Ağaçlar, Öbekler ve Huffman Kodlaması

CEN207 Veri Yapıları — Hafta 4

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

Bugünün planı (3 saat)

Saat	Konu
1	Ağaçlar, terimler, biçimler **Anim 1–2** · dolaşmalar: pre/in/post/yinelemesiz/level **Anim 3–7**
2	Dizi gösterimi **Anim 8** · ikili öbek: ekleme/çıkarma/kurma/sıralama **Anim 9–12**
3	Öncelik kuyruğu **Anim 13** · varyasyonlar **Anim 14–16** · Huffman **Anim 17–18**

Öğrenme çıktıları: ÖÇ.1 (temel veri yapılarını açıklama) · ÖÇ.2 (karmaşıklık analizi) · ÖÇ.7 (doğru yapıyı seçme)

Bu haftanın kavramları — nerede

Kavram	Nerede
Ağaç terimleri, ikili ağaç biçimleri	Bölüm 1–2
Dolaşmalar (pre/in/post/yinelemesiz/level)	Bölüm 3
Ağacın dizi gösterimi	Bölüm 4
İkili öbek, öbek sıralaması	Bölüm 5
Öncelik kuyruğu, varyasyonlar, Huffman	Bölüm 6–8

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-04/c/` ve `code/week-04/java/`
- Her programın beklenen çıktısı hafta notlarında var

Tekrar — Hafta 1–2: işaretçiler ve listeler

- `malloc` / `free`: bir kutu, bir sahip, boşken `NULL`
- Bağlı liste düğümü: bir değer + bir `next` işaretçisi
- Ağaç düğümü: bir değer + **iki** işaretçi, `left` / `right`
- Liste `NULL` de biter; ağaçta ikisi de `NULL` olan **yaprak**tır

Tekrar — Hafta 3: yığın, kuyruk, özyineleme

- Özyineleme yığının **ta kendisi**: çağrı push, dönüş pop
- Bugün: *aynı* dolaşma, kendi yığınımızla, elle
- Kuyruk (FIFO) değişmeden geri dönüyor, level order için
- Hafta 3'ün yapıları artık düğüm tutuyor, sayı değil

Haftanın haritası — bir bakışta

Ağaçlar ve dolaşmalar	Öbekler ve Huffman
Terimler, beş biçim	İkili öbek, öbek sıralaması
Beş dolaşma sırası	Öncelik kuyruğu, varyasyonlar
Dizi gösterimi	Huffman kodlaması

1. Neden Ağaç? Terimler ve Tarihçe

Başlangıç sorusu

Bir dosya yöneticisi: bir ana klasör, klasörlerin içinde klasörler, onların içinde dosyalar. Bir kök, sınırsız dallanma, ve hiçbir klasör kendi atası olamıyor.

Kısa bir tarihçe

- **1857** — Cayley, molekülleri sayarken ağaçları da sayıyor
- **1968** — Knuth, terimleri *TAOCP* Cilt 1'de sabitliyor
- Kök, yaprak, derece, seviye, yükseklik — hâlâ onun terimleri
- Matematiğin en eski biçimlerinden biri, bilgisayara ödünç

Sezgi — baş aşağı bir ağaç

- **Kök** gövdedir — **en üste** çizilir
- Her şey kökten **aşağıya** dallanır
- **Yaprak**ın çocuğu yoktur — bir dalın sonu
- Evet baş aşağı: bilgisayar bilimi hep böyle çizer

Terimler (1/3)

Terim	Anlamı
Kök (root)	Ebeveyni olmayan tek düğüm
Ebeveyn / çocuk	`A` dan `B` ye kenar varsa, `A` ebeveyn, `B` çocuk
Kardeş (sibling)	Aynı ebeveyni paylaşan iki düğüm
Yaprak (leaf)	Çocuğu olmayan düğüm — derece 0

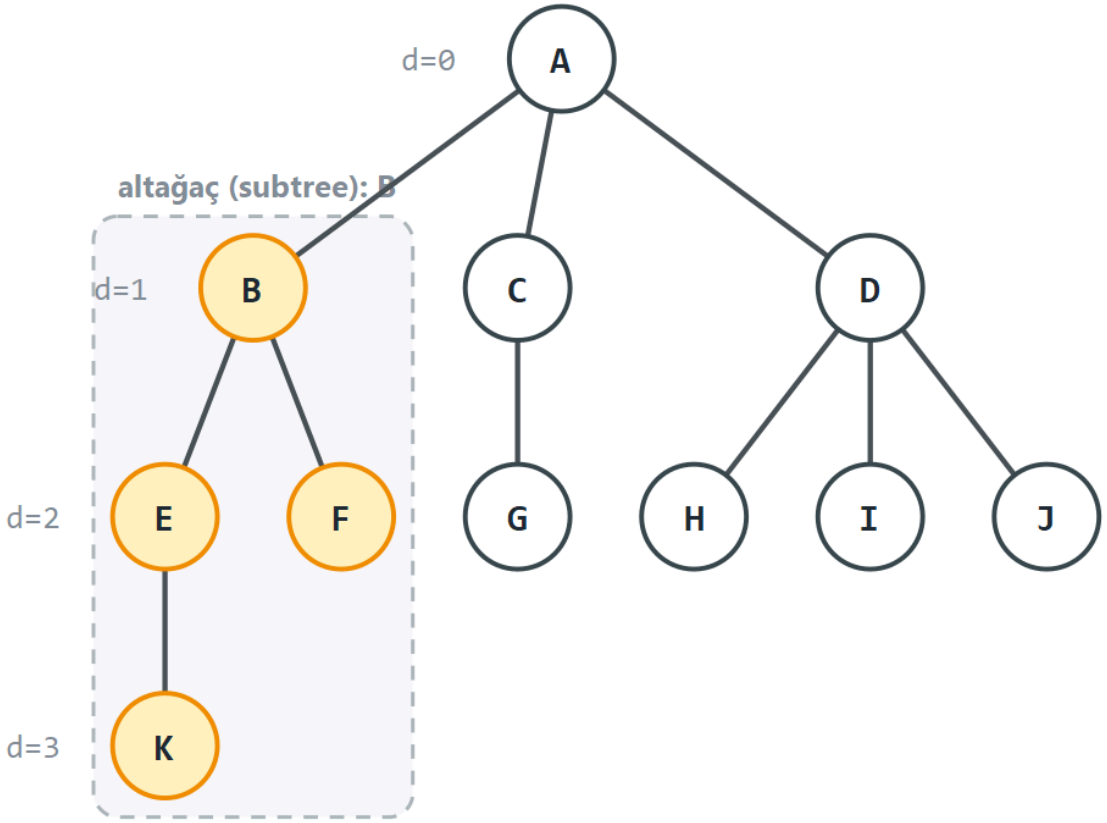
Terimler (2/3)

Terim	Anlamı
İç düğüm	En az bir çocuğu olan düğüm
Kenar (edge)	Ebeveyn-çocuk bağı; n düğüm, $n-1$ kenar
Derece (degree)	Bir düğümün çocuk sayısı
Derinlik (depth)	Kökten o düğüme inen kenar sayısı

Terimler (3/3)

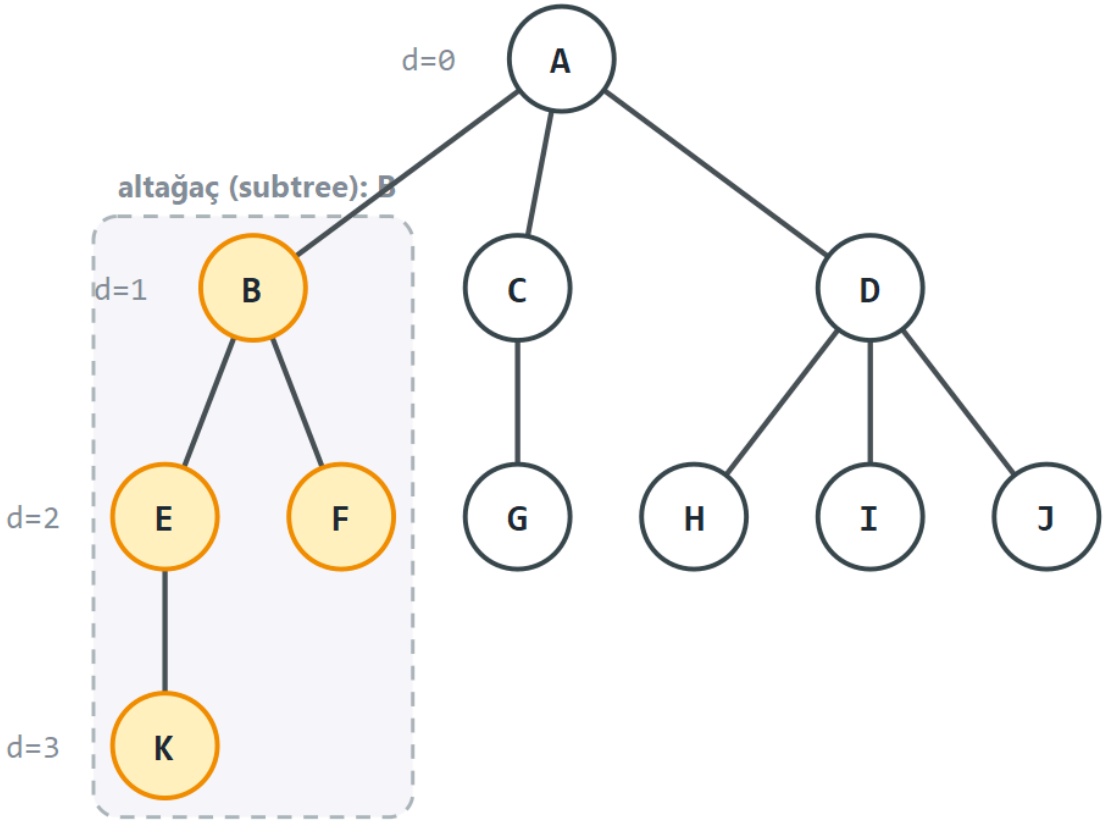
Terim	Anlamı
Yükseklik (height, düğüm)	En derin yaprağa inen en uzun yol
Yükseklik (height, ağaç)	Kökün yüksekliği
Altağaç (subtree)	Bir düğüm + altındaki her şey

Ağaç terimleri, tek tek



ALTAĞAÇ (subtree): B düğümü ve onun altındaki her şey (B, E, K, F), kesikli çerçeve içinde.

Uç durum — yıldız: kökün 10 çocuğu var



ALTAĞAÇ (subtree): B düğümü ve onun altındaki her şey (B, E, K, F), kesikli çerçeve içinde.

Kod — ağaç düğümü

```
typedef struct Node {
    char label[4];
    struct Node *children[MAX_CHILDREN];
    int child_count;   /* bu düğümün derecesi */
    int depth;
    struct Node *parent;
} Node;
```

Kod — height() (aşağıdan yukarı, özyinelemeli)

```c
int height(Node *n) {
    if (n->child_count == 0)
        return 0;          /* yaprak: yükseklik 0 */
    int best = -1;
    for (int i = 0; i < n->child_count; i++) {
        int h = height(n->children[i]);
        if (h > best) best = h;
    }
    return best + 1;       /* 1 + en yüksek çocuk */
}
```

Kod — compute_depths() (yukarıdan aşağı, BFS)

```
void compute_depths(Node *root) {
    Node *queue[MAX_NODES];
    int front = 0, rear = 0;
    root->depth = 0;
    queue[rear++] = root;
    while (front < rear) {
        Node *cur = queue[front++];
        for (int i = 0; i < cur->child_count; i++) {
            Node *ch = cur->children[i];
            ch->depth = cur->depth + 1;
            queue[rear++] = ch;
        }
    }
}
```

Karmaşıklık

- `height`: her düğüm bir kez ziyaret edilir — **O(n)**
- `compute_depths`: her düğüm bir kez ziyaret edilir — **O(n)**
- İkisi de ağacın **biçimine** bağlı değil
- İnce bir zincir, dallı bir ağaçla aynı *n* için aynı maliyette

Sık yapılan hatalar

- **Derinliki** (kökten aşağı) **yükseklikle** (yaprağa kadar) karıştırmak
- Bir yaprağın yüksekliğinin 0, 1 değil, olduğunu unutmak
- Her ağacın **ikili** olduğunu sanmak — genel bir düğümün istediği kadar çocuğu olabilir

Mini soru

18 düğümlü örnekte, `Q1` nin çocukları `S1` ve `S2` . `Q1` nin **derecesi** nedir? `Q1` nin derinliği 1 ise, `S1` in derinliği nedir?

Cevap

`Q1` nin derecesi 2dir (iki çocuğu var).
`S1` , `Q1` nin çocuğu olduğu için derinliği
`Q1` nin derinliği + 1 = 2dir.

2. İkili Ağaçlar: Biçimler ve Düğüm Sayıları

Başlangıç sorusu

Her düğümü **en çok iki** çocukla, left ve right, sınırlayın. "İstediği kadar çocuk"tan vazgeçmek neden değerli olsun?

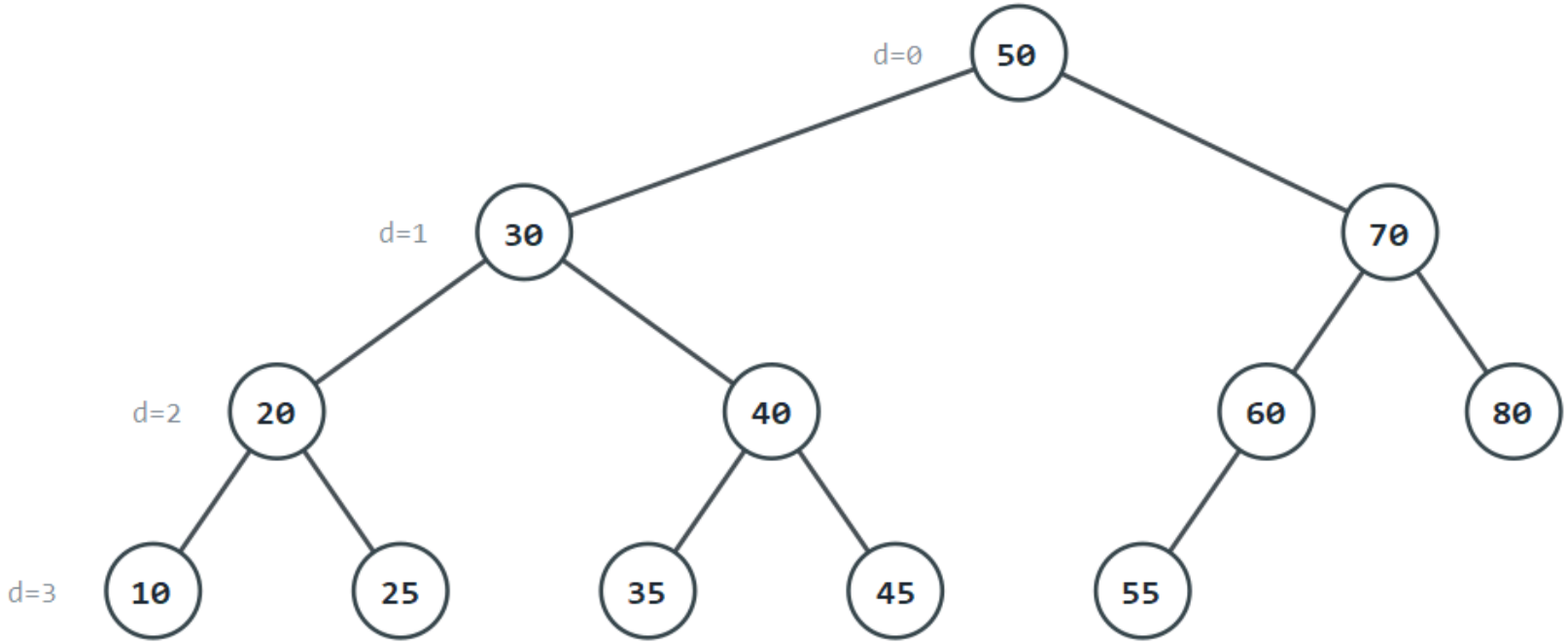
Kod — ikili ağaç düğümü

```c
typedef struct Node {
    int value;
    struct Node *left, *right;
} Node;
```

Beş biçim, kesin tanımlarıyla

Biçim	Tanım
Dolu (full)	Her düğümün 0 ya da 2 çocuğu var
Tam (complete)	Son seviye hariç her seviye dolu, soldan sağa
Mükemmel (perfect)	Dolu **ve** her yaprak aynı derinlikte
Dejenere (degenerate)	Her düğümün 0 ya da 1 çocuğu — bir zincir
Dengeli (balanced)	Sol/sağ altağaç yükseklikleri ≤ 1 fark

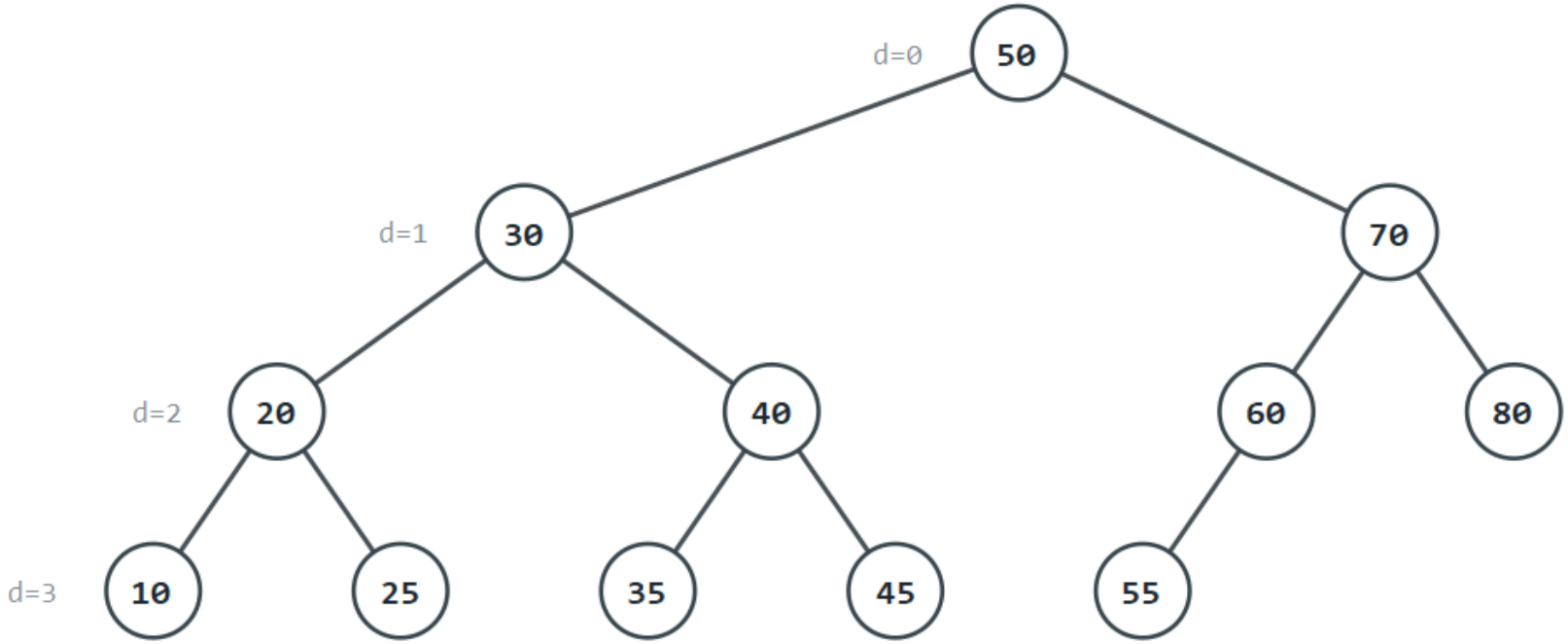
İkili ağaç biçimleri, tek tek



Özet — dolu: hayır, tam: evet, mükemmel: hayır.
dejenere: hayır, dengeli: evet (12 düğüm, yükseklik 3).

