onceki NULL değil → **onceki->sonraki = p->sonraki**: 22 artık doğrudan 12'yi gösterir; 7 zincirden çıktı.

Adım 8 / 10

```
1  bool zincir_sil(int k) {
2    int i = k % MZ;
3    dugum *p = kova[i], *onceki = NULL;
4    while (p != NULL && p->anahtar != k) {
5      onceki = p;
6      p = p->sonraki;
7    }
8    if (p == NULL) return false;          /* yok */
▶    if (onceki == NULL) kova[i] = p->sonraki;
▶    else onceki->sonraki = p->sonraki;
11   free(p);
12   return true;
13 }
```

DEĞİŞKENLER

`k = 7` `i = 2` `p = &düğüm(7)` `onceki = &düğüm(22)` `onceki→sonraki = &düğüm(12)`

ÇAĞRI YIĞINI

- `zincir_sil(7)`
- `main()`

Kod izleme: `zincir_sil(7)`



free(p): düğümün belleği geri verilir. Bundan sonra p kullanılmamalı (sarkan işaretçi).

Adım 9 / 10

```
 1 bool zincir_sil(int k) {
 2   int i = k % MZ;
 3   dugum *p = kova[i], *onceki = NULL;
 4   while (p != NULL && p->anahtar != k) {
 5     onceki = p;
 6     p = p->sonraki;
 7   }
 8   if (p == NULL) return false;          /* yok */
 9   if (onceki == NULL) kova[i] = p->sonraki;
10   else onceki->sonraki = p->sonraki;
▶    free(p);
12   return true;
13 }
```

DEĞİŞKENLER

`k = 7` `i = 2` `p = (serbest)`

`onceki = &düğüm(22)`

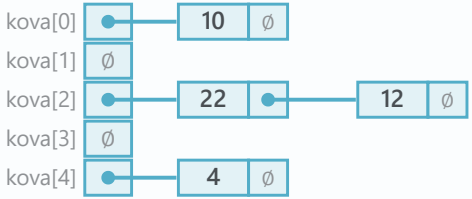
ÇAĞRI YIĞINI

`zincir_sil(7)`

`main()`

Kod izleme: `zincir_sil(7)`

sil(7)



return true. Baştaki düğüm silinseydi (onceki == NULL) `kova[i]` güncellenirdi.

Adım 10 / 10

```
1  bool zincir_sil(int k) {
2    int i = k % MZ;
3    dugum *p = kova[i], *onceki = NULL;
4    while (p != NULL && p->anahtar != k) {
5      onceki = p;
6      p = p->sonraki;
7    }
8    if (p == NULL) return false;           /* yok */
9    if (onceki == NULL) kova[i] = p->sonraki;
10   else onceki->sonraki = p->sonraki;
11   free(p);
▶    return true;
13 }
```

DEĞİŞKENLER

`k = 7` `i = 2`

ÇAĞRI YIĞINI

- **`zincir_sil(7)`**
- `main()`

Silmede tuzak: yuvayı boşaltmak aramayı keser



13'ü sileceğiz. 24'ün ev adresi 2; 24, 2 ve 3 dolu olduğu için 4'e yerleşmişti.

Adım 1 / 5

Çok silme yapılan tablolarda silindi işaretleri birikir ve aramalar uzar: periyodik olarak yeniden hash'lemek (temizlik) gerekir. Zincirlemede bu sorun yoktur.

YÜK FAKTÖRÜ VE SİLME

Silmede tuzak: yuvayı boşaltmak aramayı keser

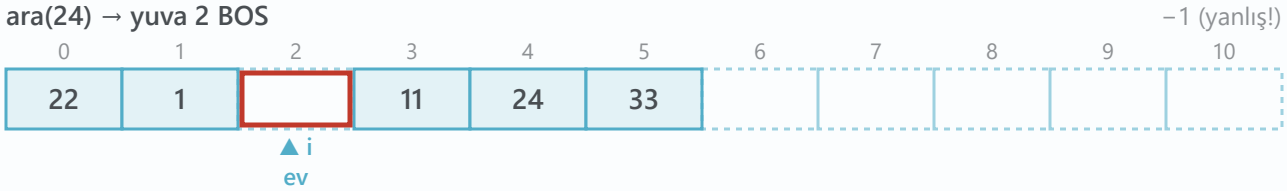


Yuvayı **BOS** yaparsak…

Adım 2 / 5

Çok silme yapılan tablolarda silindi işaretleri birikir ve aramalar uzar: periyodik olarak yeniden hash'lemek (temizlik) gerekir. Zincirlemede bu sorun yoktur.

Silmede tuzak: yuvayı boşaltmak aramayı keser



… `ara(24)` ev adresinde boş yuva görür ve “yok” der. Oysa 24 tabloda!

Çok silme yapılan tablolarda silindi işaretleri birikir ve aramalar uzar: periyodik olarak yeniden hash'lemek (temizlik) gerekir. Zincirlemede bu sorun yoktur.

