Bölüm 15: Çizgeler

C ile Veri Yapıları ve Algoritmalar

Königsberg | Tanımlar | Matris ve listeler | Çok bağlı liste | DFS | BFS | Bileşenler

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

GİRİŞ

Öğrenme Çıktıları

Tanımlamak

G = (V, E), yönlü/yönsüz çizge, komşuluk, alt çizge, yol, derece ve bağlantılılık kavramlarını kullanmak.

İspatlamak

Derece toplamı = 2e (Teorem 1) ve bağlantılılık için $e \geq n - 1$ koşulunu gerekçelendirmek.

Gerçeklemek

Komşuluk matrisi, komşuluk listesi, sıralı dizi, ortogonal liste ve çok bağlı listeyi C ile kurmak.

Gezinmek

Önce derinliğine (DFS) ve önce genişliğine (BFS) aramayı kodlayıp izlemek, O(n + e) analizini yapmak.

Uygulamak

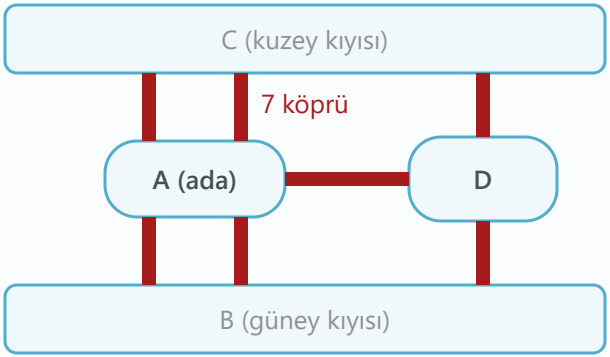
Bağlantılı bileşenleri, döngüyü ve iki parçalılığı DFS/BFS ile bulmak.

Seçmek

Problemin işlemlerine bakarak uygun gösterimi seçmek: “önce veri yapısı, sonra kod”.

GİRİŞ

Problem: Königsberg'in 7 Köprüsü



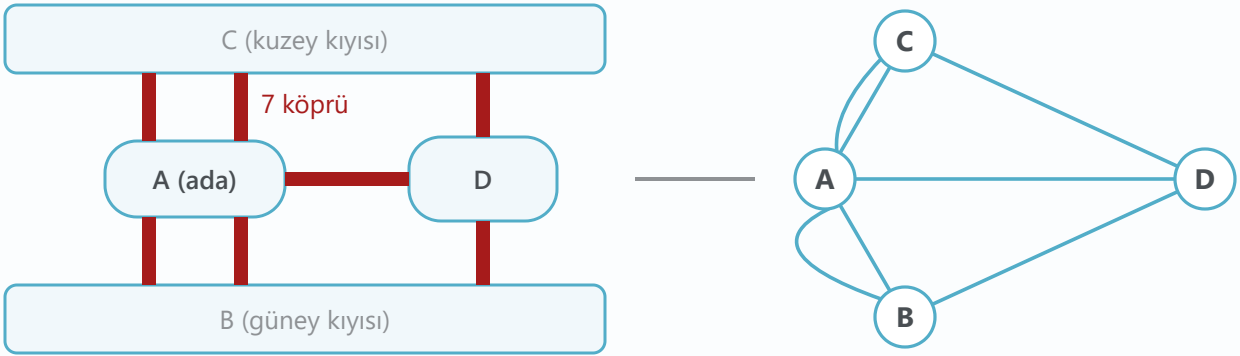
Pregel nehrinde iki ada (A, D) ve iki kıyı (B, C) 7 köprüyle bağlı. **Her köprüden bir kez geçerek başlanan yere dönülebilir mi?**

Adım 1 / 3

Çizge kuramının başlangıcı sayılır. Euler (*Oyler* okunur) sonucu teorem olarak verdi; ispatı **Çizge Kuramı** derslerine kalıyor (4. sınıf: "Çizge Kuramı ve Kombinatorik").

GİRİŞ

Problem: Königsberg'in 7 Köprüsü

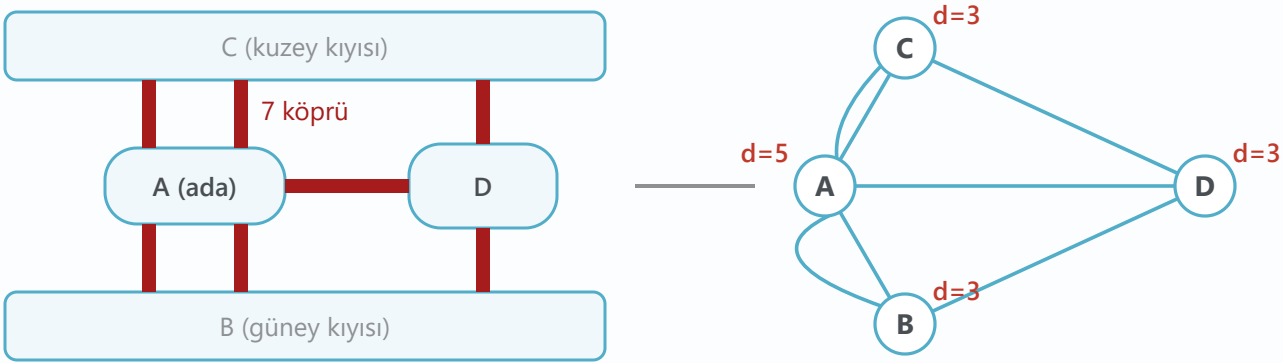


Euler (1736) kara parçalarını **düğüm**, köprüleri **kenar** yaptı: şekil değil, **bağlantı** önemlidir (notasyon/çizim farkı).

Çizge kuramının başlangıcı sayılır. Euler (*Oyler* okunur) sonucu teorem olarak verdi; ispatı **Çizge Kuramı** derslerine kalıyor (4. sınıf: "Çizge Kuramı ve Kombinatorik").

GİRİŞ

Problem: Königsberg'in 7 Köprüsü



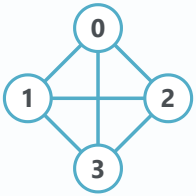
Böyle bir yürüyüş ancak **her düğümün derecesi çiftse** vardır. Burada tüm dereceler tek (5, 3, 3, 3) → **yürüyüş yok**.

Adım 3 / 3

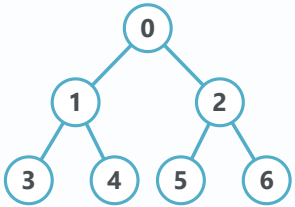
Çizge kuramının başlangıcı sayılır. Euler (*Oyler* okunur) sonucu teorem olarak verdi; ispatı **Çizge Kuramı** derslerine kalıyor (4. sınıf: "Çizge Kuramı ve Kombinatorik").

Tanım: G = (V, E)

G çizgesi **iki sonlu kümeden** oluşur: **boş olmayan** V düğümler (*vertex, node*), boş olabilen E kenarlar (*edge, arc*). Yönsüzde (v0, v1) ≡ (v1, v0); yönlüde (*digraph*) <v0, v1> ≠ <v1, v0> — <kuyruk (*tail*), baş (*head*)>: ok kuyruktan çıkar, başa saplanır.



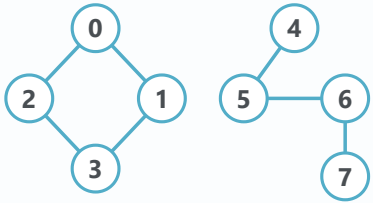
G1: bağlantılı (K4)



G2: ağaç, döngü yok



G3: yönlü



G4: iki bileşen

Çizge	V(G)	E(G)
G1	{0, 1, 2, 3}	{(0,1), (0,2), (0,3), (1,2), (1,3), (2,3)}
G2	{0, …, 6}	{(0,1), (0,2), (1,3), (1,4), (2,5), (2,6)}
G3	{0, 1, 2}	{<0,1>, <1,0>, <1,2>} — yönlüde ikisi de yazılır

E, V × V kartezyen çarpımının bir **alt kümesidir** (tüm ikilileri barındırmaz).

Çizge Veri Yapısı Nerelerde Kullanılır?

Mühendislik
Elektrik devrelerinin analizi, proje planlama (aktivite ağları), kimyasal bileşenleri tanıma.

Yol ve ağ
En kısa yol, yönlendirme (router), bilgisayar ağlarıyla ilgili birçok problem.

Ağaç ve sıralama
Ağaçlar birer çizgedir; topolojik sıralama, sonlu özdevinirler (*finite automata*).

Akış
Taşıma ağlarında akış; **max flow**: kaynaktan hedefe en çok petrol/su taşıma.

Kapsayan ağaç
Minimum kapsayan ağaç: köyleri en az kabloyla bağlamak.

Bağlılık
Bağlantılı çizge, bileşen sayısı, bir ağda kopukluk var mı?

Hocadan: "Nerelerde kullanılmaz ki! Çizgelerin çok geniş bir alanda kullanımı söz konusudur."

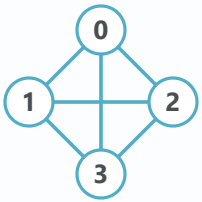
Terminoloji: Komşu Düğüm, Alt Çizge

Komşu (adjacent)

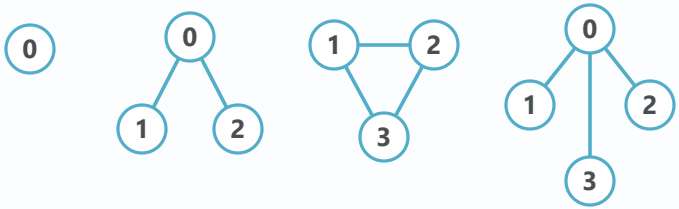
(v0, v1) yönsüz çizgede bir kenarsa v0 ile v1 **komşudur**; kenar bu iki düğüme *bitişiktir*.

Alt çizge (subgraph)

G′, G′nin alt çizgesi ise V(G′) ⊆ V(G) ve E(G′) ⊆ E(G) olur. Tek düğüm de bir alt çizgedir.



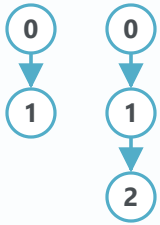
G1



G1'in olası alt çizgelerinden bazıları



G3



G3'ün alt çizgeleri

Yol, Bağlantılılık ve Derece

- **Yol** (*path*): v_p'den v_q'ya (v_p, v_{i1}), (v_{i1}, v_{i2}), …, (v_{in}, v_q) kenar olacak biçimde bir düğüm dizisi.
- v0'dan v1'e yol varsa ikisi **bağlantılıdır** (*connected*); tüm farklı (v_i, v_j) çiftleri arasında yol varsa **G bağlantılıdır**.
- **Derece**: düğümü uç kabul eden kenar sayısı. G2'de d(0) = 2, G1'de d(1) = 3, G2'de d(4) = 1.
- Yönlüde derece = **giriş** (*in-degree*) + **çıkış** (*out-degree*). G3'te d(0) = 1 + 1 = 2; d(1) = 1 + 2 = 3; d(2) = 1 + 0 = 1.



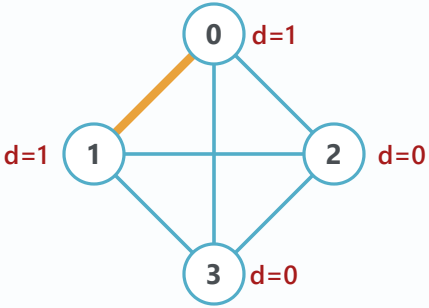
giriş: 1
çıkış: 1

G3'te düğüm 0: d = 1 + 1 = 2

"G3 çizgesinde d = 1 + 1 = 2" ifadesi **düğüm 0** içindir: derece her zaman belirli bir düğüm için söylenir.

Teorem 1: Derecelerin Toplamı = 2e

n düğüm ve e kenarlı G çizgesinde $\mathbf{2e = d_0 + d_1 + \ldots + d_{n-1}}$. **İspat:** her kenar, iki ucundaki düğümlerin derece toplamına tam **2** katkı yapar. Basit olan ve olmayan (çoklu kenarlı) çizgelerde geçerlidir. **Basit çizge:** iki düğüm arasında en fazla bir kenar, öz-döngü (*self-loop*) yok.



işlenen kenar: e = 1

Σ d = 1 + 1 + 0 + 0 = 2

2e = 2

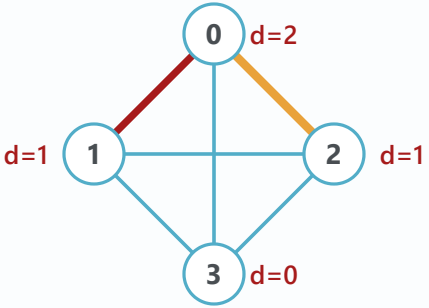
1. kenar eklendi: iki ucun derecesi 1'er arttı → toplam 2 arttı.

Adım 1 / 6

Hocadan: "'Para kazanacağın kitabı oku!' diyor hoca; 'Yok, almayacağım!' deniyor. Mesleğe ilişkin kitap okuma alışkanlığı bu yaşlarda kazanılmalı."

Teorem 1: Derecelerin Toplamı = 2e

n düğüm ve e kenarlı G çizgesinde **$2e = d_0 + d_1 + \ldots + d_{n-1}$**. **İspat:** her kenar, iki ucundaki düğümlerin derece toplamına tam **2** katkı yapar. Basit olan ve olmayan (çoklu kenarlı) çizgelerde geçerlidir. **Basit çizge:** iki düğüm arasında en fazla bir kenar, öz-döngü (*self-loop*) yok.



işlenen kenar: e = 2

Σ d = 2 + 1 + 1 + 0 = 4

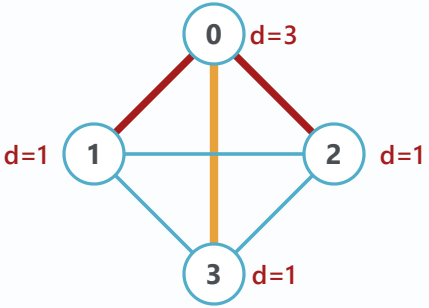
2e = 4

2. kenar eklendi: iki ucun derecesi 1'er arttı → toplam 2 arttı.

Hocadan: "'Para kazanacağın kitabı oku!' diyor hoca; 'Yok, almayacağım!' deniyor. Mesleğe ilişkin kitap okuma alışkanlığı bu yaşlarda kazanılmalı."

Teorem 1: Derecelerin Toplamı = 2e

n düğüm ve e kenarlı G çizgesinde **$2e = d_0 + d_1 + \ldots + d_{n-1}$**. **İspat:** her kenar, iki ucundaki düğümlerin derece toplamına tam **2** katkı yapar. Basit olan ve olmayan (çoklu kenarlı) çizgelerde geçerlidir. **Basit çizge:** iki düğüm arasında en fazla bir kenar, öz-döngü (*self-loop*) yok.



işlenen kenar: e = 3

Σ d = 3 + 1 + 1 + 1 = 6

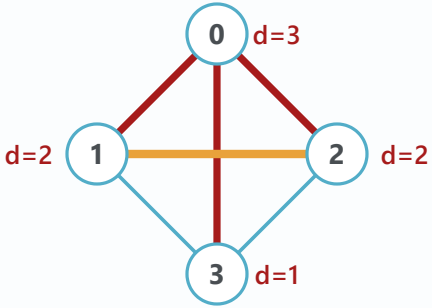
2e = 6

3. kenar eklendi: iki ucun derecesi 1'er arttı → toplam 2 arttı.

Hocadan: "'Para kazanacağın kitabı oku!' diyor hoca; 'Yok, almayacağım!' deniyor. Mesleğe ilişkin kitap okuma alışkanlığı bu yaşlarda kazanılmalı."

Teorem 1: Derecelerin Toplamı = 2e

n düğüm ve e kenarlı G çizgesinde **$2e = d_0 + d_1 + \ldots + d_{n-1}$**. **İspat:** her kenar, iki ucundaki düğümlerin derece toplamına tam **2** katkı yapar. Basit olan ve olmayan (çoklu kenarlı) çizgelerde geçerlidir. **Basit çizge:** iki düğüm arasında en fazla bir kenar, öz-döngü (*self-loop*) yok.



işlenen kenar: e = 4

Σ d = 3 + 2 + 2 + 1 = 8

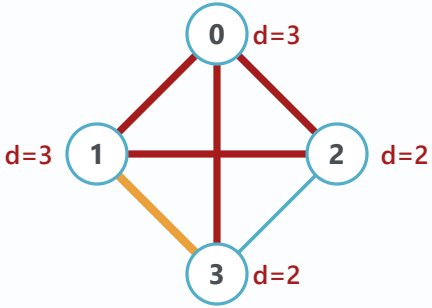
2e = 8

4. kenar eklendi: iki ucun derecesi 1'er arttı → toplam 2 arttı.

Hocadan: "'Para kazanacağın kitabı oku!' diyor hoca; 'Yok, almayacağım!' deniyor. Mesleğe ilişkin kitap okuma alışkanlığı bu yaşlarda kazanılmalı."

Teorem 1: Derecelerin Toplamı = 2e

n düğüm ve e kenarlı G çizgesinde **$2e = d_0 + d_1 + \ldots + d_{n-1}$**. **İspat:** her kenar, iki ucundaki düğümlerin derece toplamına tam **2** katkı yapar. Basit olan ve olmayan (çoklu kenarlı) çizgelerde geçerlidir. **Basit çizge:** iki düğüm arasında en fazla bir kenar, öz-döngü (*self-loop*) yok.



işlenen kenar: e = 5

Σ d = 3 + 3 + 2 + 2 = 10

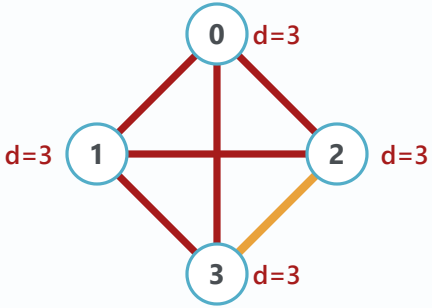
2e = 10

5. kenar eklendi: iki ucun derecesi 1'er arttı → toplam 2 arttı.

Hocadan: "'Para kazanacağın kitabı oku!' diyor hoca; 'Yok, almayacağım!' deniyor. Mesleğe ilişkin kitap okuma alışkanlığı bu yaşlarda kazanılmalı."

Teorem 1: Derecelerin Toplamı = 2e

n düğüm ve e kenarlı G çizgesinde **$2e = d_0 + d_1 + \ldots + d_{n-1}$**. **İspat:** her kenar, iki ucundaki düğümlerin derece toplamına tam **2** katkı yapar. Basit olan ve olmayan (çoklu kenarlı) çizgelerde geçerlidir. **Basit çizge:** iki düğüm arasında en fazla bir kenar, öz-döngü (*self-loop*) yok.



işlenen kenar: e = 6

Σ d = 3 + 3 + 3 + 3 = 12

2e = 12

6. kenar eklendi: iki ucun derecesi 1'er arttı → toplam 2 arttı.

Adım 6 / 6

Hocadan: "'Para kazanacağın kitabı oku!' diyor hoca; 'Yok, almayacağım!' deniyor. Mesleğe ilişkin kitap okuma alışkanlığı bu yaşlarda kazanılmalı."

GERÇEKLEMELER

Çizgeyi Bellekte Nasıl Gerçekleriz?

1. **Komşuluk matrisi**
2. **Komşuluk listesi (+ tersi)**
3. **Dizi ile sıralı gösterim**
4. **Ortogonal liste (yönlü)**
5. **Çok bağlı liste**
+ **Ağırlıklı gösterim**

Hocadan: “Önce kavramı anlatıyoruz, sonra bilgisayar ortamında gerçeklenmesini… Burada amacımız discrete mathematics anlatmak değil. ‘Çizgeyi bellekte nasıl gerçekleriz?’i anlatmak.”

GERÇEKLEMELER

1. Komşuluk Matrisi (Adjacency Matrix)

n düğümlü G için n × n matris: (i, j) ∈ E ise `a[i][j] = 1`. Yönsüzde matris **simetriktir**; yer kazanmak için alt ya da üst yarı üçgen saklanabilir — yine de bellek **O(n²)**.

G1 — (0,2) ∈ E(G1): a[0][2] = a[2][0] = 1

	0	**1**	**2**	**3**
0	0	1	**1**	1
1	1	0	1	1
2	**1**	1	0	1
3	1	1	1	0

G3 — <0,1> ve <1,2> işaretli; simetrik değil

	0	**1**	**2**
0	0	**1**	0
1	1	0	**1**
2	0	0	0

Derece bulma

Satırdaki (ya da sütundaki) 1'ler sayılır: **O(n)**. Yönlüde satır → out-degree, sütun → in-degree; toplamı O(2n). G3'te out-degree d(2) = 0, in-degree d(2) = 1.

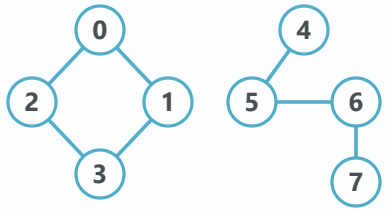
Kenar sayısı bulma

Tüm 1'ler sayılıp 2'ye bölünür: **O(n²)**. Simetri yüzünden yarısına bakmak yeter; yine de $O(n^2/2) = O(n^2)$.

G4'ün Komşuluk Matrisi — Simetri Denetimi

G4; kırmızı çerçeve: sık karıştırılan hücreler

	0	1	2	3	4	5	6	7
0	0	1	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0
2	1	0	0	1	0	0	0	0
3	0	1	1	0	0	0	0	0
4	0	0	0	0	0	1	0	0
5	0	0	0	0	1	0	1	0
6	0	0	0	0	0	1	0	1
7	0	0	0	0	0	0	1	0



G4: iki bileşen

Sık yapılan hata: 1. satırı `1 0 1 0 …`, 7. satırı `… 0 0 1` yazmak. Yönsüz çizgenin matrisi simetrik olmalıdır; bu satırlar simetriyi bozar. Doğrusu: 1. satır `1 0 0 1 0 0 0 0` (komşular 0, 3), 7. satır `0 0 0 0 0 0 1 0` (komşu 6).

Bileşen sayısı matristen de bulunur: her DFS komşuları O(n)'de taradığı için toplam $O(n^2)$.

Kod İzleme: `derece(a, 3, 1)`



	0	1	2
0	0	1	0
1	1	0	1
2	0	0	0

a[][] (G3)

`derece(a, 3, 1)` : G3'te düğüm 1'in derecesi aranıyor.

