Bölüm 6: Yığınlar

C ile Veri Yapıları ve Algoritmalar

LIFO · Yığın ADT · push / pop · Taşma · Bağlı yığın · Çağrı yığını · Çoklu yığın

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

GİRİŞ

Öğrenme Çıktıları

LIFO'yu tanımlamak

Yığını $S = \{a_0, \ldots, a_{n-1}\}$ listesi olarak tanımlayıp alt ve üst öğeyi göstermek.

Dizi ile gerçeklemek

`top` indisiyle push/pop yazmak; taşma (overflow) ve alt taşmayı (underflow) denetlemek.

Bağlı yığın kurmak

Düğüm ayırıp (`malloc`) bağlamak, çıkarılan düğümü `free` ile bırakmak.

Çağrı yığınını tanımak

Fonksiyon çağrılarının bir yığın gibi çalıştığını ve çağrı yığınının neden taştığını açıklamak.

Tasarım kararı vermek

Sabit dizi, büyüyen dizi ve bağlı liste arasında maliyete göre seçim yapmak.

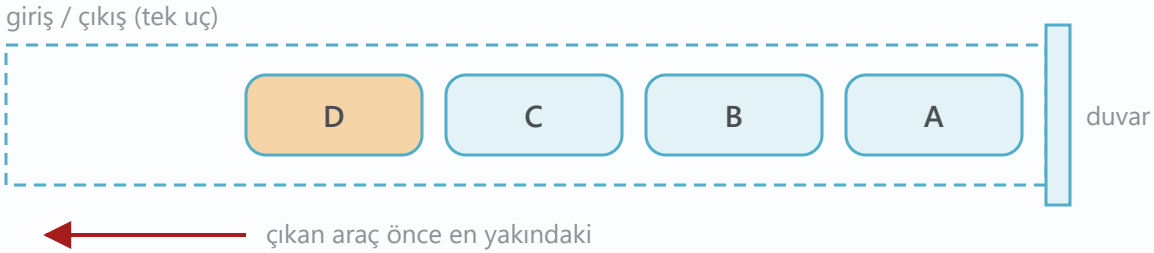
Çoklu yığını yönetmek

Tek bir bellek dizisinde n yığını `top[i]` ve `sinir[i]` ile paylaştırmak.

YIĞIN ADT

Yığın (Stack): Son Giren İlk Çıkar

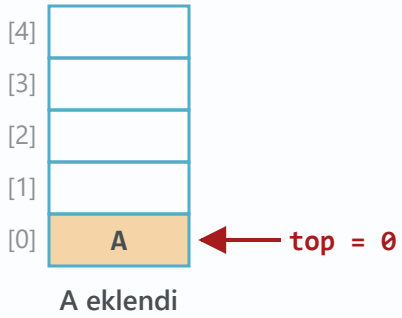
Yığın, ekleme (*insertion*) ve silmenin (*deletion*) **yalnızca bir uçtan** yapıldığı sıralı bir listedir: $S = \{a_0, a_1, \ldots, a_{n-1}\}$ için a_0 **alt**, a_{n-1} **üst** (top) öğedir. Son eklenen ilk çıkar: **LIFO** (last-in-first-out).



Tek arabalık çıkmaz sokağa park eden araçlar: son giren (D) ilk çıkmak zorunda.

Türkçe kaynaklarda stack için **yığın** ve **yığıt** ikisi de kullanılır; bu derste stack için “yığın”, heap veri yapısı için “heap” kullanılır.

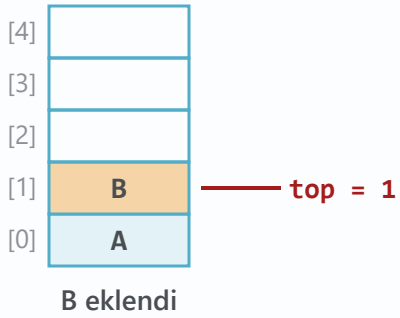
Yığının Büyümesi ve Küçülmesi



A eklendi → üst öğe artık A; top = 0.

- Ekleme sırası: A, B, C, D, E.
- Silme sırası (tümü çıkarılsaydı): E, D, C, B, A: **tam ters**.
- Ortada bir öğeye (ör. B) erişmek için üstündekilerin hepsi alınmalı.
- Bu kısıt bir kusur değil, bir sözleşmedir: “en son açılan iş ilk kapanır”.

Yığının Büyümesi ve Küçülmesi

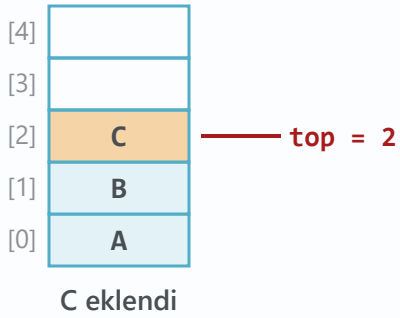


B eklendi

B eklendi → üst öğe artık B; top = 1.

- Ekleme sırası: A, B, C, D, E.
- Silme sırası (tümü çıkarılsaydı): E, D, C, B, A: **tam ters**.
- Ortada bir öğeye (ör. B) erişmek için üstündekilerin hepsi alınmalı.
- Bu kısıt bir kusur değil, bir sözleşmedir: “en son açılan iş ilk kapanır”.

Yığının Büyümesi ve Küçülmesi

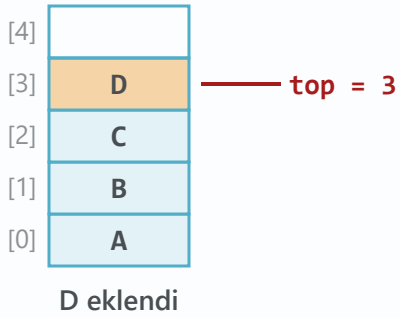


C eklendi

C eklendi → üst öğe artık C; top = 2.

- Ekleme sırası: A, B, C, D, E.
- Silme sırası (tümü çıkarılsaydı): E, D, C, B, A: **tam ters**.
- Ortada bir öğeye (ör. B) erişmek için üstündekilerin hepsi alınmalı.
- Bu kısıt bir kusur değil, bir sözleşmedir: “en son açılan iş ilk kapanır”.

Yığının Büyümesi ve Küçülmesi

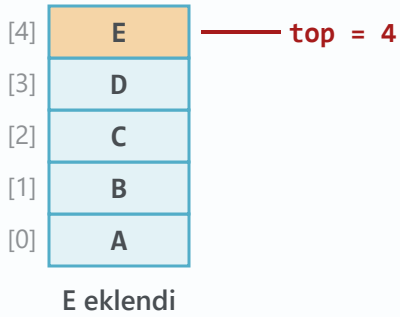


D eklendi

D eklendi → üst öğe artık D; top = 3.

- Ekleme sırası: A, B, C, D, E.
- Silme sırası (tümü çıkarılsaydı): E, D, C, B, A: **tam ters**.
- Ortada bir öğeye (ör. B) erişmek için üstündekilerin hepsi alınmalı.
- Bu kısıt bir kusur değil, bir sözleşmedir: “en son açılan iş ilk kapanır”.

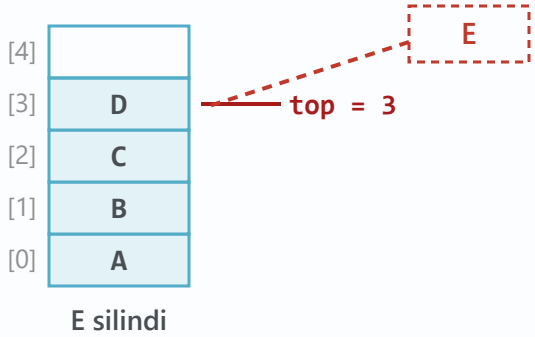
Yığının Büyümesi ve Küçülmesi



E eklendi → üst öğe artık E; top = 4.

- Ekleme sırası: A, B, C, D, E.
- Silme sırası (tümü çıkarılsaydı): E, D, C, B, A: **tam ters**.
- Ortada bir öğeye (ör. B) erişmek için üstündekilerin hepsi alınmalı.
- Bu kısıt bir kusur değil, bir sözleşmedir: “en son açılan iş ilk kapanır”.

Yığının Büyümesi ve Küçülmesi



E silindi

Silme **yalnız üstten**: ilk alınan öğe en son eklenen E olur. Üst öğe yeniden D, top = 3.

- Ekleme sırası: A, B, C, D, E.
- Silme sırası (tümü çıkarılsaydı): E, D, C, B, A: **tam ters**.
- Ortada bir öğeye (ör. B) erişmek için üstündekilerin hepsi alınmalı.
- Bu kısıt bir kusur değil, bir sözleşmedir: “en son açılan iş ilk kapanır”.

YIĞIN ADT

Soyut Veri Tipi: Yığın

İşlem	Anlamı	Ön koşul / hata
`yigit_olustur(max)`	En çok *max* öğe alan boş yığın	—
`yigit_dolu(S)`	Öğe sayısı max ise doğru	—
`yigit_ekle(S, x)` · push	x'i üste koyar	Dolu ise **taşma** (overflow)
`yigit_bos(S)`	Öğe yoksa doğru	—
`yigit_al(S)` · pop	Üstteki öğeyi çıkarıp döndürür	Boş ise **alt taşma** (underflow)
`yigit_ust(S)` · peek	Üstteki öğeyi çıkarmadan okur	Boş ise hata

ADT **ne** yapıldığını söyler, **nasıl** yapıldığını söylemez. Aynı arayüz sabit dizi, büyüyen dizi ya da bağlı listeyle gerçeklenebilir (Horowitz–Sahni, ADT 3.1).

YIĞIN ADT

Yığının Kullanıldığı Yerler

Çağrı yığını
Her fonksiyon çağrısı dönüş adresi ve yerel değişkenleriyle bir çerçeve iter.

İfade değerlendirme
Derleyiciler infix → postfix çevirisi ve postfix hesaplamada yığın kullanır.

Geri izleme
Labirentte çıkmaza girilince son yol ayrımına dönülür.

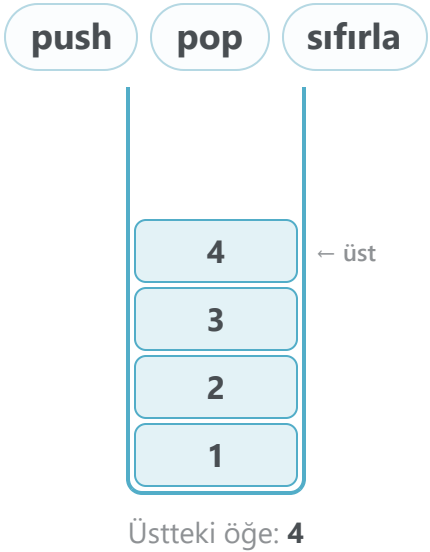
Parantez denetimi
Her kapanan, en son açılanla eşleşmeli.

Geri al (undo)
Düzenleyici her işlemi yığına koyar; Ctrl+Z en sonuncuyu geri alır.

Derinlik öncelikli arama
DFS, ziyaret edilecek düğümleri yığında tutar.

YIĞIN ADT

push ve pop Yalnız Üstte



Dört kez push, ardından pop: çıkan sıra girenin tersidir. Boş yığında pop **alt taşma**dır.

Dizi ile Yığın: `top` İndisi



Boş: top < 0

Boş yığında **top = −1**. Bu hâlde öğe almak **alt taşma** (underflow) hatasıdır.

Dolu: top ≥ MAX_STACK_SIZE − 1

Bu hâlde öğe eklemek **taşma** (overflow) hatasıdır.

Öğe sayısı = top + 1

İkisini karıştırmak klasik "bir eksik/bir fazla" hatasıdır.

Ne olur?

C dizi sınırını denetlemez: denetimsiz bir push komşu belleğin üzerine yazar.