Silmede tuzak: yuvayı boşaltmak aramayı keser

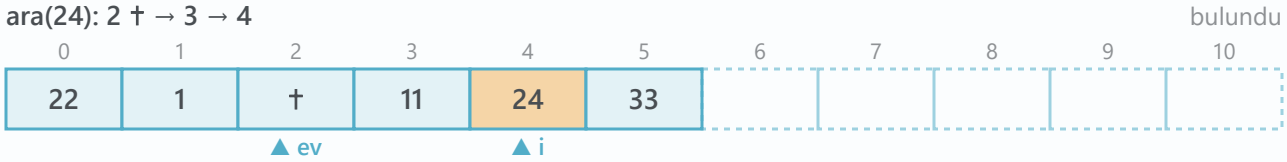


Doğrusu yuvayı **silindi işareti** (SILINDI) ile işaretlemektir.

Çok silme yapılan tablolarda silindi işaretleri birikir ve aramalar uzar: periyodik olarak yeniden hash'lemek (temizlik) gerekir. Zincirlemede bu sorun yoktur.

YÜK FAKTÖRÜ VE SİLME

Silmede tuzak: yuvayı boşaltmak aramayı keser



Arama silindi işaretini boş saymaz, devam eder; ekleme ise silindi işaretini yeniden kullanabilir.

Adım 5 / 5

Çok silme yapılan tablolarda silindi işaretleri birikir ve aramalar uzar: periyodik olarak yeniden hash'lemek (temizlik) gerekir. Zincirlemede bu sorun yoktur.

SİLİNDİ İŞARETİ · KODU ADIM ADIM ÇALIŞTIR

Kod izleme: `sil(13)`, ardından `ara(24)`



sil(13) çağrılır; önce yeri bulunmalı.

Adım 1 / 10

```
 1 int ara(int k) {
 2   int i = k % M;
 3   for (int d = 0; d < M; d++, i = (i + 1) % M) {
 4     if (tablo[i].durum == BOS) return -1;
 5     if (tablo[i].durum == DOLU &&
 6         tablo[i].anahtar == k) return i;
 7   }
 8   return -1;
 9 }
▶ bool sil(int k) {
11   int i = ara(k);
12   if (i < 0) return false;
13   tablo[i].durum = SILINDI;   /* silindi işareti */
14   return true;
15 }
```

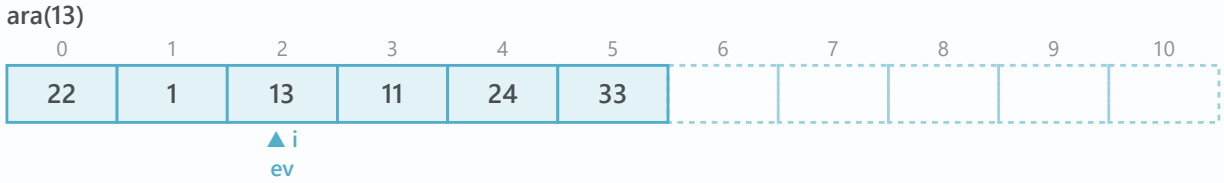
DEĞİŞKENLER

`k = 13`

ÇAĞRI YIĞINI

- **`sil(13)`**
- `main()`

Kod izleme: `sil(13)`, ardından `ara(24)`



Ev adresi i = 13 % 11 = **2**.

Adım 2 / 10

```
 1 int ara(int k) {
 2   int i = k % M;
 3   for (int d = 0; d < M; d++, i = (i + 1) % M) {
 4     if (tablo[i].durum == BOS) return -1;
 5     if (tablo[i].durum == DOLU &&
 6         tablo[i].anahtar == k) return i;
 7   }
 8   return -1;
 9 }
10 bool sil(int k) {
11   int i = ara(k);
12   if (i < 0) return false;
13   tablo[i].durum = SILINDI;   /* silindi işareti */
14   return true;
15 }
```

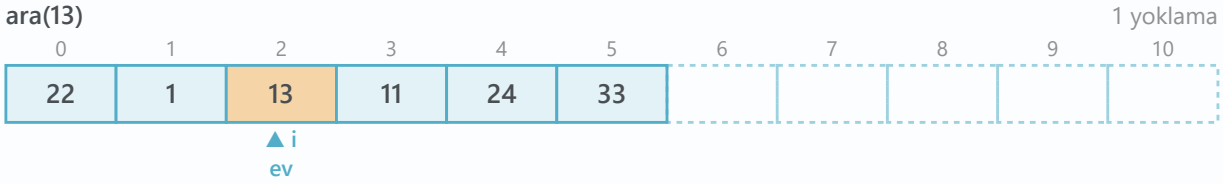
DEĞİŞKENLER

`k = 13` `i = 2`

ÇAĞRI YIĞINI

- **ara(13)**
- sil(13)
- main()

Kod izleme: `sil(13)`, ardından `ara(24)`



Yuva 2 DOLU ve anahtar 13 → **return 2** (1 yoklama).