Adım 1 / 9

```
▶ int derece(int a[][MAX_DUGUM], int n, int v)
2 {
3     int j, cikis = 0, giris = 0;
4     for (j = 0; j < n; j++)
5         cikis += a[v][j];      /* satir  */
6     for (j = 0; j < n; j++)
7         giris += a[j][v];      /* sutun */
8     return cikis + giris;
9 }
```

DEĞİŞKENLER

`v = 1` `n = 3`

ÇAĞRI YIĞINI

`derece(a, 3, 1)`

`main()`

Kod İzleme: `derece(a, 3, 1)`



	0	1	2
0	0	1	0
1	1	0	1
2	0	0	0

a[][] (G3)

Sayaçlar sıfırlanır.

Adım 2 / 9

```
1 int derece(int a[][MAX_DUGUM], int n, int v)
2 {
▶       int j, cikis = 0, giris = 0;
4       for (j = 0; j < n; j++)
5           cikis += a[v][j];       /* satir  */
6       for (j = 0; j < n; j++)
7           giris += a[j][v];       /* sutun */
8       return cikis + giris;
9 }
```

DEĞİŞKENLER

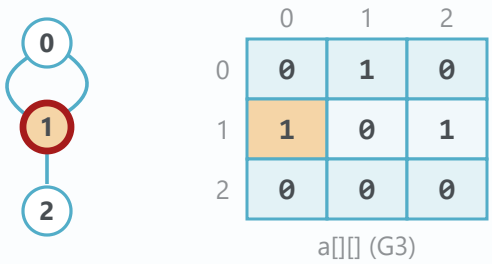
`v = 1` `n = 3` `cikis = 0` `giris = 0`

ÇAĞRI YIĞINI

`derece(a, 3, 1)`

`main()`

Kod İzleme: `derece(a, 3, 1)`



Satır 1, sütun 0: a[1][0] = 1 → cikis = 1.

Adım 3 / 9

```
1 int derece(int a[][MAX_DUGUM], int n, int v)
2 {
3     int j, cikis = 0, giris = 0;
▶     for (j = 0; j < n; j++)
▶         cikis += a[v][j];      /* satir  */
6     for (j = 0; j < n; j++)
7         giris += a[j][v];      /* sutun */
8     return cikis + giris;
9 }
```

DEĞİŞKENLER

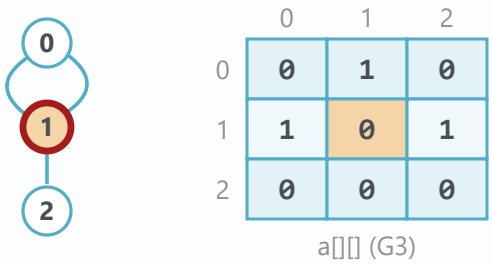
`v = 1` `n = 3` `j = 0` `cikis = 1` `giris = 0`

ÇAĞRI YIĞINI

`derece(a, 3, 1)`

`main()`

Kod İzleme: `derece(a, 3, 1)`



Satır 1, sütun 1: a[1][1] = 0 → cikis = 1.

Adım 4 / 9

```
1 int derece(int a[][MAX_DUGUM], int n, int v)
2 {
3     int j, cikis = 0, giris = 0;
▶     for (j = 0; j < n; j++)
▶         cikis += a[v][j];      /* satir  */
6     for (j = 0; j < n; j++)
7         giris += a[j][v];      /* sutun */
8     return cikis + giris;
9 }
```

DEĞİŞKENLER

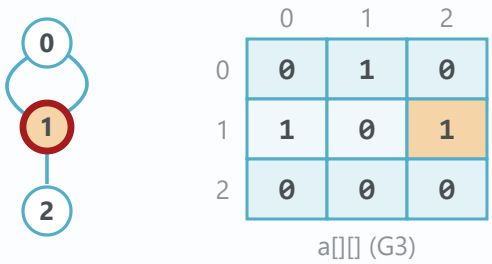
`v = 1` `n = 3` `j = 1` `cikis = 1` `giris = 0`

ÇAĞRI YIĞINI

`derece(a, 3, 1)`

`main()`

Kod İzleme: `derece(a, 3, 1)`



Satır 1, sütun 2: a[1][2] = 1 → cikis = 2.

Adım 5 / 9

```
1 int derece(int a[][MAX_DUGUM], int n, int v)
2 {
3     int j, cikis = 0, giris = 0;
▶     for (j = 0; j < n; j++)
▶         cikis += a[v][j];      /* satir  */
6     for (j = 0; j < n; j++)
7         giris += a[j][v];      /* sutun */
8     return cikis + giris;
9 }
```

DEĞİŞKENLER

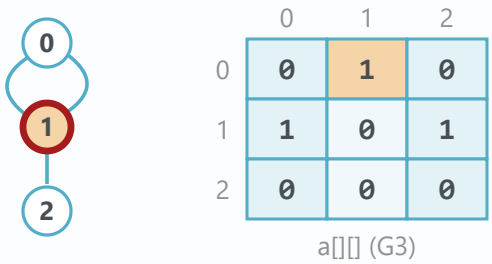
`v = 1` `n = 3` `j = 2` `cikis = 2` `giris = 0`

ÇAĞRI YIĞINI

`derece(a, 3, 1)`

`main()`

Kod İzleme: `derece(a, 3, 1)`



Sütun 1, satır 0: a[0][1] = 1 → giris = 1.

Adım 6 / 9

```
1 int derece(int a[][MAX_DUGUM], int n, int v)
2 {
3     int j, cikis = 0, giris = 0;
4     for (j = 0; j < n; j++)
5         cikis += a[v][j];      /* satir  */
▶     for (j = 0; j < n; j++)
▶         giris += a[j][v];      /* sutun */
8     return cikis + giris;
9 }
```

DEĞİŞKENLER

`v = 1` `n = 3` `j = 0` `cikis = 2` `giris = 1`

ÇAĞRI YIĞINI

`derece(a, 3, 1)`

`main()`

Kod İzleme: `derece(a, 3, 1)`



	0	1	2
0	0	1	0
1	1	0	1
2	0	0	0

a[][] (G3)

Sütun 1, satır 1: a[1][1] = 0 → giris = 1.

Adım 7 / 9

```
1 int derece(int a[][MAX_DUGUM], int n, int v)
2 {
3     int j, cikis = 0, giris = 0;
4     for (j = 0; j < n; j++)
5         cikis += a[v][j];      /* satir  */
▶     for (j = 0; j < n; j++)
▶         giris += a[j][v];      /* sutun */
8     return cikis + giris;
9 }
```

DEĞİŞKENLER

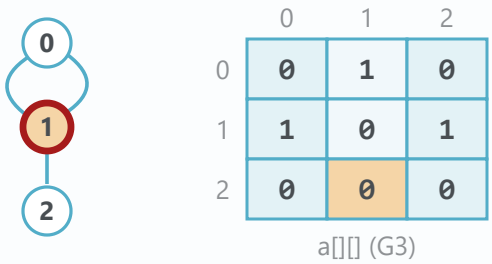
`v = 1` `n = 3` `j = 1` `cikis = 2` `giris = 1`

ÇAĞRI YIĞINI

`derece(a, 3, 1)`

`main()`

Kod İzleme: `derece(a, 3, 1)`



Sütun 1, satır 2: a[2][1] = 0 → giris = 1.

Adım 8 / 9

```
1 int derece(int a[][MAX_DUGUM], int n, int v)
2 {
3     int j, cikis = 0, giris = 0;
4     for (j = 0; j < n; j++)
5         cikis += a[v][j];      /* satir  */
▶     for (j = 0; j < n; j++)
▶         giris += a[j][v];      /* sutun */
8     return cikis + giris;
9 }
```

DEĞİŞKENLER

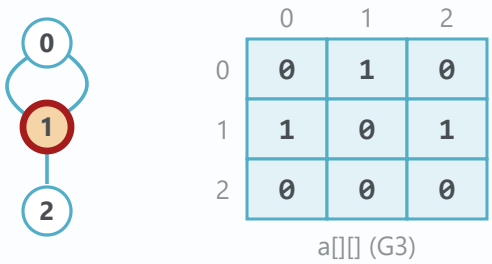
`v = 1` `n = 3` `j = 2` `cikis = 2` `giris = 1`

ÇAĞRI YIĞINI

`derece(a, 3, 1)`

`main()`

Kod İzleme: `derece(a, 3, 1)`



return 2 + 1 = 3. İki döngü de n kez döner → **O(2n) = O(n)**.

Adım 9 / 9

```
1 int derece(int a[][MAX_DUGUM], int n, int v)
2 {
3     int j, cikis = 0, giris = 0;
4     for (j = 0; j < n; j++)
5         cikis += a[v][j];      /* satir  */
6     for (j = 0; j < n; j++)
7         giris += a[j][v];      /* sutun */
▶     return cikis + giris;
9 }
```

DEĞİŞKENLER

`v = 1` `n = 3` `cikis = 2` `giris = 1`

ÇAĞRI YIĞINI

`derece(a, 3, 1)`

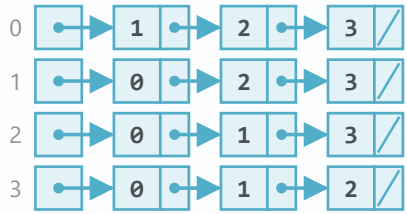
`main()`

GERÇEKLEMELER

2. Komşuluk Listesi (Adjacency List)

Matrisin her satırı yerine bir **bağlı liste**: `baslik[i]`, i'nin komşularının listesini gösterir. Dezavantaj: yönsüzde her kenar için **2 öğe** kullanılır.

```
#define MAX_DUGUM 50    /* en fazla dugum sayisi */
typedef struct oge *oge_gost;
typedef struct oge {
    int dugum;          /* komsu dugumun numarasi */
    oge_gost bag;       /* listedeki sonraki komsu */
} oge;                  /* ';' ve typedef adi unutulmamali */
oge_gost baslik[MAX_DUGUM];   /* her dugume bir liste */
```



G1'in komşuluk listesiyle gösterimi

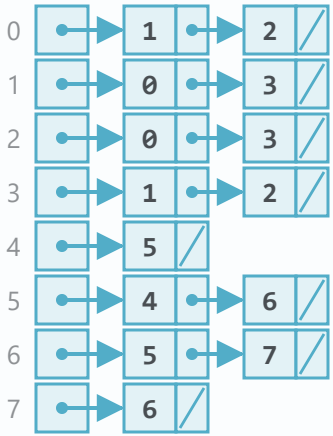
Kenar sayısı

Tüm öğeler sayılıp 2'ye bölünür (Teorem 1): **O(n + e)**. Çizge kuramında karmaşıklık genelde e'ye bağlı verilir.

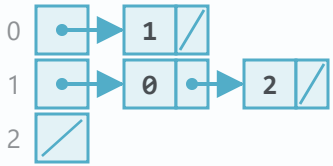
Derece

Yönsüzde i. başlıktaki öğeleri saymak yeter: $O(d_i)$.

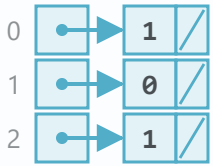
Komşuluk Listesi: Yönsüz ve Yönlü Örnekler



G4'ün komşuluk listesi



G3 (çıkan kenarlar)



G3'ün tersi (gelen kenarlar)



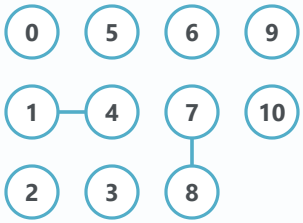
Yönsüz (G4): sıralılık avantajı

Liste sıralıysa her başlıkta kuyruğu baştan küçük (ör. (1,0)) öğeleri sayıp kalanını atlayabiliriz; düğümlerin yarısında geziniriz ama yine **O(n + e)**.

Yönlü (G3)

<0,1> ≠ <1,0> → tüm kenarlar sayılmalı. Out-degree listeden kolay; in-degree için tüm listelerde gezinilir. Çözüm: listenin **tersi (transpozu)** de tutulur; bir kez gezinerek kurulur.

Seyrek Bir Çizge: 11 Düğüm, 2 Kenar



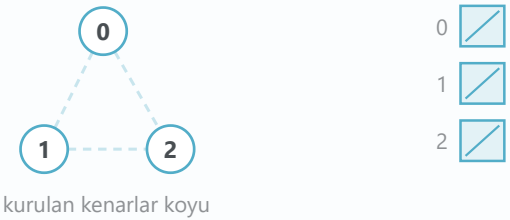
G5

Listeler: baslik[1] → 4, baslik[4] → 1, baslik[7] → 8, baslik[8] → 7. Diğer 7 başlık **NULL**.

Toplam: 11 başlık + 2e = 4 öğe. Matris: 11 × 11 = 121 hücre.

11 düğüm, 2 kenar, **9 bileşen**. Matris 121 hücrenin 117'sini boşa harcar; liste yalnızca 11 başlık + 4 öğe tutar. Seyrek çizgede liste, yoğun çizgede matris düşünülür — ama karar her zaman **yapılacak işlemlere** göre verilir.

Kod İzleme: kenarları yapıya eklemek



Kenar (0,1) → önce `kenar_ekle(0, 1)` : yeni öğe için bellek alınır (NULL kontrolü!).

Adım 1 / 12

```
1 void kenar_ekle(int u, int v)   /* u -> v */
2 {
▶     oge_gost p = malloc(sizeof(oge));
4     if (p == NULL) exit(1);       /* bellek yok */
5     p->dugum = v;
6     p->bag = baslik[u];   /* basa ekle: O(1) */
7     baslik[u] = p;
8 }
```

DEĞİŞKENLER

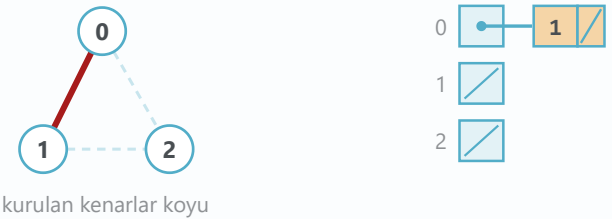
`u = 0` `v = 1` `p = &oge(?)` `baslik[u] = NULL`

ÇAĞRI YIĞINI

`kenar_ekle(0, 1)`

`main()`

Kod İzleme: kenarları yapıya eklemek



kurulan kenarlar koyu

`p→dugum = 1` , `p→bag = NULL` , `baslik[0] = p` : öğe listenin **başına** girdi.

Adım 2 / 12

```
1 void kenar_ekle(int u, int v)   /* u -> v */
2 {
3     oge_gost p = malloc(sizeof(oge));
4     if (p == NULL) exit(1);       /* bellek yok */
▶     p->dugum = v;
▶     p->bag = baslik[u];   /* basa ekle: O(1) */
▶     baslik[u] = p;
8 }
```

DEĞİŞKENLER

`u = 0` `v = 1` `p = &oge(1)`

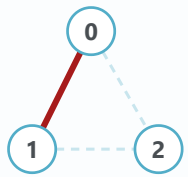
`baslik[u] = &oge(1)`

ÇAĞRI YIĞINI

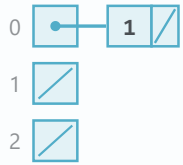
`kenar_ekle(0, 1)`

`main()`

Kod İzleme: kenarları yapıya eklemek



kurulan kenarlar koyu



Kenar (0,1) → sonra `kenar_ekle(1, 0)`: yeni öğe için bellek alınır (NULL kontrolü!).

Adım 3 / 12

```
1 void kenar_ekle(int u, int v)   /* u -> v */
2 {
▶     oge_gost p = malloc(sizeof(oge));
4     if (p == NULL) exit(1);       /* bellek yok */
5     p->dugum = v;
6     p->bag = baslik[u];   /* basa ekle: O(1) */
7     baslik[u] = p;
8 }
```

DEĞİŞKENLER

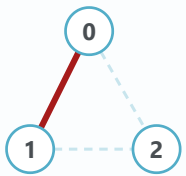
`u = 1` `v = 0` `p = &oge(?)` `baslik[u] = NULL`

ÇAĞRI YIĞINI

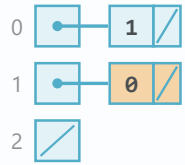
`kenar_ekle(1, 0)`

`main()`

Kod İzleme: kenarları yapıya eklemek



kurulan kenarlar koyu



`p→dugum = 0` , `p→bag = NULL` , `baslik[1] = p` : öğe listenin **başına** girdi.

Adım 4 / 12

```
1 void kenar_ekle(int u, int v)   /* u -> v */
2 {
3     oge_gost p = malloc(sizeof(oge));
4     if (p == NULL) exit(1);       /* bellek yok */
▶     p->dugum = v;
▶     p->bag = baslik[u];   /* basa ekle: O(1) */
▶     baslik[u] = p;
8 }
```

DEĞİŞKENLER

`u = 1` `v = 0` `p = &oge(0)`

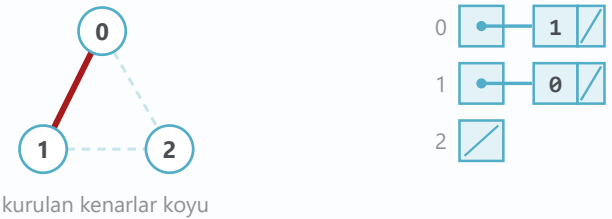
`baslik[u] = &oge(0)`

ÇAĞRI YIĞINI

`kenar_ekle(1, 0)`

`main()`

Kod İzleme: kenarları yapıya eklemek



kurulan kenarlar koyu

Kenar (0,2) → önce `kenar_ekle(0, 2)` : yeni öğe için bellek alınır (NULL kontrolü!).

Adım 5 / 12

```
1 void kenar_ekle(int u, int v)   /* u -> v */
2 {
▶     oge_gost p = malloc(sizeof(oge));
4     if (p == NULL) exit(1);       /* bellek yok */
5     p->dugum = v;
6     p->bag = baslik[u];   /* basa ekle: O(1) */
7     baslik[u] = p;
8 }
```

DEĞİŞKENLER

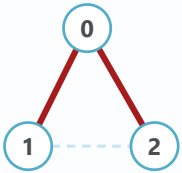
`u = 0` `v = 2` `p = &oge(?)` `baslik[u] = &oge(1)`

ÇAĞRI YIĞINI

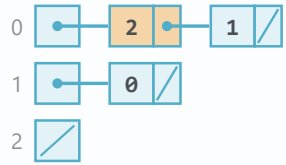
`kenar_ekle(0, 2)`

`main()`

Kod İzleme: kenarları yapıya eklemek



kurulan kenarlar koyu



`p→dugum = 2` , `p→bag = &oge(1)` , `baslik[0] = p` : öğe listenin **başına** girdi.

Adım 6 / 12

```
1 void kenar_ekle(int u, int v)   /* u -> v */
2 {
3     oge_gost p = malloc(sizeof(oge));
4     if (p == NULL) exit(1);       /* bellek yok */
▶     p->dugum = v;
▶     p->bag = baslik[u];   /* basa ekle: O(1) */
▶     baslik[u] = p;
8 }
```

DEĞİŞKENLER

`u = 0` `v = 2` `p = &oge(2)`

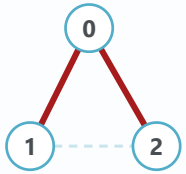
`baslik[u] = &oge(2)`

ÇAĞRI YIĞINI

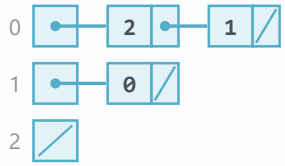
`kenar_ekle(0, 2)`

`main()`

Kod İzleme: kenarları yapıya eklemek



kurulan kenarlar koyu



Kenar (0,2) → sonra `kenar_ekle(2, 0)`: yeni öğe için bellek alınır (NULL kontrolü!).

Adım 7 / 12

```
1 void kenar_ekle(int u, int v)   /* u -> v */
2 {
▶     oge_gost p = malloc(sizeof(oge));
4     if (p == NULL) exit(1);       /* bellek yok */
5     p->dugum = v;
6     p->bag = baslik[u];   /* basa ekle: O(1) */
7     baslik[u] = p;
8 }
```

DEĞİŞKENLER

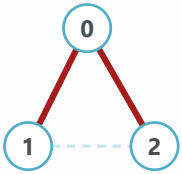
`u = 2` `v = 0` `p = &oge(?)` `baslik[u] = NULL`

ÇAĞRI YIĞINI

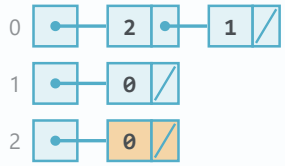
`kenar_ekle(2, 0)`

`main()`

Kod İzleme: kenarları yapıya eklemek



kurulan kenarlar koyu



`p→dugum = 0` , `p→bag = NULL` , `baslik[2] = p` : öğe listenin **başına** girdi.

Adım 8 / 12

```
1 void kenar_ekle(int u, int v)   /* u -> v */
2 {
3     oge_gost p = malloc(sizeof(oge));
4     if (p == NULL) exit(1);       /* bellek yok */
▶     p->dugum = v;
▶     p->bag = baslik[u];   /* basa ekle: O(1) */
▶     baslik[u] = p;
8 }
```

DEĞİŞKENLER

`u = 2` `v = 0` `p = &oge(0)`

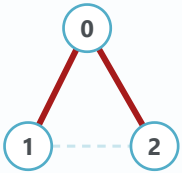
`baslik[u] = &oge(0)`

ÇAĞRI YIĞINI

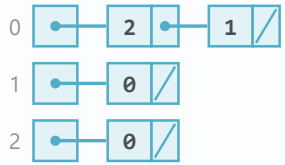
`kenar_ekle(2, 0)`

`main()`

Kod İzleme: kenarları yapıya eklemek



kurulan kenarlar koyu



Kenar (1,2) → önce `kenar_ekle(1, 2)` : yeni öğe için bellek alınır (NULL kontrolü!).

Adım 9 / 12

```
1 void kenar_ekle(int u, int v)   /* u -> v */
2 {
▶     oge_gost p = malloc(sizeof(oge));
4     if (p == NULL) exit(1);       /* bellek yok */
5     p->dugum = v;
6     p->bag = baslik[u];   /* basa ekle: O(1) */
7     baslik[u] = p;
8 }
```

DEĞİŞKENLER

`u = 1` `v = 2` `p = &oge(?)`

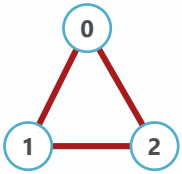
`baslik[u] = &oge(0)`

ÇAĞRI YIĞINI

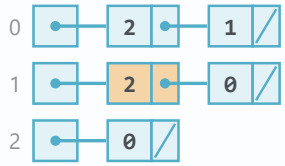
`kenar_ekle(1, 2)`

`main()`

Kod İzleme: kenarları yapıya eklemek



kurulan kenarlar koyu



`p→dugum = 2` , `p→bag = &oge(0)` , `baslik[1] = p` : öğe listenin **başına** girdi.