Dizi ile Yığın: C Tanımları

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_STACK_SIZE 101

typedef struct {
    int deger;
    /* diğer alanlar */
} element;

element yigit[MAX_STACK_SIZE];

void yigit_dolu(void) {
    printf("Yigit dolu!!!\n"); exit(-1);
}
void yigit_bos(void) {
    printf("Yigit bos!!!\n"); exit(-2);
}
```

- `element` : yığında saklanacak kayıt; şimdilik tek alan `deger` .
- Yığın **küresel** (global) bir dizi; `top` ise çağırandan **adresle** gelir.
- Hata yordamları mesaj basıp programı `exit` ile sonlandırır.

HOCADAN “Böyle yapılırsa sıfır alınır, hatta eksi not alınır. Burada bu konu kaç kere anlatıldı, değil mi?”

Ekleme ve Silme Yordamları

```
void yigit_ekle(int *top, element item) {
    if (*top >= MAX_STACK_SIZE - 1)  /* yer var mı? */
        yigit_dolu();          /* uyar ve çık */
    yigit[++*top] = item;      /* önce işaretçiyi artır */
}                              /* sonra öğeyi yerleştir */

element yigit_al(int *top) {
    if (*top < 0)
        yigit_bos();           /* boşsa uyar ve çık */
    return yigit[(*top)--];    /* döndür, sonra azalt */
}
```

`yigit[++*top]`

Önce `*top` bir artar, **sonra** yeni indise yazılır: boş yığında −1 → 0.

`yigit[(*top)--]`

Önce eski indisteki öğe okunur, **sonra** `*top` azalır. Parantez şart: `*top--` işaretçinin kendisini azaltırdı.

HOCADAN "Normal şartlarda yordamın formal parametre listesinde "*parametre" olarak gözüken değişken, yordam içinde de "*parametre" olarak kullanılmalıdır."

C'de Öncelik Tuzağı: `*` ile `++` / `--`

İfade	C bunu nasıl okur	Sonuç
`++*top`	`++(*top)` : gösterilen tamsayıyı artır	Doğru (push)
`(*top)--`	gösterilen tamsayıyı kullan, sonra azalt	Doğru (pop)
`*top--`	`*(top--)` : **işaretçiyi** azalt, eski adresi oku	**Hatalı**: top değişmez
`*top++`	`*(top++)` : işaretçiyi ilerlet	**Hatalı**

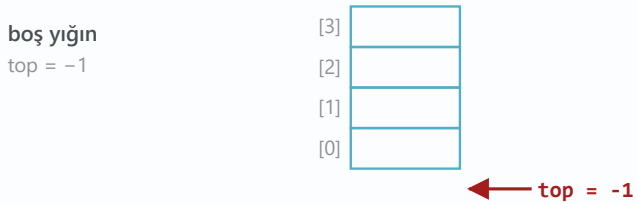
Kural

Son ek `++` / `--`, tekli `*` 'dan **daha önceliklidir**; ön ek `++` ile `*` aynı düzeydedir ve sağdan sola bağlanır.

Neden adresle?

`top` çağıranın değişkenidir; yordam onu değiştirebilsin diye `&top` geçirilir (C'de değerle çağrı vardır).

Kod İzleme: Dizi ile Yığın



`main` 'de **top = −1** (boş). Sırayla ekle A, B, C; al; ekle D, E, F. MAX_STACK_SIZE = 4.

Adım 1 / 15

```
1  void yigit_ekle(int *top, element item) {
2    if (*top >= MAX_STACK_SIZE - 1)
3      yigit_dolu();  /* "Yigit dolu!!!", exit */
4    yigit[++*top] = item;
5  }
6  element yigit_al(int *top) {
7    if (*top < 0)
8      yigit_bos();   /* "Yigit bos!!!", exit */
9    return yigit[(*top)--];
10 }
```

DEĞİŞKENLER

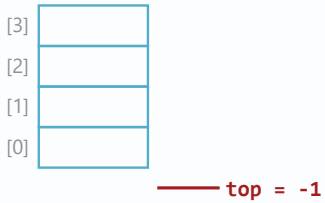
`top(main) = -1` `yigit = []`

ÇAĞRI YIĞINI

`main()`

ÇIKTI

Kod İzleme: Dizi ile Yığın



-1 ≥ 3? Hayır, yer var.

Adım 2 / 15

```
1  void yigit_ekle(int *top, element item) {
▶    if (*top >= MAX_STACK_SIZE - 1)
3      yigit_dolu();  /* "Yigit dolu!!!", exit */
4    yigit[++*top] = item;
5  }
6  element yigit_al(int *top) {
7    if (*top < 0)
8      yigit_bos();   /* "Yigit bos!!!", exit */
9    return yigit[(*top)--];
10 }
```

DEĞİŞKENLER

`*top = -1` `item = A` `yigit = []`

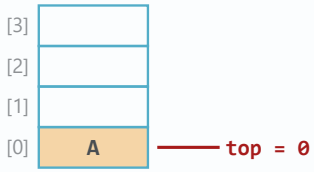
ÇAĞRI YIĞINI

`yigit_ekle(&top, A)`

`main()`

ÇIKTI

Kod İzleme: Dizi ile Yığın



++*top → top = 0, sonra **yigit[0] = A**.

Adım 3 / 15

```
1  void yigit_ekle(int *top, element item) {
2    if (*top >= MAX_STACK_SIZE - 1)
3      yigit_dolu();  /* "Yigit dolu!!!", exit */
▶    yigit[++*top] = item;
5  }
6  element yigit_al(int *top) {
7    if (*top < 0)
8      yigit_bos();   /* "Yigit bos!!!", exit */
9    return yigit[(*top)--];
10 }
```

DEĞİŞKENLER

*top = 0 | item = A | yigit = [A]

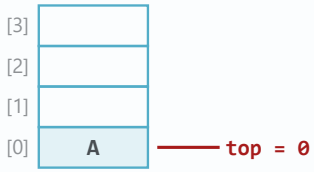
ÇAĞRI YIĞINI

yigit_ekle(&top, A)

main()

ÇIKTI

Kod İzleme: Dizi ile Yığın



0 ≥ 3? Hayır, yer var.

Adım 4 / 15

```
1  void yigit_ekle(int *top, element item) {
▶    if (*top >= MAX_STACK_SIZE - 1)
3      yigit_dolu();  /* "Yigit dolu!!!", exit */
4    yigit[++*top] = item;
5  }
6  element yigit_al(int *top) {
7    if (*top < 0)
8      yigit_bos();   /* "Yigit bos!!!", exit */
9    return yigit[(*top)--];
10 }
```

DEĞİŞKENLER

`*top = 0` `item = B` `yigit = [A]`

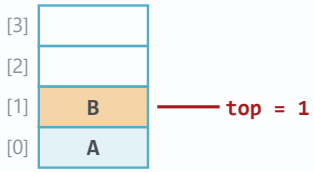
ÇAĞRI YIĞINI

`yigit_ekle(&top, B)`

`main()`

ÇIKTI

Kod İzleme: Dizi ile Yığın



++*top → top = 1, sonra **yigit[1] = B**.

Adım 5 / 15

```
1  void yigit_ekle(int *top, element item) {
2    if (*top >= MAX_STACK_SIZE - 1)
3      yigit_dolu();  /* "Yigit dolu!!!", exit */
▶    yigit[++*top] = item;
5  }
6  element yigit_al(int *top) {
7    if (*top < 0)
8      yigit_bos();   /* "Yigit bos!!!", exit */
9    return yigit[(*top)--];
10 }
```

DEĞİŞKENLER

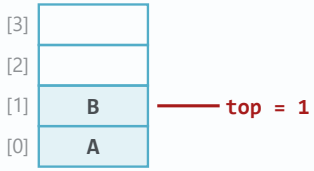
*top = 1 | item = B | yigit = [A, B]

ÇAĞRI YIĞINI

yigit_ekle(&top, B)
main()

ÇIKTI

Kod İzleme: Dizi ile Yığın



1 ≥ 3? Hayır, yer var.

Adım 6 / 15

```
1  void yigit_ekle(int *top, element item) {
▶    if (*top >= MAX_STACK_SIZE - 1)
3      yigit_dolu();  /* "Yigit dolu!!!", exit */
4    yigit[++*top] = item;
5  }
6  element yigit_al(int *top) {
7    if (*top < 0)
8      yigit_bos();   /* "Yigit bos!!!", exit */
9    return yigit[(*top)--];
10 }
```

DEĞİŞKENLER

`*top = 1` `item = C`

`yigit = [A, B]`

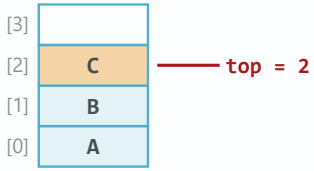
ÇAĞRI YIĞINI

`yigit_ekle(&top, C)`

`main()`

ÇIKTI

Kod İzleme: Dizi ile Yığın



++*top → top = 2, sonra **yigit[2] = C**.

Adım 7 / 15

```
1  void yigit_ekle(int *top, element item) {
2    if (*top >= MAX_STACK_SIZE - 1)
3      yigit_dolu();  /* "Yigit dolu!!!", exit */
▶    yigit[++*top] = item;
5  }
6  element yigit_al(int *top) {
7    if (*top < 0)
8      yigit_bos();   /* "Yigit bos!!!", exit */
9    return yigit[(*top)--];
10 }
```

DEĞİŞKENLER

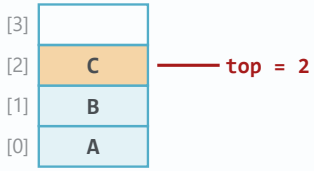
*top = 2 | item = C | yigit = [A, B, C]

ÇAĞRI YIĞINI

yigit_ekle(&top, C)
main()

ÇIKTI

Kod İzleme: Dizi ile Yığın



2 < 0? Hayır, yığın boş değil.

Adım 8 / 15

```
1  void yigit_ekle(int *top, element item) {
2    if (*top >= MAX_STACK_SIZE - 1)
3      yigit_dolu();  /* "Yigit dolu!!!", exit */
4    yigit[++*top] = item;
5  }
6  element yigit_al(int *top) {
▶    if (*top < 0)
8      yigit_bos();   /* "Yigit bos!!!", exit */
9    return yigit[(*top)--];
10 }
```

DEĞİŞKENLER

`*top = 2` `yigit = [A, B, C]`

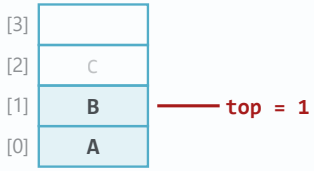
ÇAĞRI YIĞINI

`yigit_al(&top)`

`main()`

ÇIKTI

Kod İzleme: Dizi ile Yığın



yigit[2] = C döndürülür, ardından top = 1. Hücrede eski değer durur ama artık yığının parçası değildir.

Adım 9 / 15

```
1  void yigit_ekle(int *top, element item) {
2    if (*top >= MAX_STACK_SIZE - 1)
3      yigit_dolu();  /* "Yigit dolu!!!", exit */
4    yigit[++*top] = item;
5  }
6  element yigit_al(int *top) {
7    if (*top < 0)
8      yigit_bos();   /* "Yigit bos!!!", exit */
▶    return yigit[(*top)--];
10 }
```

DEĞİŞKENLER

`*top = 1` `donen = C`

`yigit = [A, B]`

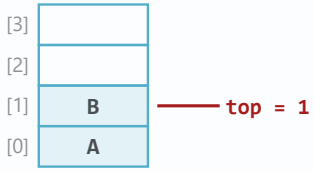
ÇAĞRI YIĞINI

`yigit_al(&top)`

`main()`

ÇIKTI

Kod İzleme: Dizi ile Yığın



1 ≥ 3? Hayır, yer var.

Adım 10 / 15

```
1  void yigit_ekle(int *top, element item) {
▶    if (*top >= MAX_STACK_SIZE - 1)
3      yigit_dolu();  /* "Yigit dolu!!!", exit */
4    yigit[++*top] = item;
5  }
6  element yigit_al(int *top) {
7    if (*top < 0)
8      yigit_bos();   /* "Yigit bos!!!", exit */
9    return yigit[(*top)--];
10 }
```

DEĞİŞKENLER

`*top = 1` `item = D`

`yigit = [A, B]`

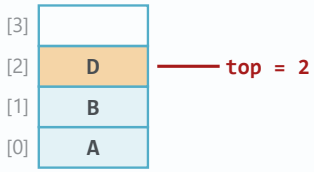
ÇAĞRI YIĞINI

`yigit_ekle(&top, D)`

`main()`

ÇIKTI

Kod İzleme: Dizi ile Yığın



++*top → top = 2, sonra **yigit[2] = D**.

Adım 11 / 15

```
1  void yigit_ekle(int *top, element item) {
2    if (*top >= MAX_STACK_SIZE - 1)
3      yigit_dolu();  /* "Yigit dolu!!!", exit */
▶    yigit[++*top] = item;
5  }
6  element yigit_al(int *top) {
7    if (*top < 0)
8      yigit_bos();   /* "Yigit bos!!!", exit */
9    return yigit[(*top)--];
10 }
```

DEĞİŞKENLER

*top = 2 | item = D

yigit = [A, B, D]

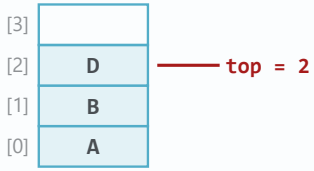
ÇAĞRI YIĞINI

yigit_ekle(&top, D)

main()

ÇIKTI

Kod İzleme: Dizi ile Yığın



2 ≥ 3? Hayır, yer var.

Adım 12 / 15

```
1  void yigit_ekle(int *top, element item) {
▶    if (*top >= MAX_STACK_SIZE - 1)
3      yigit_dolu();  /* "Yigit dolu!!!", exit */
4    yigit[++*top] = item;
5  }
6  element yigit_al(int *top) {
7    if (*top < 0)
8      yigit_bos();   /* "Yigit bos!!!", exit */
9    return yigit[(*top)--];
10 }
```

DEĞİŞKENLER

`*top = 2` `item = E`

`yigit = [A, B, D]`

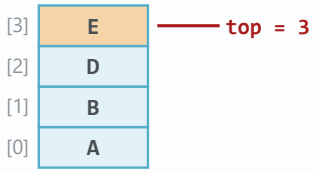
ÇAĞRI YIĞINI

`yigit_ekle(&top, E)`

`main()`

ÇIKTI

Kod İzleme: Dizi ile Yığın



++*top → top = 3, sonra **yigit[3] = E**.

Adım 13 / 15

```
1  void yigit_ekle(int *top, element item) {
2    if (*top >= MAX_STACK_SIZE - 1)
3      yigit_dolu();  /* "Yigit dolu!!!", exit */
▶    yigit[++*top] = item;
5  }
6  element yigit_al(int *top) {
7    if (*top < 0)
8      yigit_bos();   /* "Yigit bos!!!", exit */
9    return yigit[(*top)--];
10 }
```

DEĞİŞKENLER

*top = 3 | item = E

yigit = [A, B, D, E]

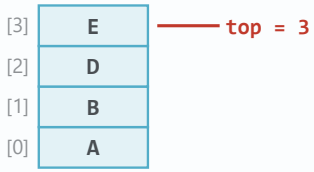
ÇAĞRI YIĞINI

yigit_ekle(&top, E)

main()

ÇIKTI

Kod İzleme: Dizi ile Yığın



3 ≥ 3? **Evet**: yer yok!