Özet: 12 düğüm, yükseklik 3. Beş sorunun cevabı yukarıda.

Uç durum — dejenere bir zincir, 10 düğüm



Özet — dolu: hayır, tam: evet, mükemmel: hayır.
dejenere: hayır, dengeli: evet (12 düğüm, yükseklik 3).

Özet: 12 düğüm, yükseklik 3. Beş sorunun cevabı yukarıda.

Kaç düğüm sığar?

- **Yükseklik *h*'de en çok:** mükemmel ağaçta $2^{h+1} - 1$ düğüm
- Seviye *k*, 2^k düğüm tutar; *h*'ye kadar toplayın
- ***n* düğüm için en az yükseklik:** $\lfloor \log_2 n \rfloor$, hiçbir dizilim daha iyisini yapamaz
- Her seviye, üstündekinin en çok iki katı düğüm tutar

Denge neden önemli

- **Dengeli** bir ağaç yüksekliği $\log_2 n$'e yakın tutar
- Kökten yapraga işlemler o zaman **O(log n)** maliyetli
- **Dejenere** bir ağacın yüksekliği $n - 1$, bir listeden farksız
- Aynı düğüm sayısı, çok farklı maliyet — biçim belirler

Kod — is_complete (boşlukları yakalamak)

```c
static bool is_complete(Node *root) {
    Node *queue[MAX_NODES]; int front = 0, rear = 0;
    queue[rear++] = root;
    bool seen_gap = false;
    while (front < rear) {
        Node *n = queue[front++];
        if (n == NULL) { seen_gap = true; continue; }
        if (seen_gap) return false;
        queue[rear++] = n->left;
        queue[rear++] = n->right;
    }
    return true;
}
```

Kod — check_balance (iki değil bir geçiş)

```
static int check_balance(Node *n) {
    if (n == NULL) return 0;
    int hl = check_balance(n->left);
    if (hl == -1) return -1;
    int hr = check_balance(n->right);
    if (hr == -1) return -1;
    if (abs(hl - hr) > 1) return -1;
    return 1 + (hl > hr ? hl : hr);
}
```

Karmaşıklık

- Beş biçim kontrolünün hepsi **O(n)** — her düğüm bir kez
- `is_complete`, kuyruğu için O(n) ekstra alana ihtiyaç duyar
- Diğer dördü yalnızca **O(h)** özyineleme yığını alanı ister

Sık yapılan hatalar

- "Tam"ın her seviyenin dolu olması demek olduğunu sanmak
- **Mükemmel**i (tam + tüm yapraklar aynı derinlikte) tam ile karıştırmak
- `height()` i düğüm başına iki kez çağırmak — O(n)'i **$O(n^2)$**'ye çevirir

Mini soru

İkili bir ağacın yüksekliği 3 ve **mükemmel**.
Kaç düğümü var? Kaçı yaprak?

Cevap

$2^4 - 1 = 15$ düğüm. Son seviye (seviye 3) $2^3 = 8$ yaprak tutar; diğer 7 düğüm iç düğümdür.

3. Dolaşmalar: Her Düğümü Bir Kez Ziyaret Etmek

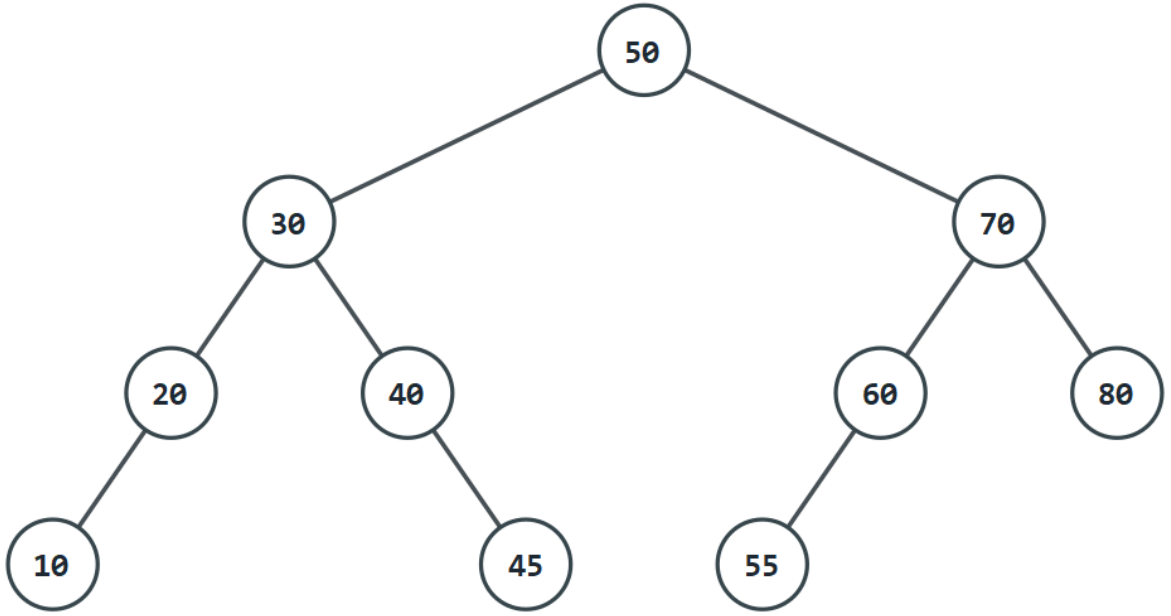
Tek bir "doğal" sıra yok

- Bir düğümün (en çok) iki çocuğu var — gerçek bir seçim
- Üç özyinelemeli sıra: **preorder, inorder, postorder**
- Özyinelemesiz iki sıra daha: kendi **yığınımız**, ve bir **kuyruk**
- Aynı ağaç, beş sıra, genelde beş farklı sonuç

Preorder: ziyaret, sol, sağ

Düğümü çocuklarından **önce** ziyaret edin.
Kök her zaman **ilk** ziyaret edilir — tam
olarak bir ağacı **sıfırdan kurmak** için gereken sıra.

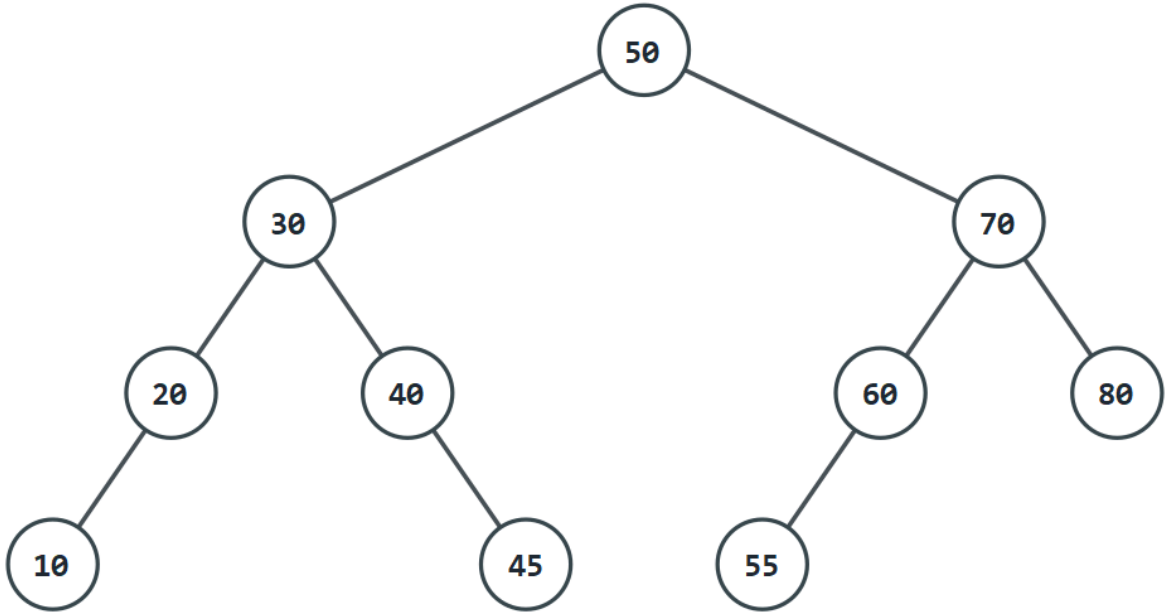
Preorder, adım adım



ziyaret sırası (preorder):

Bitti. Tam preorder sırası: 50, 30, 20, 10, 40, 45, 70, 60, 55, 80. Kök her zaman İLK ziyaret edilendir -- preorder, bir ağacın kopyasını yeniden kurmak için (üst önce) idealdir.

Uç durum — sola yığılmış bir zincir



ziyaret sırası (preorder):

Bitti. Tam preorder sırası: 50, 30, 20, 10, 40, 45, 70, 60, 55, 80. Kök her zaman İLK ziyaret edilendir -- preorder, bir ağacın kopyasını yeniden kurmak için (üst önce) idealdir.

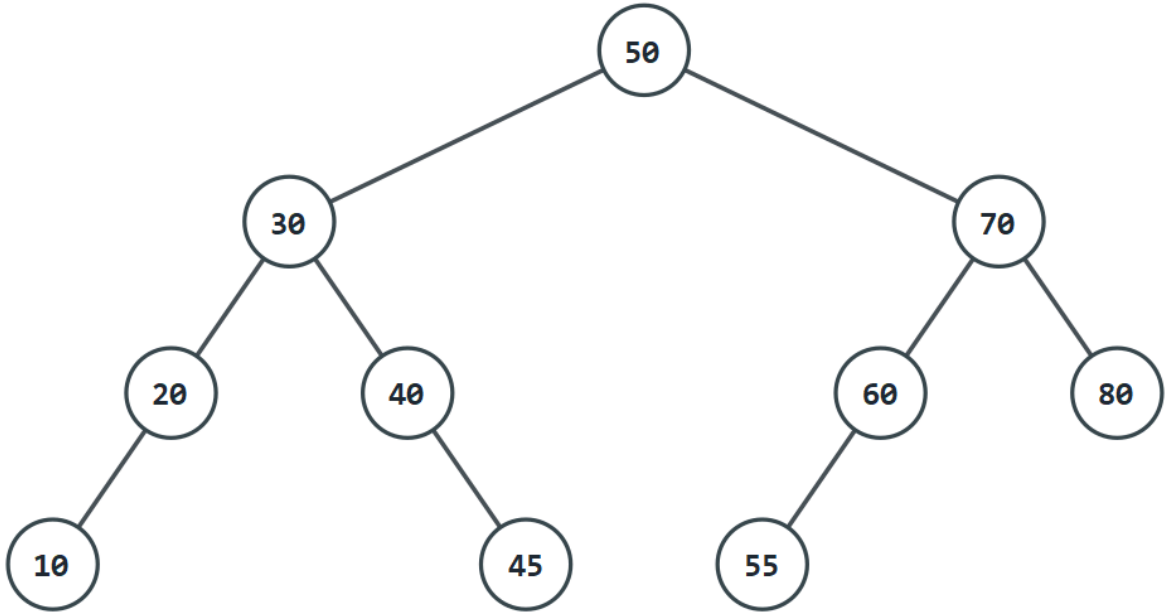
Kod — preorder (özyinelemeli)

```c
static void preorder(Node *node) {
    if (node == NULL) return;
    printf("visit %d\n", node->value);
    visited[visited_count++] = node->value;
    preorder(node->left);
    preorder(node->right);
}
```

Inorder: sol, ziyaret, sağ

Düğümü çocukları **arasında** ziyaret edin.
Bir ikili **arama** ağacında bu, her değeri
sıralı ziyaret eder — burada önizlendi, ileride kurulacak.

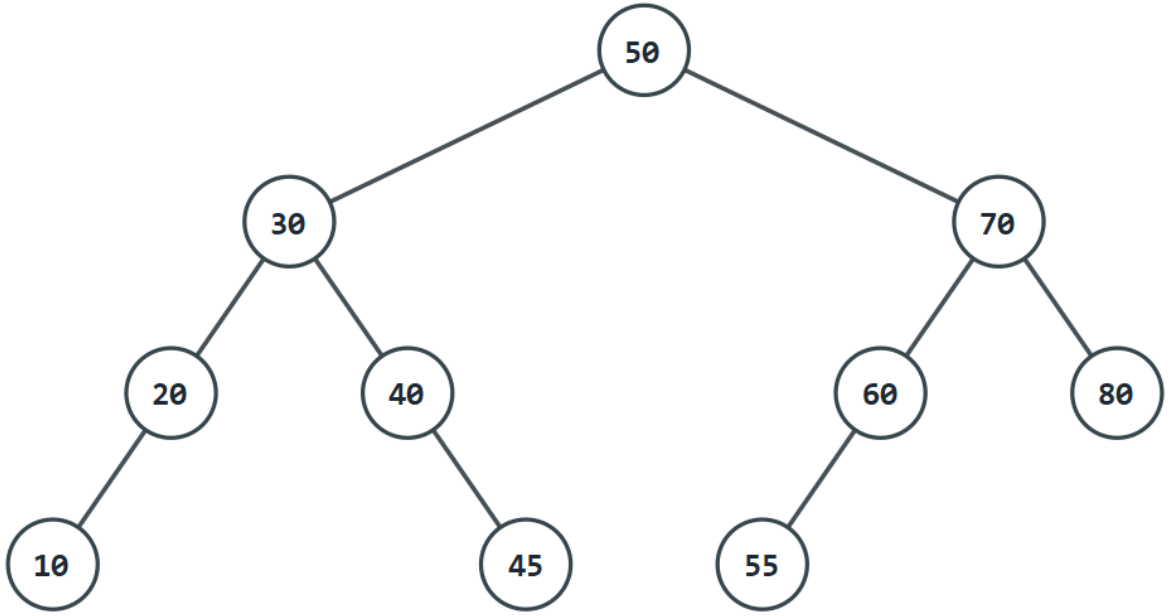
Inorder, adım adım



ziyaret sırası (inorder):

Bitti. Tam inorder sırası: 10, 20, 30, 40, 45, 50, 55, 60, 70, 80. Ağacın her düğümü tam bir kez ziyaret edilir: O(n).

Uç durum — sağa yığılmış bir zincir



ziyaret sırası (inorder):

Bitti. Tam inorder sırası: 10, 20, 30, 40, 45, 50, 55, 60, 70, 80. Ağacın her düğümü tam bir kez ziyaret edilir: O(n).