Adım 10 / 12

```
1 void kenar_ekle(int u, int v)   /* u -> v */
2 {
3     oge_gost p = malloc(sizeof(oge));
4     if (p == NULL) exit(1);       /* bellek yok */
▶     p->dugum = v;
▶     p->bag = baslik[u];   /* basa ekle: O(1) */
▶     baslik[u] = p;
8 }
```

DEĞİŞKENLER

`u = 1` `v = 2` `p = &oge(2)`

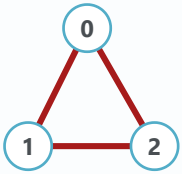
`baslik[u] = &oge(2)`

ÇAĞRI YIĞINI

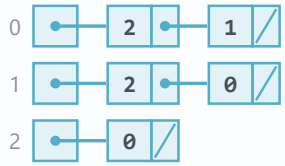
`kenar_ekle(1, 2)`

`main()`

Kod İzleme: kenarları yapıya eklemek



kurulan kenarlar koyu



Kenar (1,2) → sonra `kenar_ekle(2, 1)`: yeni öğe için bellek alınır (NULL kontrolü!).

Adım 11 / 12

```
1 void kenar_ekle(int u, int v)   /* u -> v */
2 {
▶     oge_gost p = malloc(sizeof(oge));
4     if (p == NULL) exit(1);       /* bellek yok */
5     p->dugum = v;
6     p->bag = baslik[u];   /* basa ekle: O(1) */
7     baslik[u] = p;
8 }
```

DEĞİŞKENLER

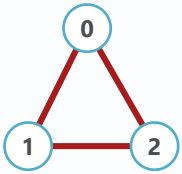
`u = 2` `v = 1` `p = &oge(?)` `baslik[u] = &oge(0)`

ÇAĞRI YIĞINI

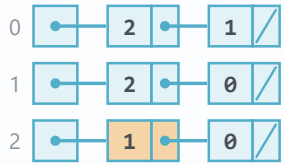
`kenar_ekle(2, 1)`

`main()`

Kod İzleme: kenarları yapıya eklemek



kurulan kenarlar koyu



`p->dugum = 1` , `p->bag = &oge(0)` , `baslik[2] = p` : öğe listenin **başına** girdi.

Adım 12 / 12

```
1 void kenar_ekle(int u, int v)   /* u -> v */
2 {
3     oge_gost p = malloc(sizeof(oge));
4     if (p == NULL) exit(1);       /* bellek yok */
▶     p->dugum = v;
▶     p->bag = baslik[u];   /* basa ekle: O(1) */
▶     baslik[u] = p;
8 }
```

DEĞİŞKENLER

`u = 2` `v = 1` `p = &oge(1)`

`baslik[u] = &oge(1)`

ÇAĞRI YIĞINI

`kenar_ekle(2, 1)`

`main()`

GERÇEKLEMELER

Çizgeyi Kim Kuracak?

> **Hocadan:** "Kenarlar geliyor, yapıya eklenebilmeli. Ben bakıp çizdim; program bakıp çizemez ki! Algoritma çalışacak."

Başa ekleme

Her kenar O(1)'de yerleşir; liste gelen sıranın **tersidir**. Toplam kurulum O(n + e).

Sona ekleme

Sıra korunur ama her seferinde NULL'a kadar yürümek gerekir (O(d)); ya da her liste için bir `son` işaretçisi tutulur.

Sıralı ekleme

Listeyi düğüm numarasına göre sıralı tutmak: kenar sayma ve gezinme sırası öngörülebilir olur, ekleme O(d).

Serbest bırakma

Program sonunda her `baslik[i]` listesinde gezinilerek `free` edilmelidir.

Kod İzleme: `devrik_kur()`



Önce `ters[]` başlıkları NULL yapılır.

Adım 1 / 6

```
1  void devrik_kur(oge_gost bas[], oge_gost ters[],
2                  int n)
3  {
4      int i;
5      oge_gost p;
▶      for (i = 0; i < n; i++) ters[i] = NULL;
7      for (i = 0; i < n; i++)
8          for (p = bas[i]; p; p = p->bag)
9              ekle(ters, p->dugum, i);  /* <j,i> */
10 }
```

DEĞİŞKENLER

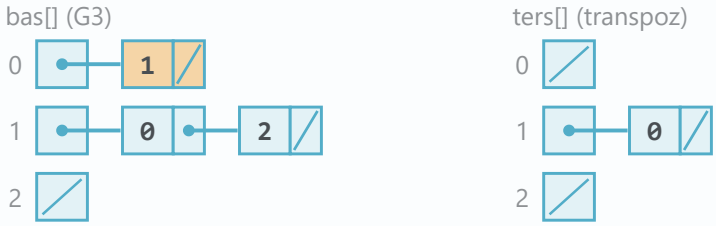
`n = 3` `i = –` `p = NULL`

ÇAĞRI YIĞINI

`devrik_kur(bas, ters, 3)`

`main()`

Kod İzleme: `devrik_kur()`



<0,1> kenarı: `ters[1]` listesine 0 eklenir (<1,0>).

Adım 2 / 6

```
1  void devrik_kur(oge_gost bas[], oge_gost ters[],
2                  int n)
3  {
4      int i;
5      oge_gost p;
6      for (i = 0; i < n; i++) ters[i] = NULL;
7      for (i = 0; i < n; i++)
▶          for (p = bas[i]; p; p = p->bag)
▶              ekle(ters, p->dugum, i);  /* <j,i> */
10 }
```

DEĞİŞKENLER

`n = 3` `i = 0` `p = &oge(1)`

ÇAĞRI YIĞINI

`devrik_kur(bas, ters, 3)`

`main()`

Kod İzleme: `devrik_kur()`



<1,0> kenarı: `ters[0]` listesine 1 eklenir (<0,1>).

Adım 3 / 6

```
1  void devrik_kur(oge_gost bas[], oge_gost ters[],
2                  int n)
3  {
4      int i;
5      oge_gost p;
6      for (i = 0; i < n; i++) ters[i] = NULL;
7      for (i = 0; i < n; i++)
▶          for (p = bas[i]; p; p = p->bag)
▶              ekle(ters, p->dugum, i);  /* <j,i> */
10 }
```

DEĞİŞKENLER

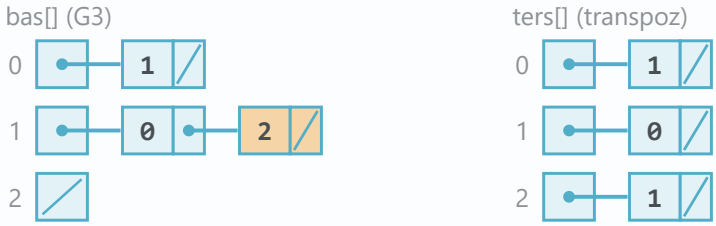
`n = 3` `i = 1` `p = &oge(0)`

ÇAĞRI YIĞINI

`devrik_kur(bas, ters, 3)`

`main()`

Kod İzleme: `devrik_kur()`



<1,2> kenarı: `ters[2]` listesine 1 eklenir (<2,1>).

Adım 4 / 6

```
1  void devrik_kur(oge_gost bas[], oge_gost ters[],
2                  int n)
3  {
4      int i;
5      oge_gost p;
6      for (i = 0; i < n; i++) ters[i] = NULL;
7      for (i = 0; i < n; i++)
▶          for (p = bas[i]; p; p = p->bag)
▶              ekle(ters, p->dugum, i);  /* <j,i> */
10 }
```

DEĞİŞKENLER

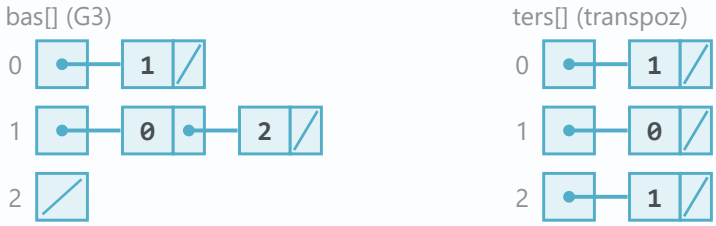
`n = 3` `i = 1` `p = &oge(2)`

ÇAĞRI YIĞINI

`devrik_kur(bas, ters, 3)`

`main()`

Kod İzleme: `devrik_kur()`



`bas[2]` boş: 2'den çıkan kenar yok.

Adım 5 / 6

```
1  void devrik_kur(oge_gost bas[], oge_gost ters[],
2                  int n)
3  {
4      int i;
5      oge_gost p;
6      for (i = 0; i < n; i++) ters[i] = NULL;
▶      for (i = 0; i < n; i++)
▶          for (p = bas[i]; p; p = p->bag)
9              ekle(ters, p->dugum, i);  /* <j,i> */
10 }
```

DEĞİŞKENLER

`n = 3` `i = 2` `p = NULL`

ÇAĞRI YIĞINI

`devrik_kur(bas, ters, 3)`

`main()`

Kod İzleme: `devrik_kur()`



Her öğeye bir kez bakıldı: transpoz **O(n + e)**'de kuruldu. Artık in-degree de O(d)'de bulunur.

Adım 6 / 6

```
1  void devrik_kur(oge_gost bas[], oge_gost ters[],
2                  int n)
3  {
4      int i;
5      oge_gost p;
6      for (i = 0; i < n; i++) ters[i] = NULL;
▶      for (i = 0; i < n; i++)
8          for (p = bas[i]; p; p = p->bag)
9              ekle(ters, p->dugum, i);  /* <j,i> */
10 }
```

DEĞİŞKENLER

`n = 3` `i = 3` `p = NULL`

ÇAĞRI YIĞINI

`devrik_kur(bas, ters, 3)`

`main()`

GERÇEKLEMELER

3. Dizi ile Sıralı (Sequential) Gösterim

Yönsüz, n düğüm e kenar: tek dizi `int dugum[n + 2*e + 1]`. İlk n + 1 hücre başlangıç indisleridir; i'nin komşuları `dugum[dugum[i]] … dugum[dugum[i+1] − 1]` aralığındadır.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
9	11	13	15	17	18	20	22	23	1	2	0	3	0	3	1	2	5	4	6	5	7	6

n + 1 = 9 başlık (düğüm[n] = n + 2e + 1 = 23) | 2e = 14 komşu kaydı

G4 için dizi. Her kenar yine 2 kez yazılır ama **işaretçi yok**.

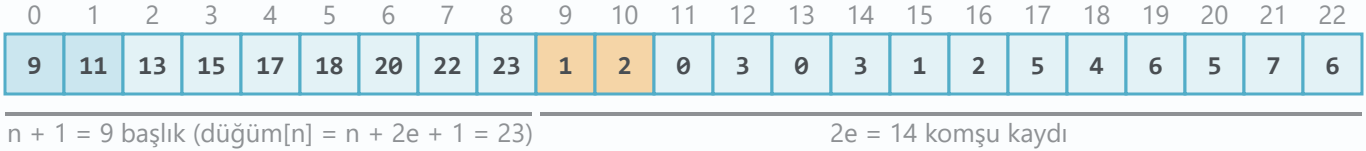
Adım 1 / 4

Hocadan: "Depolamak ayrı konudur, o veri üzerinde çalışacak algoritmaların etkinliği ayrı konudur... Çok sayıda ekleme-silme işlemi için bu yapı verimli değildir. Örneğin bir ağdaki bilgisayarlar sabittir; problem onları tutmaksa bu yapı uygun olabilir."

GERÇEKLEMELER

3. Dizi ile Sıralı (Sequential) Gösterim

Yönsüz, n düğüm e kenar: tek dizi `int dugum[n + 2*e + 1]`. İlk n + 1 hücre başlangıç indisleridir; i'nin komşuları `dugum[dugum[i]] … dugum[dugum[i+1] − 1]` aralığındadır.



0'ın komşuları: dugum[0] = 9 … dugum[1] = 11 (hariç) → 1, 2

Ör. 1: [0] = 9'dan (dahil) [1] = 11'e (hariç): indis 9 ve 10 → komşular **1, 2**.

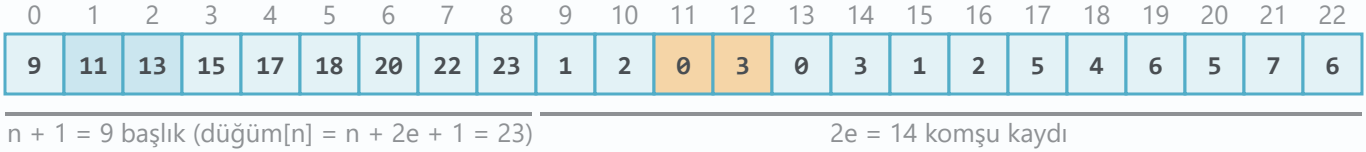
Adım 2 / 4

Hocadan: "Depolamak ayrı konudur, o veri üzerinde çalışacak algoritmaların etkinliği ayrı konudur… Çok sayıda ekleme-silme işlemi için bu yapı verimli değildir. Örneğin bir ağdaki bilgisayarlar sabittir; problem onları tutmaksa bu yapı uygun olabilir."

GERÇEKLEMELER

3. Dizi ile Sıralı (Sequential) Gösterim

Yönsüz, n düğüm e kenar: tek dizi `int dugum[n + 2*e + 1]`. İlk n + 1 hücre başlangıç indisleridir; i'nin komşuları `dugum[dugum[i]] … dugum[dugum[i+1] − 1]` aralığındadır.



1'in komşuları: dugum[1] = 11 … dugum[2] = 13 (hariç) → 0, 3

Ör. 2: [1] = 11, [2] = 13 → indis 11, 12 → komşular **0, 3**.

Adım 3 / 4

Hocadan: "Depolamak ayrı konudur, o veri üzerinde çalışacak algoritmaların etkinliği ayrı konudur... Çok sayıda ekleme-silme işlemi için bu yapı verimli değildir. Örneğin bir ağdaki bilgisayarlar sabittir; problem onları tutmaksa bu yapı uygun olabilir."

3. Dizi ile Sıralı (Sequential) Gösterim

Yönsüz, n düğüm e kenar: tek dizi `int dugum[n + 2*e + 1]`. İlk n + 1 hücre başlangıç indisleridir; i'nin komşuları `dugum[dugum[i]] … dugum[dugum[i+1] − 1]` aralığındadır.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
9	11	13	15	17	18	20	22	23	1	2	0	3	0	3	1	2	5	4	6	5	7	6

n + 1 = 9 başlık (düğüm[n] = n + 2e + 1 = 23) | 2e = 14 komşu kaydı

7'nin komşuları: dugum[7] = 22 … dugum[8] = 23 (hariç) → 6

Ör. 3: [7] = 22, [8] = 23 → yalnızca indis 22 → tek komşu **6**.

Hocadan: "Depolamak ayrı konudur, o veri üzerinde çalışacak algoritmaların etkinliği ayrı konudur... Çok sayıda ekleme-silme işlemi için bu yapı verimli değildir. Örneğin bir ağdaki bilgisayarlar sabittir; problem onları tutmaksa bu yapı uygun olabilir."

SIRALI GÖSTERİM · KODU ADIM ADIM ÇALIŞTIR

Kod İzleme: `komsulari_yaz(dugum, 5)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
9	11	13	15	17	18	20	22	23	1	2	0	3	0	3	1	2	5	4	6	5	7	6

n + 1 = 9 başlık (düğüm[n] = n + 2e + 1 = 23) | 2e = 14 komşu kaydı

`komsulari_yaz(dugum, 5)` : 5 numaralı düğümün komşuları.

Adım 1 / 4

```
▶ void komsulari_yaz(const int dugum[], int i)
2 {
3     int k;
4     for (k = dugum[i]; k < dugum[i + 1]; k++)
5         printf("%d ", dugum[k]);
6     printf("\n");
7 }
```

DEĞİŞKENLER

`i = 5`

ÇAĞRI YIĞINI

`komsulari_yaz(dugum, 5)`

`main()`

ÇIKTI

SIRALI GÖSTERİM · KODU ADIM ADIM ÇALIŞTIR

Kod İzleme: `komsulari_yaz(dugum, 5)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
9	11	13	15	17	18	20	22	23	1	2	0	3	0	3	1	2	5	4	6	5	7	6

k = 18

k = 18 < dugum[6] = 20 → `dugum[18] = 4` yazılır.

Adım 2 / 4

```
1 void komsulari_yaz(const int dugum[], int i)
2 {
3     int k;
      for (k = dugum[i]; k < dugum[i + 1]; k++)
          printf("%d ", dugum[k]);
6     printf("\n");
7 }
```

DEĞİŞKENLER

`i = 5` `k = 18` `dugum[k] = 4`

ÇAĞRI YIĞINI

`komsulari_yaz(dugum, 5)`

`main()`

ÇIKTI

```
4
```

SIRALI GÖSTERİM · KODU ADIM ADIM ÇALIŞTIR

Kod İzleme: `komsulari_yaz(dugum, 5)`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
9	11	13	15	17	18	20	22	23	1	2	0	3	0	3	1	2	5	4	6	5	7	6

k = 19

k = 19 < dugum[6] = 20 → `dugum[19] = 6` yazılır.

Adım 3 / 4

```
1 void komsulari_yaz(const int dugum[], int i)
2 {
3     int k;
▶     for (k = dugum[i]; k < dugum[i + 1]; k++)
▶         printf("%d ", dugum[k]);
6     printf("\n");
7 }
```

DEĞİŞKENLER

`i = 5` `k = 19` `dugum[k] = 6`

ÇAĞRI YIĞINI

`komsulari_yaz(dugum, 5)`

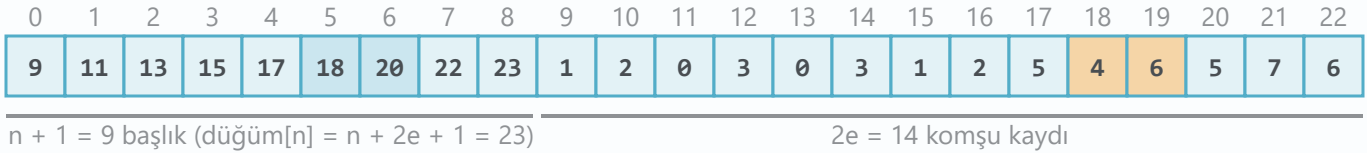
`main()`

ÇIKTI

```
4 6
```

SIRALI GÖSTERİM · KODU ADIM ADIM ÇALIŞTIR

Kod İzleme: `komsulari_yaz(dugum, 5)`



5'in komşuları: dugum[5] = 18 … dugum[6] = 20 (hariç) → 4, 6

k = 20 = dugum[6] → döngü biter. Komşu sayısı = dugum[6] − dugum[5] = derece: **O(d)**.

Adım 4 / 4

```
1 void komsulari_yaz(const int dugum[], int i)
2 {
3     int k;
▶     for (k = dugum[i]; k < dugum[i + 1]; k++)
5         printf("%d ", dugum[k]);
▶     printf("\n");
7 }
```

DEĞİŞKENLER

`i = 5` `k = 20`

ÇAĞRI YIĞINI

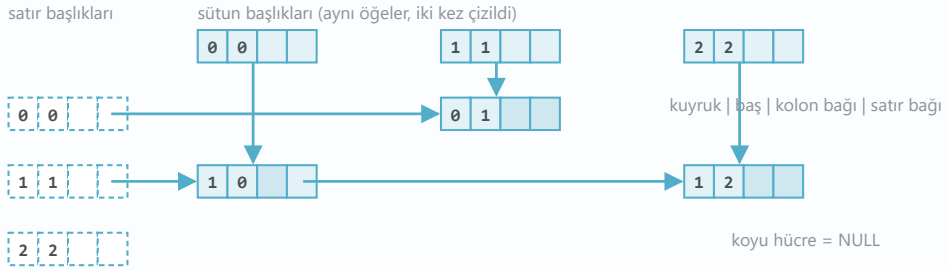
`komsulari_yaz(dugum, 5)`

`main()`

ÇIKTI

```
4 6
```

4. Yönlü Çizgelerin Ortogonal Gösterimi



G3: her kenar öğesi aynı anda bir **satır** (çıkış) ve bir **sütun** (giriş) listesinde

İki bağ

Baş için kolon bağı: bu düğüme nereden gelinir. **Kuyruk için satır bağı:** bu düğümden nereye çıkılır. Aynı kenar için **tek öğe** yeter.

Seyrek matris gibi

Bağlı seyrek matris yapısına benzer; döngülü, çift yönlü bağlı, başlıklı yapılabilir. "2e adet öğe almak rahatsız etti, böylece bu örneği bulduk."

Hocadan: "Akla gelen ilk çözüme de, o koda da hiç güvenilmemeli. Arka plandaki veri yapısı bu değil de başka bir şey olsaydı, daha verimli çözebilir miydim?"

5. Çok Bağlı Liste (Adjacency Multilist)

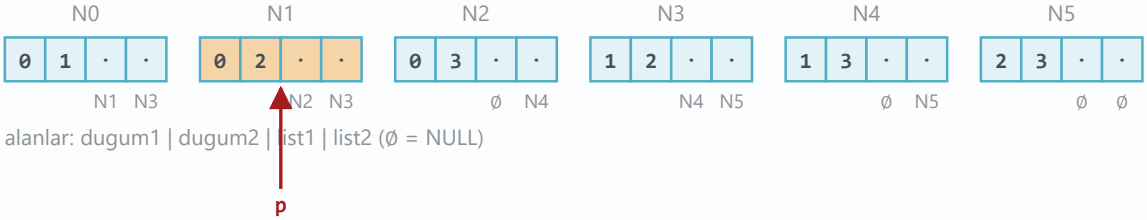
```
typedef struct kenar *kenar_gost;
typedef struct kenar {
    short isaret;         /* islendi mi */
    int dugum1, dugum2;   /* uclar */
    kenar_gost list1;     /* dugum1 sonraki */
    kenar_gost list2;     /* dugum2 sonraki */
} kenar;
kenar_gost baslik[MAX_DUGUM];
```

Kenar **temel öğedir**: (v_i, v_j) için **tek öğe** vardır ama hem v_i'nin hem v_j'nin listesinde yer alır. `isaret`, işlenmiş kenarın tekrar ele alınmasını önler. Düğüme veri taşımak gerekirse öğeye `char veri` gibi bir alan eklenebilir.

Öğe	dugum1	dugum2	list1	list2
N0	0	1	N1	N3
N1	0	2	N2	N3
N2	0	3	NULL	N4
N3	1	2	N4	N5
N4	1	3	NULL	N5
N5	2	3	NULL	NULL

G1 için başlıklar: baslik[0] = N0, baslik[1] = N0, baslik[2] = N1, baslik[3] = N2. Hiç kenarı olmayan düğümün başlığı NULL'dır.

Kod İzleme: 2'nin komşuları üzerinde gezinmek



`p = baslik[2]` = N1: 2'nin ilk kenarı.