```
1  void yigit_ekle(int *top, element item) {
▶    if (*top >= MAX_STACK_SIZE - 1)
3      yigit_dolu();  /* "Yigit dolu!!!", exit */
4    yigit[++*top] = item;
5  }
6  element yigit_al(int *top) {
7    if (*top < 0)
8      yigit_bos();   /* "Yigit bos!!!", exit */
9    return yigit[(*top)--];
10 }
```

DEĞİŞKENLER

`*top = 3` `item = F`

`yigit = [A, B, D, E]`

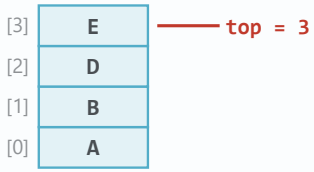
ÇAĞRI YIĞINI

`yigit_ekle(&top, F)`

`main()`

ÇIKTI

Kod İzleme: Dizi ile Yığın



`yigit_dolu()` mesajı basar ve `exit(-1)` ile **tüm programı** bitirir. Kütüphane kodunda bu kaba bir tepkidir; `bool` döndüren sürüm kararı çağırana bırakır.

Adım 15 / 15

```
1  void yigit_ekle(int *top, element item) {
2    if (*top >= MAX_STACK_SIZE - 1)
▶      yigit_dolu();  /* "Yigit dolu!!!", exit */
4    yigit[++*top] = item;
5  }
6  element yigit_al(int *top) {
7    if (*top < 0)
8      yigit_bos();   /* "Yigit bos!!!", exit */
9    return yigit[(*top)--];
10 }
```

DEĞİŞKENLER

`*top = 3` `item = F`

`yigit = [A, B, D, E]`

ÇAĞRI YIĞINI

`yigit_ekle(&top, F)`

`main()`

ÇIKTI

```
Yigit dolu!!!
```

Daha İyi Tasarım: Yığın Bir Nesne Gibi

```c
#include <stdbool.h>
#define KAP 100

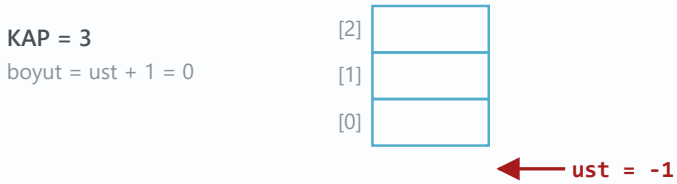
typedef struct {
    int veri[KAP];
    int ust;                /* boşken -1 */
} Yigit;

void yigit_baslat(Yigit *y) { y->ust = -1; }
bool yigit_bos(const Yigit *y)  { return y->ust < 0; }
bool yigit_dolu(const Yigit *y) { return y->ust == KAP - 1; }
int  yigit_boyut(const Yigit *y) { return y->ust + 1; }
bool yigit_ust(const Yigit *y, int *x) {
    if (yigit_bos(y)) return false;
    *x = y->veri[y->ust];  return true;
}
```

- Yığın ve `ust` **aynı yapıda**: iki yığın aynı programda karışmadan yaşar.
- Küresel değişken yok; her yordam yığını parametre olarak alır.
- Hata durumunda `exit` yerine `false` döner; kararı **çağıran** verir.

HOCADAN "Global değişken kullanmak yasak! İçeride yığın, kuyruk, local değişken, ne gerekiyorsa kullanılabilir. Ama global değişken yok!"

Kod İzleme: `bool` Döndüren push / pop



`yigit_baslat(&y)` sonrası **ust = −1**. Sırayla push 5, 8, 2, 9; sonra dört kez pop. KAP = 3.

Adım 1 / 12

```
1  bool push(Yigit *y, int x) {
2    if (y->ust == KAP - 1) return false; /* taşma */
3    y->veri[++y->ust] = x;
4    return true;
5  }
6  bool pop(Yigit *y, int *x) {
7    if (y->ust < 0) return false;   /* alt taşma */
8    *x = y->veri[y->ust--];
9    return true;
10 }
```

DEĞİŞKENLER

`y.ust = -1` `y.veri = []`

ÇAĞRI YIĞINI

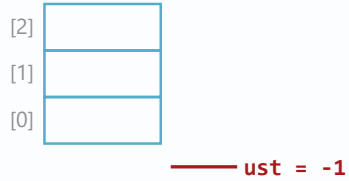
`main()`

ÇIKTI

Kod İzleme: `bool` Döndüren push / pop

KAP = 3

boyut = ust + 1 = 0



-1 == 2? Hayır, yer var.

Adım 2 / 12

```
1  bool push(Yigit *y, int x) {
▶    if (y->ust == KAP - 1) return false; /* taşma */
3    y->veri[++y->ust] = x;
4    return true;
5  }
6  bool pop(Yigit *y, int *x) {
7    if (y->ust < 0) return false;   /* alt taşma */
8    *x = y->veri[y->ust--];
9    return true;
10 }
```

DEĞİŞKENLER

`y→ust = -1` `x = 5` `y→veri = []`

ÇAĞRI YIĞINI

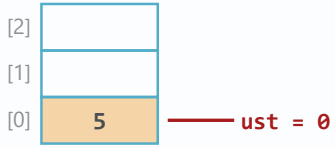
`push(&y, 5)`

`main()`

ÇIKTI

Kod İzleme: `bool` Döndüren push / pop

KAP = 3
boyut = ust + 1 = 1



++y->ust → 0; **veri[0] = 5**; true döner.

Adım 3 / 12

```
1  bool push(Yigit *y, int x) {
2    if (y->ust == KAP - 1) return false; /* taşma */
▶    y->veri[++y->ust] = x;
▶    return true;
5  }
6  bool pop(Yigit *y, int *x) {
7    if (y->ust < 0) return false;   /* alt taşma */
8    *x = y->veri[y->ust--];
9    return true;
10 }
```

DEĞİŞKENLER

`y→ust = 0` `x = 5` `y→veri = [5]`

ÇAĞRI YIĞINI

- `push(&y, 5)`
- `main()`

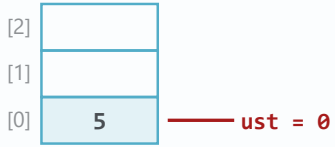
ÇIKTI

```
push 5:1
```

Kod İzleme: `bool` Döndüren push / pop

KAP = 3

boyut = ust + 1 = 1



0 == 2? Hayır, yer var.

Adım 4 / 12

```
1  bool push(Yigit *y, int x) {
▶    if (y->ust == KAP - 1) return false; /* taşma */
3    y->veri[++y->ust] = x;
4    return true;
5  }
6  bool pop(Yigit *y, int *x) {
7    if (y->ust < 0) return false;   /* alt taşma */
8    *x = y->veri[y->ust--];
9    return true;
10 }
```

DEĞİŞKENLER

`y→ust = 0` `x = 8` `y→veri = [5]`

ÇAĞRI YIĞINI

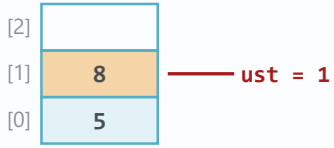
- `push(&y, 8)`
- `main()`

ÇIKTI

```
push 5:1
```

Kod İzleme: `bool` Döndüren push / pop

KAP = 3
boyut = ust + 1 = 2



++y->ust → 1; **veri[1] = 8**; true döner.

Adım 5 / 12

```
1  bool push(Yigit *y, int x) {
2    if (y->ust == KAP - 1) return false; /* taşma */
▶    y->veri[++y->ust] = x;
▶    return true;
5  }
6  bool pop(Yigit *y, int *x) {
7    if (y->ust < 0) return false;   /* alt taşma */
8    *x = y->veri[y->ust--];
9    return true;
10 }
```

DEĞİŞKENLER

`y→ust = 1` `x = 8`
`y→veri = [5, 8]`

ÇAĞRI YIĞINI

`push(&y, 8)`
`main()`

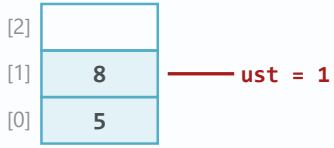
ÇIKTI

```
push 5:1 8:1
```

Kod İzleme: `bool` Döndüren push / pop

KAP = 3

boyut = ust + 1 = 2



1 == 2? Hayır, yer var.

Adım 6 / 12

```
1  bool push(Yigit *y, int x) {
▶    if (y->ust == KAP - 1) return false; /* taşma */
3    y->veri[++y->ust] = x;
4    return true;
5  }
6  bool pop(Yigit *y, int *x) {
7    if (y->ust < 0) return false;   /* alt taşma */
8    *x = y->veri[y->ust--];
9    return true;
10 }
```

DEĞİŞKENLER

`y→ust = 1` `x = 2`

`y→veri = [5, 8]`

ÇAĞRI YIĞINI

`push(&y, 2)`

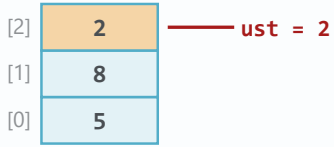
`main()`

ÇIKTI

```
push 5:1 8:1
```

Kod İzleme: `bool` Döndüren push / pop

KAP = 3
boyut = ust + 1 = 3



++y->ust → 2; **veri[2] = 2**; true döner.

Adım 7 / 12

```
1  bool push(Yigit *y, int x) {
2    if (y->ust == KAP - 1) return false; /* taşma */
▶    y->veri[++y->ust] = x;
▶    return true;
5  }
6  bool pop(Yigit *y, int *x) {
7    if (y->ust < 0) return false;   /* alt taşma */
8    *x = y->veri[y->ust--];
9    return true;
10 }
```

DEĞİŞKENLER

`y→ust = 2` `x = 2`
`y→veri = [5, 8, 2]`

ÇAĞRI YIĞINI

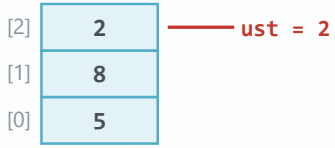
`push(&y, 2)`
`main()`

ÇIKTI

```
push 5:1 8:1 2:1
```

Kod İzleme: `bool` Döndüren push / pop

KAP = 3
boyut = ust + 1 = 3



ust == KAP − 1 (2 == 2) → **return false**. Program sürer; çağıran taşmayı öğrenir.

Adım 8 / 12

```
1  bool push(Yigit *y, int x) {
▶    if (y->ust == KAP - 1) return false; /* taşma */
3    y->veri[++y->ust] = x;
4    return true;
5  }
6  bool pop(Yigit *y, int *x) {
7    if (y->ust < 0) return false;   /* alt taşma */
8    *x = y->veri[y->ust--];
9    return true;
10 }
```

DEĞİŞKENLER

`y→ust = 2` `x = 9`

`y→veri = [5, 8, 2]`

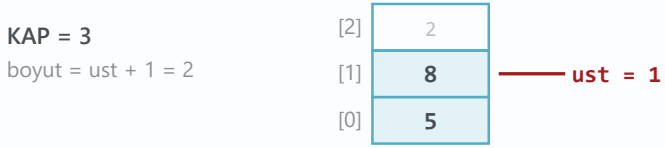
ÇAĞRI YIĞINI

`push(&y, 9)`
`main()`

ÇIKTI

```
push 5:1 8:1 2:1 9:0
```

Kod İzleme: `bool` Döndüren push / pop



Boş değil; ***x = veri[2] = 2**, sonra ust = 1.

Adım 9 / 12

```
1  bool push(Yigit *y, int x) {
2    if (y->ust == KAP - 1) return false; /* taşma */
3    y->veri[++y->ust] = x;
4    return true;
5  }
6  bool pop(Yigit *y, int *x) {
▶    if (y->ust < 0) return false;   /* alt taşma */
▶    *x = y->veri[y->ust--];
9    return true;
10 }
```

DEĞİŞKENLER

`y→ust = 1` `*x = 2`

`y→veri = [5, 8]`

ÇAĞRI YIĞINI

`pop(&y, &x)`

`main()`

ÇIKTI

```
push 5:1 8:1 2:1 9:0
pop 2
```

Kod İzleme: `bool` Döndüren push / pop

KAP = 3

boyut = ust + 1 = 1



Boş değil; ***x = veri[1] = 8**, sonra ust = 0.

Adım 10 / 12

```
1  bool push(Yigit *y, int x) {
2    if (y->ust == KAP - 1) return false; /* taşma */
3    y->veri[++y->ust] = x;
4    return true;
5  }
6  bool pop(Yigit *y, int *x) {
▶    if (y->ust < 0) return false;   /* alt taşma */
▶    *x = y->veri[y->ust--];
9    return true;
10 }
```

DEĞİŞKENLER

`y→ust = 0` `*x = 8` `y→veri = [5]`

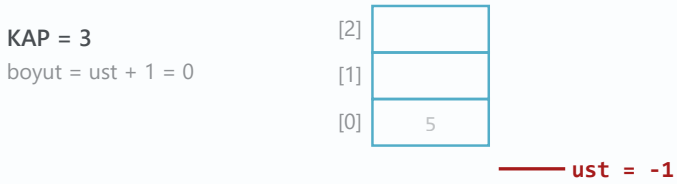
ÇAĞRI YIĞINI

- `pop(&y, &x)`
- `main()`

ÇIKTI