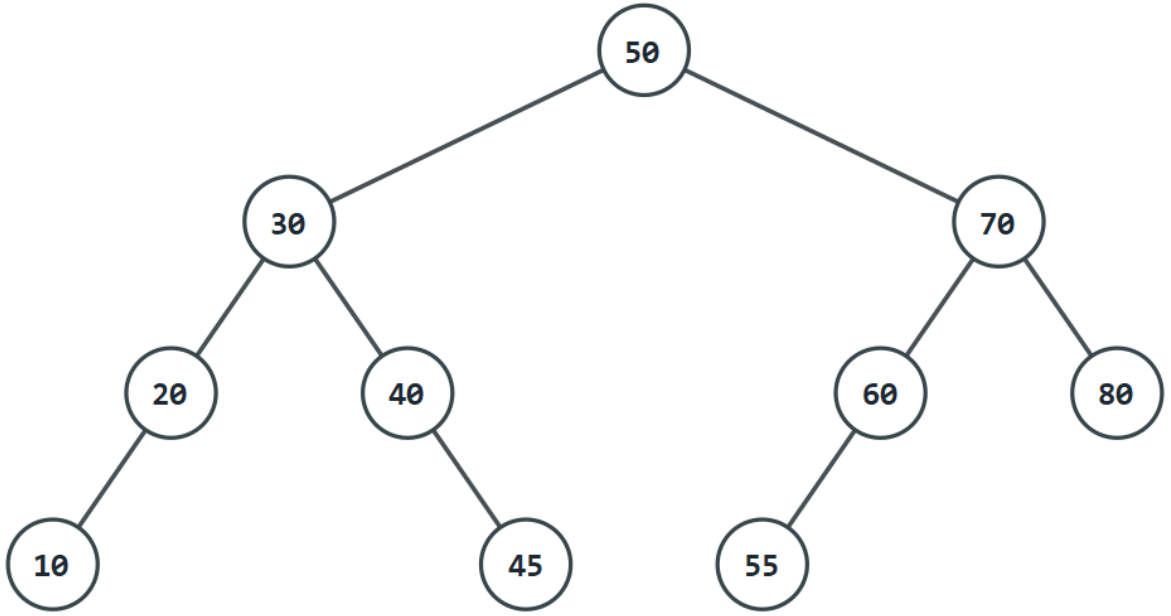
Kod — inorder (özyinelemeli)

```c
static void inorder(Node *node) {
    if (node == NULL) return;
    inorder(node->left);
    printf("visit %d\n", node->value);
    visited[visited_count++] = node->value;
    inorder(node->right);
}
```

Postorder: sol, sağ, ziyaret

Düğümü her iki çocuktan **sonra** ziyaret edin. Kök her zaman **en son** ziyaret edilir — tam olarak bir ağacı güvenle **silmek** için gereken sıra.

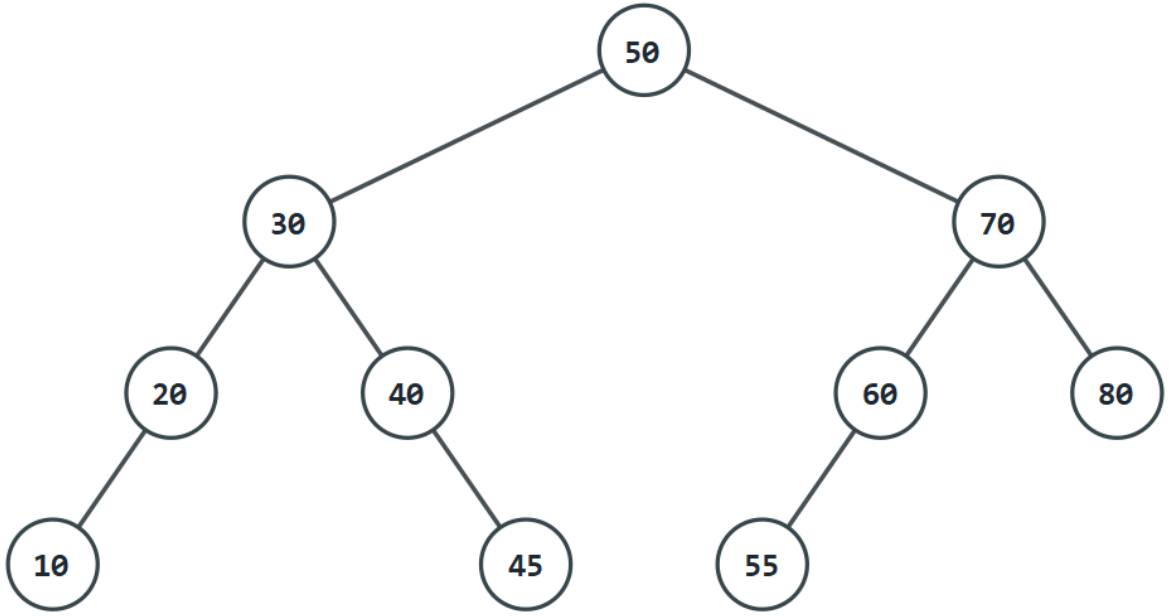
Postorder, adım adım



ziyaret sırası (postorder):

Bitti. Tam postorder sırası: 10, 20, 45, 40, 30, 55, 60, 80, 70, 50. Kök her zaman EN SON ziyaret edilendir -- postorder, bir ağacı güvenle silmek için kullanılır (önce çocuklar, sonra ebeveyn).

Uç durum — sağa yığılmış zincir, tersten



ziyaret sırası (postorder):

Bitti. Tam postorder sırası: 10, 20, 45, 40, 30, 55, 60, 80, 70, 50. Kök her zaman EN SON ziyaret edilendir -- postorder, bir ağacı güvenle silmek için kullanılır (önce çocuklar, sonra ebeveyn).

Kod — postorder (özyinelemeli)

```c
static void postorder(Node *node) {
    if (node == NULL) return;
    postorder(node->left);
    postorder(node->right);
    printf("visit %d\n", node->value);
    visited[visited_count++] = node->value;
}
```

Karmaşıklık — üç dolaşma da

- **O(n)** zaman: her düğüm bir kez, düğümde O(1) iş
- **O(h)** özyineleme yığını alanı — ağacın yüksekliği
- Dengeli ağaçta O(log n), ama bir zincirde **O(n)**
- Bu en kötü durum, bölüm 3.4'ün kendi yığınını doğuruyor

Sık yapılan hatalar

- Taban durumu unutmak — `node == NULL` dönmeli
- Senaryolar arasında paylaşılan ziyaret sayacını sıfırlamamak
- Üç sıranın "genelde aynı" olduğunu sanmak — tek bir ek düğüm bile ayırır
- Yanlış dolaşmayı seçmek: kurmak preorder, silmek postorder ister

Mini soru

Bir ağacı bir dosyaya, değerleri geri okuyup sırayla ekleyerek aynı ağacı yeniden kuracak şekilde kaydetmeniz gerekiyor. Hangi sıra?

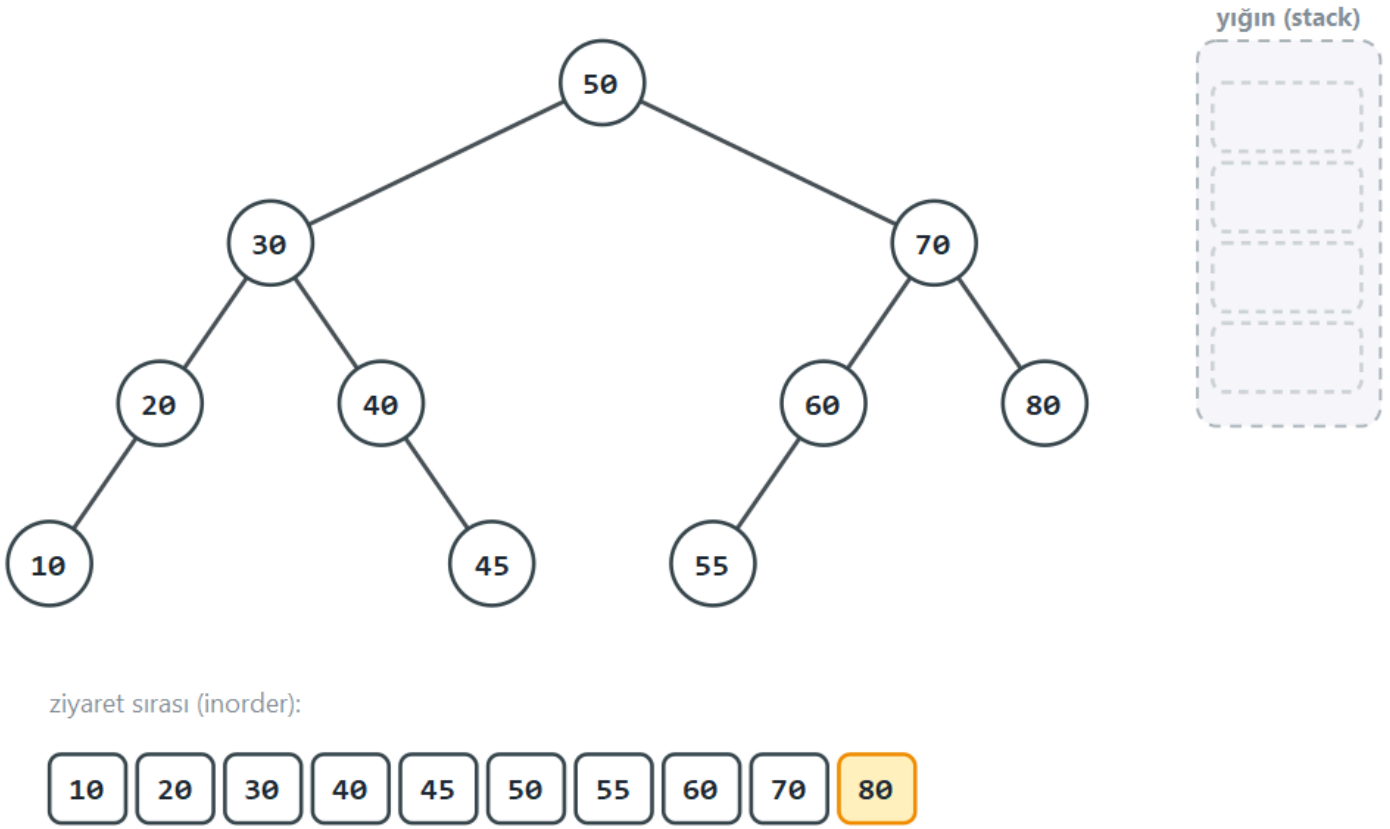
Cevap

Preorder. Geri okunan ilk değer kök olmalı, ve preorder, kökü her iki altağaçtan önce ziyaret eden tek sıradır.

Özyinelemesiz inorder

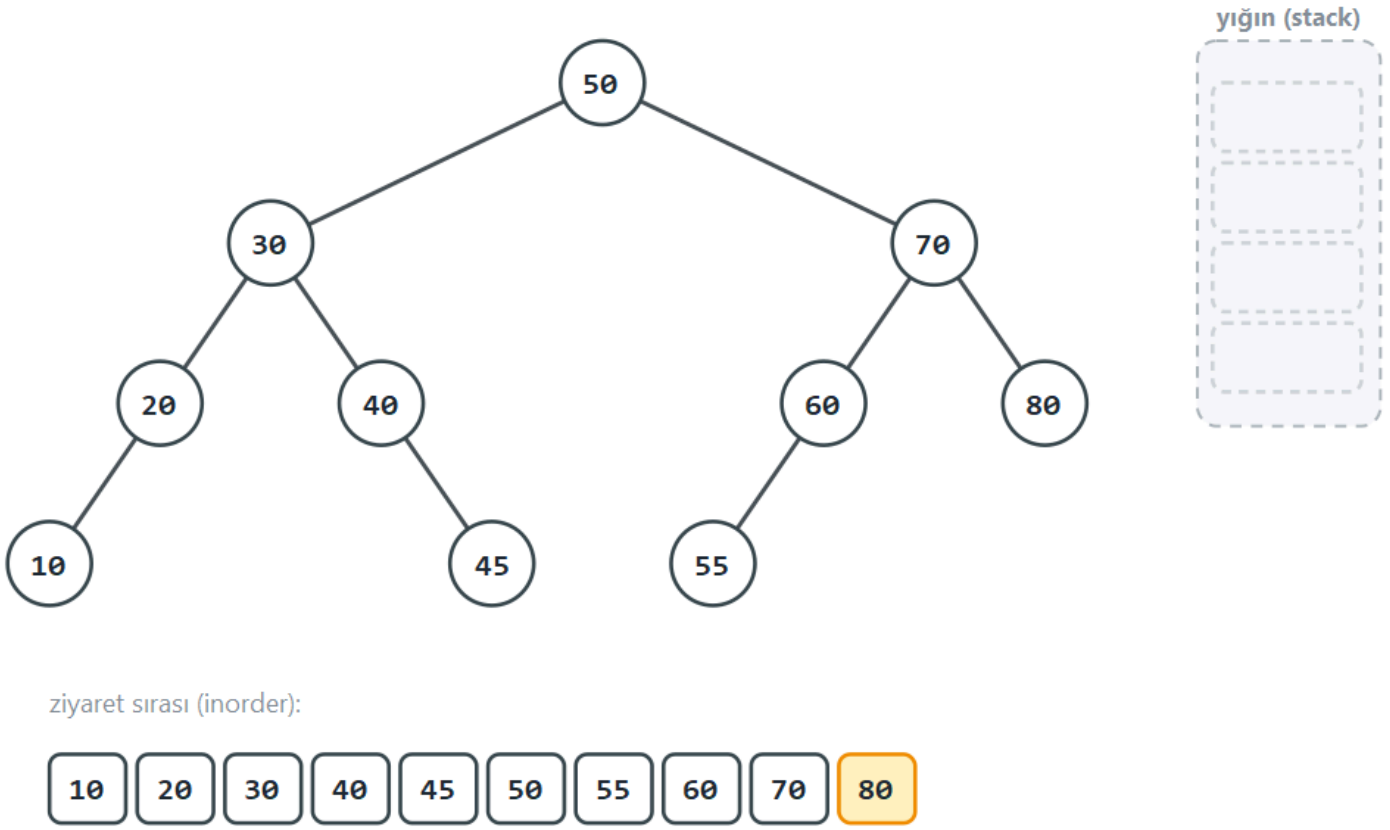
Her özyinelemeli dolaşma, gizlice derleyicinin **çağrı yığınını** kullanır. Bu kez özyineleme yok — Hafta 3'ten kendi **yığınımız**, `int` yerine `Node *` tutan.

Özyinelemesiz inorder, adım adım



Bitti. Tam sıra: 10, 20, 30, 40, 45, 50, 55, 60, 70, 80 -- özyinelemeli inorder ile birebir aynı. Fark: yığını artık biz elle yönetiyoruz, derleyici değil.

Uç durum — yığın en derin noktasına ulaşıyor



Bitti. Tam sıra: 10, 20, 30, 40, 45, 50, 55, 60, 70, 80 -- özyinelemeli inorder ile birebir aynı. Fark: yığını artık biz elle yönetiyoruz, derleyici değil.

Kod — push / pop (kendi yığınımız)

```
static void push(Node *n) {
    top = top + 1;
    stack_data[top] = n;
}
static Node *pop(void) {
    Node *n = stack_data[top];
    top = top - 1;
    return n;
}
```

Kod — ana döngü

```
Node *cur = root;
while (cur != NULL || !is_empty()) {
    while (cur != NULL) {
        push(cur);
        cur = cur->left;
    }
    cur = pop();
    printf("visit %d\n", cur->value);
    cur = cur->right;
}
```

Karmaşıklık

- **O(n)** zaman — her düğüm bir kez push, bir kez pop
- **O(h)** alan — özyinelemeli sürümle tam olarak aynı
- Bellek kazancı yok; muhasebe yalnızca kendi yığınımıza taşındı
- Kazanç: kendi yığınımız, sabit bir özyineleme sınırını aşabilir

Sık yapılan hatalar

- `cur != NULL || !is_empty()` in herhangi bir yarısını atlamak
- Bir düğümü pop edip ziyaret ettikten sonra `cur = cur->right` i unutmak
- Atlarsanız, her sağ altağaç sessizce kaybolur

Mini soru

Mükemmel yükseklik *h* bir ağaçta, bu dolaşma boyunca yığında aynı anda en çok kaç düğüm oturur?

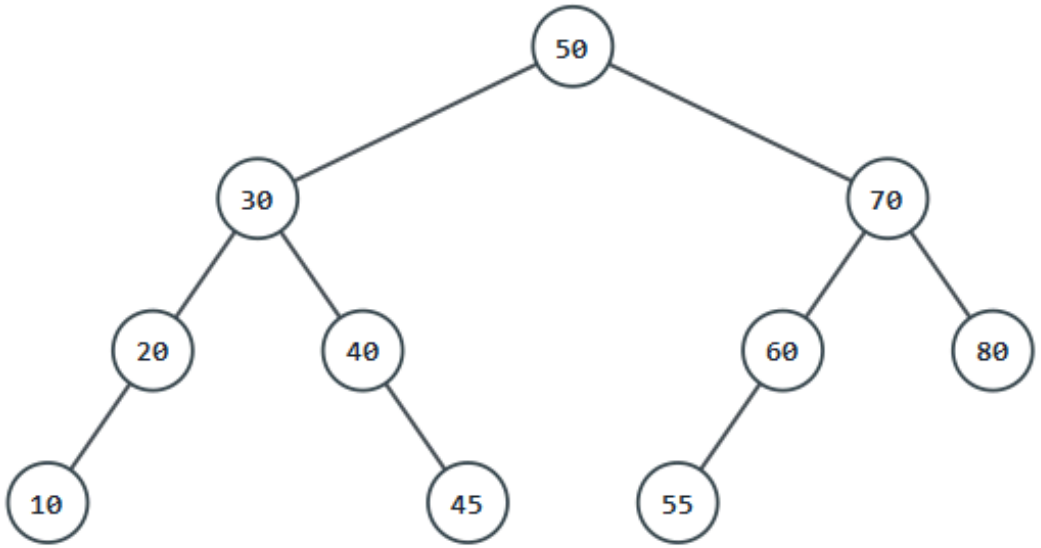
Cevap

***h* + 1.** Yığın her zaman yalnızca o anki sol omurgayı tutar, ve mükemmel bir ağacın en uzun sol omurgası *h* + 1 düğümlüdür.

