

Bölüm 5: Trie Veri Yapısı

Algoritmalar

Önek ağacı

Düğüm yapısı

Ekleme

Arama ve önek sorgusu

Otomatik tamamlama

Silme

Karmaşıklık ve bellek

Sıkıştırılmış ve son ek trie'leri

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

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

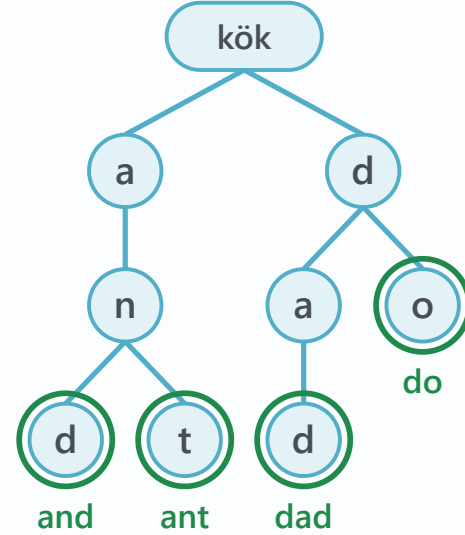
İşlemlerin $O(L)$ zamanını ve $O(N \cdot M)$ belleğini gerekçelendirmek; trie'yi hash tablosu ve BST ile karşılaştırmak.

Türevler

Sıkıştırılmış trie, sonек trie'si ve sonек dizisinin hangi sorunu çözdüğünü açıklamak.

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 *retrieval* sözcüğünden gelir (E. Fredkin, 1960). Çift çember: o düğümde bir **kelime biter**.

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 $O(n \cdot L)$. Kelime sayısı büyüdükçe yavaşlar.

Hash tablosu

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

Trie

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

Ana fikir: 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)

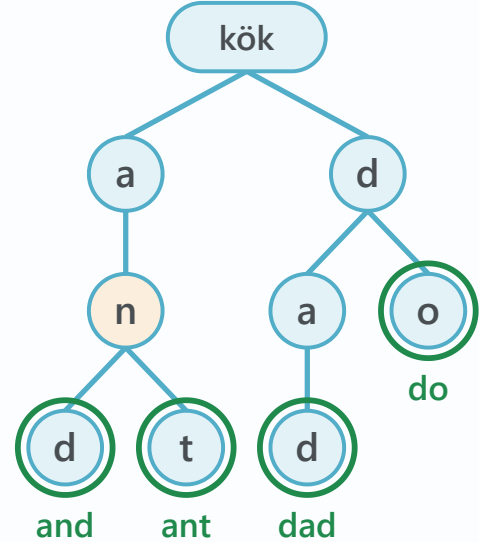
Kökten o düğüme kadarki yol bir **önek**ti; her düğüm çocuk düğümlere bağlantılar tutar.

Kenarlar (edges)

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

Kelime sonu işareti

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



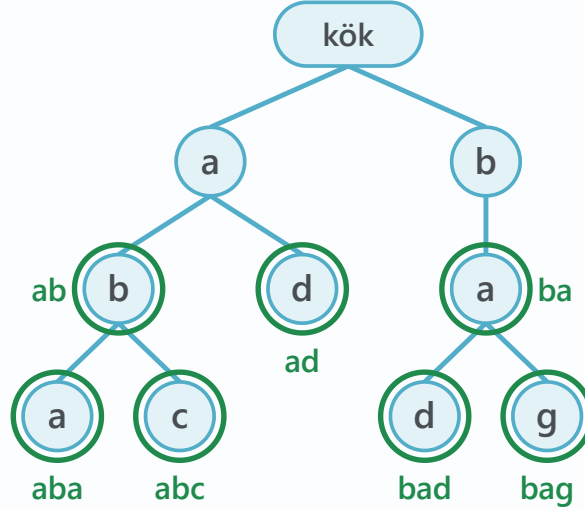
Renkli düğüm "an" önekini temsil eder; "an" bir kelime değildir (işaretsiz), ama "and" ve "ant" in önekidir.

TRIE KAVRAMI

Örnek: Yedi Kelimelik Trie

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

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

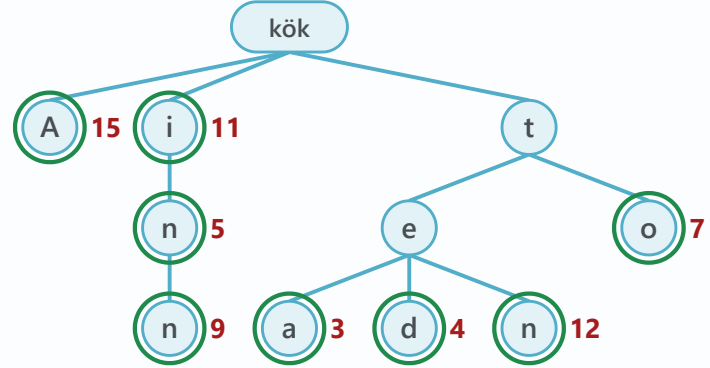


"ab" hem bir **kelime** hem de "aba" ile "abc"nin **öneki**dir. Yalnızca yapıya bakarak bunu anlayamayız; bu yüzden `kelimeSonu` işareti şarttır.

TRIE KAVRAMI

Örnek: Değer Taşıyan Trie

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



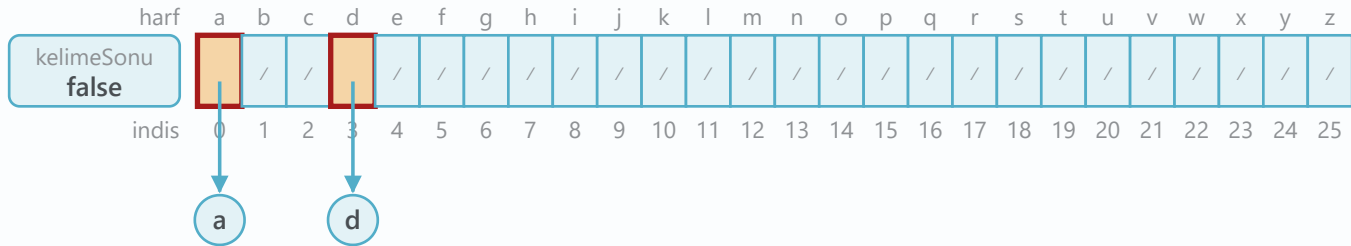
Trie bir **sözlük** (map) gibi de kullanılabilir: kelimeSonu yerine düğümde bir değer saklanır. "t" ve "te" değer taşımaz; yalnızca önektir. Büyük harf "A", 26 küçük harflik dizi yerine daha geniş bir alfabe (ör. HashMap) gerektirir.

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).

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

- 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.

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

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** başla.
2. Kelimenin her karakteri için ilgili çocuğa bak:
`cocuk[ch - 'a']`.
3. Çocuk `null` ise **yeni düğüm** oluştur;
varsa (ortak önek) **mevcut düğümü kullan**.
4. O çocuğa ilerle.
5. Karakterler bitince son düğümde
`kelimeSonu = true` yap.

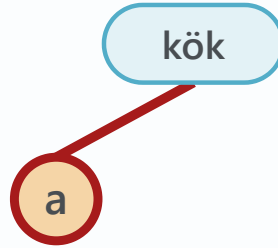
Zaman: **O(L)** (L = kelime uzunluğu); en fazla L yeni düğüm oluşur. Aynı kelimeyi iki kez eklemek yapıyı değiştirmez.