```
push 5:1 8:1 2:1 9:0
pop 2 8
```

Kod İzleme: `bool` Döndüren push / pop



Boş değil; ***x = veri[0] = 5**, sonra ust = -1.

Adım 11 / 12

```
1  bool push(Yigit *y, int x) {
2    if (y->ust == KAP - 1) return false; /* taşma */
3    y->veri[++y->ust] = x;
4    return true;
5  }
6  bool pop(Yigit *y, int *x) {
▶    if (y->ust < 0) return false;   /* alt taşma */
▶    *x = y->veri[y->ust--];
9    return true;
10 }
```

DEĞİŞKENLER

`y→ust = -1` `*x = 5` `y→veri = []`

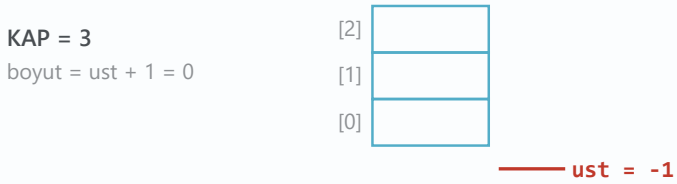
ÇAĞRI YIĞINI

- `pop(&y, &x)`
- `main()`

ÇIKTI

```
push 5:1 8:1 2:1 9:0
pop 2 8 5
```

Kod İzleme: `bool` Döndüren push / pop



ust < 0 → return false: alt taşma, `*x` 'e dokunulmadı.

Adım 12 / 12

```
1  bool push(Yigit *y, int x) {
2    if (y->ust == KAP - 1) return false; /* taşma */
3    y->veri[++y->ust] = x;
4    return true;
5  }
6  bool pop(Yigit *y, int *x) {
▶    if (y->ust < 0) return false;   /* alt taşma */
8    *x = y->veri[y->ust--];
9    return true;
10 }
```

DEĞİŞKENLER

`y→ust = -1` `y→veri = []`

ÇAĞRI YIĞINI

- `pop(&y, &x)`
- `main()`

ÇIKTI

```
push 5:1 8:1 2:1 9:0
pop 2 8 5 false
```

Bileşen Gibi Yaz: `yigit.h` ve `yigit.c`

```c
/* yigit.h: kullanıcı yalnız bunu görür */
#ifndef YIGIT_H
#define YIGIT_H
#include <stdbool.h>
typedef struct Yigit Yigit;  /* opak */
Yigit *yigit_olustur(int kapasite);
void yigit_yok_et(Yigit *y);
bool yigit_ekle(Yigit *y, int x);
bool yigit_al(Yigit *y, int *x);
bool yigit_bos(const Yigit *y);
#endif
```

Kullanan kod `struct`'ın içini göremez; yalnız işlemleri çağırır. Gerçekleme diziden bağlı listeye geçse bile kullanan kod değişmez.

```c
/* yigit.c: gerçekleme */
struct Yigit { int *veri; int ust, kap; };

Yigit *yigit_olustur(int kap) {
  Yigit *y = malloc(sizeof *y);
  if (y == NULL) return NULL;
  y->veri = malloc(kap * sizeof *y->veri);
  if (y->veri == NULL) {
    free(y);  return NULL;
  }
  y->ust = -1;  y->kap = kap;
  return y;
}
void yigit_yok_et(Yigit *y) {
  if (y) { free(y->veri); free(y); }
}
```

Kod İzleme: Yığınla Dizge Ters Çevirme



Girdi **"YIGIT"**; boş yığın kuruldu.

Adım 1 / 12

```
1 void ters_cevir(char *s) {
2   Yigit y;  int c, i;
▶   yigit_baslat(&y);
4   for (i = 0; s[i] != '\0'; i++)
5     push(&y, s[i]);      /* KAP yeterli varsayıldı */
6   for (i = 0; pop(&y, &c); i++)
7     s[i] = (char)c;
8 }
```

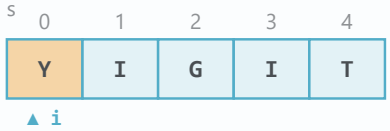
DEĞİŞKENLER

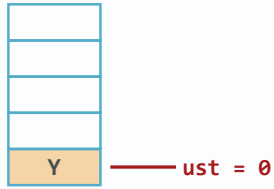
`s = "YIGIT"` `i = ?` `y.ust = -1`

ÇAĞRI YIĞINI

- **ters_cevir(s)**
- main()

Kod İzleme: Yığınla Dizge Ters Çevirme





s[0] = 'Y' ≠ '\0' → **push('Y')**.

Adım 2 / 12

```
1 void ters_cevir(char *s) {
2   Yigit y;  int c, i;
3   yigit_baslat(&y);
▶   for (i = 0; s[i] != '\0'; i++)
▶     push(&y, s[i]);     /* KAP yeterli varsayıldı */
6   for (i = 0; pop(&y, &c); i++)
7     s[i] = (char)c;
8 }
```

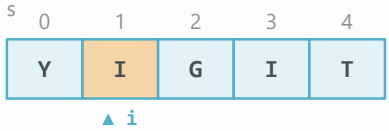
DEĞİŞKENLER

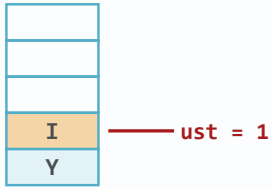
`s = "YIGIT"` `i = 0` `y.ust = 0`

ÇAĞRI YIĞINI

- `ters_cevir(s)`
- `main()`

Kod İzleme: Yığınla Dizge Ters Çevirme





s[1] = 'I' ≠ '\0' → **push('I')**.


```
1 void ters_cevir(char *s) {
2   Yigit y;  int c, i;
3   yigit_baslat(&y);
▶   for (i = 0; s[i] != '\0'; i++)
▶     push(&y, s[i]);     /* KAP yeterli varsayıldı */
6   for (i = 0; pop(&y, &c); i++)
7     s[i] = (char)c;
8 }
```

DEĞİŞKENLER

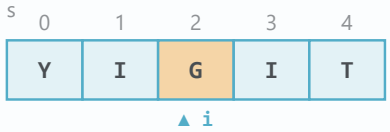
`s = "YIGIT"` `i = 1` `y.ust = 1`

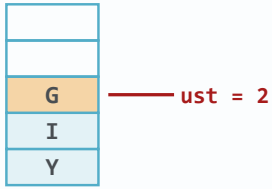
ÇAĞRI YIĞINI

`ters_cevir(s)`

`main()`

Kod İzleme: Yığınla Dizge Ters Çevirme





s[2] = 'G' ≠ '\0' → **push('G')**.

Adım 4 / 12

```
1 void ters_cevir(char *s) {
2   Yigit y;  int c, i;
3   yigit_baslat(&y);
▶   for (i = 0; s[i] != '\0'; i++)
▶     push(&y, s[i]);     /* KAP yeterli varsayıldı */
6   for (i = 0; pop(&y, &c); i++)
7     s[i] = (char)c;
8 }
```

DEĞİŞKENLER

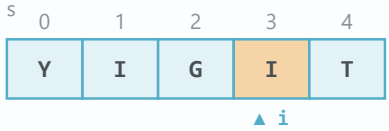
`s = "YIGIT"` `i = 2` `y.ust = 2`

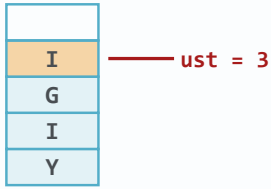
ÇAĞRI YIĞINI

`ters_cevir(s)`

`main()`

Kod İzleme: Yığınla Dizge Ters Çevirme





s[3] = 'I' ≠ '\0' → **push('I')**.

Adım 5 / 12

```
1 void ters_cevir(char *s) {
2   Yigit y;  int c, i;
3   yigit_baslat(&y);
▶   for (i = 0; s[i] != '\0'; i++)
▶     push(&y, s[i]);     /* KAP yeterli varsayıldı */
6   for (i = 0; pop(&y, &c); i++)
7     s[i] = (char)c;
8 }
```

DEĞİŞKENLER

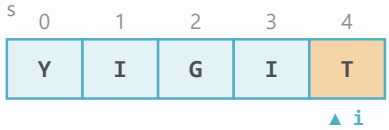
`s = "YIGIT"` `i = 3` `y.ust = 3`

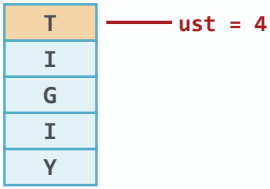
ÇAĞRI YIĞINI

`ters_cevir(s)`

`main()`

Kod İzleme: Yığınla Dizge Ters Çevirme





s[4] = 'T' ≠ '\0' → **push('T')**.

Adım 6 / 12

```
1 void ters_cevir(char *s) {
2   Yigit y;  int c, i;
3   yigit_baslat(&y);
▶   for (i = 0; s[i] != '\0'; i++)
▶     push(&y, s[i]);      /* KAP yeterli varsayıldı */
6   for (i = 0; pop(&y, &c); i++)
7     s[i] = (char)c;
8 }
```

DEĞİŞKENLER

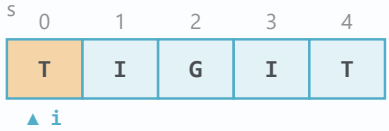
`s = "YIGIT"` `i = 4` `y.ust = 4`

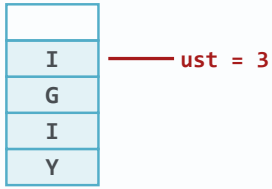
ÇAĞRI YIĞINI

`ters_cevir(s)`

`main()`

Kod İzleme: Yığınla Dizge Ters Çevirme





pop → **'T'**, **s[0] = 'T'**.

Adım 7 / 12

```
1 void ters_cevir(char *s) {
2   Yigit y;  int c, i;
3   yigit_baslat(&y);
4   for (i = 0; s[i] != '\0'; i++)
5     push(&y, s[i]);     /* KAP yeterli varsayıldı */
▶   for (i = 0; pop(&y, &c); i++)
▶     s[i] = (char)c;
8 }
```

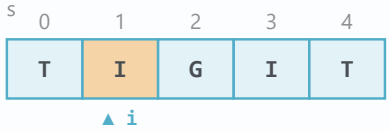
DEĞİŞKENLER

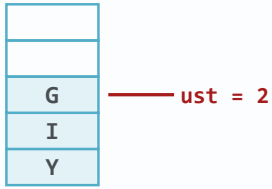
`s = "TIGIT"` `i = 0` `c = 'T'` `y.ust = 3`

ÇAĞRI YIĞINI

- `ters_cevir(s)`
- `main()`

Kod İzleme: Yığınla Dizge Ters Çevirme





pop → **'I'**, **s[1] = 'I'**.

Adım 8 / 12

```
1 void ters_cevir(char *s) {
2   Yigit y;  int c, i;
3   yigit_baslat(&y);
4   for (i = 0; s[i] != '\0'; i++)
5     push(&y, s[i]);     /* KAP yeterli varsayıldı */
▶   for (i = 0; pop(&y, &c); i++)
▶     s[i] = (char)c;
8 }
```

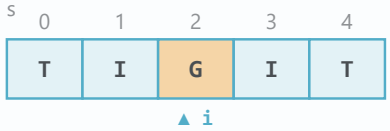
DEĞİŞKENLER

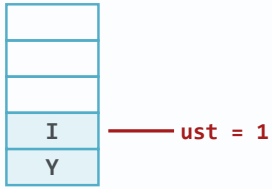
`s = "TIGIT"` `i = 1` `c = 'I'` `y.ust = 2`

ÇAĞRI YIĞINI

- `ters_cevir(s)`
- `main()`

Kod İzleme: Yığınla Dizge Ters Çevirme





pop → **'G'**, **s[2] = 'G'**.

Adım 9 / 12

```
1 void ters_cevir(char *s) {
2   Yigit y;  int c, i;
3   yigit_baslat(&y);
4   for (i = 0; s[i] != '\0'; i++)
5     push(&y, s[i]);     /* KAP yeterli varsayıldı */
▶   for (i = 0; pop(&y, &c); i++)
▶     s[i] = (char)c;
8 }
```

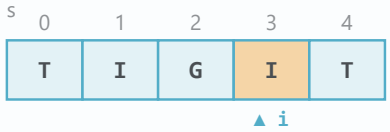
DEĞİŞKENLER

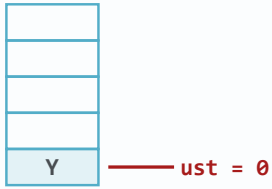
`s = "TIGIT"` `i = 2` `c = 'G'` `y.ust = 1`

ÇAĞRI YIĞINI

- `ters_cevir(s)`
- `main()`

Kod İzleme: Yığınla Dizge Ters Çevirme





pop → **'I'**, **s[3] = 'I'**.

Adım 10 / 12

```
1 void ters_cevir(char *s) {
2   Yigit y;  int c, i;
3   yigit_baslat(&y);
4   for (i = 0; s[i] != '\0'; i++)
5     push(&y, s[i]);      /* KAP yeterli varsayıldı */
▶   for (i = 0; pop(&y, &c); i++)
▶     s[i] = (char)c;
8 }
```

DEĞİŞKENLER

`s = "TIGIT"` `i = 3` `c = 'I'` `y.ust = 0`

ÇAĞRI YIĞINI

- `ters_cevir(s)`
- `main()`

Kod İzleme: Yığınla Dizge Ters Çevirme



pop → **'Y'**, **s[4] = 'Y'**.


