Bölüm 11: Trie Veri Yapısı

Algoritmalar

Önek ağacı · Düğüm yapısı · Ekleme · Arama ve önek sorgusu · Otomatik tamamlama · Silme · Karmaşıklık ve bellek · Sıkıştırılmış ve sonek trie'leri

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

© Sercan Külcü. Tüm hakları saklıdır. · Sunumun etkileşimli HTML sürümü: https://sercankulcu.github.io/files/algorithms/slides/Bolum_11_Trie_Veri_Yapisi.html

GİRİŞ

Öğrenme Çıktıları

Yapı

Trie'nin kök, düğüm, kenar ve kelime sonu işaretinden oluşan yapısını açıklamak; bir kelime kümesinin trie'sini çizmek.

İşlemler

Ekleme, arama, önek sorgusu, otomatik tamamlama ve silme işlemlerini Java ile yazmak ve adım adım izlemek.

Analiz

İşlemlerin O(L) zamanını ve O(N · M) belleğini gerekçelendirmek; trie'yi hash tablosu ve BST ile karşılaştırmak.

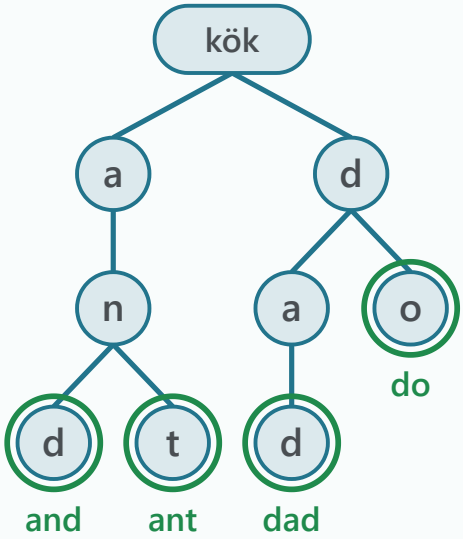
Türevler

Sıkıştırılmış trie, sonek trie'si ve Aho–Corasick'in hangi sorunu çözdüğünü açıklamak.

TRIE KAVRAMI

Trie (Önek Ağacı, Prefix Tree)

- **Dizgileri** (string) hızlı arama yapılabilecek biçimde saklayan **ağaç benzeri** bir veri yapısıdır.
- **Kök düğüm boştur**; kökten aşağı her adım bir **karakteri** temsil eder.
- Kelimeler, **kökten başlayan yollarla** saklanır; ortak **önekler** (prefix) tek kez saklanır.
- Kullanım alanları: **otomatik tamamlama**, **yazım denetimi**, **IP yönlendirme**.



Ad, bilgi **geri getirme** anlamındaki İngilizce *re**trie**val* sözcüğünden gelir (E. Fredkin, 1960). Çift çember: o düğümde bir **kelime biter**.

TRIE KAVRAMI

Neden Trie?

Bir sözlükte **"an" ile başlayan tüm kelimeleri** bulmak istiyoruz. Farklı yapılar bu soruya nasıl yanıt verir?

Dizi / liste

Her kelime tek tek karşılaştırılır: n kelime için O(n · L). Kelime sayısı büyüdükçe yavaşlar.

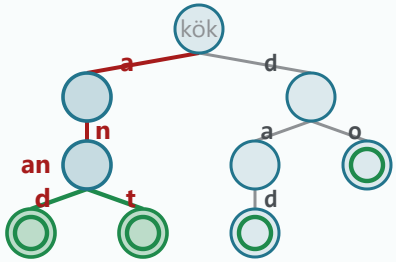
Hash tablosu

Tam kelimeyi ortalama O(L)'de bulur, ama **önek sorgusu yapamaz**: "an" ile "and"in hash değerleri ilişkisizdir.

Trie

Önce "a", sonra "n" dalına inilir (2 adım); o düğümün altındaki **her kelime** "an" ile başlar.

Trie'de arama süresi saklanan kelime **sayısına** değil, aranan kelimenin **uzunluğuna** (L) bağlıdır.



TRIE KAVRAMI

Temel Bileşenler

Kök (root)

Boş düğüm; hiçbir karakteri temsil etmez, her aramanın başlangıcıdır.

Düğümler (nodes)

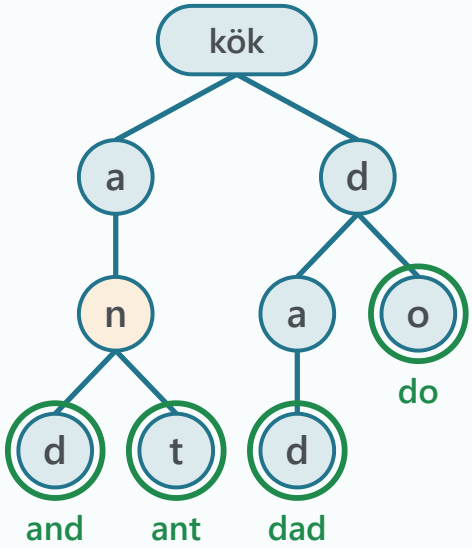
Kökten o düğüme kadarki yol bir **önek**tir; her düğüm çocuk düğümlere bağlantılar tutar.

Kenarlar (edges)

Düğümler arası bağlantılardır; her kenar **bir karakter** taşır.

Kelime sonu işareti

`kelimeSonu` (end of word): bu düğümde bir kelimenin bittiğini belirtir.

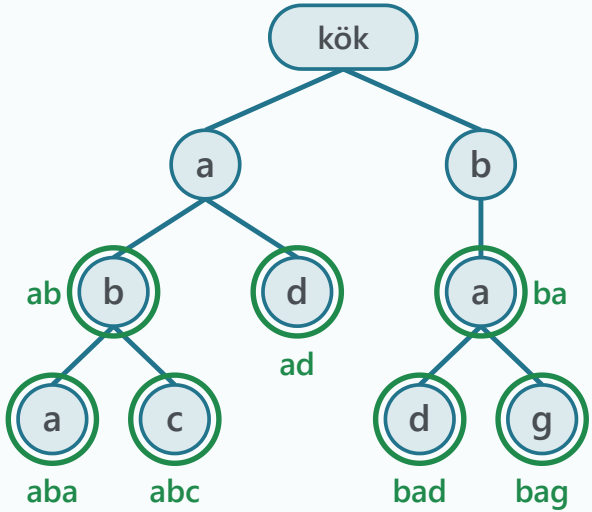


Renkli düğüm "an" önekini temsil eder; "an" bir kelime değildir (işaretsiz), ama "and" ve "ant"in önekidir.

Örnek: Yedi Kelimelik Trie

Kelimeler
ab · aba · abc
ad
ba · bad · bag

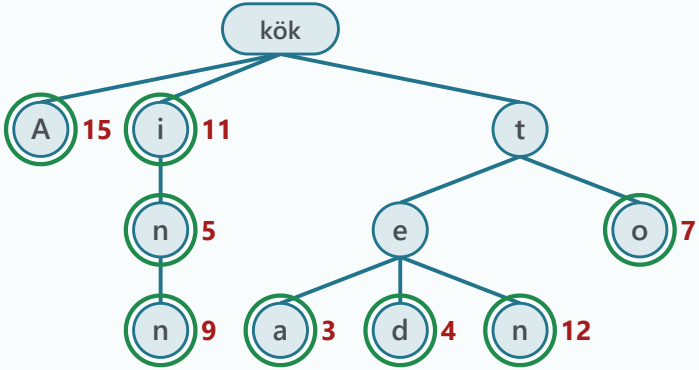
7 kelime, toplam 18 karakter → yalnızca **9 düğüm** (+ kök).



"ab" hem bir **kelime** hem de "aba" ile "abc"nin **öneki**dir. Yalnızca yapıya bakarak bunu anlayamayız; bu yüzden `kelimeSonu` işareti şarttır.

Örnek: Değer Taşıyan Trie

Anahtar	Değer	Anahtar	Değer
A	15	tea	3
i	11	ted	4
in	5	ten	12
inn	9	to	7



Trie bir **sözlük** (map) gibi de kullanılabilir: `kelimeSonu` yerine düğümde bir değer saklanır. "t" ve "te" değer taşımaz; yalnızca önektir. Büyük harf "A", 26 küçük harflik dizi yerine daha geniş bir alfabe (ör. `HashMap`) gerektirir.

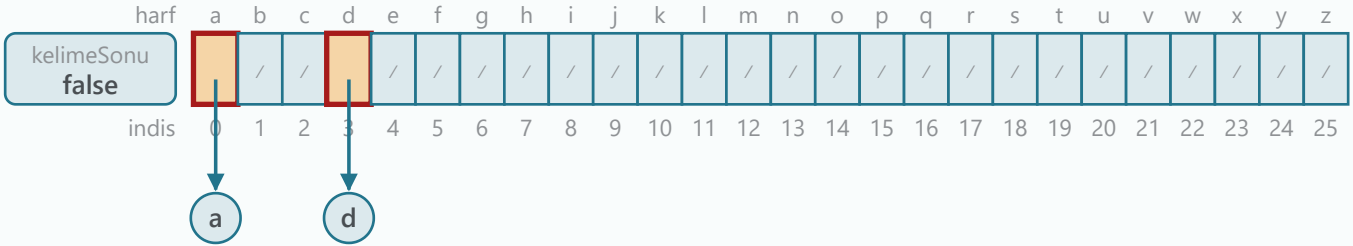
UYGULAMA

Trie Düğüm Sınıfı

```
public class TrieDugumu {
  TrieDugumu[] cocuk;  // 'a'..'z'
  boolean kelimeSonu;  // kelime biter mi?

  public TrieDugumu() {
    kelimeSonu = false;
    cocuk = new TrieDugumu[26];
  }
}
```

- Her düğüm **26 çocuk referansı** tutar; başlangıçta hepsi `null`.
- Karakter düğümde **saklanmaz**: hangi karakter olduğu, düğümün ebeveyninin **hangi indisinde** durduğundan anlaşılır.
- `kelimeSonu`: kökten buraya gelen yol bir kelime mi?

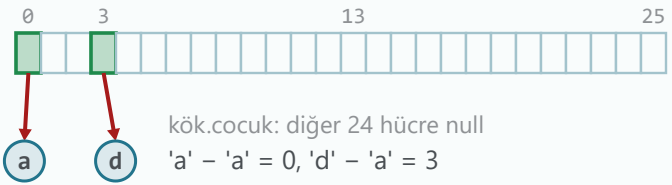


and, ant, dad, do eklendikten sonra kökün `cocuk` dizisi: 26 hücreden yalnızca 2'si dolu (*/* = null).

UYGULAMA

Karakterden İndise: `ch - 'a'`

ch	Unicode	ch – 'a'
'a'	97	0
'd'	100	3
'n'	110	13
'o'	111	14
't'	116	19
'z'	122	25



kök.cocuk: diğer 24 hücre null
'a' – 'a' = 0, 'd' – 'a' = 3

- Java'da `char` bir sayıdır; `ch - 'a'` küçük harfi 0..25 aralığına eşler.
- Çocuğa erişim dizi indisiyle **O(1)**'dir: `dugum.cocuk[ch - 'a']`.
- "and" için sırasıyla `cocuk[0]`, `cocuk[13]`, `cocuk[3]` kullanılır.

`'A' - 'a'` = −32, `'ç' - 'a'` = 134 → `ArrayIndexOutOfBoundsException`. Büyük harf ve Türkçe karakter için girdi dönüştürülmeli ya da `HashMap` kullanılmalı.

EKLEME

Ekleme (Insert)

```
public void ekle(String kelime) {
  TrieDugumu dugum = kok;
  for (char ch : kelime.toCharArray()) {
    int i = ch - 'a';
    if (dugum.cocuk[i] == null)
      dugum.cocuk[i] = new TrieDugumu();
    dugum = dugum.cocuk[i];
  }
  dugum.kelimeSonu = true;
}
```

1. **Kökten** başlanır.
2. Kelimenin her karakteri için ilgili çocuğa bakılır: `cocuk[ch - 'a']`.
3. Çocuk `null` ise **yeni düğüm** oluşturulur; varsa (ortak önek) **mevcut düğüm kullanılır**.
4. O çocuğa ilerlenir.
5. Karakterler bitince son düğümde `kelimeSonu = true` yapılır.

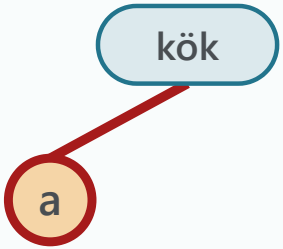
Zaman: **O(L)** (L = kelime uzunluğu); en fazla L yeni düğüm oluşur. Aynı kelimeyi iki kez eklemek yapıyı değiştirmez.

Ekleme Adım Adım: and, ant, dad, do



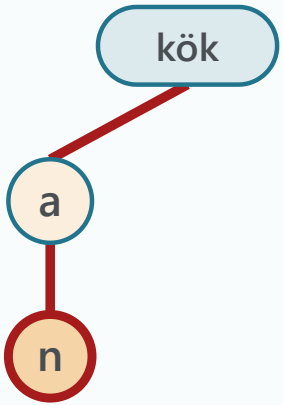
Başlangıçta trie yalnızca **kökten** oluşur; kökün 26 çocuğunun hepsi `null`, `kelimeSonu` = false.

Ekleme Adım Adım: and, ant, dad, do



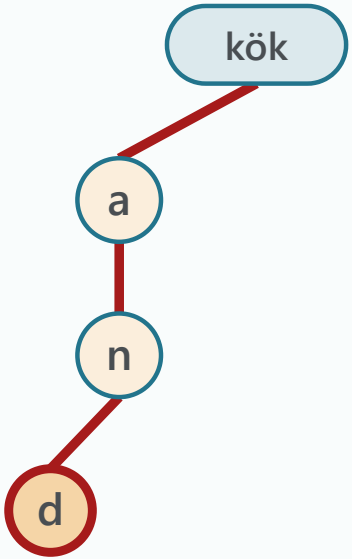
"and" ekleniyor, karakter 'a': `cocuk[0]` null → **yeni düğüm** oluşturulur ve ona ilerlenir.

Ekleme Adım Adım: and, ant, dad, do



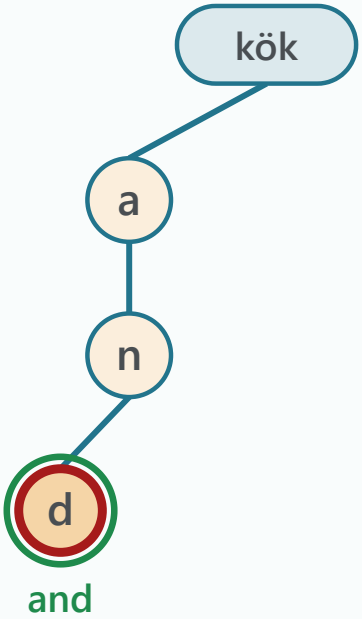
"and" ekleniyor, karakter 'n': `cocuk[13]` null → **yeni düğüm** oluşturulur ve ona ilerlenir.

Ekleme Adım Adım: and, ant, dad, do



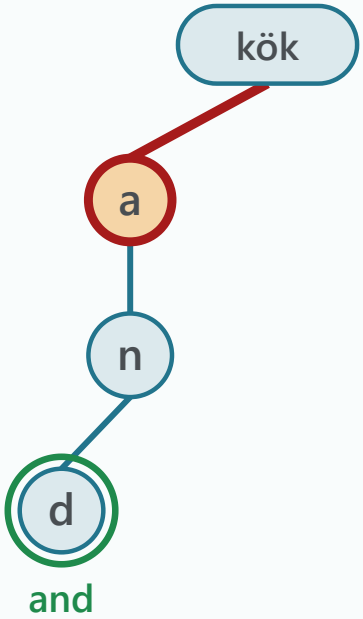
"and" ekleniyor, karakter 'd': `cocuk[3]` null → **yeni düğüm** oluşturulur ve ona ilerlenir.

Ekleme Adım Adım: and, ant, dad, do



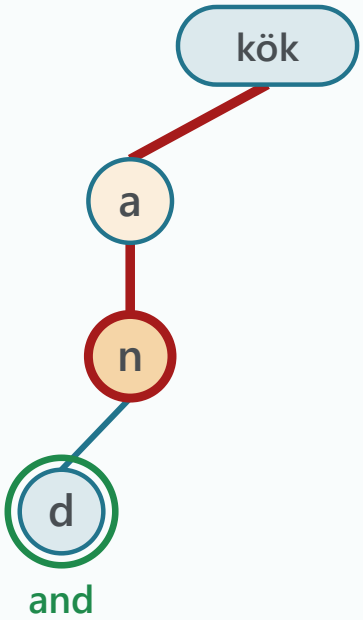
Karakterler bitti: son düğümde `kelimeSonu = true` (çift çember). "and" eklendi. Kökten itibaren `cocuk[0]` → `cocuk[13]` → `cocuk[3]` yolu kuruldu.

Ekleme Adım Adım: and, ant, dad, do



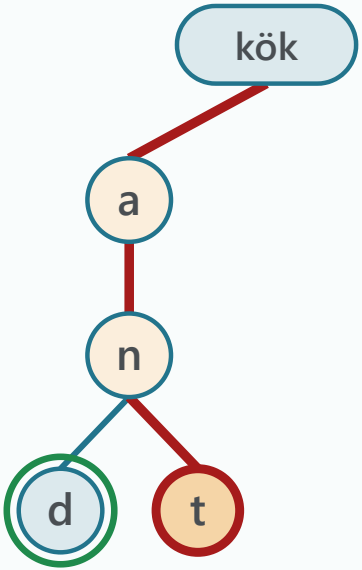
"ant" ekleniyor, karakter 'a': `cocuk[0]` dolu → **mevcut** "a" düğümüne ilerlenir (ortak önek).

Ekleme Adım Adım: and, ant, dad, do



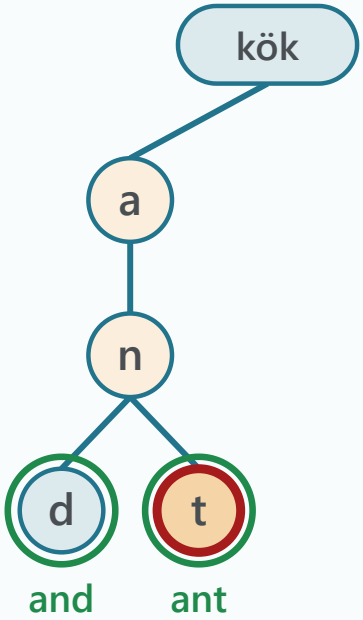
"ant" ekleniyor, karakter 'n': `cocuk[13]` dolu → **mevcut** "an" düğümüne ilerlenir (ortak önek).

Ekleme Adım Adım: and, ant, dad, do



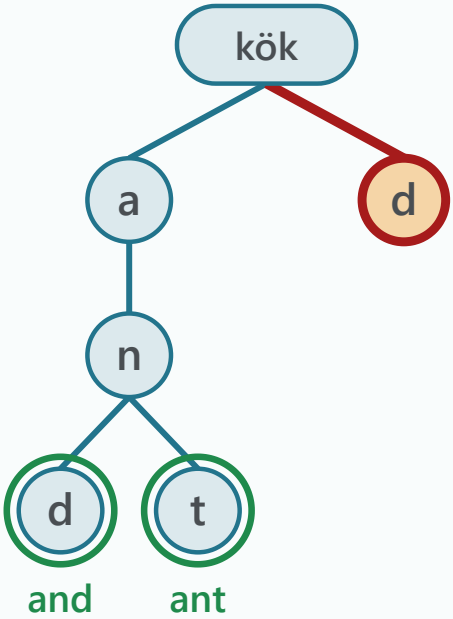
"ant" ekleniyor, karakter 't': `cocuk[19]` null → **yeni düğüm** oluşturulur ve ona ilerlenir.

Ekleme Adım Adım: and, ant, dad, do



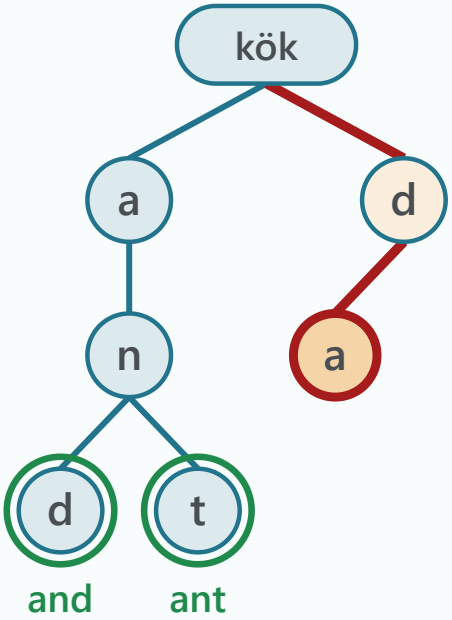
Karakterler bitti: son düğümde `kelimeSonu = true` (çift çember). "ant" eklendi. Yalnızca 1 yeni düğüm gerekti: "an" öneki paylaşıldı.