Adım 3 / 10

```
1  int ara(int k) {
2    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (tablo[i].durum == BOS) return -1;
▶      if (tablo[i].durum == DOLU &&
▶          tablo[i].anahtar == k) return i;
7    }
8    return -1;
9  }
10 bool sil(int k) {
11   int i = ara(k);
12   if (i < 0) return false;
13   tablo[i].durum = SILINDI;    /* silindi işareti */
14   return true;
15 }
```

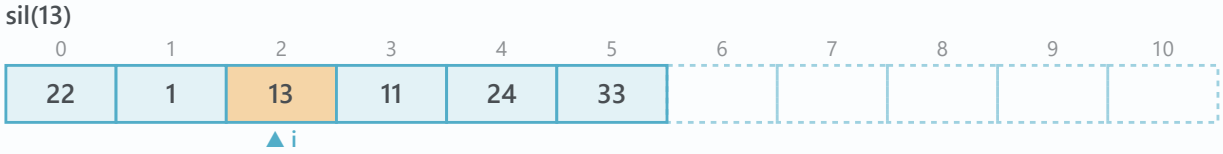
DEĞİŞKENLER

`k = 13` `i = 2` `d = 0` `durum = DOLU`

ÇAĞRI YIĞINI

- **ara(13)**
- sil(13)
- main()

Kod izleme: `sil(13)`, ardından `ara(24)`



`ara` 2 döndürdü; i ≥ 0, silinecek öğe var.

Adım 4 / 10

```
1  int ara(int k) {
2    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
4      if (tablo[i].durum == BOS) return -1;
5      if (tablo[i].durum == DOLU &&
6          tablo[i].anahtar == k) return i;
7    }
8    return -1;
9  }
10 bool sil(int k) {
▶    int i = ara(k);
▶    if (i < 0) return false;
13   tablo[i].durum = SILINDI;   /* silindi işareti */
14   return true;
15 }
```

DEĞİŞKENLER

`k = 13` `i = 2`

ÇAĞRI YIĞINI

`sil(13)`

`main()`

Kod izleme: `sil(13)`, ardından `ara(24)`



tablo[2].durum = SILINDI: silindi işareti.

Adım 5 / 10

```
 1 int ara(int k) {
 2   int i = k % M;
 3   for (int d = 0; d < M; d++, i = (i + 1) % M) {
 4     if (tablo[i].durum == BOS) return -1;
 5     if (tablo[i].durum == DOLU &&
 6         tablo[i].anahtar == k) return i;
 7   }
 8   return -1;
 9 }
10 bool sil(int k) {
11   int i = ara(k);
12   if (i < 0) return false;
▶  tablo[i].durum = SILINDI;   /* silindi işareti */
14   return true;
15 }
```

DEĞİŞKENLER

`k = 13` `i = 2`

`tablo[i].durum = SILINDI`

ÇAĞRI YIĞINI

`sil(13)`

`main()`

Kod izleme: `sil(13)`, ardından `ara(24)`



return true.

Adım 6 / 10

```
1  int ara(int k) {
2    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
4      if (tablo[i].durum == BOS) return -1;
5      if (tablo[i].durum == DOLU &&
6          tablo[i].anahtar == k) return i;
7    }
8    return -1;
9  }
10 bool sil(int k) {
11   int i = ara(k);
12   if (i < 0) return false;
13   tablo[i].durum = SILINDI;   /* silindi işareti */
▶    return true;
15 }
```

DEĞİŞKENLER

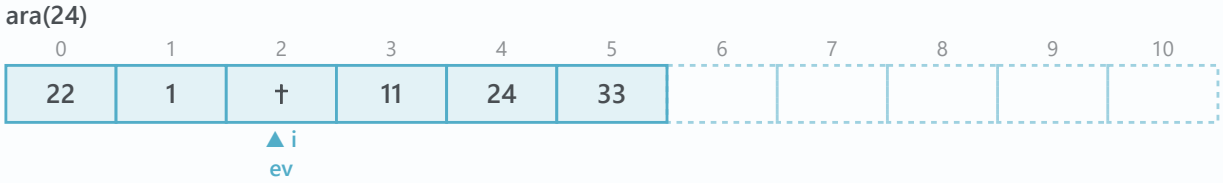
`k = 13` `i = 2`

ÇAĞRI YIĞINI

sil(13)

main()

Kod izleme: `sil(13)`, ardından `ara(24)`



Ev adresi i = 24 % 11 = **2**.

Adım 7 / 10

```
1  int ara(int k) {
▶    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
4      if (tablo[i].durum == BOS) return -1;
5      if (tablo[i].durum == DOLU &&
6          tablo[i].anahtar == k) return i;
7    }
8    return -1;
9  }
10 bool sil(int k) {
11   int i = ara(k);
12   if (i < 0) return false;
13   tablo[i].durum = SILINDI;   /* silindi işareti */
14   return true;
15 }
```

DEĞİŞKENLER

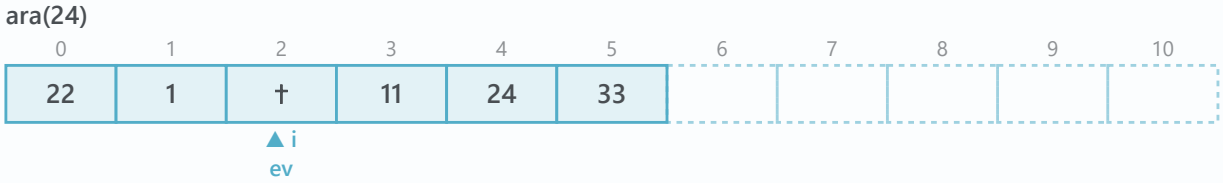
`k = 24` `i = 2`

ÇAĞRI YIĞINI

ara(24)

main()

Kod izleme: `sil(13)`, ardından `ara(24)`



Yuva 2: silindi işareti: silinmiş, aramaya devam → i = 3.

