طراحی الگوریتم‌ها – مرتب‌سازی با هیپ

Introduction to Algorithm

مهدی جوانمردی

۱۴۰۱

مرور جلسه قبل

چالش‌های روش جایگذاری

حدس $T(n) = O(n) \longrightarrow T(n) \le cn \; \times \longrightarrow$ $\boxed{T(n) \le cn - d, \text{ where } d \ge 0 \text{ is a constant.}}$

۱. درجه جمله اضافه کمتر از حکم باشد: به حکم یک جمله از درجه کمتر اضافه میکنیم

۲. درجه جمله اضافه با حکم برابر باشد: یک فاکتور لگاریتم در حکم کمتر حدس زدیم

۳. درجه جمله اضافه بیشتر از حکم باشد: باید حکم از درجه بالاتری باشد

تغییر متغیر برای حل رابطه بازگشتی

$$\boxed{T(n) = 2T(\sqrt{n}) + \lg n}, \quad m = \lg n \quad \Rightarrow \quad T(2^m) = 2T(2^{m/2}) + m, \quad S(m) = T(2^m) \quad \Rightarrow \quad S(m) = 2S(m/2) + m$$

$$\downarrow$$

$$S(m) = O(m \lg m)$$

$$\downarrow$$

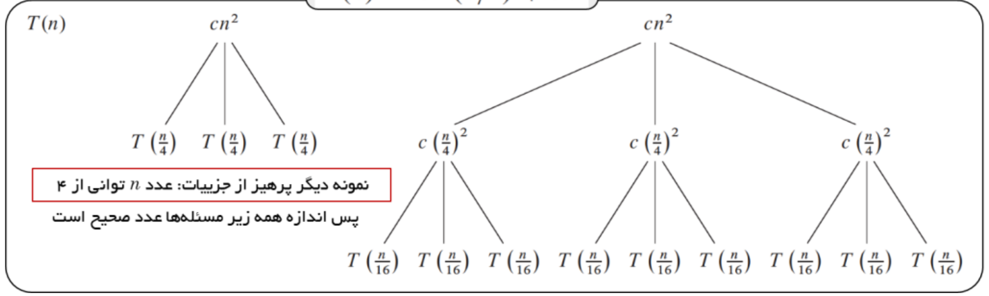
$$\boxed{T(n) = T(2^m) = S(m) = O(m \lg m) = O(\lg n \lg \lg n)}$$

حل رابطه بازگشتی با درخت بازگشتی

$$T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$$

نمونه پرهیز از جزییات: محاسبه حد بالا با حذف کف

$$T(n) = 3T(n/4) + cn^2$$



اثبات حدس به دست آمده از طریق استقراء

حل روابط بازگشت با قضیه اصلی

$$T(n) = aT(n/b) + f(n) \qquad a \ge 1 \text{ and } b > 1$$

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $\boxed{T(n) = \Theta(n^{\log_b a})}$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $\boxed{T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(f(n) \lg n)}$
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \le cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $\boxed{T(n) = \Theta(f(n))}$. شرط منظم بودن

$$f(n) \; \square \; n^{\log_b a}$$

فصل ششم: مرتب‌سازی هرمی | Heap sort

- مرتب‌سازی هرمی یا Heap sort
  - Heap چیست
  - نگهداری خاصیت Heap
  - درست کردن Heap
  - الگوریتم مرتب‌سازی هرمی
  - صف‌های اولویت

II Sorting and Order Statistics

Introduction ***147***

6 Heapsort ***151***

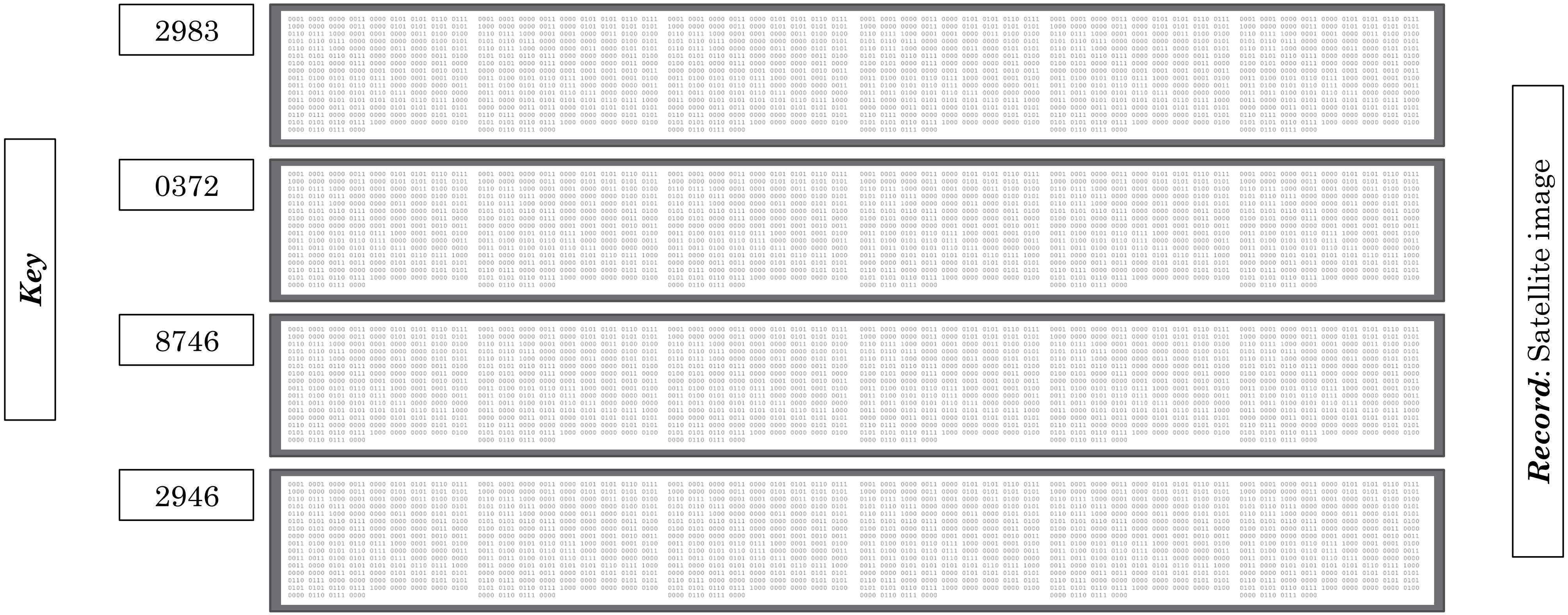
6.1 Heaps *151*
6.2 Maintaining the heap property *154*
6.3 Building a heap *156*
6.4 The heapsort algorithm *159*
6.5 Priority queues *162*