Ekleme Adım Adım: and, ant, dad, do



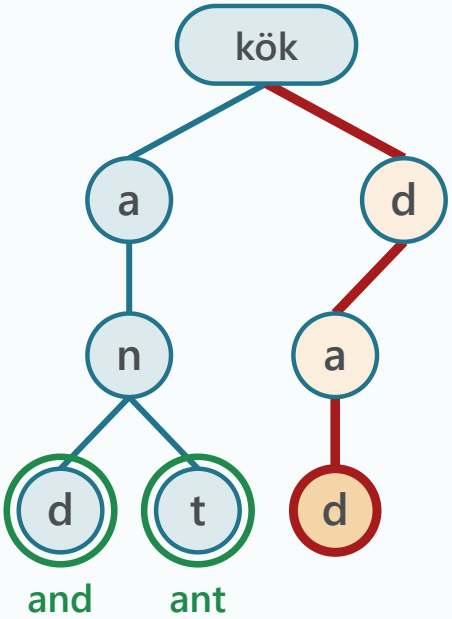
"dad" ekleniyor, karakter 'd': `cocuk[3]` null → **yeni düğüm** oluşturulur ve ona ilerlenir.

Ekleme Adım Adım: and, ant, dad, do



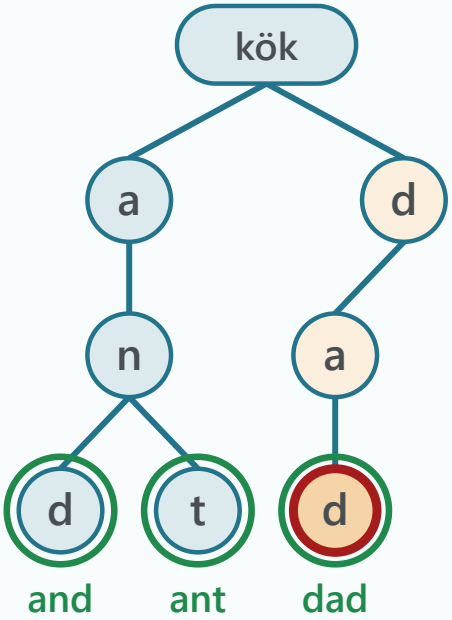
"dad" ekleniyor, karakter 'a': `cocuk[0]` null → **yeni düğüm** oluşturulur ve ona ilerlenir.

Ekleme Adım Adım: and, ant, dad, do



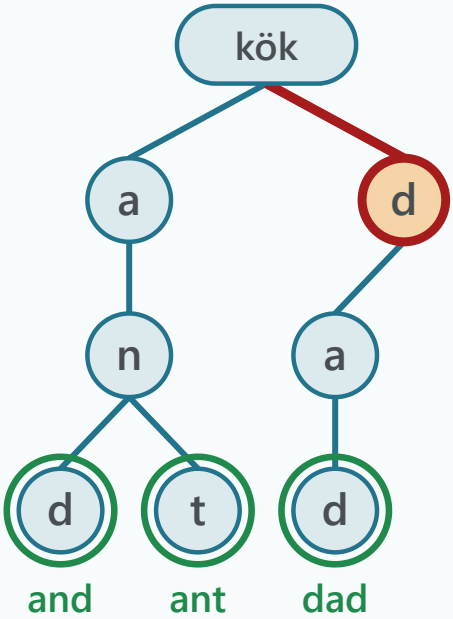
"dad" ekleniyor, karakter 'd': `cocuk[3]` null → **yeni düğüm** oluşturulur ve ona ilerlenir.

Ekleme Adım Adım: and, ant, dad, do



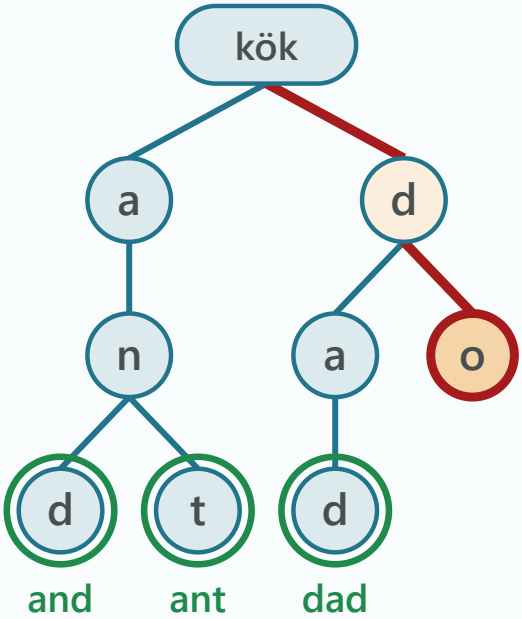
Karakterler bitti: son düğümde `kelimeSonu = true` (çift çember). "dad" eklendi.

Ekleme Adım Adım: and, ant, dad, do



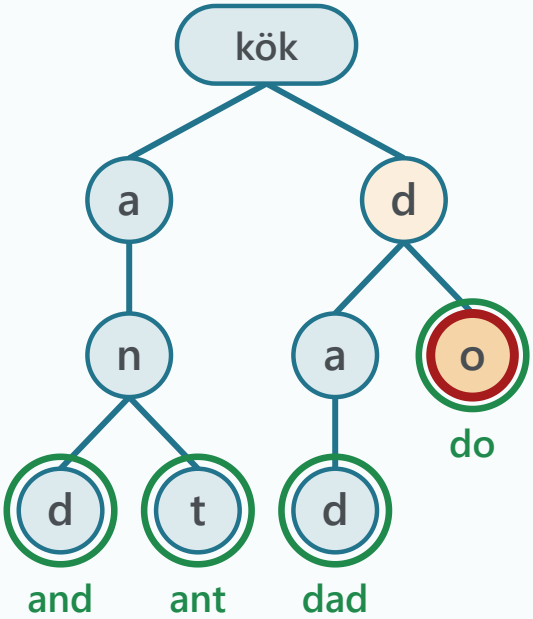
"do" ekleniyor, karakter 'd': `cocuk[3]` dolu → **mevcut** "d" düğümüne ilerlenir (ortak önek).

Ekleme Adım Adım: and, ant, dad, do



"do" ekleniyor, karakter 'o': `cocuk[14]` null → **yeni düğüm** oluşturulur ve ona ilerlenir.

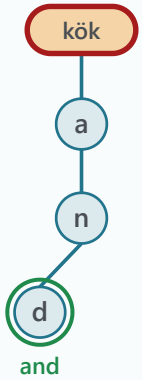
Ekleme Adım Adım: and, ant, dad, do



Karakterler bitti: son düğümde `kelimeSonu = true` (çift çember). "do" eklendi. Son durum: 4 kelime, 11 karakter, kök hariç **8 düğüm**.

ekle("ant"): Kodu Adım Adım İzle

```
1  public void ekle(String kelime) {
▶    TrieDugumu dugum = kok;
3    for (char ch : kelime.toCharArray()) {
4      int i = ch - 'a';
5      if (dugum.cocuk[i] == null)
6        dugum.cocuk[i] = new TrieDugumu();
7      dugum = dugum.cocuk[i];
8    }
9    dugum.kelimeSonu = true;
10 }
```



and

Trie'de yalnızca "and" var. `dugum` kökten başlar.

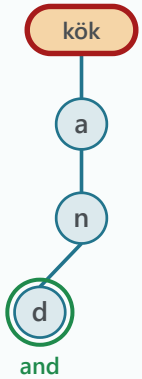
Adım 1 / 12

DEĞİŞKENLER

`kelime = "ant"` `dugum = kök` `ch = –` `i = –`

ekle("ant"): Kodu Adım Adım İzle

```
1  public void ekle(String kelime) {
2    TrieDugumu dugum = kok;
▶    for (char ch : kelime.toCharArray()) {
▶      int i = ch - 'a';
5      if (dugum.cocuk[i] == null)
6        dugum.cocuk[i] = new TrieDugumu();
7      dugum = dugum.cocuk[i];
8    }
9    dugum.kelimeSonu = true;
10 }
```



Karakter 'a' → `i = 'a' - 'a' = 0` .

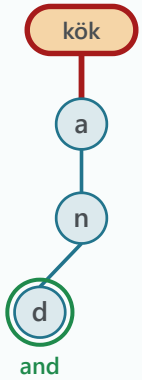
Adım 2 / 12

DEĞİŞKENLER

`kelime = "ant"` `dugum = kök` `ch = 'a'` `i = 0`

ekle("ant"): Kodu Adım Adım İzle

```
1  public void ekle(String kelime) {
2    TrieDugumu dugum = kok;
3    for (char ch : kelime.toCharArray()) {
4      int i = ch - 'a';
▶      if (dugum.cocuk[i] == null)
6        dugum.cocuk[i] = new TrieDugumu();
7      dugum = dugum.cocuk[i];
8    }
9    dugum.kelimeSonu = true;
10 }
```



and

`dugum.cocuk[0]` null değil → yeni düğüm **gerekmez**.

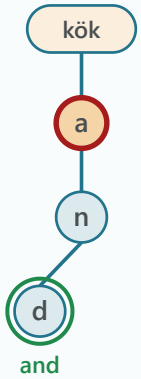
Adım 3 / 12

DEĞİŞKENLER

`kelime = "ant"` `dugum = kök` `ch = 'a'` `i = 0`

ekle("ant"): Kodu Adım Adım İzle

```
1  public void ekle(String kelime) {
2    TrieDugumu dugum = kok;
3    for (char ch : kelime.toCharArray()) {
4      int i = ch - 'a';
5      if (dugum.cocuk[i] == null)
6        dugum.cocuk[i] = new TrieDugumu();
▶      dugum = dugum.cocuk[i];
8    }
9    dugum.kelimeSonu = true;
10 }
```



`dugum` artık "a" düğümü.

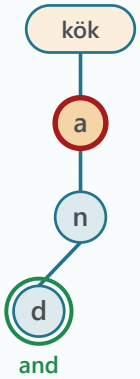
Adım 4 / 12

DEĞİŞKENLER

`kelime = "ant"` `dugum = "a"` `ch = 'a'` `i = 0`

ekle("ant"): Kodu Adım Adım İzle

```
1  public void ekle(String kelime) {
2    TrieDugumu dugum = kok;
▶    for (char ch : kelime.toCharArray()) {
▶      int i = ch - 'a';
5      if (dugum.cocuk[i] == null)
6        dugum.cocuk[i] = new TrieDugumu();
7      dugum = dugum.cocuk[i];
8    }
9    dugum.kelimeSonu = true;
10 }
```



Karakter 'n' → `i = 'n' - 'a' = 13` .

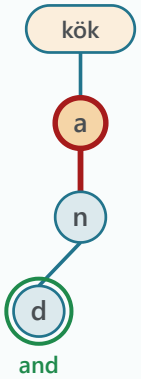
Adım 5 / 12

DEĞİŞKENLER

`kelime = "ant"` `dugum = "a"` `ch = 'n'` `i = 13`

ekle("ant"): Kodu Adım Adım İzle

```
1  public void ekle(String kelime) {
2    TrieDugumu dugum = kok;
3    for (char ch : kelime.toCharArray()) {
4      int i = ch - 'a';
▶      if (dugum.cocuk[i] == null)
6        dugum.cocuk[i] = new TrieDugumu();
7      dugum = dugum.cocuk[i];
8    }
9    dugum.kelimeSonu = true;
10 }
```



`dugum.cocuk[13]` null değil → yeni düğüm **gerekmez**.

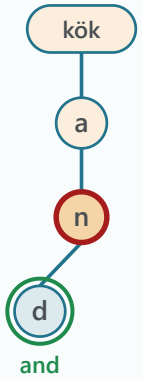
Adım 6 / 12

DEĞİŞKENLER

`kelime = "ant"` `dugum = "a"` `ch = 'n'` `i = 13`

ekle("ant"): Kodu Adım Adım İzle

```
1  public void ekle(String kelime) {
2    TrieDugumu dugum = kok;
3    for (char ch : kelime.toCharArray()) {
4      int i = ch - 'a';
5      if (dugum.cocuk[i] == null)
6        dugum.cocuk[i] = new TrieDugumu();
▶      dugum = dugum.cocuk[i];
8    }
9    dugum.kelimeSonu = true;
10 }
```



`dugum` artık "an" düğümü.

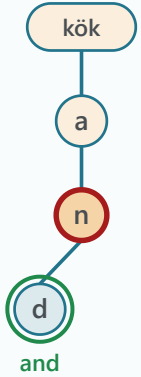
Adım 7 / 12

DEĞİŞKENLER

`kelime = "ant"` `dugum = "an"` `ch = 'n'` `i = 13`

ekle("ant"): Kodu Adım Adım İzle

```
1  public void ekle(String kelime) {
2    TrieDugumu dugum = kok;
▶    for (char ch : kelime.toCharArray()) {
▶      int i = ch - 'a';
5      if (dugum.cocuk[i] == null)
6        dugum.cocuk[i] = new TrieDugumu();
7      dugum = dugum.cocuk[i];
8    }
9    dugum.kelimeSonu = true;
10 }
```



Karakter 't' → `i = 't' - 'a' = 19` .

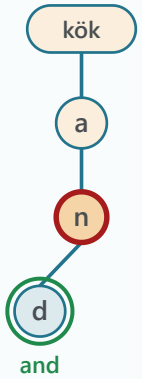
Adım 8 / 12

DEĞİŞKENLER

`kelime = "ant"` `dugum = "an"` `ch = 't'` `i = 19`

ekle("ant"): Kodu Adım Adım İzle

```
1  public void ekle(String kelime) {
2    TrieDugumu dugum = kok;
3    for (char ch : kelime.toCharArray()) {
4      int i = ch - 'a';
▶      if (dugum.cocuk[i] == null)
6        dugum.cocuk[i] = new TrieDugumu();
7      dugum = dugum.cocuk[i];
8    }
9    dugum.kelimeSonu = true;
10 }
```



`dugum.cocuk[19]` == null → koşul doğru.

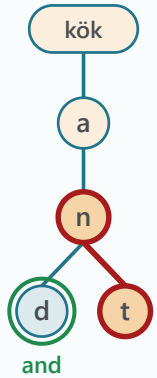
Adım 9 / 12

DEĞİŞKENLER

`kelime = "ant"` `dugum = "an"` `ch = 't'` `i = 19`

ekle("ant"): Kodu Adım Adım İzle

```
1  public void ekle(String kelime) {
2    TrieDugumu dugum = kok;
3    for (char ch : kelime.toCharArray()) {
4      int i = ch - 'a';
5      if (dugum.cocuk[i] == null)
▶        dugum.cocuk[i] = new TrieDugumu();
7      dugum = dugum.cocuk[i];
8    }
9    dugum.kelimeSonu = true;
10 }
```



and

Yeni bir `TrieDugumu` oluşturulup `cocuk[19]`'e bağlanır (`kelimeSonu` = false).

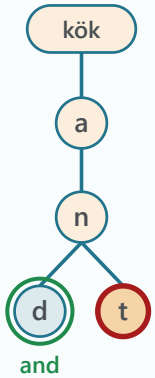
Adım 10 / 12

DEĞİŞKENLER

`kelime = "ant"` `dugum = "an"` `ch = 't'` `i = 19`

ekle("ant"): Kodu Adım Adım İzle

```
1  public void ekle(String kelime) {
2    TrieDugumu dugum = kok;
3    for (char ch : kelime.toCharArray()) {
4      int i = ch - 'a';
5      if (dugum.cocuk[i] == null)
6        dugum.cocuk[i] = new TrieDugumu();
▶      dugum = dugum.cocuk[i];
8    }
9    dugum.kelimeSonu = true;
10 }
```



`dugum` artık "ant" düğümü.

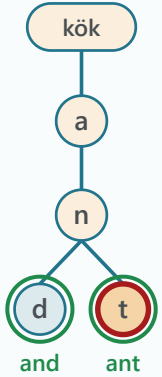
Adım 11 / 12

DEĞİŞKENLER

`kelime = "ant"` `dugum = "ant"` `ch = 't'` `i = 19`

ekle("ant"): Kodu Adım Adım İzle

```
1  public void ekle(String kelime) {
2    TrieDugumu dugum = kok;
3    for (char ch : kelime.toCharArray()) {
4      int i = ch - 'a';
5      if (dugum.cocuk[i] == null)
6        dugum.cocuk[i] = new TrieDugumu();
7      dugum = dugum.cocuk[i];
8    }
▶    dugum.kelimeSonu = true;
10 }
```



Döngü bitti: `dugum.kelimeSonu = true`. "ant" eklendi; yalnızca 1 düğüm oluşturuldu.

Adım 12 / 12

DEĞİŞKENLER

`kelime = "ant"` `dugum = "ant"` `ch = 't'` `i = 19`

ARAMA

Arama (Search)

```
public boolean ara(String kelime) {
  TrieDugumu dugum = kok;
  for (char ch : kelime.toCharArray()) {
    int i = ch - 'a';
    if (dugum.cocuk[i] == null)
      return false;
    dugum = dugum.cocuk[i];
  }
  return dugum.kelimeSonu;
}
```

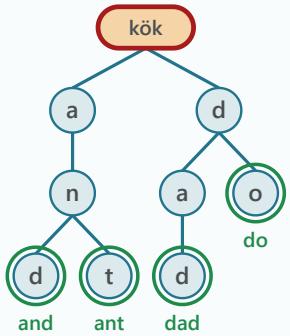
- **Kökten** başlanır.
- Her karakter için ilgili **dala** (çocuğa) gidilir.
  - Dal yoksa (`null`) kelime trie'de **yoktur**.
- Kelimenin sonuna gelince `kelimeSonu` denetlenir:
  - `true` → kelime **bulundu**;
  - `false` → kelime yok; yalnızca var olan bir kelimenin **önekidir**.

Zaman: **O(L)**. Trie'de bir milyon kelime de olsa "dad" en fazla 3 adımda bulunur.

ARAMA · KODU ADIM ADIM ÇALIŞTIR

ara("dad"): Kodu Adım Adım İzle

```
1  public boolean ara(String kelime) {
▶    TrieDugumu dugum = kok;
3    for (char ch : kelime.toCharArray()) {
4      int i = ch - 'a';
5      if (dugum.cocuk[i] == null)
6        return false;
7      dugum = dugum.cocuk[i];
8    }
9    return dugum.kelimeSonu;
10 }
```



Trie: and, ant, dad, do. `dugum` kökten başlar.

Adım 1 / 11

DEĞİŞKENLER

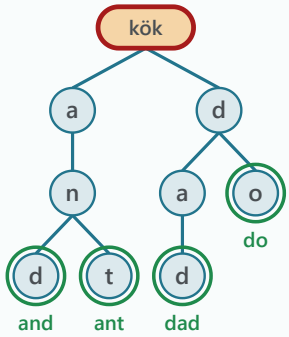
`kelime = "dad"` `dugum = kök` `ch = –` `i = –`

ÇIKTI

ARAMA · KODU ADIM ADIM ÇALIŞTIR

ara("dad"): Kodu Adım Adım İzle

```
1  public boolean ara(String kelime) {
2    TrieDugumu dugum = kok;
▶    for (char ch : kelime.toCharArray()) {
▶      int i = ch - 'a';
5      if (dugum.cocuk[i] == null)
6        return false;
7      dugum = dugum.cocuk[i];
8    }
9    return dugum.kelimeSonu;
10 }
```



Karakter 'd' → `i = 3`.

Adım 2 / 11

DEĞİŞKENLER

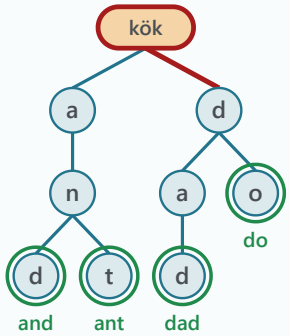
`kelime = "dad"` `dugum = kök` `ch = 'd'` `i = 3`

ÇIKTI

ARAMA · KODU ADIM ADIM ÇALIŞTIR

ara("dad"): Kodu Adım Adım İzle

```
1  public boolean ara(String kelime) {
2    TrieDugumu dugum = kok;
3    for (char ch : kelime.toCharArray()) {
4      int i = ch - 'a';
▶      if (dugum.cocuk[i] == null)
6        return false;
7      dugum = dugum.cocuk[i];
8    }
9    return dugum.kelimeSonu;
10 }
```



`cocuk[3]` null değil → dal var, devam.