Adım 8 / 10

```
1  int ara(int k) {
2    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (tablo[i].durum == BOS) return -1;
▶      if (tablo[i].durum == DOLU &&
6          tablo[i].anahtar == k) return i;
7    }
8    return -1;
9  }
10 bool sil(int k) {
11   int i = ara(k);
12   if (i < 0) return false;
13   tablo[i].durum = SILINDI;   /* silindi işareti */
14   return true;
15 }
```

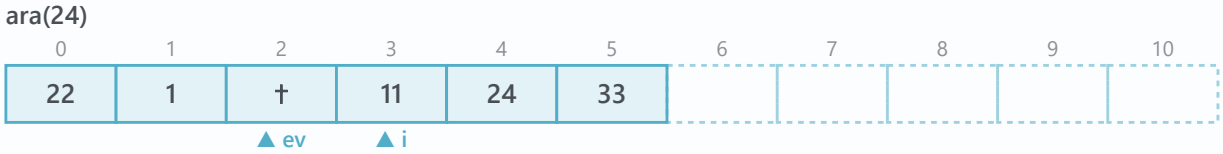
DEĞİŞKENLER

`k = 24` `i = 2` `d = 0` `durum = SILINDI`

ÇAĞRI YIĞINI

- **ara(24)**
- main()

Kod izleme: `sil(13)`, ardından `ara(24)`



Yuva 3: başka anahtar (11) → i = 4.


```
1  int ara(int k) {
2    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (tablo[i].durum == BOS) return -1;
▶      if (tablo[i].durum == DOLU &&
6          tablo[i].anahtar == k) return i;
7    }
8    return -1;
9  }
10 bool sil(int k) {
11   int i = ara(k);
12   if (i < 0) return false;
13   tablo[i].durum = SILINDI;   /* silindi işareti */
14   return true;
15 }
```

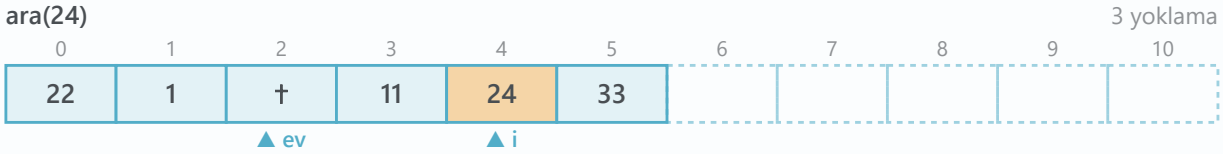
DEĞİŞKENLER

`k = 24` `i = 3` `d = 1` `durum = DOLU`

ÇAĞRI YIĞINI

- **ara(24)**
- main()

Kod izleme: `sil(13)`, ardından `ara(24)`



Yuva 4 DOLU ve anahtar 24 → **return 4** (3 yoklama).

Adım 10 / 10

```
1  int ara(int k) {
2    int i = k % M;
3    for (int d = 0; d < M; d++, i = (i + 1) % M) {
▶      if (tablo[i].durum == BOS) return -1;
▶      if (tablo[i].durum == DOLU &&
▶          tablo[i].anahtar == k) return i;
7    }
8    return -1;
9  }
10 bool sil(int k) {
11   int i = ara(k);
12   if (i < 0) return false;
13   tablo[i].durum = SILINDI;    /* silindi işareti */
14   return true;
15 }
```

DEĞİŞKENLER

`k = 24` `i = 4` `d = 2` `durum = DOLU`

ÇAĞRI YIĞINI

- `ara(24)`
- `main()`

YÜK FAKTÖRÜ VE SİLME

Yük faktörü α = n/m ve beklenen yoklama sayısı

α	Zincir ✓	Zincir ✗	Doğrusal ✓	Doğrusal ✗	Düzgün* ✓	Düzgün* ✗
0,25	1,13	1,25	1,2	1,4	1,15	1,3
0,5	1,25	1,50	1,5	2,5	1,39	2,0
0,75	1,38	1,75	2,5	8,5	1,85	4,0
0,9	1,45	1,90	5,5	50,5	2,56	10,0

✓ başarılı, ✗ başarısız arama. Formüller: zincirleme $1 + \alpha/2$ ve $1 + \alpha$ ($\Theta(1 + \alpha)$); doğrusal yoklama (Knuth) $\frac{1}{2}(1 + 1/(1-\alpha))$ ve $\frac{1}{2}(1 + 1/(1-\alpha)^2)$; * düzgün hash varsayımıyla açık adresleme $(1/\alpha)\cdot\ln(1/(1-\alpha))$ ve $1/(1-\alpha)$ (CLRS).

α sabit tutulursa tüm işlemler ortalama **O(1)**'dir. Pratik eşik: açık adreslemede α ≤ 0,5–0,7, zincirlemede α ≤ 1 civarı. Eşik aşılınca tablo büyütülür.

Yeniden boyutlandırma (rehashing)

```c
typedef struct {
  yuva *a;      /* m yuvalık dinamik dizi */
  int m, n;     /* boyut, dolu yuva sayısı */
} Tablo;
/* ekledikten sonra: */
if (2 * t->n > t->m)              /* α > 1/2 */
  yeniden_boyutla(t, sonraki_asal(2 * t->m));
```

Neden yeniden hash?

h(k) = k mod m, m'ye bağlıdır: öğeler yeni tabloda başka yuvalara düşer; kopyalamak yetmez.