Level order (genişlik öncelikli), bir kuyrukla

Şimdiye kadarki her dolaşma **derine** gider, **genişe** gitmeden önce. Level order kökü, sonra derinlik 1'deki **her** düğümü ziyaret eder — bir **kuyruk** gerektirir, yığın değil.

Level order, adım adım



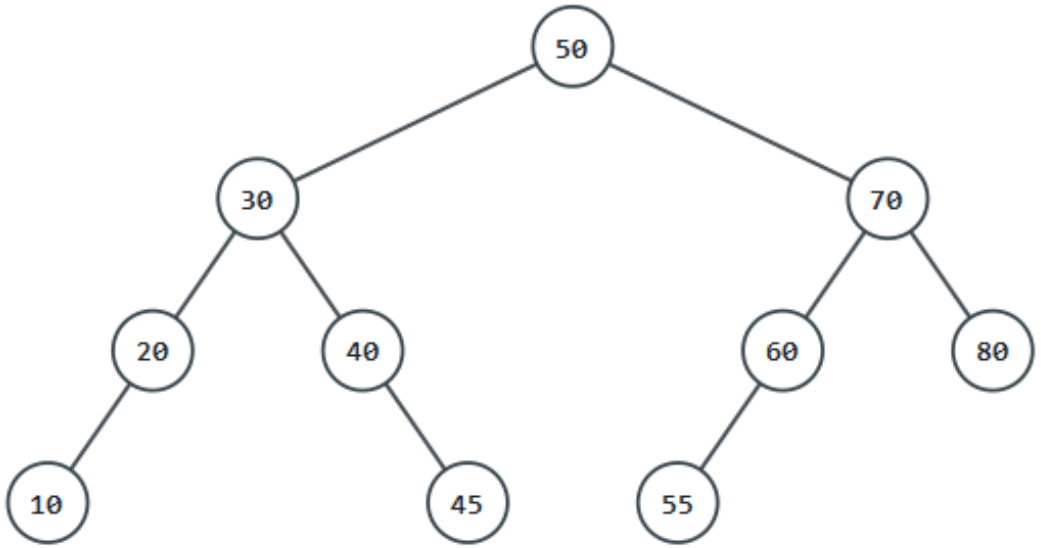
kuyruk (queue)

ziyaret sırası (level-order):

50 30 70 20 40 60 80 10 45 55

Kuyruk boş, bitti. Tam sıra: 50, 30, 70, 20, 40, 60, 80, 10, 45, 55 -- her seviyeyi soldan sağa tamamladıktan sonra bir alt seviyeye geçtik.

Uç durum — kuyruk hep bir eleman tutuyor



kuyruk (queue)

ziyaret sırası (level-order):

50	30	70	20	40	60	80	10	45	55

Kuyruk boş, bitti. Tam sıra: 50, 30, 70, 20, 40, 60, 80, 10, 45, 55 -- her seviyeyi soldan sağa tamamladıktan sonra bir alt seviyeye geçtik.

Kod — enqueue / dequeue

```
static void enqueue(Node *n) {
    rear = (rear + 1) % QUEUE_CAP;
    queue_data[rear] = n;
    count++;
}
static Node *dequeue(void) {
    Node *n = queue_data[front];
    front = (front + 1) % QUEUE_CAP;
    count--;
    return n;
}
```

Kod — ana döngü

```
enqueue(root);
while (!is_empty()) {
    Node *cur = dequeue();
    printf("visit %d\n", cur->value);
    if (cur->left != NULL) enqueue(cur->left);
    if (cur->right != NULL) enqueue(cur->right);
}
```

Karmaşıklık

- **O(n)** zaman — her düğüm bir kez enqueue, bir kez dequeue
- **O(w)** alan, *w* ağacın en büyük **genişliği**
- Dallı, dengeli bir ağaçta *w*, O(n) kadar büyük olabilir
- Her derinlik öncelikli dolaşmanın O(h) alanıyla karşılaştırın

Sık yapılan hatalar

- `dequeue`i "zaten aynı kod" diye `pop` ile değiştirmek — sessizce derinlik öncelikliye döner
- Yanlışlıkla `NULL` çocukları kuyruğa eklemek — bir sonraki dequeue'da çöker

Mini soru

İki **farklı** ikili ağaç, tam olarak aynı
level-order değer dizisini paylaşabilir mi?
Doğru mu yanlış mı?

Cevap

Doğru. Bir level-order dizisi, bir seviyede herhangi bir boşluk olduğunda, ebeveyn-çocuk ilişkilerini tek başına kodlamıyor.

4. Tam Bir Ağaç Bir Dizide

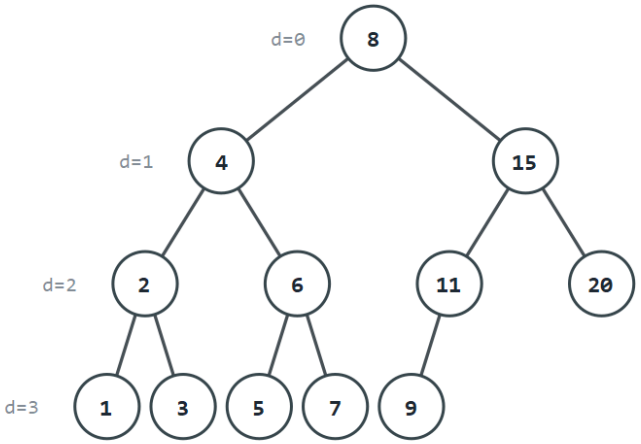
Başlangıç sorusu

İşaretçiler bellek maliyeti taşır, ve bir ebeveyni bulmak ya saklı bir işaretçi ya da bir arama ister. **Tam** bir ağaç için daha ucuz bir yol var mı?

İndis formülleri

- İndis *i*'deki düğümün **sol çocuğu**: `2*i + 1`
- **Sağ çocuğu**: `2*i + 2`
- **Ebeveyni**: `(i - 1) / 2` (tam sayı bölmesi)
- Hiçbir yerde `left`, `right`, `parent` alanı yok — yalnızca bir dizi

Dizi gösterimi, adım adım



dizi (array), soldan sağa indis 0, 1, 2...

A =

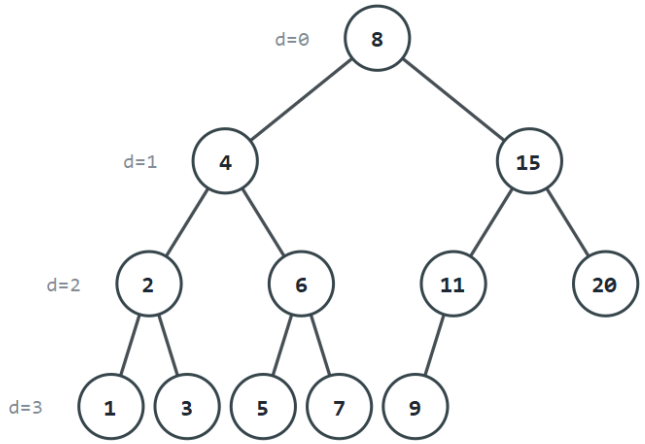
0	1	2	3	4	5	6	7	8	9	10	11
8	4	15	2	6	11	20	1	3	5	7	9

gözlemler:

```
parent(0) = -- (kök)
left(0) = 1
right(0) = 2
parent(3) = 1
left(3) = 7
right(3) = 8
kontrol: 0..11 arası boşluk yok
```

Sonuç: 12 düğüm, tam (complete) mi? EVET. Tam bir ağaç için dizi temsili mükemmeldir (boşa giden hücre yok); tam olmayan bir ağaç içinse dizi çoğu zaman gerçek düğüm sayısının çok üstünde hücreye ihtiyaç duyar.

Uç durum — TAM DEĞİL: bir boşluk



dizi (array), soldan sağa indis 0, 1, 2...

A =

0	1	2	3	4	5	6	7	8	9	10	11
8	4	15	2	6	11	20	1	3	5	7	9

gözlemler:

```
parent(0) = -- (kök)
left(0) = 1
right(0) = 2
parent(3) = 1
left(3) = 7
right(3) = 8
kontrol: 0..11 arası boşluk yok
```

Sonuç: 12 düğüm, tam (complete) mi? EVET. Tam bir ağaç için dizi temsili mükemmeldir (boşa giden hücre yok); tam olmayan bir ağaç içinse dizi çoğu zaman gerçek düğüm sayısının çok üstünde hücreye ihtiyaç duyar.

Kod — parent / left / right, is_complete

```
static int parent(int i) { return (i - 1) / 2; }
static int left(int i)   { return 2 * i + 1; }
static int right(int i)  { return 2 * i + 2; }

static bool is_complete(int arr[], int n, int last_real) {
    for (int i = 0; i <= last_real; i++)
        if (arr[i] == EMPTY) return false;
    return true;
}
```

Karmaşıklık

- `parent`, `left`, `right`: saf aritmetik — **O(1)**
- 10 düğüm ya da 10 milyon için maliyet aynı
- `is_complete` in kendisi: **O(n)**, dizinin bir taraması

Sık yapılan hatalar

- Bu formülleri, gerçekte tam **olmayan** bir ağaçta kullanmak
- `(i - 1) / 2` ye, dilinizin tam sayı bölme kuralını kontrol etmeden güvenmek
- Kökün (`i = 0`) ebeveyni olmadığını, özel bir durum olduğunu unutmak

Mini soru

Tam bir ikili ağaç, bir dizide saklı. İndis 11'deki düğümün iki çocuğu var. Hangi indislerde? Ebeveyni hangi indiste?

Cevap

Çocuklar `2*11+1 = 23` ve `2*11+2 = 24` de.
Ebeveyn `(11-1)/2 = 5` te.

5. İkili Öbek

Başlangıç sorusu

Bir işletim sistemi, yüz bekleyen süreçten en acilinin hangisi olduğunu, anında, tekrar tekrar bilmeli. Her gelişte tüm listeyi sıralamak boşa iş. En iyi değeri her zaman elin altında tutan en ucuz yapı nedir?

Kısa bir tarihçe

- **1964** — J. W. J. Williams, öbeği ve **öbek sıralamasını** birlikte tanıtıyor
- **1964** — R. W. Floyd, O(n)'lik bir **kurma** yöntemi yayınlıyor
- Öbek, tam olarak sıralamayı hızlandırmak için icat edildi

Öbek özelliği

- **Min-öbek:** her ebeveyn ≤ her iki çocuk — en küçük kökte
- **Max-öbek:** her ebeveyn ≥ her iki çocuk — en büyük kökte
- Sıralı bir yapı **değil** — kardeşler arasında zorunlu sıra yok
- Yalnızca her ebeveyn kendi iki çocuğunu yeniyor, fazlası değil

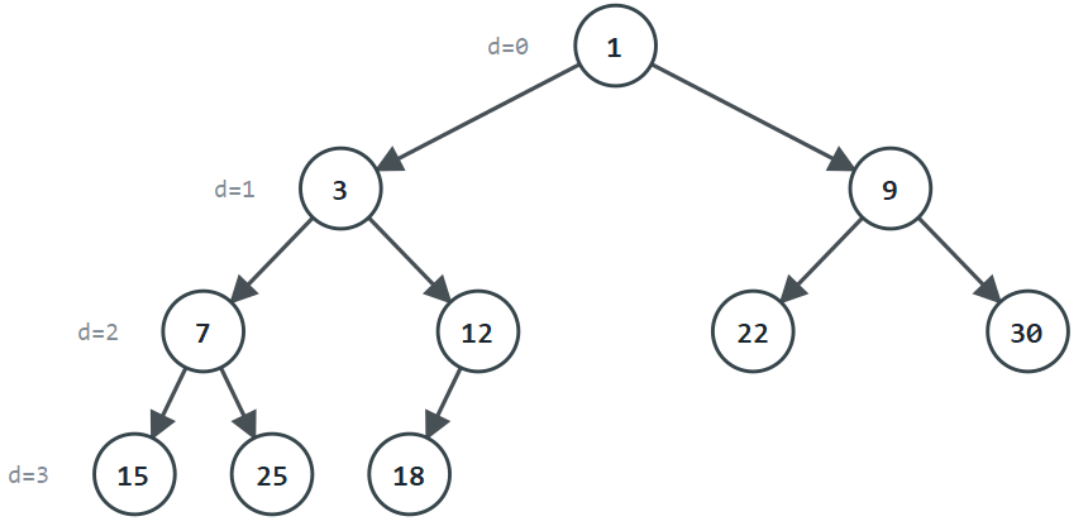
Öbek ADT'si

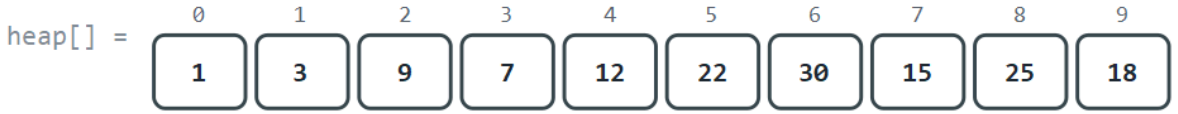
İşlem	Ne yapar	Karmaşıklık
`peek()`	Kökü silmeden döndürür	O(1)
`insert(x)`	`x` i ekler, **sift-up** ile onarır	O(log n)
`extract()`	Kökü siler, **sift-down** ile onarır	O(log n)
`build_heap(arr)`	Bir diziyi tek seferde öbeğe çevirir	O(n)

Ekleme: sift-up

Yeni değeri bir sonraki boş slota koyun —
ağacı tam tutar. Sonra öbek özelliği bozukken
tekrar tekrar **ebeveyniyle** yer değiştirin.
Buna **sift-up** denir.

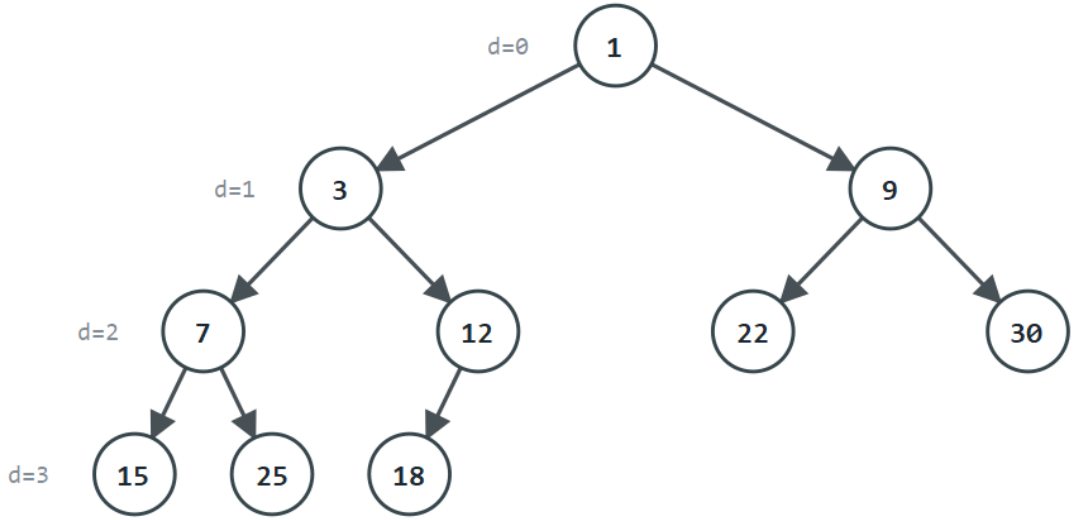
Sift-up ile ekleme, adım adım

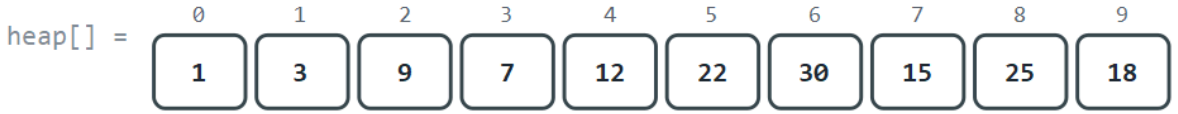




Bitti: son öbek dizisi [1, 3, 9, 7, 12, 22, 30, 15, 25, 18]. Kökte (indis 0) her zaman en iyi değer var: 1. Her insert, ağacın yüksekliği kadar (O(log n)) karşılaştırma yapar.

Uç durum — zaten sıralı, sifting gerekmiyor





Bitti: son öbek dizisi [1, 3, 9, 7, 12, 22, 30, 15, 25, 18]. Kökte (indis 0) her zaman en iyi değer var: 1. Her insert, ağacın yüksekliği kadar (O(log n)) karşılaştırma yapar.