Adım 3 / 11

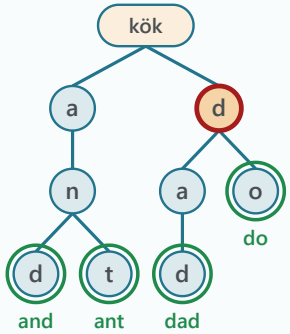
DEĞİŞKENLER

`kelime = "dad"` `dugum = kök` `ch = 'd'` `i = 3`

ÇIKTI

ara("dad"): Kodu Adım Adım İzle

```
1  public boolean ara(String kelime) {
2    TrieDugumu dugum = kok;
3    for (char ch : kelime.toCharArray()) {
4      int i = ch - 'a';
5      if (dugum.cocuk[i] == null)
6        return false;
▶      dugum = dugum.cocuk[i];
8    }
9    return dugum.kelimeSonu;
10 }
```



`dugum` = "d".

Adım 4 / 11

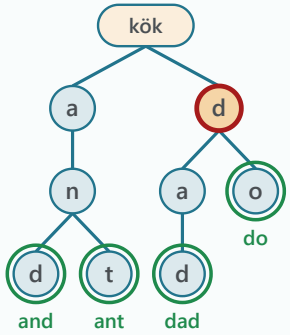
DEĞİŞKENLER

`kelime = "dad"` `dugum = "d"` `ch = 'd'` `i = 3`

ÇIKTI

ara("dad"): Kodu Adım Adım İzle

```
1  public boolean ara(String kelime) {
2    TrieDugumu dugum = kok;
▶    for (char ch : kelime.toCharArray()) {
▶      int i = ch - 'a';
5      if (dugum.cocuk[i] == null)
6        return false;
7      dugum = dugum.cocuk[i];
8    }
9    return dugum.kelimeSonu;
10 }
```



Karakter 'a' → `i = 0`.

Adım 5 / 11

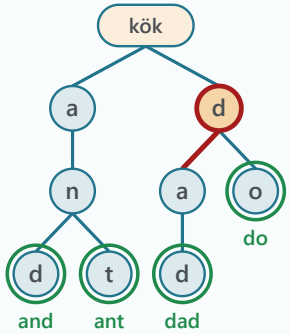
DEĞİŞKENLER

`kelime = "dad"` `dugum = "d"` `ch = 'a'` `i = 0`

ÇIKTI

ara("dad"): Kodu Adım Adım İzle

```
1  public boolean ara(String kelime) {
2    TrieDugumu dugum = kok;
3    for (char ch : kelime.toCharArray()) {
4      int i = ch - 'a';
▶      if (dugum.cocuk[i] == null)
6        return false;
7      dugum = dugum.cocuk[i];
8    }
9    return dugum.kelimeSonu;
10 }
```



`cocuk[0]` null değil → dal var, devam.

Adım 6 / 11

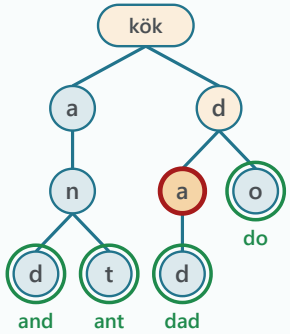
DEĞİŞKENLER

`kelime = "dad"` `dugum = "d"` `ch = 'a'` `i = 0`

ÇIKTI

ara("dad"): Kodu Adım Adım İzle

```
1  public boolean ara(String kelime) {
2    TrieDugumu dugum = kok;
3    for (char ch : kelime.toCharArray()) {
4      int i = ch - 'a';
5      if (dugum.cocuk[i] == null)
6        return false;
▶      dugum = dugum.cocuk[i];
8    }
9    return dugum.kelimeSonu;
10 }
```



`dugum` = "da".

Adım 7 / 11

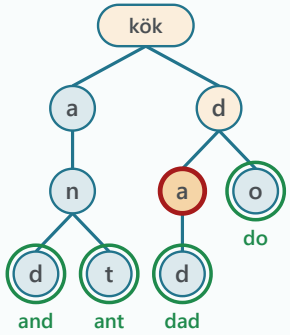
DEĞİŞKENLER

`kelime = "dad"` `dugum = "da"` `ch = 'a'` `i = 0`

ÇIKTI

ara("dad"): Kodu Adım Adım İzle

```
1  public boolean ara(String kelime) {
2    TrieDugumu dugum = kok;
▶    for (char ch : kelime.toCharArray()) {
▶      int i = ch - 'a';
5      if (dugum.cocuk[i] == null)
6        return false;
7      dugum = dugum.cocuk[i];
8    }
9    return dugum.kelimeSonu;
10 }
```



Karakter 'd' → `i = 3`.

Adım 8 / 11

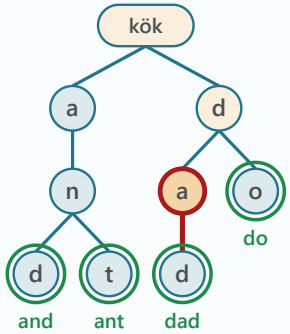
DEĞİŞKENLER

`kelime = "dad"` `dugum = "da"` `ch = 'd'` `i = 3`

ÇIKTI

ara("dad"): Kodu Adım Adım İzle

```
1  public boolean ara(String kelime) {
2    TrieDugumu dugum = kok;
3    for (char ch : kelime.toCharArray()) {
4      int i = ch - 'a';
▶      if (dugum.cocuk[i] == null)
6        return false;
7      dugum = dugum.cocuk[i];
8    }
9    return dugum.kelimeSonu;
10 }
```



`cocuk[3]` null değil → dal var, devam.

Adım 9 / 11

DEĞİŞKENLER

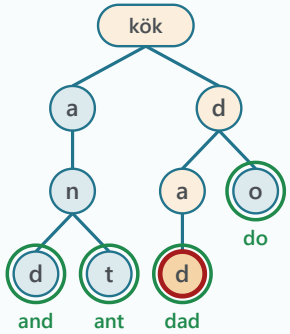
`kelime = "dad"` `dugum = "da"` `ch = 'd'` `i = 3`

ÇIKTI

ARAMA · KODU ADIM ADIM ÇALIŞTIR

ara("dad"): Kodu Adım Adım İzle

```
1  public boolean ara(String kelime) {
2    TrieDugumu dugum = kok;
3    for (char ch : kelime.toCharArray()) {
4      int i = ch - 'a';
5      if (dugum.cocuk[i] == null)
6        return false;
▶      dugum = dugum.cocuk[i];
8    }
9    return dugum.kelimeSonu;
10 }
```



`dugum` = "dad".

Adım 10 / 11

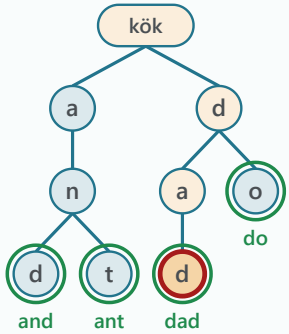
DEĞİŞKENLER

`kelime = "dad"` `dugum = "dad"` `ch = 'd'` `i = 3`

ÇIKTI

ara("dad"): Kodu Adım Adım İzle

```
1  public boolean ara(String kelime) {
2    TrieDugumu dugum = kok;
3    for (char ch : kelime.toCharArray()) {
4      int i = ch - 'a';
5      if (dugum.cocuk[i] == null)
6        return false;
7      dugum = dugum.cocuk[i];
8    }
▶    return dugum.kelimeSonu;
10 }
```



Kelime bitti; `dugum.kelimeSonu` = true → **true** döner (kelime bulundu).

Adım 11 / 11

DEĞİŞKENLER

`kelime = "dad"` `dugum = "dad"` `ch = 'd'` `i = 3`

ÇIKTI

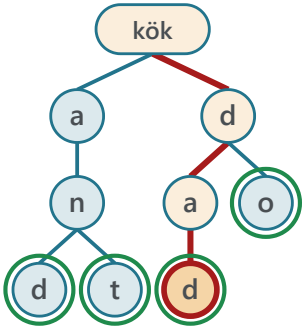
```
ara("dad") → true
```

ARAMA

Aramanın Üç Olası Sonucu

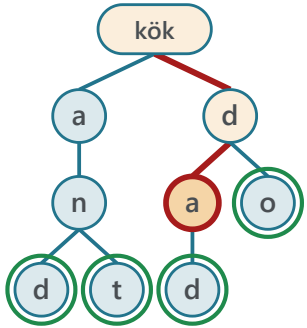
ara("dad") → true

Tüm karakterler bulundu ve son düğüm işaretli.



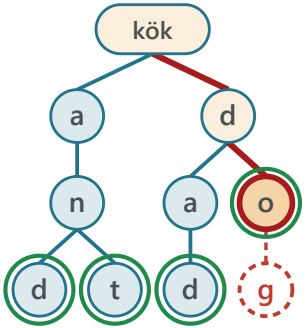
ara("da") → false

Yol var, ama "da" düğümünde `kelimeSonu` = false: "da" yalnızca **önek**.



ara("dog") → false

"do" düğümünde `cocuk[6]` ('g') null: **dal yok**, erken çıkış.



Kesikli kırmızı çember: aranan ama trie'de bulunmayan düğüm.

ARAMA

Önek Sorgusu: `onekVarMi` (startsWith)

```
public boolean onekVarMi(String onek) {
  TrieDugumu dugum = kok;
  for (char ch : onek.toCharArray()) {
    int i = ch - 'a';
    if (dugum.cocuk[i] == null)
      return false;
    dugum = dugum.cocuk[i];
  }
  return true;  // ara: dugum.kelimeSonu
}
```

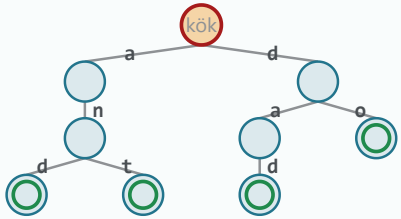
Sorgu	ara	onekVarMi
"dad"	true	true
"da"	false	true
"an"	false	true
"dog"	false	false

Tek fark **son satırdadır**: önek sorgusu kelime sonunu denetlemez; yolun var olması yeterlidir. Trie: and, ant, dad, do. Zaman: O(P), P = önek uzunluğu.

ARAMA · KODU ADIM ADIM ÇALIŞTIR

onekVarMi("da") ve onekVarMi("dog")

```
1  public boolean onekVarMi(String onek) {
▶    TrieDugumu dugum = kok;
3    for (char ch : onek.toCharArray()) {
4      int i = ch - 'a';
5      if (dugum.cocuk[i] == null)
6        return false;
7      dugum = dugum.cocuk[i];
8    }
9    return true;  // ara: dugum.kelimeSonu
10 }
```



Trie'de and, ant, dad, do var (halkalı düğüm = kelime sonu). Önce onekVarMi("da"): dugum köke konur.

Adım 1 / 8

DEĞİŞKENLER

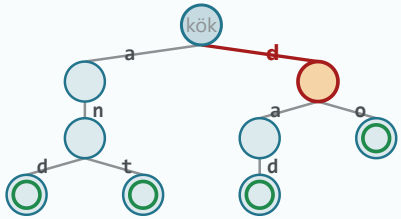
onek = "da" | ch = – | i = – | dugum = kök

ÇIKTI

ARAMA · KODU ADIM ADIM ÇALIŞTIR

onekVarMi("da") ve onekVarMi("dog")

```
1  public boolean onekVarMi(String onek) {
2    TrieDugumu dugum = kok;
3    for (char ch : onek.toCharArray()) {
▶      int i = ch - 'a';
▶      if (dugum.cocuk[i] == null)
6        return false;
▶      dugum = dugum.cocuk[i];
8    }
9    return true;  // ara: dugum.kelimeSonu
10 }
```



ch = 'd', i = 3: cocuk[3] dolu → dugum o çocuğa iner ("d").

Adım 2 / 8

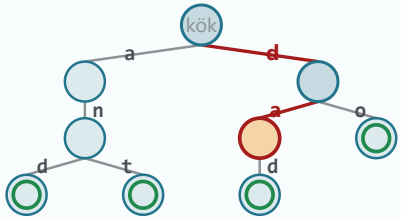
DEĞİŞKENLER

onek = "da" | ch = 'd' | i = 3 | dugum = "d"

ÇIKTI

onekVarMi("da") ve onekVarMi("dog")

```
1  public boolean onekVarMi(String onek) {
2    TrieDugumu dugum = kok;
3    for (char ch : onek.toCharArray()) {
▶      int i = ch - 'a';
▶      if (dugum.cocuk[i] == null)
6        return false;
▶      dugum = dugum.cocuk[i];
8    }
9    return true;  // ara: dugum.kelimeSonu
10 }
```



ch = 'a', i = 0: cocuk[0] dolu → dugum o çocuğa iner ("da").

Adım 3 / 8

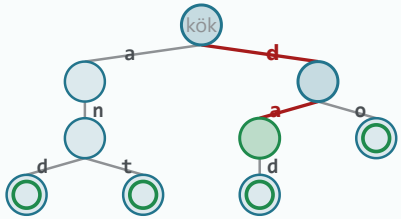
DEĞİŞKENLER

`onek = "da"` `ch = 'a'` `i = 0` `dugum = "da"`

ÇIKTI

onekVarMi("da") ve onekVarMi("dog")

```
1  public boolean onekVarMi(String onek) {
2    TrieDugumu dugum = kok;
3    for (char ch : onek.toCharArray()) {
4      int i = ch - 'a';
5      if (dugum.cocuk[i] == null)
6        return false;
7      dugum = dugum.cocuk[i];
8    }
▶    return true;  // ara: dugum.kelimeSonu
10 }
```



Önek bitti ve yol kesilmedi → **true**. "da" düğümü işaretsiz olduğu için `ara("da")` false verirdi; önek sorgusu kelime sonuna bakmaz.

Adım 4 / 8

DEĞİŞKENLER

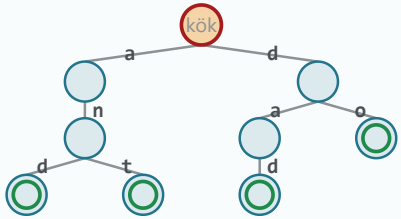
`onek = "da"` `ch = 'a'` `i = 0` `dugum = "da"`

ÇIKTI

```
onekVarMi("da") = true
```

onekVarMi("da") ve onekVarMi("dog")

```
1  public boolean onekVarMi(String onek) {
2    TrieDugumu dugum = kok;
3    for (char ch : onek.toCharArray()) {
4      int i = ch - 'a';
5      if (dugum.cocuk[i] == null)
6        return false;
7      dugum = dugum.cocuk[i];
8    }
9    return true;  // ara: dugum.kelimeSonu
10 }
```



İkinci çağrı onekVarMi("dog"): dugum yeniden köke konur.

Adım 5 / 8

DEĞİŞKENLER

onek = "dog" | ch = – | i = – | dugum = kök

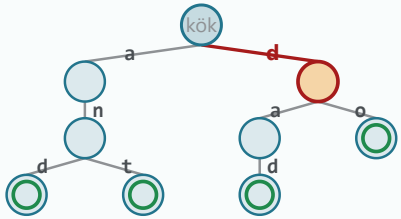
ÇIKTI

```
onekVarMi("da") = true
```

ARAMA · KODU ADIM ADIM ÇALIŞTIR

onekVarMi("da") ve onekVarMi("dog")

```
1  public boolean onekVarMi(String onek) {
2    TrieDugumu dugum = kok;
3    for (char ch : onek.toCharArray()) {
▶      int i = ch - 'a';
▶      if (dugum.cocuk[i] == null)
6        return false;
▶      dugum = dugum.cocuk[i];
8    }
9    return true;  // ara: dugum.kelimeSonu
10 }
```



ch = 'd', i = 3: cocuk[3] dolu → dugum o çocuğa iner ("d").

Adım 6 / 8

DEĞİŞKENLER

onek = "dog"

ch = 'd' | i = 3

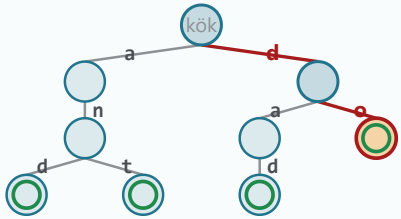
dugum = "d"

ÇIKTI

```
onekVarMi("da") = true
```

onekVarMi("da") ve onekVarMi("dog")

```
1  public boolean onekVarMi(String onek) {
2    TrieDugumu dugum = kok;
3    for (char ch : onek.toCharArray()) {
▶      int i = ch - 'a';
▶      if (dugum.cocuk[i] == null)
6        return false;
▶      dugum = dugum.cocuk[i];
8    }
9    return true;  // ara: dugum.kelimeSonu
10 }
```



ch = 'o', i = 14: cocuk[14] dolu → dugum o çocuğa iner ("do").

Adım 7 / 8

DEĞİŞKENLER

onek = "dog"

ch = 'o' i = 14

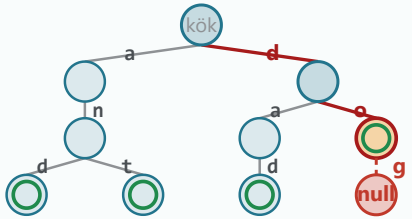
dugum = "do"

ÇIKTI

```
onekVarMi("da") = true
```

onekVarMi("da") ve onekVarMi("dog")

```
1  public boolean onekVarMi(String onek) {
2    TrieDugumu dugum = kok;
3    for (char ch : onek.toCharArray()) {
▶      int i = ch - 'a';
▶      if (dugum.cocuk[i] == null)
▶        return false;
7      dugum = dugum.cocuk[i];
8    }
9    return true;  // ara: dugum.kelimeSonu
10 }
```



ch = 'g', i = 6: "do" düğümünün cocuk[6]'sı null → **false**. "do" bir kelime olsa da "dog" ile başlayan hiçbir kelime yok.

Adım 8 / 8

DEĞİŞKENLER

onek = **"dog"**

ch = **'g'** i = **6**

dugum = **"do"**

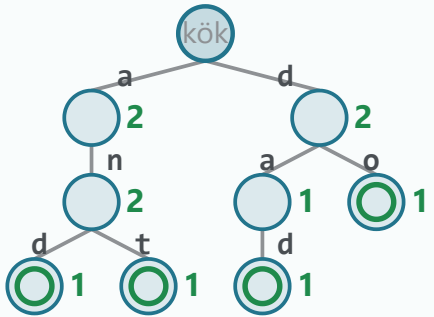
ÇIKTI

```
onekVarMi("da") = true
onekVarMi("dog") = false
```

ARAMA

Önek Sayacı: `onekSayisi` O(P)

```
void ekle(String kelime) {
  TrieDugumu d = kok;
  d.gecen++;
  for (char ch : kelime.toCharArray()) {
    int i = ch - 'a';
    if (d.cocuk[i] == null) d.cocuk[i] = new TrieDugumu();
    d = d.cocuk[i];
    d.gecen++;          // bir kelime daha geçti
  }
  d.kelimeSonu = true;
}
int onekSayisi(String onek) {
  TrieDugumu d = kok;
  for (char ch : onek.toCharArray()) {
    d = d.cocuk[ch - 'a'];
    if (d == null) return 0;
  }
  return d.gecen;
}
```



Yeşil sayılar: gecen (kökte 4).

Sorgu	sonuç
onekSayisi("an")	2
onekSayisi("da")	1
onekSayisi("x")	0

`TrieDugumu` 'na `int gecen` alanı eklenir. Sayaçsız aynı soru alt ağacı gezmeyi (O(P + S · M)) gerektirirdi; bedeli düğüm başına bir `int` . Silmede yol boyunca sayaçlar azaltılır.

ARAMA

Joker Karakterli Arama: `"d.d"`

```
boolean joker(TrieDugumu d, String s, int j) {
  if (d == null) return false;
  if (j == s.length()) return d.kelimeSonu;
  char ch = s.charAt(j);
  if (ch != '.')
    return joker(d.cocuk[ch - 'a'], s, j + 1);
  for (int i = 0; i < 26; i++)
    if (joker(d.cocuk[i], s, j + 1)) return true;
  return false;
}
```

Desen	sonuç	eşleşen
"d.d"	true	dad
".n."	true	and
".o"	true	do
"b.."	false	b dalı yok
"...."	false	4 harfli yok

- Trie: and, ant, dad, do. Çağrı `joker(kok, desen, 0)`.
- Normal harf aramayı tek yola indirir; '.' her dolu dalı dener (geri izleme). İlk eşleşmede `true` yukarı taşınır.
- En kötü durum k joker için 26^k dal, ama boş dallar hemen döner: gerçek maliyet trie'de var olan yollarla sınırlıdır. Hash tablosu bu sorguyu ancak tüm kelimeleri tarayarak yanıtlar.