Adım 1 / 8

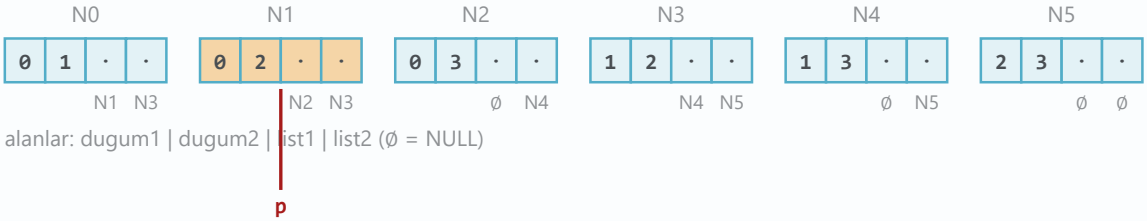
```
1  void komsulari_yaz(int v)
2  {
▶      kenar_gost p = baslik[v];
4      while (p) {
5          if (p->dugum1 == v) {       /* v: dugum1 */
6              printf("%d ", p->dugum2);  p = p->list1;
7          } else {                    /* v: dugum2 */
8              printf("%d ", p->dugum1);  p = p->list2;
9          }
10     }
11 }
```

DEĞİŞKENLER

v = **2** p = **&N1**

ÇIKTI

Kod İzleme: 2'nin komşuları üzerinde gezinmek



N1 = (0,2): v, **dugum2**'de → öteki uç 0 yazılır.

Adım 2 / 8

```
1  void komsulari_yaz(int v)
2  {
3      kenar_gost p = baslik[v];
4      while (p) {
▶          if (p->dugum1 == v) {        /* v: dugum1 */
6              printf("%d ", p->dugum2);  p = p->list1;
7          } else {                     /* v: dugum2 */
▶              printf("%d ", p->dugum1);  p = p->list2;
9          }
10     }
11 }
```

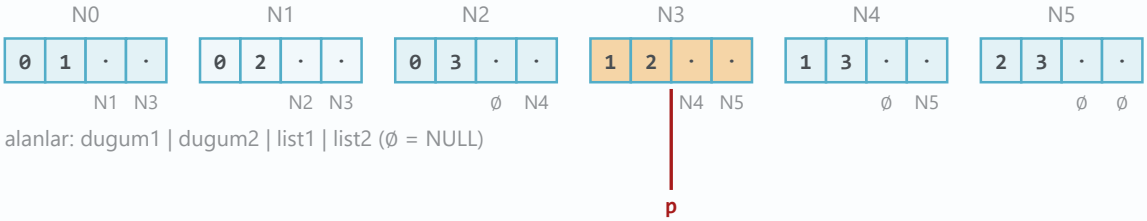
DEĞİŞKENLER

v = **2** | p = **&N1**

ÇIKTI

```
0
```

Kod İzleme: 2'nin komşuları üzerinde gezinmek



Bağ izlenir: `p = p→list2` = N3.

Adım 3 / 8

```
1  void komsulari_yaz(int v)
2  {
3      kenar_gost p = baslik[v];
4      while (p) {
5          if (p->dugum1 == v) {        /* v: dugum1 */
6              printf("%d ", p->dugum2);  p = p->list1;
7          } else {                     /* v: dugum2 */
▶              printf("%d ", p->dugum1);  p = p->list2;
9          }
10     }
11 }
```

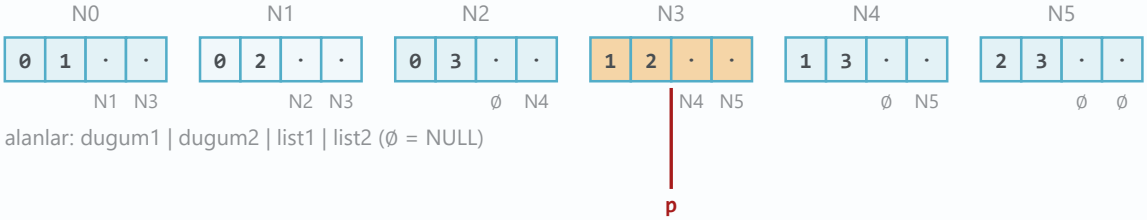
DEĞİŞKENLER

`v = 2` `p = &N3`

ÇIKTI

```
0
```

Kod İzleme: 2'nin komşuları üzerinde gezinmek



N3 = (1,2): v, **dugum2**'de → öteki uç 1 yazılır.

Adım 4 / 8

```
1  void komsulari_yaz(int v)
2  {
3      kenar_gost p = baslik[v];
4      while (p) {
▶          if (p->dugum1 == v) {        /* v: dugum1 */
6              printf("%d ", p->dugum2);  p = p->list1;
7          } else {                     /* v: dugum2 */
▶              printf("%d ", p->dugum1);  p = p->list2;
9          }
10     }
11 }
```

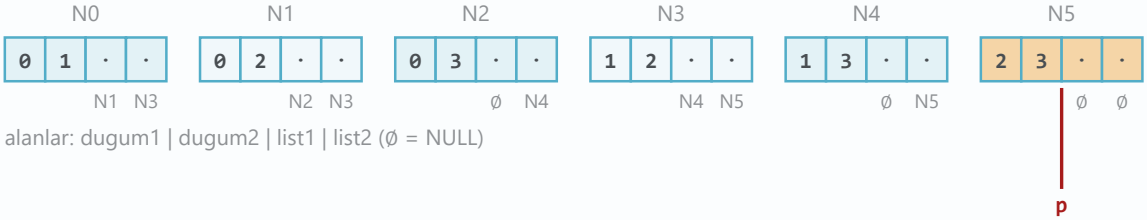
DEĞİŞKENLER

v = **2** p = **&N3**

ÇIKTI

```
0 1
```

Kod İzleme: 2'nin komşuları üzerinde gezinmek



Bağ izlenir: `p = p→list2` = N5.

Adım 5 / 8

```
1  void komsulari_yaz(int v)
2  {
3      kenar_gost p = baslik[v];
4      while (p) {
5          if (p->dugum1 == v) {        /* v: dugum1 */
6              printf("%d ", p->dugum2);  p = p->list1;
7          } else {                     /* v: dugum2 */
▶              printf("%d ", p->dugum1);  p = p->list2;
9          }
10     }
11 }
```

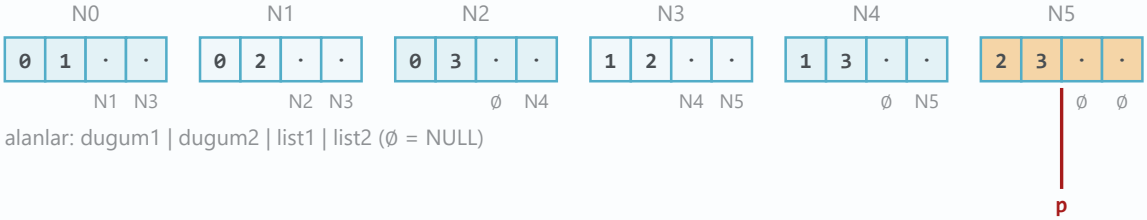
DEĞİŞKENLER

`v = 2` `p = &N5`

ÇIKTI

```
0 1
```

Kod İzleme: 2'nin komşuları üzerinde gezinmek



N5 = (2,3): v, **dugum1**'de → öteki uç 3 yazılır.

Adım 6 / 8

```
1  void komsulari_yaz(int v)
2  {
3      kenar_gost p = baslik[v];
4      while (p) {
▶          if (p->dugum1 == v) {       /* v: dugum1 */
▶              printf("%d ", p->dugum2);  p = p->list1;
7          } else {                    /* v: dugum2 */
8              printf("%d ", p->dugum1);  p = p->list2;
9          }
10     }
11 }
```

DEĞİŞKENLER

v = **2** p = **&N5**

ÇIKTI

```
0 1 3
```

Kod İzleme: 2'nin komşuları üzerinde gezinmek



Bağ izlenir: `p = p→list1` = NULL.

Adım 7 / 8

```
1  void komsulari_yaz(int v)
2  {
3      kenar_gost p = baslik[v];
4      while (p) {
5          if (p->dugum1 == v) {        /* v: dugum1 */
▶              printf("%d ", p->dugum2);  p = p->list1;
7          } else {                     /* v: dugum2 */
8              printf("%d ", p->dugum1);  p = p->list2;
9          }
10     }
11 }
```

DEĞİŞKENLER

`v = 2` `p = NULL`

ÇIKTI

```
0 1 3
```

Kod İzleme: 2'nin komşuları üzerinde gezinmek



NULL: aranan özellikte öğe kalmadı. 2'nin komşuları 0, 1, 3 — her kenar yalnız **bir kez** saklandı.

Adım 8 / 8

```
1  void komsulari_yaz(int v)
2  {
3      kenar_gost p = baslik[v];
▶      while (p) {
5          if (p->dugum1 == v) {        /* v: dugum1 */
6              printf("%d ", p->dugum2);  p = p->list1;
7          } else {                     /* v: dugum2 */
8              printf("%d ", p->dugum1);  p = p->list2;
9          }
10     }
11 }
```

DEĞİŞKENLER

v = **2** | p = **NULL**

ÇIKTI

```
0 1 3
```

GERÇEKLEMELER

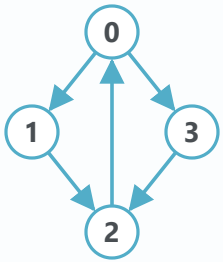
Çok Bağlı Listeye Kenar Eklemek

- **Gezinme:** dugum1'de v varsa list1, dugum2'de v varsa list2 izlenir. NULL görünce biter. "Başka var mı? Bağı takip et!"
- **Eklemek:** ilgili listede NULL'a kadar ilerleyip sona eklemek mümkün; ama **başa eklemek** sona kadar gezinme maliyetinden kurtarır: O(1).
- Yönsüzde bir **baş dizisi** yeter; yönlüde baş dizisiyle birlikte bir **kuyruk dizisi** de tutulur.
- Kazanç: listede 2e öğe yerine yalnızca **e** öğe.

```
void ml_ekle(int u, int v)
{
    kenar_gost p = malloc(sizeof(kenar));
    if (p == NULL) exit(1);
    p->isaret = 0;
    p->dugum1 = u;  p->dugum2 = v;
    p->list1 = baslik[u];  /* u listesine */
    p->list2 = baslik[v];  /* v listesine */
    baslik[u] = baslik[v] = p;  /* ikisine de basa */
}
```

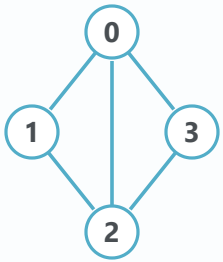
Hocadan: "Bu sırayla gelip "O(1)'de yerleşsin" denilirse liste başına eklenecek! Böyle (liste sonuna ekleyerek) çizene puan vermem. Çünkü liste sonuna eklediğini görüyorum."

Alıştırma: <0,1>, <0,3>, <1,2>, <2,0>, <3,2>



Kenar listesi verilen yönlü çizge için **komşuluk matrisini**, **komşuluk listesini** (ve tersini) ve **çok bağlı listeyi** veriniz.

Alıştırma: <0,1>, <0,3>, <1,2>, <2,0>, <3,2>

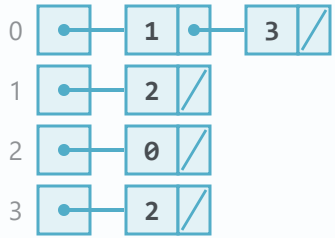


satır = kuyruk, sütun = baş

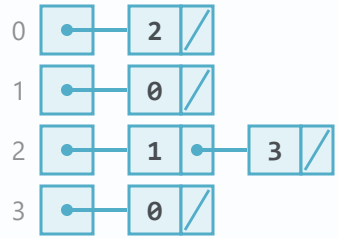
	0	1	2	3
0	0	**1**	0	**1**
1	0	0	**1**	0
2	**1**	0	0	0
3	0	0	**1**	0

Matris: <i,j> varsa a[i][j] = 1. Satır toplamları out-degree (2, 1, 1, 1), sütun toplamları in-degree (1, 1, 2, 1).

Alıştırma: <0,1>, <0,3>, <1,2>, <2,0>, <3,2>



çıkan (komşuluk listesi)



gelen (listenin transpozu)

Liste ve tersi: toplam 2e = 10 öğe.

Alıştırma: <0,1>, <0,3>, <1,2>, <2,0>, <3,2>

Öğe	kuyruk	baş	kuyruk bağı	baş bağı
N1	0	1	N2	NULL
N2	0	3	NULL	NULL
N3	1	2	NULL	N5
N4	2	0	NULL	NULL
N5	3	2	NULL	NULL

Düğüm	kuyruk (çıkış)	baş (giriş)
0	N1	N4
1	N3	N1
2	N4	N3
3	N5	N2

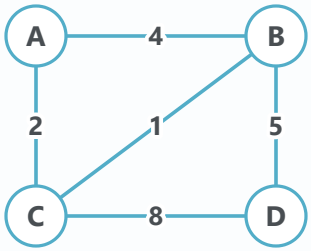
Çok bağlı: yalnızca e = 5 öğe (N1…N5). Her öğe, kuyruğunun çıkış listesinde ve başının giriş listesinde yer alır.

Çıkarım: Önce Verinin Nasıl Depolanacağı Düşünülür

Gösterim	Bellek	(u,v) var mı?	u'nun komşuları	Ekleme/silme
Komşuluk matrisi	$O(n^2)$	O(1)	O(n)	O(1)
Komşuluk listesi	O(n + 2e)	$O(d_u)$	$O(d_u)$	başa O(1) / arama O(d)
Sıralı dizi	n + 2e + 1	$O(d_u)$	$O(d_u)$	pahalı (kaydırma)
Ortogonal (yönlü)	O(n + e)	O(d)	giriş ve çıkış O(d)	O(1) başa
Çok bağlı	O(n + e)	$O(d_u)$	$O(d_u)$	O(1) başa

Hocadan: "Akla gelen ilk çözüm iyi bir yöntem olmayabilir. Belki n^3'lük bir algoritmayı n'e indirmek mümkün olabilir. Bu noktada veri yapısını iyi tasarlamak gerekir. Daha kod yazmadan önce veriyi nasıl depolayacağımı düşünüyorum. Veriyi depolamak o kadar kolay değil!"

Ağırlıklı Çizge Gösterimi



ağırlık = mesafe, maliyet, süre…

maliyet matrisi: kenar yoksa ∞ (C'de büyük sabit)

	A	B	C	D
A	0	4	2	∞
B	4	0	1	5
C	2	1	0	8
D	∞	5	8	0

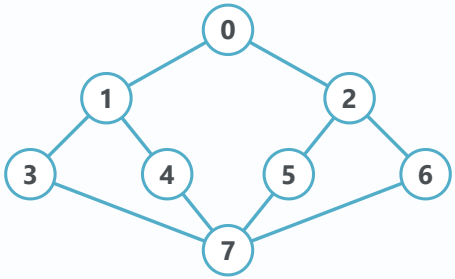
```c
#define SONSUZ 1000000   /* INT_MAX degil: toplamda tasar */
int maliyet[MAX_DUGUM][MAX_DUGUM];

typedef struct aoge *aoge_gost;
typedef struct aoge {
    int dugum;
    int agirlik;         /* kenarin maliyeti */
    aoge_gost bag;
} aoge;
aoge_gost abaslik[MAX_DUGUM];
```

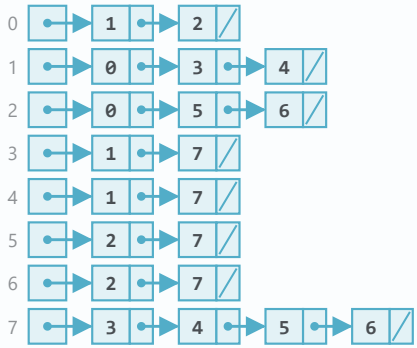
- Matriste 1 yerine **ağırlık**, kenar yoksa **∞**.
- Listede öğeye `agirlik` alanı eklenir.
- ∞ için `INT_MAX` kullanmak `d + w` toplamında taşmaya yol açar; güvenli büyük sabit seçilir.
- MST ve en kısa yol algoritmaları bu gösterimi kullanır.

Çizge Üzerinde Temel İşlemler: Gezinme

- Yönsüz G = (V, E) ve bir v düğümü verildiğinde, v'den ulaşılabilen **tüm düğümlere** uğramak (ziyaret etmek = veri alanını yazdırmak).
- **Önce derinliğine arama** (*depth first search*, DFS) ağaçtaki **kök önde** gezinmeye benzer.
- **Önce genişliğine arama** (*breadth first search*, BFS) ağaçtaki **düzey düzey** gezinmeye benzer.
- Her ikisinde çizge **komşuluk listesiyle** gerçeklenmiştir; listeler sıralıdır.



Örnek çizge



baslik[] komşuluk listesi

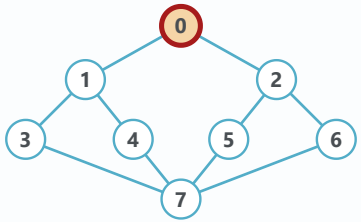
Önce Derinliğine Arama (DFS)

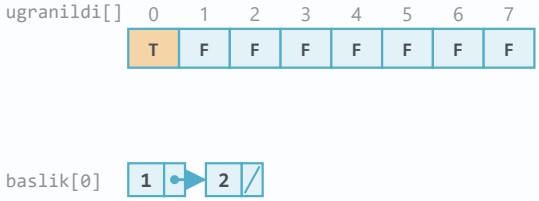
- v ziyaret edilir; v'nin listesinden **ziyaret edilmemiş** bir w seçilir ve arama **w üzerinde** sürer.
- Listedeki konumu saklamak için **yığın** gerekir; u'da gidecek yer kalmayınca yığından bir düğüm alınıp devam edilir.
- Karmaşık görünen çözümleme **özyinelemeli yordamla** kolaylaşır: yığın, çağrı yığınıdır.
- Global `ugranildi[]` dizisi başta FALSE olmalıdır.

```
#include <stdio.h>
#include <stdbool.h>
#define TRUE  true
#define FALSE false
bool ugranildi[MAX_DUGUM];   /* ';' unutulmamali */
```

```
void dfs(int v)   /* once derinligine arama */
{
    oge_gost w;
    ugranildi[v] = TRUE;    /* ugradim */
    printf("%d ", v);
    for (w = baslik[v]; w; w = w->bag)
        if (!ugranildi[w->dugum])
            dfs(w->dugum);
}
```

Kod İzleme: `dfs(0)`





`dfs(0)` çağrıldı: **0** işaretlenir ve yazılır.

Adım 1 / 14

```
1 void dfs(int v)   /* once derinligine arama */
2 {
3     oge_gost w;
▶     ugranildi[v] = TRUE;    /* ugradim */
▶     printf("%d ", v);
6     for (w = baslik[v]; w; w = w->bag)
7         if (!ugranildi[w->dugum])
8             dfs(w->dugum);
9 }
```

DEĞİŞKENLER

`v = 0` `w = –`

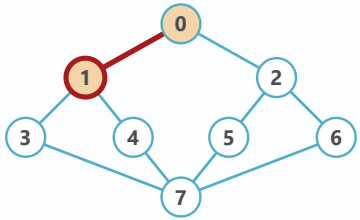
ÇAĞRI YIĞINI

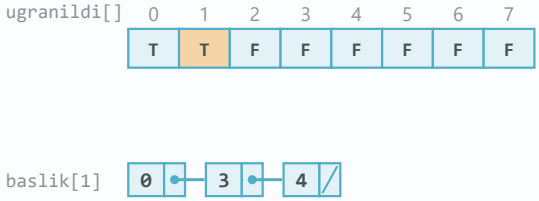
`dfs(0)` `main()`

ÇIKTI

```
0
```

Kod İzleme: `dfs(0)`





0 → 1: 1 ziyaret edilmemiş → `dfs(1)`. 1 işaretlenir, yazılır.

Adım 2 / 14

```
1 void dfs(int v)   /* once derinligine arama */
2 {
3     oge_gost w;
▶     ugranildi[v] = TRUE;    /* ugradim */
▶     printf("%d ", v);
6     for (w = baslik[v]; w; w = w->bag)
7         if (!ugranildi[w->dugum])
8             dfs(w->dugum);
9 }
```

DEĞİŞKENLER

`v = 1` `w = –`

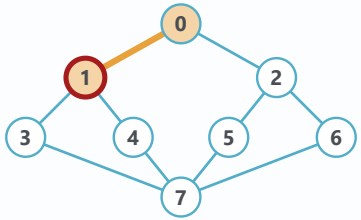
ÇAĞRI YIĞINI

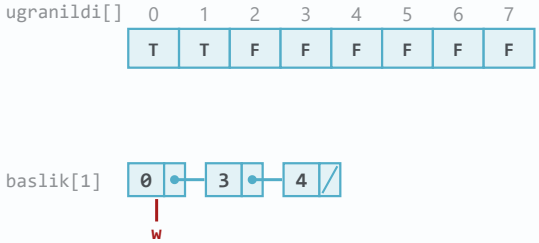
`dfs(1)` `dfs(0)` `main()`

ÇIKTI

```
0 1
```

Kod İzleme: `dfs(0)`





w → 0: `ugranildi[0]` TRUE, atlanır.

Adım 3 / 14

```
1 void dfs(int v)   /* once derinligine arama */
2 {
3     oge_gost w;
4     ugranildi[v] = TRUE;     /* ugradim */
5     printf("%d ", v);
▶     for (w = baslik[v]; w; w = w->bag)
▶         if (!ugranildi[w->dugum])
8             dfs(w->dugum);
9 }
```

DEĞİŞKENLER

`v = 1` `w = &oge(0)`

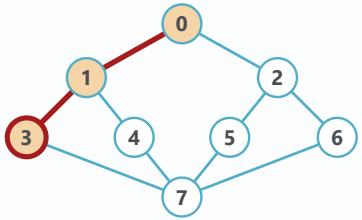
ÇAĞRI YIĞINI

`dfs(1)` `dfs(0)` `main()`

ÇIKTI

```
0 1
```

Kod İzleme: `dfs(0)`



ugranildi[]

0	1	2	3	4	5	6	7
T	T	F	T	F	F	F	F

baslik[3] → 1 → 7

1 → 3: 3 ziyaret edilmemiş → `dfs(3)`. 3 işaretlenir, yazılır.