Ekleme Adım Adım: and, ant, dad, do

kök

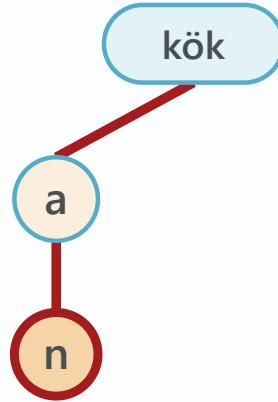
Başlangıçta trie yalnızca **kökten** oluşur; kökün 26 çocuğunun hepsi `null`, `kelimeSonu` = `false`.

Ekleme Adım Adım: and, ant, dad, do



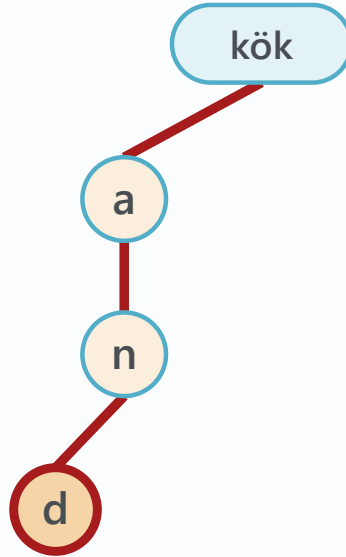
"and" ekleniyor, karakter 'a': `cocuk[0]` null → **yeni düğüm** oluşturulur ve ona ilerlenir.

Ekleme Adım Adım: and, ant, dad, do



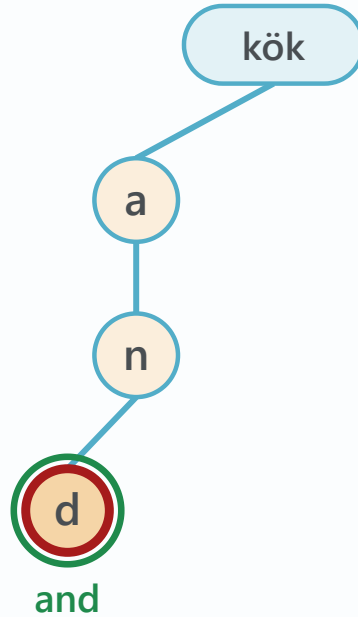
"and" ekleniyor, karakter 'n': `cocuk[13]` null → **yeni düğüm** oluşturulur ve ona ilerlenir.

Ekleme Adım Adım: and, ant, dad, do



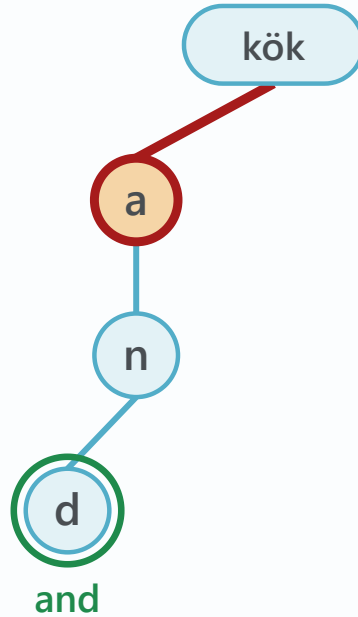
"and" ekleniyor, karakter 'd': `cocuk[3]` null → **yeni düğüm** oluşturulur ve ona ilerlenir.

Ekleme Adım Adım: and, ant, dad, do



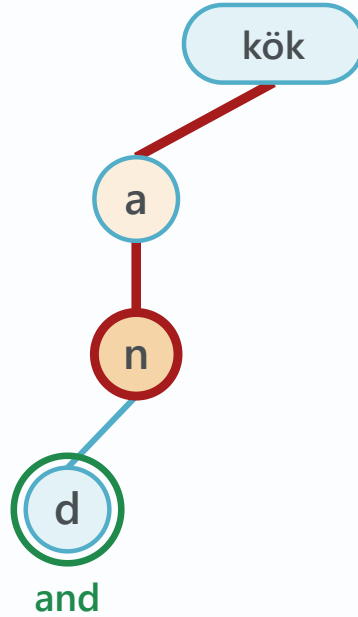
Karakterler bitti: son düğümde `kelimeSonu = true` (çift çember). "and" eklendi. Kökten itibaren `cocuk[0]` → `cocuk[13]` → `cocuk[3]` yolu kuruldu.

Ekleme Adım Adım: and, ant, dad, do



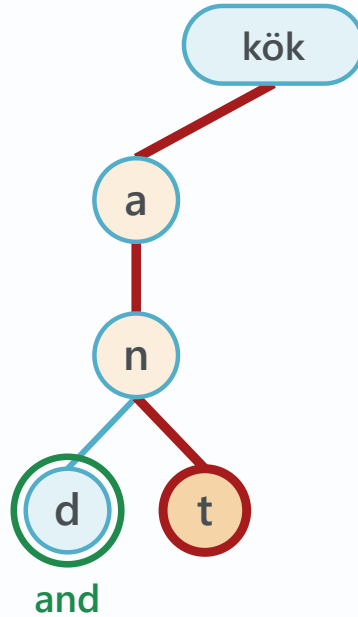
"ant" ekleniyor, karakter 'a': `cocuk[0]` dolu → **mevcut** "a" düğümüne ilerlenir (ortak önek).

Ekleme Adım Adım: and, ant, dad, do



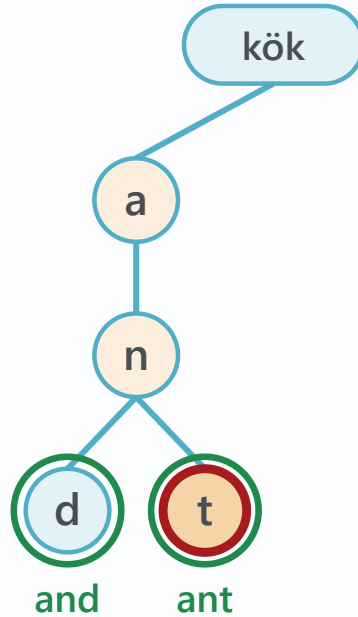
"ant" ekleniyor, karakter 'n': `cocuk[13]` dolu → **mevcut** "an" düğümüne ilerlenir (ortak önek).

Ekleme Adım Adım: and, ant, dad, do



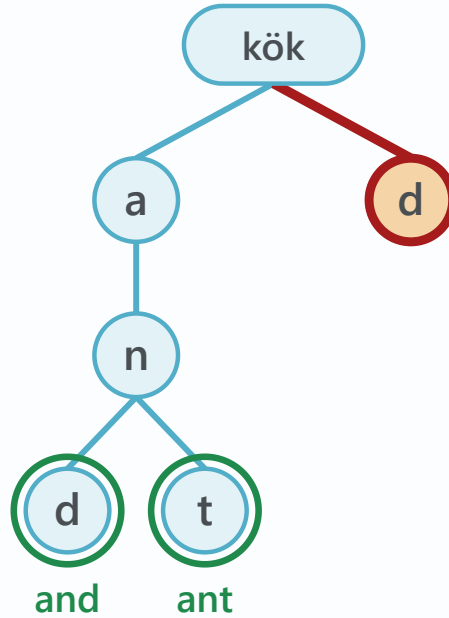
"ant" ekleniyor, karakter 't': `cocuk[19]` null → **yeni düğüm** oluşturulur ve ona ilerlenir.