ARAMA

Yazım Denetimi: Trie Üzerinde Düzenleme Uzaklığı

- Soru: sözlükte "dod" kelimesine düzenleme uzaklığı (ekle / sil / değiştir) **en çok 1** olan kelimeler hangileri?
- Her kelimeyle ayrı ayrı uzaklık hesaplamak yerine trie derinlik öncelikli gezilir; her düğüm, ebeveyninin DP satırından **tek bir satır** üretir.
- Ortak önekin satırı bir kez hesaplanır: "d" satırı hem dad hem do için kullanılır.
- **Budama:** satırın en küçüğü k'yi aşarsa alt ağaçtaki hiçbir kelime k içinde olamaz; "an" satırının en küçüğü 2 > 1, and ve ant'a hiç inilmez.

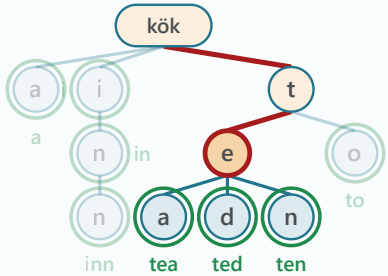
düğüm	"" d o d	son	karar
a	1 1 2 3	—	devam
an	2 2 2 3	—	buda (min 2)
d	1 0 1 2	—	devam
da	2 1 1 2	—	devam
dad	3 2 2 1	1	öner
do	2 1 0 1	1	öner

Satırın son hücresi, düğümün kelimesi ile "dod" arasındaki uzaklıktır.

Öneriler: **dad** (o → a) ve **do** (son d silinir). Satır başına O(|sorgu|) iş; budama sayesinde sözlüğün çoğu hiç ziyaret edilmez. Aynı gezinti klavyede yan yana harfleri ucuz sayan ağırlıklı uzaklıkla da çalışır; k büyüdükçe budama zayıflar ve ziyaret edilen düğüm sayısı hızla artar.

Otomatik Tamamlama: tamamla("te")

```
1  List<String> tamamla(String onek) {
▶    List<String> sonuc = new ArrayList<>();
▶    TrieDugumu d = dugumBul(onek);
4    if (d != null)
5      topla(d, new StringBuilder(onek), sonuc);
6    return sonuc;
7  }
8  void topla(TrieDugumu d, StringBuilder yol,
9             List<String> sonuc) {
10   if (d.kelimeSonu)
   sonuc.add(yol.toString());
11   for (int i = 0; i < 26; i++)
12     if (d.cocuk[i] != null) {
13       yol.append((char) ('a' + i));
14       topla(d.cocuk[i], yol, sonuc);
15       yol.setLength(yol.length() - 1);
16     }
17 }
```



Önce `dugumBul("te")` kökten t → e inerek "te" düğümünü bulur (önek sorgusu gibi).

Adım 1 / 16

DEĞİŞKENLER

`onek = "te"`

`yol = –` `i = –`

ÇAĞRI YIĞINI

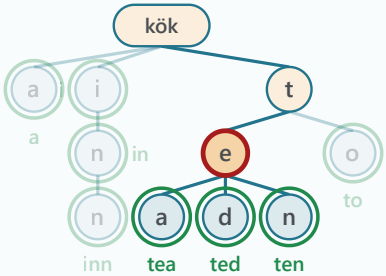
`tamamla("te")`

ÇIKTI

```
sonuc = []
```

Otomatik Tamamlama: tamamla("te")

```
1  List<String> tamamla(String onek) {
2    List<String> sonuc = new ArrayList<>();
3    TrieDugumu d = dugumBul(onek);
4    if (d != null)
▶      topla(d, new StringBuilder(onek), sonuc);
6    return sonuc;
7  }
8  void topla(TrieDugumu d, StringBuilder yol,
9             List<String> sonuc) {
10   if (d.kelimeSonu)
   sonuc.add(yol.toString());
11   for (int i = 0; i < 26; i++)
12     if (d.cocuk[i] != null) {
13       yol.append((char) ('a' + i));
14       topla(d.cocuk[i], yol, sonuc);
15       yol.setLength(yol.length() - 1);
16     }
17 }
```



"te" düğümünden başlayarak alt ağaç **derinlik öncelikli** gezilir.

Adım 2 / 16

DEĞİŞKENLER

`onek = "te"`

`yol = "te"` `i = –`

ÇAĞRI YIĞINI

`topla("te")`

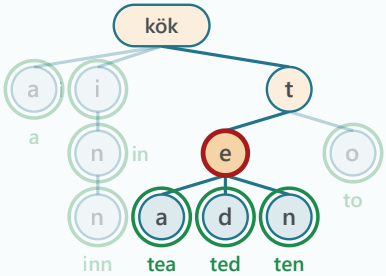
`tamamla("te")`

ÇIKTI

```
sonuc = []
```

Otomatik Tamamlama: tamamla("te")

```
1  List<String> tamamla(String onek) {
2    List<String> sonuc = new ArrayList<>();
3    TrieDugumu d = dugumBul(onek);
4    if (d != null)
5      topla(d, new StringBuilder(onek), sonuc);
6    return sonuc;
7  }
8  void topla(TrieDugumu d, StringBuilder yol,
9             List<String> sonuc) {
▶    if (d.kelimeSonu)
   sonuc.add(yol.toString());
11   for (int i = 0; i < 26; i++)
12     if (d.cocuk[i] != null) {
13       yol.append((char) ('a' + i));
14       topla(d.cocuk[i], yol, sonuc);
15       yol.setLength(yol.length() - 1);
16     }
17 }
```



"te" bir kelime değil (`kelimeSonu` = false) → listeye eklenmez.

Adım 3 / 16

DEĞİŞKENLER

`onek = "te"`

`yol = "te"` `i = –`

ÇAĞRI YIĞINI

`topla("te")`

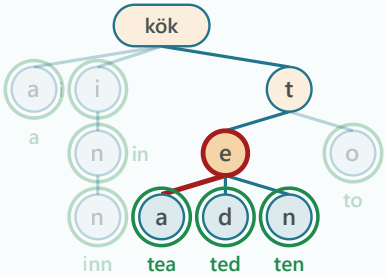
`tamamla("te")`

ÇIKTI

```
sonuc = []
```

Otomatik Tamamlama: tamamla("te")

```
1  List<String> tamamla(String onek) {
2    List<String> sonuc = new ArrayList<>();
3    TrieDugumu d = dugumBul(onek);
4    if (d != null)
5      topla(d, new StringBuilder(onek), sonuc);
6    return sonuc;
7  }
8  void topla(TrieDugumu d, StringBuilder yol,
9             List<String> sonuc) {
10   if (d.kelimeSonu)
   sonuc.add(yol.toString());
11   for (int i = 0; i < 26; i++)
▶      if (d.cocuk[i] != null) {
▶        yol.append((char) ('a' + i));
14       topla(d.cocuk[i], yol, sonuc);
15       yol.setLength(yol.length() - 1);
16     }
17 }
```



`i = 0` : `cocuk[0]` ('a') dolu → `yol` = "tea".

Adım 4 / 16

DEĞİŞKENLER

onek = **"te"**

yol = **"tea"** i = **0**

ÇAĞRI YIĞINI

topla("te")

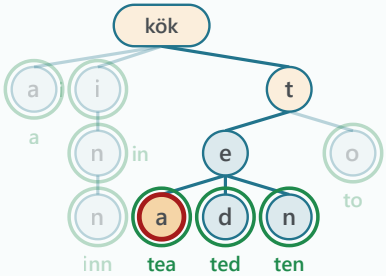
tamamla("te")

ÇIKTI

```
sonuc = []
```

Otomatik Tamamlama: tamamla("te")

```
1  List<String> tamamla(String onek) {
2    List<String> sonuc = new ArrayList<>();
3    TrieDugumu d = dugumBul(onek);
4    if (d != null)
5      topla(d, new StringBuilder(onek), sonuc);
6    return sonuc;
7  }
8  void topla(TrieDugumu d, StringBuilder yol,
9             List<String> sonuc) {
10   if (d.kelimeSonu)
   sonuc.add(yol.toString());
11   for (int i = 0; i < 26; i++)
12     if (d.cocuk[i] != null) {
13       yol.append((char) ('a' + i));
▶        topla(d.cocuk[i], yol, sonuc);
15       yol.setLength(yol.length() - 1);
16     }
17 }
```



Özyinelemeli çağrı: `topla("tea")`.

Adım 5 / 16

DEĞİŞKENLER

`onek = "te"`

`yol = "tea"` `i = –`

ÇAĞRI YIĞINI

`topla("tea")`

`topla("te")`

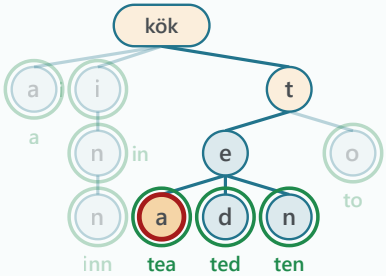
`tamamla("te")`

ÇIKTI

```
sonuc = []
```

Otomatik Tamamlama: tamamla("te")

```
1  List<String> tamamla(String onek) {
2    List<String> sonuc = new ArrayList<>();
3    TrieDugumu d = dugumBul(onek);
4    if (d != null)
5      topla(d, new StringBuilder(onek), sonuc);
6    return sonuc;
7  }
8  void topla(TrieDugumu d, StringBuilder yol,
9             List<String> sonuc) {
▶    if (d.kelimeSonu)
   sonuc.add(yol.toString());
11   for (int i = 0; i < 26; i++)
12     if (d.cocuk[i] != null) {
13       yol.append((char) ('a' + i));
14       topla(d.cocuk[i], yol, sonuc);
15       yol.setLength(yol.length() - 1);
16     }
17 }
```



"tea" işaretli → `sonuc` 'a eklenir. Çocuğu yok; döngü 26 indisi boş geçer.

Adım 6 / 16

DEĞİŞKENLER

`onek = "te"`

`yol = "tea"` `i = –`

ÇAĞRI YIĞINI

`topla("tea")`

`topla("te")`

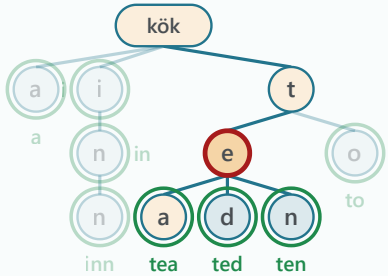
`tamamla("te")`

ÇIKTI

```
sonuc = [tea]
```

Otomatik Tamamlama: tamamla("te")

```
1  List<String> tamamla(String onek) {
2    List<String> sonuc = new ArrayList<>();
3    TrieDugumu d = dugumBul(onek);
4    if (d != null)
5      topla(d, new StringBuilder(onek), sonuc);
6    return sonuc;
7  }
8  void topla(TrieDugumu d, StringBuilder yol,
9             List<String> sonuc) {
10   if (d.kelimeSonu)
   sonuc.add(yol.toString());
11   for (int i = 0; i < 26; i++)
12     if (d.cocuk[i] != null) {
13       yol.append((char) ('a' + i));
14       topla(d.cocuk[i], yol, sonuc);
▶        yol.setLength(yol.length() - 1);
16     }
17 }
```



Çağrı döndü; son harf silinir (`yol` = "te") → geri izleme (backtracking).

Adım 7 / 16

DEĞİŞKENLER

`onek = "te"`

`yol = "te"` `i = 0`

ÇAĞRI YIĞINI

`topla("te")`

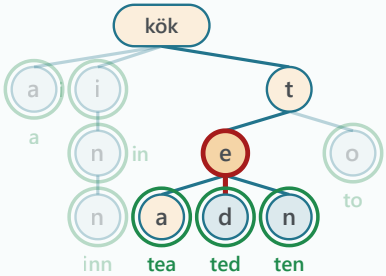
`tamamla("te")`

ÇIKTI

```
sonuc = [tea]
```

Otomatik Tamamlama: tamamla("te")

```
1  List<String> tamamla(String onek) {
2    List<String> sonuc = new ArrayList<>();
3    TrieDugumu d = dugumBul(onek);
4    if (d != null)
5      topla(d, new StringBuilder(onek), sonuc);
6    return sonuc;
7  }
8  void topla(TrieDugumu d, StringBuilder yol,
9             List<String> sonuc) {
10   if (d.kelimeSonu)
   sonuc.add(yol.toString());
11   for (int i = 0; i < 26; i++)
▶      if (d.cocuk[i] != null) {
▶        yol.append((char) ('a' + i));
14       topla(d.cocuk[i], yol, sonuc);
15       yol.setLength(yol.length() - 1);
16     }
17 }
```



`i = 3` : `cocuk[3]` ('d') dolu → `yol` = "ted".

Adım 8 / 16

DEĞİŞKENLER

onek = **"te"**

yol = **"ted"** i = **3**

ÇAĞRI YIĞINI

topla("te")

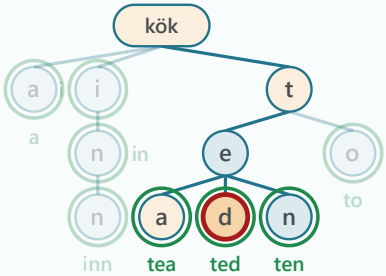
tamamla("te")

ÇIKTI

```
sonuc = [tea]
```

Otomatik Tamamlama: tamamla("te")

```
1  List<String> tamamla(String onek) {
2    List<String> sonuc = new ArrayList<>();
3    TrieDugumu d = dugumBul(onek);
4    if (d != null)
5      topla(d, new StringBuilder(onek), sonuc);
6    return sonuc;
7  }
8  void topla(TrieDugumu d, StringBuilder yol,
9             List<String> sonuc) {
10   if (d.kelimeSonu)
   sonuc.add(yol.toString());
11   for (int i = 0; i < 26; i++)
12     if (d.cocuk[i] != null) {
13       yol.append((char) ('a' + i));
▶        topla(d.cocuk[i], yol, sonuc);
15       yol.setLength(yol.length() - 1);
16     }
17 }
```



Özyinelemeli çağrı: `topla("ted")`.

Adım 9 / 16

DEĞİŞKENLER

`onek = "te"`

`yol = "ted"` `i = –`

ÇAĞRI YIĞINI

`topla("ted")`

`topla("te")`

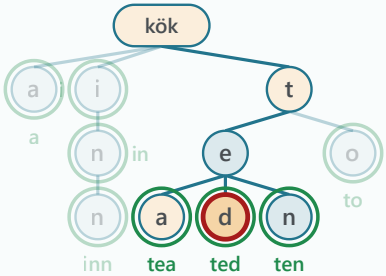
`tamamla("te")`

ÇIKTI

```
sonuc = [tea]
```

Otomatik Tamamlama: tamamla("te")

```
1  List<String> tamamla(String onek) {
2    List<String> sonuc = new ArrayList<>();
3    TrieDugumu d = dugumBul(onek);
4    if (d != null)
5      topla(d, new StringBuilder(onek), sonuc);
6    return sonuc;
7  }
8  void topla(TrieDugumu d, StringBuilder yol,
9             List<String> sonuc) {
▶    if (d.kelimeSonu)
   sonuc.add(yol.toString());
11   for (int i = 0; i < 26; i++)
12     if (d.cocuk[i] != null) {
13       yol.append((char) ('a' + i));
14       topla(d.cocuk[i], yol, sonuc);
15       yol.setLength(yol.length() - 1);
16     }
17 }
```



"ted" işaretli → `sonuc` 'a eklenir. Çocuğu yok; döngü 26 indisi boş geçer.

Adım 10 / 16

DEĞİŞKENLER

`onek = "te"`

`yol = "ted"` `i = –`

ÇAĞRI YIĞINI

`topla("ted")`

`topla("te")`

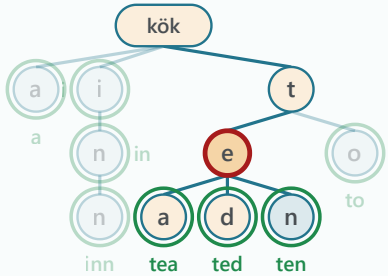
`tamamla("te")`

ÇIKTI

```
sonuc = [tea, ted]
```

Otomatik Tamamlama: tamamla("te")

```
1  List<String> tamamla(String onek) {
2    List<String> sonuc = new ArrayList<>();
3    TrieDugumu d = dugumBul(onek);
4    if (d != null)
5      topla(d, new StringBuilder(onek), sonuc);
6    return sonuc;
7  }
8  void topla(TrieDugumu d, StringBuilder yol,
9             List<String> sonuc) {
10   if (d.kelimeSonu)
     sonuc.add(yol.toString());
11   for (int i = 0; i < 26; i++)
12     if (d.cocuk[i] != null) {
13       yol.append((char) ('a' + i));
14       topla(d.cocuk[i], yol, sonuc);
▶        yol.setLength(yol.length() - 1);
16     }
17 }
```



Çağrı döndü; son harf silinir (`yol` = "te") → geri izleme (backtracking).

Adım 11 / 16

DEĞİŞKENLER

onek = "te"

yol = "te" i = 3

ÇAĞRI YIĞINI

topla("te")

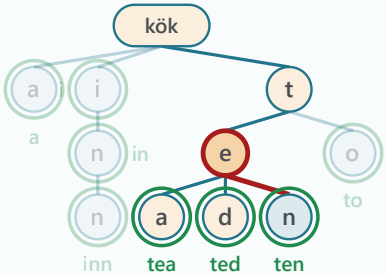
tamamla("te")

ÇIKTI

```
sonuc = [tea, ted]
```

Otomatik Tamamlama: tamamla("te")

```
1  List<String> tamamla(String onek) {
2    List<String> sonuc = new ArrayList<>();
3    TrieDugumu d = dugumBul(onek);
4    if (d != null)
5      topla(d, new StringBuilder(onek), sonuc);
6    return sonuc;
7  }
8  void topla(TrieDugumu d, StringBuilder yol,
9             List<String> sonuc) {
10   if (d.kelimeSonu)
   sonuc.add(yol.toString());
11   for (int i = 0; i < 26; i++)
▶      if (d.cocuk[i] != null) {
▶        yol.append((char) ('a' + i));
14       topla(d.cocuk[i], yol, sonuc);
15       yol.setLength(yol.length() - 1);
16     }
17 }
```



`i = 13` : `cocuk[13]` ('n') dolu → `yol` = "ten".

Adım 12 / 16

DEĞİŞKENLER

onek = **"te"**

yol = **"ten"** i = **13**

ÇAĞRI YIĞINI

topla("te")

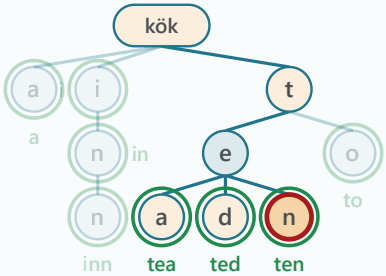
tamamla("te")

ÇIKTI

```
sonuc = [tea, ted]
```

Otomatik Tamamlama: tamamla("te")

```
1  List<String> tamamla(String onek) {
2    List<String> sonuc = new ArrayList<>();
3    TrieDugumu d = dugumBul(onek);
4    if (d != null)
5      topla(d, new StringBuilder(onek), sonuc);
6    return sonuc;
7  }
8  void topla(TrieDugumu d, StringBuilder yol,
9             List<String> sonuc) {
10   if (d.kelimeSonu)
   sonuc.add(yol.toString());
11   for (int i = 0; i < 26; i++)
12     if (d.cocuk[i] != null) {
13       yol.append((char) ('a' + i));
▶        topla(d.cocuk[i], yol, sonuc);
15       yol.setLength(yol.length() - 1);
16     }
17 }
```



Özyinelemeli çağrı: `topla("ten")`.

Adım 13 / 16

DEĞİŞKENLER

`onek = "te"`

`yol = "ten"` `i = –`

ÇAĞRI YIĞINI

`topla("ten")`

`topla("te")`

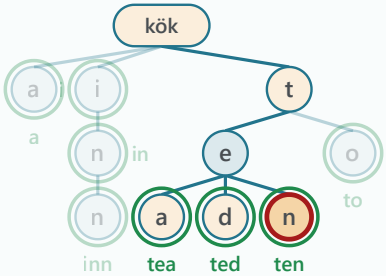
`tamamla("te")`

ÇIKTI

```
sonuc = [tea, ted]
```

Otomatik Tamamlama: tamamla("te")

```
1  List<String> tamamla(String onek) {
2    List<String> sonuc = new ArrayList<>();
3    TrieDugumu d = dugumBul(onek);
4    if (d != null)
5      topla(d, new StringBuilder(onek), sonuc);
6    return sonuc;
7  }
8  void topla(TrieDugumu d, StringBuilder yol,
9             List<String> sonuc) {
▶    if (d.kelimeSonu)
   sonuc.add(yol.toString());
11   for (int i = 0; i < 26; i++)
12     if (d.cocuk[i] != null) {
13       yol.append((char) ('a' + i));
14       topla(d.cocuk[i], yol, sonuc);
15       yol.setLength(yol.length() - 1);
16     }
17 }
```



"ten" işaretli → `sonuc` 'a eklenir. Çocuğu yok; döngü 26 indisi boş geçer.