7 Quicksort ***170***

7.1 Description of quicksort *170*
7.2 Performance of quicksort *174*
7.3 A randomized version of quicksort *179*
7.4 Analysis of quicksort *180*

مرتب‌سازی و ساختار داده‌ای

- درواقعیت اعدادی که نیاز به مرتب‌سازی دارند معمولا اعداد منفرد نیستند



- در عمل درهنگام جابجایی key توسط الگوریتم مرتبط‌سازی record نیز باید جابجایی شود
- اگر حجم داده record زیاد باشد آرایه‌ای از اشاره‌گرها به داده جابجا میشوند و نه خود داده

مقایسه الگوریتم‌های مرتب‌سازی

only a constant number of elements of the input are ever stored outside the array.

• الگوریتم Insertion sort مرتبه زمانی: worst-case running time $\Theta(n^2)$ expected running time $\Theta(n^2)$	$\boxed{\text{sorts } \textbf{\textit{in place}}}$	fast in-place sorting algorithm for small input
• الگوریتم Merge sort مرتبه زمانی: running time $\Theta(n \lg n)$	~~sorts *in place*~~ MERGE procedure	
• الگوریتم Heap sort مرتبه زمانی: worst-case running time $O(n \lg n)$	$\boxed{\text{sorts } \textbf{\textit{in place}}}$	important data structure, called a heap priority queue
• الگوریتم Quick sort مرتبه زمانی: worst-case running time $\Theta(n^2)$ expected running time $\Theta(n \lg n)$ outperforms heapsort in practice	$\boxed{\text{sorts } \textbf{\textit{in place}}}$	quicksort has tight code. popular algorithm for sorting large input arrays

مقایسه الگوریتم‌های مرتب‌سازی

$\boxed{\text{we can beat this lower bound of } \Omega(n \lg n)}$

if we can gather information about the sorted order of the input

- الگوریتم Counting sort

the input numbers are in the set $\{0, 1, \ldots, k\}$

worst-case running time $\Theta(k + n)$

expected running time $\Theta(k + n)$

مرتبه زمانی

- الگوریتم Radix sort

there are n integers to sort

integer has d digits

digit can take on up to k possible values

worst-case running time $\Theta(d(n + k))$

expected running time $\Theta(d(n + k))$

مرتبه زمانی

- الگوریتم Bucket sort

requires knowledge of the probabilistic distribution of numbers in the input array

real numbers uniformly distributed in the half-open interval $[0, 1)$

worst-case running time $\Theta(n^2)$

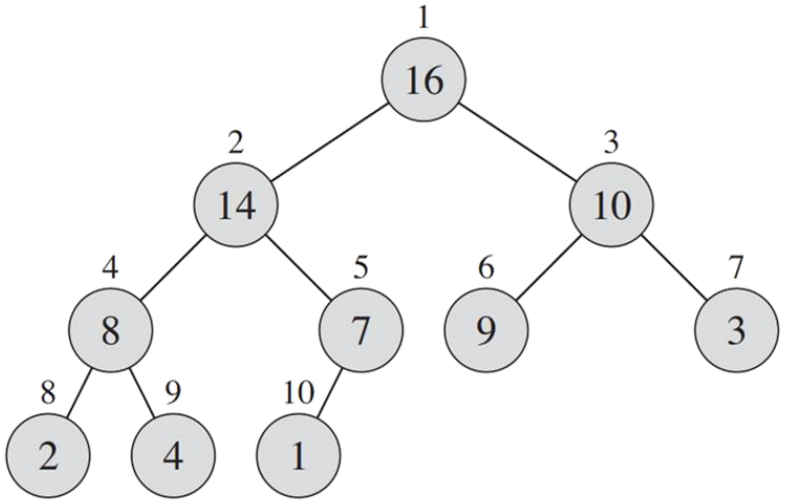
average-case running time $\Theta(n)$

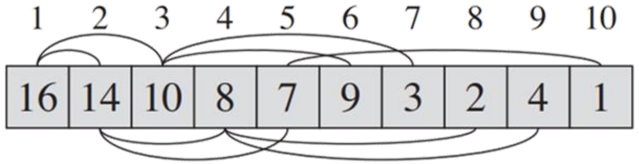
مرتبه زمانی

ساختار Heap و ویژگی‌های آن – ۱

- Heap عبارت است از یک درخت دودویی کامل (به غیر از پایین‌ترین سطح)
  - هر گره از درخت یک المان از آرایه
  - max heap: هر گره بزرگترمساوی فرزندان $A[\text{PARENT}(i)] \geq A[i]$
  - min heap: هر گره کوچکتر مساوی فرزندان $A[\text{PARENT}(i)] \leq A[i]$