Kod — insert() / sift-up

```
void insert(int value) {
    heap[size] = value;
    int i = size;
    size++;
    while (i > 0) {
        int parent = (i - 1) / 2;
        if (!better(heap[i], heap[parent]))
            break;
        int tmp = heap[parent];
        heap[parent] = heap[i];
        heap[i] = tmp;
        i = parent;
    }
}
```

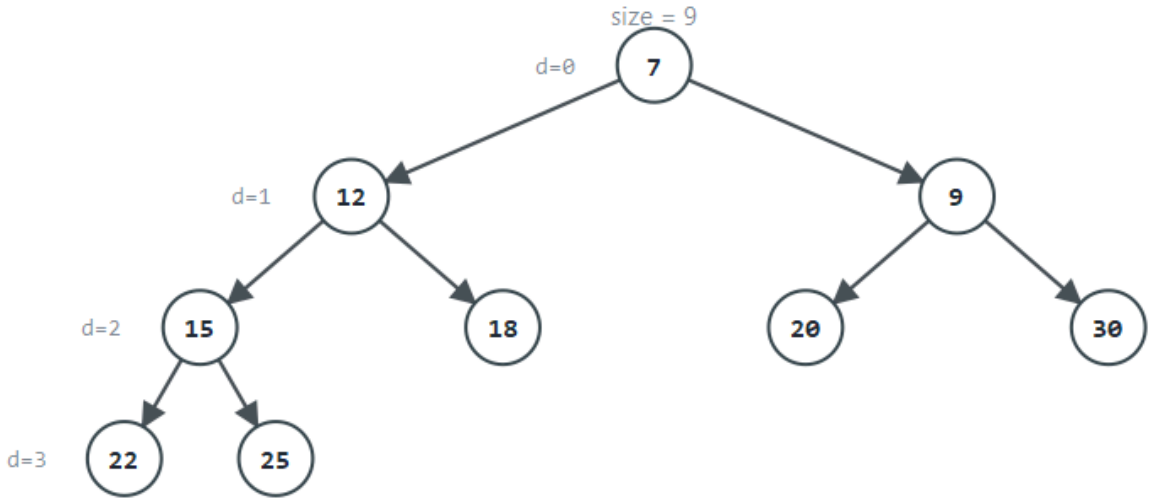
Karmaşıklık

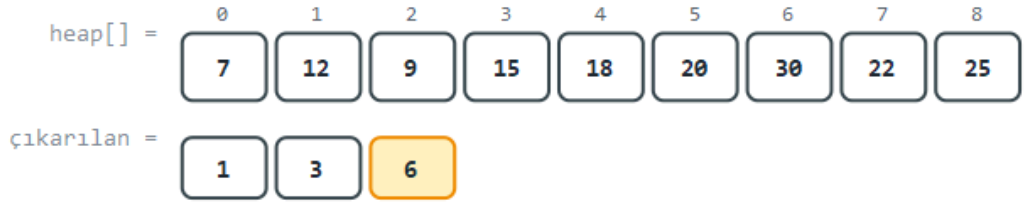
- `insert`: her **seviye**de en çok bir yer değiştirme — **O(log n)**
- log n, n düğümlü tam bir ağacın yüksekliği
- `peek` (`heap[0]` i okumak): **O(1)**

Çıkarma: sift-down

Kökü kaydedin, dizinin **son** elemanını yerine taşıyın — ağacı tam tutar. Sonra tekrar tekrar **daha iyi çocuğuyla** yer değiştirin. Buna **sift-down** denir.

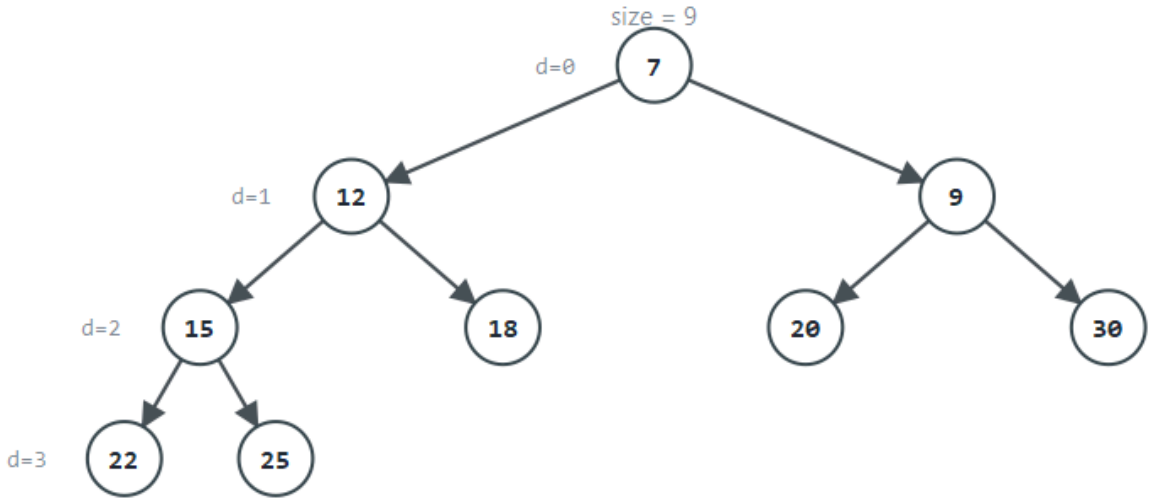
Sift-down ile çıkarma, adım adım

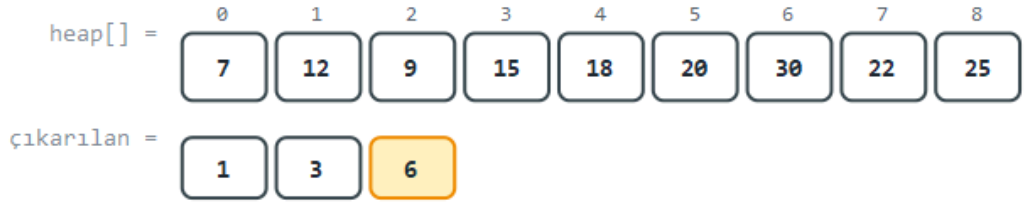




Bitti: çıkarılan sıra [1, 3, 6]; kalan 9 değer: [7, 9, 12, 15, 18, 20, 22, 25, 30]. Her extract, O(log n) sürer.

Uç durum — sonuna kadar: 10 değerin tamamı





Bitti: çıkarılan sıra [1, 3, 6]; kalan 9 değer: [7, 9, 12, 15, 18, 20, 22, 25, 30]. Her extract, O(log n) sürer.

Kod — extract() / sift-down

```
int extract(void) {
    int best = heap[0];
    size--;
    heap[0] = heap[size];   /* ... */
    int i = 0;
    while (1) {              /* sift-down */
        /* ... target = sol/sağ çocuktan iyi olan ... */
        if (target == i)
            break;
        /* ... heap[i]/heap[target] değiştir ... */
    }
    return best;
}
```

Karmaşıklık

- `extract` : her **seviye**de en çok bir yer değiştirme — **O(log n)**
- sift-up'ın maliyetinin tam ayna görüntüsü
- Aynı O(log n) sınırı, ters yönde

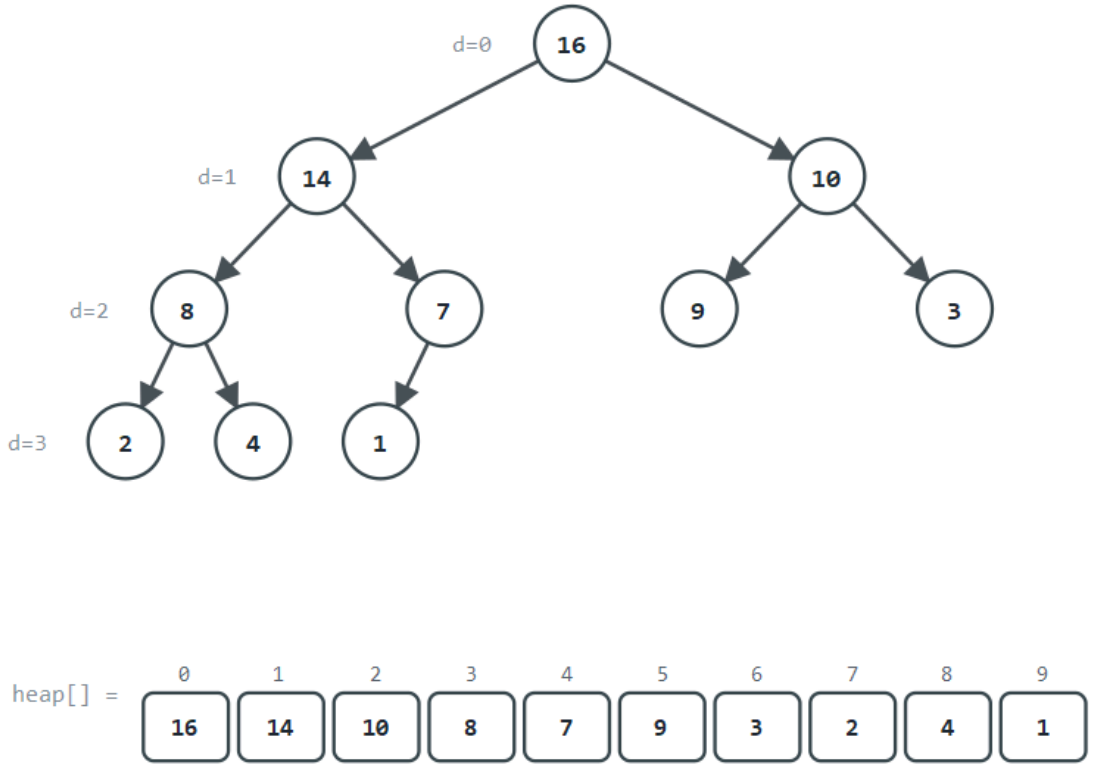
Sık yapılan hatalar (ekleme ve çıkarma)

- `extract`i önce `size > 0` kontrol etmeden çağırmak
- sift-down'da yalnızca **bir** çocukla karşılaştırmak, ikisiyle değil
- `better`de `<` yerine `<=` kullanmak — zararsız, ama eşitlik davranışını değiştirir

Bir öbeği O(n)'de kurmak

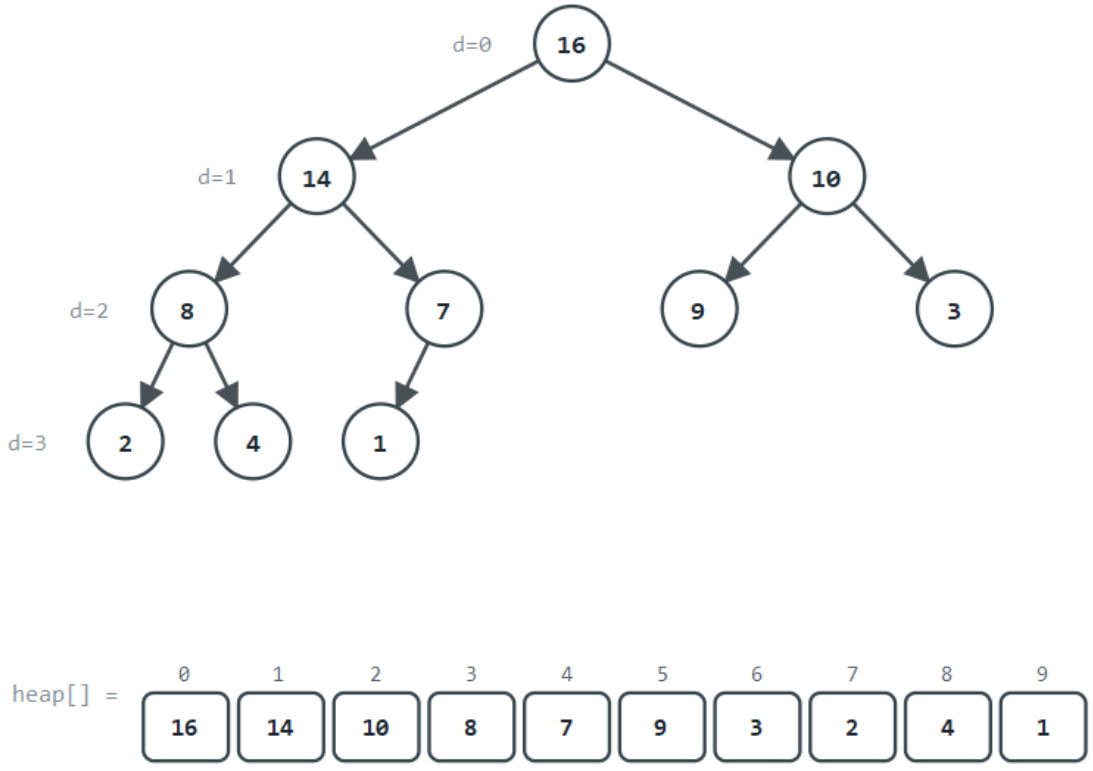
n tek tek ekleme toplamda O(n log n) maliyetli. **Floyd'un algoritması** daha iyisini yapar: her **yaprak** zaten geçerli bir öbek; yalnızca **iç** düğümleri, sondan köke, sift edin.

Öbek kurma, adım adım



Bitti: [16, 14, 10, 8, 7, 9, 3, 2, 4, 1] artık geçerli bir max-öbek. Her çağrının maliyeti farklı olsa da toplam iş O(n)'dir (O(n log n) DEĞİL) -- alt seviyelerdeki çok sayıda düğüm çok az sifting yapar.

Uç durum — girdi zaten geçerli bir öbek



Bitti: [16, 14, 10, 8, 7, 9, 3, 2, 4, 1] artık geçerli bir max-öbek. Her çağrının maliyeti farklı olsa da toplam iş O(n)'dir (O(n log n) DEĞİL) -- alt seviyelerdeki çok sayıda düğüm çok az sifting yapar.

Kod — build_heap()

```
void build_heap(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--)
        sift_down(arr, n, i);
}
```

Neden O(n), O(n log n) değil?

- Kabaca n/2 düğüm **yaprak** — tamamen atlanıyor
- Kabaca n/4 düğüm en çok 1, n/8'i en çok 2 yer değiştirme
- "Yükseklik *h*'deki sayı çarpı *h*" toplamı O(n)'e yakınsıyor
- Alttaki çok ucuz sift'ler, üstteki az sayıda maliyetliyi eziyor

Sık yapılan hatalar

- Döngüye `i = 0` dan başlamak, `i = n/2 - 1` den değil — düzeltilmemiş altağaçları sift eder
- `build_heap` i "yalnızca n tane insert" sanmak — geçerli, ama aynı dizilim ya da maliyet değil

Mini soru

`build_heap` ve `heap_sort`ın ana döngüsü ikisi de tekrar tekrar `sift_down` çağırır.

Neden birincisi toplamda **O(n)**, ikincisi **O(n log n)**?

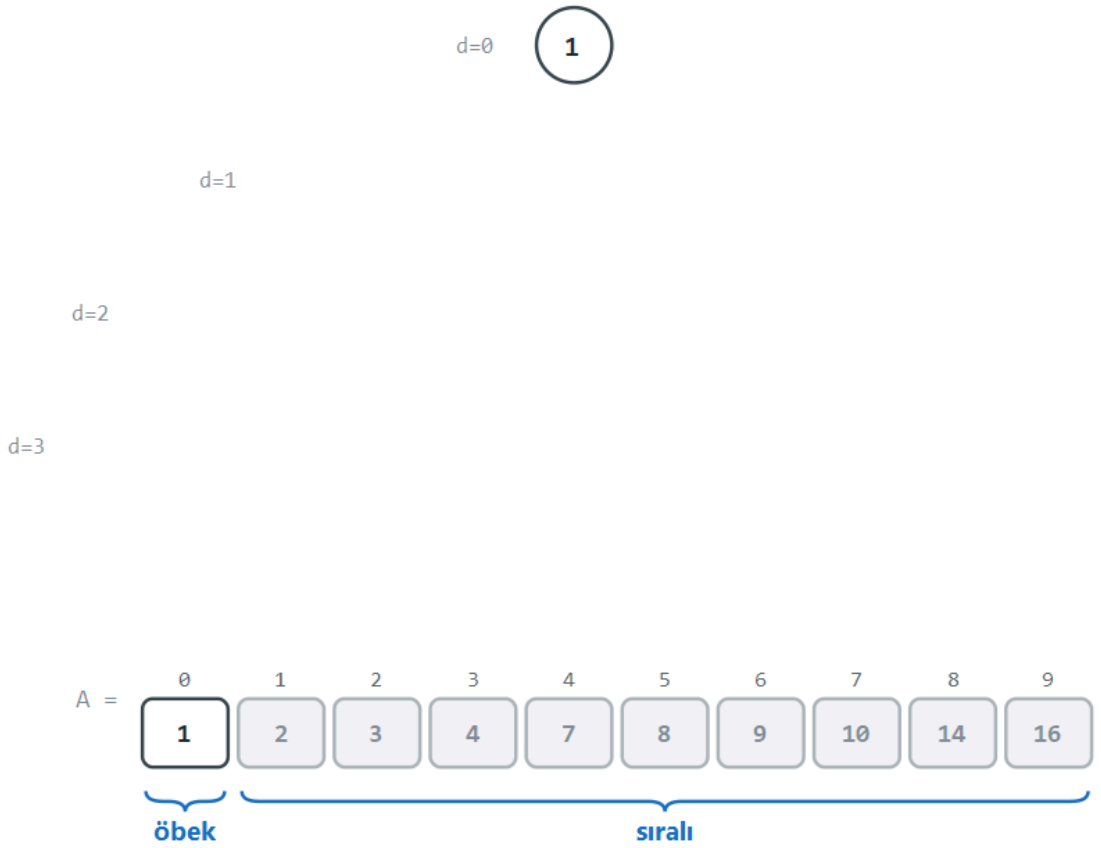
Cevap

`build_heap`, alttaki çoğunlukla ucuz düğümleri sift eder. `heap_sort`, her **çıkarma**da bir kez, her zaman **kök**ten başlar — ortalanacak ucuz bir çoğunluk yok.