Adım 14 / 16

DEĞİŞKENLER

`onek = "te"`

`yol = "ten"` `i = –`

ÇAĞRI YIĞINI

`topla("ten")`

`topla("te")`

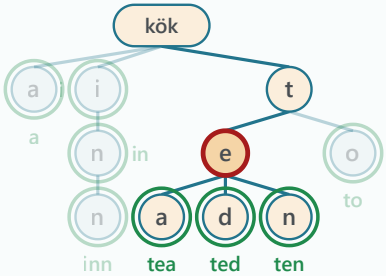
`tamamla("te")`

ÇIKTI

```
sonuc = [tea, ted, ten]
```

Otomatik Tamamlama: tamamla("te")

```
1  List<String> tamamla(String onek) {
2    List<String> sonuc = new ArrayList<>();
3    TrieDugumu d = dugumBul(onek);
4    if (d != null)
5      topla(d, new StringBuilder(onek), sonuc);
6    return sonuc;
7  }
8  void topla(TrieDugumu d, StringBuilder yol,
9             List<String> sonuc) {
10   if (d.kelimeSonu)
     sonuc.add(yol.toString());
11   for (int i = 0; i < 26; i++)
12     if (d.cocuk[i] != null) {
13       yol.append((char) ('a' + i));
14       topla(d.cocuk[i], yol, sonuc);
▶        yol.setLength(yol.length() - 1);
16     }
17 }
```



Çağrı döndü; son harf silinir (`yol` = "te") → geri izleme (backtracking).

Adım 15 / 16

DEĞİŞKENLER

`onek = "te"`

`yol = "te"` `i = 13`

ÇAĞRI YIĞINI

`topla("te")`

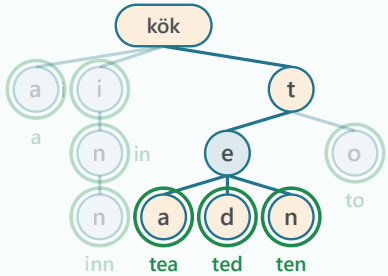
`tamamla("te")`

ÇIKTI

```
sonuc = [tea, ted, ten]
```

Otomatik Tamamlama: tamamla("te")

```
1  List<String> tamamla(String onek) {
2    List<String> sonuc = new ArrayList<>();
3    TrieDugumu d = dugumBul(onek);
4    if (d != null)
5      topla(d, new StringBuilder(onek), sonuc);
▶    return sonuc;
7  }
8  void topla(TrieDugumu d, StringBuilder yol,
9             List<String> sonuc) {
10   if (d.kelimeSonu)
   sonuc.add(yol.toString());
11   for (int i = 0; i < 26; i++)
12     if (d.cocuk[i] != null) {
13       yol.append((char) ('a' + i));
14       topla(d.cocuk[i], yol, sonuc);
15       yol.setLength(yol.length() - 1);
16     }
17 }
```



Döngü bitti; `tamamla("te")` → [tea, ted, ten]. İndis sırası (a < d < n) sayesinde sonuçlar **alfabetik** sıradadır.

Adım 16 / 16

DEĞİŞKENLER

`onek = "te"`

`yol = –` `i = –`

ÇAĞRI YIĞINI

`tamamla("te")`

ÇIKTI

```
sonuc = [tea, ted, ten]
```

enUzunOrtakOnek(): Kod İzleme

```
1  String enUzunOrtakOnek() {
▶    StringBuilder sb = new StringBuilder();
▶    TrieDugumu d = kok;
4    while (!d.kelimeSonu) {
5      int tek = -1, sayi = 0;
6      for (int i = 0; i < 26; i++)
7        if (d.cocuk[i] != null) { sayi++; tek = i; }
8      if (sayi != 1) break;  // dal ya da yaprak
9      sb.append((char) ('a' + tek));
10     d = d.cocuk[tek];
11   }
12   return sb.toString();
13 }
```



Trie: kar, kart, karton. Kökten **tek çocuklu** ve **işaretsiz** düğümler boyunca inilir.

Adım 1 / 6

DEĞİŞKENLER

`sb = ""` `d = kök` `sayi = –` `tek = –`

enUzunOrtakOnek(): Kod İzleme

```
1  String enUzunOrtakOnek() {
2    StringBuilder sb = new StringBuilder();
3    TrieDugumu d = kok;
▶    while (!d.kelimeSonu) {
▶      int tek = -1, sayi = 0;
▶      for (int i = 0; i < 26; i++)
▶        if (d.cocuk[i] != null) { sayi++; tek = i; }
▶      if (sayi != 1) break;  // dal ya da yaprak
9      sb.append((char) ('a' + tek));
10     d = d.cocuk[tek];
11   }
12   return sb.toString();
13 }
```



Kök işaretsiz; çocuk sayılır: yalnız 'k' dolu → sayi = 1, tek = 10.

Adım 2 / 6

DEĞİŞKENLER

`sb = ""` `d = kök` `sayi = 1` `tek = 10`

enUzunOrtakOnek(): Kod İzleme

```
1  String enUzunOrtakOnek() {
2    StringBuilder sb = new StringBuilder();
3    TrieDugumu d = kok;
▶    while (!d.kelimeSonu) {
▶      int tek = -1, sayi = 0;
▶      for (int i = 0; i < 26; i++)
▶        if (d.cocuk[i] != null) { sayi++; tek = i; }
▶      if (sayi != 1) break;  // dal ya da yaprak
▶      sb.append((char) ('a' + tek));
▶      d = d.cocuk[tek];
11   }
12   return sb.toString();
13 }
```



'k' eklenir, d = "k". Bu düğümde de tek çocuk ('a', tek = 0) var.

Adım 3 / 6

DEĞİŞKENLER

`sb = "k"` `d = "k"` `sayi = 1` `tek = 0`

enUzunOrtakOnek(): Kod İzleme

```
1  String enUzunOrtakOnek() {
2    StringBuilder sb = new StringBuilder();
3    TrieDugumu d = kok;
▶    while (!d.kelimeSonu) {
▶      int tek = -1, sayi = 0;
▶      for (int i = 0; i < 26; i++)
▶        if (d.cocuk[i] != null) { sayi++; tek = i; }
▶      if (sayi != 1) break;  // dal ya da yaprak
▶      sb.append((char) ('a' + tek));
▶      d = d.cocuk[tek];
11   }
12   return sb.toString();
13 }
```



'a' eklenir, d = "ka"; tek çocuk 'r' (tek = 17).

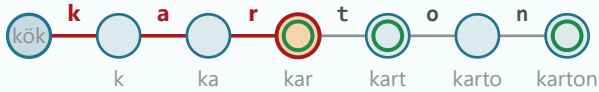
Adım 4 / 6

DEĞİŞKENLER

`sb = "ka"` `d = "ka"` `sayi = 1` `tek = 17`

enUzunOrtakOnek(): Kod İzleme

```
1  String enUzunOrtakOnek() {
2    StringBuilder sb = new StringBuilder();
3    TrieDugumu d = kok;
▶    while (!d.kelimeSonu) {
5      int tek = -1, sayi = 0;
6      for (int i = 0; i < 26; i++)
7        if (d.cocuk[i] != null) { sayi++; tek = i; }
8      if (sayi != 1) break;  // dal ya da yaprak
▶      sb.append((char) ('a' + tek));
▶      d = d.cocuk[tek];
11   }
12   return sb.toString();
13 }
```



'r' eklenir, d = "kar". "kar" bir **kelime** → döngü biter: tek çocuğu olsa da "kar" kelimesi daha uzun öneki paylaşmaz.

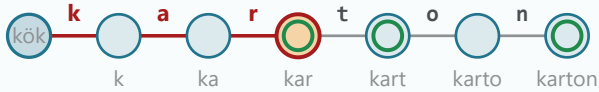
Adım 5 / 6

DEĞİŞKENLER

`sb = "kar"` `d = "kar"` `sayi = 1` `tek = 17`

enUzunOrtakOnek(): Kod İzleme

```
1  String enUzunOrtakOnek() {
2    StringBuilder sb = new StringBuilder();
3    TrieDugumu d = kok;
4    while (!d.kelimeSonu) {
5      int tek = -1, sayi = 0;
6      for (int i = 0; i < 26; i++)
7        if (d.cocuk[i] != null) { sayi++; tek = i; }
8      if (sayi != 1) break;  // dal ya da yaprak
9      sb.append((char) ('a' + tek));
10     d = d.cocuk[tek];
11   }
▶    return sb.toString();
13 }
```



"kar" döner. Trie'de yalnız kart ve karton olsaydı inişe "kart"a kadar devam edilirdi, orada kelimeSonu durdururdu: "kart".

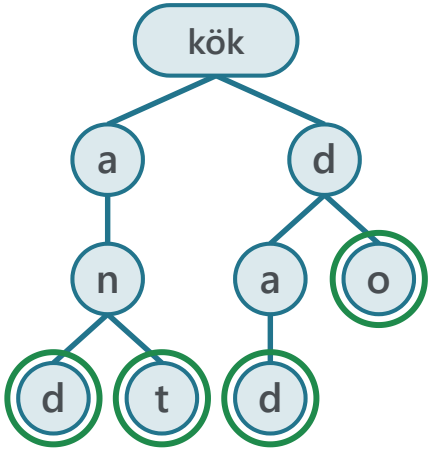
Adım 6 / 6

DEĞİŞKENLER

`sb = "kar"`

Deneyin: Trie Üzerinde İşlemler

Kelime `dot` ekle ara önek / tamamla sil örnek sıfırla



Örnek trie: and, ant, dad, do (9 düğüm, kök dahil). Bir kelime yazıp bir işlem seçin.

Yalnızca a–z küçük harfler (en fazla 10 harf, 14 kelime). Çift çember: kelime sonu · turuncu: son ziyaret edilen düğüm · kesikli kırmızı: bulunamayan dal. Enter = ekle.

SİLME

Silme (Delete): Üç Durum

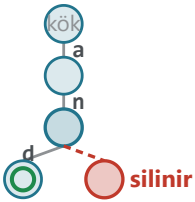
1. Kelime başka bir kelimenin öneki

Son düğümün çocuğu vardır → yalnızca `kelimeSonu = false` yapılır, düğüm silinmez.



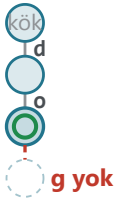
2. Kelimenin kendine ait bir kuyruğu var

Sondan başa doğru, **çocuğu kalmayan ve işaretsiz** düğümler silinir; paylaşılan önekte durulur.



3. Kelime trie'de yok

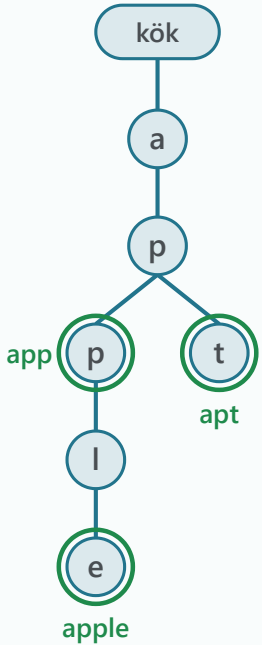
Dal bulunamaz ya da son düğüm işaretsizdir → hiçbir şey değişmez.



Soru: Trie'de "apple" ve "app" varsa, "app" silindiğinde yapı nasıl değişir?

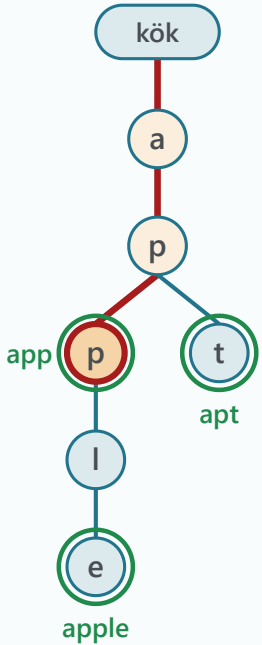
▶ **Cevabı göster**

Silme Adım Adım: app, apple, apt



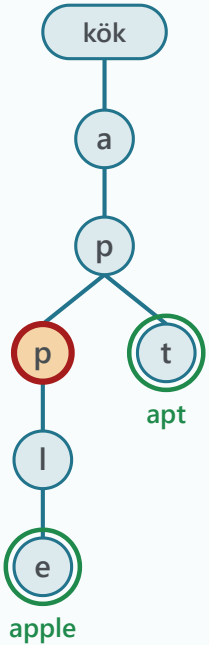
Başlangıç: app, apple, apt (kök hariç 6 düğüm). Aşağıdaki iki senaryo bu trie'den başlar.

Silme Adım Adım: app, apple, apt



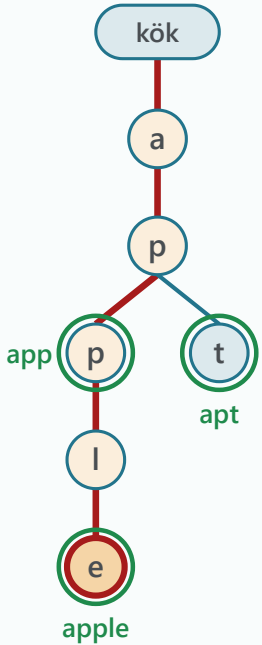
sil("app"): a → p → p yolu izlenir; "app" düğümü işaretli, yani kelime var.

Silme Adım Adım: app, apple, apt



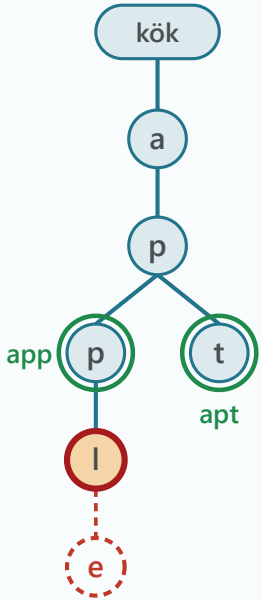
`kelimeSonu = false`. Düğümün çocuğu ('l') olduğu için **silinmez** → durum 1. "apple" hâlâ bulunur.

Silme Adım Adım: app, apple, apt



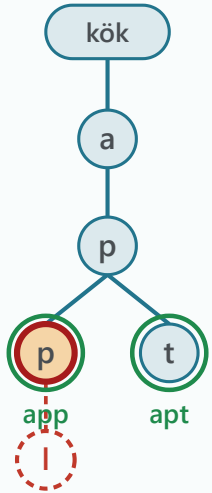
sil("apple") (yeniden ilk trie'den): yol a-p-p-l-e izlenir; "apple" bulundu.

Silme Adım Adım: app, apple, apt



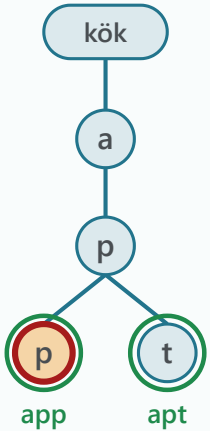
'e' düğümünün işareti kalkar; çocuğu yok → ebeveyn onu siler (`cocuk[4] = null`).

Silme Adım Adım: app, apple, apt



'l' düğümü artık çocuksuz ve işaretsiz → o da silinir.

Silme Adım Adım: app, apple, apt



"app" düğümü **işaretli** (başka bir kelime) → silme burada durur. Sonuç: app, apt; kök hariç 4 düğüm (durum 2).

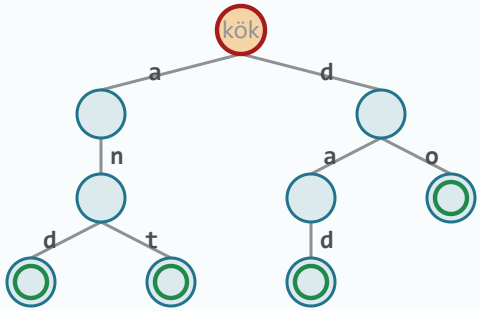
Silme: Özyinelemeli Kod

```
// true: ebeveyn d'yi silebilir
boolean sil(TrieDugumu d, String k, int j) {
  if (j == k.length()) {        // kelime bitti
    if (!d.kelimeSonu) return false;
    d.kelimeSonu = false;
    return bosMu(d);            // çocuksuz mu?
  }
  int i = k.charAt(j) - 'a';
  TrieDugumu c = d.cocuk[i];
  if (c == null) return false;  // kelime yok
  if (sil(c, k, j + 1)) {
    d.cocuk[i] = null;          // çocuğu sil
    return !d.kelimeSonu && bosMu(d);
  }
  return false;
}
```

- Çağrı: `sil(kok, kelime, 0)`; kökün dönüş değeri yok sayılır.
- Önce kelimenin sonuna **inilir**, silme kararları **dönüşte** (aşağıdan yukarı) verilir.
- Düğüm, ancak **çocuğu yoksa** ve **işaretsizse** silinir.
- `bosMu(d)`: 26 çocuğun hepsi null mı? (O(M)).
- Zaman: O(L · M) dizi ile; çocuk sayacı tutulursa O(L).

sil(kok, "dad", 0)

```
1  // true: ebeveyn d'yi silebilir
2  boolean sil(TrieDugumu d, String k,
3              int j) {
4    if (j == k.length()) {  // bitti
5      if (!d.kelimeSonu) return false;
6      d.kelimeSonu = false;
7      return bosMu(d);   // çocuksuz?
8    }
▶    int i = k.charAt(j) - 'a';
▶    TrieDugumu c = d.cocuk[i];
▶    if (c == null) return false;
▶    if (sil(c, k, j + 1)) {
13     d.cocuk[i] = null;  // çocuğu sil
14     return !d.kelimeSonu && bosMu(d);
15   }
16   return false;
17 }
```



sil(kök, "dad", 0): kelimenin ilk karakteri 'd' → i = 3, c = kökün 'd' çocuğu. Silme kararları dönüşte verilecek; önce aşağı inilir.

Adım 1 / 10

DEĞİŞKENLER

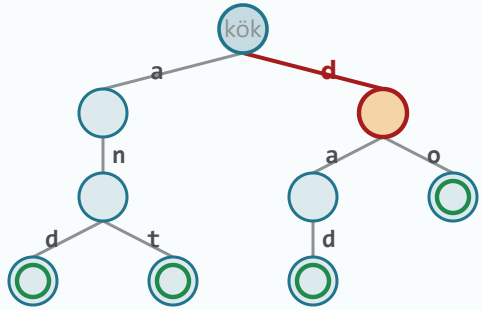
`d = kök` `j = 0` `i = 3` `c = "d"`

ÇAĞRI YIĞINI

`sil(kök, 0)`

sil(kok, "dad", 0)

```
1  // true: ebeveyn d'yi silebilir
2  boolean sil(TrieDugumu d, String k,
3              int j) {
4    if (j == k.length()) {  // bitti
5      if (!d.kelimeSonu) return false;
6      d.kelimeSonu = false;
7      return bosMu(d);   // çocuksuz?
8    }
▶    int i = k.charAt(j) - 'a';
▶    TrieDugumu c = d.cocuk[i];
▶    if (c == null) return false;
▶    if (sil(c, k, j + 1)) {
13     d.cocuk[i] = null;  // çocuğu sil
14     return !d.kelimeSonu && bosMu(d);
15   }
16   return false;
17 }
```



sil(d, "dad", 1): 'a' → i = 0, c = "da" düğümü. Özyinelemeli çağrı bir düzey daha iner.

Adım 2 / 10

DEĞİŞKENLER

`d = d` `j = 1` `i = 0` `c = "da"`

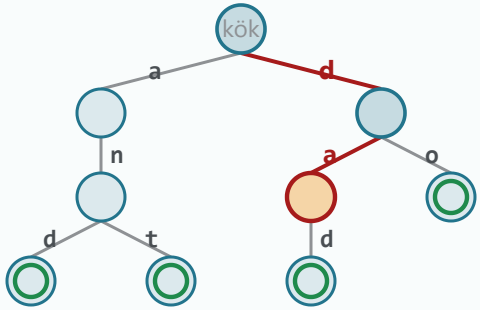
ÇAĞRI YIĞINI

`sil(d, 1)`

`sil(kök, 0)`

sil(kok, "dad", 0)

```
1  // true: ebeveyn d'yi silebilir
2  boolean sil(TrieDugumu d, String k,
3              int j) {
4    if (j == k.length()) {  // bitti
5      if (!d.kelimeSonu) return false;
6      d.kelimeSonu = false;
7      return bosMu(d);   // çocuksuz?
8    }
▶    int i = k.charAt(j) - 'a';
▶    TrieDugumu c = d.cocuk[i];
▶    if (c == null) return false;
▶    if (sil(c, k, j + 1)) {
13     d.cocuk[i] = null;  // çocuğu sil
14     return !d.kelimeSonu && bosMu(d);
15   }
16   return false;
17 }
```



sil(da, "dad", 2): 'd' → i = 3, c = "dad" düğümü.