max heap

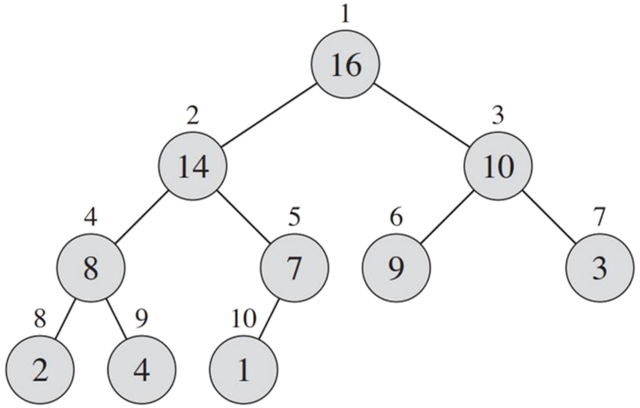


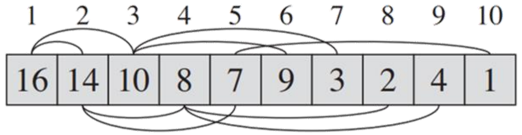


root: $A[1]$
node: $A[i]$
left-child: $A[2i]$
right-child: $A[2i+1]$
leaves: $A[(\lfloor n/2 \rfloor+1) .. n]$

ساختار Heap و ویژگی‌های آن – ۲

- Heap عبارت است از یک درخت دودویی کامل





PARENT(i)
  return $\lfloor i/2 \rfloor$

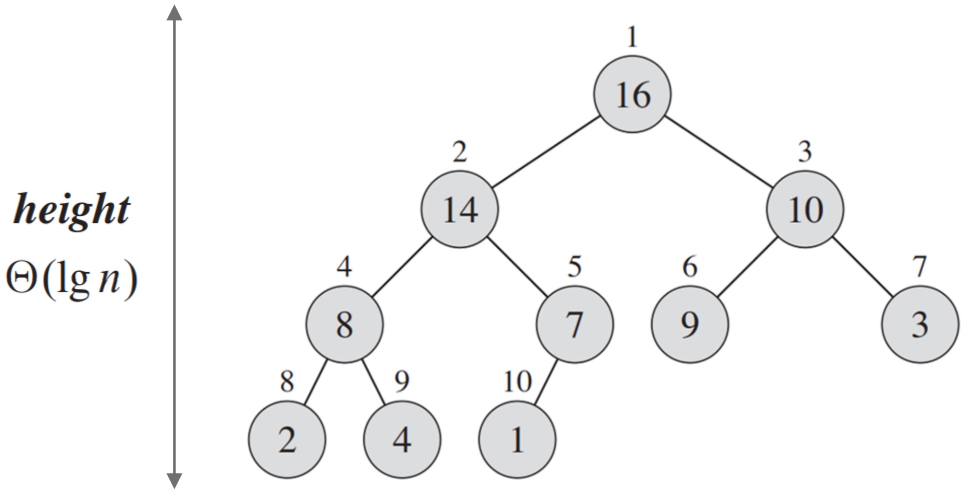
LEFT(i)
  return $2i$

RIGHT(i)
  return $2i + 1$

HEAP: array A

- $A.length$: تعداد اعداد در آرایه $A[1\,..\,A.length]$
- $A.heap\_size$: چه تعداد المان‌های هرم در آرایه مرتب شده است
  $A[1\,..\,A.heap\text{-}size]$, where $0 \le A.heap\text{-}size \le A$