Adım 4 / 14

```
1 void dfs(int v)   /* once derinligine arama */
2 {
3     oge_gost w;
▶     ugranildi[v] = TRUE;    /* ugradim */
▶     printf("%d ", v);
6     for (w = baslik[v]; w; w = w->bag)
7         if (!ugranildi[w->dugum])
8             dfs(w->dugum);
9 }
```

DEĞİŞKENLER

`v = 3` `w = –`

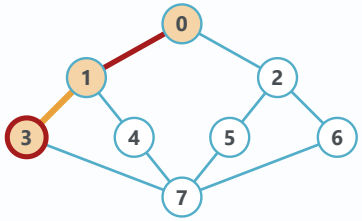
ÇAĞRI YIĞINI

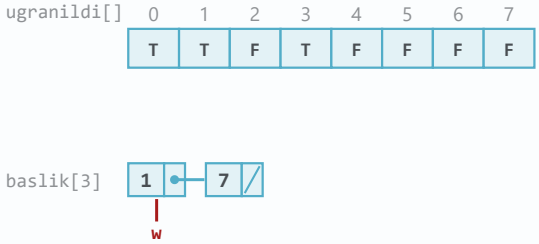
`dfs(3)` `dfs(1)` `dfs(0)` `main()`

ÇIKTI

```
0 1 3
```

Kod İzleme: `dfs(0)`





w → 1: `ugranildi[1]` TRUE, atlanır.

Adım 5 / 14

```
1 void dfs(int v)   /* once derinligine arama */
2 {
3     oge_gost w;
4     ugranildi[v] = TRUE;     /* ugradim */
5     printf("%d ", v);
▶     for (w = baslik[v]; w; w = w->bag)
▶         if (!ugranildi[w->dugum])
8             dfs(w->dugum);
9 }
```

DEĞİŞKENLER

`v = 3` `w = &oge(1)`

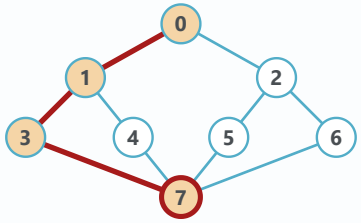
ÇAĞRI YIĞINI

`dfs(3)` `dfs(1)` `dfs(0)` `main()`

ÇIKTI

```
0 1 3
```

Kod İzleme: `dfs(0)`



ugranildi[]

0	1	2	3	4	5	6	7
T	T	F	T	F	F	F	T

baslik[7]



3 → 7: 7 ziyaret edilmemiş → `dfs(7)`. 7 işaretlenir, yazılır.

Adım 6 / 14

```
1 void dfs(int v)   /* once derinligine arama */
2 {
3     oge_gost w;
▶     ugranildi[v] = TRUE;    /* ugradim */
▶     printf("%d ", v);
6     for (w = baslik[v]; w; w = w->bag)
7         if (!ugranildi[w->dugum])
8             dfs(w->dugum);
9 }
```

DEĞİŞKENLER

`v = 7` `w = –`

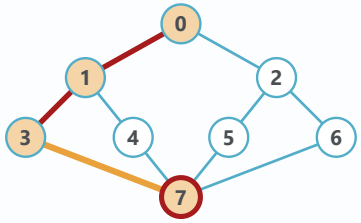
ÇAĞRI YIĞINI

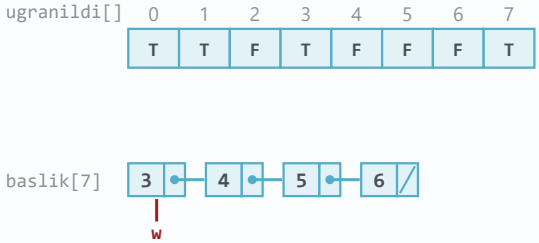
`dfs(7)` `dfs(3)` `dfs(1)` `dfs(0)` `main()`

ÇIKTI

```
0 1 3 7
```

Kod İzleme: `dfs(0)`





w → 3: `ugranildi[3]` TRUE, atlanır.

Adım 7 / 14

```
1 void dfs(int v)   /* once derinligine arama */
2 {
3     oge_gost w;
4     ugranildi[v] = TRUE;     /* ugradim */
5     printf("%d ", v);
▶     for (w = baslik[v]; w; w = w->bag)
▶         if (!ugranildi[w->dugum])
8             dfs(w->dugum);
9 }
```

DEĞİŞKENLER

`v = 7` `w = &oge(3)`

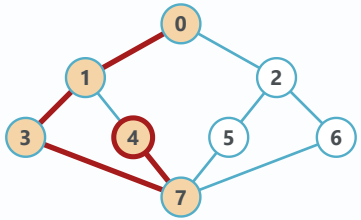
ÇAĞRI YIĞINI

`dfs(7)` `dfs(3)` `dfs(1)` `dfs(0)` `main()`

ÇIKTI

```
0 1 3 7
```

Kod İzleme: `dfs(0)`



ugranildi[]

0	1	2	3	4	5	6	7
T	T	F	T	T	F	F	T

baslik[4] → 1 → 7

7 → 4: 4 ziyaret edilmemiş → `dfs(4)`. 4 işaretlenir, yazılır.

Adım 8 / 14

```
1 void dfs(int v)   /* once derinligine arama */
2 {
3     oge_gost w;
▶     ugranildi[v] = TRUE;    /* ugradim */
▶     printf("%d ", v);
6     for (w = baslik[v]; w; w = w->bag)
7         if (!ugranildi[w->dugum])
8             dfs(w->dugum);
9 }
```

DEĞİŞKENLER

`v = 4` `w = –`

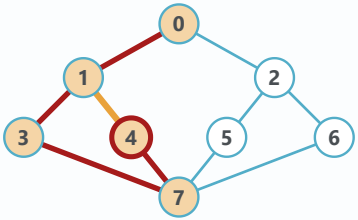
ÇAĞRI YIĞINI

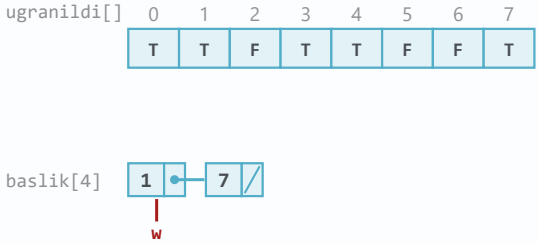
`dfs(4)` `dfs(7)` `dfs(3)` `dfs(1)` `dfs(0)` `main()`

ÇIKTI

```
0 1 3 7 4
```

Kod İzleme: `dfs(0)`





w → 1: `ugranildi[1]` TRUE, atlanır.

Adım 9 / 14

```
1 void dfs(int v)   /* once derinligine arama */
2 {
3     oge_gost w;
4     ugranildi[v] = TRUE;     /* ugradim */
5     printf("%d ", v);
▶     for (w = baslik[v]; w; w = w->bag)
▶         if (!ugranildi[w->dugum])
8             dfs(w->dugum);
9 }
```

DEĞİŞKENLER

`v = 4` `w = &oge(1)`

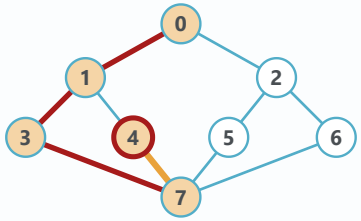
ÇAĞRI YIĞINI

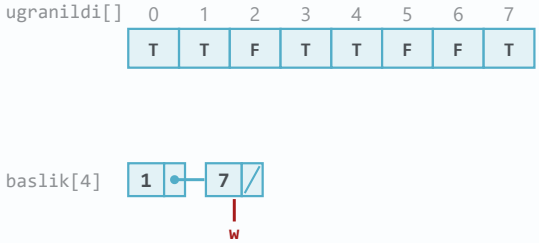
`dfs(4)` `dfs(7)` `dfs(3)` `dfs(1)` `dfs(0)` `main()`

ÇIKTI

```
0 1 3 7 4
```

Kod İzleme: `dfs(0)`





w → 7: `ugranildi[7]` TRUE, atlanır.

Adım 10 / 14

```
1 void dfs(int v)   /* once derinligine arama */
2 {
3     oge_gost w;
4     ugranildi[v] = TRUE;     /* ugradim */
5     printf("%d ", v);
▶     for (w = baslik[v]; w; w = w->bag)
▶         if (!ugranildi[w->dugum])
8             dfs(w->dugum);
9 }
```

DEĞİŞKENLER

`v = 4` `w = &oge(7)`

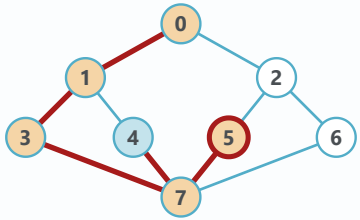
ÇAĞRI YIĞINI

`dfs(4)` `dfs(7)` `dfs(3)` `dfs(1)` `dfs(0)` `main()`

ÇIKTI

```
0 1 3 7 4
```

Kod İzleme: `dfs(0)`



ugranildi[]

0	1	2	3	4	5	6	7
T	T	F	T	T	T	F	T

baslik[5] → 2 → 7 → /

(`dfs(4)` bitti.) 7 → 5: 5 ziyaret edilmemiş → `dfs(5)`. 5 işaretlenir, yazılır.

Adım 11 / 14

```
1 void dfs(int v)   /* once derinligine arama */
2 {
3     oge_gost w;
▶     ugranildi[v] = TRUE;    /* ugradim */
▶     printf("%d ", v);
6     for (w = baslik[v]; w; w = w->bag)
7         if (!ugranildi[w->dugum])
8             dfs(w->dugum);
9 }
```

DEĞİŞKENLER

`v = 5` `w = –`

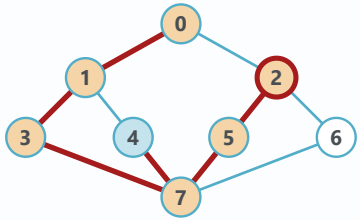
ÇAĞRI YIĞINI

`dfs(5)` `dfs(7)` `dfs(3)` `dfs(1)` `dfs(0)` `main()`

ÇIKTI

```
0 1 3 7 4 5
```

Kod İzleme: `dfs(0)`



ugranildi[]

0	1	2	3	4	5	6	7
T	T	T	T	T	T	F	T

baslik[2] → 0 → 5 → 6 /

5 → 2: 2 ziyaret edilmemiş → `dfs(2)`. 2 işaretlenir, yazılır.

Adım 12 / 14

```
1 void dfs(int v)   /* once derinligine arama */
2 {
3     oge_gost w;
▶     ugranildi[v] = TRUE;    /* ugradim */
▶     printf("%d ", v);
6     for (w = baslik[v]; w; w = w->bag)
7         if (!ugranildi[w->dugum])
8             dfs(w->dugum);
9 }
```

DEĞİŞKENLER

`v = 2` `w = –`

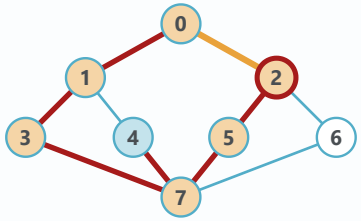
ÇAĞRI YIĞINI

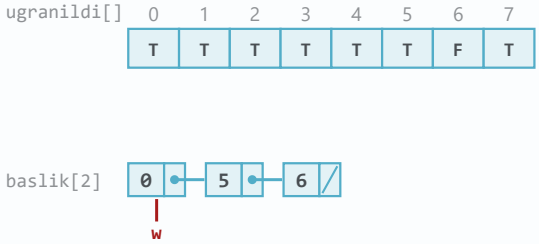
`dfs(2)` `dfs(5)` `dfs(7)` `dfs(3)` `dfs(1)` `dfs(0)` `main()`

ÇIKTI

```
0 1 3 7 4 5 2
```

Kod İzleme: `dfs(0)`





w → 0: `ugranildi[0]` TRUE, atlanır.

Adım 13 / 14

```
1 void dfs(int v)   /* once derinligine arama */
2 {
3     oge_gost w;
4     ugranildi[v] = TRUE;     /* ugradim */
5     printf("%d ", v);
▶     for (w = baslik[v]; w; w = w->bag)
▶         if (!ugranildi[w->dugum])
8             dfs(w->dugum);
9 }
```

DEĞİŞKENLER

`v = 2` `w = &oge(0)`

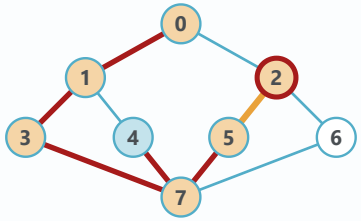
ÇAĞRI YIĞINI

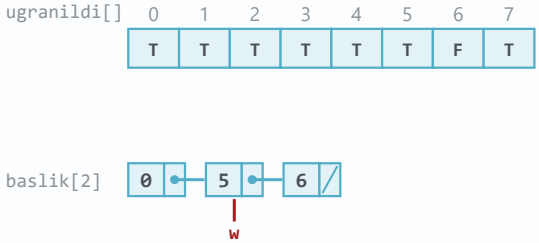
`dfs(2)` `dfs(5)` `dfs(7)` `dfs(3)` `dfs(1)` `dfs(0)` `main()`

ÇIKTI

```
0 1 3 7 4 5 2
```

Kod İzleme: `dfs(0)`





w → 5: `ugranildi[5]` TRUE, atlanır.

Adım 14 / 14

```
1 void dfs(int v)   /* once derinligine arama */
2 {
3     oge_gost w;
4     ugranildi[v] = TRUE;     /* ugradim */
5     printf("%d ", v);
▶     for (w = baslik[v]; w; w = w->bag)
▶         if (!ugranildi[w->dugum])
8             dfs(w->dugum);
9 }
```

DEĞİŞKENLER

`v = 2` `w = &oge(5)`

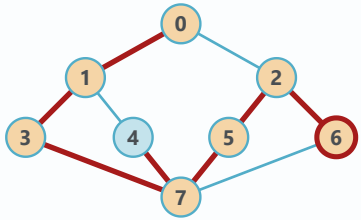
ÇAĞRI YIĞINI

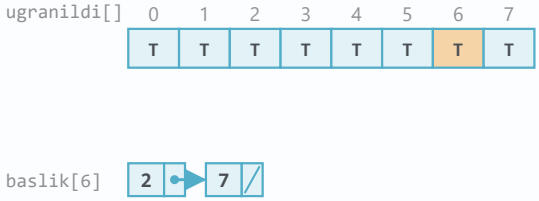
`dfs(2)` `dfs(5)` `dfs(7)` `dfs(3)` `dfs(1)` `dfs(0)` `main()`

ÇIKTI

```
0 1 3 7 4 5 2
```

Kod İzleme: `dfs(0)` (devam)





2 → 6: 6 ziyaret edilmemiş → `dfs(6)`. 6 işaretlenir, yazılır.

Adım 1 / 8

```
1 void dfs(int v)   /* once derinligine arama */
2 {
3     oge_gost w;
▶     ugranildi[v] = TRUE;    /* ugradim */
▶     printf("%d ", v);
6     for (w = baslik[v]; w; w = w->bag)
7         if (!ugranildi[w->dugum])
8             dfs(w->dugum);
9 }
```

DEĞİŞKENLER

`v = 6` `w = –`

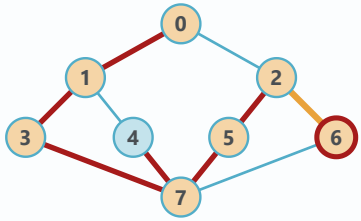
ÇAĞRI YIĞINI

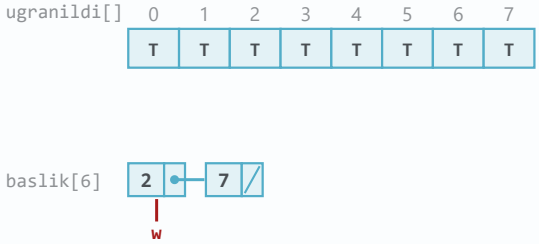
`dfs(6)` `dfs(2)` `dfs(5)` `dfs(7)` `dfs(3)` `dfs(1)` `dfs(0)` `main()`

ÇIKTI

```
0 1 3 7 4 5 2 6
```

Kod İzleme: `dfs(0)` (devam)





w → 2: `ugranildi[2]` TRUE, atlanır.

Adım 2 / 8

```
1 void dfs(int v)   /* once derinligine arama */
2 {
3     oge_gost w;
4     ugranildi[v] = TRUE;     /* ugradim */
5     printf("%d ", v);
▶     for (w = baslik[v]; w; w = w->bag)
▶         if (!ugranildi[w->dugum])
8             dfs(w->dugum);
9 }
```

DEĞİŞKENLER

`v = 6` `w = &oge(2)`

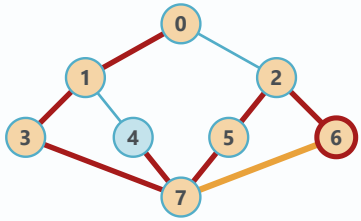
ÇAĞRI YIĞINI

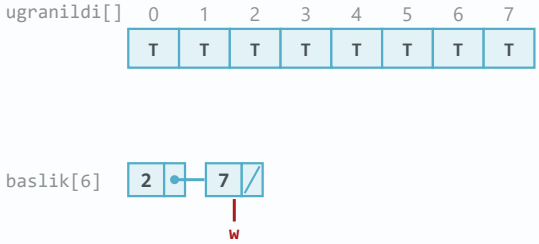
`dfs(6)` `dfs(2)` `dfs(5)` `dfs(7)`
`dfs(3)` `dfs(1)` `dfs(0)` `main()`

ÇIKTI

```
0 1 3 7 4 5 2 6
```

Kod İzleme: `dfs(0)` (devam)





w → 7: `ugranildi[7]` TRUE, atlanır.

Adım 3 / 8

```
1 void dfs(int v)   /* once derinligine arama */
2 {
3     oge_gost w;
4     ugranildi[v] = TRUE;     /* ugradim */
5     printf("%d ", v);
▶     for (w = baslik[v]; w; w = w->bag)
▶         if (!ugranildi[w->dugum])
8             dfs(w->dugum);
9 }
```

DEĞİŞKENLER

`v = 6` `w = &oge(7)`

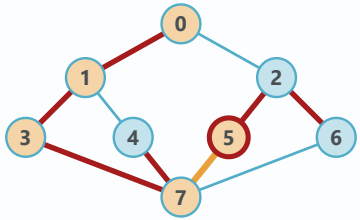
ÇAĞRI YIĞINI

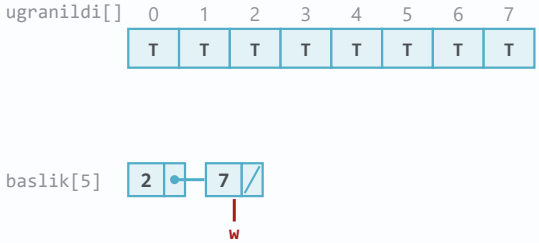
`dfs(6)` `dfs(2)` `dfs(5)` `dfs(7)`
`dfs(3)` `dfs(1)` `dfs(0)` `main()`

ÇIKTI

```
0 1 3 7 4 5 2 6
```

Kod İzleme: `dfs(0)` (devam)





`dfs(6), dfs(2)` bitti; 5'te kalınan yerden devam. w → 7: `ugranildi[7]` TRUE, atlanır.

Adım 4 / 8

```
1 void dfs(int v)   /* once derinligine arama */
2 {
3     oge_gost w;
4     ugranildi[v] = TRUE;     /* ugradim */
5     printf("%d ", v);
▶     for (w = baslik[v]; w; w = w->bag)
▶         if (!ugranildi[w->dugum])
8             dfs(w->dugum);
9 }
```

DEĞİŞKENLER

`v = 5` `w = &oge(7)`

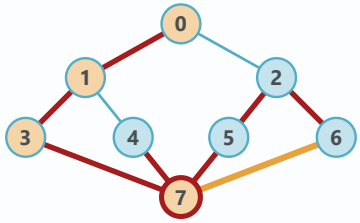
ÇAĞRI YIĞINI

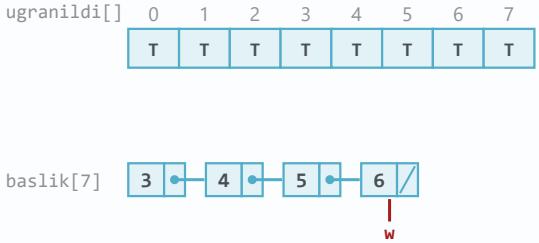
`dfs(5)` `dfs(7)` `dfs(3)` `dfs(1)` `dfs(0)` `main()`

ÇIKTI

```
0 1 3 7 4 5 2 6
```

Kod İzleme: `dfs(0)` (devam)





`dfs(5)` bitti; 7'de kalınan yerden devam. w → 6: `ugranildi[6]` TRUE, atlanır.

Adım 5 / 8

```
1 void dfs(int v)   /* once derinligine arama */
2 {
3     oge_gost w;
4     ugranildi[v] = TRUE;     /* ugradim */
5     printf("%d ", v);
▶     for (w = baslik[v]; w; w = w->bag)
▶         if (!ugranildi[w->dugum])
8             dfs(w->dugum);
9 }
```

DEĞİŞKENLER

`v = 7` `w = &oge(6)`

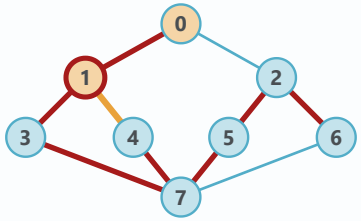
ÇAĞRI YIĞINI

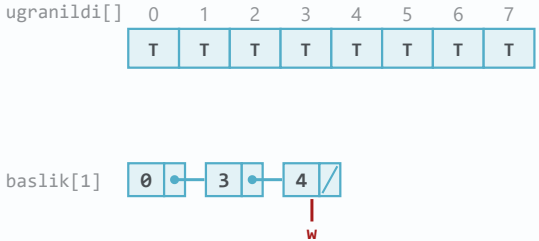
`dfs(7)` `dfs(3)` `dfs(1)` `dfs(0)` `main()`

ÇIKTI

```
0 1 3 7 4 5 2 6
```

Kod İzleme: `dfs(0)` (devam)





`dfs(7), dfs(3)` bitti; 1'de kalınan yerden devam. w → 4: `ugranildi[4]` TRUE, atlanır.

Adım 6 / 8

```
1 void dfs(int v)   /* once derinligine arama */
2 {
3     oge_gost w;
4     ugranildi[v] = TRUE;     /* ugradim */
5     printf("%d ", v);
▶     for (w = baslik[v]; w; w = w->bag)
▶         if (!ugranildi[w->dugum])
8             dfs(w->dugum);
9 }
```

DEĞİŞKENLER

`v = 1` `w = &oge(4)`

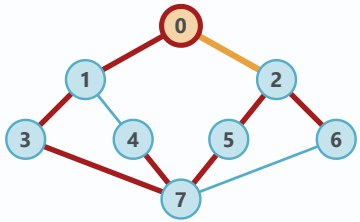
ÇAĞRI YIĞINI

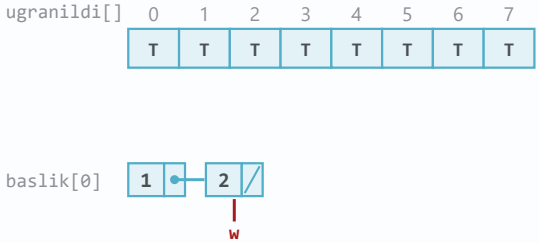
`dfs(1)` `dfs(0)` `main()`

ÇIKTI

```
0 1 3 7 4 5 2 6
```

Kod İzleme: `dfs(0)` (devam)





`dfs(1)` bitti; 0'da kalınan yerden devam. w → 2: `ugranildi[2]` TRUE, atlanır.