Ekleme Adım Adım: and, ant, dad, do



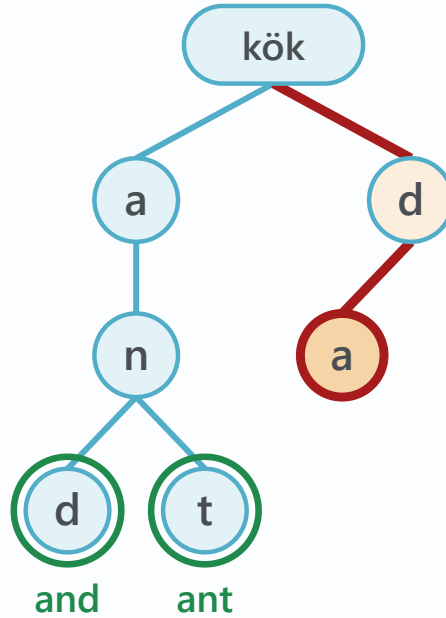
Karakterler bitti: son düğümde `keLimeSonu = true` (çift çember). "ant" eklendi. Yalnızca 1 yeni düğüm gerekti: "an" öneki paylaşıldı.

Ekleme Adım Adım: and, ant, dad, do



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

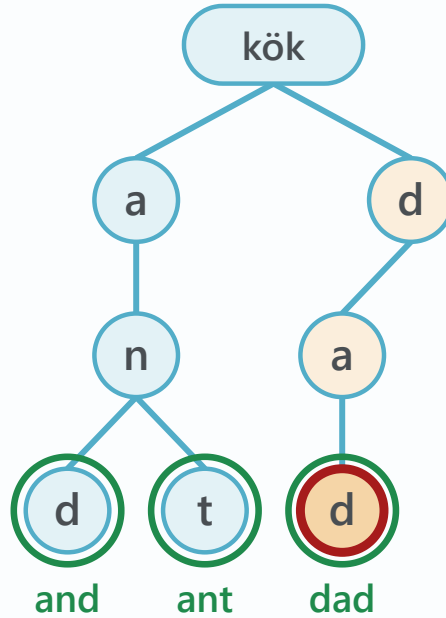
Ekleme Adım Adım: and, ant, dad, do



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

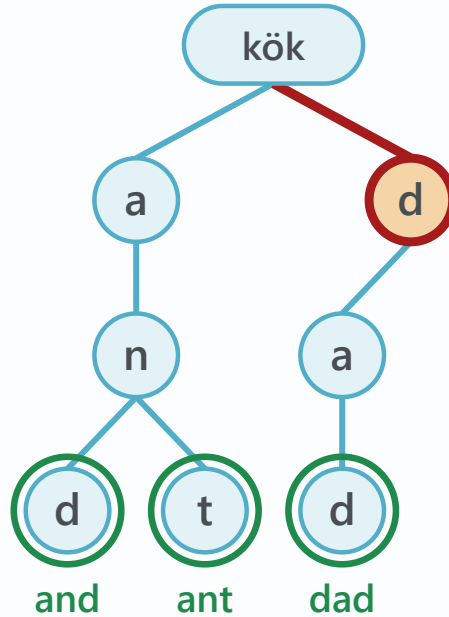


Ekleme Adım Adım: and, ant, dad, do



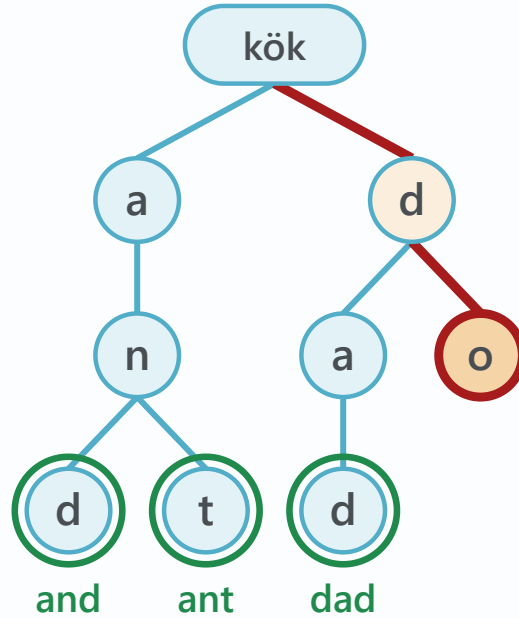
Karakterler bitti: son düğümde `kelimeSonu = true` (çift çember). "dad" eklendi.

Ekleme Adım Adım: and, ant, dad, do



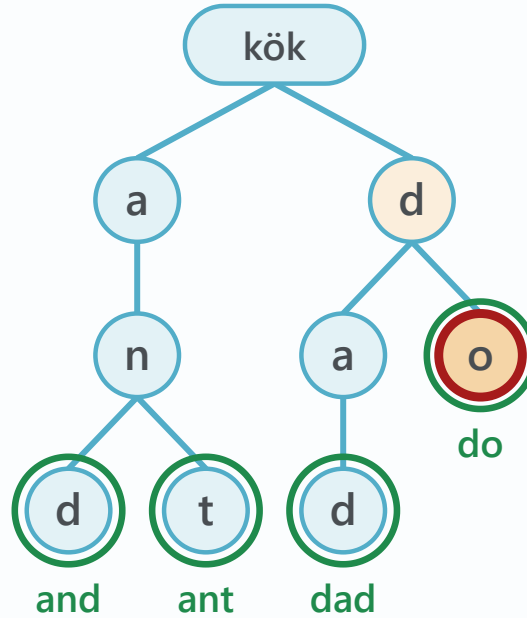
"do" ekleniyor, karakter 'd': `cocuk[3]` dolu → **mevcut** "d" düğümüne ilerlenir (ortak önek).

Ekleme Adım Adım: and, ant, dad, do



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

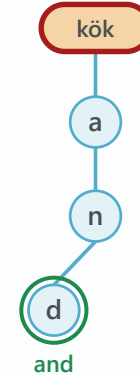
Ekleme Adım Adım: and, ant, dad, do



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) {  
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     }  
9     dugum.kelimeSonu = true;  
10 }
```



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

Adım 1 / 12

DEĞİŞKENLER

kelime = "ant"

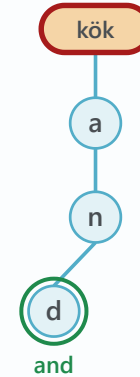
dugum = kök

ch = -

i = -

ekle("ant"): Kodu Adım Adım İzle

```
1 public void ekle(String kelime) {  
2     TrieDugumu dugum = kok;  
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 }
```



Karakter 'a' → $i = 'a' - 'a' = 0$.

Adım 2 / 12

DEĞİŞKENLER

kelime = "ant"

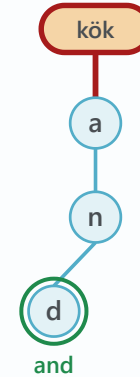
dugum = kök

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';  
5         if (dugum.cocuk[i] == null)  
6             dugum.cocuk[i] = new TrieDugumu();  
7         dugum = dugum.cocuk[i];  
8     }  
9     dugum.kelimeSonu = true;  
10 }
```



