

Yığınlar ve Kuyruklar

CEN207 Veri Yapıları — Hafta 3

Dr. Öğr. Üyesi Uğur CORUH · 2026-2027 Güz

Bugünün planı (3 saat)

Saat	Konu
1	Yığın ADT · dizi yığını · taşma Anim 1–2 · bağlı yığın Anim 3
2	Yığın uygulamaları: parantez, postfix, prefix, infix Anim 4–8 · özyineleme Anim 9–10
3	Hanoi Anim 11 · kuyruk türevleri Anim 12–16 · kendini sınıama

Öğrenme çıktıları: ÖÇ.1 (temel veri yapılarını açıklama) · ÖÇ.7 (doğru yapıyı seçme)

Bu haftanın kavramları — nerede

Kavram	Nerede
Yığın ADT'si, LIFO, push/pop	Bölüm 1
İfade algoritmaları (postfix, infix)	Bölüm 2
Özyineleme, çağrı yığını, Hanoi Kulesi	Bölüm 3–4
Kuyruk ADT'si, FIFO, dairesel kuyruk, deque	Bölüm 5

Kod örnekleri nasıl çalışıyor

- Her fikrin eksiksiz bir **C** ve **Java** programı var
- C: `gcc -std=c11 -Wall -Wextra -o /tmp/x file.c && /tmp/x`
- Java: `javac -d /tmp/j File.java && java -cp /tmp/j File`
- Kaynaklar: `code/week-03/c/` ve `code/week-03/java/`
- Her programın beklenen çıktısı hafta notlarında var

Hafta 1–2'den köprü

- Hafta 1–2 size iki araç verdi: **işaretçiler/bellek** ve **bağlı liste**
- Bu hafta ikisini de, sürekli, yeni biçimlerde kullanıyor
- Bu haftanın yeni fikri: bir yapıya *nereden* dokunabileceğinizi kısıtlamak

Tekrar — Hafta 1: işaretçiler ve bellek

- `int *p` bir sayı değil, bir **adres** tutar
- `malloc` işletim sisteminden kutu ister; `free` geri verir
- `free` 'i unutmak → **bellek sızıntısı**
- `free` 'den sonra kullanmak → **sarkan işaretçi**

Tekrar — Hafta 2: bağlı listeler

- Bağlı liste, **düğüm**lerden oluşan bir zincirdir
- Her düğüm: bir değer + sonraki düğümü gösteren işaretçi
- Ekleme/çıkarma birkaç işaretçiyi yeniden bağlar — kaydırma yok
- Bugünün bağlı yığını/kuyruğu tam olarak bu fikri kullanır

Haftanın haritası — tek kural

- Yığın ve kuyruk ikisi de **doğrusal** yapılardır
- Bir **yığın** yalnız **bir** ucuna dokunmanıza izin verir
- Bir **kuyruk** sizi bir uçtan eklemeye, diğerinden çıkarmaya zorlar
- Bu tek kural farkı, her yapının ne için iyi olduğunu belirler

Haftanın haritası — tek bakışta

Yığın konuları	Kuyruk konuları
Dizi yığını, bağlı yığın	Dizi kuyruğu, dairesel kuyruk
Parantez, postfix, infix	Bağlı liste ile kuyruk
Özyineleme, çağrı yığını	Deque, çok seviyeli kuyruk
Hanoi Kulesi	—

1. Yığın: Son Giren, İlk Çıkar

Başlangıç sorusu

Art arda üç web sayfası ziyaret edin, sonra **Geri**'ye üç kez tıklayın.

Sırasıyla sayfa 2'ye, sonra sayfa 1'e inersiniz, sonra hiçbir yere.
Ziyaret edilen *son* sayfa, tekrar ziyaret edilen *ilk* sayfadır.

Kısa bir tarihçe

- 1950'lerin sonu: "yığın", "push", "pop" zaten CS literatüründe
- **1957** — Bauer ve Samelson (Münih) donanımsal bir yığın patentler
- **1946** — Turing, ACE bilgisayarında benzer bir fikir kullanmıştı
- Bilgisayar biliminin en eski fikirlerinden biri — hâlâ her programda

Sezgi — bir tabak yığını

- Bir tabağı yalnız **en üstten** alın
- Bir tabağı yalnız **en üste** koyun
- Üstündekileri kaldırmadan ortadan bir tabak çekemezsiniz
- En son konan tabak, çıkarılan ilk tabaktır — **LIFO**

Üç işlem

- **push** — en üste yeni bir eleman koy
- **pop** — en üstteki elemanı çıkar ve döndür
- **peek** (ya da **top**) — çıkarmadan en üste bak
- Üçü de yalnız **bir uca** dokunur

Yığın ADT'si

İşlem	Ne yapar	Ön koşul	Karmaşıklık
<code>push(x)</code>	<code>x</code> 'i en üste ekle	dolu değil (dizi)	$O(1)$
<code>pop()</code>	En üstü çıkar/döndür	boş değil	$O(1)$
<code>peek()</code>	En üstü döndür, tut	boş değil	$O(1)$
<code>isEmpty()</code>	sıfır eleman mı?	yok	$O(1)$

Dizi ile yığın — fikir

- Sabit boyutlu bir `data[CAP]` dizisi ayır
- Tek bir tamsayı tut, `top` : en üstteki hücrenin indisi
- Boş yığın: `top == -1` — "henüz üst yok"
- `push` : `top++` , yaz; `pop` : oku, `top--`

Array stack: push and pop



Bitti: 10 push, 4 pop. Yığında (alttan üste): 12, 7, 25, 3, 18, 9. Her işlem $O(1)$.

Kod — push()

```
bool push(int x) {  
    if (top == CAP - 1) return false;  
    top = top + 1;  
    data[top] = x;  
    return true;  
}
```

Kod — pop()

```
bool pop(int *out) {  
    if (top == -1) return false;  
    *out = data[top];  
    top = top - 1;  
    return true;  
}
```

Beklenen çıktı

```
push(12) -> true   [top = 0]
push(21) -> true   [top = 9]
pop()      -> 21    [top = 8]
pop()      -> 5     [top = 7]
pop()      -> 14    [top = 6]
pop()      -> 30    [top = 5]
```

Neden her işlem $O(1)$

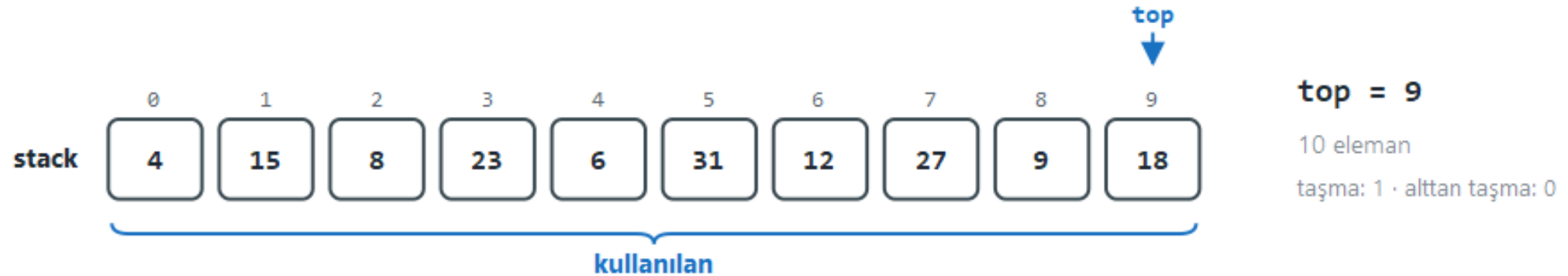
- `push` / `pop` , **sabit sayıda adım** atar
- Bir koşul denetimi, `top` 'un bir hareketi, bir dizi erişimi
- Yığında 1 eleman ya da bir milyon olsun fark etmez
- Hiçbir döngü diğer elemanlara dokunmaz

Taşma ve alttan taşma — soru

Bir dizinin **sabit** bir boyutu var. Dolu bir yığına **push**, boş bir yığından **pop** yapılırsa ne olur?

Doğru bir yığın **önce denetlemeli ve reddetmelidir**.

Stack overflow and underflow



çıkanlar =

Bitti: 10 başarılı push, 0 başarılı pop, 1 taşma, 0 alttan taşma. Yığında (alttan üste): 4, 15, 8, 23, 6, 31, 12, 27, 9, 18. İki basit if kontrolü, taşmayı ve alttan taşmayı önceden yakalar.