Adım 3 / 10

DEĞİŞKENLER

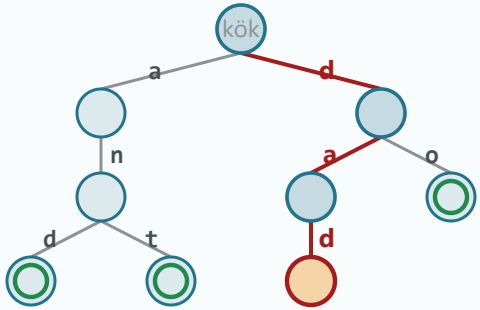
`d = da` `j = 2` `i = 3` `c = "dad"`

ÇAĞRI YIĞINI

- `sil(da, 2)`
- `sil(d, 1)`
- `sil(kök, 0)`

sil(kok, "dad", 0)

```
1  // true: ebeveyn d'yi silebilir
2  boolean sil(TrieDugumu d, String k,
3              int j) {
▶    if (j == k.length()) {  // bitti
▶      if (!d.kelimeSonu) return false;
▶      d.kelimeSonu = false;
7      return bosMu(d);   // çocuksuz?
8    }
9    int i = k.charAt(j) - 'a';
10   TrieDugumu c = d.cocuk[i];
11   if (c == null) return false;
12   if (sil(c, k, j + 1)) {
13     d.cocuk[i] = null;  // çocuğu sil
14     return !d.kelimeSonu && bosMu(d);
15   }
16   return false;
17 }
```



j = 3 = k.length(): kelime bitti. "dad" işaretli olduğu için `kelimeSonu = false` yapılır.

Adım 4 / 10

DEĞİŞKENLER

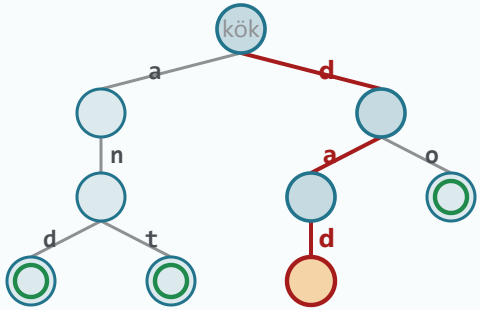
`d = dad` `j = 3` `i = –` `c = –`

ÇAĞRI YIĞINI

- `sil(dad, 3)`
- `sil(da, 2)`
- `sil(d, 1)`
- `sil(kök, 0)`

sil(kok, "dad", 0)

```
1  // true: ebeveyn d'yi silebilir
2  boolean sil(TrieDugumu d, String k,
3              int j) {
4    if (j == k.length()) {  // bitti
5      if (!d.kelimeSonu) return false;
6      d.kelimeSonu = false;
▶      return bosMu(d);   // çocuksuz?
8    }
9    int i = k.charAt(j) - 'a';
10   TrieDugumu c = d.cocuk[i];
11   if (c == null) return false;
12   if (sil(c, k, j + 1)) {
13     d.cocuk[i] = null;  // çocuğu sil
14     return !d.kelimeSonu && bosMu(d);
15   }
16   return false;
17 }
```



bosMu(dad) = true: düğümün hiç çocuğu yok → **true** döner; ebeveyn bu düğümü silebilir.

Adım 5 / 10

DEĞİŞKENLER

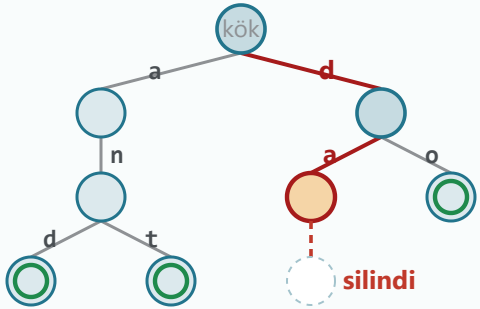
`d = dad` `j = 3` `i = –` `c = –`

ÇAĞRI YIĞINI

- `sil(dad, 3)`
- `sil(da, 2)`
- `sil(d, 1)`
- `sil(kök, 0)`

sil(kok, "dad", 0)

```
1  // true: ebeveyn d'yi silebilir
2  boolean sil(TrieDugumu d, String k,
3              int j) {
4    if (j == k.length()) {  // bitti
5      if (!d.kelimeSonu) return false;
6      d.kelimeSonu = false;
7      return bosMu(d);   // çocuksuz?
8    }
9    int i = k.charAt(j) - 'a';
10   TrieDugumu c = d.cocuk[i];
11   if (c == null) return false;
12   if (sil(c, k, j + 1)) {
▶      d.cocuk[i] = null;  // çocuğu sil
14     return !d.kelimeSonu && bosMu(d);
15   }
16   return false;
17 }
```



sil(da) true döndü → `d.cocuk[3] = null` : "dad" düğümü koparıldı.

Adım 6 / 10

DEĞİŞKENLER

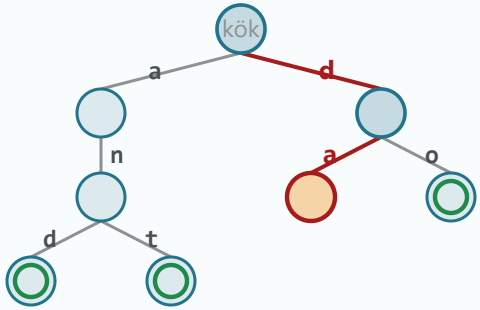
`d = da` `j = 2` `i = 3` `c = "dad"`

ÇAĞRI YIĞINI

- **sil(da, 2)**
- sil(d, 1)
- sil(kök, 0)

sil(kok, "dad", 0)

```
1  // true: ebeveyn d'yi silebilir
2  boolean sil(TrieDugumu d, String k,
3              int j) {
4    if (j == k.length()) {  // bitti
5      if (!d.kelimeSonu) return false;
6      d.kelimeSonu = false;
7      return bosMu(d);   // çocuksuz?
8    }
9    int i = k.charAt(j) - 'a';
10   TrieDugumu c = d.cocuk[i];
11   if (c == null) return false;
12   if (sil(c, k, j + 1)) {
13     d.cocuk[i] = null;  // çocuğu sil
▶      return !d.kelimeSonu && bosMu(d);
15   }
16   return false;
17 }
```



"da" işaretsiz ve artık çocuksuz → **true**: bu düğüm de silinebilir.

Adım 7 / 10

DEĞİŞKENLER

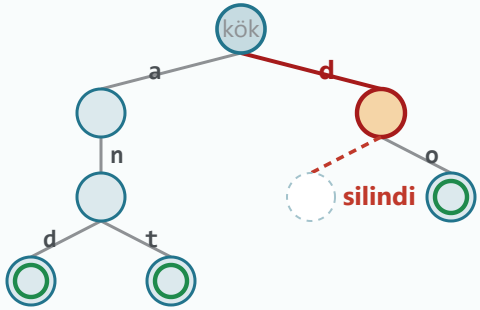
`d = da` `j = 2` `i = 3` `c = "dad"`

ÇAĞRI YIĞINI

- `sil(da, 2)`
- `sil(d, 1)`
- `sil(kök, 0)`

sil(kok, "dad", 0)

```
1  // true: ebeveyn d'yi silebilir
2  boolean sil(TrieDugumu d, String k,
3              int j) {
4    if (j == k.length()) {  // bitti
5      if (!d.kelimeSonu) return false;
6      d.kelimeSonu = false;
7      return bosMu(d);   // çocuksuz?
8    }
9    int i = k.charAt(j) - 'a';
10   TrieDugumu c = d.cocuk[i];
11   if (c == null) return false;
12   if (sil(c, k, j + 1)) {
▶      d.cocuk[i] = null;  // çocuğu sil
14     return !d.kelimeSonu && bosMu(d);
15   }
16   return false;
17 }
```



sil(d) true döndü → `d.cocuk[0] = null` : "da" düğümü koparıldı.

Adım 8 / 10

DEĞİŞKENLER

`d = d` `j = 1` `i = 0` `c = "da"`

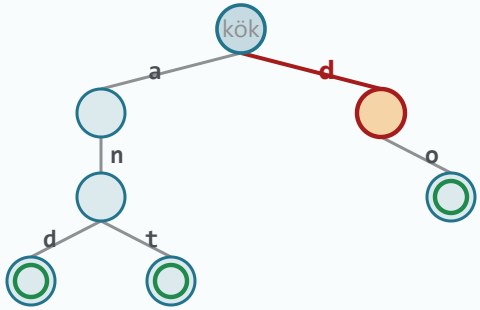
ÇAĞRI YIĞINI

`sil(d, 1)`

`sil(kök, 0)`

sil(kok, "dad", 0)

```
1  // true: ebeveyn d'yi silebilir
2  boolean sil(TrieDugumu d, String k,
3              int j) {
4    if (j == k.length()) {  // bitti
5      if (!d.kelimeSonu) return false;
6      d.kelimeSonu = false;
7      return bosMu(d);   // çocuksuz?
8    }
9    int i = k.charAt(j) - 'a';
10   TrieDugumu c = d.cocuk[i];
11   if (c == null) return false;
12   if (sil(c, k, j + 1)) {
13     d.cocuk[i] = null;  // çocuğu sil
▶      return !d.kelimeSonu && bosMu(d);
15   }
16   return false;
17 }
```



"d" işaretsiz ama "do" için 'o' çocuğu var → bosMu false → **false**. Silme burada durur.

Adım 9 / 10

DEĞİŞKENLER

`d = d` `j = 1` `i = 0` `c = "da"`

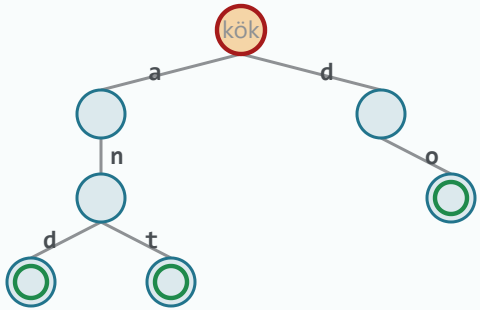
ÇAĞRI YIĞINI

`sil(d, 1)`

`sil(kök, 0)`

sil(kok, "dad", 0)

```
1  // true: ebeveyn d'yi silebilir
2  boolean sil(TrieDugumu d, String k,
3              int j) {
4    if (j == k.length()) {  // bitti
5      if (!d.kelimeSonu) return false;
6      d.kelimeSonu = false;
7      return bosMu(d);   // çocuksuz?
8    }
9    int i = k.charAt(j) - 'a';
10   TrieDugumu c = d.cocuk[i];
11   if (c == null) return false;
12   if (sil(c, k, j + 1)) {
13     d.cocuk[i] = null;  // çocuğu sil
14     return !d.kelimeSonu && bosMu(d);
15   }
▶    return false;
17 }
```



Kökte sil(d) false döndü → kök bir şey silmez, false döner (dönüş değeri yok sayılır). Kalan kelimeler: and, ant, do.

Adım 10 / 10

DEĞİŞKENLER

`d = kök` `j = 0` `i = 3` `c = "d"`

ÇAĞRI YIĞINI

`sil(kök, 0)`

ANALİZ

Karmaşıklık

İşlem	Zaman	Açıklama
ekle(kelime)	O(L)	Her karakter için bir adım; en fazla L yeni düğüm
ara(kelime)	O(L)	Dal yoksa daha erken biter
onekVarMi(onek)	O(P)	P: önek uzunluğu
tamamla(onek)	O(P + S · M)	S: önek düğümünün alt ağacındaki düğüm sayısı
sil(kelime)	O(L · M)	`bosMu` 26 hücreyi tarar; çocuk sayacıyla O(L)

Bellek: O(N · M)

N: düğüm sayısı, M: alfabe büyüklüğü (26). N ≤ (tüm kelimelerin toplam uzunluğu) + 1.

n'den bağımsız

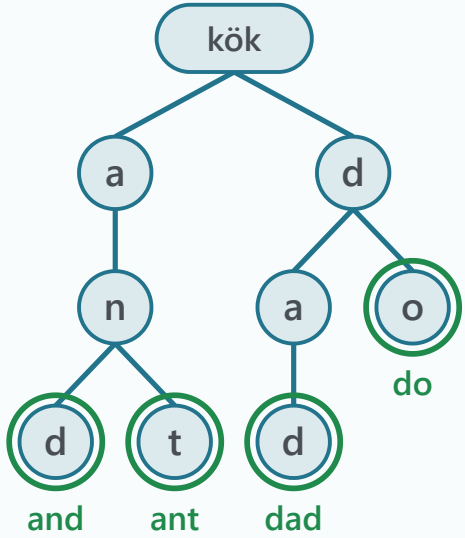
Süreler saklanan kelime **sayısına** bağlı değildir; BST'deki log n çarpanı yoktur.

ANALİZ

Bellek: Sayısal Örnek

Trie: **and, ant, dad, do**

Toplam karakter	3 + 3 + 3 + 2 = 11
Düğüm sayısı N	1 (kök) + 8 = 9
Çocuk referansı	9 × 26 = 234
Dolu referans	8 (kenar sayısı)
Boş (null) referans	226 (≈ %97)



Ortak önekler 11 karakteri 8 düğüme indirir, ama 26 öğeli diziler belleğin büyük kısmını **null** ile doldurur. Alfabe büyüdükçe (Unicode: 65.536+) sorun büyür.

ANALİZ

Avantajlar ve Dezavantajlar

Avantajlar

+ Hızlı **önek** (prefix) tabanlı arama: otomatik tamamlama.
+ Kelime arama **O(L)** zaman (L: kelime uzunluğu).
+ Çakışan önekler tek kez saklanır: **bellek paylaşımı**.
+ Gezinme kelimeleri **sözlük sırasında** verir; hash çakışması yoktur.

Dezavantajlar

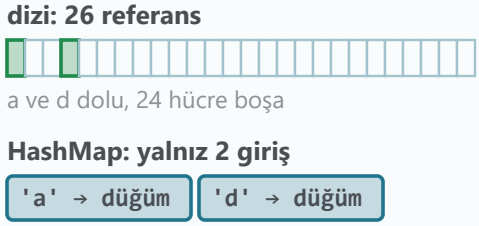
– **Yüksek bellek** tüketimi, özellikle büyük alfabelerde.
– Alan karmaşıklığı **O(N · M)** (N: düğüm sayısı, M: alfabe büyüklüğü).
– Ortak öneki az olan kümelerde (ör. rastgele anahtarlar) kazanç yoktur.
– Çok sayıda küçük nesne: önbellek (cache) dostu değildir.

ANALİZ

Çocukları Saklamanın Yolları

Gösterim	Çocuğa erişim	Düğüm başına bellek	Not
`TrieDugumu[26]` dizi	O(1)	M referans	Basit, hızlı; seyrek düğümlerde israf
`HashMap<Character, TrieDugumu>`	O(1) ort.	çocuk sayısı kadar	Her alfabe (Türkçe, Unicode) çalışır; sıra korunmaz
`TreeMap` / sıralı liste	O(log M)	çocuk sayısı kadar	Alfabetik gezinme korunur
Üçlü arama trie'si (TST)	O(log M) ort.	3 referans	Her düğümde küçük / eşit / büyük bağlantısı

Trie'nin **mantığı** değişmez; yalnızca `dugum.cocuk[i]` yerine `dugum.cocuk.get(ch)` gibi bir çağrı gelir. Seçim: alfabe büyüklüğü, bellek ve sıralı çıktı ihtiyacına göre yapılır.



ANALİZ

Trie mi, Hash Tablosu mu, BST mi?

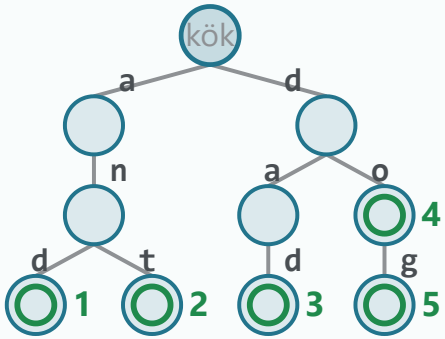
İşlem (n kelime, uzunluk L)	Trie	Hash tablosu	Dengeli BST
Tam kelime arama	O(L)	O(L) ortalama	O(L · log n)
Önek sorgusu	O(P)	desteklenmez	O(P · log n)
Önekle başlayan k kelime	O(P + S · M)	tüm tabloyu tara	O(P · log n + k · L)
Sıralı listeleme	DFS, O(N · M)	sıralama gerekir	kök ortada, O(n)
En kötü durum	O(L)	O(n · L) (çakışma)	O(L · log n)
Bellek	yüksek	düşük	düşük

Hash tablosu da kelimenin hash değerini hesaplamak için L karakterin tümünü okur; bu yüzden tam aramada ikisi aynı mertebededir. Trie'nin farkı **önek** ve **sıra** sorgularındadır. S: önek düğümünün alt ağacındaki düğüm sayısı.

ANALİZ

Trie ile Sözlük Sırası: Dizgileri Sıralamak

- Girdi sırası: dog, ant, do, and, dad. Hepsi trie'ye eklenir.
- Kökten **önce-kök** (preorder) gezilir; çocuklar 'a' → 'z' sırasıyla ziyaret edilir, işaretli düğümde kelime yazılır.
- Çıktı: and, ant, dad, do, dog. Düğüm, çocuklarından önce yazıldığı için **önek, uzantısından önce** gelir (do < dog).
- Toplam karakter sayısı N ve sabit alfabe için ekleme O(N), gezinme O(N·M) (M = 26 çocuk denetimi): karşılaştırmalı sıralamanın O(n log n) karşılaştırmasına ve her karşılaştırmanın O(L) maliyetine gerek kalmaz.
- Aynı fikir “burst sort” ve MSD basamak sıralamasının dizgi sürümlerinin temelidir.



Yeşil: önce-kök gezinmede yazılma sırası.

Kullanım Alanları

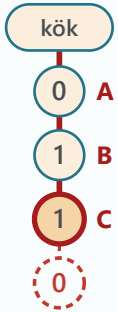
Otomatik tamamlama
Arama kutusu, IDE: yazılan önek düğümüne inilir, alt ağaçtaki kelimeler (sık kullanılanlar önce) önerilir.

Yazım denetimi
Sözlük trie'de tutulur; kelime yoksa trie üzerinde küçük düzenlemelerle (ekle / sil / değiştir) yakın kelimeler aranır.

IP yönlendirme
İkili trie (alfabe {0, 1}) ile **en uzun önek eşleşmesi**: 0110… adresi, 0* / 01* / 011* yollarından en uzununa, 011*'e gider.

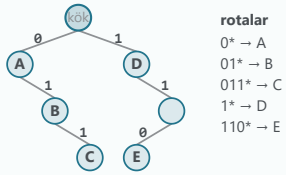
Kelime oyunları ve T9
Boggle / kelime bulmaca: yol bir önek değilse arama hemen kesilir (budama). Tuş dizisinden kelime önerme.

En uzun önek eşleşmesi: rotalar 0* → A, 01* → B, 011* → C. Adres 0110… için kökten 0 → 1 → 1 inilir; '0' dalı yoktur. Yol boyunca görülen **son** rota C'dir → paket C'ye gider.



enUzunEslesme(): İkili Trie ile Yönlendirme

```
1  String enUzunEslesme(String adres) {
▶    IkiliDugum d = kok;
▶    String enIyi = null;
4    for (int k = 0; k < adres.length(); k++) {
5      if (d.rota != null) enIyi = d.rota;  // daha uzun önek
6      d = d.cocuk[adres.charAt(k) - '0'];
7      if (d == null) return enIyi;
8    }
9    return d.rota != null ? d.rota : enIyi;
10 }
```



İkili trie (alfabe {0, 1}); bazı düğümler bir rota taşır. adres = "0110": d = kök, enIyi = null.

Adım 1 / 9

DEĞİŞKENLER

`adres = "0110"`

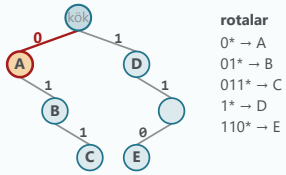
`k = –` `d = kök`

`enIyi = null`

ÇIKTI

enUzunEslesme(): İkili Trie ile Yönlendirme

```
1  String enUzunEslesme(String adres) {
2    IkiliDugum d = kok;
3    String enIyi = null;
▶    for (int k = 0; k < adres.length(); k++) {
▶      if (d.rota != null) enIyi = d.rota;  // daha uzun önek
▶      d = d.cocuk[adres.charAt(k) - '0'];
▶      if (d == null) return enIyi;
8    }
9    return d.rota != null ? d.rota : enIyi;
10 }
```



k = 0: kökte rota yok. Bit '0' → d = "0".