Amortize O(1)

Boyut her seferinde iki katına çıkınca n ekleme için toplam kopyalama ≤ 2n: ekleme başına sabit.

Silindi işaretleri

Yeniden hash'leme silindi işaretlerini de temizler.

Kod izleme: `yeniden_boyutla(t, 11)`



α = 0,6

n = 3, m = 5 → α = 0,6 > 0,5: tablo 11 yuvaya büyütülecek (≈ 2m, asal).

Adım 1 / 11

```
▶ bool yeniden_boyutla(Tablo *t, int yeni_m) {
2   yuva *eski = t->a;
3   int eski_m = t->m;
4   yuva *a = calloc(yeni_m, sizeof *a);  /* BOS */
5   if (a == NULL) return false;
6   t->a = a; t->m = yeni_m; t->n = 0;
7   for (int i = 0; i < eski_m; i++)
8     if (eski[i].durum == DOLU)
9       tablo_ekle(t, eski[i].anahtar);
10  free(eski);
11  return true;
12 }
```

DEĞİŞKENLER

`t→m = 5` `t→n = 3`

`yeni_m = 11`

ÇAĞRI YIĞINI

`yeniden_boyutla(t, 11)`

`main()`

Kod izleme: `yeniden_boyutla(t, 11)`



Eski dizinin adresi ve boyutu saklanır.

Adım 2 / 11

```
 1 bool yeniden_boyutla(Tablo *t, int yeni_m) {
 2   yuva *eski = t->a;
 3   int eski_m = t->m;
 4   yuva *a = calloc(yeni_m, sizeof *a);  /* BOS */
 5   if (a == NULL) return false;
 6   t->a = a; t->m = yeni_m; t->n = 0;
 7   for (int i = 0; i < eski_m; i++)
 8     if (eski[i].durum == DOLU)
 9       tablo_ekle(t, eski[i].anahtar);
10   free(eski);
11   return true;
12 }
```

DEĞİŞKENLER

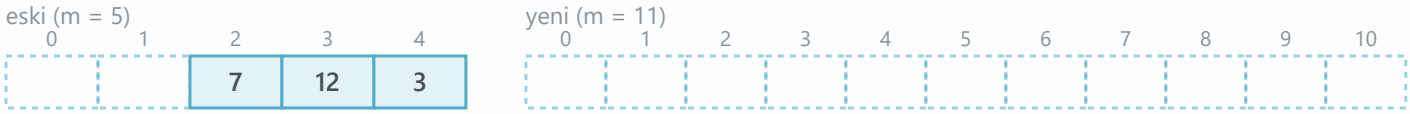
`eski = &yuva[5]` `eski_m = 5` `yeni_m = 11`

ÇAĞRI YIĞINI

`yeniden_boyutla(t, 11)`

`main()`

Kod izleme: `yeniden_boyutla(t, 11)`



`calloc` sıfırlanmış 11 yuva verir: hepsi BOS (= 0).

Adım 3 / 11

```
1  bool yeniden_boyutla(Tablo *t, int yeni_m) {
2    yuva *eski = t->a;
3    int eski_m = t->m;
▶    yuva *a = calloc(yeni_m, sizeof *a);  /* BOS */
5    if (a == NULL) return false;
6    t->a = a; t->m = yeni_m; t->n = 0;
7    for (int i = 0; i < eski_m; i++)
8      if (eski[i].durum == DOLU)
9        tablo_ekle(t, eski[i].anahtar);
10   free(eski);
11   return true;
12 }
```

DEĞİŞKENLER

`eski = &yuva[5]` `eski_m = 5`

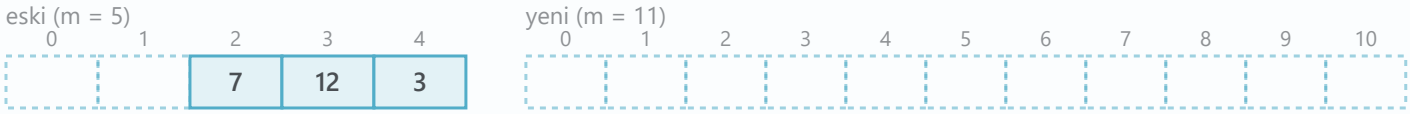
`a = &yuva[11]`

ÇAĞRI YIĞINI

`yeniden_boyutla(t, 11)`

`main()`

Kod izleme: `yeniden_boyutla(t, 11)`



Bellek alındı; tablo yeni diziye bağlanır, n = 0.


```
 1 bool yeniden_boyutla(Tablo *t, int yeni_m) {
 2   yuva *eski = t->a;
 3   int eski_m = t->m;
 4   yuva *a = calloc(yeni_m, sizeof *a);  /* BOS */
 ▶   if (a == NULL) return false;
 ▶   t->a = a; t->m = yeni_m; t->n = 0;
 7   for (int i = 0; i < eski_m; i++)
 8     if (eski[i].durum == DOLU)
 9       tablo_ekle(t, eski[i].anahtar);
10   free(eski);
11   return true;
12 }
```