Sık yapılan hatalar (dizi yığını)

- `top == CAP` denetlemek, `top == CAP - 1` yerine
- Önce `isEmpty` denetlemeden pop yapmak
- `push` 'un `bool` dönüş değerini yok saymak

Mini soru

`top = -1` , "boş" için neden `top = 0` 'dan daha iyi bir seçim?

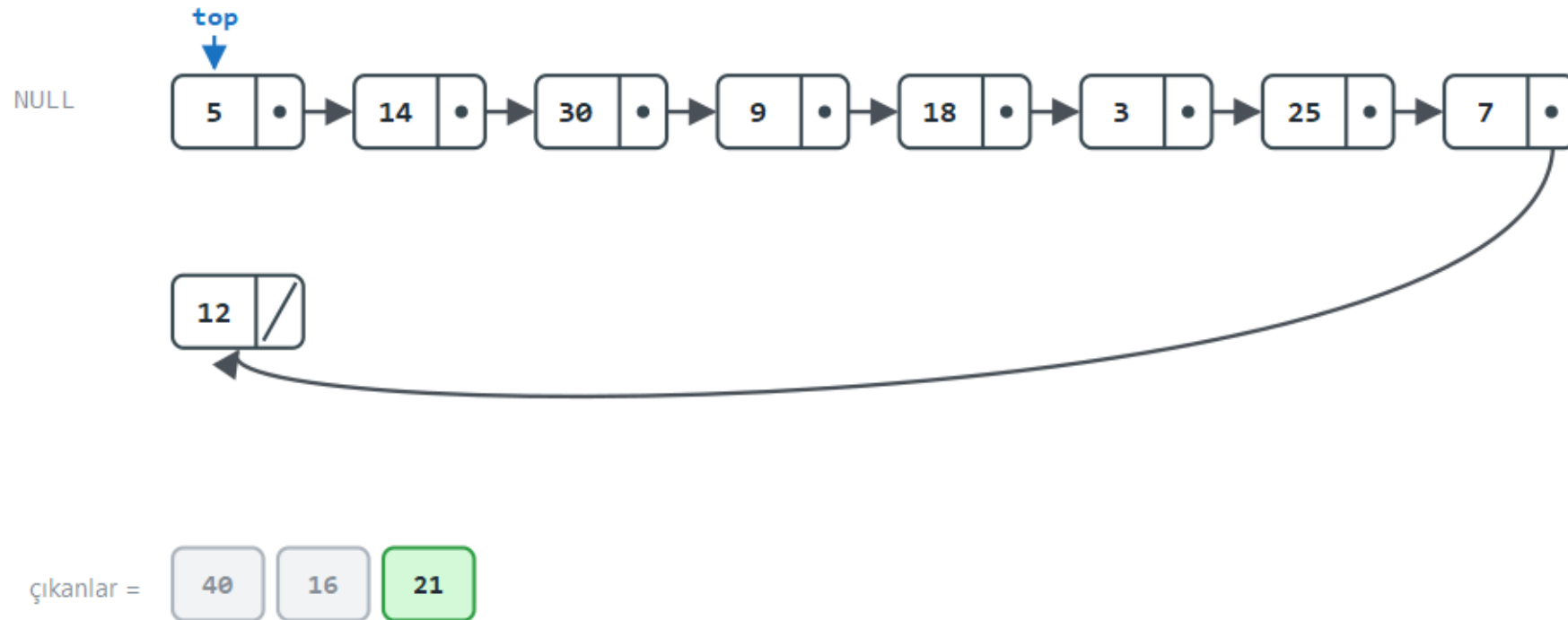
Yanıt

İndis `0` geçerli bir hücredir. `top = 0` "boş" anlamına gelseydi, boş bir yığınla indis 0'da tek elemanı olan bir yığını ayırt edemezsiniz.

Hiç taşmayan bir yığın

- Dizi ile yığının sert bir tavanı var: CAP
- **Bağlı liste ile yığın**: her eleman kendi düğümü
- **top** artık bir indis değil, bir **işaretçi**
- **push** ayırır; **pop** serbest bırakır

Linked-list stack: push and pop



Bitti: 12 push, 3 başarılı pop, 0 alttan taşma. Yığında (alttan üste): 12, 7, 25, 3, 18, 9, 30, 14, 5. Bağlı yığın bellek bitmedikçe taşmaz; bedeli her düğümdeki fazladan next işaretçisidir.

Kod — struct Node + push()

```
typedef struct Node {  
    int data;  
    struct Node *next;  
} Node;  
Node *top = NULL;  
  
void push(int x) {  
    Node *n = malloc(sizeof(Node));  
    n->data = x;  
    n->next = top;  
    top = n;  
}
```

Kod — pop()

```
bool pop(int *out) {  
    if (top == NULL) return false;  
    Node *tmp = top;  
    *out = tmp->data;  
    top = top->next;  
    free(tmp);  
    return true;  
}
```

push neden hâlâ $O(1)$

- Tek, sabit boyutlu düğüm için `malloc` sabit zamanlı
- Kaç düğüm zaten var olduğuna bağlı değil
- Var olan elemanlar üzerinde döngü yok
- Bedel: düğüm başına bir işaretçi (`next`), dizide sıfır

Sık yapılan hatalar (bağlı yığın)

- `pop` 'ta `free(tmp)` 'i unutmak — yavaş bellek sızıntısı
- `free` edilmiş bir işaretçi kullanmak — tanımsız davranış
- Java: eski bir referansı tutmak çöp toplamayı engeller

Dizi vs. bağlı liste yığını

	Dizi yığını	Bağlı yığın
Kapasite	Sabit, taşabilir	Bellekle sınırlı
Ekstra bellek	Yok	Düğüm başına 1 işaretçi
Önbellek davranışı	Bitişik, hızlı	Dağınık, daha yavaş

Mini soru

CAP = 8 . 5 push ve 2 pop'tan sonra top nedir?

Yanıt

$-1 + 5 - 2 = 2$. Üç eleman kalır
(indis 0, 1, 2), ve $top == 2$.

2. Yığın Uygulamaları: İfadeler

Infix bilgisayar için neden can sıkıcı

- $A + B * C$ — $*$ mi önce hesaplanır? **Öncelik** gerekir
- İki başka yazım, işleci öyle taşır ki değerlendirme sırasında hiçbir öncelik kuralı gerekmez

Kısa bir tarihçe

- 1920'ler — Polonyalı mantıkçı **Jan Łukasiewicz**, prefix ("Polish") ve postfix ("Reverse Polish") yazımı tanıtır
- **HP hesap makineleri** sayesinde ünlenir: parantez tuşu gerekmez
- Küçük bir yığın makinesiyle değerlendirilir — birazdan kuracağınız

Üç yazım

Yazım	İşlecin yeri	A + B * C
Infix	İşlenenler arasında	A + B * C
Postfix (RPN)	Her iki işlenenden sonra	A B C * +
Prefix (Polish)	Her iki işlenenden önce	+ A * B C

Parantez dengesi — soru

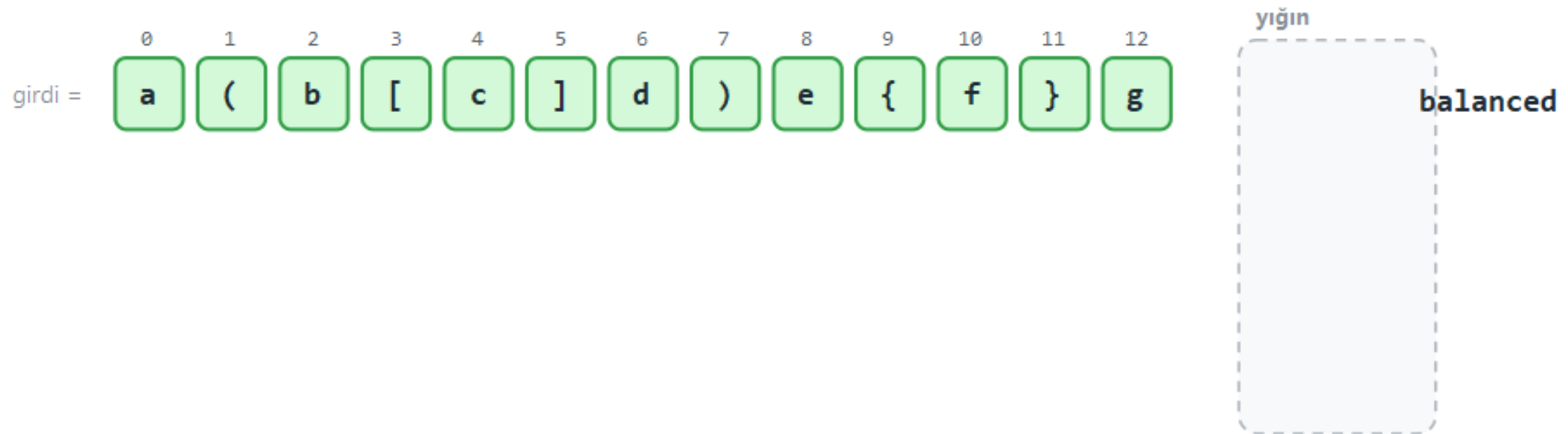
`{([ ])([ ])` geçerli biçimde iç içe mi?