```
1 void ters_cevir(char *s) {
2   Yigit y;  int c, i;
3   yigit_baslat(&y);
4   for (i = 0; s[i] != '\0'; i++)
5     push(&y, s[i]);     /* KAP yeterli varsayıldı */
▶   for (i = 0; pop(&y, &c); i++)
▶     s[i] = (char)c;
8 }
```

DEĞİŞKENLER

`s = "TIGIY"` `i = 4` `c = 'Y'` `y.ust = -1`

ÇAĞRI YIĞINI

- `ters_cevir(s)`
- `main()`

Kod İzleme: Yığınla Dizge Ters Çevirme



Yığın boş: `pop` false döner, döngü biter. Sonuç **"TIGIY"**: n push + n pop → **O(n)**.

Adım 12 / 12

```
1 void ters_cevir(char *s) {
2   Yigit y;  int c, i;
3   yigit_baslat(&y);
4   for (i = 0; s[i] != '\0'; i++)
5     push(&y, s[i]);     /* KAP yeterli varsayıldı */
▶   for (i = 0; pop(&y, &c); i++)
7     s[i] = (char)c;
8 }
```

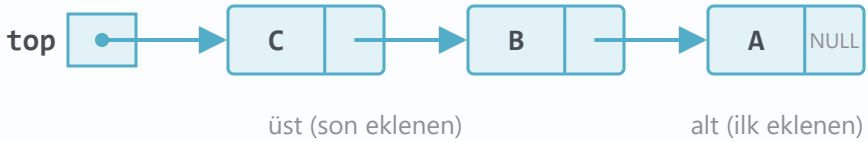
DEĞİŞKENLER

`s = "TIGIY"` `i = 5` `y.ust = -1`

ÇAĞRI YIĞINI

- **ters_cevir(s)**
- main()

Bağlı Liste Kullanarak Yığın



A, B, C sırasıyla eklendi: `top` liste başını gösterir.

- Her öğe bir **düğüm** (node): `deger` + sonraki düğüme `bag`.
- Ekleme ve silme **liste başında**: ikisi de O(1), kaydırma yok.
- Kapasite sınırı yok: yalnız `malloc` başarısız olursa taşma olur.

Neden baş, neden son değil? Silerken son öğeden geri gelmek gerekir; tek yönlü bağlı listede bir önceki düğümü bulmak O(n)'dir; bu yüzden **top = liste başı**.

HOCADAN "Top, liste başını tutar. Silerken son öğeden geriye gelmek gerekeceği için ya iki işaretçiyle son tutulur ya da tek işaretçiyle baş tutulur."

Kod İzleme: Bağlı `push`

*top NULL

Başlangıç: `dugum *top = NULL;` (boş yığın). Sırayla push 10, 20, 30.

Adım 1 / 14

```
1 bool push(dugum **top, int x) {
2   dugum *yeni = malloc(sizeof *yeni);
3   if (yeni == NULL) return false;  /* bellek yok */
4   yeni->deger = x;
5   yeni->bag = *top;       /* eski üstü göster */
6   *top = yeni;            /* yeni üst */
7   return true;
8 }
```

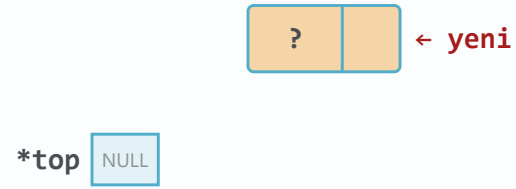
DEĞİŞKENLER

top(main) = NULL

ÇAĞRI YIĞINI

main()

Kod İzleme: Bağlı `push`



malloc yeni bir düğüm ayırır; NULL değilse devam. Alanları henüz belirsiz.


```
1 bool push(dugum **top, int x) {
▶   dugum *yeni = malloc(sizeof *yeni);
▶   if (yeni == NULL) return false;  /* bellek yok */
4   yeni->deger = x;
5   yeni->bag = *top;       /* eski üstü göster */
6   *top = yeni;            /* yeni üst */
7   return true;
8 }
```

DEĞİŞKENLER

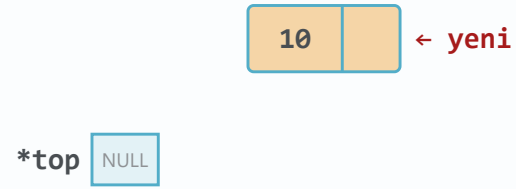
`*top = NULL` `x = 10` `yeni = düğüm(?)`

ÇAĞRI YIĞINI

`push(&top, 10)`

`main()`

Kod İzleme: Bağlı `push`



yeni->deger = 10.


```
1 bool push(dugum **top, int x) {
2   dugum *yeni = malloc(sizeof *yeni);
3   if (yeni == NULL) return false;  /* bellek yok */
▶   yeni->deger = x;
5   yeni->bag = *top;       /* eski üstü göster */
6   *top = yeni;            /* yeni üst */
7   return true;
8 }
```

DEĞİŞKENLER

`*top = NULL` `x = 10`

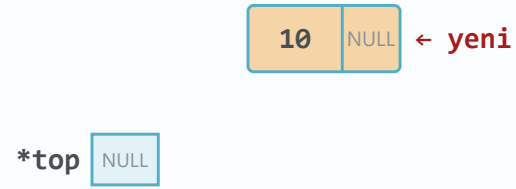
`yeni = düğüm(10)`

ÇAĞRI YIĞINI

`push(&top, 10)`

`main()`

Kod İzleme: Bağlı `push`



yeni->bag = *top: yeni düğüm eski üstü (NULL) gösterir. Sıra önemli: önce bağla, sonra `*top` 'u değiştir.

Adım 4 / 14

```
1 bool push(dugum **top, int x) {
2   dugum *yeni = malloc(sizeof *yeni);
3   if (yeni == NULL) return false;  /* bellek yok */
4   yeni->deger = x;
▶   yeni->bag = *top;       /* eski üstü göster */
6   *top = yeni;            /* yeni üst */
7   return true;
8 }
```

DEĞİŞKENLER

`*top = NULL` `x = 10`

`yeni = düğüm(10)`

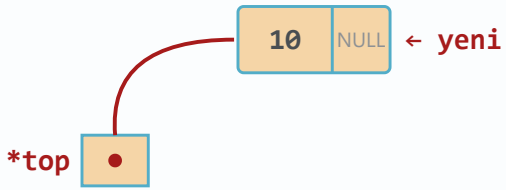
`yeni→bag = NULL`

ÇAĞRI YIĞINI

`push(&top, 10)`

`main()`

Kod İzleme: Bağlı `push`



***top = yeni**: çağıranın `top`'u artık 10'u gösterir. Yığın: 10.

Adım 5 / 14

```
1 bool push(dugum **top, int x) {
2   dugum *yeni = malloc(sizeof *yeni);
3   if (yeni == NULL) return false;  /* bellek yok */
4   yeni->deger = x;
5   yeni->bag = *top;       /* eski üstü göster */
▶   *top = yeni;            /* yeni üst */
7   return true;
8 }
```

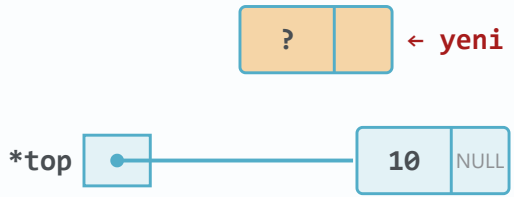
DEĞİŞKENLER

- `*top` = **düğüm(10)**
- `x` = **10**
- `yeni` = **düğüm(10)**

ÇAĞRI YIĞINI

- `push(&top, 10)`
- `main()`

Kod İzleme: Bağlı `push`



malloc yeni bir düğüm ayırır; NULL değilse devam. Alanları henüz belirsiz.

Adım 6 / 14

```
1 bool push(dugum **top, int x) {
▶   dugum *yeni = malloc(sizeof *yeni);
▶   if (yeni == NULL) return false;  /* bellek yok */
4   yeni->deger = x;
5   yeni->bag = *top;       /* eski üstü göster */
6   *top = yeni;            /* yeni üst */
7   return true;
8 }
```

DEĞİŞKENLER

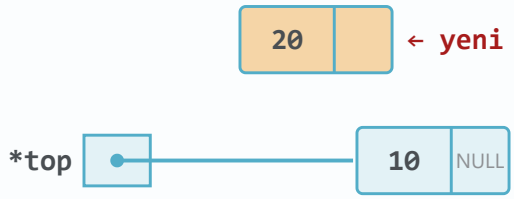
`*top = düğüm(10)` `x = 20` `yeni = düğüm(?)`

ÇAĞRI YIĞINI

`push(&top, 20)`

`main()`

Kod İzleme: Bağlı `push`



yeni->deger = 20.


```
1 bool push(dugum **top, int x) {
2   dugum *yeni = malloc(sizeof *yeni);
3   if (yeni == NULL) return false;  /* bellek yok */
▶   yeni->deger = x;
5   yeni->bag = *top;       /* eski üstü göster */
6   *top = yeni;            /* yeni üst */
7   return true;
8 }
```

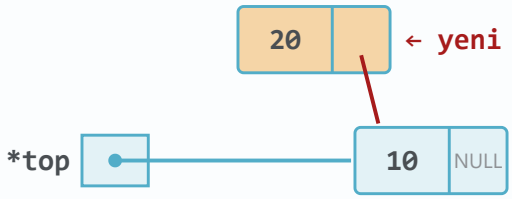
DEĞİŞKENLER

`*top` = **düğüm(10)** | `x` = **20** | `yeni` = **düğüm(20)**

ÇAĞRI YIĞINI

- **push(&top, 20)**
- main()

Kod İzleme: Bağlı `push`



yeni->bag = *top: yeni düğüm eski üstü (düğüm(10)) gösterir. Sıra önemli: önce bağla, sonra `*top` 'u değiştir.

Adım 8 / 14

```
1 bool push(dugum **top, int x) {
2   dugum *yeni = malloc(sizeof *yeni);
3   if (yeni == NULL) return false;  /* bellek yok */
4   yeni->deger = x;
▶   yeni->bag = *top;       /* eski üstü göster */
6   *top = yeni;            /* yeni üst */
7   return true;
8 }
```

DEĞİŞKENLER

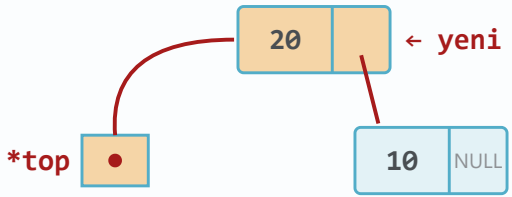
`*top` = **düğüm(10)** | `x` = **20** | `yeni` = **düğüm(20)** | `yeni→bag` = **düğüm(10)**

ÇAĞRI YIĞINI

`push(&top, 20)`

`main()`

Kod İzleme: Bağlı `push`



***top = yeni**: çağıranın `top`'u artık 20'yi gösterir. Yığın: 20 → 10.

Adım 9 / 14

```
1 bool push(dugum **top, int x) {
2   dugum *yeni = malloc(sizeof *yeni);
3   if (yeni == NULL) return false;  /* bellek yok */
4   yeni->deger = x;
5   yeni->bag = *top;       /* eski üstü göster */
▶   *top = yeni;            /* yeni üst */
7   return true;
8 }
```

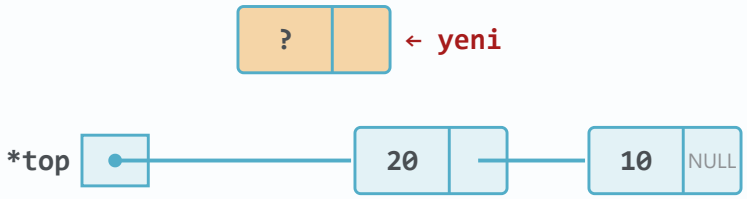
DEĞİŞKENLER

`*top` = **düğüm(20)** | `x` = **20** | `yeni` = **düğüm(20)**

ÇAĞRI YIĞINI

- `push(&top, 20)`
- `main()`

Kod İzleme: Bağlı `push`



malloc yeni bir düğüm ayırır; NULL değilse devam. Alanları henüz belirsiz.

Adım 10 / 14

```
1 bool push(dugum **top, int x) {
▶   dugum *yeni = malloc(sizeof *yeni);
▶   if (yeni == NULL) return false;  /* bellek yok */
4   yeni->deger = x;
5   yeni->bag = *top;       /* eski üstü göster */
6   *top = yeni;            /* yeni üst */
7   return true;
8 }
```

DEĞİŞKENLER

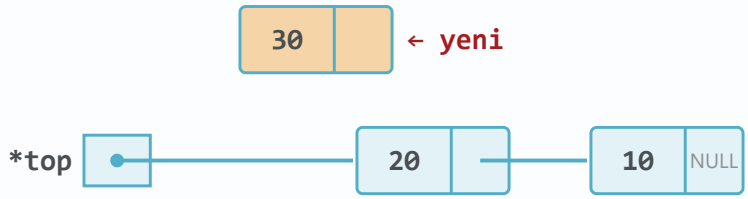
`*top = düğüm(20)` `x = 30` `yeni = düğüm(?)`

ÇAĞRI YIĞINI

`push(&top, 30)`

`main()`

Kod İzleme: Bağlı `push`



yeni->deger = 30.