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

Adım 3 / 12

DEĞİŞKENLER

kelime = "ant"

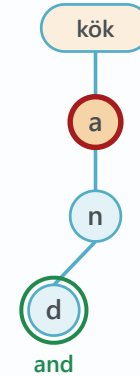
dugum = kök

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();  
7         ▶     dugum = dugum.cocuk[i];  
8     }  
9     dugum.kelimeSonu = true;  
10 }
```



dugum artık "a" düğümü.

Adım 4 / 12

DEĞİŞKENLER

kelime = "ant"

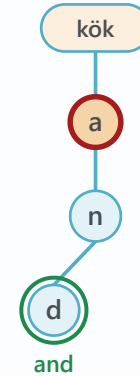
dugum = "a"

ch = 'a'

i = 0

ekle("ant"): Kodu Adım Adım İzle

```
1 public void ekle(String kelime) {  
2     TrieDugumu dugum = kok;  
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 }
```



Karakter 'n' → $i = 'n' - 'a' = 13$.

Adım 5 / 12

DEĞİŞKENLER

kelime = "ant"

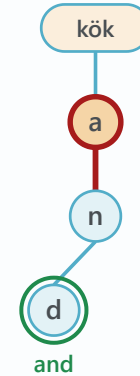
dugum = "a"

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';  
5         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"

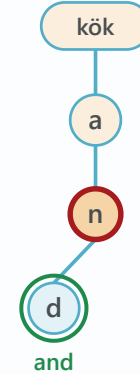
dugum = "a"

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();  
7         ▶     dugum = dugum.cocuk[i];  
8     }  
9     dugum.kelimeSonu = true;  
10 }
```



dugum artık "an" düğümü.

Adım 7 / 12

DEĞİŞKENLER

kelime = "ant"

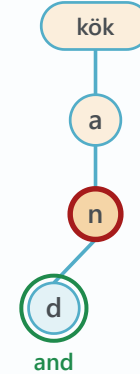
dugum = "an"

ch = 'n'

i = 13

ekle("ant"): Kodu Adım Adım İzle

```
1 public void ekle(String kelime) {  
2     TrieDugumu dugum = kok;  
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 }
```



Karakter 't' → $i = 't' - 'a' = 19$.

Adım 8 / 12

DEĞİŞKENLER

kelime = "ant"

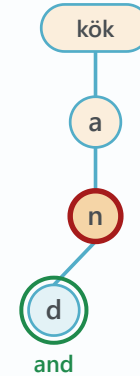
dugum = "an"

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';  
5         ▶ 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"

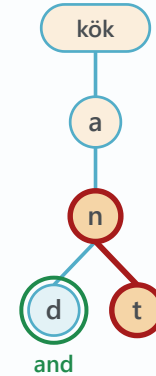
dugum = "an"

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)  
6             ▶ dugum.cocuk[i] = new TrieDugumu();  
7         dugum = dugum.cocuk[i];  
8     }  
9     dugum.kelimeSonu = true;  
10 }
```



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

Adım 10 / 12

DEĞİŞKENLER

kelime = "ant"

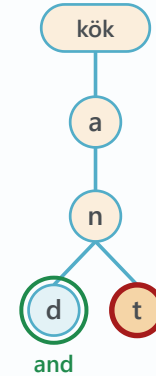
dugum = "an"

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();  
7         ▶     dugum = dugum.cocuk[i];  
8     }  
9     dugum.kelimeSonu = true;  
10 }
```



dugum artık "ant" düğümü.

Adım 11 / 12

DEĞİŞKENLER

kelime = "ant"

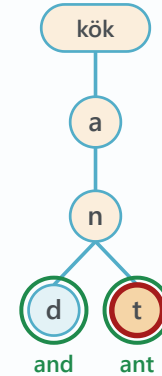
dugum = "ant"

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     }  
9     ▶ 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 (Search)

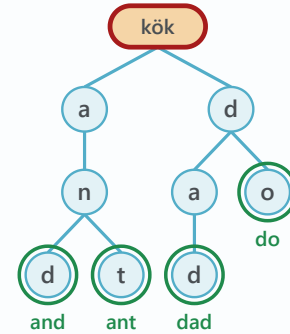
```
public boolean ara(String kelime) {
    TrieDugumu dugum = kok;
    for (char ch : kelime.toCharArray()) {
        int i = ch - 'a';
        if (dugum.cocuk[i] == null)
            return false;
        dugum = dugum.cocuk[i];
    }
    return dugum.kelimeSonu;
}
```

- **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.

ara("dad"): Kodu Adım Adım İzle

```
1 public boolean ara(String kelime) {  
2     TrieDugumu dugum = kök;  
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

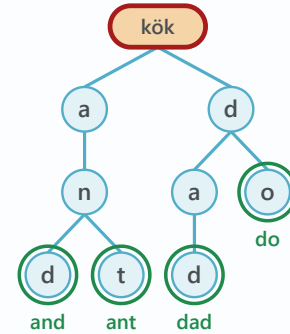
ch = -

i = -

ÇIKTI

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     }  
9     return dugum.kelimeSonu;  
10 }
```



Karakter 'd' → **i = 3**.

Adım 2 / 11

DEĞİŞKENLER

kelime = "dad"

dugum = kök

ch = 'd'

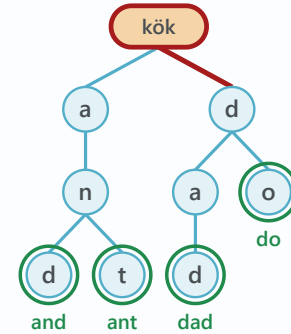
i = 3

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;  
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

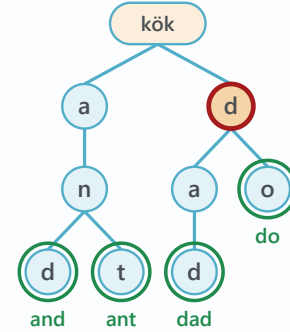
ch = 'd'

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;  
7         ▶     dugum = dugum.cocuk[i];  
8     }  
9     return dugum.kelimeSonu;  
10 }
```



dugum = "d".

Adım 4 / 11

DEĞİŞKENLER

kelime = "dad"

dugum = "d"

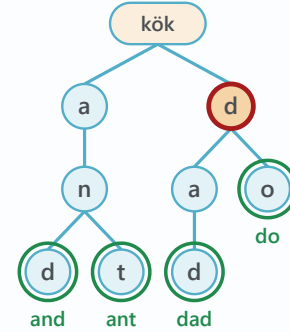
ch = 'd'

i = 3

ÇIKTI

ara("dad"): Kodu Adım Adım İzle

```
1 public boolean ara(String kelime) {  
2     TrieDugumu dugum = kok;  
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 }
```



Karakter 'a' → **i = 0**.

Adım 5 / 11

DEĞİŞKENLER

kelime = "dad"

dugum = "d"

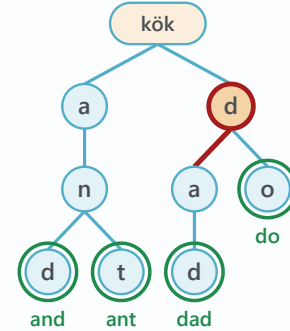
ch = 'a'

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';  
5         ► 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'

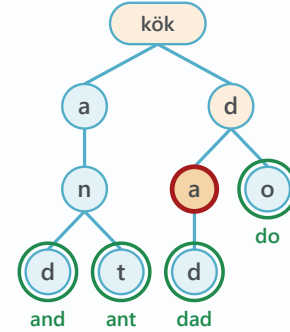
i = 0

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;  
7         ▶ dugum = dugum.cocuk[i];  
8     }  
9     return dugum.kelimeSonu;  
10 }
```



dugum = "da".