Her kapanan, hâlâ açık olan **en son açılan** parantezle eşleşmelidir.

Kural

- Dizgiyi bir kez tara
- Her **açan**: it (push)
- Her **kapanan**: çek (pop), eşleşmeli
- Kapanan geldiğinde yığın boşsa → dengesiz
- Sonunda yığın boş değilse → dengesiz

Checking brackets with a stack



Girdi bitti, yığın boş: her açan kendi kapananını buldu → dengeli. Her karakter bir kez işlenir: $O(n)$.

Kod — matches()

```
static bool matches(char open, char close) {  
    return (open == '(' && close == ')') ||  
           (open == '[' && close == ']') ||  
           (open == '{' && close == '}');  
}
```

Kod — balanced()

```
bool balanced(const char *s) {
    char st[100]; int top = -1;
    for (int i = 0; s[i] != '\0'; i++) {
        char c = s[i];
        if (c == '(' || c == '[' || c == '{') {
            st[++top] = c;
        } else if (c == ')' || c == ']' || c == '}') {
            if (top == -1) return false;
            char o = st[top--];
            if (!matches(o, c)) return false;
        }
    }
    return top == -1;
}
```

Karmaşıklık

Her karaktere **tam olarak bir kez** bakılır,
her yığın işlemi $O(1)$ → **balanced** **$O(n)$** çalışır.

En kötü durum bellek: $O(n)$ — tamamı açan bir dizgi.

Tuzak — dengesiz olmanın üç yolu

1. Kapanan gelir, tepe eşleşmez — (]
2. Kapanan gelir, yığın zaten boş —)
3. Dizgi biter, yığın boş **değil** — ((

Mini soru

`((A+B)` , `balanced` tarafından kabul edilir mi? Neden?

Yanıt

Hayır. Her (itilir, dizgi yığında hâlâ biri varken biter — `top == -1` yanlıştır, bu yüzden `balanced` doğru biçimde `false` döndürür.

Postfix değerlendirme — soru

5 3 + 8 2 - * 6 + 12 - , $(5+3)*(8-2)=48$,
48+6=54 , 54-12=42 olarak değerlendirilmeli.

Hiçbir öncelik kuralı gerekmiyor — yalnız bir tarama.

Evaluating a postfix expression



Girdi bitti; yığında tek değer kaldı: 42. Her belirteç bir kez işlenir: $O(n)$.

Kod — apply()

```
static int apply(char op, int a, int b) {  
    switch (op) {  
        case '+': return a + b;  
        case '-': return a - b;  
        case '*': return a * b;  
        case '/': return a / b;  
        default: return 0;  
    }  
}
```

Kod — eval_postfix()

```
int eval_postfix(char *tok[], int n) {
    int st[100]; int top = -1;
    for (int i = 0; i < n; i++) {
        char *t = tok[i];
        if (isdigit(t[0])) {
            st[++top] = atoi(t);
        } else {
            int b = st[top--];
            int a = st[top--];
            st[++top] = apply(t[0], a, b);
        }
    }
    return st[top];
}
```

Karmaşıklık

Belirteç sayısında $O(n)$: her belirteç en çok bir kez itilir, en çok bir kez çekilir.

Tuzak — işlenen sırası önemli

$8\ 2\ -$, $8\ -\ 2$ demektir. İlk çekilen (b)
sağ işlenendir; ikinci çekilen (a) **sol**.
Yer değiştirirseniz $2\ -\ 8$ hesaplarsınız.

Mini soru

`eval_postfix` neden yalnız **tek** yığına ihtiyaç duyarken, sıradaki `to_postfix` ayrıca bir çıktı alanına da ihtiyaç duyar?

Yanıt

Değerlendirme iki işleneni ve bir işleci hemen tek bir sayıya **çökertir** — hatırlanacak fazladan bir şey yok. Dönüştürme yalnız sembolleri **yeniden sıralar**, bu yüzden onları biriktirecek bir yere ihtiyaç duyar.

Bir prefix ifadeyi değerlendirme

- İşleç, işlenenlerinden **önce** gelir: `+ A B`
- Postfix'le aynı fikir, ama **sağdan sola** taranır
- İlk çekilen değer artık **sol** işlenendir

Bir prefix ifadeyi değerlendirme



Girdi bitti; yığında tek değer kaldı: 56. Prefix ve postfix aynı fikrin iki yönüdür; ikisi de $O(n)$.

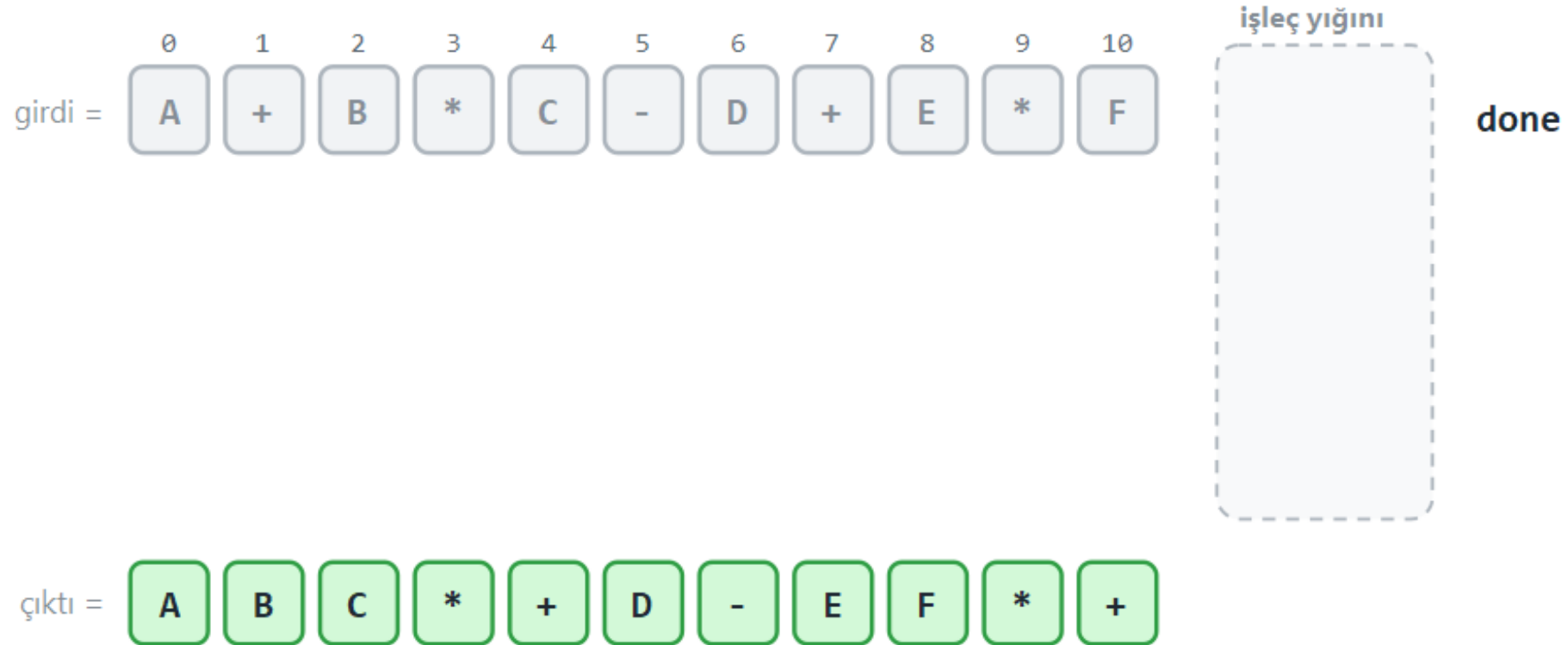
Prefix vs. postfix — ayna

- Sağdan sola tara, soldan sağa değil
- Önce **sol** işleneni çek, sonra sağ
- Aynı **$O(n)$** karmaşıklık, aynı hata denetimleri (sıfıra bölme, çok az/çok işlenen)

Infix'ten postfix'e — fikir

- İnsanlar infix yazar; değerlendircimiz postfix ister
- **Tren makası algoritması** (shunting-yard, Dijkstra) — yük vagonlarını yeniden sıralayan istasyonlardan
- **İşleç** tutan ikinci bir yığın, önceliği bir kerede çözer

Converting infix to postfix



Girdi bitti: yığında kalan işleçler sırayla çıktıya boşaltılır. Sonuç: A B C * + D - E F * +. Her karakter bir kez itilir, bir kez çekilir: $O(n)$.