Öbek sıralaması — fikir

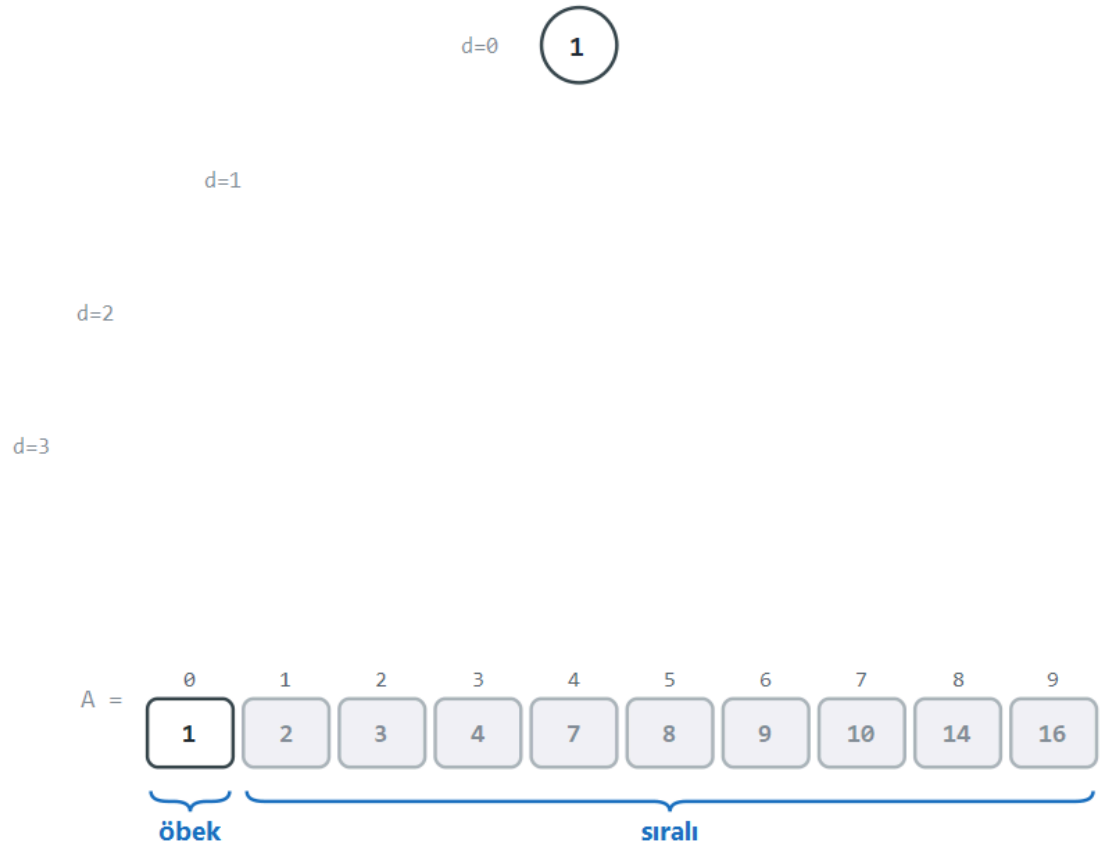
`build_heap`, sonra tekrar tekrar kökü sıralı kuyruğa taşıyın, küçültün, sift-down yapın. Max-öbek **artan** sıralar; min-öbek **azalan** sıralar.

Öbek sıralaması, adım adım



Bitti: [1, 2, 3, 4, 7, 8, 9, 10, 14, 16] -- tamamen sıralı. Öbek sıralaması yerinde (in-place) çalışır ve O(n log n) sürer, ama kararlı (stable) DEĞİLDİR.

Uç durum — zaten artan, maliyet yine aynı



Bitti: [1, 2, 3, 4, 7, 8, 9, 10, 14, 16] -- tamamen sıralı. Öbek sıralaması yerinde (in-place) çalışır ve O(n log n) sürer, ama kararlı (stable) DEĞİLDİR.

Kod — heap_sort()

```
void heap_sort(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--)
        sift_down(arr, n, i);
    for (int heap_size = n; heap_size > 1; heap_size--) {
        int tmp = arr[0];
        arr[0] = arr[heap_size - 1];
        arr[heap_size - 1] = tmp;
        sift_down(arr, heap_size - 1, 0);
    }
}
```

Karmaşıklık

- `build_heap`: O(n); döngü sonra n − 1 kez çalışır
- Her tur: bir O(1) yer değiştirme + bir **O(log n)** sift-down
- Toplam: **O(n log n)** — yerinde sıralar, ekstra dizi yok

Sık yapılan hatalar

- Öbek sıralamasının **kararlı** olduğunu sanmak — değil
- Küçülen bölge yerine tüm dizi üzerinde sifting yapmak — sıralanmış kuyruğu bozar

6. Öncelik Kuyruğu: Bir Öbeğin Hizmet Ettiği ADT

ADT

Bir **öncelik kuyruğu**, her biri bir **önceliğe** sahip öğeleri yönetir. `insert` / `peek` / `extract`, az önce kurduğunuz öbek işlemlerine doğrudan karşılık gelir.

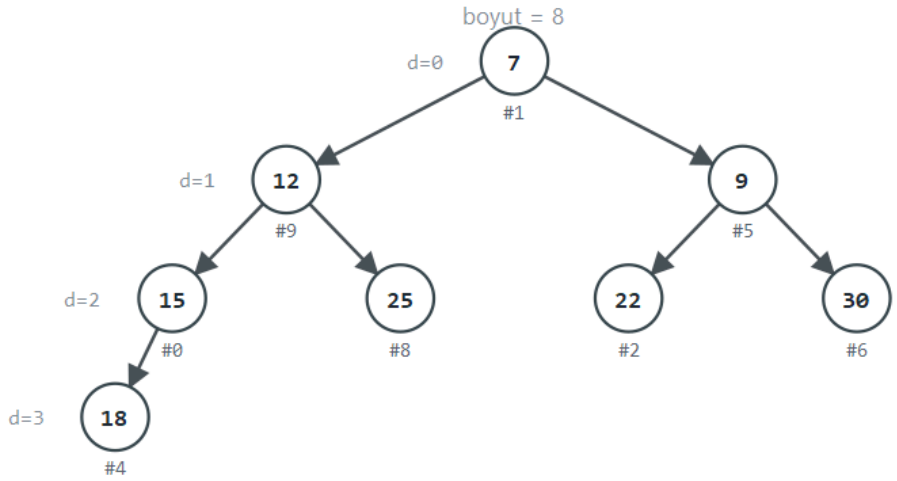
Öncelik kuyruğu ADT'si

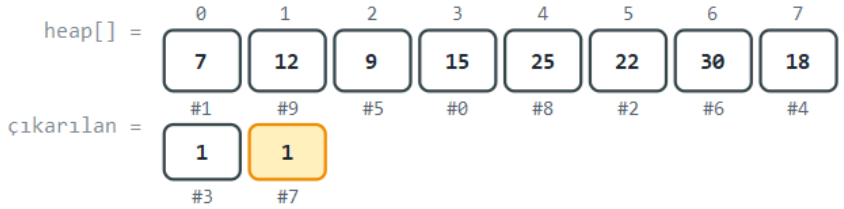
İşlem	Ne yapar	Karmaşıklık
`insert(x)`	`x` i önceliğiyle ekler	O(log n)
`peek()`	En üstteki öğeyi döndürür, tutar	O(1)
`extract()`	En üstteki öğeyi siler, döndürür	O(log n)
`update_key(id, p)`	Bir öğenin önceliğini değiştirir	O(n) naif

update_key — yeni olan tek parça

- Öncelik değişir → öğe taşınması gerekebilir
- Şimdi daha iyi: **decrease-key** — **yukarı** sift
- Şimdi daha kötü: **increase-key** — **aşağı** sift
- Gerçek kullanım: Dijkstra'nın algoritması, İS zamanlayıcıları

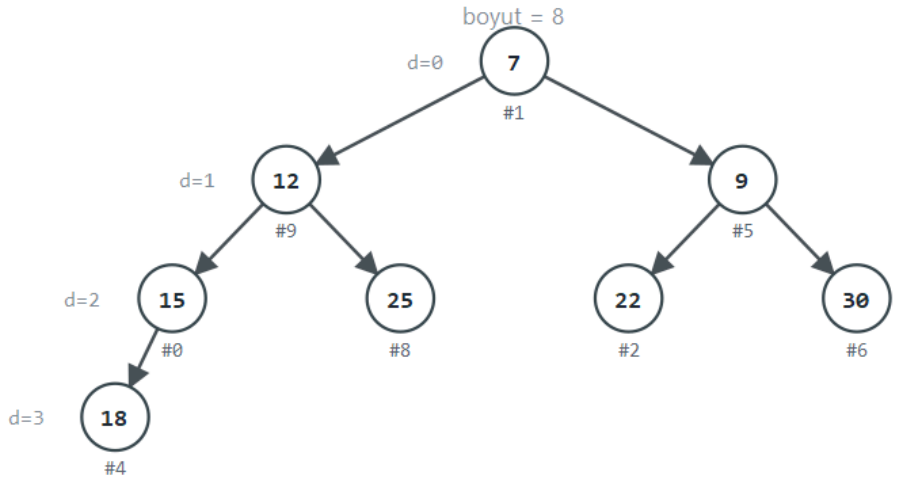
Öncelik kuyruğu işlemleri, adım adım

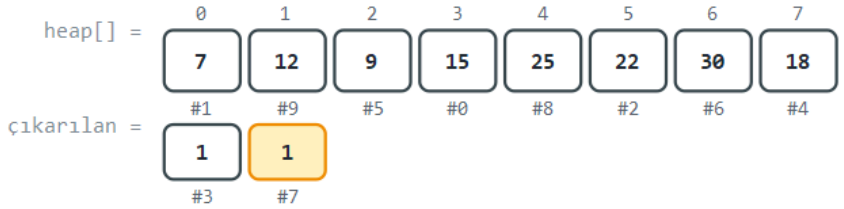




Bitti: 10 insert, 2 extract, 1 update-key. Kalan 8 eleman: [7, 9, 12, 15, 18, 22, 25, 30]. Her işlem O(log n) (peek O(1)); update_key'deki doğrusal id taraması O(n) -- gerçek uygulamalarda bir id→indis tablosuyla O(log n)'e indirilir.

Uç durum — boşken extract/peek





Bitti: 10 insert, 2 extract, 1 update-key. Kalan 8 eleman: [7, 9, 12, 15, 18, 22, 25, 30]. Her işlem O(log n) (peek O(1)); update_key'deki doğrusal id taraması O(n) -- gerçek uygulamalarda bir id→indis tablosuyla O(log n)'e indirilir.

Kod — Item, insert()

```
typedef struct {
    int id;
    int key;
} Item;

void insert(int id, int key) {
    heap[size] = (Item){id, key};
    sift_up(size);
    size++;
}
```

Kod — update_key()

```
void update_key(int id, int new_key) {
    int i = find_by_id(id);
    if (i == -1) { /* … print a message */ return; }
    heap[i].key = new_key;
    if (i > 0 && better(heap[i], heap[(i - 1) / 2]))
        sift_up(i);
    else
        sift_down(i);
}
```

Karmaşıklık

- `insert`, `extract`: **O(log n)**, öbekten devralınan
- `peek`: **O(1)**
- `update_key`: id için doğrusal taramayla **O(n)**
- id→indis bir hash tablosu bunu O(log n)'e indirir

Sık yapılan hatalar

- Bir öğenin **id**'sini (kalıcı) **dizi indisi**yle (değil) karıştırmak
- Kuyruğun boş olmadığını kontrol etmeden `peek` / `extract` çağırmak
- `update_key` den sonra yanlış yönde sift yapmak

Mini soru

Min-öncelik bir zamanlayıcı "teslime kalan süre"yle anahtarlanıyor. Çalışan bir süreç, çok daha acil bir istekle şimdi kesildi. Decrease-key mi increase-key mi? Hangi sift?

Cevap

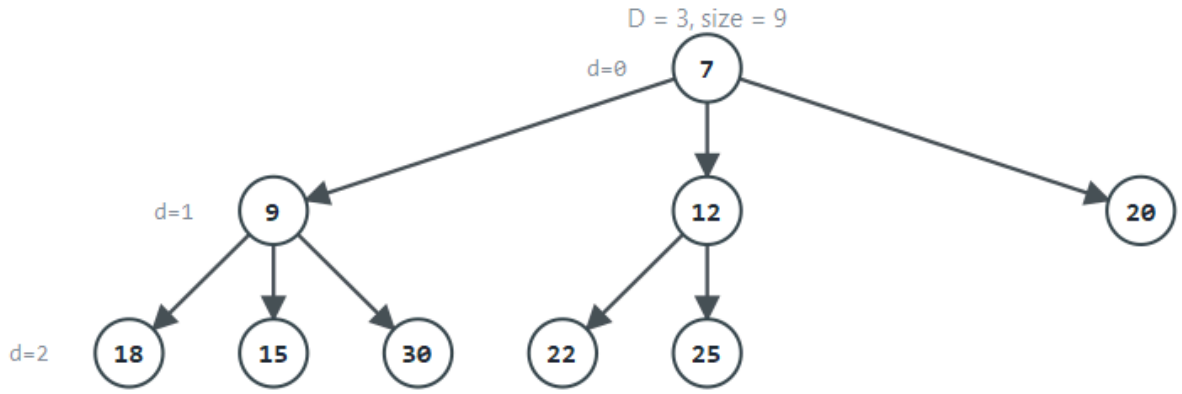
Decrease-key — daha acil, daha **küçük** bir key demek. Bu, köke doğru yüzen **sift-up**ı tetikler; kök en küçüğü tutar.

7. Öbek Varyasyonları: Bir Özelliği Başka Bir Özellikle Takas Etmek

d-ary öbek

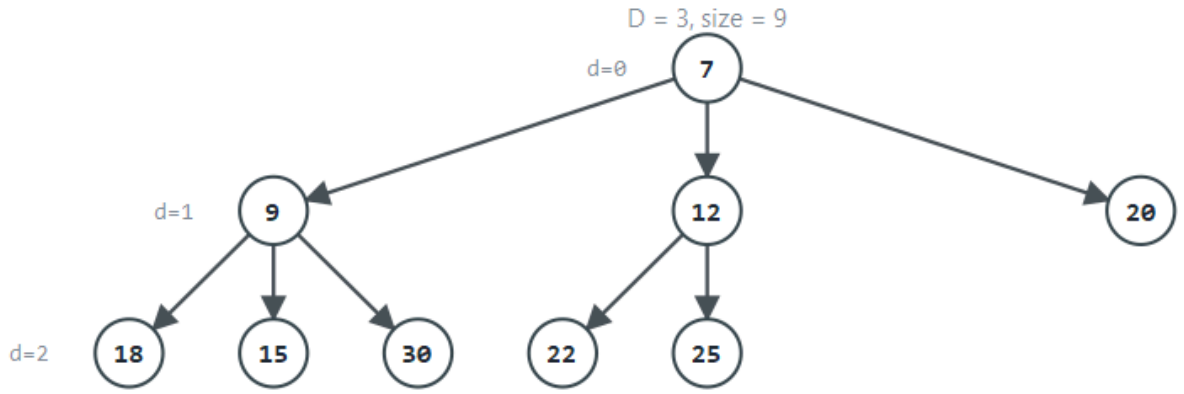
Aynı dizi tabanlı fikir, ama her düğümün 2 yerine en fazla **D** çocuğu var. Düğüm *i*'nin *c*. çocuğu `D*i + 1 + c` de; ebeveyni `(i-1)/D` de. D = 2, düz ikili öbeği verir.

D-ary öbek çıkarma, adım adım



Bitti: çıkarılan sıra [1, 3, 6]; kalan 9 değer: [7, 9, 12, 15, 18, 20, 22, 25, 30]. Yükseklik yalnızca log_D(n) olduğundan sift-down O(log_D n) sürer -- daha büyük D, daha kısa ağaç ama karşılaştırma başına daha çok çocuk demek.

Uç durum — D=3, sonuna kadar



Bitti: çıkarılan sıra [1, 3, 6]; kalan 9 değer: [7, 9, 12, 15, 18, 20, 22, 25, 30]. Yükseklik yalnızca log_D(n) olduğundan sift-down O(log_D n) sürer -- daha büyük D, daha kısa ağaç ama karşılaştırma başına daha çok çocuk demek.