Adım 7 / 11

DEĞİŞKENLER

kelime = "dad"

dugum = "da"

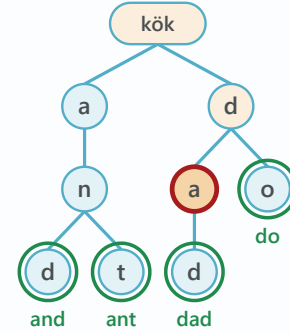
ch = 'a'

i = 0

ÇIKTI

ara("dad"): Kodu Adım Adım İzle

```
1 public boolean ara(String kelime) {  
2     TrieDugumu dugum = kok;  
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 }
```



Karakter 'd' → **i = 3**.

Adım 8 / 11

DEĞİŞKENLER

kelime = "dad"

dugum = "da"

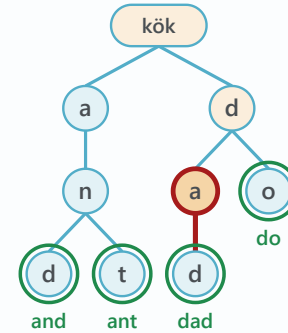
ch = 'd'

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';  
5         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"

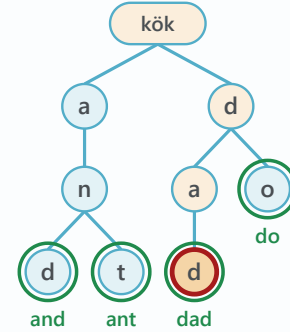
ch = 'd'

i = 3

ÇIKTI

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     }  
9     return dugum.kelimeSonu;  
10 }
```



dugum = "dad".

Adım 10 / 11

DEĞİŞKENLER

kelime = "dad"

dugum = "dad"

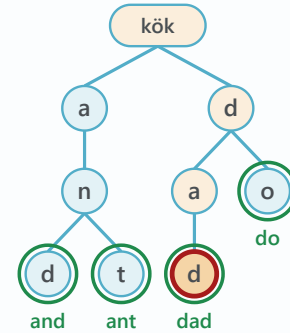
ch = 'd'

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     }  
9     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

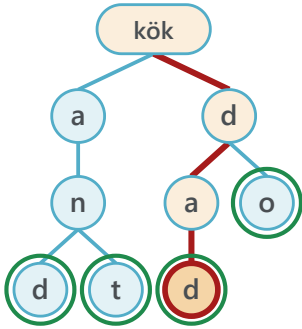
ÇIKTI

ara("dad") → true

Aramanın Üç Olası Sonucu

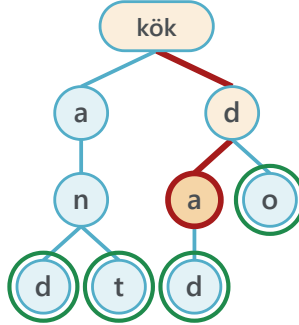
ara("dad") → true

Tüm karakterler bulundu ve son düğüm işaretli.



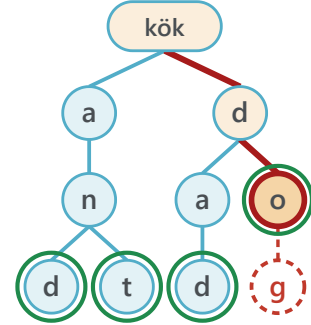
ara("da") → false

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.

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

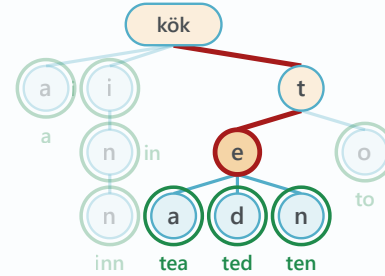
Tek fark **son satırdadır**: önek sorgusu kelime sonunu denetlemez; yolun var olması yeterlidir.
Trie: and, ant, dad, do. Zaman: $O(P)$, P = önek uzunluğu.

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)
11        sonuc.add(yol.toString());
12    for (int i = 0; i < 26; i++)
13        if (d.cocuk[i] != null) {
14            yol.append((char) ('a' + i));
15            topla(d.cocuk[i], yol, sonuc);
16            yol.setLength(yol.length() - 1);
17        }
18 }

```



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

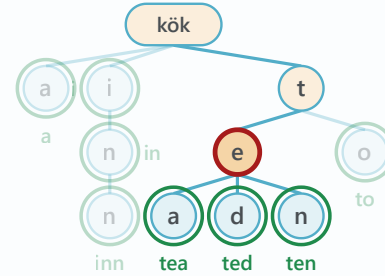
sonuc = []

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)
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)
11        sonuc.add(yol.toString());
12    for (int i = 0; i < 26; i++)
13        if (d.cocuk[i] != null) {
14            yol.append((char) ('a' + i));
15            topla(d.cocuk[i], yol, sonuc);
16            yol.setLength(yol.length() - 1);
17        }
18 }

```



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

Adım 2 / 16

DEĞİŞKENLER

onek = "te"

yol = "te" i = -

ÇAĞRI YIĞINI

topla("te")

tamamla("te")

ÇIKTI

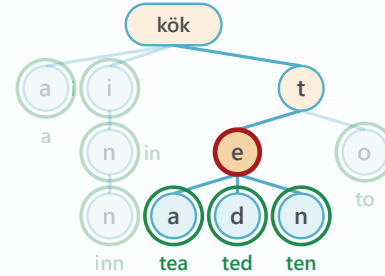
sonuc = []

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

```



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

Adım 3 / 16

DEĞİŞKENLER

onek = "te"

yol = "te" i = -

ÇAĞRI YİĞİNİ

toplama("te")

tamamla("te")

ÇIKTI

sonuc = []

Adım 4 / 16

ÇAĞRI YIĞINI

```
topla("te")
```

```
tamamla("te")
```

```
sonuc = []
```

Adım 5 / 16

ÇAĞRI YIĞINI

```
topla("tea")
```

 $i = -$

```
topla("te")
```

```
tamamla("te")
```

```
sonuc = []
```

```

graph TD
    kök --> a1[a]
    kök --> i[i]
    kök --> t[t]
    a1 --> a2[a]
    a1 --> n1[n]
    a1 --> n2[n]
    i --> n3[n]
    t --> e[e]
    t --> o[o]
    e --> a3[a]
    e --> d[d]
    e --> n3[n]
    a2 --- a_syll[a]
    n1 --- in_syll[in]
    n2 --- inn_syll[inn]
    a3 --- tea_syll[tea]
    d --- ted_syll[ted]
    n3 --- ten_syll[ten]
    o --- to_syll[to]
  