Kod — prec()

```
static int prec(char op) {  
    if (op == '+' || op == '-') return 1;  
    if (op == '*' || op == '/') return 2;  
    return 0;  
}
```

Kod — to_postfix()

```
void to_postfix(const char *in, char *out) {
    char ops[100]; int top = -1, k = 0;
    for (int i = 0; in[i]; i++) {
        char c = in[i];
        if (isalnum(c)) { out[k++] = c; }
        else {
            while (top >= 0 && prec(ops[top]) >= prec(c))
                out[k++] = ops[top--];
            ops[++top] = c;
        }
    }
    while (top >= 0) out[k++] = ops[top--];
    out[k] = '\\0';
}
```

Karmaşıklık

$O(n)$: her karakter işleç yığınınına en çok bir kez itilir, en çok bir kez çekilir.

Tuzak — buradaki üç hata

- `>` yerine `>=` kullanmamak sola birleşmeyi bozar
- Son boşaltmayı unutmak bekleyen işleçleri kaybettirir
- Parantez desteği yok (doğal bir uzatma olarak bırakıldı)

Mini soru

$A*B+C$ 'yi elle postfix'e çevirin.

Yanıt

AB^*C+ . $A \rightarrow \text{çıktı}$. $* \rightarrow \text{it}$. $B \rightarrow \text{çıktı}$.

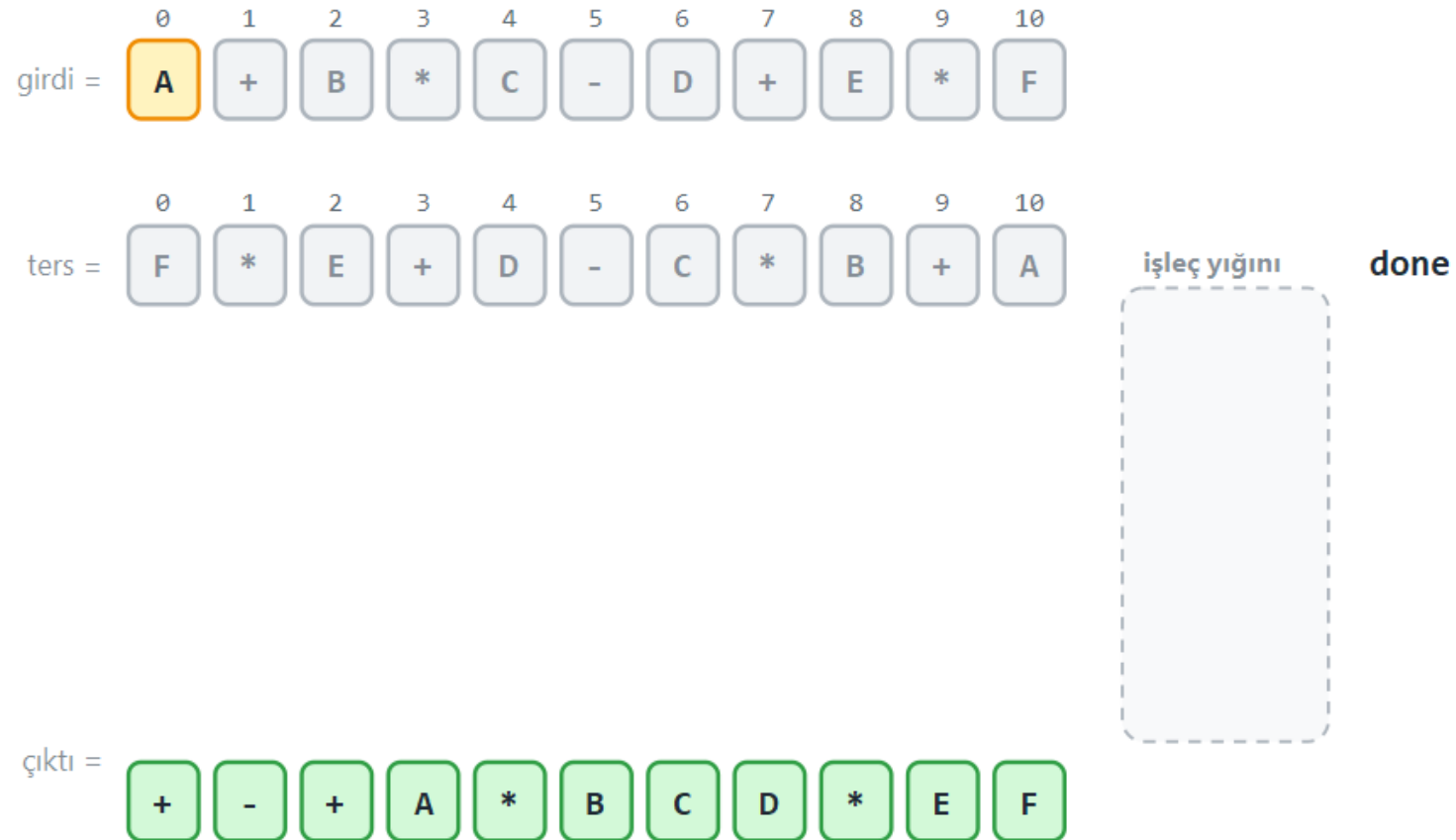
$+$, tepedeki $*$ 'i görür (öncelik $2 \geq 1$): $*$ 'i çek $\rightarrow$ çıktı, $+$ 'ı it.

$C \rightarrow$ çıktı. $+$ 'ı boşalt. Sonuç: $A B^* C +$.

Infix'i prefix'e çevirme

- Girdiyi **ters çevir**, (↔) değiştirerek
- Tren makasını yine çalıştır, ama bekleyen bir işleci yalnızca **kesin biçimde** daha güçlüyse çek (> , >= değil)
- Sonucu **ters çevir**

Infix'i prefix'e çevirme



3) Ara sonucu ters çevir: + - + A * B C D * E F — işte prefix. Her karakter bir kez itilir, bir kez çekilir: O(n).

Tuzak — kesin `>`, `>=` değil

- Prefix dönüşümü yalnızca **kesin biçimde daha güçlü** bekleyen bir işleci çeker — `>=` kullanmak son ters çevirmeyi bozar
- Ters çevirirken `( / )` 'yi değiştirmeyi unutmak gruplamayı bozar

3. Özyineleme ve Çağrı Yığını

Özyineleme nedir — bir soru

$\text{fact}(n) = n * \text{fact}(n - 1)$, temel durum $\text{fact}(0) = 1$.