عملیات‌های Heap



Maintain/Restore the max-heap property

MAX-HEAPIFY

Create a max-heap from an unordered array

BUILD-MAX-HEAP

Sort an array in place

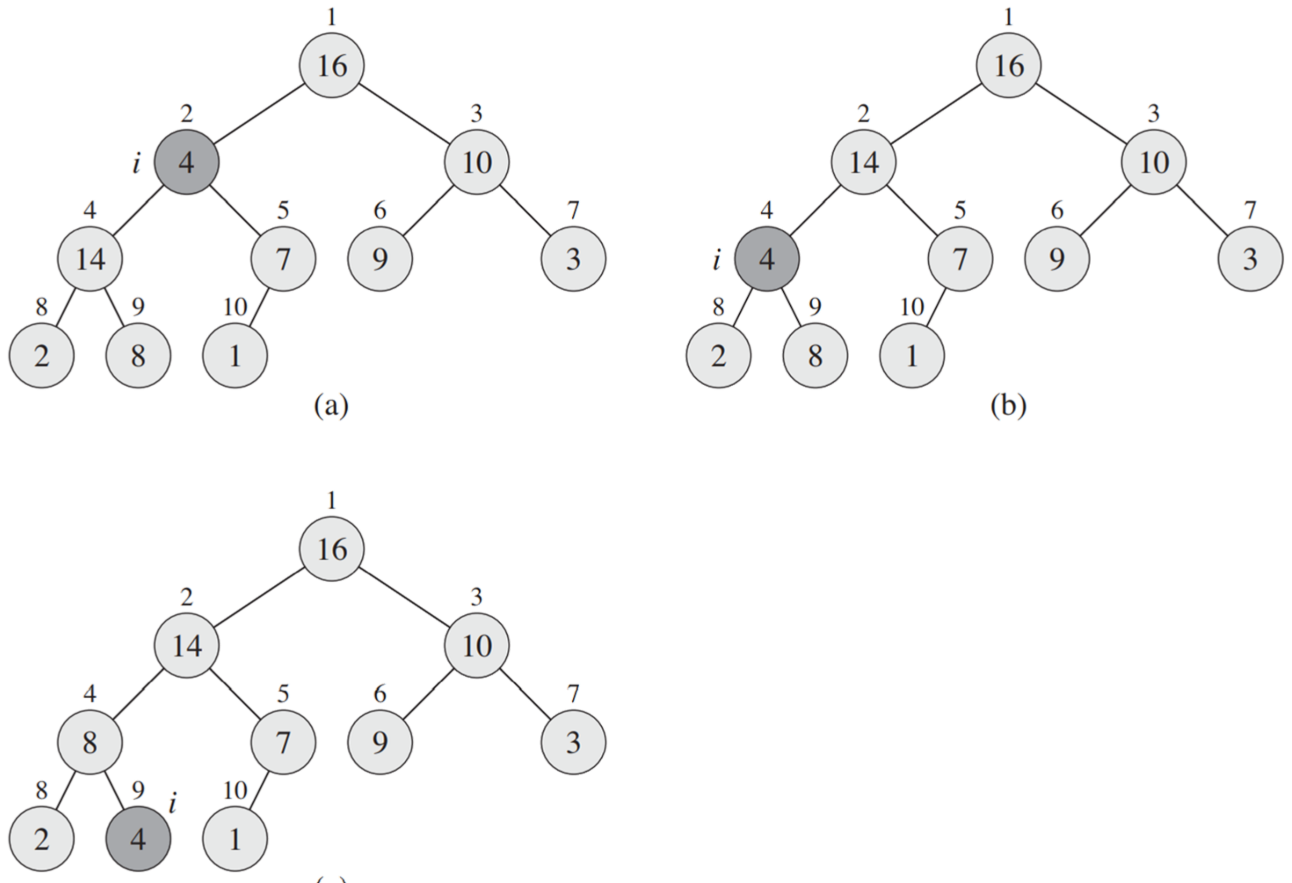
HEAPSORT

Priority queues

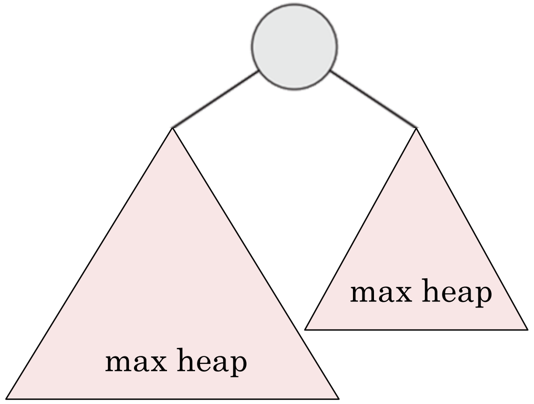
نگهداری ویژگی heap با عملیات MAX-HEAPIFY

مثال

MAX-HEAPIFY$(A, 2)$ $\qquad$ $A.heap\text{-}size = 10$.



(a) $\qquad$ (b) $\qquad$ (c)



MAX-HEAPIFY(A, i)

```
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

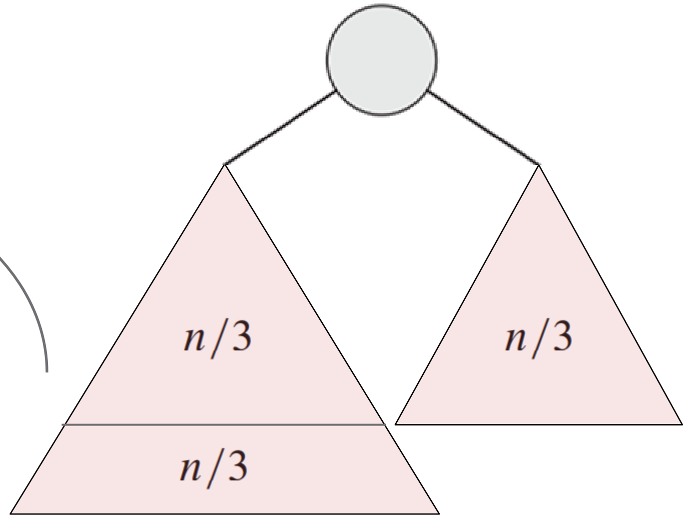
تحلیل زمانی عملیات MAX-HEAPIFY


```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

$\Theta(1)$

$T(2n/3)$

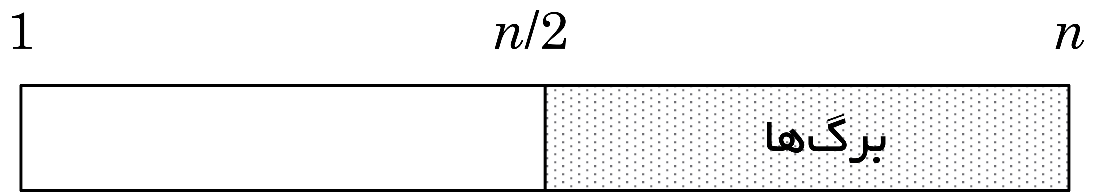
worst case



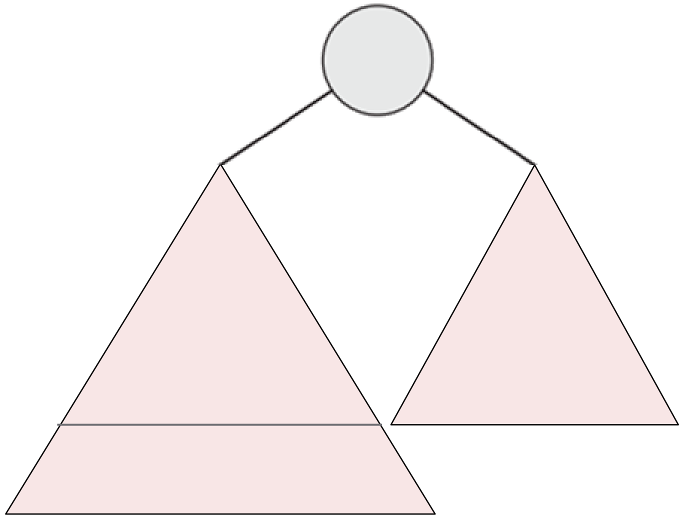
$$\boxed{T(n) \le T(2n/3) + \Theta(1)} \xrightarrow[\text{the master theorem}]{\text{case 2 of}} \boxed{T(n) = O(\lg n)}$$