```

Adım 6 / 16

```
onek = "te"
```

```
yol = "tea"
```

$$i = -$$

```
topla("tea")
```

```
topla("te")
```

```
tamamla("te")
```

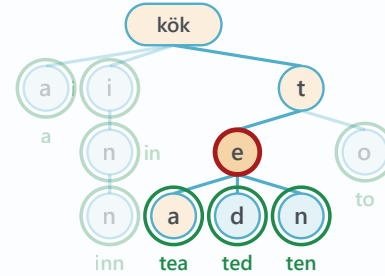
```
sonuc = [tea]
```

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)
11        sonuc.add(yol.toString());
12    for (int i = 0; i < 26; i++)
13        if (d.cocuk[i] != null) {
14            yol.append((char) ('a' + i));
15            topla(d.cocuk[i], yol, sonuc);
16            yol.setLength(yol.length() - 1);
17        }
18 }

```



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

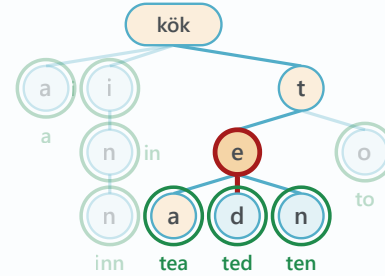
`sonuc = [tea]`

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     toplama(d, new StringBuilder(onek), sonuc);
6   return sonuc;
7 }
8 void toplama(TrieDugumu d, StringBuilder yol,
9              List<String> sonuc) {
10  if (d.kelimeSonu)
11    sonuc.add(yol.toString());
12  for (int i = 0; i < 26; i++)
13    if (d.cocuk[i] != null) {
14      yol.append((char) ('a' + i));
15      toplama(d.cocuk[i], yol, sonuc);
16      yol.setLength(yol.length() - 1);
17    }
18 }

```



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

Adım 8 / 16

DEĞİŞKENLER

onek = "te"

yol = "ted" i = 3

ÇAĞRI YIĞINI

topla("te")

tamamla("te")

ÇIKTI

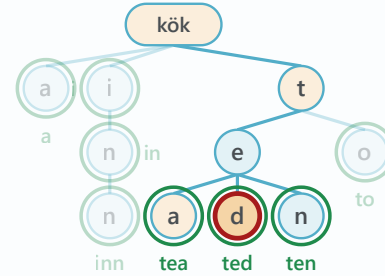
sonuc = [tea]

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         toplad(d, new StringBuilder(onek), sonuc);
6     return sonuc;
7 }
8 void toplad(TrieDugumu d, StringBuilder yol,
9             List<String> sonuc) {
10    if (d.kelimeSonu)
11        sonuc.add(yol.toString());
12    for (int i = 0; i < 26; i++)
13        if (d.cocuk[i] != null) {
14            yol.append((char) ('a' + i));
15            ► toplad(d.cocuk[i], yol, sonuc);
16            yol.setLength(yol.length() - 1);
17        }
18 }

```



Özyinelemeli çağrı: `topla("ted")` .

Adım 9 / 16

DEĞİŞKENLER

onek = "te"

yol = "ted"

i = -

ÇAĞRI YIĞINI

topla("ted")

topla("te")

tamamla("te")

ÇIKTI

sonuc = [tea]

```

graph TD
    kitten[kitten] --> k[k]
    kitten --> i[i]
    kitten --> ten[ten]
    i --> a[a]
    i --> n1[n]
    a --> n2[n]
    n1 --> inn[inn]
    ten --> t[t]
    ten --> e[e]
    ten --> n3[n]
    t --> e2[e]
    t --> o[o]
    e --> a2[a]
    e --> d[d]
    e --> n4[n]
    a2 --> tea[tea]
    d --> ted[ted]
    n4 --> ten2[ten]
  
```

Adım 10 / 16

```
onek = "te"
```

```
yol = "ted"
```

$$i = -$$

```
topla("ted")
```

```
topla("te")
```

```
tamamla("te")
```

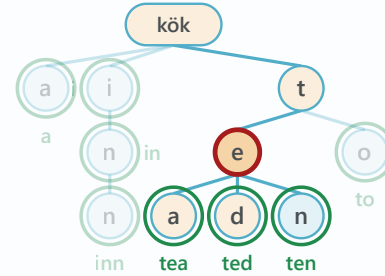
```
sonuc = [tea, ted]
```

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)
11        sonuc.add(yol.toString());
12    for (int i = 0; i < 26; i++)
13        if (d.cocuk[i] != null) {
14            yol.append((char) ('a' + i));
15            topla(d.cocuk[i], yol, sonuc);
16            yol.setLength(yol.length() - 1);
17        }
18 }

```



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]`

Adım 12 / 16

ÇAĞRI YIĞINI

```
topla("te")
```

```
tamamla("te")
```

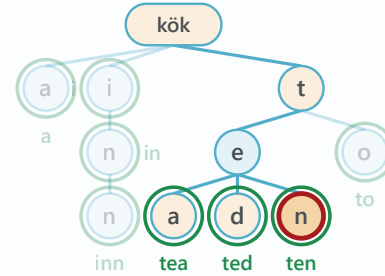
```
sonuc = [tea, ted]
```

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         toplamla(d, new StringBuilder(onek), sonuc);
6     return sonuc;
7 }
8 void toplamla(TrieDugumu d, StringBuilder yol,
9               List<String> sonuc) {
10    if (d.kelimeSonu)
11        sonuc.add(yol.toString());
12    for (int i = 0; i < 26; i++)
13        if (d.cocuk[i] != null) {
14            yol.append((char) ('a' + i));
15            toplamla(d.cocuk[i], yol, sonuc);
16            yol.setLength(yol.length() - 1);
17        }
18 }

```



Özyinelemeli çağrı: `toplama("ten")`.

Adım 13 / 16

DEĞİŞKENLER

onek = "te"

yol = "ten"

i = -

ÇAĞRI YIĞINI

toplama("ten")

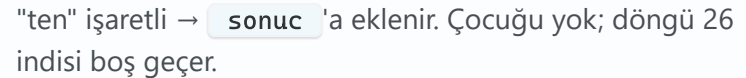
toplama("te")

tamamla("te")

ÇIKTI

sonuc = [tea, ted]

```
▶ if (d.kelimeSonu)
    sonuc.add(yol.toString());
```



DEĞİŞKENLER

```
onek = "te"
```

```
    yol = "ten"
```

$$i = -$$

ÇAĞRI YIĞINI

```
topla("ten")
```

```
topla("te")
```

```
tamamla("te")
```

ÇIKTI

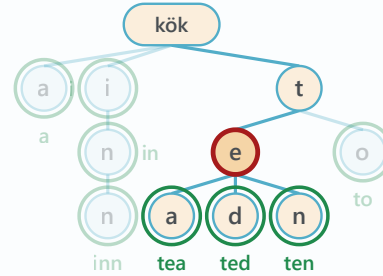
```
sonuc = [tea, ted, ten]
```

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)
11        sonuc.add(yol.toString());
12    for (int i = 0; i < 26; i++)
13        if (d.cocuk[i] != null) {
14            yol.append((char) ('a' + i));
15            topla(d.cocuk[i], yol, sonuc);
16            yol.setLength(yol.length() - 1);
17        }
18 }