Adım 7 / 8

```
1 void dfs(int v)   /* once derinligine arama */
2 {
3     oge_gost w;
4     ugranildi[v] = TRUE;     /* ugradim */
5     printf("%d ", v);
▶     for (w = baslik[v]; w; w = w->bag)
▶         if (!ugranildi[w->dugum])
8             dfs(w->dugum);
9 }
```

DEĞİŞKENLER

`v = 0` `w = &oge(2)`

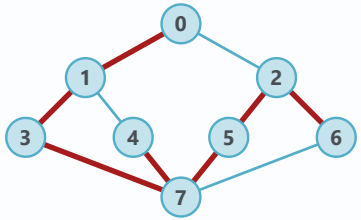
ÇAĞRI YIĞINI

`dfs(0)` `main()`

ÇIKTI

```
0 1 3 7 4 5 2 6
```

Kod İzleme: `dfs(0)` (devam)



ugranildi[]

0	1	2	3	4	5	6	7
T	T	T	T	T	T	T	T

`dfs(0)` bitti. Gezinme sırası **0 1 3 7 4 5 2 6**; turuncu kenarlar bir **ağaç** oluşturdu.

Adım 8 / 8

```
1 void dfs(int v)   /* once derinligine arama */
2 {
3     oge_gost w;
4     ugranildi[v] = TRUE;     /* ugradim */
5     printf("%d ", v);
6     for (w = baslik[v]; w; w = w->bag)
7         if (!ugranildi[w->dugum])
8             dfs(w->dugum);
▶ }
```

DEĞİŞKENLER

`v = 0` `w = NULL`

ÇAĞRI YIĞINI

`main()`

ÇIKTI

```
0 1 3 7 4 5 2 6
```

DFS'nin Sonucu Neye Bağlı?

Başlangıç düğümü

0'dan: 0 1 3 7 4 5 2 6. Başlangıç 0 olmak zorunda değil: 6'dan 6 2 0 1 3 7 4 5; 4'ten 4 1 0 2 5 7 3 6. Aynı öğeler, farklı diziliş.

Listenin sıraya etkisi

0'ın listesi {1, 2} değil de **{2, 1}** olsaydı sıra: 0 2 5 7 3 1 4 6. Liste başa mı sona mı ekleyerek kuruldu, sıralı mı — bunlar gezinmeyi değiştirir.

Analiz

dfs her düğüm için en fazla **1 kez** çağrılır; her liste bir kez taranır: **O(n + e)**. Kenar sayısı en fazla n(n − 1)/2.

Veri yapısı

Matrisle gösterilseydi v'nin komşularını bulmak O(n), toplam **$O(n^2)$** olurdu. Listeyle doğrusal zamanda geziniyoruz.

Gezinme sonucunda bir **ağaç** oluştu (önce derinliğine kapsayan ağaç). Bu ağaca bir kenar daha eklersek döngü oluşur ve ağaç olmaktan çıkar.

Önce Genişliğine Arama (BFS)

- v'den başlanır, v işaretlenir; sonra v'nin listesindeki düğümlerin **hepsi** ziyaret edilir.
- Sonra v'nin listesindeki **ilk** düğümün ziyaret edilmemiş komşuları, sonra ikincisininki...
- Ziyaret edilen düğüm **kuyruğa** atılır (ilk giren ilk çıkar); liste bitince kuyruktan bir düğüm alınır ve onun listesinde ilerlenir.
- Kuyruk boşalınca arama biter.

> **Hocadan:** "Nasıl ki, önce derinliğine aramada problemin doğası gereği yığın kullanma zorunluluğu vardı; bunda da kuyruk kullanma zorunluluğu vardır."

```
void bfs(int v)   /* once genisligine arama */
{
    oge_gost w;
    kuyruk_gost on = NULL, arka = NULL;
    printf("%d ", v);
    ugranildi[v] = TRUE;
    kuyruk_ekle(&on, &arka, v);
    while (on) {
        v = kuyruk_al(&on, &arka);
        for (w = baslik[v]; w; w = w->bag)
            if (!ugranildi[w->dugum]) {
                printf("%d ", w->dugum);
                kuyruk_ekle(&on, &arka, w->dugum);
                ugranildi[w->dugum] = TRUE;
            }
    }
}
```

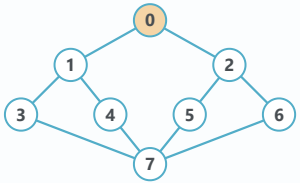
BFS'nin Kullandığı Dinamik Kuyruk

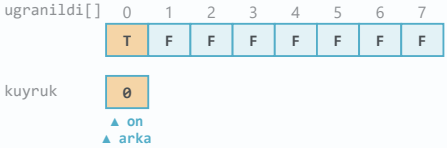
```
typedef struct kuyruk *kuyruk_gost;
typedef struct kuyruk {
    int dugum;
    kuyruk_gost bag;
} kuyruk;
void kuyruk_ekle(kuyruk_gost *on,
                 kuyruk_gost *arka, int d)
{
    kuyruk_gost p = malloc(sizeof(kuyruk));
    if (p == NULL) exit(1);
    p->dugum = d;  p->bag = NULL;
    if (*on) (*arka)->bag = p;  else *on = p;
    *arka = p;
}
```

```
int kuyruk_al(kuyruk_gost *on,
              kuyruk_gost *arka)
{
    kuyruk_gost p = *on;    /* on != NULL */
    int d = p->dugum;
    *on = p->bag;
    if (*on == NULL) *arka = NULL;
    free(p);
    return d;
}
```

`int kuyruk_al(kuyruk_gost *)` gibi yalnız ön işaretçiyi alan bir prototip yetmez. Kuyruk boşalınca `arka` da NULL yapılmalı; bu yüzden `kuyruk_al`'a `arka` da geçirilir.

Kod İzleme: `bfs(0)`





0 yazılır, işaretlenir ve kuyruğa eklenir.

Adım 1 / 12

```
1  on = arka = NULL;
▶  printf("%d ", v);  ugranildi[v] = TRUE;
▶  kuyruk_ekle(&on, &arka, v);
4  while (on) {
5      v = kuyruk_al(&on, &arka);
6      for (w = baslik[v]; w; w = w->bag)
7          if (!ugranildi[w->dugum]) {
8              printf("%d ", w->dugum);
9              kuyruk_ekle(&on, &arka, w->dugum);
10             ugranildi[w->dugum] = TRUE;
11         }
12 }
```

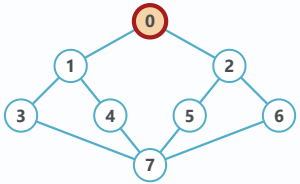
DEĞİŞKENLER

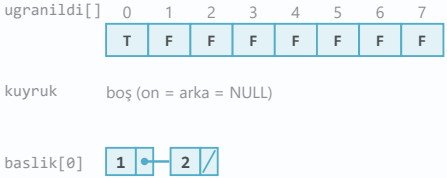
`v = 0` `w = –`

ÇIKTI

```
0
```

Kod İzleme: `bfs(0)`





Kuyruk boş değil: öndeki **0** alınır; şimdi 0'ın listesi taranacak.

Adım 2 / 12

```
1  on = arka = NULL;
2  printf("%d ", v);  ugranildi[v] = TRUE;
3  kuyruk_ekle(&on, &arka, v);
▶  while (on) {
▶      v = kuyruk_al(&on, &arka);
6      for (w = baslik[v]; w; w = w->bag)
7          if (!ugranildi[w->dugum]) {
8              printf("%d ", w->dugum);
9              kuyruk_ekle(&on, &arka, w->dugum);
10             ugranildi[w->dugum] = TRUE;
11         }
12 }
```

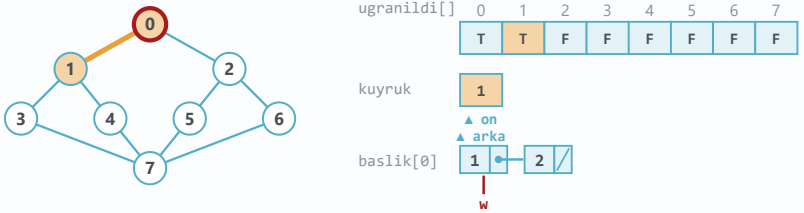
DEĞİŞKENLER

`v = 0` `w = –`

ÇIKTI

```
0
```

Kod İzleme: `bfs(0)`



w → 1: ziyaret edilmemiş → yazılır, kuyruğun **arkasına** eklenir, işaretlenir.

Adım 3 / 12

```
1  on = arka = NULL;
2  printf("%d ", v);  ugranildi[v] = TRUE;
3  kuyruk_ekle(&on, &arka, v);
4  while (on) {
5      v = kuyruk_al(&on, &arka);
6      for (w = baslik[v]; w; w = w->bag)
▶          if (!ugranildi[w->dugum]) {
▶              printf("%d ", w->dugum);
▶              kuyruk_ekle(&on, &arka, w->dugum);
▶              ugranildi[w->dugum] = TRUE;
11         }
12 }
```

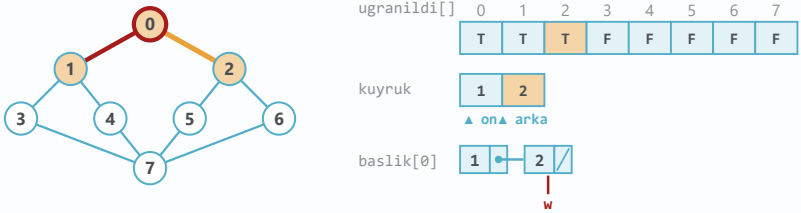
DEĞİŞKENLER

`v = 0` `w = &oge(1)`

ÇIKTI

```
0 1
```

Kod İzleme: `bfs(0)`



w → 2: ziyaret edilmemiş → yazılır, kuyruğun **arkasına** eklenir, işaretlenir.

Adım 4 / 12

```
1  on = arka = NULL;
2  printf("%d ", v);  ugranildi[v] = TRUE;
3  kuyruk_ekle(&on, &arka, v);
4  while (on) {
5      v = kuyruk_al(&on, &arka);
6      for (w = baslik[v]; w; w = w->bag)
▶          if (!ugranildi[w->dugum]) {
▶              printf("%d ", w->dugum);
▶              kuyruk_ekle(&on, &arka, w->dugum);
▶              ugranildi[w->dugum] = TRUE;
11         }
12 }
```

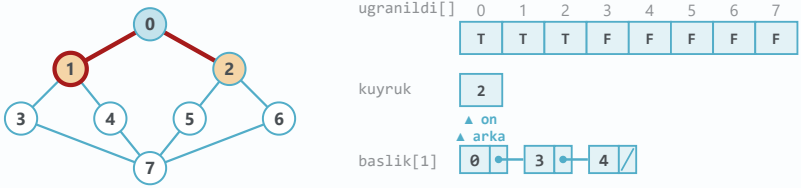
DEĞİŞKENLER

`v = 0` `w = &oge(2)`

ÇIKTI

```
0 1 2
```

Kod İzleme: `bfs(0)`



Kuyruk boş değil: öndeki **1** alınır; şimdi 1'in listesi taranacak.

Adım 5 / 12

```
1  on = arka = NULL;
2  printf("%d ", v);  ugranildi[v] = TRUE;
3  kuyruk_ekle(&on, &arka, v);
▶  while (on) {
▶      v = kuyruk_al(&on, &arka);
6      for (w = baslik[v]; w; w = w->bag)
7          if (!ugranildi[w->dugum]) {
8              printf("%d ", w->dugum);
9              kuyruk_ekle(&on, &arka, w->dugum);
10             ugranildi[w->dugum] = TRUE;
11         }
12 }
```

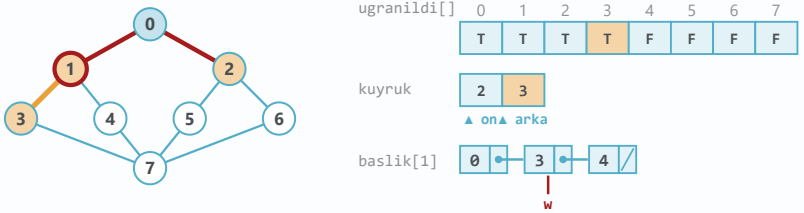
DEĞİŞKENLER

`v = 1` `w = –`

ÇIKTI

```
0 1 2
```

Kod İzleme: `bfs(0)`



w → 3: ziyaret edilmemiş → yazılır, kuyruğun **arkasına** eklenir, işaretlenir. (0 zaten işaretliydi.)

Adım 6 / 12

```
1  on = arka = NULL;
2  printf("%d ", v);  ugranildi[v] = TRUE;
3  kuyruk_ekle(&on, &arka, v);
4  while (on) {
5      v = kuyruk_al(&on, &arka);
6      for (w = baslik[v]; w; w = w->bag)
▶          if (!ugranildi[w->dugum]) {
▶              printf("%d ", w->dugum);
▶              kuyruk_ekle(&on, &arka, w->dugum);
▶              ugranildi[w->dugum] = TRUE;
11         }
12 }
```

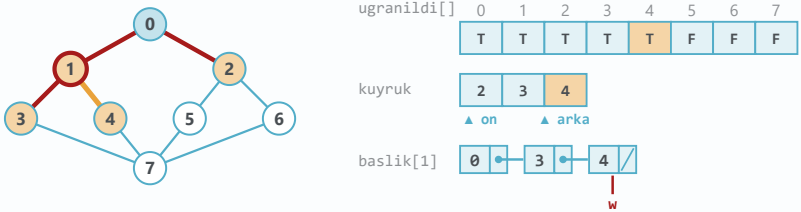
DEĞİŞKENLER

`v = 1` `w = &oge(3)`

ÇIKTI

```
0 1 2 3
```

Kod İzleme: `bfs(0)`



w → 4: ziyaret edilmemiş → yazılır, kuyruğun **arkasına** eklenir, işaretlenir.

Adım 7 / 12

```
1  on = arka = NULL;
2  printf("%d ", v);  ugranildi[v] = TRUE;
3  kuyruk_ekle(&on, &arka, v);
4  while (on) {
5      v = kuyruk_al(&on, &arka);
6      for (w = baslik[v]; w; w = w->bag)
▶          if (!ugranildi[w->dugum]) {
▶              printf("%d ", w->dugum);
▶              kuyruk_ekle(&on, &arka, w->dugum);
▶              ugranildi[w->dugum] = TRUE;
11         }
12 }
```

DEĞİŞKENLER

`v = 1` `w = &oge(4)`

ÇIKTI

```
0 1 2 3 4
```

Kod İzleme: `bfs(0)`



Kuyruk boş değil: öndeki **2** alınır; şimdi 2'nin listesi taranacak.

Adım 8 / 12

```
1  on = arka = NULL;
2  printf("%d ", v);  ugranildi[v] = TRUE;
3  kuyruk_ekle(&on, &arka, v);
▶  while (on) {
▶      v = kuyruk_al(&on, &arka);
6      for (w = baslik[v]; w; w = w->bag)
7          if (!ugranildi[w->dugum]) {
8              printf("%d ", w->dugum);
9              kuyruk_ekle(&on, &arka, w->dugum);
10             ugranildi[w->dugum] = TRUE;
11         }
12 }
```

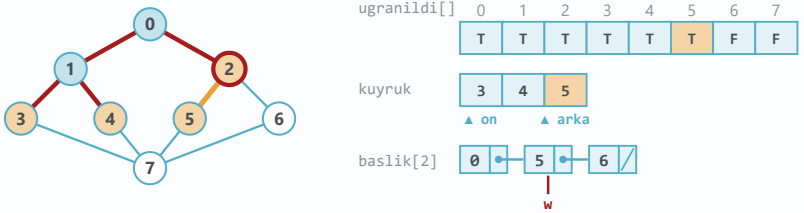
DEĞİŞKENLER

`v = 2` `w = –`

ÇIKTI

```
0 1 2 3 4
```

Kod İzleme: `bfs(0)`



w → 5: ziyaret edilmemiş → yazılır, kuyruğun **arkasına** eklenir, işaretlenir. (0 zaten işaretliydi.)

Adım 9 / 12

```
1  on = arka = NULL;
2  printf("%d ", v);  ugranildi[v] = TRUE;
3  kuyruk_ekle(&on, &arka, v);
4  while (on) {
5      v = kuyruk_al(&on, &arka);
6      for (w = baslik[v]; w; w = w->bag)
▶          if (!ugranildi[w->dugum]) {
▶              printf("%d ", w->dugum);
▶              kuyruk_ekle(&on, &arka, w->dugum);
▶              ugranildi[w->dugum] = TRUE;
11         }
12 }
```

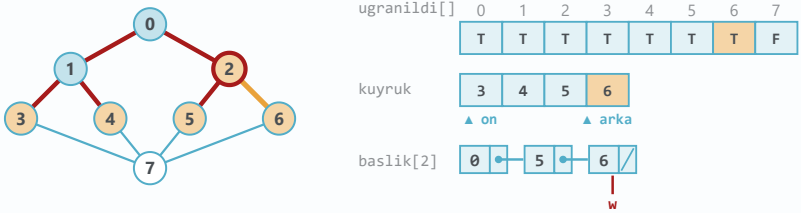
DEĞİŞKENLER

`v = 2` `w = &oge(5)`

ÇIKTI

```
0 1 2 3 4 5
```

Kod İzleme: `bfs(0)`



w → 6: ziyaret edilmemiş → yazılır, kuyruğun **arkasına** eklenir, işaretlenir.

Adım 10 / 12

```
1  on = arka = NULL;
2  printf("%d ", v);  ugranildi[v] = TRUE;
3  kuyruk_ekle(&on, &arka, v);
4  while (on) {
5      v = kuyruk_al(&on, &arka);
6      for (w = baslik[v]; w; w = w->bag)
▶          if (!ugranildi[w->dugum]) {
▶              printf("%d ", w->dugum);
▶              kuyruk_ekle(&on, &arka, w->dugum);
▶              ugranildi[w->dugum] = TRUE;
11         }
12 }
```

DEĞİŞKENLER

`v = 2` `w = &oge(6)`

ÇIKTI

```
0 1 2 3 4 5 6
```

Kod İzleme: `bfs(0)`



Kuyruk boş değil: öndeki **3** alınır; şimdi 3'ün listesi taranacak.

Adım 11 / 12

```
1  on = arka = NULL;
2  printf("%d ", v);  ugranildi[v] = TRUE;
3  kuyruk_ekle(&on, &arka, v);
▶  while (on) {
▶      v = kuyruk_al(&on, &arka);
6      for (w = baslik[v]; w; w = w->bag)
7          if (!ugranildi[w->dugum]) {
8              printf("%d ", w->dugum);
9              kuyruk_ekle(&on, &arka, w->dugum);
10             ugranildi[w->dugum] = TRUE;
11         }
12 }
```

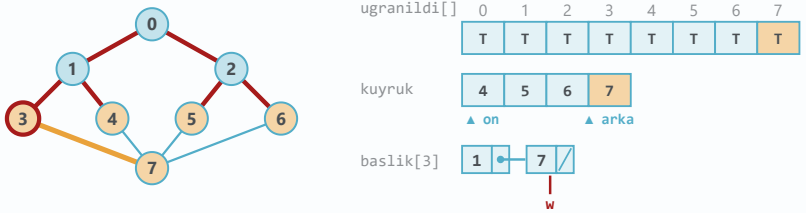
DEĞİŞKENLER

`v = 3` `w = –`

ÇIKTI

```
0 1 2 3 4 5 6
```

Kod İzleme: `bfs(0)`



w → 7: ziyaret edilmemiş → yazılır, kuyruğun **arkasına** eklenir, işaretlenir. (1 zaten işaretliydi.)

Adım 12 / 12

```
 1 on = arka = NULL;
 2 printf("%d ", v);  ugranildi[v] = TRUE;
 3 kuyruk_ekle(&on, &arka, v);
 4 while (on) {
 5     v = kuyruk_al(&on, &arka);
 6     for (w = baslik[v]; w; w = w->bag)
▶          if (!ugranildi[w->dugum]) {
▶              printf("%d ", w->dugum);
▶              kuyruk_ekle(&on, &arka, w->dugum);
▶              ugranildi[w->dugum] = TRUE;
11         }
12 }
```

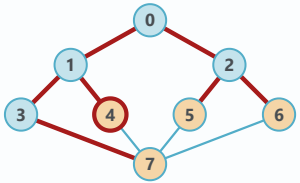
DEĞİŞKENLER

`v = 3` `w = &oge(7)`

ÇIKTI

```
0 1 2 3 4 5 6 7
```

Kod İzleme: `bfs(0)` (devam)





Kuyruk boş değil: öndeki **4** alınır; şimdi 4'ün listesi taranacak.

Adım 1 / 5

```
1  on = arka = NULL;
2  printf("%d ", v);  ugranildi[v] = TRUE;
3  kuyruk_ekle(&on, &arka, v);
▶  while (on) {
▶      v = kuyruk_al(&on, &arka);
6      for (w = baslik[v]; w; w = w->bag)
7          if (!ugranildi[w->dugum]) {
8              printf("%d ", w->dugum);
9              kuyruk_ekle(&on, &arka, w->dugum);
10             ugranildi[w->dugum] = TRUE;
11         }
12 }
```

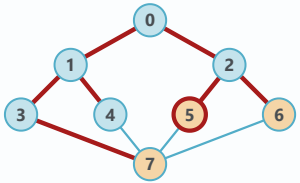
DEĞİŞKENLER

`v = 4` `w = –`

ÇIKTI

```
0 1 2 3 4 5 6 7
```

Kod İzleme: `bfs(0)` (devam)





Kuyruk boş değil: öndeki **5** alınır; şimdi 5'in listesi taranacak.

Adım 2 / 5

```
1  on = arka = NULL;
2  printf("%d ", v);  ugranildi[v] = TRUE;
3  kuyruk_ekle(&on, &arka, v);
▶  while (on) {
▶      v = kuyruk_al(&on, &arka);
6      for (w = baslik[v]; w; w = w->bag)
7          if (!ugranildi[w->dugum]) {
8              printf("%d ", w->dugum);
9              kuyruk_ekle(&on, &arka, w->dugum);
10             ugranildi[w->dugum] = TRUE;
11         }
12 }
```

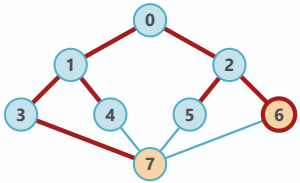
DEĞİŞKENLER

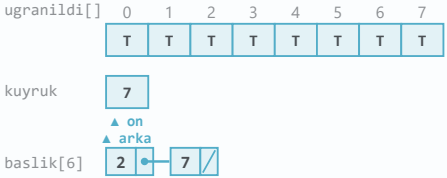
`v = 5` `w = –`

ÇIKTI

```
0 1 2 3 4 5 6 7
```

Kod İzleme: `bfs(0)` (devam)





Kuyruk boş değil: öndeki **6** alınır; şimdi 6'nın listesi taranacak.