running time of MAX-HEAPIFY on a node of height h as $O(h)$

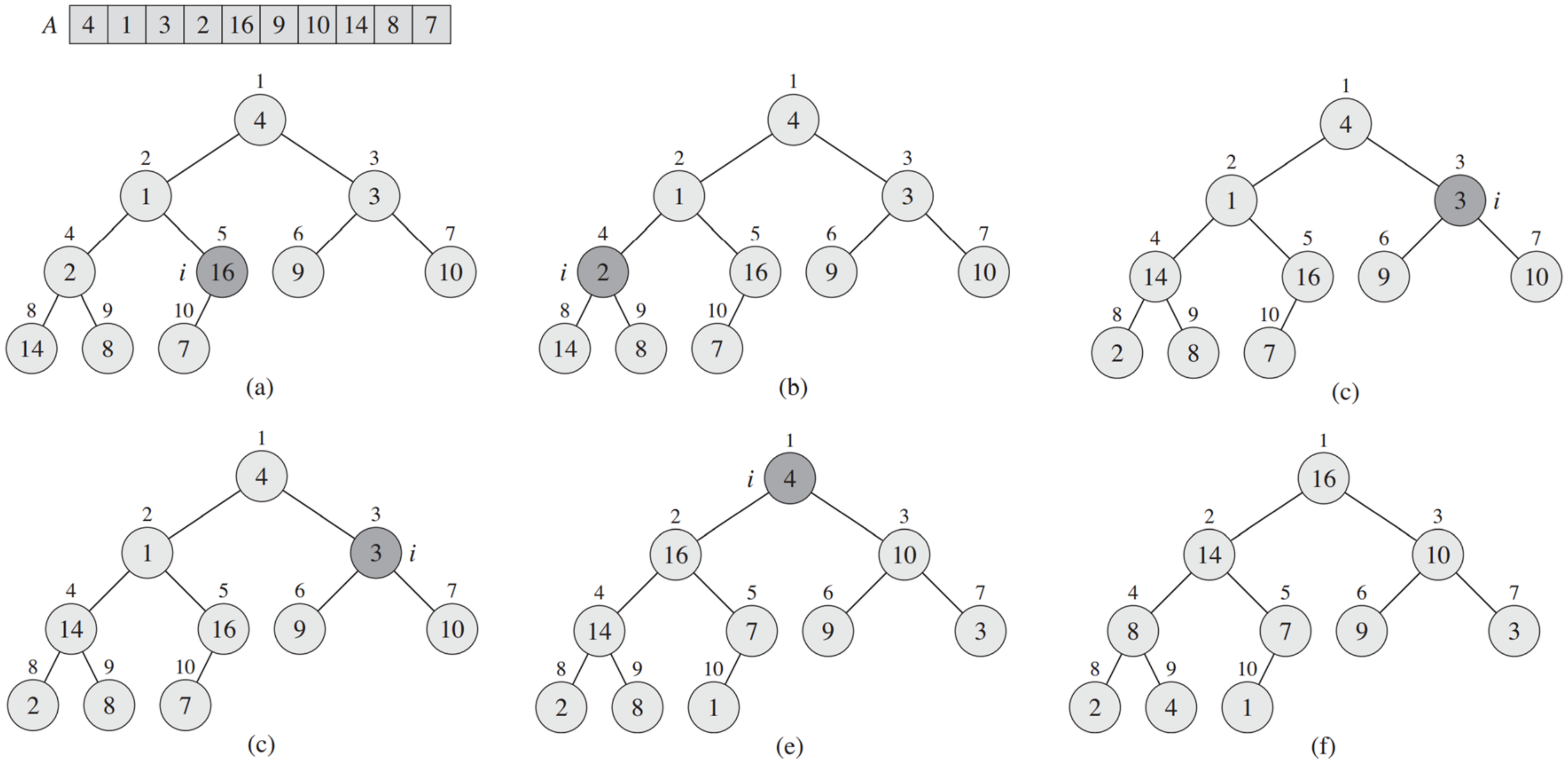
درست کردن heap




```
BUILD-MAX-HEAP(A)
  A.heap-size = A.length
  for i = ⌊A.length/2⌋ downto 1
      MAX-HEAPIFY(A, i)
```



درست کردن heap – مثال



اثبات الگوریتم BUILD-MAX-HEAP

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

مستقل از حلقه

At the start of each iteration of the **for** loop of lines 2–3, each node $i+1, i+2, \ldots, n$ is the root of a max-heap.

Initialization: Prior to the first iteration of the loop, $i = \lfloor n/2 \rfloor$. Each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ is a leaf and is thus the root of a trivial max-heap.

Maintenance: To see that each iteration maintains the loop invariant, observe that the children of node i are numbered higher than i. By the loop invariant, therefore, they are both roots of max-heaps. This is precisely the condition required for the call MAX-HEAPIFY(A, i) to make node i a max-heap root. Moreover, the MAX-HEAPIFY call preserves the property that nodes $i+1, i+2, \ldots, n$ are all roots of max-heaps. Decrementing i in the **for** loop update reestablishes the loop invariant for the next iteration.

Termination: At termination, $i = 0$. By the loop invariant, each node $1, 2, \ldots, n$ is the root of a max-heap. In particular, node 1 is.