DEĞİŞKENLER

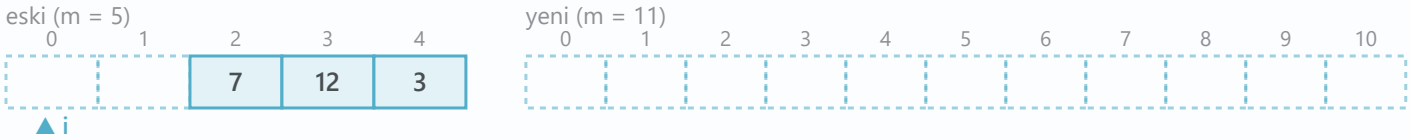
`eski_m = 5` `t→m = 11` `t→n = 0`

ÇAĞRI YIĞINI

`yeniden_boyutla(t, 11)`

`main()`

Kod izleme: `yeniden_boyutla(t, 11)`



eski[0] BOS: atlanır.


```
1  bool yeniden_boyutla(Tablo *t, int yeni_m) {
2    yuva *eski = t->a;
3    int eski_m = t->m;
4    yuva *a = calloc(yeni_m, sizeof *a);  /* BOS */
5    if (a == NULL) return false;
6    t->a = a; t->m = yeni_m; t->n = 0;
▶    for (int i = 0; i < eski_m; i++)
▶      if (eski[i].durum == DOLU)
9        tablo_ekle(t, eski[i].anahtar);
10   free(eski);
11   return true;
12 }
```

DEĞİŞKENLER

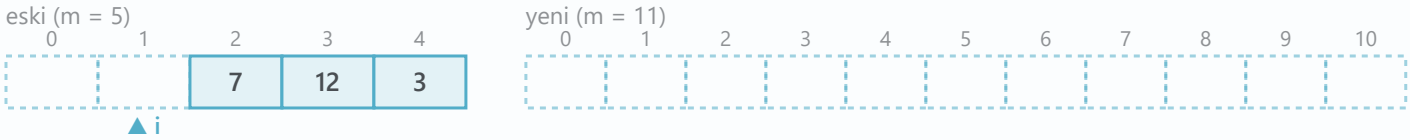
`i = 0` `eski[i].durum = BOS` `t→n = 0`

ÇAĞRI YIĞINI

`yeniden_boyutla(t, 11)`

`main()`

Kod izleme: `yeniden_boyutla(t, 11)`



eski[1] BOS: atlanır.


```
1  bool yeniden_boyutla(Tablo *t, int yeni_m) {
2    yuva *eski = t->a;
3    int eski_m = t->m;
4    yuva *a = calloc(yeni_m, sizeof *a);  /* BOS */
5    if (a == NULL) return false;
6    t->a = a; t->m = yeni_m; t->n = 0;
▶    for (int i = 0; i < eski_m; i++)
▶      if (eski[i].durum == DOLU)
9        tablo_ekle(t, eski[i].anahtar);
10   free(eski);
11   return true;
12 }
```

DEĞİŞKENLER

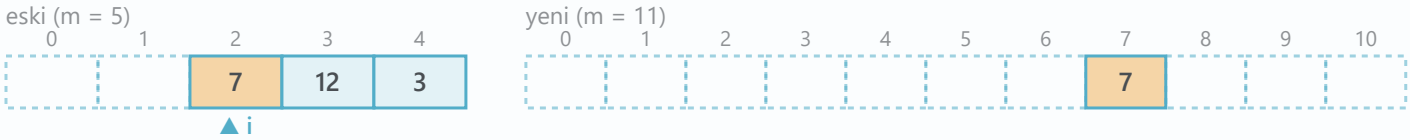
`i = 1` `eski[i].durum = BOS` `t→n = 0`

ÇAĞRI YIĞINI

`yeniden_boyutla(t, 11)`

`main()`

Kod izleme: `yeniden_boyutla(t, 11)`



eski[2] = 7 DOLU → yeniden hash: 7 % 11 = 7 → yuva **7**. (Eski ev adresi 7 % 5 = 2 idi.)


```
 1 bool yeniden_boyutla(Tablo *t, int yeni_m) {
 2   yuva *eski = t->a;
 3   int eski_m = t->m;
 4   yuva *a = calloc(yeni_m, sizeof *a);  /* BOS */
 5   if (a == NULL) return false;
 6   t->a = a; t->m = yeni_m; t->n = 0;
 7   for (int i = 0; i < eski_m; i++)
 8     if (eski[i].durum == DOLU)
▶          tablo_ekle(t, eski[i].anahtar);
10   free(eski);
11   return true;
12 }
```

DEĞİŞKENLER

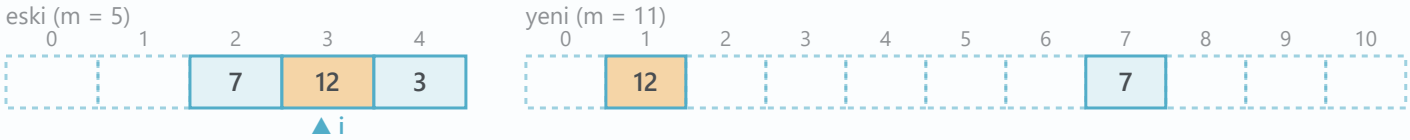
`i = 2` `eski[i].anahtar = 7` `t->n = 1`

ÇAĞRI YIĞINI

`yeniden_boyutla(t, 11)`

`main()`

Kod izleme: `yeniden_boyutla(t, 11)`



eski[3] = 12 DOLU → yeniden hash: 12 % 11 = 1 → yuva **1**. (Eski ev adresi 12 % 5 = 2 idi.)