Adım 3 / 5

```
1 on = arka = NULL;
2 printf("%d ", v);  ugranildi[v] = TRUE;
3 kuyruk_ekle(&on, &arka, v);
▶ while (on) {
▶     v = kuyruk_al(&on, &arka);
6     for (w = baslik[v]; w; w = w->bag)
7         if (!ugranildi[w->dugum]) {
8             printf("%d ", w->dugum);
9             kuyruk_ekle(&on, &arka, w->dugum);
10            ugranildi[w->dugum] = TRUE;
11        }
12 }
```

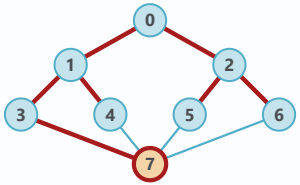
DEĞİŞKENLER

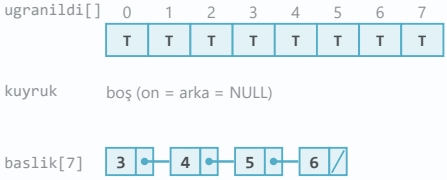
`v = 6` `w = –`

ÇIKTI

```
0 1 2 3 4 5 6 7
```

Kod İzleme: `bfs(0)` (devam)





Kuyruk boş değil: öndeki **7** alınır; şimdi 7'nin listesi taranacak.

Adım 4 / 5

```
1  on = arka = NULL;
2  printf("%d ", v);  ugranildi[v] = TRUE;
3  kuyruk_ekle(&on, &arka, v);
▶  while (on) {
▶      v = kuyruk_al(&on, &arka);
6      for (w = baslik[v]; w; w = w->bag)
7          if (!ugranildi[w->dugum]) {
8              printf("%d ", w->dugum);
9              kuyruk_ekle(&on, &arka, w->dugum);
10             ugranildi[w->dugum] = TRUE;
11         }
12 }
```

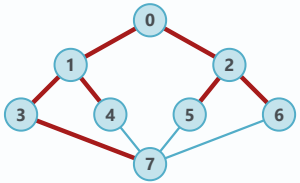
DEĞİŞKENLER

`v = 7` `w = –`

ÇIKTI

```
0 1 2 3 4 5 6 7
```

Kod İzleme: `bfs(0)` (devam)



ugranildi[]

0	1	2	3	4	5	6	7
T	T	T	T	T	T	T	T

kuyruk — boş (on = arka = NULL)

Kuyruk boşaldı → arama biter. Sıra **0 1 2 3 4 5 6 7**: düzey düzey. Turuncu kenarlar önce genişliğine kapsayan ağaç.

Adım 5 / 5

```
1  on = arka = NULL;
2  printf("%d ", v);  ugranildi[v] = TRUE;
3  kuyruk_ekle(&on, &arka, v);
▶  while (on) {
5      v = kuyruk_al(&on, &arka);
6      for (w = baslik[v]; w; w = w->bag)
7          if (!ugranildi[w->dugum]) {
8              printf("%d ", w->dugum);
9              kuyruk_ekle(&on, &arka, w->dugum);
10             ugranildi[w->dugum] = TRUE;
11         }
12 }
```

DEĞİŞKENLER

`v = 7` `w = NULL`

ÇIKTI

```
0 1 2 3 4 5 6 7
```

BFS: Analiz ve İlerisi

Analiz

Her düğüm kuyruğa **1 kez** girer → while n kez döner. Listelerde toplam 2e öğe taranır: $d_0 + \ldots + d_{n-1} = 2e$. Toplam **O(n + e)**; matrisle **$O(n^2)$**.

Ne verir?

Başlangıca olan **kenar sayısı** (düzey) sırasıyla ziyaret; ağırlıksız çizgede en kısa yol ağacı.

İlerisi

Önce derinliğine/genişliğine arama kavramları; yapay zekâda arama uzayını ağaca benzetip gezinmede, min-max, B* araması, dal-sınır (*branch and bound*) konularında geçer.

DFS mi BFS mi?

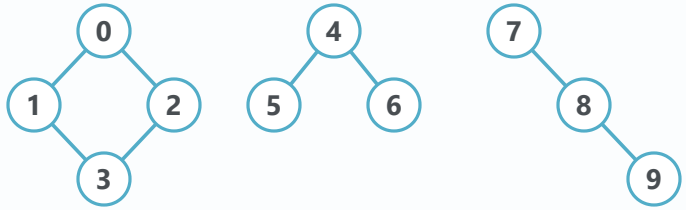
DFS: yığın/özyineleme, derin gider (bileşen, döngü, topolojik düşünme). BFS: kuyruk, katman katman (en az kenarlı yol).

Hocadan: "Oyun problemlerinde de "Hamleler şunlar olabilir." deniyorsa genişliğine açılıyor demektir."

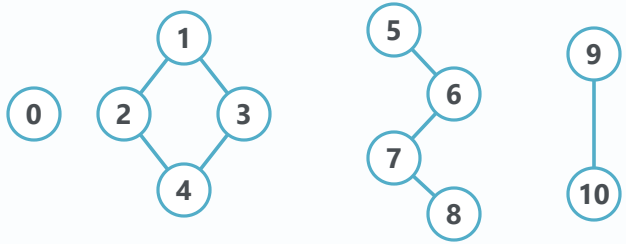
Bağlantılı Bileşenler (Connected Components)

```
void bitisik_bilesenler(int n)
{
    int i;
    for (i = 0; i < n; i++)
        if (!ugranildi[i]) {   /* yeni */
            dfs(i);
            printf("\n");
        }
}
```

Yönsüz çizge bağlantılı mı, kaç bileşenli? DFS (ya da BFS) her çağrıldığında bir bileşeni tümüyle işaretler; her yeni satırda bir sayaç artırılırsa **bileşen sayısı** da bulunur.



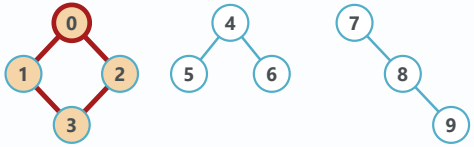
Çıktı: 0 1 3 2 / 4 5 6 / 7 8 9



Çıktı: 0 / 1 2 4 3 / 5 6 7 8 / 9 10

2. satırın çıktısı **1 2 4 3**'tür; 5 sonraki bileşendedir.

Kod İzleme: `bitisik_bilesenler(10)`



ugranildi[]

0	1	2	3	4	5	6	7	8	9
T	T	T	T	F	F	F	F	F	F

bileşen sayısı: 1

i = 0: işaretsiz → **yeni bileşen**; `dfs(0)` gezer: 0 1 3 2.

Adım 1 / 14

```
1 void bitisik_bilesenler(int n)
2 {
3     int i;
4     for (i = 0; i < n; i++)
▶         if (!ugranildi[i]) {   /* yeni */
▶             dfs(i);
7             printf("\n");
8         }
9 }
```

DEĞİŞKENLER

`n = 10` `i = 0`

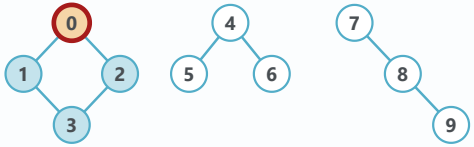
ÇAĞRI YIĞINI

`dfs(0)` `bitisik_bilesenler(10)` `main()`

ÇIKTI

```
0 1 3 2
```

Kod İzleme: `bitisik_bilesenler(10)`



ugranildi[]

0	1	2	3	4	5	6	7	8	9
T	T	T	T	F	F	F	F	F	F

bileşen sayısı: 1

dfs döndü; yeni satıra geçilir.

Adım 2 / 14

```
1 void bitisik_bilesenler(int n)
2 {
3     int i;
4     for (i = 0; i < n; i++)
5         if (!ugranildi[i]) {   /* yeni */
6             dfs(i);
▶             printf("\n");
8         }
9 }
```

DEĞİŞKENLER

`n = 10` `i = 0`

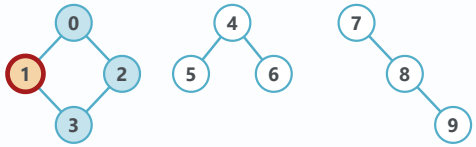
ÇAĞRI YIĞINI

`bitisik_bilesenler(10)` `main()`

ÇIKTI

```
0 1 3 2
```

Kod İzleme: `bitisik_bilesenler(10)`



ugranildi[]

0	1	2	3	4	5	6	7	8	9
T	T	T	T	F	F	F	F	F	F

bileşen sayısı: 1

i = 1: zaten işaretli (bir önceki DFS'de gezildi) → if'e girilmez, **O(1)**.

Adım 3 / 14

```
1 void bitisik_bilesenler(int n)
2 {
3     int i;
▶     for (i = 0; i < n; i++)
▶         if (!ugranildi[i]) {   /* yeni */
6             dfs(i);
7             printf("\n");
8         }
9 }
```

DEĞİŞKENLER

`n = 10` `i = 1`

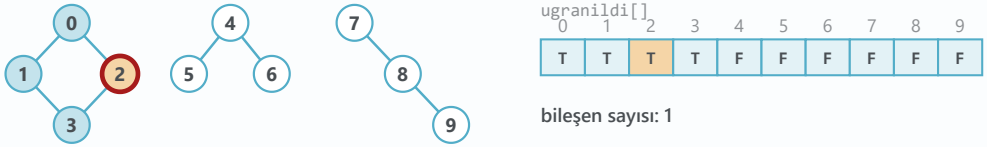
ÇAĞRI YIĞINI

`bitisik_bilesenler(10)` `main()`

ÇIKTI

```
0 1 3 2
```

Kod İzleme: `bitisik_bilesenler(10)`



i = 2: zaten işaretli (bir önceki DFS'de gezildi) → if'e girilmez, **O(1)**.

Adım 4 / 14

```
1 void bitisik_bilesenler(int n)
2 {
3     int i;
▶     for (i = 0; i < n; i++)
▶         if (!ugranildi[i]) {   /* yeni */
6             dfs(i);
7             printf("\n");
8         }
9 }
```

DEĞİŞKENLER

`n = 10` `i = 2`

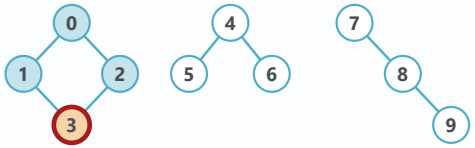
ÇAĞRI YIĞINI

`bitisik_bilesenler(10)` `main()`

ÇIKTI

```
0 1 3 2
```

Kod İzleme: `bitisik_bilesenler(10)`



ugranildi[]

0	1	2	3	4	5	6	7	8	9
T	T	T	T	F	F	F	F	F	F

bileşen sayısı: 1

i = 3: zaten işaretli (bir önceki DFS'de gezildi) → if'e girilmez, **O(1)**.

Adım 5 / 14

```
1 void bitisik_bilesenler(int n)
2 {
3     int i;
▶     for (i = 0; i < n; i++)
▶         if (!ugranildi[i]) {   /* yeni */
6             dfs(i);
7             printf("\n");
8         }
9 }
```

DEĞİŞKENLER

`n = 10` `i = 3`

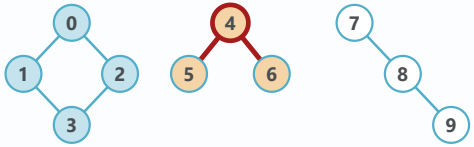
ÇAĞRI YIĞINI

`bitisik_bilesenler(10)` `main()`

ÇIKTI

```
0 1 3 2
```

Kod İzleme: `bitisik_bilesenler(10)`



ugranildi[]

0	1	2	3	4	5	6	7	8	9
T	T	T	T	T	T	T	F	F	F

bileşen sayısı: 2

i = 4: işaretsiz → **yeni bileşen**; `dfs(4)` gezer: 4 5 6.

Adım 6 / 14

```
1 void bitisik_bilesenler(int n)
2 {
3     int i;
4     for (i = 0; i < n; i++)
▶         if (!ugranildi[i]) {   /* yeni */
▶             dfs(i);
7             printf("\n");
8         }
9 }
```

DEĞİŞKENLER

`n = 10` `i = 4`

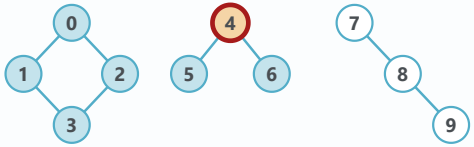
ÇAĞRI YIĞINI

`dfs(4)` `bitisik_bilesenler(10)` `main()`

ÇIKTI

```
0 1 3 2
4 5 6
```

Kod İzleme: `bitisik_bilesenler(10)`



ugranildi[]

0	1	2	3	4	5	6	7	8	9
T	T	T	T	T	T	T	F	F	F

bileşen sayısı: 2

dfs döndü; yeni satıra geçilir.

Adım 7 / 14

```
1 void bitisik_bilesenler(int n)
2 {
3     int i;
4     for (i = 0; i < n; i++)
5         if (!ugranildi[i]) {   /* yeni */
6             dfs(i);
▶             printf("\n");
8         }
9 }
```

DEĞİŞKENLER

`n = 10` `i = 4`

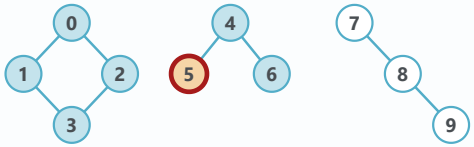
ÇAĞRI YIĞINI

`bitisik_bilesenler(10)` `main()`

ÇIKTI

```
0 1 3 2
4 5 6
```

Kod İzleme: `bitisik_bilesenler(10)`



ugranildi[]

0	1	2	3	4	5	6	7	8	9
T	T	T	T	T	T	T	F	F	F

bileşen sayısı: 2

i = 5: zaten işaretli (bir önceki DFS'de gezildi) → if'e girilmez, **O(1)**.

Adım 8 / 14

```
1 void bitisik_bilesenler(int n)
2 {
3     int i;
▶     for (i = 0; i < n; i++)
▶         if (!ugranildi[i]) {   /* yeni */
6             dfs(i);
7             printf("\n");
8         }
9 }
```

DEĞİŞKENLER

`n = 10` `i = 5`

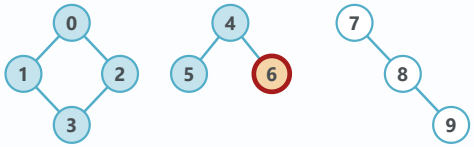
ÇAĞRI YIĞINI

`bitisik_bilesenler(10)` `main()`

ÇIKTI

```
0 1 3 2
4 5 6
```

Kod İzleme: `bitisik_bilesenler(10)`



ugranildi[]

0	1	2	3	4	5	6	7	8	9
T	T	T	T	T	T	T	F	F	F

bileşen sayısı: 2

i = 6: zaten işaretli (bir önceki DFS'de gezildi) → if'e girilmez, **O(1)**.

Adım 9 / 14

```
1 void bitisik_bilesenler(int n)
2 {
3     int i;
▶     for (i = 0; i < n; i++)
▶         if (!ugranildi[i]) {   /* yeni */
6             dfs(i);
7             printf("\n");
8         }
9 }
```

DEĞİŞKENLER

`n = 10` `i = 6`

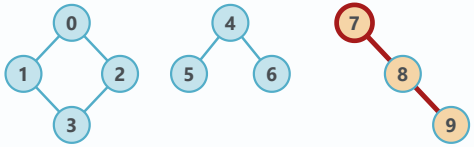
ÇAĞRI YIĞINI

`bitisik_bilesenler(10)` `main()`

ÇIKTI

```
0 1 3 2
4 5 6
```

Kod İzleme: `bitisik_bilesenler(10)`



ugranildi[]

0	1	2	3	4	5	6	7	8	9
T	T	T	T	T	T	T	T	T	T

bileşen sayısı: 3

i = 7: işaretsiz → **yeni bileşen**; `dfs(7)` gezer: 7 8 9.

Adım 10 / 14

```
1 void bitisik_bilesenler(int n)
2 {
3     int i;
4     for (i = 0; i < n; i++)
▶         if (!ugranildi[i]) {   /* yeni */
▶             dfs(i);
7             printf("\n");
8         }
9 }
```

DEĞİŞKENLER

`n = 10` `i = 7`

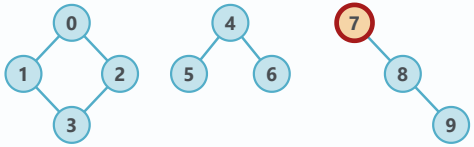
ÇAĞRI YIĞINI

`dfs(7)` `bitisik_bilesenler(10)` `main()`

ÇIKTI

```
0 1 3 2
4 5 6
7 8 9
```

Kod İzleme: `bitisik_bilesenler(10)`



ugranildi[]

0	1	2	3	4	5	6	7	8	9
T	T	T	T	T	T	T	T	T	T

bileşen sayısı: 3

dfs döndü; yeni satıra geçilir.

Adım 11 / 14

```
1 void bitisik_bilesenler(int n)
2 {
3     int i;
4     for (i = 0; i < n; i++)
5         if (!ugranildi[i]) {   /* yeni */
6             dfs(i);
▶             printf("\n");
8         }
9 }
```

DEĞİŞKENLER

`n = 10` `i = 7`

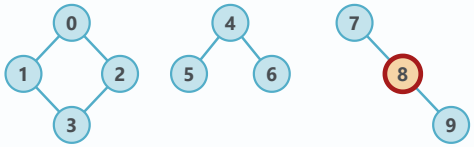
ÇAĞRI YIĞINI

`bitisik_bilesenler(10)` `main()`

ÇIKTI

```
0 1 3 2
4 5 6
7 8 9
```

Kod İzleme: `bitisik_bilesenler(10)`



ugranildi[]

0	1	2	3	4	5	6	7	8	9
T	T	T	T	T	T	T	T	T	T

bileşen sayısı: 3

i = 8: zaten işaretli (bir önceki DFS'de gezildi) → if'e girilmez, **O(1)**.

Adım 12 / 14

```
1 void bitisik_bilesenler(int n)
2 {
3     int i;
▶     for (i = 0; i < n; i++)
▶         if (!ugranildi[i]) {   /* yeni */
6             dfs(i);
7             printf("\n");
8         }
9 }
```

DEĞİŞKENLER

`n = 10` `i = 8`

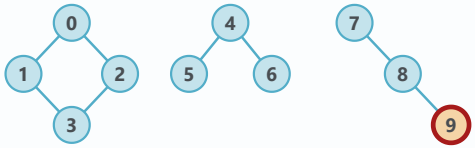
ÇAĞRI YIĞINI

`bitisik_bilesenler(10)` `main()`

ÇIKTI

```
0 1 3 2
4 5 6
7 8 9
```

Kod İzleme: `bitisik_bilesenler(10)`



ugranildi[]

0	1	2	3	4	5	6	7	8	9
T	T	T	T	T	T	T	T	T	T

bileşen sayısı: 3

i = 9: zaten işaretli (bir önceki DFS'de gezildi) → if'e girilmez, **O(1)**.

Adım 13 / 14

```
1 void bitisik_bilesenler(int n)
2 {
3     int i;
▶     for (i = 0; i < n; i++)
▶         if (!ugranildi[i]) {   /* yeni */
6             dfs(i);
7             printf("\n");
8         }
9 }
```

DEĞİŞKENLER

`n = 10` `i = 9`

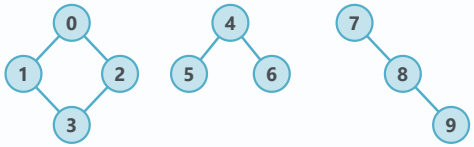
ÇAĞRI YIĞINI

`bitisik_bilesenler(10)` `main()`

ÇIKTI

```
0 1 3 2
4 5 6
7 8 9
```

Kod İzleme: `bitisik_bilesenler(10)`



ugranildi[]

0	1	2	3	4	5	6	7	8	9
T	T	T	T	T	T	T	T	T	T

bileşen sayısı: 3

Döngü bitti: **3 bileşen**. Her düğüm ve her liste toplamda bir kez işlendi.

Adım 14 / 14

```
1 void bitisik_bilesenler(int n)
2 {
3     int i;
▶     for (i = 0; i < n; i++)
5         if (!ugranildi[i]) {   /* yeni */
6             dfs(i);
7             printf("\n");
8         }
9 }
```

DEĞİŞKENLER

`n = 10` `i = 10`

ÇAĞRI YIĞINI

`bitisik_bilesenler(10)` `main()`

ÇIKTI

```
0 1 3 2
4 5 6
7 8 9
```

Analiz: n × O(n + e) mi?

Dıştaki for n kez dönüyor, içteki `dfs()` O(n + e) idi. Karmaşıklık ikisinin çarpımı mı, $O(n^2)$ mi olur?

> **Hocadan:** "Olur mu? Olur mu? Olur mu? O kadar çalışır mı? Hayır! … "Hepsini çarparım, sonucu söylerim." Değil!"

Neden O(n + e)?

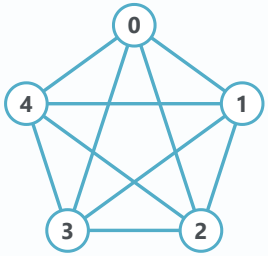
Tek bileşen varsa ilk turda O(n + e) çalışır ve tüm `ugranildi[]` TRUE olur; i'nin diğer değerlerinde if'e girilmez. Bileşenler ayrık olduğundan toplam iş **O(n + e)**.

Matrisle olsaydı?

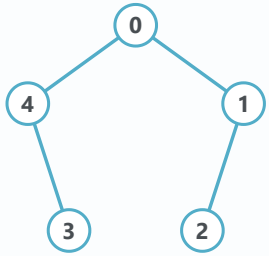
Her dfs çağrısı komşuları O(n)'de arar → bileşen bulma **$O(n^2)$**. "O da etkili yazılırsa…"

Teorem: Bağlantılı Çizgede En Az n − 1 Kenar

- **Teorem:** Çizgenin bağlantılı olması için en az **n − 1** kenar bulunmalıdır.
- **Tek yönlüdür:** n − 1 kenar *gerekli* ama *yeterli değil*; n − 1 kenarlı çizge bağlantılı olabilir de olmayabilir de. Daha azsa kesinlikle bağlantılı değildir.
- En fazla: tüm düğümler birbirine komşuysa (tam çizge K_n) (n − 1) + (n − 2) + … + 1 = **n(n − 1)/2**. Demek ki e, n^2'li bir ifade olabilir.
- Bağlantılı çizgede: **n − 1 ≤ e ≤ n(n − 1)/2**.