$\text{fact}(3) = 3 * (2 * (1 * \text{fact}(0))) = 6$

Temel durum zorunludur

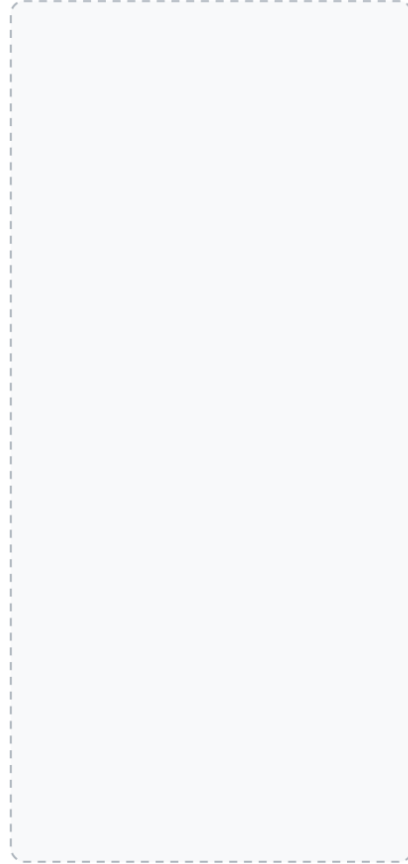
- İsteğe bağlı değil — özyinelemeyi **durduran** odur
- Temel durum olmadan: fonksiyon kendini **sonsuz** **kadar** çağırır
- "Sonsuz kadar" gerçekte şu demek: bellek bitene kadar

En basit özyineleme: geri sayım

- n 'i yazdır, $n - 1$ üzerinde özyinele, temel duruma kadar
- Temel durum $n \leq 0$ olmalı, $n == 0$ **değil**
- $n = 0$ ya da negatif n , ilk çağrıda durmalıdır

Özyineleme: geri sayım

çağrı yığını



ekran:

10

9

8

7

6

5

4

3

2

1

Liftoff!

Her çağrı bir çerçeve itti, her dönüş bir çerçeve çekti: özyineleme aslında bir yığındır. Derinlik 11 → $O(11)$ bellek. Ekrana yazılanlar: 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, Liftoff!.

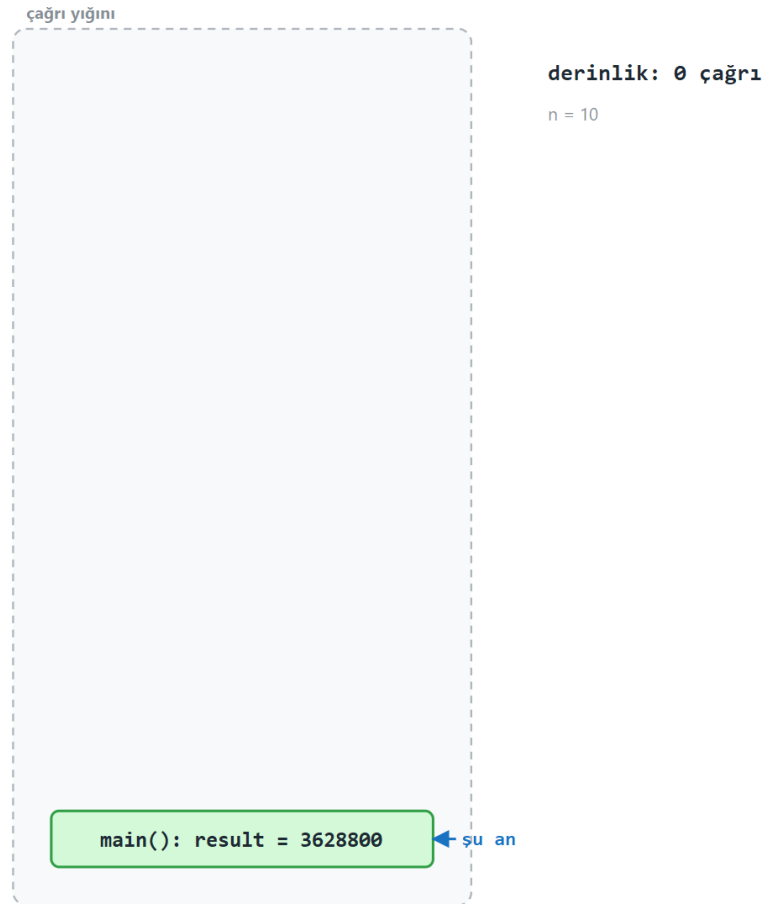
Tuzak — yanlış temel durum

Yalnız `n == 0` : negatif bir başlangıç değeri onu
hiçbir zaman karşılamaz — sonsuz özyineleme.
`n <= 0` bunu düzeltir: negatif girdi ilk çağrıda durur.

Çağrı yığını: özyineleme bir yığındır

- Her fonksiyon çağrısı bir **çerçeve** iter:
yerel değişkenler + dönüş adresi
- Bir fonksiyondan dönmek o çerçeveyi **çeker**
- Bu, siz yazmasanız da her programın içine gömülü,
gerçek bir **LIFO yığınıdır**

Recursion and the call stack: fact(n)



Bitti: fact(10) = 3628800. Her çağrı O(1) ek iş yapar; derinliği n olan bir özyinelemeli çağrı O(n) yığın belleği kullanır.

Kod — fact()

```
int fact(int n) {  
    if (n == 0) return 1;  
    return n * fact(n - 1);  
}
```

Beklenen çıktı

```
fact(10) = 3628800
```

Karmaşıklık

`fact(n)`, `n` özyinelemeli çağrı yapar: **$O(n)$ zaman**,
ve — unutulması kolay — **$O(n)$ yığın belleği**,
bekleyen her çağrı için bir çerçeve.

Aynı çarpımı hesaplayan bir döngü $O(1)$ bellek kullanır.

Tuzak — sessiz `int` taşması

`13! = 6227020800` — 32 bitlik bir `int` 'e sığmaz
(en fazla `2147483647`). Program **çökmez**;
sessizce `1932053504` 'e sarılır.

Tuzak — eksik temel durum

Ulaşılabilir bir temel durum olmadan, özyinelemeli bir fonksiyon **her çağrıda yeni bir çerçeveye** iter, sonsuza kadar — ta ki çağrı yığınının kendisi taşana kadar.

Tuzak — küçülmeyen bir problem

`fact(n)`, kendini kesinlikle daha küçük olan `n - 1` ile çağırır. Her özyinelemeli çağrı temel duruma **ölçülebilir** biçimde yaklaşmalıdır.

Mini soru

`fact(3)` özgün çağrı. `fact(0)` başladığı anda çağrı yığnında kaç çerçeve vardır?

Yanıt

Dört: `fact(3)` , `fact(2)` , `fact(1)` , `fact(0)` ,
her biri hâlâ altındakinin dönmesini bekliyor.

Mini soru

Temel durum `if (n == 1) return 1;` olsaydı
ve biri `fact(-1)` çağırıyorsa ne olurdu?

Yanıt

-1 hiçbir zaman 1'e eşit olmaz; fonksiyon
-2, -3, -4, ... üzerinde sonsuza kadar özyinelenir,
sonunda **çağrı yığınınını taşırır**.

4. Hanoi Kulesi

Bulmaca

- Üç çubuk; A çubuğunda büyükten küçüğe istiflenmiş diskler
- Bütün istifi C çubuğuna, bir seferde tek disk, taşı
- Büyük bir diski asla küçüğün üstüne koyma
- **Édouard Lucas** tarafından icat edildi, 1883 (keşişler efsanesi)

Özyinelemeli fikir

n diski from 'dan to 'ya, via 'yı kullanarak taşımak için:

1. Üstteki $n - 1$ diski via 'ya taşı
2. En büyük diski to 'ya taşı
3. $n - 1$ diski via 'dan onun üstüne taşı

Hanoi Kulesi

hamle 15 / 15



hamleler =

1A→B	2A→C	1B→C	3A→B	1C→A	2C→B	1A→B	4A→C	1B→C	2B→A	1C→A	3B→C	1A→B	2A→C
1B→C													

Bitti: 4 disk, 15 hamlede taşındı ($2^4 - 1 = 15$) — hamle sayısı üstel büyür.

Kod — hanoi()

```
void hanoi(int n, char from, char to, char via) {  
    if (n == 0) return;  
    hanoi(n - 1, from, via, to);  
    move_disk(n, from, to);  
    hanoi(n - 1, via, to, from);  
}
```

Beklenen çıktı

```
move 1: disk 1 from A to B  
move 2: disk 2 from A to C  
move 3: disk 1 from B to C  
move 4: disk 3 from A to B  
...  
total moves = 15
```

Karmaşıklık

$$T(n) = 2 * T(n-1) + 1, \quad T(0) = 0$$

→ $T(n) = 2^n - 1$ — **üstel** büyüme.

$n = 4$: $2^4 - 1 = 15$, yukarıdaki programla eşleşiyor.

Efsanenin 64 diski

$2^{64} - 1 \approx 1,8 \times 10^{19}$ hamle.

Saniyede bir hamleyle: **~585 milyar yıl** —
evrenin şu anki yaşının onlarca katı.