```
1 bool push(dugum **top, int x) {
2   dugum *yeni = malloc(sizeof *yeni);
3   if (yeni == NULL) return false;  /* bellek yok */
▶   yeni->deger = x;
5   yeni->bag = *top;       /* eski üstü göster */
6   *top = yeni;            /* yeni üst */
7   return true;
8 }
```

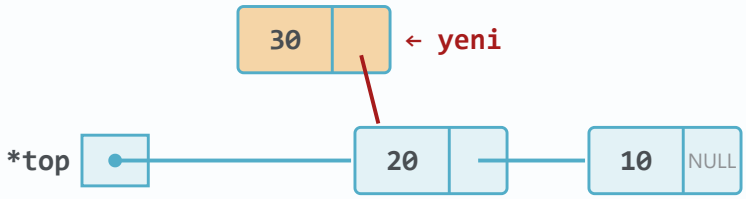
DEĞİŞKENLER

`*top` = **düğüm(20)** | `x` = **30** | `yeni` = **düğüm(30)**

ÇAĞRI YIĞINI

- **push(&top, 30)**
- main()

Kod İzleme: Bağlı `push`



yeni->bag = *top: yeni düğüm eski üstü (düğüm(20)) gösterir. Sıra önemli: önce bağla, sonra `*top` 'u değiştir.

Adım 12 / 14

```
1 bool push(dugum **top, int x) {
2   dugum *yeni = malloc(sizeof *yeni);
3   if (yeni == NULL) return false;  /* bellek yok */
4   yeni->deger = x;
▶   yeni->bag = *top;       /* eski üstü göster */
6   *top = yeni;            /* yeni üst */
7   return true;
8 }
```

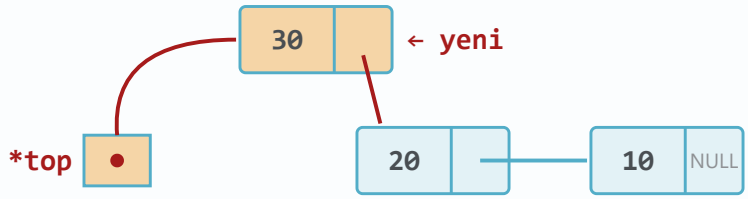
DEĞİŞKENLER

- `*top` = **düğüm(20)**
- `x` = **30**
- `yeni` = **düğüm(30)**
- `yeni→bag` = **düğüm(20)**

ÇAĞRI YIĞINI

- **push(&top, 30)**
- main()

Kod İzleme: Bağlı `push`



***top = yeni**: çağıranın `top`'u artık 30'u gösterir. Yığın: 30 → 20 → 10.

Adım 13 / 14

```
1 bool push(dugum **top, int x) {
2   dugum *yeni = malloc(sizeof *yeni);
3   if (yeni == NULL) return false;  /* bellek yok */
4   yeni->deger = x;
5   yeni->bag = *top;       /* eski üstü göster */
▶   *top = yeni;            /* yeni üst */
7   return true;
8 }
```

DEĞİŞKENLER

- `*top = düğüm(30)`
- `x = 30`
- `yeni = düğüm(30)`

ÇAĞRI YIĞINI

- `push(&top, 30)`
- `main()`

Kod İzleme: Bağlı `push`



Üç push sonrası: 30 → 20 → 10 → NULL. Her push sabit sayıda iş: **O(1)**.

Adım 14 / 14

```
1 bool push(dugum **top, int x) {
2   dugum *yeni = malloc(sizeof *yeni);
3   if (yeni == NULL) return false;  /* bellek yok */
4   yeni->deger = x;
5   yeni->bag = *top;       /* eski üstü göster */
6   *top = yeni;            /* yeni üst */
7   return true;
8 }
```

DEĞİŞKENLER

`top(main) = düğüm(30)`

ÇAĞRI YIĞINI

`main()`

Kod İzleme: Bağlı `pop`



Yığın: 30 → 20 → 10. Dört kez `pop` çağrılacak; her başarılı sonucu `main` yazdırır.

Adım 1 / 14

```
1 bool pop(dugum **top, int *x) {
2   dugum *eski = *top;      /* malloc yok! */
3   if (eski == NULL) return false;  /* boş */
4   *x = eski->deger;
5   *top = eski->bag;        /* üstü ilerlet */
6   free(eski);              /* belleği bırak */
7   return true;
8 }
```

DEĞİŞKENLER

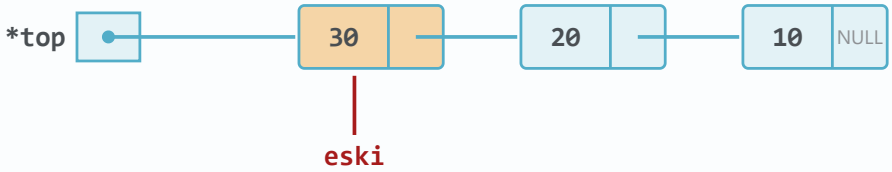
`top(main) = düğüm(30)`

ÇAĞRI YIĞINI

`main()`

ÇIKTI

Kod İzleme: Bağlı `pop`



eski = *top: yalnızca bir işaretçi kopyası; yeni bellek **ayrılmaz**.

Adım 2 / 14

```
1 bool pop(dugum **top, int *x) {
▶   dugum *eski = *top;       /* malloc yok! */
3   if (eski == NULL) return false;  /* boş */
4   *x = eski->deger;
5   *top = eski->bag;         /* üstü ilerlet */
6   free(eski);               /* belleği bırak */
7   return true;
8 }
```

DEĞİŞKENLER

`*top` = **düğüm(30)**

`eski` = **düğüm(30)**

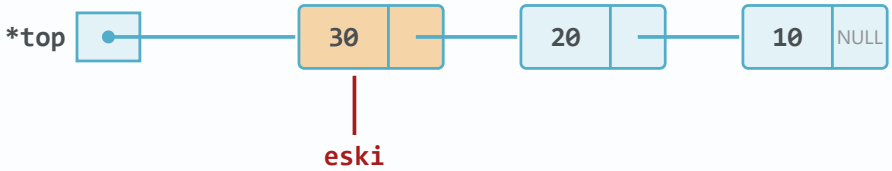
ÇAĞRI YIĞINI

`pop(&top, &x)`

`main()`

ÇIKTI

Kod İzleme: Bağlı `pop`



***x = 30** (değer, düğüm silinmeden önce alınmalı).

Adım 3 / 14

```
1 bool pop(dugum **top, int *x) {
2   dugum *eski = *top;      /* malloc yok! */
3   if (eski == NULL) return false;  /* boş */
▶   *x = eski->deger;
5   *top = eski->bag;        /* üstü ilerlet */
6   free(eski);              /* belleği bırak */
7   return true;
8 }
```

DEĞİŞKENLER

`*top` = **düğüm(30)**

`eski` = **düğüm(30)** `*x` = **30**

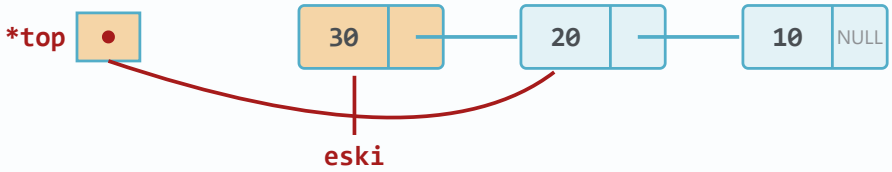
ÇAĞRI YIĞINI

pop(&top, &x)

main()

ÇIKTI

Kod İzleme: Bağlı `pop`



***top = eski->bag** → düğüm(20).

Adım 4 / 14

```
1 bool pop(dugum **top, int *x) {
2   dugum *eski = *top;      /* malloc yok! */
3   if (eski == NULL) return false;  /* boş */
4   *x = eski->deger;
▶   *top = eski->bag;        /* üstü ilerlet */
6   free(eski);              /* belleği bırak */
7   return true;
8 }
```

DEĞİŞKENLER

`*top` = **düğüm(20)**

`eski` = **düğüm(30)** `*x` = **30**

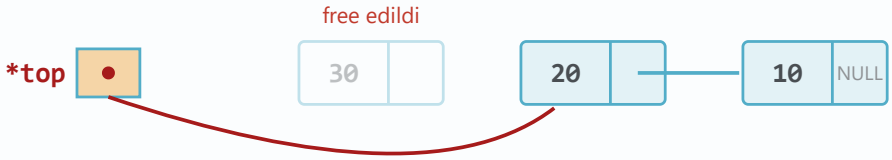
ÇAĞRI YIĞINI

`pop(&top, &x)`

`main()`

ÇIKTI

Kod İzleme: Bağlı `pop`



free(eski): 30 düğümünün belleği sisteme döner. `main` 30 yazar.

Adım 5 / 14

```
1 bool pop(dugum **top, int *x) {
2   dugum *eski = *top;      /* malloc yok! */
3   if (eski == NULL) return false;  /* boş */
4   *x = eski->deger;
5   *top = eski->bag;        /* üstü ilerlet */
▶   free(eski);              /* belleği bırak */
7   return true;
8 }
```

DEĞİŞKENLER

`*top = düğüm(20)`

`eski = (serbest)` `*x = 30`

ÇAĞRI YIĞINI

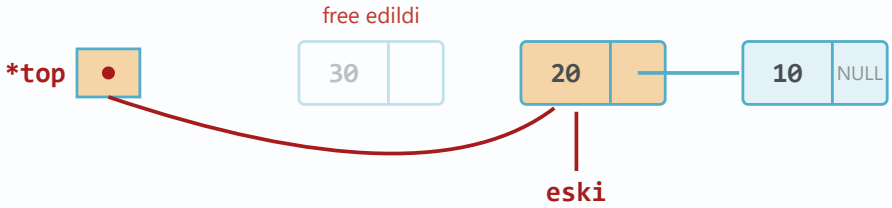
`pop(&top, &x)`

`main()`

ÇIKTI

```
30
```

Kod İzleme: Bağlı `pop`



eski = *top: yalnızca bir işaretçi kopyası; yeni bellek **ayrılmaz**.

Adım 6 / 14

```
1 bool pop(dugum **top, int *x) {
▶   dugum *eski = *top;       /* malloc yok! */
3   if (eski == NULL) return false;  /* boş */
4   *x = eski->deger;
5   *top = eski->bag;         /* üstü ilerlet */
6   free(eski);               /* belleği bırak */
7   return true;
8 }
```

DEĞİŞKENLER

`*top` = **düğüm(20)**

`eski` = **düğüm(20)**

ÇAĞRI YIĞINI

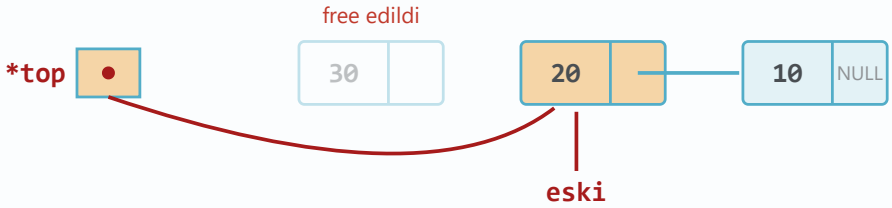
`pop(&top, &x)`

`main()`

ÇIKTI

```
30
```

Kod İzleme: Bağlı `pop`



***x = 20** (değer, düğüm silinmeden önce alınmalı).

Adım 7 / 14

```
1 bool pop(dugum **top, int *x) {
2   dugum *eski = *top;      /* malloc yok! */
3   if (eski == NULL) return false;  /* boş */
▶   *x = eski->deger;
5   *top = eski->bag;        /* üstü ilerlet */
6   free(eski);              /* belleği bırak */
7   return true;
8 }
```

DEĞİŞKENLER

`*top` = **düğüm(20)**

`eski` = **düğüm(20)** `*x` = **20**

ÇAĞRI YIĞINI

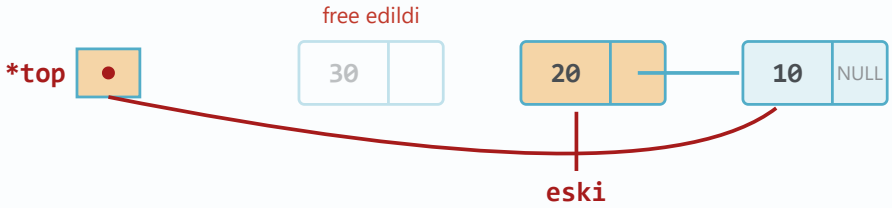
pop(&top, &x)

main()

ÇIKTI

```
30
```

Kod İzleme: Bağlı `pop`



***top = eski->bag** → düğüm(10).

Adım 8 / 14