تحلیل زمانی BUILD-MAX-HEAP

BUILD-MAX-HEAP(A)
1 $A.heap\text{-}size = A.length$
2 **for** $i = \lfloor A.length/2 \rfloor$ **downto** 1
3 MAX-HEAPIFY(A, i)

$$\underbrace{\text{MAX-HEAPIFY costs } O(\lg n) \times \text{BUILD-MAX-HEAP makes } O(n) \text{ such calls}}_{\text{the running time is } O(n \lg n)}$$

$$T(n) = \sum_{h=0}^{lg(n)} \left\lceil \frac{n}{2^{h+1}} \right\rceil * O(h)$$

$$= O(n * \sum_{h=0}^{lg(n)} \frac{h}{2^h})$$

$$= O(n * \sum_{h=0}^{\infty} \frac{h}{2^h})$$

$$= O(n * \frac{\frac{1}{2}}{(1-\frac{1}{2})^2})$$

$$= O(n * 2)$$

$$= \boxed{O(n)}$$

$$\sum_{n=0}^{\infty} x^n = \frac{1}{1-x}$$

$$\sum_{n=1}^{\infty} nx^{n-1} = \frac{d}{dx}\left[\sum_{n=0}^{\infty} x^n\right] = \frac{d}{dx}\left[\frac{1}{1-x}\right] = \frac{1}{(1-x)^2}$$

$$\sum_{n=0}^{\infty} nx^n = \frac{x}{(1-x)^2}$$

الگوریتم heap sort

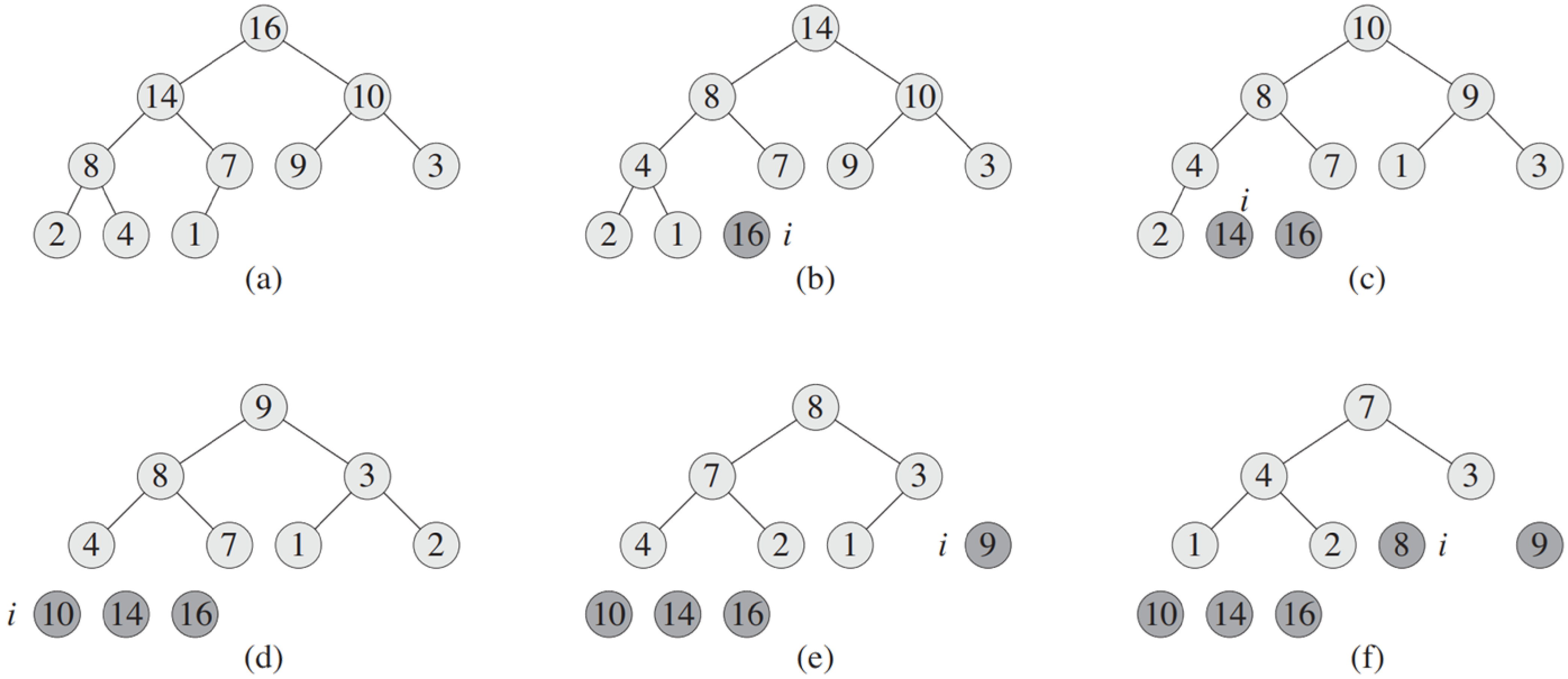
HEAPSORT(A)

BUILD-MAX-HEAP(A) $\qquad O(n)$
for $i = A.length$ **downto** 2
    exchange $A[1]$ with $A[i]$
    $A.heap\text{-}size = A.heap\text{-}size - 1$
    MAX-HEAPIFY($A, 1$) $\qquad O(\lg n)$