Biraz daha ileriye bakış

"Daha küçük bir sürümünü çöz, sonra birleştir",
önümüzdeki iki haftada gelecek **derinlik öncelikli**
arama (DFS)'nin bir ağacı ya da çizgeyi gezme biçiminin ta kendisi.

Mini soru

5 diskli bir Hanoi Kulesi kaç hamle gerektirir?

Yanıt

$2^5 - 1 = 31$ hamle.

Mini soru

`hanoi(n - 1, from, via, to)` içinde, son iki argüman dış çağrıya göre neden yer değiştirmiş?

Yanıt

Üstteki $n - 1$ diski yoldan çekmek için,
hedef yedek çubuktur, *yedek* ise özgün
hedefdir — roller özyinelemenin her seviyesinde döner.

5. Kuyruk: İlk Giren, İlk Çıkar

Sezgi — bir bekleme sırası

- Kasa sırası, yazdırma kuyruğu, web istek kuyruğu
- Yeni gelenler **arkaya** katılır
- En uzun süredir bekleyen **önden** ayrılır

FIFO — LIFO'nun aynası

İlk Giren, İlk Çıkar. Bir yığınla aynı iki temel işlem, farklı adlarla, ve şimdi **farklı uçlara** dokunuyorlar.

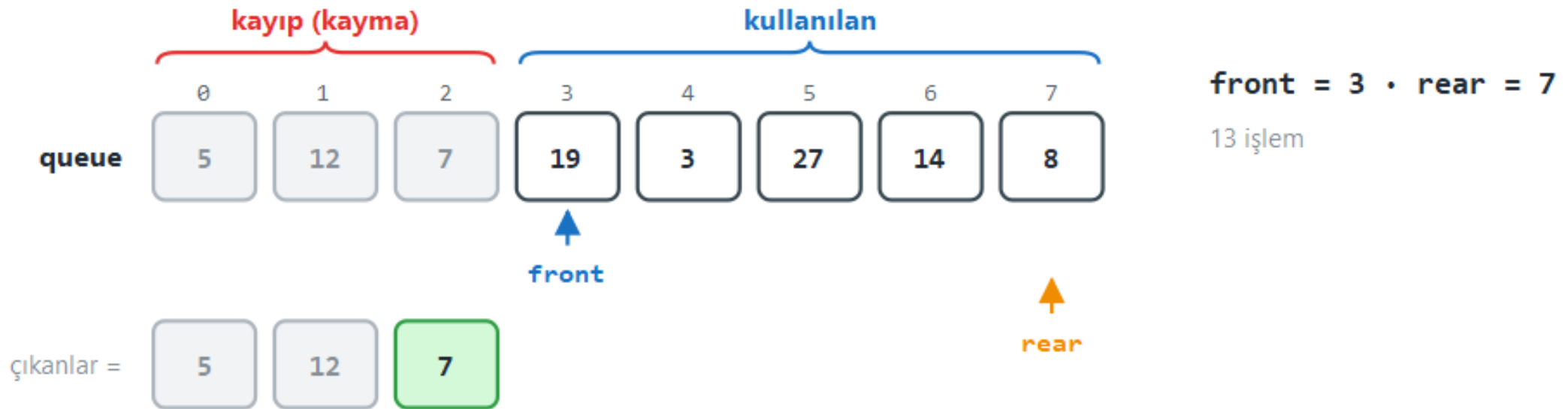
Kuyruk ADT'si

İşlem	Ne yapar	Ön koşul	Karmaşıklık
<code>enqueue(x)</code>	Arkaya ekle	dolu değil (dizi)	$O(1)$
<code>dequeue()</code>	Önü çıkar/döndür	boş değil	$O(1)$
<code>peek()</code>	Önü döndür, tut	boş değil	$O(1)$
<code>isEmpty()</code>	sıfır eleman mı?	yok	$O(1)$

Dizi kuyruğu — fikir

- İki indis tut: `front` (en eski), `rear` (en yeni)
- `enqueue` : `rear` 'ı ilerlet, oraya yaz
- `dequeue` : `front` 'ta oku, `front` 'u ilerlet
- Eleman kaydırılmaz — ama indis 0'daki yere bakın

Queue in a plain array



Bitti: 8 enqueue, 3 dequeue, 2 taşma. Kuyrukta: 19, 3, 27, 14, 8. front = 3 — yani 3 hücre bir daha asla kullanılamayacak (kapasitenin %38'i boşa gitti).

Kod — enqueue()/dequeue() (saf)

```
bool enqueue(int x) {  
    if (rear == CAP - 1) return false;  
    q[++rear] = x;  
    return true;  
}  
  
bool dequeue(int *out) {  
    if (front > rear) return false;  
    *out = q[front++];  
    return true;  
}
```

Beklenen çıktı

```
enqueue(5,12,7,19,3,27,14,8): front=0, rear=7  
dequeue() x3:    front=3, rear=7  
enqueue(99) -> false  
enqueue(42) -> false
```

Tuzak — kayma sorunu

rear yalnız **ileri** gider. Kuyruk, önde hücreler boşken "dolu" der — dizinin önünden **kaymıştır**.

Mini soru

Başında üç boş hücre olsa bile bu kuyruk neden "dolu" der?

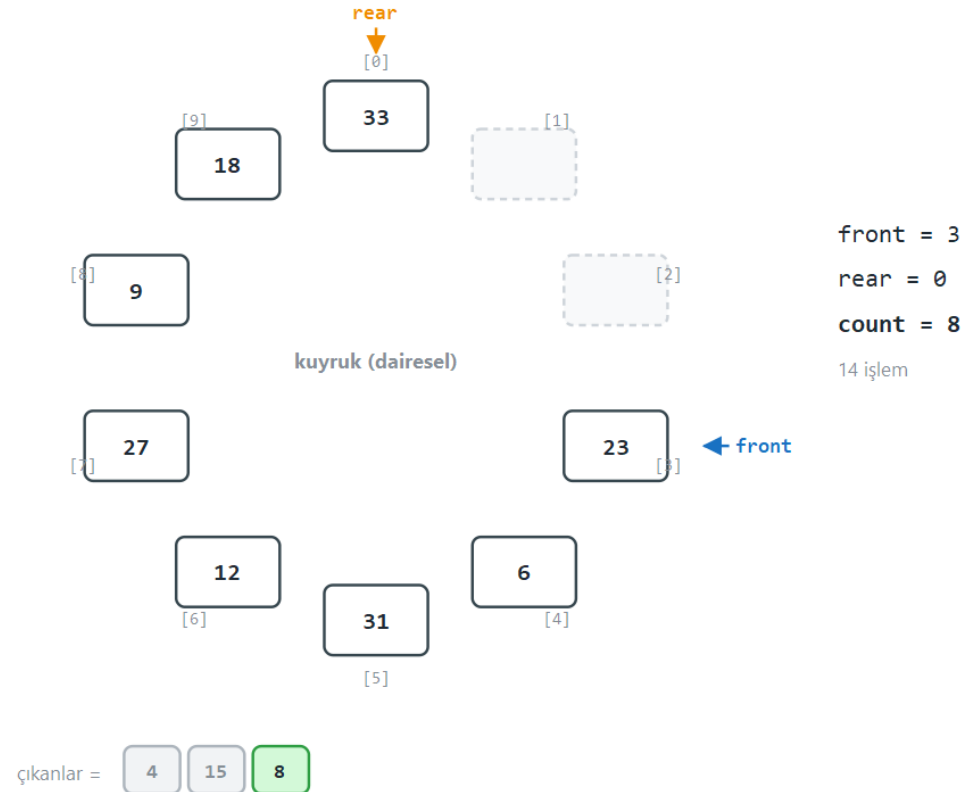
Yanıt

enqueue yalnız `rear == CAP - 1` 'i denetler;
`front` 'tan önceki hücrelerin boş olup olmadığına
hiç bakmaz. Gerçek çözüm sırada: diziyi bir **halka** gibi düşünün.

Dairesel kuyruk — halka gibi düşün

- Son indisten sonra yine ilk indis gelir
- $(\text{indis} + 1) \% \text{CAP}$, ileri yürür, başa döner
- $\text{front} == \text{rear}$ artık belirsiz (boş mu, dolu mu?)
- Bunu çözmek için bir alan daha tut: `count`

Circular queue



Bitti: 11 enqueue, 3 dequeue. Kuyrukta: 23, 6, 31, 12, 27, 9, 18, 33 (count = 8). front == rear burada yalnızca "tam bir eleman var" demek; asıl tuzak front == (rear + 1) % CAP — bu, hem boş hem dolu durumda aynı görünür, yalnız count ayırt eder.