Adım 2 / 9

DEĞİŞKENLER

`adres = "0110"`

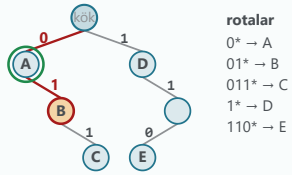
`k = 0` `d = "0"`

`enIyi = null`

ÇIKTI

enUzunEslesme(): İkili Trie ile Yönlendirme

```
1  String enUzunEslesme(String adres) {
2    IkiliDugum d = kok;
3    String enIyi = null;
▶    for (int k = 0; k < adres.length(); k++) {
▶      if (d.rota != null) enIyi = d.rota;  // daha uzun önek
▶      d = d.cocuk[adres.charAt(k) - '0'];
▶      if (d == null) return enIyi;
8    }
9    return d.rota != null ? d.rota : enIyi;
10 }
```



k = 1: "0" düğümünde A var → enIyi = A. Bit '1' → d = "01".

Adım 3 / 9

DEĞİŞKENLER

`adres = "0110"`

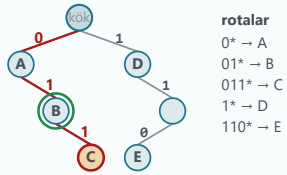
`k = 1` `d = "01"`

`enIyi = A`

ÇIKTI

enUzunEslesme(): İkili Trie ile Yönlendirme

```
1  String enUzunEslesme(String adres) {
2    IkiliDugum d = kok;
3    String enIyi = null;
▶    for (int k = 0; k < adres.length(); k++) {
▶      if (d.rota != null) enIyi = d.rota;  // daha uzun önek
▶      d = d.cocuk[adres.charAt(k) - '0'];
▶      if (d == null) return enIyi;
8    }
9    return d.rota != null ? d.rota : enIyi;
10 }
```



k = 2: "01"de B → enIyi = B (daha uzun önek kazanır). d = "011".

Adım 4 / 9

DEĞİŞKENLER

`adres = "0110"`

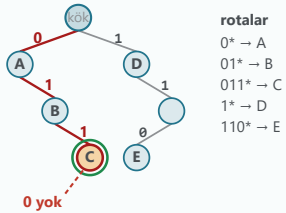
`k = 2` `d = "011"`

`enIyi = B`

ÇIKTI

enUzunEslesme(): İkili Trie ile Yönlendirme

```
1  String enUzunEslesme(String adres) {
2    IkiliDugum d = kok;
3    String enIyi = null;
▶    for (int k = 0; k < adres.length(); k++) {
▶      if (d.rota != null) enIyi = d.rota;  // daha uzun önek
▶      d = d.cocuk[adres.charAt(k) - '0'];
▶      if (d == null) return enIyi;
8    }
9    return d.rota != null ? d.rota : enIyi;
10 }
```



k = 3: "011"de C → enIyi = C. Bit '0' dalı yok → d = null, **C** döner.

Adım 5 / 9

DEĞİŞKENLER

`adres = "0110"`

`k = 3` `d = null`

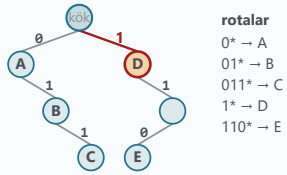
`enIyi = C`

ÇIKTI

```
C
```

enUzunEslesme(): İkili Trie ile Yönlendirme

```
1  String enUzunEslesme(String adres) {
▶    IkiliDugum d = kok;
▶    String enIyi = null;
▶    for (int k = 0; k < adres.length(); k++) {
▶      if (d.rota != null) enIyi = d.rota;  // daha uzun önek
▶      d = d.cocuk[adres.charAt(k) - '0'];
▶      if (d == null) return enIyi;
8    }
9    return d.rota != null ? d.rota : enIyi;
10 }
```



Yeni adres "1001": k = 0, kökte rota yok; d = "1".

Adım 6 / 9

DEĞİŞKENLER

`adres = "1001"`

`k = 0` `d = "1"`

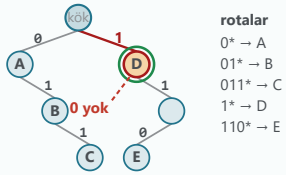
`enIyi = null`

ÇIKTI

```
C
```

enUzunEslesme(): İkili Trie ile Yönlendirme

```
1  String enUzunEslesme(String adres) {
2    IkiliDugum d = kok;
3    String enIyi = null;
▶    for (int k = 0; k < adres.length(); k++) {
▶      if (d.rota != null) enIyi = d.rota;  // daha uzun önek
▶      d = d.cocuk[adres.charAt(k) - '0'];
▶      if (d == null) return enIyi;
8    }
9    return d.rota != null ? d.rota : enIyi;
10 }
```



k = 1: "1"de D → enIyi = D. '0' dalı yok → **D** döner. Özel rota yoksa genel olan kullanılır.

DEĞİŞKENLER

`adres = "1001"`

`k = 1` `d = null`

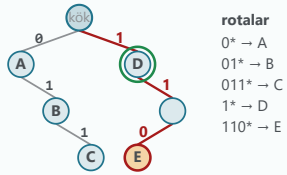
`enIyi = D`

ÇIKTI

```
C
D
```

enUzunEslesme(): İkili Trie ile Yönlendirme

```
1  String enUzunEslesme(String adres) {
▶    IkiliDugum d = kok;
▶    String enIyi = null;
▶    for (int k = 0; k < adres.length(); k++) {
▶      if (d.rota != null) enIyi = d.rota;  // daha uzun önek
▶      d = d.cocuk[adres.charAt(k) - '0'];
▶      if (d == null) return enIyi;
8    }
9    return d.rota != null ? d.rota : enIyi;
10 }
```



"1101": iniş 1 → 11 → 110. "11" ara düğümdür (rotası yok), enIyi D olarak kalır.

Adım 8 / 9

DEĞİŞKENLER

`adres = "1101"`

`k = 2` `d = "110"`

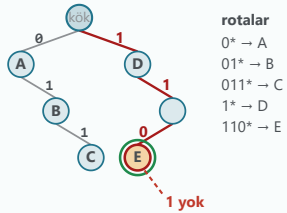
`enIyi = D`

ÇIKTI

```
C
D
```

enUzunEslesme(): İkili Trie ile Yönlendirme

```
1  String enUzunEslesme(String adres) {
2    IkiliDugum d = kok;
3    String enIyi = null;
▶    for (int k = 0; k < adres.length(); k++) {
▶      if (d.rota != null) enIyi = d.rota;  // daha uzun önek
▶      d = d.cocuk[adres.charAt(k) - '0'];
▶      if (d == null) return enIyi;
8    }
9    return d.rota != null ? d.rota : enIyi;
10 }
```



k = 3: "110"da E → enIyi = E; '1' dalı yok → **E**. Adım sayısı adres uzunluğuyla sınırlı.

Adım 9 / 9

DEĞİŞKENLER

`adres = "1101"`

`k = 3` `d = null`

`enIyi = E`

ÇIKTI

```
C
D
E
```

İkili Trie ile En Büyük XOR Çifti

- Soru: [3, 10, 5, 25, 2, 8] içinde a ⊕ b'yi en büyük yapan çift? Kaba kuvvet n^2 çift dener.
- Sayılar, **en anlamlı bitten** başlayarak 5 bitlik ikili trie'ye eklenir.
- Her x için kökten inilirken her bitte **ters** bit dalı seçilir (varsa): o bit sonuçta 1 olur. Üst bitteki 1, alttaki bütün bitlerin toplamından değerlidir; bu yüzden açgözlü seçim doğrudur.
- x = 5 = 00101 için istenen 11010: 1 ✓ (25), 1 ✓, 0 ✓, 1 yok → 0, 0 yok → 1: varılan 11001 = 25, 5 ⊕ 25 = **28**.

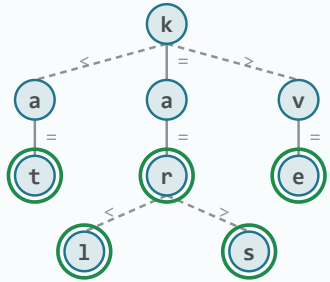
sayı	ikili (5 bit)
3	`00011`
10	`01010`
5	`00101`
25	`11001`
2	`00010`
8	`01000`
5 ⊕ 25	`11100 = 28`

Yöntem	zaman	n = 10^5, B = 30
Tüm çiftler	$O(n^2)$	≈ $5 \cdot 10^9$ işlem
İkili trie	$O(n \cdot B)$	≈ $6 \cdot 10^6$ adım

Üçlü Arama Trie'si (Ternary Search Trie)

- Her düğüm **bir karakter** ve üç bağlantı taşır: `sol` (küçük), `orta` (eşit, sonraki karaktere geç), `sag` (büyük).
- Bir düğümün sol/sağ zinciri, trie'deki "kardeş" çocuklar için küçük bir ikili arama ağacıdır.
- Kelime sonu düğümde işaretlenir (halka). Ekleme sırası şekli değiştirir: önce eklenen kelime "omurga" olur.

kar, kal, at, ve, kas	düğüm	bağlantı
26 öğeli dizili trie	10	10 × 26 = 260
TST	9	9 × 3 = 27

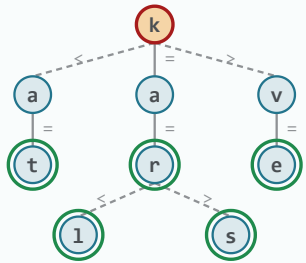


Düz çizgi: orta (=); kesikli: sol (<) ve sağ (>).

Arama L karakter için L orta adım + her düzeyde kardeşler arasında ikili arama: ortalama O(L + log n). Alfabe büyükken (Unicode) dizi yerine TST bellek ve hız dengesini korur (Bentley ve Sedgewick, 1997).

Üçlü Arama Trie'sinde ara("kas") ve ara("al")

```
1  boolean ara(String s) {
▶    TstDugum d = kok;
▶    int i = 0;
4    while (d != null) {
5      char ch = s.charAt(i);
6      if (ch < d.c) d = d.sol;
7      else if (ch > d.c) d = d.sag;
8      else if (i == s.length() - 1) return
   d.son;
9      else { d = d.orta; i++; }
10   }
11   return false;
12 }
```



TST: kar, kal, at, ve, kas. ara("kas"): d = kök ('k'), i = 0.

Adım 1 / 8

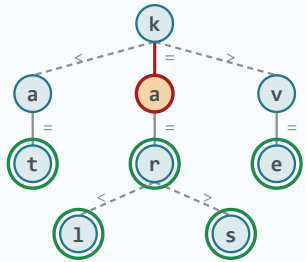
DEĞİŞKENLER

`s = "kas"` `i = 0` `ch = –` `d = 'k'`

ÇIKTI

Üçlü Arama Trie'sinde ara("kas") ve ara("al")

```
1  boolean ara(String s) {
2    TstDugum d = kok;
3    int i = 0;
▶    while (d != null) {
▶      char ch = s.charAt(i);
▶      if (ch < d.c) d = d.sol;
▶      else if (ch > d.c) d = d.sag;
▶      else if (i == s.length() - 1) return
   d.son;
▶      else { d = d.orta; i++; }
10   }
11   return false;
12 }
```



ch = 'k' = d.c ve son karakter değil → orta bağlantı, i = 1. d = 'a'.

Adım 2 / 8

DEĞİŞKENLER

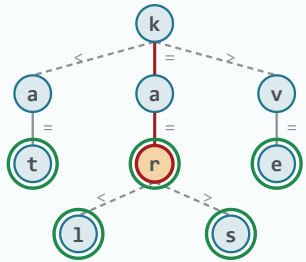
`s = "kas"` `i = 1`

`ch = 'k'` `d = 'a'`

ÇIKTI

Üçlü Arama Trie'sinde ara("kas") ve ara("al")

```
1  boolean ara(String s) {
2    TstDugum d = kok;
3    int i = 0;
▶    while (d != null) {
▶      char ch = s.charAt(i);
▶      if (ch < d.c) d = d.sol;
▶      else if (ch > d.c) d = d.sag;
▶      else if (i == s.length() - 1) return
   d.son;
▶      else { d = d.orta; i++; }
10   }
11   return false;
12 }
```



ch = 'a' eşit → orta, i = 2. d = 'r'.

Adım 3 / 8

DEĞİŞKENLER

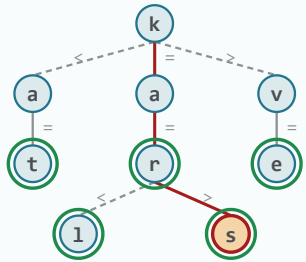
`s = "kas"` `i = 2`

`ch = 'a'` `d = 'r'`

ÇIKTI

Üçlü Arama Trie'sinde ara("kas") ve ara("al")

```
1  boolean ara(String s) {
2    TstDugum d = kok;
3    int i = 0;
▶    while (d != null) {
▶      char ch = s.charAt(i);
▶      if (ch < d.c) d = d.sol;
▶      else if (ch > d.c) d = d.sag;
8      else if (i == s.length() - 1) return
   d.son;
9      else { d = d.orta; i++; }
10   }
11   return false;
12 }
```



ch = 's' > 'r' → **sağ** bağlantı; i değişmez (aynı karakter, kardeş aranıyor). d = 's'.

Adım 4 / 8

DEĞİŞKENLER

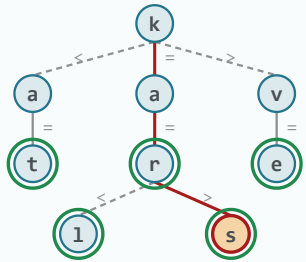
`s = "kas"` `i = 2`

`ch = 's'` `d = 's'`

ÇIKTI

Üçlü Arama Trie'sinde ara("kas") ve ara("al")

```
1  boolean ara(String s) {
2    TstDugum d = kok;
3    int i = 0;
▶    while (d != null) {
▶      char ch = s.charAt(i);
▶      if (ch < d.c) d = d.sol;
▶      else if (ch > d.c) d = d.sag;
▶      else if (i == s.length() - 1) return
   d.son;
9      else { d = d.orta; i++; }
10   }
11   return false;
12 }
```



ch = 's' eşit ve i = 2 son indis → d.son döner: **true**. 3 karakter için 4 düğüm ziyaret edildi.

Adım 5 / 8

DEĞİŞKENLER

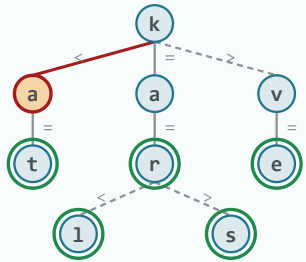
`s = "kas"` `i = 2` `ch = 's'` `d = 's'`

ÇIKTI

```
true
```

Üçlü Arama Trie'sinde ara("kas") ve ara("al")

```
1  boolean ara(String s) {
▶    TstDugum d = kok;
▶    int i = 0;
▶    while (d != null) {
▶      char ch = s.charAt(i);
▶      if (ch < d.c) d = d.sol;
7      else if (ch > d.c) d = d.sag;
8      else if (i == s.length() - 1) return
   d.son;
9      else { d = d.orta; i++; }
10   }
11   return false;
12 }
```



ara("al"): ch = 'a' < 'k' → **sol**, d = 'a' (at kelimesinin dalı).

Adım 6 / 8

DEĞİŞKENLER

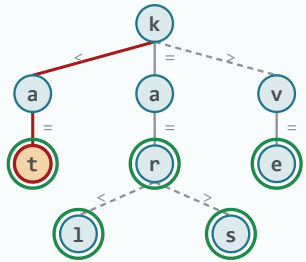
`s = "al"` `i = 0` `ch = 'a'` `d = 'a'`

ÇIKTI

```
true
```

Üçlü Arama Trie'sinde ara("kas") ve ara("al")

```
1  boolean ara(String s) {
2    TstDugum d = kok;
3    int i = 0;
▶    while (d != null) {
▶      char ch = s.charAt(i);
▶      if (ch < d.c) d = d.sol;
▶      else if (ch > d.c) d = d.sag;
▶      else if (i == s.length() - 1) return
   d.son;
▶      else { d = d.orta; i++; }
10   }
11   return false;
12 }
```



'a' eşit, son karakter değil → orta, i = 1, d = 't'.

Adım 7 / 8

DEĞİŞKENLER

`s = "al"` `i = 1`

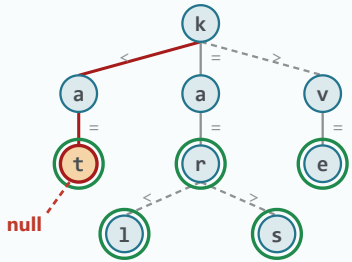
`ch = 'a'` `d = 't'`

ÇIKTI

```
true
```

Üçlü Arama Trie'sinde ara("kas") ve ara("al")

```
1  boolean ara(String s) {
2    TstDugum d = kok;
3    int i = 0;
▶    while (d != null) {
▶      char ch = s.charAt(i);
▶      if (ch < d.c) d = d.sol;
7      else if (ch > d.c) d = d.sag;
8      else if (i == s.length() - 1) return
     d.son;
9      else { d = d.orta; i++; }
▶    }
▶    return false;
12 }
```



ch = 'l' < 't' → sol; 't'nin sol bağlantısı null → döngü biter, **false**.

Adım 8 / 8

DEĞİŞKENLER

`s = "al"` `i = 1`

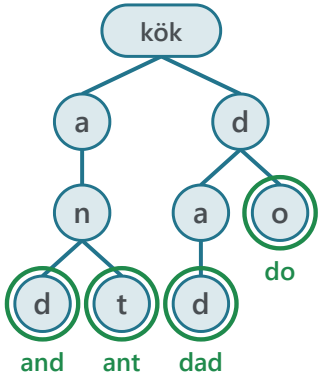
`ch = 'l'` `d = null`

ÇIKTI

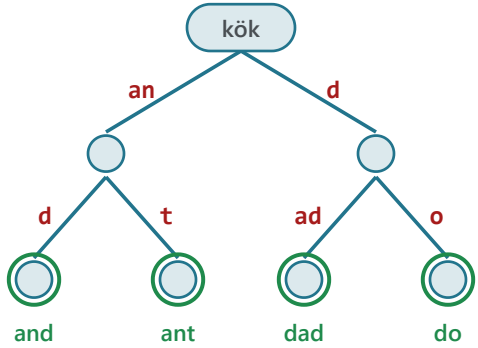
```
true
false
```

Sıkıştırılmış Trie (Radix / Patricia)

Standart trie: 8 düğüm



Sıkıştırılmış trie: 6 düğüm



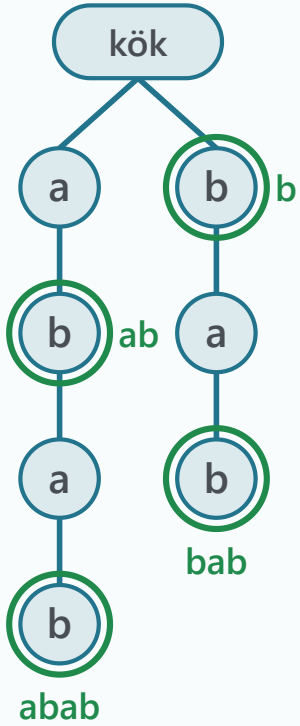
- **Tek çocuklu ve işaretsiz** düğüm zincirleri tek bir kenarda birleştirilir; kenar artık bir **alt dizgi** taşır ("an", "ad").
- Her iç düğümün en az 2 çocuğu (ya da kelime sonu işareti) olduğundan düğüm sayısı **O(n)** olur (n: kelime sayısı).
- Patricia (Morrison, 1968) bu fikrin ikili alfabe sürümüdür; IP yönlendirme tablolarında kullanılır.

Sonek Trie'si (Suffix Trie)

T = **"abab"** metninin **tüm sonekleri** bir trie'ye eklenir:

`abab` · `bab` · `ab` · `b`

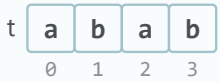
- P, T'nin bir **alt dizgisi** ⇔ P, bir sonekin öneki ⇔ P kökten başlayan bir yol.
- Sorgu: `onekVarMi(P)` → **O(m)** (m = |P|), metnin uzunluğundan bağımsız. Örn. "ba" var, "bb" yok.
- Bedel: n uzunluklu metin için **O(n^2)** düğüm olabilir.
- **Sonek ağacı** (suffix tree) = sıkıştırılmış sonek trie'si: O(n) düğüm, Ukkonen algoritmasıyla O(n) zamanda kurulur.



sonekTrie("abab")

```
1  TrieDugumu sonekTrie(String t) {
2    TrieDugumu kok = new TrieDugumu();
3    for (int s = 0; s < t.length(); s++) {
4      TrieDugumu d = kok;
5      for (int j = s; j < t.length(); j++) {
6        int i = t.charAt(j) - 'a';
7        if (d.cocuk[i] == null)
8          d.cocuk[i] = new TrieDugumu();
9        d = d.cocuk[i];
10     }
11     d.kelimeSonu = true;  // sonek bitti
12   }
13   return kok;
14 }
```





t = "abab": 4 sonek (abab, bab, ab, b) sırayla eklenecek. Trie yalnızca kökten oluşuyor.