```



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

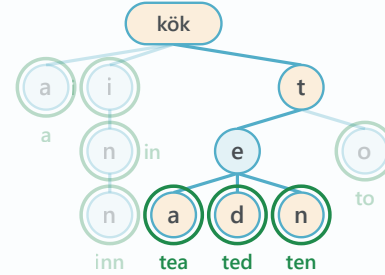
`sonuc = [tea, ted, ten]`

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);
6     return sonuc;
7 }
8 void topla(TrieDugumu d, StringBuilder yol,
9           List<String> sonuc) {
10    if (d.kelimeSonu)
11        sonuc.add(yol.toString());
12    for (int i = 0; i < 26; i++)
13        if (d.cocuk[i] != null) {
14            yol.append((char) ('a' + i));
15            topla(d.cocuk[i], yol, sonuc);
16            yol.setLength(yol.length() - 1);
17        }
18 }

```



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]

Deneyin: Trie Üzerinde İşlemler

Kelime

ekle

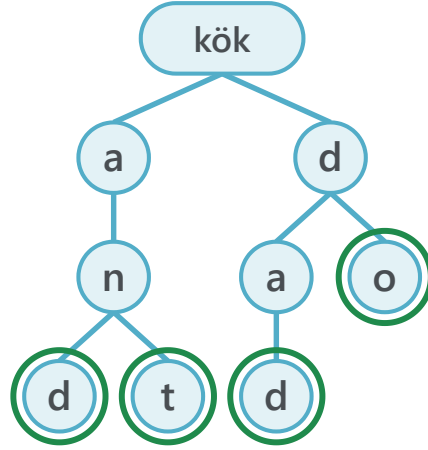
ara

örnek / 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 işaretli** düğümler silinir; paylaşılan önekte durulur.

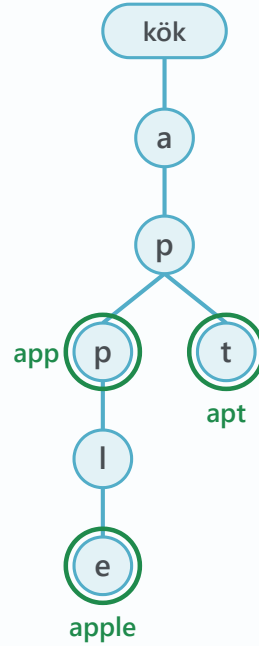
3. Kelime trie'de yok

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

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

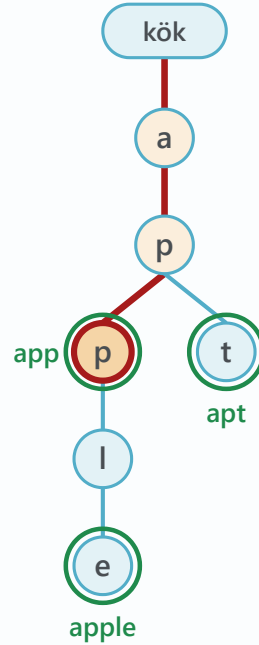
► Cevabı göster

Silme Adım Adım: app, apple, apt



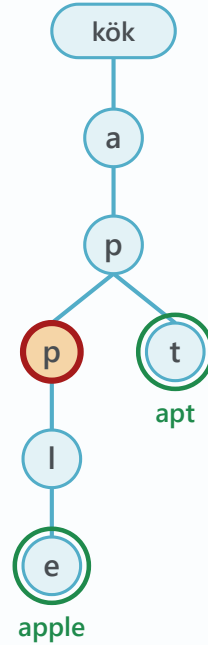
Başlangıç: app, apple, apt (kök hariç 6 düğüm). Aşağıdaki iki senaryo bu trie'den başlar.

Silme Adım Adım: app, apple, apt



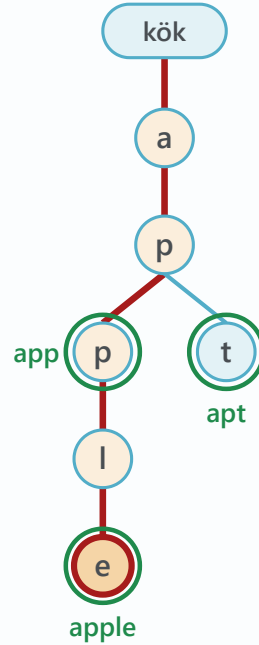
sil("app"): $a \rightarrow p \rightarrow p$ yolu izlenir; "app" düğümü işaretli, yani kelime var.

Silme Adım Adım: app, apple, apt



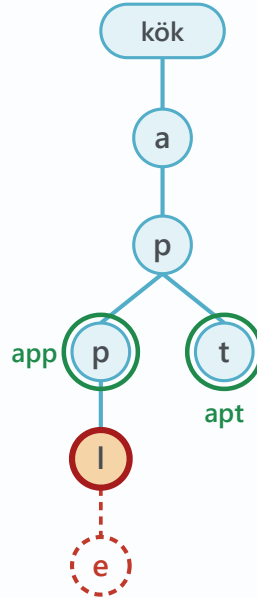
`kelimeSonu = false` . Düğümün çocuğu ('l') olduğu için **silinmez** → durum 1. "apple" hâlâ bulunur.

Silme Adım Adım: app, apple, apt



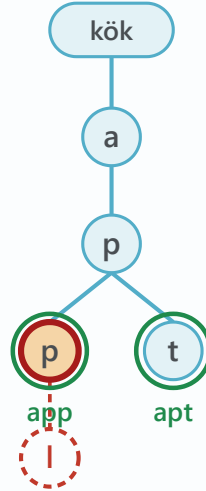
sil("apple") (yeniden ilk trie'den): yol a-p-p-l-e izlenir; "apple" bulundu.

Silme Adım Adım: app, apple, apt



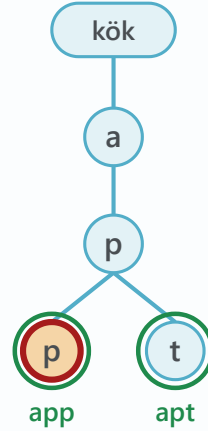
'e' düğümünün işareti kalkar; çocuğu yok → ebeveyn onu siler (`cocuk[4] = null`).

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: $O(L \cdot M)$ dizi ile; çocuk sayısı tutulursa $O(L)$.

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)	$O(P + S \cdot M)$	S: önek düğümünün alt ağacındaki düğüm sayısı
sil(kelime)	$O(L \cdot M)$	bosMu 26 hücreyi tarar; çocuk sayacıyla $O(L)$

Bellek: $O(N \cdot M)$

N: düğüm sayısı, M: alfabe boyutu (26). $N \leq$ (tüm kelimelerin toplam uzunluğu) + 1.

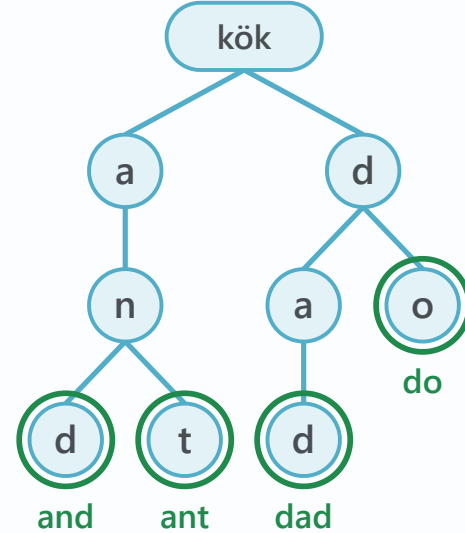
n'den bağımsız

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

Bellek: Sayısal Örnek

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

Toplam karakter	$3 + 3 + 3 + 2 = 11$
Düğüm sayısı N	$1 (\text{kök}) + 8 = 9$
Çocuk referansı	$9 \times 26 = 234$
Dolu referans	8 (kenar sayısı)
Boş (null) referans	$226 (\approx \%97)$



Ortak önekler 11 karakteri 8 düğüme indirir, ama 26 öğeli diziler belleğin büyük kısmını **null** ile doldurur. Alfabe büyüdükçe (Unicode: 65 536+) sorun büyür.

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.
- Bellek karmaşıklığı **$O(N \cdot M)$** (N: düğüm sayısı, M: alfabe boyutu).
- Ortak öneki az olan kümelerde (ör. rastgele anahtarlar) kazanç yoktur.
- Çok sayıda küçük nesne: ön bellek (cache) dostu değildir.

Çocukları Saklamanın Yolları

Gösterim	Çocuğa erişim	Düğüm başına bellek	Not
<code>TrieDugumu[26]</code> dizi	$O(1)$	M referans	Basit, hızlı; seyrek düğümlerde israf
<code>HashMap<Character, TrieDugumu></code>	$O(1)$ ort.	çocuk sayısı kadar	Her alfabe (Türkçe, Unicode) çalışır; sıra korunmaz
<code>TreeMap</code> / 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 **mantiği** değişmez; yalnızca `dugum.cocuk[i]` yerine `dugum.cocuk.get(ch)` gibi bir çağrı gelir. Seçim: alfabe boyutu, 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	$O(L \cdot \log n)$
Önek sorgusu	$O(P)$	desteklenmez	$O(P \cdot \log n)$
Önekle başlayan k kelime	$O(P + S \cdot M)$	tüm tabloyu tara	$O(P \cdot \log n + k \cdot L)$
Sıralı listeleme	DFS, $O(N \cdot M)$	sıralama gerekir	kök ortada, $O(n)$
En kötü durum	$O(L)$	$O(n \cdot L)$ (çakışma)	$O(L \cdot \log n)$
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.

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

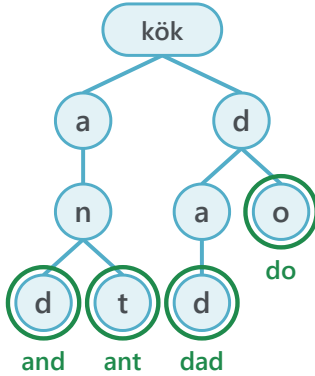
Boggle / kelime bulmaca: yol bir önek değilse arama hemen kesilir (budama). Tuş dizisinden kelime önerme.

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

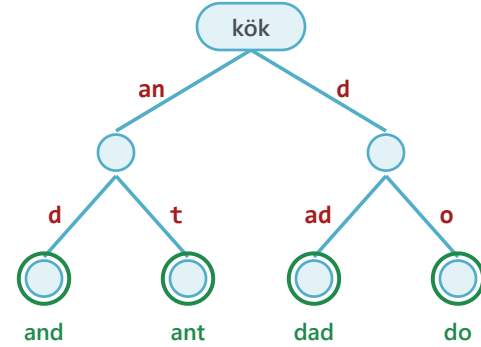


Sıkıştırılmış Trie (Radix / Patricia)

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.

abab · bab · ab · b

-
- ```