Adım 8 / 11

```
 1 bool yeniden_boyutla(Tablo *t, int yeni_m) {
 2   yuva *eski = t->a;
 3   int eski_m = t->m;
 4   yuva *a = calloc(yeni_m, sizeof *a);  /* BOS */
 5   if (a == NULL) return false;
 6   t->a = a; t->m = yeni_m; t->n = 0;
 7   for (int i = 0; i < eski_m; i++)
 8     if (eski[i].durum == DOLU)
▶          tablo_ekle(t, eski[i].anahtar);
10   free(eski);
11   return true;
12 }
```

DEĞİŞKENLER

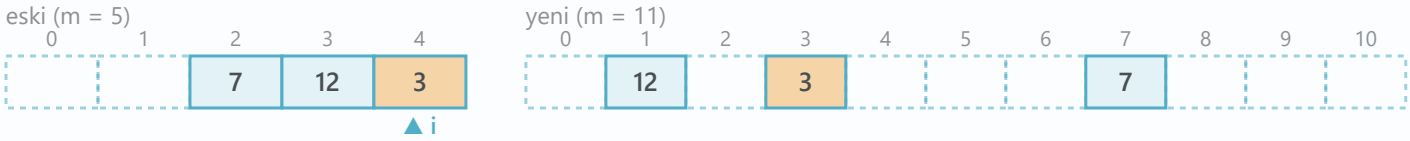
`i = 3` `eski[i].anahtar = 12` `t→n = 2`

ÇAĞRI YIĞINI

`yeniden_boyutla(t, 11)`

`main()`

Kod izleme: `yeniden_boyutla(t, 11)`



eski[4] = 3 DOLU → yeniden hash: 3 % 11 = 3 → yuva **3**. (Eski ev adresi 3 % 5 = 3 idi.)


```
 1 bool yeniden_boyutla(Tablo *t, int yeni_m) {
 2   yuva *eski = t->a;
 3   int eski_m = t->m;
 4   yuva *a = calloc(yeni_m, sizeof *a);  /* BOS */
 5   if (a == NULL) return false;
 6   t->a = a; t->m = yeni_m; t->n = 0;
 7   for (int i = 0; i < eski_m; i++)
 8     if (eski[i].durum == DOLU)
▶          tablo_ekle(t, eski[i].anahtar);
10   free(eski);
11   return true;
12 }
```

DEĞİŞKENLER

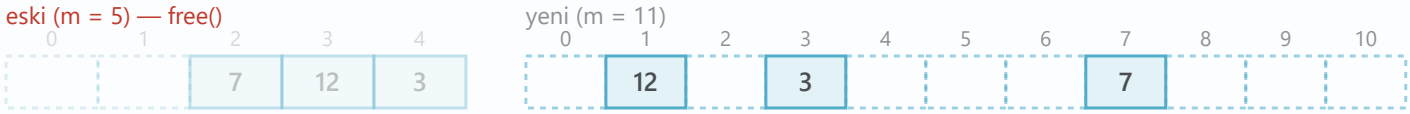
`i = 4` `eski[i].anahtar = 3` `t->n = 3`

ÇAĞRI YIĞINI

`yeniden_boyutla(t, 11)`

`main()`

Kod izleme: `yeniden_boyutla(t, 11)`



Eski dizi **free** edilir. Yeni α = 3/11 ≈ 0,27.

Adım 10 / 11

```
 1 bool yeniden_boyutla(Tablo *t, int yeni_m) {
 2   yuva *eski = t->a;
 3   int eski_m = t->m;
 4   yuva *a = calloc(yeni_m, sizeof *a);  /* BOS */
 5   if (a == NULL) return false;
 6   t->a = a; t->m = yeni_m; t->n = 0;
 7   for (int i = 0; i < eski_m; i++)
 8     if (eski[i].durum == DOLU)
 9       tablo_ekle(t, eski[i].anahtar);
▶10   free(eski);
11   return true;
12 }
```

DEĞİŞKENLER

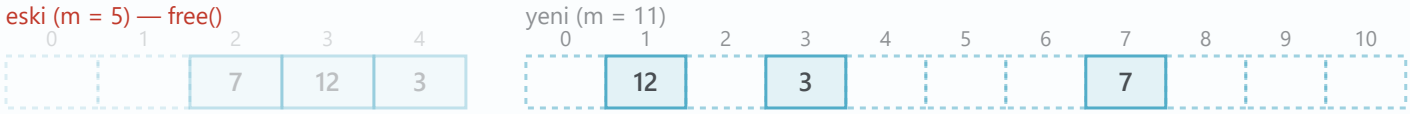
`t→m = 11` `t→n = 3`

ÇAĞRI YIĞINI

`yeniden_boyutla(t, 11)`

`main()`

Kod izleme: `yeniden_boyutla(t, 11)`



α ≈ 0,27

return true. Tüm öğeler yeniden eklendi: O(n + m).