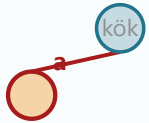
Adım 1 / 13

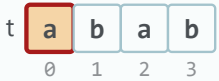
DEĞİŞKENLER

s = – j = – i = –

sonekTrie("abab")

```
1  TrieDugumu sonekTrie(String t) {
2    TrieDugumu kok = new TrieDugumu();
3    for (int s = 0; s < t.length(); s++) {
4      TrieDugumu d = kok;
5      for (int j = s; j < t.length(); j++) {
▶        int i = t.charAt(j) - 'a';
▶        if (d.cocuk[i] == null)
▶          d.cocuk[i] = new TrieDugumu();
▶        d = d.cocuk[i];
10     }
11     d.kelimeSonu = true;  // sonek bitti
12   }
13   return kok;
14 }
```





s = 0: sonek "abab" eklenmeye başlar; d = kök. s = 0, j = 0: 'a' dalı yok → yeni düğüm, d oraya iner ("a").

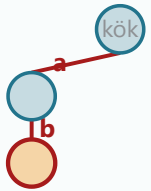
Adım 2 / 13

DEĞİŞKENLER

`s = 0` `j = 0` `i = 0`

sonekTrie("abab")

```
1  TrieDugumu sonekTrie(String t) {
2    TrieDugumu kok = new TrieDugumu();
3    for (int s = 0; s < t.length(); s++) {
4      TrieDugumu d = kok;
5      for (int j = s; j < t.length(); j++) {
▶        int i = t.charAt(j) - 'a';
▶        if (d.cocuk[i] == null)
▶          d.cocuk[i] = new TrieDugumu();
▶        d = d.cocuk[i];
10     }
11     d.kelimeSonu = true;  // sonek bitti
12   }
13   return kok;
14 }
```





s = 0, j = 1: 'b' dalı yok → yeni düğüm, d oraya iner ("ab").

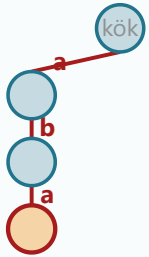
Adım 3 / 13

DEĞİŞKENLER

s = 0 | j = 1 | i = 1

sonekTrie("abab")

```
1  TrieDugumu sonekTrie(String t) {
2    TrieDugumu kok = new TrieDugumu();
3    for (int s = 0; s < t.length(); s++) {
4      TrieDugumu d = kok;
5      for (int j = s; j < t.length(); j++) {
▶        int i = t.charAt(j) - 'a';
▶        if (d.cocuk[i] == null)
▶          d.cocuk[i] = new TrieDugumu();
▶        d = d.cocuk[i];
10     }
11     d.kelimeSonu = true;  // sonek bitti
12   }
13   return kok;
14 }
```





s = 0, j = 2: 'a' dalı yok → yeni düğüm, d oraya iner ("aba").

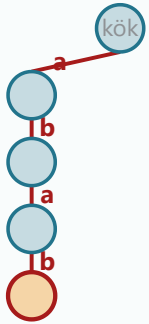
Adım 4 / 13

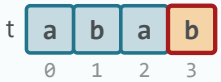
DEĞİŞKENLER

`s = 0` `j = 2` `i = 0`

sonekTrie("abab")

```
1  TrieDugumu sonekTrie(String t) {
2    TrieDugumu kok = new TrieDugumu();
3    for (int s = 0; s < t.length(); s++) {
4      TrieDugumu d = kok;
5      for (int j = s; j < t.length(); j++) {
▶        int i = t.charAt(j) - 'a';
▶        if (d.cocuk[i] == null)
▶          d.cocuk[i] = new TrieDugumu();
▶        d = d.cocuk[i];
10     }
11     d.kelimeSonu = true;  // sonek bitti
12   }
13   return kok;
14 }
```





s = 0, j = 3: 'b' dalı yok → yeni düğüm, d oraya iner ("abab").

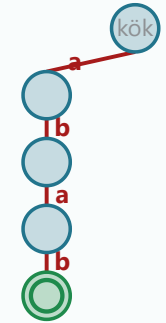
Adım 5 / 13

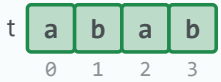
DEĞİŞKENLER

s = 0 | j = 3 | i = 1

sonekTrie("abab")

```
1  TrieDugumu sonekTrie(String t) {
2    TrieDugumu kok = new TrieDugumu();
3    for (int s = 0; s < t.length(); s++) {
4      TrieDugumu d = kok;
5      for (int j = s; j < t.length(); j++) {
6        int i = t.charAt(j) - 'a';
7        if (d.cocuk[i] == null)
8          d.cocuk[i] = new TrieDugumu();
9        d = d.cocuk[i];
10     }
▶      d.kelimeSonu = true;  // sonek bitti
12   }
13   return kok;
14 }
```





Sonek "abab" bitti → son düğüm işaretlenir (halka).

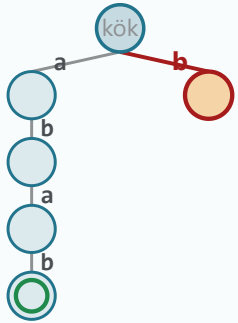
Adım 6 / 13

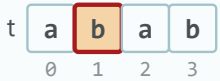
DEĞİŞKENLER

`s = 0` `j = 4` `i = 1`

sonekTrie("abab")

```
1  TrieDugumu sonekTrie(String t) {
2    TrieDugumu kok = new TrieDugumu();
3    for (int s = 0; s < t.length(); s++) {
4      TrieDugumu d = kok;
5      for (int j = s; j < t.length(); j++) {
▶        int i = t.charAt(j) - 'a';
▶        if (d.cocuk[i] == null)
▶          d.cocuk[i] = new TrieDugumu();
▶        d = d.cocuk[i];
10     }
11     d.kelimeSonu = true;  // sonek bitti
12   }
13   return kok;
14 }
```





s = 1: sonek "bab" eklenmeye başlar; d = kök. s = 1, j = 1: 'b' dalı yok → yeni düğüm, d oraya iner ("b").

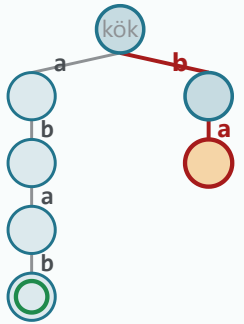
Adım 7 / 13

DEĞİŞKENLER

`s = 1` `j = 1` `i = 1`

sonekTrie("abab")

```
1  TrieDugumu sonekTrie(String t) {
2    TrieDugumu kok = new TrieDugumu();
3    for (int s = 0; s < t.length(); s++) {
4      TrieDugumu d = kok;
5      for (int j = s; j < t.length(); j++) {
▶        int i = t.charAt(j) - 'a';
▶        if (d.cocuk[i] == null)
▶          d.cocuk[i] = new TrieDugumu();
▶        d = d.cocuk[i];
10     }
11     d.kelimeSonu = true;  // sonek bitti
12   }
13   return kok;
14 }
```





s = 1, j = 2: 'a' dalı yok → yeni düğüm, d oraya iner ("ba").

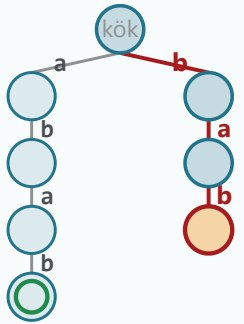
Adım 8 / 13

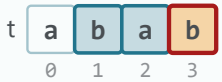
DEĞİŞKENLER

s = 1 | j = 2 | i = 0

sonekTrie("abab")

```
1  TrieDugumu sonekTrie(String t) {
2    TrieDugumu kok = new TrieDugumu();
3    for (int s = 0; s < t.length(); s++) {
4      TrieDugumu d = kok;
5      for (int j = s; j < t.length(); j++) {
▶        int i = t.charAt(j) - 'a';
▶        if (d.cocuk[i] == null)
▶          d.cocuk[i] = new TrieDugumu();
▶        d = d.cocuk[i];
10     }
11     d.kelimeSonu = true;  // sonek bitti
12   }
13   return kok;
14 }
```





s = 1, j = 3: 'b' dalı yok → yeni düğüm, d oraya iner ("bab").

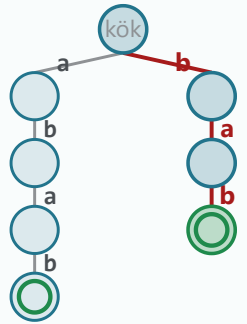
Adım 9 / 13

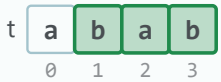
DEĞİŞKENLER

s = 1 | j = 3 | i = 1

sonekTrie("abab")

```
1  TrieDugumu sonekTrie(String t) {
2    TrieDugumu kok = new TrieDugumu();
3    for (int s = 0; s < t.length(); s++) {
4      TrieDugumu d = kok;
5      for (int j = s; j < t.length(); j++) {
6        int i = t.charAt(j) - 'a';
7        if (d.cocuk[i] == null)
8          d.cocuk[i] = new TrieDugumu();
9        d = d.cocuk[i];
10     }
▶      d.kelimeSonu = true;  // sonek bitti
12   }
13   return kok;
14 }
```





Sonek "bab" bitti → son düğüm işaretlenir (halka).

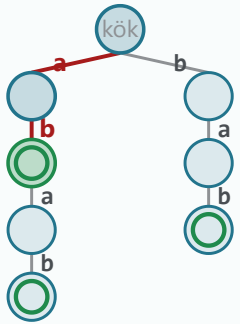
Adım 10 / 13

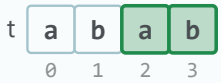
DEĞİŞKENLER

`s = 1` `j = 4` `i = 1`

sonekTrie("abab")

```
1  TrieDugumu sonekTrie(String t) {
2    TrieDugumu kok = new TrieDugumu();
3    for (int s = 0; s < t.length(); s++) {
4      TrieDugumu d = kok;
       for (int j = s; j < t.length(); j++) {
         int i = t.charAt(j) - 'a';
         if (d.cocuk[i] == null)
8          d.cocuk[i] = new TrieDugumu();
         d = d.cocuk[i];
10     }
       d.kelimeSonu = true;  // sonek bitti
12   }
13   return kok;
14 }
```





s = 2: "ab" yolu abab eklenirken zaten oluşmuştu; yeni düğüm açılmadan inilir ve "ab" işaretlenir.

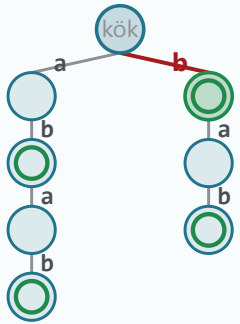
Adım 11 / 13

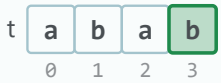
DEĞİŞKENLER

s = 2 | j = 4 | i = 1

sonekTrie("abab")

```
1  TrieDugumu sonekTrie(String t) {
2    TrieDugumu kok = new TrieDugumu();
3    for (int s = 0; s < t.length(); s++) {
4      TrieDugumu d = kok;
       for (int j = s; j < t.length(); j++) {
         int i = t.charAt(j) - 'a';
         if (d.cocuk[i] == null)
8          d.cocuk[i] = new TrieDugumu();
         d = d.cocuk[i];
10     }
       d.kelimeSonu = true;  // sonek bitti
12   }
13   return kok;
14 }
```





s = 3: "b" yolu da var; yalnızca işaret eklenir. Ortak önekler bir kez saklandı: kök dahil 8 düğüm.

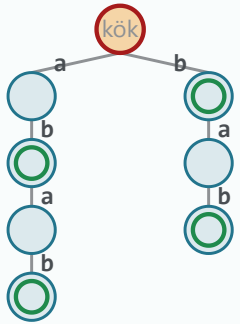
Adım 12 / 13

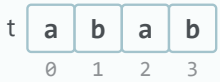
DEĞİŞKENLER

s = 3 | j = 4 | i = 1

sonekTrie("abab")

```
1  TrieDugumu sonekTrie(String t) {
2    TrieDugumu kok = new TrieDugumu();
3    for (int s = 0; s < t.length(); s++) {
4      TrieDugumu d = kok;
5      for (int j = s; j < t.length(); j++) {
6        int i = t.charAt(j) - 'a';
7        if (d.cocuk[i] == null)
8          d.cocuk[i] = new TrieDugumu();
9        d = d.cocuk[i];
10     }
11     d.kelimeSonu = true;  // sonek bitti
12   }
▶    return kok;
14 }
```





Sonek trie'si hazır. "ba" kökten bir yol → alt dizgi; "bb" yolu yok → alt dizgi değil. Sorgu O(m).

Adım 13 / 13

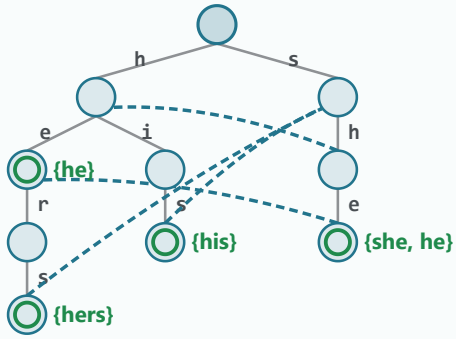
DEĞİŞKENLER

`s = 4` `j = 4` `i = 1`

Aho–Corasick: Birçok Örüntüyü Tek Geçişte Aramak

- Örüntüler (he, she, his, hers) bir trie'ye eklenir; metin **bir kez** soldan sağa okunur.
- **Hata bağlantısı** (kesikli): düğümün dizgisinin trie'de bulunan en uzun özel soneki. "she" → "he", "sh" → "h", "hers" → "s".
- Dal yoksa hata bağlantısıyla geri çekilinir; KMP önek tablosunun çok örüntülü karşılığıdır.
- Çıktı kümesi hata zinciri boyunca birleşir: "she" düğümü he'yi de bildirir.

Metin "ushers"	durum	bulunan
u, s, h	kök, s, sh	—
e (3. indis)	she	she, he
r	she → he → her	—
s (5. indis)	hers	hers



Zaman O(n + m + z): n metin, m örüntü uzunlukları toplamı, z eşleşme sayısı (Aho ve Corasick, 1975). Virüs imzası tarama ve yasaklı kelime süzgeçlerinde kullanılır.

UYGULAMALAR VE TÜREVLER

Trie Ailesi: Hangi Soruya Hangi Yapı?

Yapı	Sakladığı	Güçlü olduğu soru	Sorgu
Standart trie	kelime kümesi	tam kelime, önek, otomatik tamamlama	O(L)
Önek sayaçlı trie	+ düğümden geçen kelime sayısı	“bu önekle kaç kelime var?”	O(P)
Sıkıştırılmış trie	tek çocuklu zincirler birleşik	uzun ortak önekli anahtarlar, bellek	O(L)
Üçlü arama trie'si	düğüm başına 3 bağlantı	büyük alfabe, az bellek	O(L + log n) ort.
İkili trie	bit dizileri (IP, tam sayı)	en uzun önek eşleşmesi, en büyük XOR	O(B)
Sonek trie'si / dizisi	tüm sonekler (dizi: O(n) bellek)	“P, T'nin alt dizgisi mi?”	O(m) / O(m log n)
Aho–Corasick	örüntü trie'si + hata bağlantıları	birçok örüntüyü metinde aynı anda bulmak	O(n + z)

Ortak fikir hep aynıdır: anahtarı karakter (ya da bit) bit ilerleyerek okumak ve ortak önekleri paylaşmak. Farklar, düğümde neyin saklandığında ve çocuklara nasıl erişildiğindedir. Seçimde önce sorunun türü (tam eşleşme, önek, alt dizgi, çok örüntü), sonra alfabe büyüklüğü ve bellek bütçesi belirleyicidir.

ÖZET

Özet

Konu	Hatırlanacaklar
Yapı	boş kök; her kenar bir karakter; kelime = kökten yol; ortak önekler paylaşılır
Düğüm	`TrieDugumu[26] cocuk` + `boolean kelimeSonu`; indis = ch − 'a'
Ekleme	yoksa düğüm oluştur, varsa ilerle; sonda işaretle — O(L)
Arama	dal yoksa false; sonda `kelimeSonu` döner — O(L)
Önek	aynı yol, işaret denetimi yok — O(P); tamamla = önek düğümünden DFS
Silme	işareti kaldır; çocuksuz ve işaretsiz düğümleri aşağıdan yukarı sil
Bellek	O(N · M); seyrek düğümlerde HashMap / TST / sıkıştırılmış trie
Türevler	radix/Patricia trie, sonek trie'si → sonek ağacı, sonek dizisi

ÖZET

Etkileşimli Simülatörler

Trie ve Sonek Yapıları

Virgülle ayrılmış kelimelerden trie kurulumu ve önek sorgusu; bir metnin sonek dizisi ile sonek trie'sinin karşılaştırması.

Önek Fonksiyonu ve Z Algoritması

Aynı dizgi için doğrusal zamanda kurulan KMP önek (π) dizisi ve Z dizisi yan yana.

ÖZET

Sık Yapılan Hatalar

`kelimeSonu`'nu unutmak

Aramada yalnızca yolun varlığına bakmak her öneki kelime sayar: "da" bulunmuş gibi görünür.

ara ile onekVarMi'yi karıştırmak

Önek sorgusu son düğümün işaretine bakmaz; tam kelime araması bakmak **zorundadır**.

Alfabe dışı karakter

`'A' - 'a'` = −32, `'ş' - 'a'` = 254: dizi sınırı aşılır. Girdi küçük harfe çevrilmeli ya da HashMap kullanılmalıdır.

Silmede ortak düğümü silmek

Yolu baştan sona silmek, öneki paylaşan diğer kelimeleri de yok eder. Yalnızca çocuksuz ve işaretsiz düğümler silinir.

Karmaşıklığı n ile ifade etmek

Trie işlemleri O(log n) ya da O(n) değil, **O(L)**'dir: kelime sayısından bağımsızdır.

Belleği hafife almak

Her düğüm 26 referans taşır; milyonlarca düğümde yüzlerce MB. Seyrek trie'de HashMap ya da sıkıştırma gerekir.

ÖZET

Örnek Problemler

Kelimeler sırayla eklenir: **car, cart, care, cat, dog** (26 öğeli dizili trie).

1. Trie'yi çizin. Kök dahil kaç düğüm vardır?
2. `ara("ca")` ve `onekVarMi("ca")` ne döndürür?
3. `tamamla("car")` sonucunu sırasıyla yazın.
4. `sil("car")` ve (ilk trie'den) `sil("cart")` kaç düğüm siler?
5. Toplam kaç çocuk referansı `null` 'dır?
6. Aynı beş kelimenin **sıkıştırılmış** trie'si kök dahil kaç düğümlüdür?

▶ **Cevapları göster**

ÖZET

Alıştırmalar

1. `int kelimeSayisi(TrieDugumu d)` : trie'deki kelime sayısını bulun.
2. `int onekSayisi(String onek)` : bu önekle başlayan kelime sayısını O(P)'de döndürün.
3. `String enUzunOrtakOnek()` : trie'deki tüm kelimelerin en uzun ortak önekini bulun.
4. Çocukları `HashMap<Character, TrieDugumu>` ile saklayan, Türkçe karakterleri destekleyen trie yazın.
5. A, to, tea, ted, ten, i, in, inn anahtarlarının trie'sinde kök dahil kaç düğüm vardır?
6. Sonek trie'si ile "abab" içinde "bab" ve "bb" alt dizgilerini arayın; kaç karşılaştırma yapılır?

▶ **Cevaplar / ipuçları**

Kaynaklar

- Sedgewick ve Wayne, *Algorithms*, 4. baskı, 2011 (Bölüm 5.2: Tries).
- Cormen, Leiserson, Rivest ve Stein, *Introduction to Algorithms*, 4. baskı, 2022.
- E. Fredkin, "Trie Memory", *Communications of the ACM* 3(9), 1960.
- D. R. Morrison, "PATRICIA — Practical Algorithm To Retrieve Information Coded in Alphanumeric", *Journal of the ACM* 15(4), 1968.
- U. Manber ve G. Myers, "Suffix Arrays: A New Method for On-Line String Searches", *SIAM Journal on Computing* 22(5), 1993.
- D. Gusfield, *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, 1997.

Son

Bölüm 11: Trie Veri Yapısı

Önceki bölüm: Bölüm 10: İleri Dizgi Algoritmaları · Sonraki ders: İleri Algoritmalar · Terim sözlüğü