graph TD
 kök([kök]) --> a1((a))
 kök --> b1((b))
 a1 --> b2((b))
 b1 --> a2((a))
 b2 --> a3((a))
 a3 --> b3((b))
 a2 --> b4((b))
 style kök fill:#d9e1f2,stroke:#4682b4,stroke-width:2px
 style a1 fill:#d9e1f2,stroke:#4682b4,stroke-width:2px
 style b1 fill:#d9e1f2,stroke:#4682b4,stroke-width:2px
 style b2 fill:#d9e1f2,stroke:#4682b4,stroke-width:2px
 style a2 fill:#d9e1f2,stroke:#4682b4,stroke-width:2px
 style a3 fill:#d9e1f2,stroke:#4682b4,stroke-width:2px
 style b3 fill:#d9e1f2,stroke:#4682b4,stroke-width:2px
 style b4 fill:#d9e1f2,stroke:#4682b4,stroke-width:2px
 style b2 stroke:#008000,stroke-width:4px
 style b4 stroke:#008000,stroke-width:4px
 linkStyle 0,2,4,6,8 stroke:#4682b4,stroke-width:2px
 linkStyle 1,3,5,7,9 stroke:#008000,stroke-width:4px

```

## Sonek Dizisi (Suffix Array)

| sıra | SA[sıra] | sonek    |
|------|----------|----------|
| 0    | 6        | a        |
| 1    | 2        | abama    |
| 2    | 0        | a labama |
| 3    | 4        | ama      |
| 4    | 3        | bama     |
| 5    | 1        | labama   |
| 6    | 5        | ma       |

- Sonek trie'sinin bellek dostu alternatifi: soneklerin **başlangıç indisleri**, sözlük sırasına göre dizilir.
- "alabama" → SA = [6, 2, 0, 4, 3, 1, 5].
- Kurulum: tüm sonekleri oluşturup `Arrays.sort` ile sırala →  $O(n^2 \log n)$  en kötü; özel algoritmalarla  $O(n \log n)$  ya da  $O(n)$ .
- Arama: SA üzerinde **ikili arama** →  $O(m \log n)$ .  
Örn. "ama" için 3. sıradaki sonek (indis 4) bulunur.
- Yalnızca  $n$  tam sayı: trie'deki  $O(n^2)$  düğüm yerine  **$O(n)$**  bellek.

## ÖZET

# Özet

| Konu     | Hatırlanacaklar                                                                                                  |
|----------|------------------------------------------------------------------------------------------------------------------|
| Yapı     | boş kök; her kenar bir karakter; kelime = kökten yol; ortak önekler paylaşılır                                   |
| Düğüm    | <code>TrieDugumu[26]</code> <code>cocuk</code> + <code>boolean kelimeSonu</code> ; <code>indis = ch - 'a'</code> |
| Ekleme   | yoksa düğüm oluştur, varsa ilerle; sonda işaretle — $O(L)$                                                       |
| Arama    | dal yoksa false; sonda <code>kelimeSonu</code> 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   | $O(N \cdot M)$ ; seyrek düğümlerde HashMap / TST / sıkıştırılmış trie                                            |
| Türevler | radix/Patricia trie, sonek trie'si → sonek ağacı, sonek dizisi                                                   |

# Etkileşimli Simülatörler

---

### Trie ve Sonek Yapıları

Virgülle ayrılmış kelimelerden trie kurun ve önek sorgusunu izleyin; bir metnin sonek dizisini ve sonek trie'sini karşılaştırın.

### Önek Fonksiyonu ve Z Algoritması

Aynı dizgi için KMP önek ( $\pi$ ) dizisini ve Z dizisini doğrusal zamanda kurup yan yana karşılaştırın.

## 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. Girdiyi küçük harfe çevirin ya da HashMap kullanın.

### Silmede ortak düğümü silmek

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

### Karmaşıklık 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. "banana" metninin son ek dizisini bulun.

► 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 5: Trie Veri Yapısı

Önceki bölüm: Bölüm 5: Dizgi Algoritmaları (2) · Sonraki bölüm: Bölüm 6: Dinamik Programlama · [Terim sözlüğü](#)