```
1 bool pop(dugum **top, int *x) {
2   dugum *eski = *top;      /* malloc yok! */
3   if (eski == NULL) return false;  /* boş */
4   *x = eski->deger;
▶   *top = eski->bag;        /* üstü ilerlet */
6   free(eski);              /* belleği bırak */
7   return true;
8 }
```

DEĞİŞKENLER

`*top` = **düğüm(10)**

`eski` = **düğüm(20)** `*x` = **20**

ÇAĞRI YIĞINI

`pop(&top, &x)`

`main()`

ÇIKTI

```
30
```

Kod İzleme: Bağlı `pop`



free(eski): 20 düğümünün belleği sisteme döner. `main` 20 yazar.

Adım 9 / 14

```
1 bool pop(dugum **top, int *x) {
2   dugum *eski = *top;      /* malloc yok! */
3   if (eski == NULL) return false;  /* boş */
4   *x = eski->deger;
5   *top = eski->bag;        /* üstü ilerlet */
▶   free(eski);              /* belleği bırak */
7   return true;
8 }
```

DEĞİŞKENLER

`*top = düğüm(10)`

`eski = (serbest)` `*x = 20`

ÇAĞRI YIĞINI

`pop(&top, &x)`

`main()`

ÇIKTI

```
30 20
```

Kod İzleme: Bağlı `pop`



eski = *top: yalnızca bir işaretçi kopyası; yeni bellek **ayrılmaz**.

Adım 10 / 14

```
1 bool pop(dugum **top, int *x) {
▶   dugum *eski = *top;       /* malloc yok! */
3   if (eski == NULL) return false;  /* boş */
4   *x = eski->deger;
5   *top = eski->bag;         /* üstü ilerlet */
6   free(eski);               /* belleği bırak */
7   return true;
8 }
```

DEĞİŞKENLER

`*top` = **düğüm(10)**

`eski` = **düğüm(10)**

ÇAĞRI YIĞINI

`pop(&top, &x)`

`main()`

ÇIKTI

```
30 20
```

Kod İzleme: Bağlı `pop`



***x = 10** (değer, düğüm silinmeden önce alınmalı).

Adım 11 / 14

```
1 bool pop(dugum **top, int *x) {
2   dugum *eski = *top;      /* malloc yok! */
3   if (eski == NULL) return false;  /* boş */
▶   *x = eski->deger;
5   *top = eski->bag;        /* üstü ilerlet */
6   free(eski);              /* belleği bırak */
7   return true;
8 }
```

DEĞİŞKENLER

`*top` = **düğüm(10)**

`eski` = **düğüm(10)** `*x` = **10**

ÇAĞRI YIĞINI

`pop(&top, &x)`

`main()`

ÇIKTI

```
30 20
```

Kod İzleme: Bağlı `pop`



***top = eski->bag** → NULL.

Adım 12 / 14

```
1 bool pop(dugum **top, int *x) {
2   dugum *eski = *top;       /* malloc yok! */
3   if (eski == NULL) return false;  /* boş */
4   *x = eski->deger;
▶   *top = eski->bag;         /* üstü ilerlet */
6   free(eski);               /* belleği bırak */
7   return true;
8 }
```

DEĞİŞKENLER

`*top = NULL` `eski = düğüm(10)` `*x = 10`

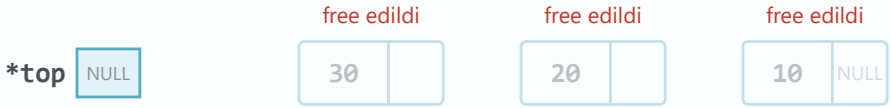
ÇAĞRI YIĞINI

`pop(&top, &x)`
`main()`

ÇIKTI

```
30 20
```

Kod İzleme: Bağlı `pop`



free(eski): 10 düğümünün belleği sisteme döner. `main` 10 yazar.

Adım 13 / 14

```
1 bool pop(dugum **top, int *x) {
2   dugum *eski = *top;      /* malloc yok! */
3   if (eski == NULL) return false;  /* boş */
4   *x = eski->deger;
5   *top = eski->bag;        /* üstü ilerlet */
▶   free(eski);              /* belleği bırak */
7   return true;
8 }
```

DEĞİŞKENLER

`*top = NULL` `eski = (serbest)` `*x = 10`

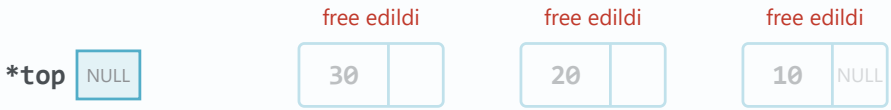
ÇAĞRI YIĞINI

`pop(&top, &x)`
`main()`

ÇIKTI

```
30 20 10
```

Kod İzleme: Bağlı `pop`



eski = NULL → yığın boş, **return false**. Döngü burada durur.

Adım 14 / 14

```
1 bool pop(dugum **top, int *x) {
▶   dugum *eski = *top;      /* malloc yok! */
▶   if (eski == NULL) return false;  /* boş */
4   *x = eski->deger;
5   *top = eski->bag;        /* üstü ilerlet */
6   free(eski);              /* belleği bırak */
7   return true;
8 }
```

DEĞİŞKENLER

`*top = NULL` `eski = NULL`

ÇAĞRI YIĞINI

`pop(&top, &x)`

`main()`

ÇIKTI

```
30 20 10
```

Bağlı Yığında Bellek Sorumluluğu

```c
typedef struct dugum {
    int deger;
    struct dugum *bag;
} dugum;

void yigit_temizle(dugum **top) {
    while (*top != NULL) {
        dugum *eski = *top;
        *top = eski->bag;
        free(eski);
    }
}                   /* sonunda *top == NULL */
```

- `eski` bir **işaretçi değişkenidir**, yeni düğüm değil: ona `malloc` gerekmez.
- Önce bağ kurtarılır (`*top = eski->bag`), **sonra** `free` çağrılır: tersi serbest bırakılmış belleği okur.
- Program bitmeden `yigit_temizle` çağrılır: her `malloc` 'un bir `free` 'si olmalı.

HOCADAN "Her pointer'a mallocla yer ayırmak gerekmiyor, yok öyle bir şey!"

HOCADAN "Pointerlar bu yaşlarda öğrenilirse öğrenilir; yoksa 4. sınıfta bile pointer öğrenilmemiş olur."

Dizi Tabanlı mı, Bağlı mı?

Ölçüt	Sabit dizi	Büyüyen dizi	Bağlı liste
Kapasite	Derleme anında sabit	Gerektikçe ×2	Bellek bitene kadar
push / pop	O(1)	O(1) amortize	O(1), her push'ta malloc
Öğe başına ek bellek	Yok	Yok (boş kapasite hariç)	Bir işaretçi (8 bayt)
Önbellek (cache)	Çok iyi	Çok iyi	Zayıf (düğümler dağınık)
Taşma	Mümkün	Yalnız realloc başarısızsa	Yalnız malloc başarısızsa
Hata riski	Sınır denetimi	realloc sonucu	free unutmak, sarkan işaretçi

Üst sınır biliniyorsa sabit dizi; bilinmiyorsa büyüyen dizi pratikte en hızlısıdır. Bağlı yığın, düğümlerin zaten başka yapılarda yaşadığı ya da tek tek yaratılıp silindiği durumlar için uygundur.

Çağrı Yığını da Taşar

Çağrı yığını bir yığındır: her çağrı dönüş adresi, parametreler ve yerellerden oluşan bir çerçeveyi push eder, `return` onu pop eder; en son çağrılan fonksiyon ilk biter.

Sonsuz özyineleme

Taban durumu olmayan ya da ona ulaşmayan özyineleme her çağrıda çerçeve iter; sistem yığını (tipik olarak birkaç MB) dolunca program **stack overflow** ile çöker.

Büyük yerel diziler

`int a[1000000];` gibi yerel bir dizi çerçevenin içindedir; tek başına yığını taşırabilir. Büyük veriler için `malloc` ile heap bellek kullanılır.

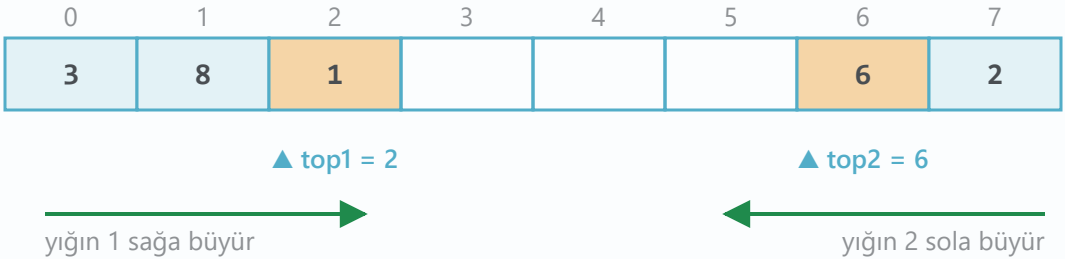
Yerel adres döndürme

`return &yerel;` hatalıdır: çerçeve pop edilince o bellek başka çağrılarca yeniden kullanılır (sarkan işaretçi).

Özyinelemeden kurtulmak

Her özyinelemeli algoritma, kendi yığınını tutan bir döngüye çevrilebilir.

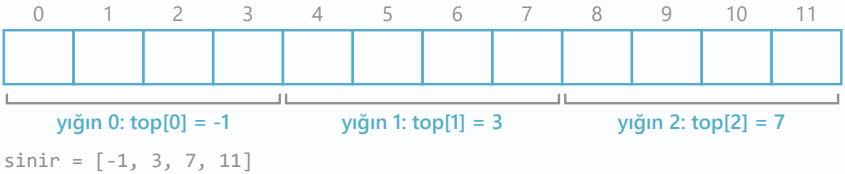
Çoklu Yığın: Önce İki Yığın



- Yığın 1 **0'dan yukarı**, yığın 2 **m−1'den aşağı** büyür.
- Boş: `top1 = -1` , `top2 = m` .
- Dolu: `top1 + 1 == top2` : ikisi ortada buluşunca.

Avantaj: bir yığın az, öteki çok öğe tutsa bile **hiç hücre boşa gitmez**; dizi ancak toplam m öğe olunca dolar. Üç ve daha fazla yığın için bu hile yetmez.

Kod İzleme: Tek Bellekte n Yığın



MEM = 12, n = 3: her yığına 4 hücre. Başta **top[i] = sinir[i] = (m/n)·i − 1**, **sinir[n] = m − 1**.

Adım 1 / 9

```
1  /* yığın i: bellek[sinir[i]+1 .. sinir[i+1]] */
2  bool ekle(int i, int x) {
3    if (top[i] == sinir[i + 1]) return false; /* dolu */
4    bellek[++top[i]] = x;
5    return true;
6  }
7  bool al(int i, int *x) {
8    if (top[i] == sinir[i]) return false;     /* boş */
9    *x = bellek[top[i]--];
10   return true;
11 }
```

DEĞİŞKENLER

`top = [-1, 3, 7]`

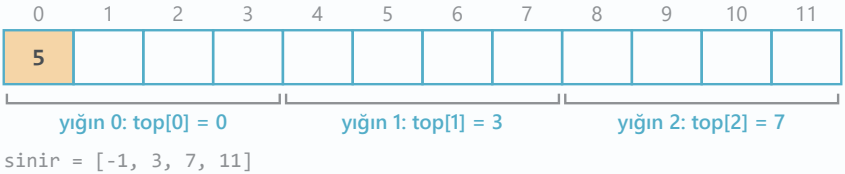
`sinir = [-1, 3, 7, 11]`

ÇAĞRI YIĞINI

`main()`

ÇIKTI

Kod İzleme: Tek Bellekte n Yığın



Yer var; **bellek[0] = 5**, top[0] = 0.

Adım 2 / 9

```
1  /* yığın i: bellek[sinir[i]+1 .. sinir[i+1]] */
2  bool ekle(int i, int x) {
▶    if (top[i] == sinir[i + 1]) return false; /* dolu */
▶    bellek[++top[i]] = x;
5    return true;
6  }
7  bool al(int i, int *x) {
8    if (top[i] == sinir[i]) return false;     /* boş */
9    *x = bellek[top[i]--];
10   return true;
11 }
```

DEĞİŞKENLER

`i = 0` `x = 5` `top = [0, 3, 7]`

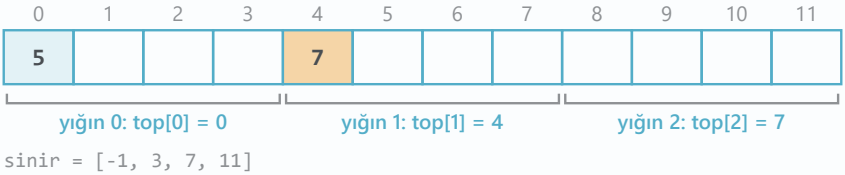
ÇAĞRI YIĞINI

`ekle(0, 5)`

`main()`

ÇIKTI

Kod İzleme: Tek Bellekte n Yığın



Yer var; **bellek[4] = 7**, top[1] = 4.

Adım 3 / 9