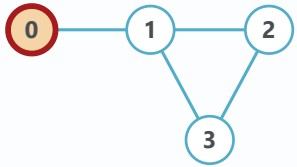
K5: 10 kenar

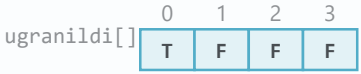


n − 1 = 4 kenar, bağlantılı

Hocadan: "Tic tac toe oyunu… "Rakibin hamlesini nasıl bozarım?" Okununca bu kavramlar daha iyi oturur, tahmin ediyorum."

Kod İzleme: yönsüz çizgede döngü var mı?





`dongu_var(0, -1)` : 0 işaretlenir.

Adım 1 / 7

```
1  bool dongu_var(int v, int ata)
2  {
3      oge_gost w;
▶      ugranildi[v] = TRUE;
5      for (w = baslik[v]; w; w = w->bag)
6          if (!ugranildi[w->dugum]) {
7              if (dongu_var(w->dugum, v)) return TRUE;
8          } else if (w->dugum != ata)
9              return TRUE;          /* geri kenar */
10     return FALSE;
11 }
```

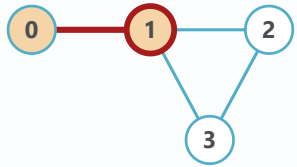
DEĞİŞKENLER

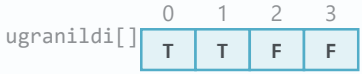
`v = 0` `ata = -1`

ÇAĞRI YIĞINI

`dongu_var(0, -1)` `main()`

Kod İzleme: yönsüz çizgede döngü var mı?





`dongu_var(1, 0)` : 1 işaretlenir.

Adım 2 / 7

```
1  bool dongu_var(int v, int ata)
2  {
3      oge_gost w;
▶      ugranildi[v] = TRUE;
5      for (w = baslik[v]; w; w = w->bag)
6          if (!ugranildi[w->dugum]) {
7              if (dongu_var(w->dugum, v)) return TRUE;
8          } else if (w->dugum != ata)
9              return TRUE;          /* geri kenar */
10     return FALSE;
11 }
```

DEĞİŞKENLER

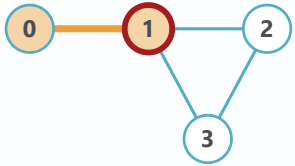
`v = 1` `ata = 0`

ÇAĞRI YIĞINI

`dongu_var(1, 0)`

`dongu_var(0, -1)` `main()`

Kod İzleme: yönsüz çizgede döngü var mı?



ugranildi[]

0	1	2	3
T	T	F	F

w → 0: işaretli ama 1'in ebeveyni; geldiğimiz kenar, döngü sayılmaz.

Adım 3 / 7

```
1  bool dongu_var(int v, int ata)
2  {
3      oge_gost w;
4      ugranildi[v] = TRUE;
5      for (w = baslik[v]; w; w = w->bag)
6          if (!ugranildi[w->dugum]) {
7              if (dongu_var(w->dugum, v)) return TRUE;
▶          } else if (w->dugum != ata)
9              return TRUE;          /* geri kenar */
10     return FALSE;
11 }
```

DEĞİŞKENLER

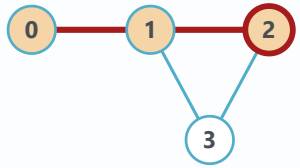
`v = 1` `ata = 0` `w = &oge(0)`

ÇAĞRI YIĞINI

`dongu_var(1, 0)`

`dongu_var(0, -1)` `main()`

Kod İzleme: yönsüz çizgede döngü var mı?



ugranildi[]	0	1	2	3
	T	T	T	F

`dongu_var(2, 1)` : 2 işaretlenir.

Adım 4 / 7

```
1  bool dongu_var(int v, int ata)
2  {
3      oge_gost w;
▶      ugranildi[v] = TRUE;
5      for (w = baslik[v]; w; w = w->bag)
6          if (!ugranildi[w->dugum]) {
7              if (dongu_var(w->dugum, v)) return TRUE;
8          } else if (w->dugum != ata)
9              return TRUE;          /* geri kenar */
10     return FALSE;
11 }
```

DEĞİŞKENLER

`v = 2` `ata = 1`

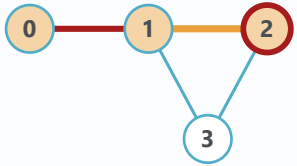
ÇAĞRI YIĞINI

`dongu_var(2, 1)`

`dongu_var(1, 0)`

`dongu_var(0, -1)` `main()`

Kod İzleme: yönsüz çizgede döngü var mı?



ugranildi[]

0	1	2	3
T	T	T	F

w → 1: işaretli ama 2'nin ebeveyni; geldiğimiz kenar, döngü sayılmaz.

Adım 5 / 7

```
1  bool dongu_var(int v, int ata)
2  {
3      oge_gost w;
4      ugranildi[v] = TRUE;
5      for (w = baslik[v]; w; w = w->bag)
6          if (!ugranildi[w->dugum]) {
7              if (dongu_var(w->dugum, v)) return TRUE;
▶          } else if (w->dugum != ata)
9              return TRUE;          /* geri kenar */
10     return FALSE;
11 }
```

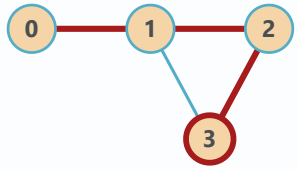
DEĞİŞKENLER

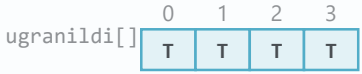
`v = 2` `ata = 1` `w = &oge(1)`

ÇAĞRI YIĞINI

- `dongu_var(2, 1)`
- `dongu_var(1, 0)`
- `dongu_var(0, -1)` `main()`

Kod İzleme: yönsüz çizgede döngü var mı?





`dongu_var(3, 2)` : 3 işaretlenir.

Adım 6 / 7

```
1  bool dongu_var(int v, int ata)
2  {
3      oge_gost w;
▶      ugranildi[v] = TRUE;
5      for (w = baslik[v]; w; w = w->bag)
6          if (!ugranildi[w->dugum]) {
7              if (dongu_var(w->dugum, v)) return TRUE;
8          } else if (w->dugum != ata)
9              return TRUE;          /* geri kenar */
10     return FALSE;
11 }
```

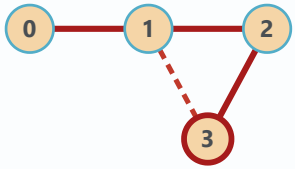
DEĞİŞKENLER

`v = 3` `ata = 2`

ÇAĞRI YIĞINI

- `dongu_var(3, 2)`
- `dongu_var(2, 1)`
- `dongu_var(1, 0)`
- `dongu_var(0, -1)` `main()`

Kod İzleme: yönsüz çizgede döngü var mı?



ugranildi[]	0	1	2	3
	T	T	T	T

w → 1: işaretli **ve** ebeveyn (2) değil → **geri kenar** (3,1): döngü var! TRUE yukarı taşınır.

Adım 7 / 7

```
1  bool dongu_var(int v, int ata)
2  {
3      oge_gost w;
4      ugranildi[v] = TRUE;
5      for (w = baslik[v]; w; w = w->bag)
6          if (!ugranildi[w->dugum]) {
7              if (dongu_var(w->dugum, v)) return TRUE;
▶          } else if (w->dugum != ata)
▶              return TRUE;          /* geri kenar */
10     return FALSE;
11 }
```

DEĞİŞKENLER

`v = 3` `ata = 2` `w = &oge(1)`

ÇAĞRI YIĞINI

- `dongu_var(3, 2)`
- `dongu_var(2, 1)`
- `dongu_var(1, 0)`
- `dongu_var(0, -1)` `main()`

Döngü Tespiti: Yönsüz ve Yönlü

Yönsüz

DFS'de ziyaretli ama **ebeveyn olmayan** bir komşu = geri kenar = döngü. O(n + e).

Yönlü

"Şu an yığında olan" (gri) düğüme giden kenar döngüdür; üç renk: beyaz/gri/siyah. Topolojik sıralama yordamı da döngüyü yakalar.

Neden önemli?

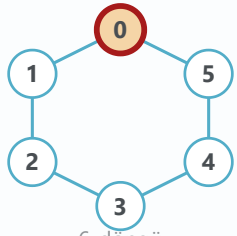
Önceliği olan işlerde döngü, projenin hiç bitmemesi demektir; MST'de döngü, fazladan kenar demektir.

Matrisle

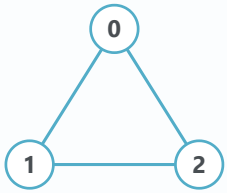
Aynı DFS matrisle $O(n^2)$ olur.

İki Parçalı (Bipartite) Çizge Testi

- V, iki kümeye ayrılabilsin ve her kenar **farklı** kümeleri bağlasın: iki parçalı çizge (ör. $K_{3,3}$, öğrenci–ders).
- Test: BFS ile iki renge boyanır; komşu ters renge. Aynı renkli bir kenar çıkarsa değildir.
- Teorem: çizge iki parçalıdır ⇔ **tek uzunlukta döngü** yoktur.
- Maliyet BFS ile aynı: **O(n + e)**; her bileşen için ayrı başlatılır.



6-döngü



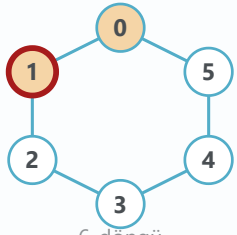
üçgen

BFS: 0 boyandı (renk 0). Komşu, kendisinin **tersi** renge boyanır.

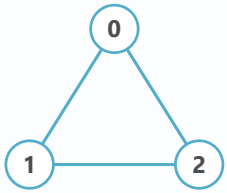
Adım 1 / 10

İki Parçalı (Bipartite) Çizge Testi

- V, iki kümeye ayrılabilsin ve her kenar **farklı** kümeleri bağlasın: iki parçalı çizge (ör. $K_{3,3}$, öğrenci–ders).
- Test: BFS ile iki renge boyanır; komşu ters renge. Aynı renkli bir kenar çıkarsa değildir.
- Teorem: çizge iki parçalıdır ⇔ **tek uzunlukta döngü** yoktur.
- Maliyet BFS ile aynı: **O(n + e)**; her bileşen için ayrı başlatılır.



6-döngü

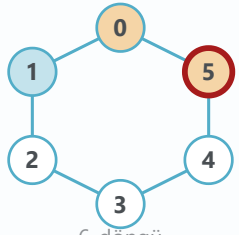


üçgen

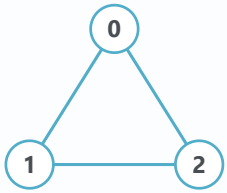
BFS: 1 boyandı (renk 1). Komşu, kendisinin **tersi** renge boyanır.

İki Parçalı (Bipartite) Çizge Testi

- V, iki kümeye ayrılabilsin ve her kenar **farklı** kümeleri bağlasın: iki parçalı çizge (ör. $K_{3,3}$, öğrenci–ders).
- Test: BFS ile iki renge boyanır; komşu ters renge. Aynı renkli bir kenar çıkarsa değildir.
- Teorem: çizge iki parçalıdır ⇔ **tek uzunlukta döngü** yoktur.
- Maliyet BFS ile aynı: **O(n + e)**; her bileşen için ayrı başlatılır.



6-döngü

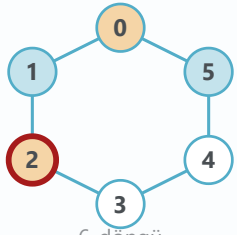


üçgen

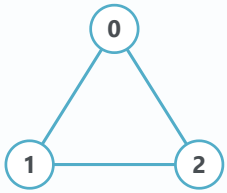
BFS: 5 boyandı (renk 1). Komşu, kendisinin **tersi** renge boyanır.

İki Parçalı (Bipartite) Çizge Testi

- V, iki kümeye ayrılabilsin ve her kenar **farklı** kümeleri bağlasın: iki parçalı çizge (ör. $K_{3,3}$, öğrenci–ders).
- Test: BFS ile iki renge boyanır; komşu ters renge. Aynı renkli bir kenar çıkarsa değildir.
- Teorem: çizge iki parçalıdır ⇔ **tek uzunlukta döngü** yoktur.
- Maliyet BFS ile aynı: **O(n + e)**; her bileşen için ayrı başlatılır.



6-döngü

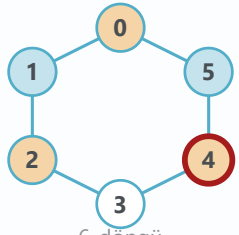


üçgen

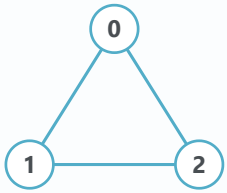
BFS: 2 boyandı (renk 0). Komşu, kendisinin **tersi** renge boyanır.

İki Parçalı (Bipartite) Çizge Testi

- V, iki kümeye ayrılabilsin ve her kenar **farklı** kümeleri bağlasın: iki parçalı çizge (ör. $K_{3,3}$, öğrenci–ders).
- Test: BFS ile iki renge boyanır; komşu ters renge. Aynı renkli bir kenar çıkarsa değildir.
- Teorem: çizge iki parçalıdır ⇔ **tek uzunlukta döngü** yoktur.
- Maliyet BFS ile aynı: **O(n + e)**; her bileşen için ayrı başlatılır.



6-döngü

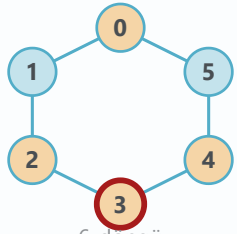


üçgen

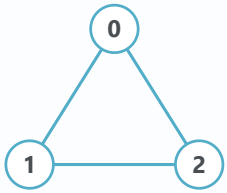
BFS: 4 boyandı (renk 0). Komşu, kendisinin **tersi** renge boyanır.

İki Parçalı (Bipartite) Çizge Testi

- V, iki kümeye ayrılabilsin ve her kenar **farklı** kümeleri bağlasın: iki parçalı çizge (ör. $K_{3,3}$, öğrenci–ders).
- Test: BFS ile iki renge boyanır; komşu ters renge. Aynı renkli bir kenar çıkarsa değildir.
- Teorem: çizge iki parçalıdır ⇔ **tek uzunlukta döngü** yoktur.
- Maliyet BFS ile aynı: **O(n + e)**; her bileşen için ayrı başlatılır.



6-döngü

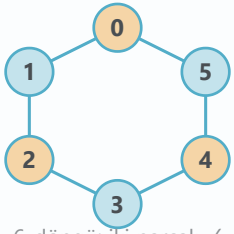


üçgen

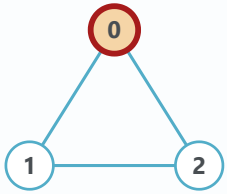
BFS: 3 boyandı (renk 1). Komşu, kendisinin **tersi** renge boyanır.

İki Parçalı (Bipartite) Çizge Testi

- V, iki kümeye ayrılabilsin ve her kenar **farklı** kümeleri bağlasın: iki parçalı çizge (ör. $K_{3,3}$, öğrenci–ders).
- Test: BFS ile iki renge boyanır; komşu ters renge. Aynı renkli bir kenar çıkarsa değildir.
- Teorem: çizge iki parçalıdır ⇔ **tek uzunlukta döngü** yoktur.
- Maliyet BFS ile aynı: **O(n + e)**; her bileşen için ayrı başlatılır.



6-döngü: iki parçalı ✓

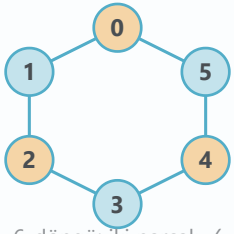


üçgen

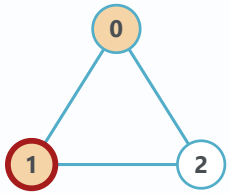
Üçgen: 0 boyandı.

İki Parçalı (Bipartite) Çizge Testi

- V, iki kümeye ayrılabilsin ve her kenar **farklı** kümeleri bağlasın: iki parçalı çizge (ör. $K_{3,3}$, öğrenci–ders).
- Test: BFS ile iki renge boyanır; komşu ters renge. Aynı renkli bir kenar çıkarsa değildir.
- Teorem: çizge iki parçalıdır ⇔ **tek uzunlukta döngü** yoktur.
- Maliyet BFS ile aynı: **O(n + e)**; her bileşen için ayrı başlatılır.



6-döngü: iki parçalı ✓

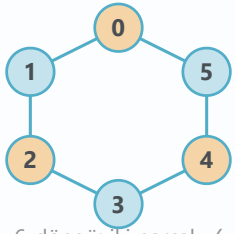


üçgen

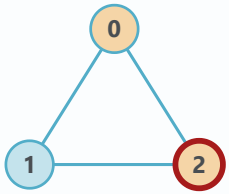
Üçgen: 1 boyandı.

İki Parçalı (Bipartite) Çizge Testi

- V, iki kümeye ayrılabilsin ve her kenar **farklı** kümeleri bağlasın: iki parçalı çizge (ör. $K_{3,3}$, öğrenci–ders).
- Test: BFS ile iki renge boyanır; komşu ters renge. Aynı renkli bir kenar çıkarsa değildir.
- Teorem: çizge iki parçalıdır ⇔ **tek uzunlukta döngü** yoktur.
- Maliyet BFS ile aynı: **O(n + e)**; her bileşen için ayrı başlatılır.



6-döngü: iki parçalı ✓

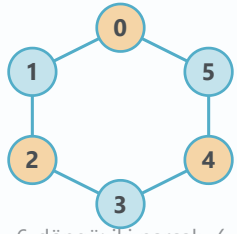


üçgen

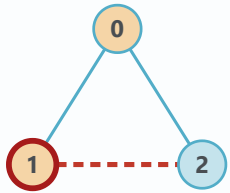
Üçgen: 2 boyandı.

İki Parçalı (Bipartite) Çizge Testi

- V, iki kümeye ayrılabilsin ve her kenar **farklı** kümeleri bağlasın: iki parçalı çizge (ör. $K_{3,3}$, öğrenci–ders).
- Test: BFS ile iki renge boyanır; komşu ters renge. Aynı renkli bir kenar çıkarsa değildir.
- Teorem: çizge iki parçalıdır ⇔ **tek uzunlukta döngü** yoktur.
- Maliyet BFS ile aynı: **O(n + e)**; her bileşen için ayrı başlatılır.



6-döngü: iki parçalı ✓



üçgen

Aynı renkli iki komşu → **iki parçalı değil** (tek uzunluklu döngü var).

ÖZET

Karmaşıklık Özeti

İşlem	Matris	Liste	Not
Bellek	$O(n^2)$	$O(n + e)$	yönsüz listede 2e öğe
Derece (yönsüz)	$O(n)$	$O(d)$	yönlüde giriş için ters liste
Kenar sayısı	$O(n^2)$	$O(n + e)$	Teorem 1: toplam / 2
DFS / BFS	$O(n^2)$	$O(n + e)$	her düğüm bir kez
Bağlantılı bileşenler	$O(n^2)$	$O(n + e)$	çarpım değil!
Döngü / iki parçalılık	$O(n^2)$	$O(n + e)$	DFS / BFS üzerine
Transpoz liste kurma	—	$O(n + e)$	bir kez gezinerek

n: düğüm sayısı · e: kenar sayısı · d: düğümün derecesi · $n - 1 \le e \le n(n - 1)/2$ (bağlantılı, basit çizge)

ÖZET

Sık Yapılan Hatalar

1. Yönlü çizgede <u,v> varken <v,u>'yu da var saymak; matrisi simetrik sanmak.
2. BFS'de düğümü kuyruğa eklerken değil, çıkarırken işaretlemek → aynı düğüm kuyruğa birden çok kez girer.
3. `ugranildi[]`'yi her gezinmeden önce FALSE yapmayı unutmak (global dizi!).
4. Dış for × DFS'yi çarpıp bileşenler için O(n(n + e)) demek.
5. Liste kurarken `malloc` sonucunu kontrol etmemek; programın sonunda listeleri `free` etmemek.
6. Kenarı sona ekleyip her seferinde listeyi baştan yürümek; O(1)'lik başa eklemeyi kaçırmak.

ÖZET

Alıştırmalar

1. Derece dizisi (3, 3, 2, 2, 1) olan basit yönsüz bir çizge var mıdır? Kaç kenarı vardır?
2. G4 için dizi ile sıralı gösterimde 5 numaralı düğümün komşularını hangi indisler tutar?
3. 8 düğümlü örnek çizgede DFS ve BFS'yi 7'den başlatın (listeler sıralı).
4. BFS kodunda `ugranildi[w→dugum] = TRUE` satırı `kuyruk_al`'dan sonraya taşınırsa ne olur?
5. Komşuluk matrisiyle verilen çizgenin bileşen sayısını bulan yordamı yazın; karmaşıklığı nedir?
6. Yönlü çizgede in-degree'yi O(1)'de verebilmek için hangi yapıyı, nasıl güncellersiniz?

▶ **Cevaplar / ipuçları**

ÖZET

Bölüm Özeti

Kavramlar

G = (V, E); yönlü/yönsüz, komşu, alt çizge, yol, derece; $\Sigma d = 2e$; bağlantılılık için $e \geq n - 1$.

Gösterimler

Matris $O(n^2)$; liste $O(n + e)$; sıralı dizi; ortogonal (yönlü); çok bağlı (her kenar tek öğe).

Gezinmeler

DFS = özyineleme/yığın; BFS = kuyruk. İkisi de listeyle $O(n + e)$, matrisle $O(n^2)$.

Uygulamalar

Bağlantılı bileşenler $O(n + e)$; döngü tespiti; iki parçalılık testi; kapsayan ağaçlara hazırlık.

Yöntem

Önce problem, sonra verinin nasıl depolanacağı düşünülür; ilk çözüme güvenilmez.

ÖZET

Kaynaklar

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2. baskı, Bölüm 6 (Graphs).
- T. H. Cormen ve ark., *Introduction to Algorithms*, 4. baskı, Bölüm 20 (Elementary Graph Algorithms).
- R. Sedgewick, *Algorithms in C, Part 5: Graph Algorithms*, 3. baskı.

← Bölüm 14: Hash Tabloları

Ders sayfası

Bölüm 16: Kümeler ve Union-Find →

Son

Bölüm 15: Çizgeler

Sonraki bölüm: Bölüm 16: Kümeler ve Union-Find · Ders sayfası