Kod — enqueue() (daireysel)

```
bool enqueue(int x) {  
    if (count == CAP) return false;  
    rear = (rear + 1) % CAP;  
    q[rear] = x;  
    count++;  
    return true;  
}
```

Kod — dequeue() (daireysel)

```
bool dequeue(int *out) {  
    if (count == 0) return false;  
    *out = q[front];  
    front = (front + 1) % CAP;  
    count--;  
    return true;  
}
```

Neden `count` 'a da ihtiyacımız var

`front == rear`, "tek eleman", "boş" ya da "tamamen dolu" anlamına gelebilir — düz bir indis karşılaştırması artık bunları ayırt edemez.

Karmaşıklık — hâlâ $O(1)$, kayma yok

Kuyruk "dolu" demeden önce beş hücrenin tamamı kullanılır. Kayma sorunu tamamen ortadan kalktı, ve her işlem hâlâ $O(1)$.

Sık yapılan hatalar (daireseel kuyruk)

- Yalnız `front == rear` 'ı "boş" anlamına kullanmak
- İki indisten **birinde** modu unutmak
- İkisi de halkayı bir başa dönüşten sonra bozar

Mini soru

`count` , `CAP` 'e ulaştıktan sonra, `front == rear`
şimdi ne anlama gelir, boşken ne anlama geldiğine kıyasla?

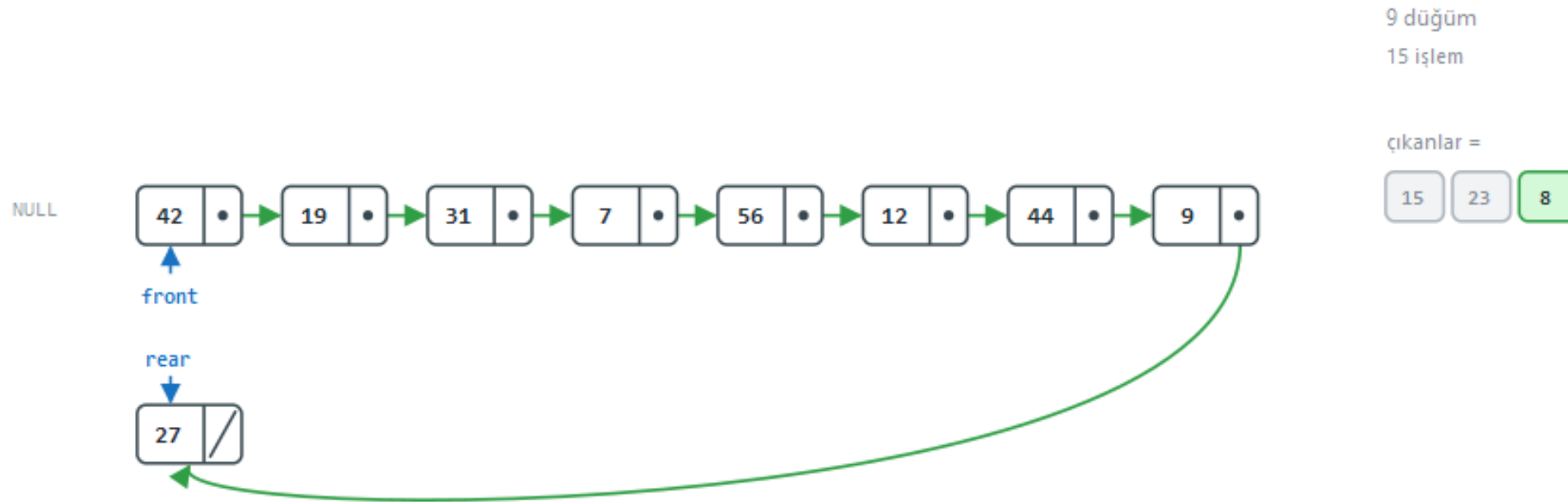
Yanıt

Her iki durum da `front == rear` gösterebilir;
ikisini ayıran tam olarak `count` 'tur, çünkü
indisler tek başına belirsizdir.

Hiç taşmayan bir kuyruk

- Bağlı liste ile kuyruk: eleman başına bir düğüm, `CAP` yok
- **İki** işaretçi gerekir: `front` (çıkart),
`rear` (ekle) — ikisi de $O(1)$ kalsın diye
- `rear` olmadan `enqueue` tüm listeyi gezerdi

Linked-list queue



Bitti: 12 enqueue, 3 dequeue çağrısı. Kuyrukta (front → rear): 42, 19, 31, 7, 56, 12, 44, 9, 27. İki işaretçi sayesinde her işlem $O(1)$; kapasite sınırı yok.

Kod — enqueue() (bağlı)

```
void enqueue(int x) {  
    QNode *n = malloc(sizeof(QNode));  
    n->data = x; n->next = NULL;  
    if (rear == NULL) {  
        front = rear = n;  
    } else {  
        rear->next = n;  
        rear = n;  
    }  
}
```

Kod — dequeue() (bağlı)

```
bool dequeue(int *out) {  
    if (front == NULL) return false;  
    QNode *tmp = front;  
    *out = tmp->data;  
    front = front->next;  
    if (front == NULL) rear = NULL;  
    free(tmp);  
    return true;  
}
```

Sık yapılan hatalar (bağlı kuyruk)

- Boşaldığında `rear` 'ı `NULL` 'a çekmeyi unutmak
- Tek düğümle yalnız `front` / `rear` 'dan birini güncellemek
- İkisi de sonraki enqueue'da **sarkan işaretçiye** yol açar

Mini soru

Bağlı kuyruk neden bir `rear` işaretçisine ihtiyaç duyarken, bağlı yığın duymaz?

Yanıt

Yığın **aynı** uçtan (**top**) ekler/çıkartır — tek işaretçi yeter. Kuyruk bir uçtan ekler, diğerinden çıkarır — $O(1)$ kalmak için **her iki uca** da bir işaretçiye ihtiyaç duyar.

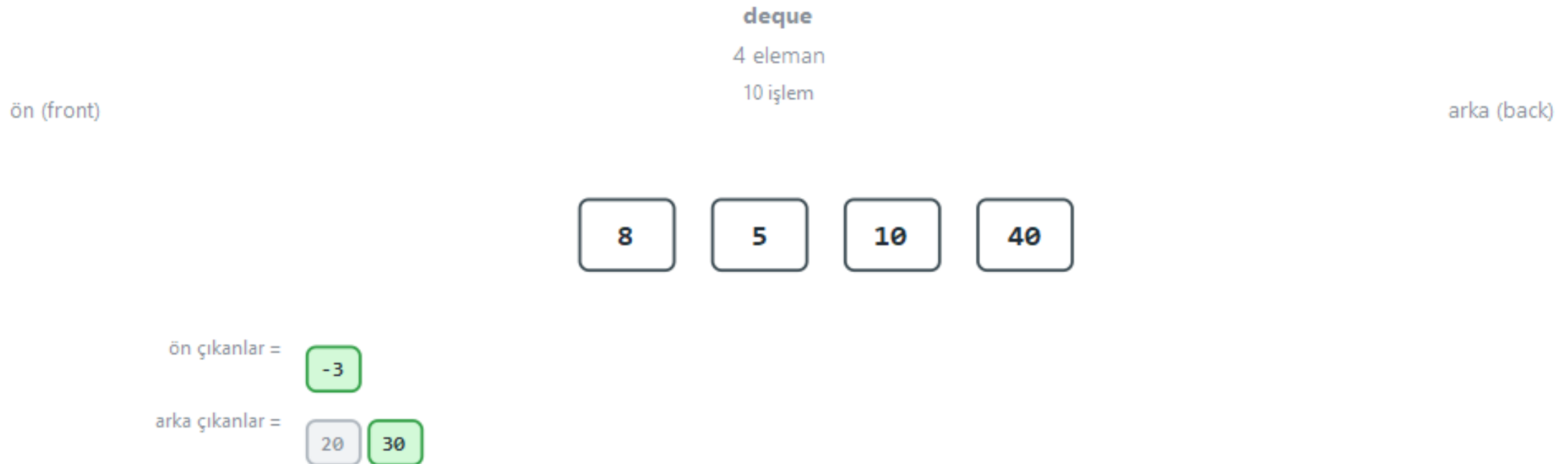
Deque — her iki uç

Bir **deque** (double-ended queue), "yalnız bir uç" kuralını kaldırır: hem önde hem arkada push/pop'a izin verir.