```
1  /* yığın i: bellek[sinir[i]+1 .. sinir[i+1]] */
2  bool ekle(int i, int x) {
▶    if (top[i] == sinir[i + 1]) return false; /* dolu */
▶    bellek[++top[i]] = x;
5    return true;
6  }
7  bool al(int i, int *x) {
8    if (top[i] == sinir[i]) return false;     /* boş */
9    *x = bellek[top[i]--];
10   return true;
11 }
```

DEĞİŞKENLER

`i = 1` `x = 7` `top = [0, 4, 7]`

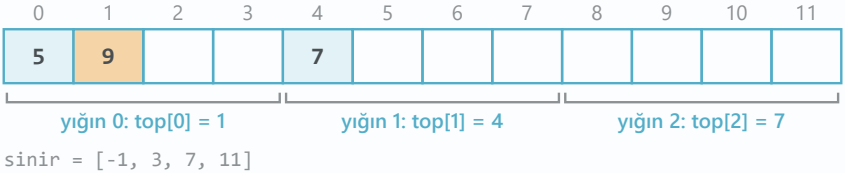
ÇAĞRI YIĞINI

`ekle(1, 7)`

`main()`

ÇIKTI

Kod İzleme: Tek Bellekte n Yığın



Yer var; **bellek[1] = 9**, top[0] = 1.

Adım 4 / 9

```
1  /* yığın i: bellek[sinir[i]+1 .. sinir[i+1]] */
2  bool ekle(int i, int x) {
▶    if (top[i] == sinir[i + 1]) return false; /* dolu */
▶    bellek[++top[i]] = x;
5    return true;
6  }
7  bool al(int i, int *x) {
8    if (top[i] == sinir[i]) return false;     /* boş */
9    *x = bellek[top[i]--];
10   return true;
11 }
```

DEĞİŞKENLER

`i = 0` `x = 9` `top = [1, 4, 7]`

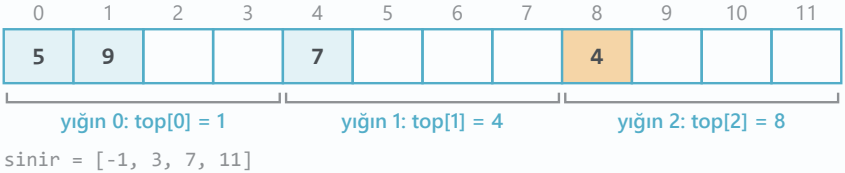
ÇAĞRI YIĞINI

`ekle(0, 9)`

`main()`

ÇIKTI

Kod İzleme: Tek Bellekte n Yığın



Yer var; **bellek[8] = 4**, top[2] = 8.

Adım 5 / 9

```
1  /* yığın i: bellek[sinir[i]+1 .. sinir[i+1]] */
2  bool ekle(int i, int x) {
▶    if (top[i] == sinir[i + 1]) return false; /* dolu */
▶    bellek[++top[i]] = x;
5    return true;
6  }
7  bool al(int i, int *x) {
8    if (top[i] == sinir[i]) return false;     /* boş */
9    *x = bellek[top[i]--];
10   return true;
11 }
```

DEĞİŞKENLER

`i = 2` `x = 4` `top = [1, 4, 8]`

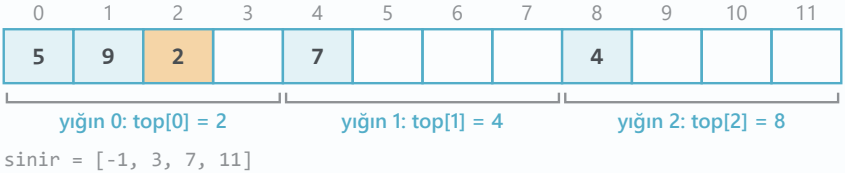
ÇAĞRI YIĞINI

`ekle(2, 4)`

`main()`

ÇIKTI

Kod İzleme: Tek Bellekte n Yığın



Yer var; **bellek[2] = 2**, top[0] = 2.

Adım 6 / 9

```
1  /* yığın i: bellek[sinir[i]+1 .. sinir[i+1]] */
2  bool ekle(int i, int x) {
▶    if (top[i] == sinir[i + 1]) return false; /* dolu */
▶    bellek[++top[i]] = x;
5    return true;
6  }
7  bool al(int i, int *x) {
8    if (top[i] == sinir[i]) return false;     /* boş */
9    *x = bellek[top[i]--];
10   return true;
11 }
```

DEĞİŞKENLER

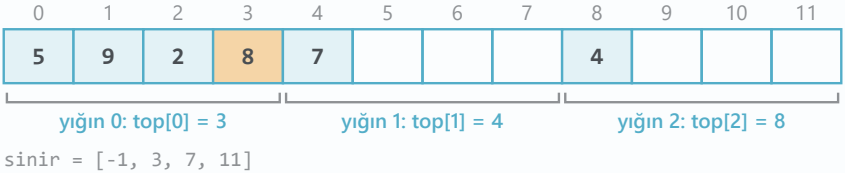
`i = 0` `x = 2` `top = [2, 4, 8]`

ÇAĞRI YIĞINI

- `ekle(0, 2)`
- `main()`

ÇIKTI

Kod İzleme: Tek Bellekte n Yığın



Yer var; **bellek[3] = 8**, top[0] = 3.

Adım 7 / 9

```
1  /* yığın i: bellek[sinir[i]+1 .. sinir[i+1]] */
2  bool ekle(int i, int x) {
▶    if (top[i] == sinir[i + 1]) return false; /* dolu */
▶    bellek[++top[i]] = x;
5    return true;
6  }
7  bool al(int i, int *x) {
8    if (top[i] == sinir[i]) return false;     /* boş */
9    *x = bellek[top[i]--];
10   return true;
11 }
```

DEĞİŞKENLER

`i = 0` `x = 8` `top = [3, 4, 8]`

ÇAĞRI YIĞINI

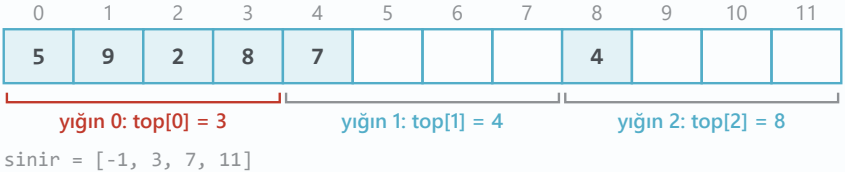
`ekle(0, 8)`

`main()`

ÇIKTI

ÇOKLU YIĞIN · KODU ADIM ADIM ÇALIŞTIR

Kod İzleme: Tek Bellekte n Yığın



top[0] == sinir[1] (3 == 3) → **yığın 0 dolu**, oysa bellekte 6 boş hücre var! Horowitz'in `stack_full` 'u burada yığınları kaydırıp yer açar.

Adım 8 / 9

```
1  /* yığın i: bellek[sinir[i]+1 .. sinir[i+1]] */
2  bool ekle(int i, int x) {
▶    if (top[i] == sinir[i + 1]) return false; /* dolu */
4    bellek[++top[i]] = x;
5    return true;
6  }
7  bool al(int i, int *x) {
8    if (top[i] == sinir[i]) return false;     /* boş */
9    *x = bellek[top[i]--];
10   return true;
11 }
```

DEĞİŞKENLER

`i = 0` `x = 6` `top = [3, 4, 8]`

ÇAĞRI YIĞINI

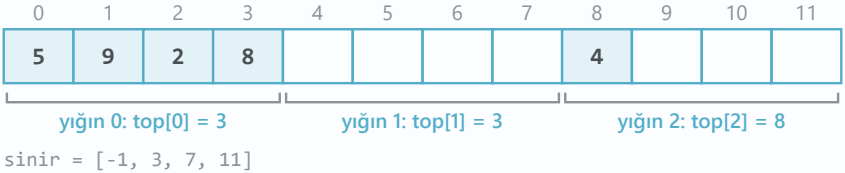
`ekle(0, 6)`

`main()`

ÇIKTI

```
ekle(0,6): dolu
```

Kod İzleme: Tek Bellekte n Yığın



top[1] ≠ sinir[1] → boş değil; ***x = 7**, top[1] = 3.

Adım 9 / 9

```
1  /* yığın i: bellek[sinir[i]+1 .. sinir[i+1]] */
2  bool ekle(int i, int x) {
3    if (top[i] == sinir[i + 1]) return false; /* dolu */
4    bellek[++top[i]] = x;
5    return true;
6  }
7  bool al(int i, int *x) {
▶    if (top[i] == sinir[i]) return false;     /* boş */
▶    *x = bellek[top[i]--];
10   return true;
11 }
```

DEĞİŞKENLER

`i = 1` `*x = 7` `top = [3, 3, 8]`

ÇAĞRI YIĞINI

- `al(1, &x)`
- `main()`

ÇIKTI

```
ekle(0,6): dolu
al(1): 7
```

ÇOKLU YIĞIN

n Yığının Paylaşımı: Artılar ve Eksiler

Tek boyutlu bir dizi (`memory[MEMORY_SIZE]`) n kesime (segment) bölünerek n yığın kurulabilir; yığınların `top` ve `sınır` değerleri iki yardımcı dizide tutulur.

Artılar

+ Tek `malloc` , tek dizi: yönetimi kolay.
+ push/pop yine O(1).
+ Aynı kuyruk fikri de uygulanabilir (n kuyruk).

Eksiler

– Bir kesim dolunca, bellekte yer olsa bile "dolu" hatası.
– Yer açmak için yığınları kaydırmak O(m): kaydırma pahalıdır.
– Kesim boyutlarını önceden kestirmek zor.

Horowitz–Sahni'nin çözümü: `stack_full(i)` , önce sağdaki, yoksa soldaki boş yeri bulur ve arada kalan yığınları bir hücre kaydırır. En kötü durumda her ekleme O(m) olabilir.

ÖZET

Hangi Çıkış Sırası Mümkün?

1, 2, 3, 4 bu sırayla push edilecek; aralara istenen sayıda pop konabilir. Aşağıdaki çıkış sıralarından hangileri elde edilebilir?

Çıkış sırası	Mümkün mü?
`1 2 3 4`	▶ göster
`4 3 2 1`	▶ göster
`2 4 3 1`	▶ göster
`3 1 4 2`	▶ göster

Genel kural: çıkışta bir x'ten sonra, x'ten önce itilmiş ve henüz çıkmamış öğeler **ters sırada** çıkmak zorundadır. n öğe için olası sıra sayısı Catalan sayısıdır (n = 4 için 14).

ÖZET

Karmaşıklık Özeti

İşlem	Sabit dizi	Büyüyen dizi	Bağlı
push	O(1)	O(1) amortize, en kötü O(n)	O(1)
pop	O(1)	O(1)	O(1)
peek / bos / dolu	O(1)	O(1)	O(1)
n öğelik bellek	MAX (sabit)	≤ 2n	n · (öğe + işaretçi)
Temizleme	O(1)	O(1) (tek free)	O(n)
fakt(n) çağrı yığını	O(n) çerçeve (gerçeklemeden bağımsız)		
Çoklu yığın ekle	O(1); yer açma O(m)	—	—

ÖZET

Sık Yapılan Hatalar

1. Boş yığından pop / dolu yığına push: sınır denetimini atlamak.
2. `*top--` yazmak: işaretçiyi azaltır; doğrusu `(*top)--`.
3. `top` değerini öğe sayısıyla karıştırmak (sayı = top + 1).
4. Bağlı push'ta önce `*top = yeni` yazıp eski listeyi kaybetmek.
5. pop'ta değeri `free`'den sonra okumak ya da düğümü hiç bırakmamak.
6. Geçici işaretçiye gereksiz `malloc` yapıp bellek sızdırmak.

ÖZET

Alıştırmalar

1. Dizi yığınında `yigit_ekle`'yi `exit` yerine `bool` döndürecek biçimde yeniden yaz.
2. Tek dizide iki yığın için `ekle(k, x)` ve `al(k)` yordamlarını yaz (k = 1 ya da 2).
3. Bağlı yığını, `top` ile birlikte `boyut` alanı da tutan bir `struct` içine taşı.
4. Bir dizgeyi yığın kullanarak ters çeviren `void ters(char *s)` yordamını yaz.
5. Yalnız push/pop kullanarak bir yığının en alttaki öğesini döndüren yordamı yaz (yardımcı yığın serbest).
6. `fakt(5)` çağrısında yığında en çok kaç çerçeve olur? `main` dahil çiz.

▶ **İpuçları / cevaplar**

ÖZET

Bölüm Özeti

LIFO
Ekleme ve silme yalnız üstte; son giren ilk çıkar.

Dizi + top
top = −1 boş, top = MAX − 1 dolu; `++*top` ve `(*top)--`.

Hata yönetimi
exit yerine durum döndür; taşma/alt taşma çağıranın kararı.

Bağlı yığın
top = liste başı; push/pop O(1); her çıkan düğüme free.

Çağrı yığını
Her çağrı bir çerçeve: dönüş adresi, yereller, önceki çerçeve.

Çoklu yığın
Tek bellek, top[i] ve sinir[i]; kesim dolarsa kaydırma gerekir.

ÖZET

Kaynaklar ve Devam

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2. bs., Bölüm 3.1–3.2, 3.7 ve 4.3.
- T. H. Cormen vd., *Introduction to Algorithms*, 3. bs., Bölüm 10.1 (yığınlar) ve 17 (amortize analiz).
- R. Sedgewick, *Algorithms in C*, Kısım 1–4, Bölüm 4.

← Bölüm 5: Polinomlar ve Seyrek Matrisler

Ders sayfası

Bölüm 7: Kuyruklar →

Son

Bölüm 6: Yığınlar

Sonraki bölüm: Bölüm 7: Kuyruklar · Ders sayfası