Kod — D çocuklu extract()

```
int extract(void) {
    int best = heap[0];
    size--;
    heap[0] = heap[size];
    int i = 0;
    while (1) {
        int target = i, base = D * i + 1;
        /* ... base..base+D-1'de D çocuğu kontrol et ... */
        if (target == i) break;
        /* ... heap[i]/heap[target] değiştir ... */
    }
    return best;
}
```

Karmaşıklık

- `extract` : $O(D \cdot \log_D n)$ — daha az seviye, seviyede daha çok
- `insert` : $O(\log_D n)$ — her zaman tek bir ebeveynle karşılaştırır
- Büyük D: ucuz ekleme, potansiyel olarak maliyetli çıkarma

Sık yapılan hatalar

- İkili öbeğin `2*i+1` / `2*i+2` formüllerini değiştirmeden yeniden kullanmak
- Sadece daha kısa bir ağaç için büyük bir D seçmek, extract'in maliyetini gözden kaçırarak

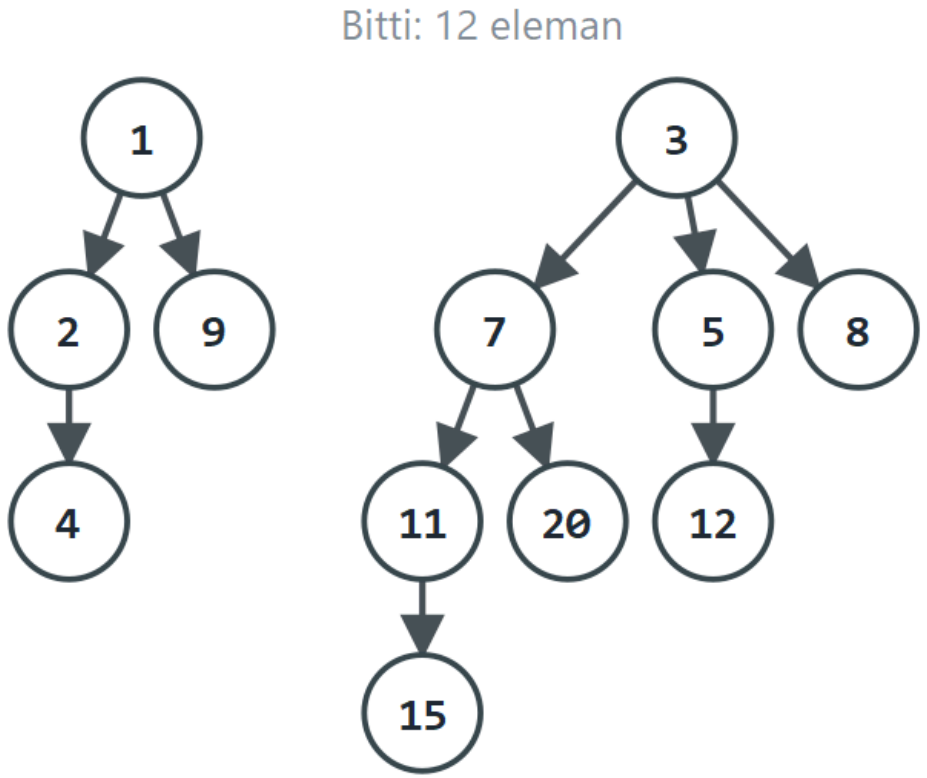
Binom öbeği

Jean Vuillemin tarafından 1978'de tanıtıldı. Binom ağaçlarından bir **orman**; *k*. dereceden ağacın 2^k düğümü var. *n* elemanlı bir öbeğin dereceleri, tam olarak n'nin ikilik tabandaki **1 bitleri**.

Union: elde ile ikilik toplama

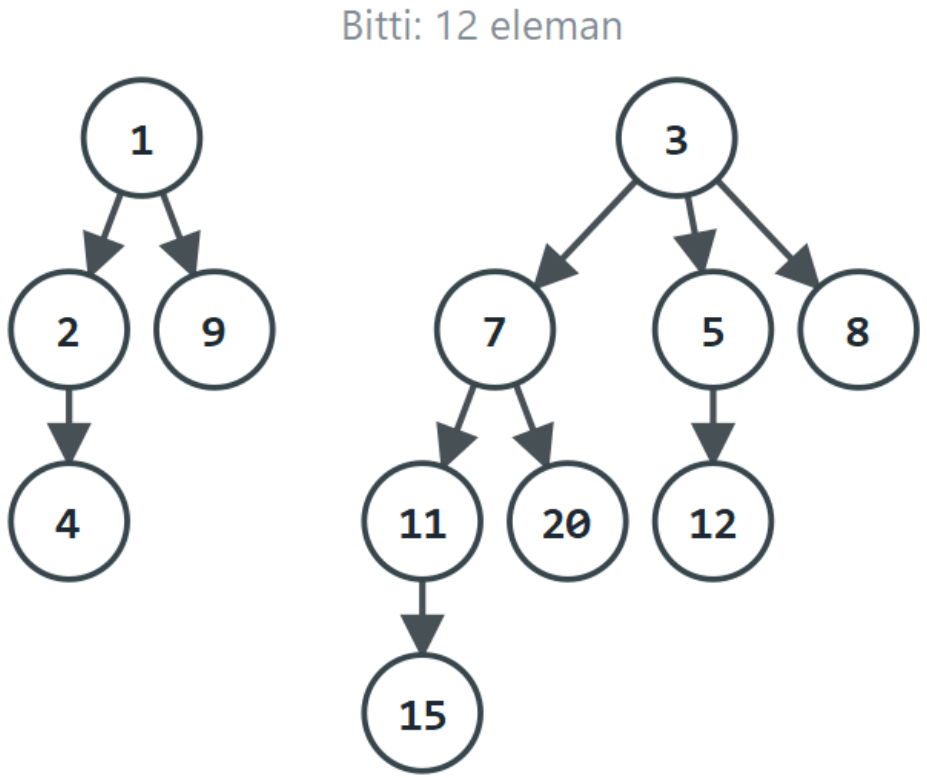
- Dereceleri düşükten yükseğe gezin, iki sayı toplar gibi
- Yalnızca biri bu derecede ağaç tutuyorsa → doğrudan geçer
- **İkisi** de tutuyorsa → **link**lenirler, bir derece yukarı "elde" olur
- Aynı anda en çok **üç** ağaç buluşabilir: A, B, ve gelen elde

Binom öbeği birleştirme, adım adım



Bitti: birleşmiş öbek, sıralar [2, 3], toplam 12 eleman. En iyi değer (1) bir kökte -- her binom ağacının kökü, kendi alt ağacındaki en iyi değerdir. union maliyeti O(log n): en fazla log n sıra gezilir.

Uç durum — tek büyük bir elde zinciri



Bitti: birleşmiş öbek, sıralar [2, 3], toplam 12 eleman. En iyi değer (1) bir kökte -- her binom ağacının kökü, kendi alt ağacındaki en iyi değerdir. union maliyeti O(log n): en fazla log n sıra gezilir.

Kod — union_heaps()

```
void union_heaps(Node *a[], Node *b[], Node *result[]) {
    Node *carry = NULL;
    for (int order = 0; order < MAX_ORDER; order++) {
        Node *group[3]; int g = 0;
        if (a[order]) group[g++] = a[order];
        if (b[order]) group[g++] = b[order];
        if (carry) group[g++] = carry;
        carry = NULL;
        /* ... g==0: none; g==1: pass through ... */
        /* ... g>=2: link() two trees, carry up ... */
    }
}
```

Karmaşıklık

- `link` : **O(1)** — birkaç işaretçi ataması
- `union` : en çok O(log n) dereceyi ziyaret eder — **O(log n)**
- `insert` , tek bir 0. derece ağaçla `union` olarak tanımlanır — o da O(log n)

Sık yapılan hatalar

- Bir derecede yalnızca **iki** ağacın buluştuğunu düşünmek, elde üçüncüyü unutarak
- `insert` i sıfırdan türetmek, sadece `union` çağırmak yerine

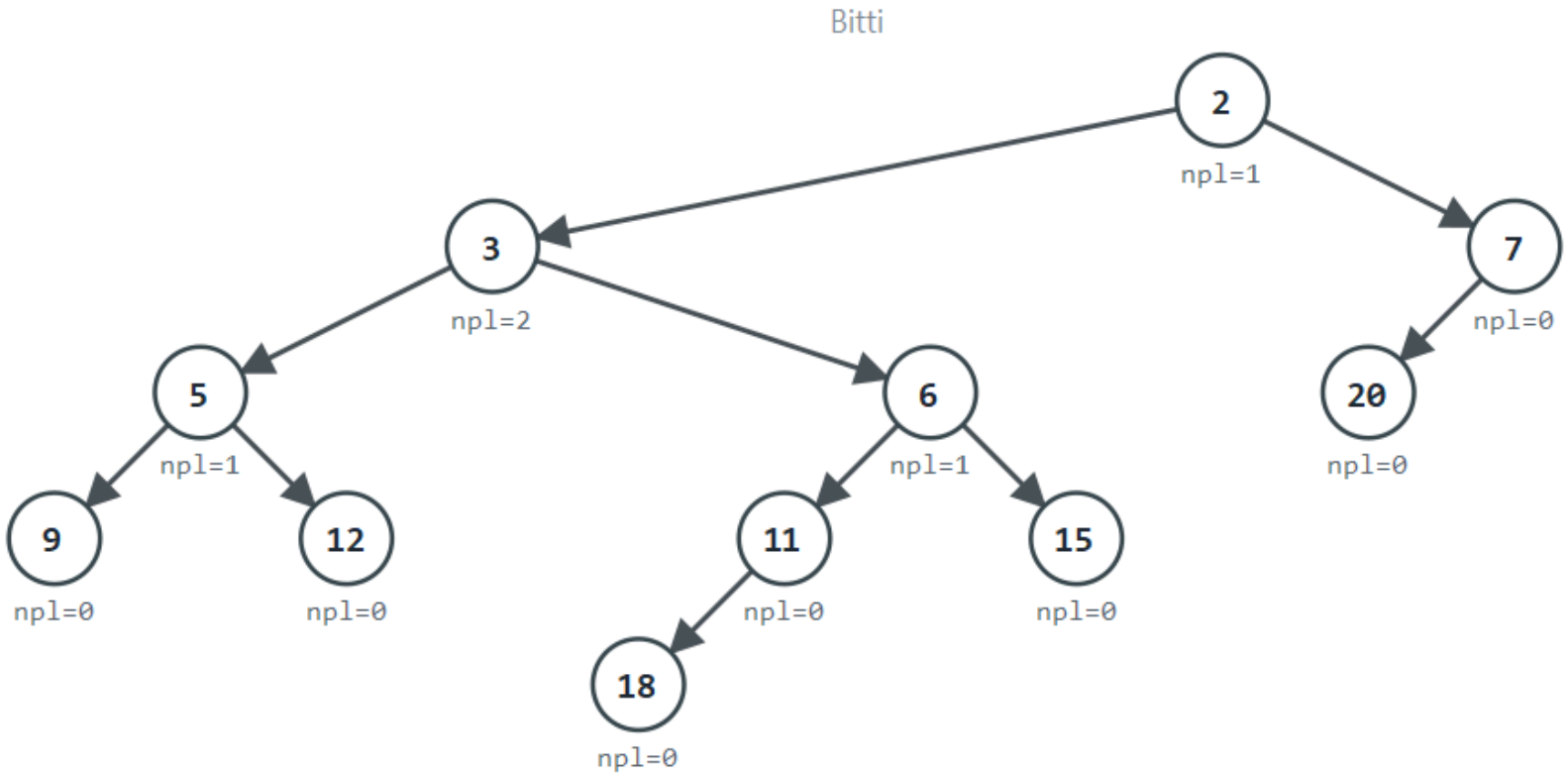
Solcu öbek

C. A. Crane tarafından 1972'de tanıtıldı.
Genelde tam **olmayan** bir işaretçi ağacı,
tek bir işlem etrafında kurulu: **merge**.
`insert` ve `extract`, onun özel durumları.

Solcu özellik

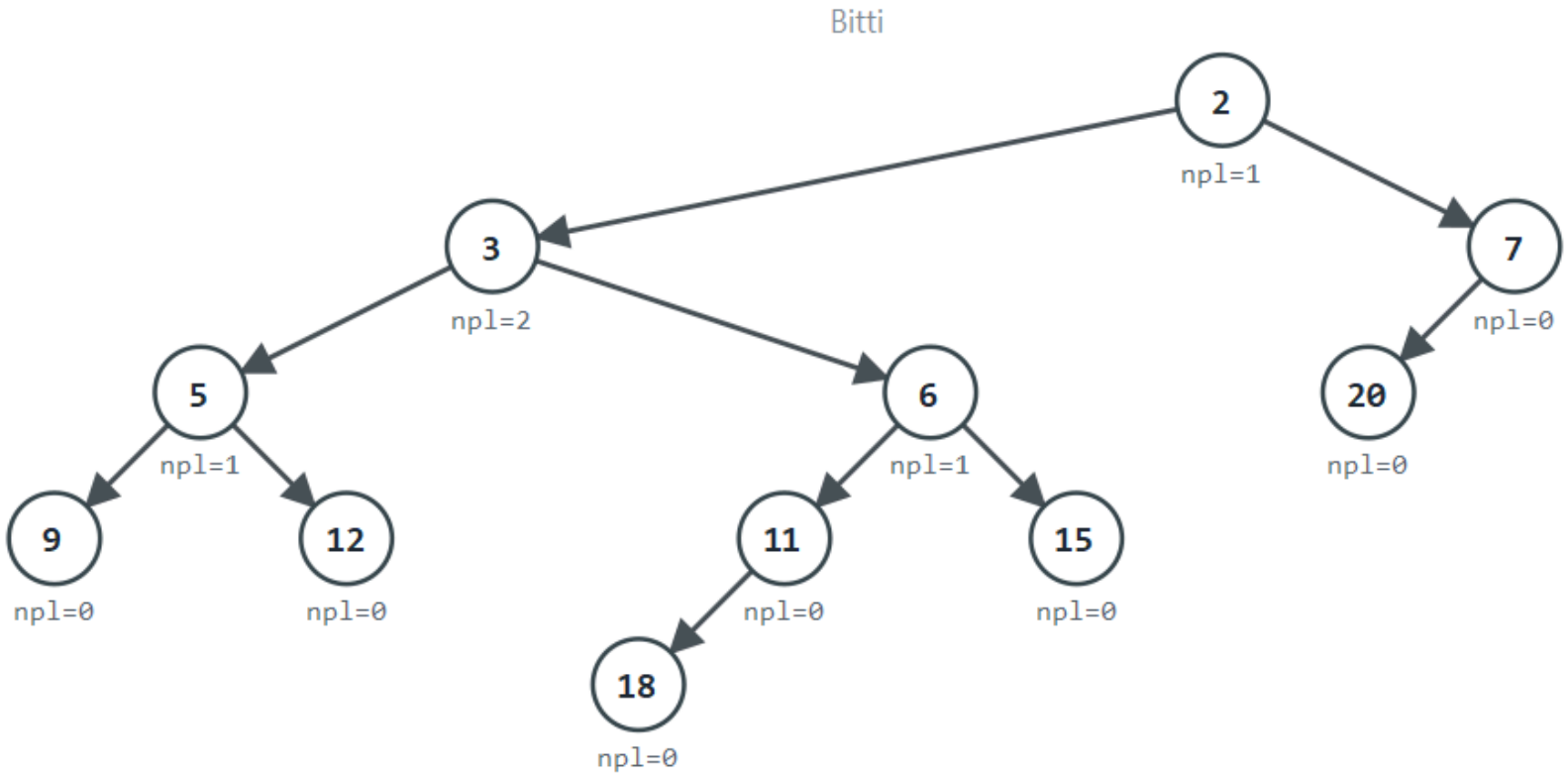
- Her düğüm kendi **null path length**ini (npl) izler
- `NULL` ın npl'si −1; bir yaprağınki 0
- **Solcu:** sol çocuğun npl'si sağınkinden asla küçük değil
- Sonuç: **sağ omurga** her zaman O(log n) uzunlukta

Solcu öbekte birleştirme, adım adım



Bitti: birleşmiş solcu öbek, 11 eleman, kökte en iyi değer: 2. merge maliyeti yalnızca sağ omurga uzunluklarına bağlı, O(log n) -- ve insert/extract de aslında birer merge çağrısıdır.

Uç durum — boş bir öbekle birleştirme



Bitti: birleşmiş solcu öbek, 11 eleman, kökte en iyi değer: 2. merge maliyeti yalnızca sağ omurga uzunluklarına bağlı, O(log n) -- ve insert/extract de aslında birer merge çağrısıdır.

Kod — merge() (özyinelemeli)

```
Node *merge(Node *t1, Node *t2) {
    if (t1 == NULL) return t2;
    if (t2 == NULL) return t1;
    if (!better(t1->key, t2->key)) {
        Node *tmp = t1; t1 = t2; t2 = tmp;
    }
    t1->right = merge(t1->right, t2);
    if (npl(t1->left) < npl(t1->right)) {
        /* ... t1->left ve t1->right yer değiştir ... */
    }
    t1->npl = npl(t1->right) + 1;
    return t1;
}
```

Karmaşıklık

- `merge` : **O(log n)**, her iki sağ omurgayla sınırlı
- `insert` , `extract` : ikisi de `merge` üzerinden tanımlı — aynı O(log n)
- Ağacın kalanı ne kadar dengesiz olursa olsun geçerli

Sık yapılan hatalar

- **npl**i (en yakın eksik çocuğa uzaklık) **yükseklik**le karıştırmak
- merge'den sonra çocuk yer değiştirmeyi atlamak — solcu özelliği sessizce bozar

Mini soru

İki büyük öbeği, tek tek eleman eklemekten
çok daha sık, birleştirmeniz gerekiyor.
Hangi varyasyon en uygun?

Cevap

Solcu öbek (ya da **binom öbeği**). İkisi de merge/union'ı yalnızca O(log n)'e mal edecek şekilde kurulu — düz ya da d-ary öbekte böyle bir yol yok.

8. Huffman Kodlaması

Başlangıç sorusu

Düz ASCII her karaktere 8 bit harcar, `E` de `Z` de aynı. Sık geçen karakterler **kısa** kod alsaydı, nadirler **uzun** — Morse kodunun zaten yaptığı gibi?