Deque ADT'si

İşlem	Ne yapar	Karmaşıklık
<code>push_back(x)</code>	Arkaya ekle	$O(1)$
<code>push_front(x)</code>	Öne ekle	$O(1)$
<code>pop_back()</code>	Arkayı çıkar	$O(1)$
<code>pop_front()</code>	Önü çıkar	$O(1)$

Double-ended queue (deque)



Bitti: 4 push_back, 3 push_front, 2 pop_back, 1 pop_front çağrısı. Deque (front → back): 8, 5, 10, 40. Dört işlem de $O(1)$: deque bir yığın ve bir kuyruğun yapabildiği her şeyi yapar.

Kod — push_back()

```
void push_back(Deque *d, int x) {
    DNode *n = malloc(sizeof(DNode));
    n->data = x;
    n->next = NULL;
    n->prev = d->back;
    if (d->back != NULL)
        d->back->next = n;
    else
        d->front = n;
    d->back = n;
}
```

Kod — pop_back()

```
bool pop_back(Deque *d) {  
    if (d->back == NULL) return false;  
    DNode *tmp = d->back;  
    d->back = tmp->prev;  
    if (d->back != NULL)  
        d->back->next = NULL;  
    else  
        d->front = NULL;  
    free(tmp);  
    return true;  
}
```

Not — bir deque, yığın ya da kuyruk olabilir

- Yalnız `push_back` / `pop_back` → tam olarak **yığın** gibi
- Yalnız `push_back` / `pop_front` → tam olarak **kuyruk** gibi
- Java'nın `ArrayDeque` 'i, ikisi için de daha hızlı, genel bir alternatif olarak önerilir

Mini soru

Boş deque: `push_front(1)` , `push_back(2)` ,
`push_front(3)` . Önden arkaya — nedir?

Yanıt

3 1 2 . `push_front(1)` → `[1]` .

`push_back(2)` → `[1, 2]` .

`push_front(3)` → `[3, 1, 2]` .

Çok seviyeli kuyruk

- Gerçek zamanlayıcılar nadiren her işe eşit davranır
- İşletim sistemi zamanlayıcısı aynı anda **birkaç** FIFO kuyruğu tutar
- Önce etkileşimli, sonra arka plan, sonra toplu iş
- Her seviye **sıradan bir kuyruktur**; bir politika birini seçer

Çok seviyeli kuyruk — seviyeler

Seviye	İş türü	Öncelik
Kuyruk 1	Etkileşimli	En yüksek
Kuyruk 2	Arka plan	Orta
Kuyruk 3	Toplu iş	En düşük

Çok seviyeli kuyruk zamanlaması



Bitti: 12 süreç, servis sırası — P3, P5, P7, P10, P1, P4, P8, P11, P2, P6, P9, P12.

Tuzak — açlık riski

Dokuz sistem/etkileşimli süreç arasında erken gelen bir **toplu iş** süreci en **son** sunulur — yalnız öncelik sırası açlığı önleyemez.

Mini soru

Çok seviyeli kuyruk, katı anlamda neden yeni bir veri yapısı değildir?

Yanıt

Her seviye **sıradan bir kuyruktur**; "çok seviyeli", birkaç var olan kuyruk arasından seçim yapan bir *zamanlama politikasını* tanımlar — yeni bir depolama yolu değil.

Özet — yığın tabanlı yapılar

Yapı	Kural	Ekle/çıkar	Tipik kullanım
Dizi yığını	LIFO	$O(1)$, sabit kapasite	Küçük, sınırlı yığınlar
Bağlı yığın	LIFO	$O(1)$, sınırsız	Öngörülemez boyut
Çağrı yığını	LIFO	Otomatik	Çalışan her program

Özet — kuyruk tabanlı yapılar (I)

Yapı	Kural	Ekle/çıkıkar
Dizi kuyruğu (saf)	FIFO	$O(1)$, ama kayar
Dairesel kuyruk	FIFO	$O(1)$, kayma yok
Bağlı liste ile kuyruk	FIFO	$O(1)$, sınırsız

Özet — kuyruk tabanlı yapılar (II)

Yapı	Kural	Tipik kullanım
Deque	Her iki uç	Kayan pencere, geri al/yinele
Çok seviyeli kuyruk	Birkaç FIFO + politika	İşletim sistemi zamanlaması

Özet — yığın uygulamaları

Uygulama	Yapı	Ana fikir
Parantez dengesi	Yığın	En son, ilk kapanan
Postfix değerlendirme	Yığın	İki çek, uygula, sonucu it
Infix → postfix	İşleç yığını + çıktı	Önceliği bir kerede çöz
Hanoi Kulesi	Çağrı yığını	n-1'i çöz, hareket et, n-1'i tekrar çöz

Büyük resim

Tek bir kural — **hangi uca/uçlara dokunabilirsiniz** —
bu dersteki her yapıyı ayırıyor: bir uç (yığın),
iki uç (kuyruk/deque), ya da birkaç paralel kuyruk (çok seviyeli).

Alıştırma

- Bugünün her konusu için alışırmalar hafta notlarında: `docs/week-3/cen207-week-3.tr.md`
- Peek, dengesiz parantez denetimi, prefix'siz infix-prefix çevirisi, count'suz dairesel kuyruk, palindrom denetimi, daha adil çok seviyeli zamanlama

Kendini sinama turu

On kısa soru. Yanıt bir sonraki slaytta görünmeden önce düşünün.

1. LIFO ne anlama gelir?

Son Giren, İlk Çıkar — yığın.

2. FIFO ne anlama gelir?

İlk Giren, İlk Çıkar — kuyruk.

3. Dizide `push` neden `top == CAP - 1` 'i denetler, `top == CAP` 'i değil?

Geçerli indisler 0..CAP-1; yığın, **top** CAP-1'e ulaştığı an dolar.

4. Postfix değerlendirmede hangi işlenen önce çekilir?

Sağdaki işlenen — en son itildiği için ilk o çıkar.

5. Saf bir dizi kuyruğu, önde boş hücreler olsa bile neden sonunda "dolu" der?

rear yalnız artar, **dequeue** 'nun önde boşalttığı hücreleri hiç yeniden kullanmaz.

6. Dairesel kuyruk, `front` / `rear` 'ın ötesinde hangi ekstra duruma ihtiyaç duyar?

Şu anki eleman sayısını tutan bir `count` — indisler tek başına boşu doludan ayıramaz.

7. $\text{fact}(n) = n * \text{fact}(n - 1)$ 'in temel durumu nedir?

`fact(0) = 1` . Bu olmadan, çağrılar hiç durmaz ve çağrı yığını taşar.

8. Çağrı yığını neden gerçek bir yığın sayılmayı hak eder?

Tam olarak LIFO'ya uyar: en son çağrı her zaman bitirecek bir sonraki çağrıdır.

9. n disk için kaç Hanoi hamlesi gerekir?

$2^n - 1$ hamle — n 'de üstel büyüme.

10. Ne sıradan bir yığının ne de sıradan bir kuyruğun yapamadığı bir şey söyleyin.

Hem önden hem arkadan, ikisi de $O(1)$, ekleme ya da çıkarma.

Bir sonraki hafta

4. hafta — Ağaçlar: İkili Ağaçlar, Dolaşmalar, Öbekler

Bugünkü her yığın ve kuyruk, elemanları katı,
dallanmayan bir sırada tuttu. Ağaçlar, bir düğümün
birden çok çocuğu olmasına izin verir — Hanoi
Kulesi için kullanılan tam olarak aynı özyinelemeli örüntüyle dolaşılır.

Kaynaklar (1/2)

- Ders izlencesi, Hafta 3: `docs/syllabus/syllabus.tr.md`
- Cormen, Leiserson, Rivest, Stein. *Introduction to Algorithms*, 3. baskı. MIT Press
- Sedgewick, Wayne. *Algorithms*, 4. baskı. Addison-Wesley, 2011

Kaynaklar (2/2)

- Deitel & Deitel. *C How to Program*, 7. baskı.
- Y. D. Liang. *Introduction to Java Programming*, 10. baskı.
- É. Lucas, *Récréations mathématiques*, 1883
- J. Łukasiewicz, Polish/Reverse Polish yazımı, 1920'ler