Adım 11 / 11

```
 1 bool yeniden_boyutla(Tablo *t, int yeni_m) {
 2   yuva *eski = t->a;
 3   int eski_m = t->m;
 4   yuva *a = calloc(yeni_m, sizeof *a);  /* BOS */
 5   if (a == NULL) return false;
 6   t->a = a; t->m = yeni_m; t->n = 0;
 7   for (int i = 0; i < eski_m; i++)
 8     if (eski[i].durum == DOLU)
 9       tablo_ekle(t, eski[i].anahtar);
10   free(eski);
▶    return true;
12 }
```

DEĞİŞKENLER

`t→m = 11` `t→n = 3`

ÇAĞRI YIĞINI

`yeniden_boyutla(t, 11)`

`main()`

ÖZET

Zincirleme mi, açık adresleme mi?

Zincirleme

+ α > 1 olabilir; dolmaz, yavaşlar.
+ Silme basit, silindi işareti yok.
+ Kötü hash'e karşı daha dayanıklı.
+ Düğüm başına işaretçi + `malloc` maliyeti; önbellek dostu değil.

Açık adresleme

− Ek işaretçi yok; bellek bitişik, önbellek dostu.
− $\alpha \leq 1$; $\alpha \to 1$ iken maliyet patlar.
− Silme için silindi işareti gerekir.
− Kümelenmeye karşı iyi yoklama (çift hash) seçilmeli.

Kural: öğe sayısı öngörülemiyorsa ve çok silme varsa zincirleme; bellek ve hız kritikse, silme az ve α düşük tutulabiliyorsa açık adresleme.

ÖZET

Karmaşıklık özeti

İşlem	Ortalama (α sabit)	En kötü	Not
ara	O(1)	O(n)	tüm anahtarlar aynı kovaya düşerse
ekle	O(1)	O(n)	zincirde başa ekleme her zaman O(1)
sil	O(1)	O(n)	açık adreslemede silindi işareti
yeniden boyutlandır	O(n + m)	O(n + m)	ekleme başına amortize O(1)
sıralı gezinme / min / max	O(m + n lg n)	O(m + n lg n)	hash sıra tutmaz; min/max yalnız tarama: O(n + m)

En kötü durum kötü niyetli girdilerle tetiklenebilir (hash flooding). Önlem: fonksiyonu çalışma anında rastgele bir aileden seçmek — **evrensel hash** (universal hashing, CLRS 11.3.3).

ÖZET

Sık yapılan hatalar

1. Negatif anahtarda `k % m`'nin negatif çıkıp dizi dışına yazması.
2. m'yi 10 ya da 2'nin kuvveti seçip yalnızca son basamaklara/bitlere bakmak.
3. Açık adreslemede silinen yuvayı BOS yapmak: aramalar yarıda kalır.
4. Tablo doluyken sonsuz döngüye girmek: deneme sayısı m ile sınırlanmalı.
5. `malloc` sonucunu denetlememek; `free` sonrası düğümü kullanmak.
6. Yeniden boyutlandırmada öğeleri eski indisleriyle kopyalamak (yeniden hash'lemeden).

ÖZET

Alıştırmalar

1. m = 7, h(k) = k mod 7, doğrusal yoklama: 10, 17, 24, 3 sırayla eklenince tablo ne olur?
2. Aynı anahtarları zincirleme (başa ekleme) ile yerleştirin; en uzun zincir kaç düğümdür?
3. m = 11 ve karesel yoklamada 0, 1, 4, 9, 16, 25 (mod 11) yoklama dizisini yazın.
4. α = 0,9 olan doğrusal yoklamalı tabloda başarısız arama kaç yoklama bekler?
5. `"ab"` ve `"ba"` için toplama hash'i ve 31 tabanlı polinom hash'i (mod 11) hesaplayın.
6. Açık adreslemede `ekle`, SILINDI yuvaya yazmadan önce neden `ara` çağırmalıdır?

▶ **Cevapları göster**

ÖZET

Bölüm özeti

Fikir

Anahtardan konum hesaplanır: ortalama O(1) ara/ekle/sil.

Hash fonksiyonu

Bölme (m asal), çarpma (A ≈ 0,618), katlama, polinom dizgi hash'i.

Açık adresleme

Doğrusal, karesel, çift hash; kümelenme ve silindi işareti.

Zincirleme

Kova başına bağlı liste; başa ekleme O(1), silme kolay.

Yük faktörü

α = n/m küçük tutulmalı; eşik aşılınca yeniden boyutlandırılır.

Sınırlar

Sıra bilgisi yok; en kötü O(n); evrensel hash ile koruma.

ÖZET

Kaynaklar ve devam

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, Bölüm 8 (Hashing).
- T. H. Cormen vd., *Introduction to Algorithms* (CLRS), Bölüm 11 (Hash Tables).
- R. Sedgewick, *Algorithms in C*, Kısım 1–4, Bölüm 14 (Hashing).
- D. E. Knuth, *The Art of Computer Programming*, Cilt 3, Bölüm 6.4.

← Bölüm 13: Öncelikli Kuyruk ve Heap

Ders sayfası

Bölüm 15: Çizgeler →

Son

Bölüm 14: Hash Tabloları

Sonraki bölüm: Bölüm 15: Çizgeler · Ders sayfası