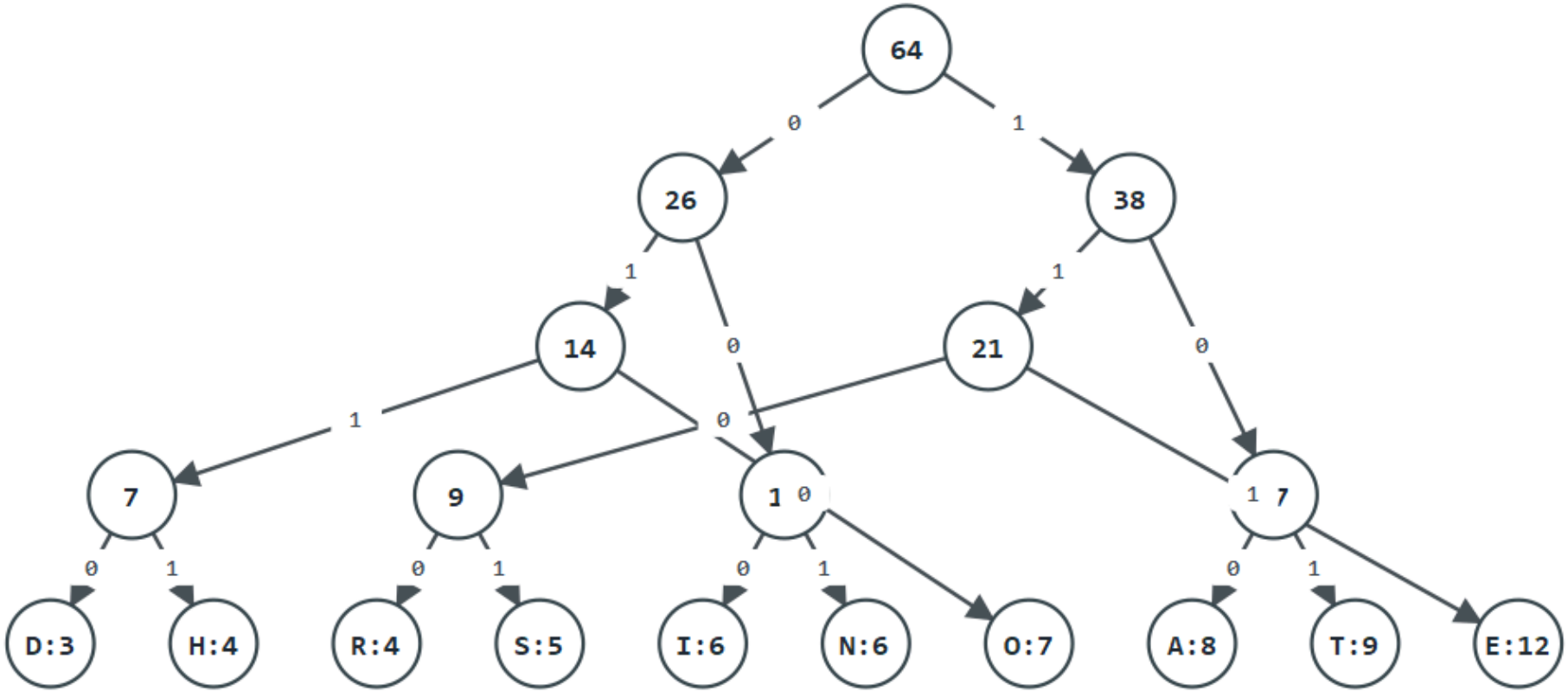
Kısa bir tarihçe

- **1952** — David Huffman, MIT'de bir dönem ödevi
- Hoca Fano'nun kendi yöntemi iyiydi, ama optimal değildi
- Huffman'ın açgözlü ağaç birleştirmesi **kanıtlanabilir** optimal
- Bugün hâlâ ZIP, JPEG ve MP3'ün içinde

Ağacı kurmak

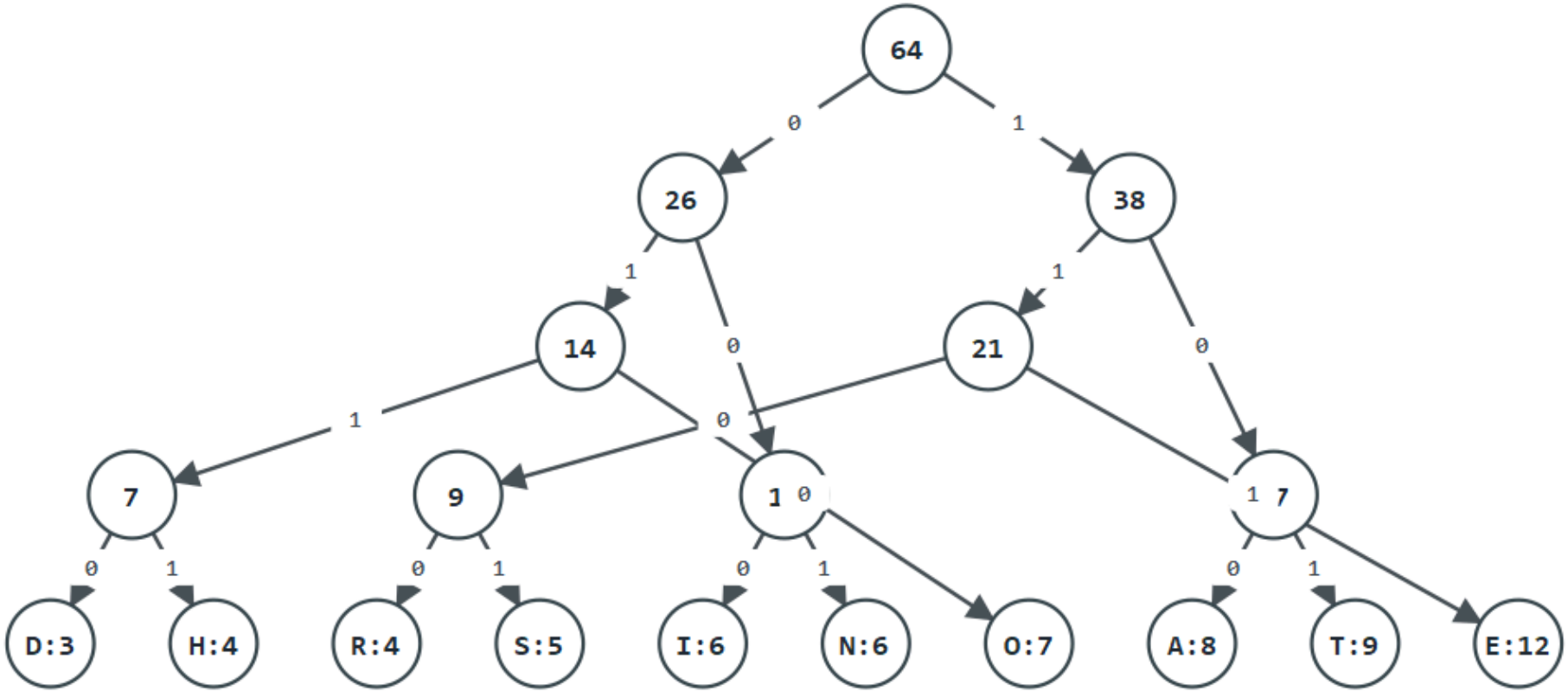
Her sembolü sıklığıyla anahtarlanmış bir **min-öbeğe** koyun. Tekrar tekrar: en **küçük** iki kökü pop edin, bu ikisini çocuk yapan yeni bir iç düğüm kurun, geri push edin. Bire kalana kadar tekrarlayın.

Huffman ağacını kurmak, adım adım



Bitti: tek düğüm kaldı, bu Huffman ağacının kökü. Toplam birleştirme maliyeti (ağırlıklı yol uzunluğu) = 208. Her kenar 0 ya da 1 -- kökten bir yaprağa giden yol, o sembolün kodudur.

Uç durum — en küçük anlamlı örnek



Bitti: tek düğüm kaldı, bu Huffman ağacının kökü. Toplam birleştirme maliyeti (ağırlıklı yol uzunluğu) = 208. Her kenar 0 ya da 1 -- kökten bir yaprağa giden yol, o sembolün kodudur.

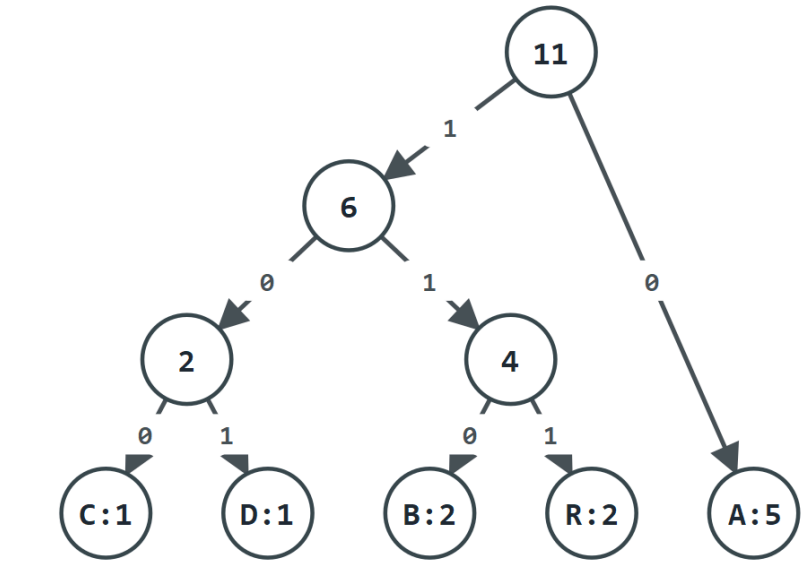
Kod — birleştirme döngüsü

```
int merge_id = 256;
while (heap_size > 1) {
    Node *a = heap_pop();
    Node *b = heap_pop();
    Node *parent = new_internal(a, b, merge_id++);
    heap_push(parent);
}
Node *root = heap_pop();
```

Kodlamak ve kod açmak

Bir sembolün **kod**u, köke kadar yolu: sol = `0`, sağ = `1`. **Kod açmak** kökten bit bit yürür; bir yaprak bir karakter üretir ve yeniden başlar. Bu, kod **önek-serbest** olduğu için çalışır.

Kodlama ve kod açma, adım adım



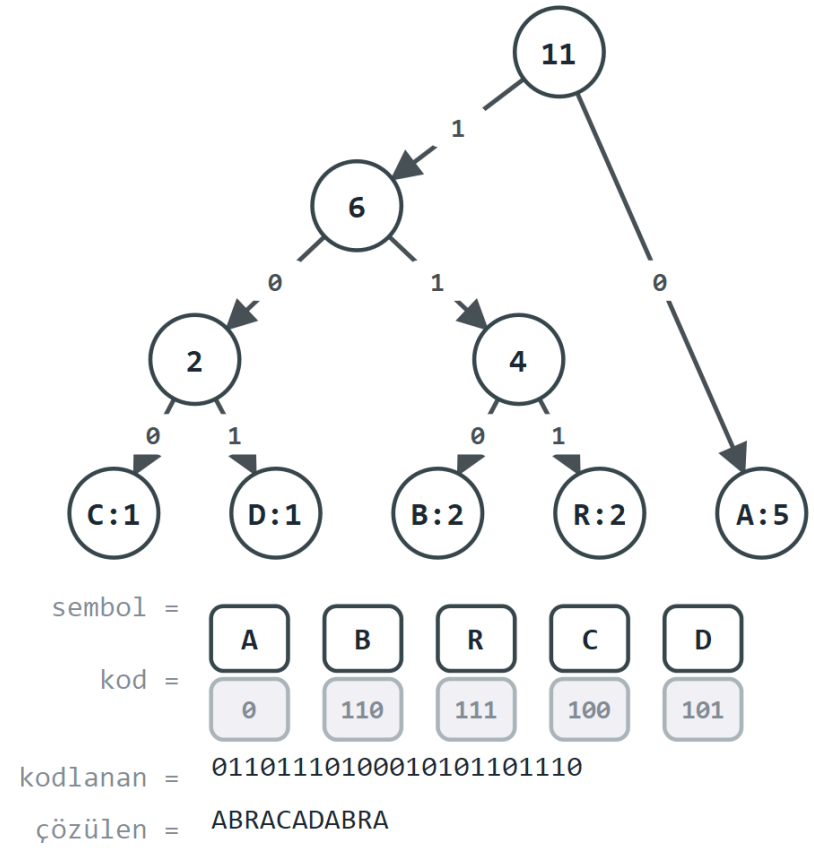
sembol =	A	B	R	C	D
kod =	0	110	111	100	101

kodlanan = 01101110100010101101110

çözülen = ABRACADABRA

Kod çözme bitti: 011011101000101011011 10 → "ABRACADABRA". Özgün metinle BİREBİR aynı. Huffman kodu önek-özgür (prefix-free) olduğu için hiçbir kod bir başkasının başlangıcı değildir -- bu yüzden tek geçişte, geri dönmeden çözülür.

Uç durum — çok çarpık: 9 A, 1 B



Kod çözme bitti: 01101110100010101101110 → "ABRACADABRA". Özgün metinle BİREBİR aynı. Huffman kodu önek-özgür (prefix-free) olduğu için hiçbir kod bir başkasının başlangıcı değildir -- bu yüzden tek geçişte, geri dönmeden çözülür.

Kod — assign_codes() (ağacı gezmek)

```
static void assign_codes(Node *node, char *path,
                         int depth) {
    if (node->left == NULL && node->right == NULL) {
        path[depth] = '\0';
        strcpy(codes[(unsigned char) node->ch], path);
        return;
    }
    path[depth] = '0';
    assign_codes(node->left, path, depth + 1);
    path[depth] = '1';
    assign_codes(node->right, path, depth + 1);
}
```

Kod — decode() (bit bit yürümek)

```
static char *decode(const char *bits, Node *root,
                    char *out) {
    int n = 0;
    Node *node = root;
    for (int i = 0; bits[i] != '\0'; i++) {
        node = bits[i] == '0' ? node->left : node->right;
        if (node->left == NULL && node->right == NULL) {
            out[n++] = node->ch;
            node = root;
        }
    }
    out[n] = '\0';
    return out;
}
```

Karmaşıklık

- Kurma: n − 1 birleşme, her biri O(log n) — **O(n log n)**
- Kodlama: **O(L)** — karakter başına bir arama ve ekleme
- Kod açma: **O(B)** — kodlanmış bit başına bir ağaç adımı

Sık yapılan hatalar

- Belirleyici bir tie-breaker olmadan — eşit sıklıklar iki farklı, ikisi de optimal ağaç kurabilir
- Kod tablosunun ASCII gibi sabit olduğunu sanmak — **mesaj başına** kurulur
- Yalnızca `decode` i test etmek, tam `decode(encode(text)) == text` turunu değil

Mini soru

Bir Huffman ağacında, bir sembolün kodu başka bir sembolün kodunun **öneki** olabilir mi?

Cevap

Hayır. Her sembol tam olarak bir
yapraktır, ve bir yaprağın çocuğu yok —
o yüzden hiçbir kod daha uzun, farklı bir koda uzatılamaz.

Özet — ağaçlar, biçimler, dolaşmalar, dizi

Fikir	Anahtar gerçek
Ağaç	Bir kök, çevrim yok, n düğüme n−1 kenar
İkili ağaç biçimleri	Dolu, tam, mükemmel, dejenere, dengeli
Dolaşmalar	Pre/in/post (özyinelemeli), yinelemesiz, level order
Dizi gösterimi	`2i+1` , `2i+2` , `(i-1)/2` — hepsi O(1)

Özet — öbekler, öncelik kuyruğu, Huffman

Fikir	Anahtar gerçek
İkili öbek	insert/extract O(log n); kurma O(n)
Öbek sıralaması	O(n log n), yerinde, **kararlı değil**
Öncelik kuyruğu	`update_key` kalıcı bir id ister
Varyasyonlar	d-ary (hızlı insert), binom/solcu (hızlı merge)
Huffman kodlaması	Ağaçlardan bir min-öbek; önek-serbest kodlar

Büyük resim

Tek bir yapı, **öbek**, iki hareketten —
sift-up, sift-down — kurulu, bir sıralama,
bir öncelik kuyruğu, üç varyasyon, ve
optimal bir sıkıştırma kodu üretiyor.

Kendini sınama turu

Dört kısa soru. Cevap bir sonraki slaytta gelmeden önce düşünün. Tam alıştırmalar ve on soruluk bir sınav hafta notlarında.

1. İkili bir ağacın yüksekliği 4 ise en çok kaç düğümü olabilir?

$2^5 - 1 = 31$ düğüm — o yükseklikte mükemmel bir ağaç.

2. Tam bir ağaç bir dizide saklıyken, 9. düğümün ebeveyn indisi nedir?

`(9 - 1) / 2 = 4`, tam sayı bölmesiyle.

3. n elemandan bir öbek kurmak neden "mantıklı görünen" O(n log n) değil, O(n)?

Çoğu düğüm alttadır, sift-down'ın ucuz olduğu yerde; toplam O(n)'e yakınsar.

4. Bir öncelik kuyruğunda her öğe neden kalıcı, sabit bir id ister?

Bir öğenin dizideki konumu neredeyse her işlemde değişir; yalnızca id sabit kalır.

Gelecek hafta

Hafta 5 — Çizgeler ve Dolaşmalar

"Çevrim yok, bir ebeveyn" kuralını bırakın,
bir ağaç bir **çizge** olur. Level order,
BFS'e genelleşir; bugün kurduğunuz öncelik
kuyruğu, Dijkstra'nın algoritmasının motoru olur.

Kaynaklar (1/2)

- Ders izlencesi, Hafta 4: `docs/syllabus/syllabus.tr.md`
- Cormen, Leiserson, Rivest, Stein. *Introduction to Algorithms*, 4. baskı. MIT Press
- Sedgewick, Wayne. *Algorithms*, 4. baskı. Addison-Wesley
- Knuth. *The Art of Computer Programming, Cilt 1*, 3. baskı

Kaynaklar (2/2)

- Huffman (1952) · Williams (1964) · Floyd (1964)
- Vuillemin (1978) — binom öbeği
- Cayley (1857) — ilk ağaç sayma makalesi
- williamfiset/Algorithms · Programiz DSA