n times = $O(n \lg n)$

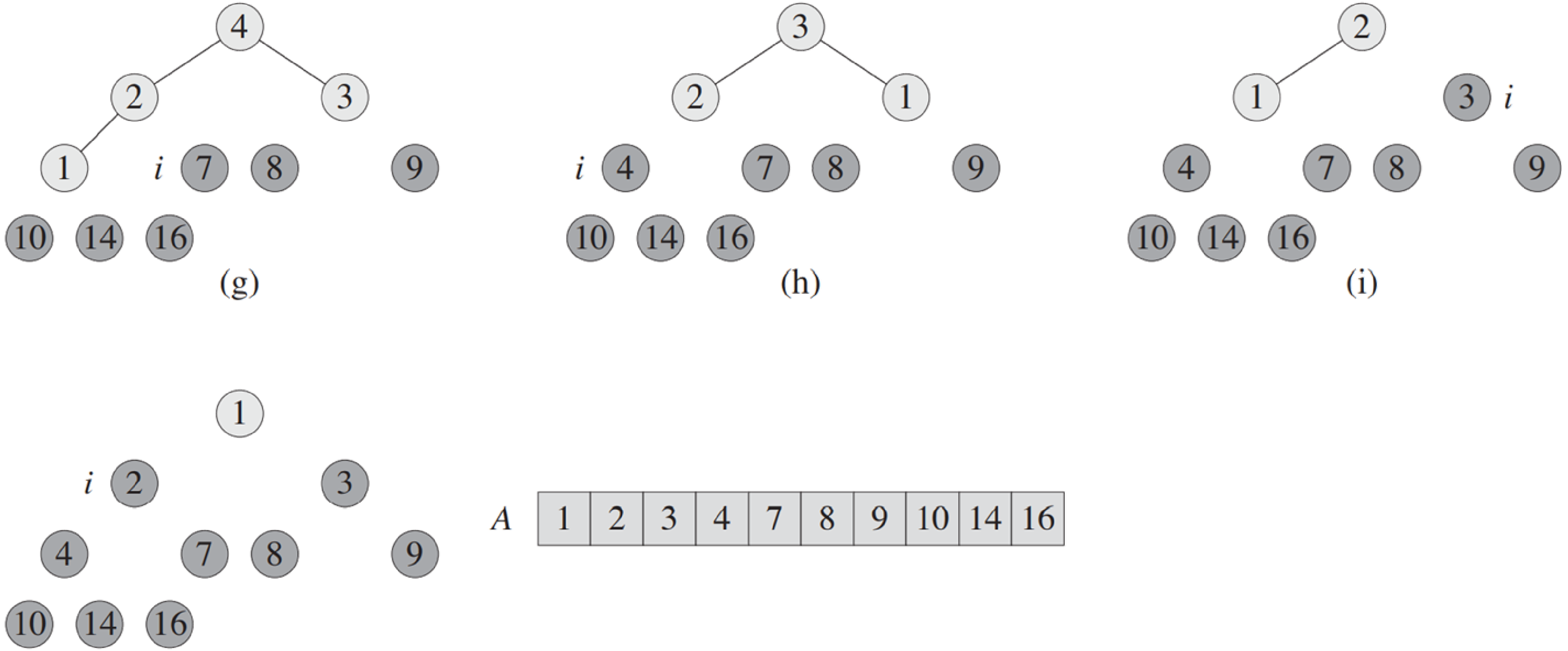
مثال heap sort

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
```



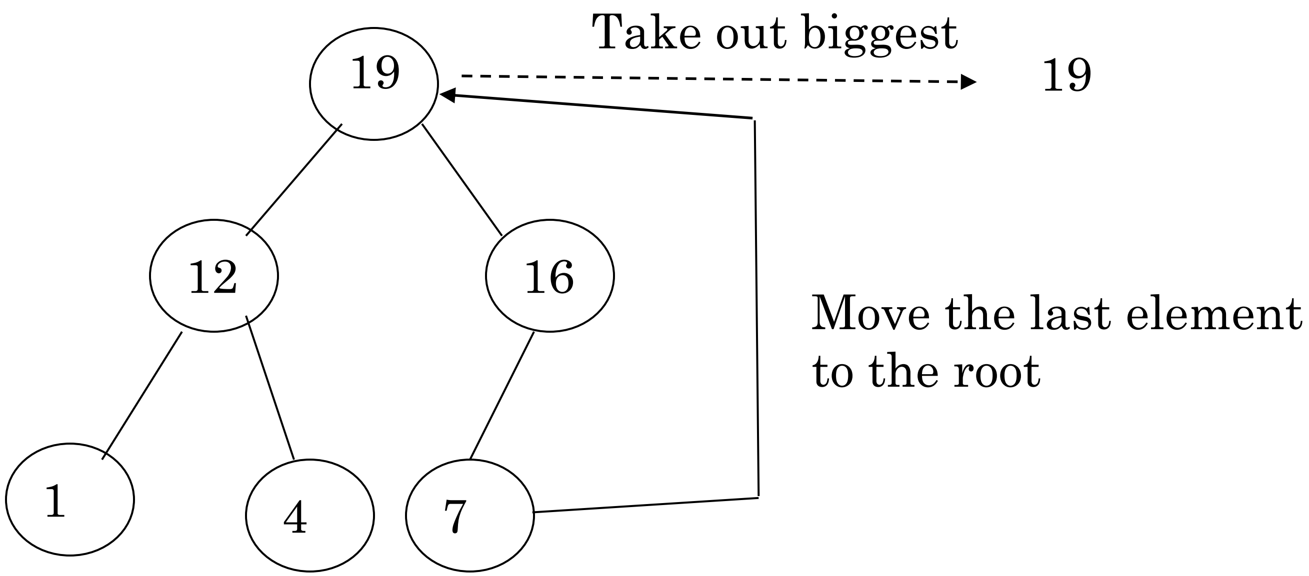
مثال heap sort

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
```



(g) (h) (i) (j) (k)

Example of Heap Sort



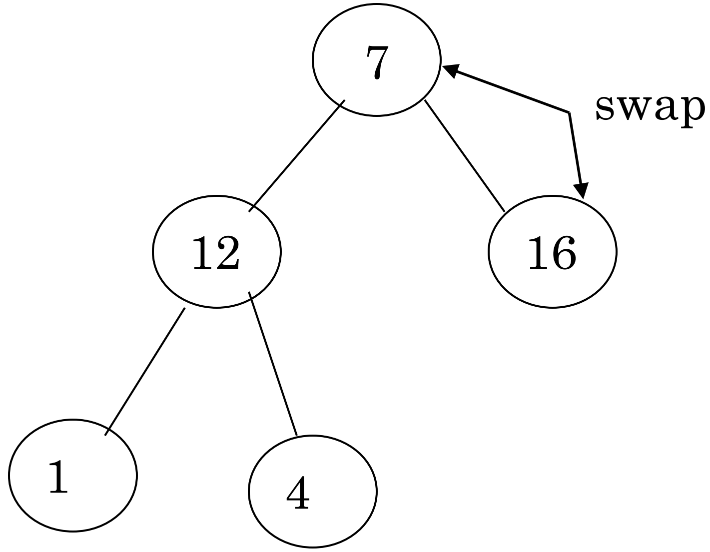
Array A

12	16	1	4	7

Sorted:

19

HEAPIFY()

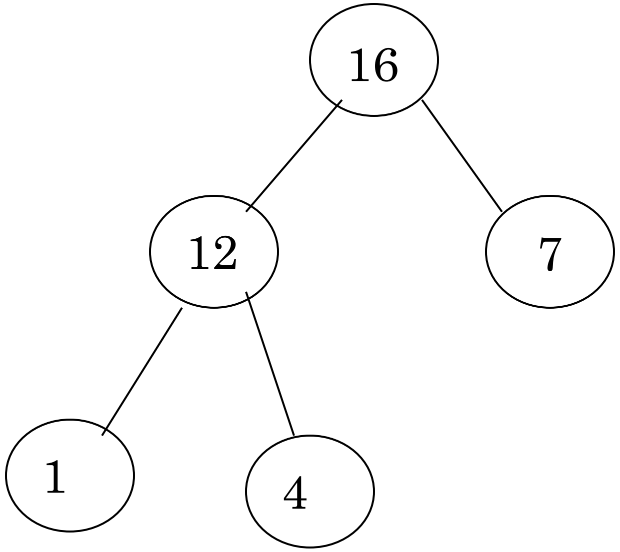


Array A

7	12	16	1	4

Sorted:

19

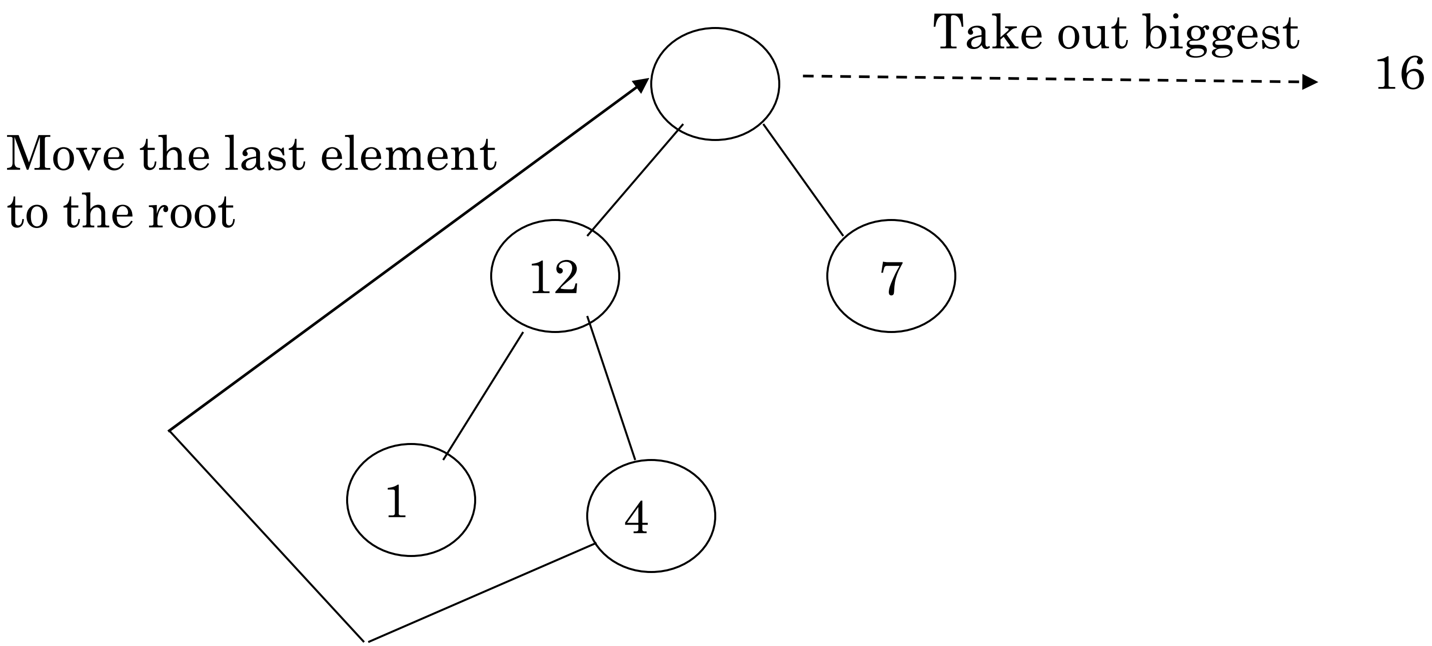


Sorted:

Array A

16	12	7	1	4

19

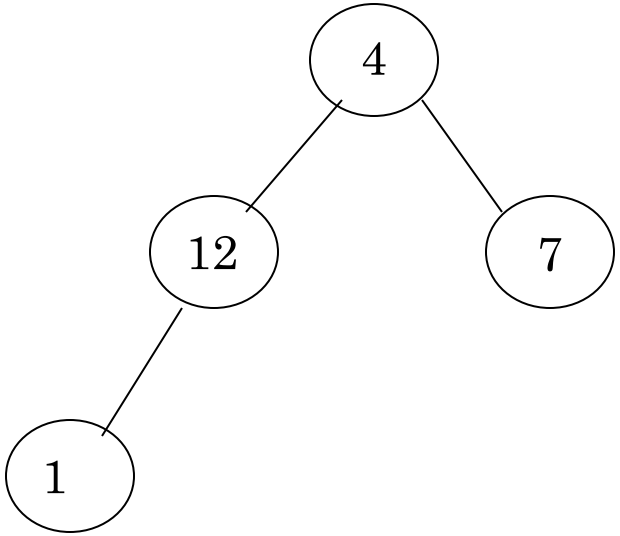


Sorted:

Array A

12	7	1	4

16	19



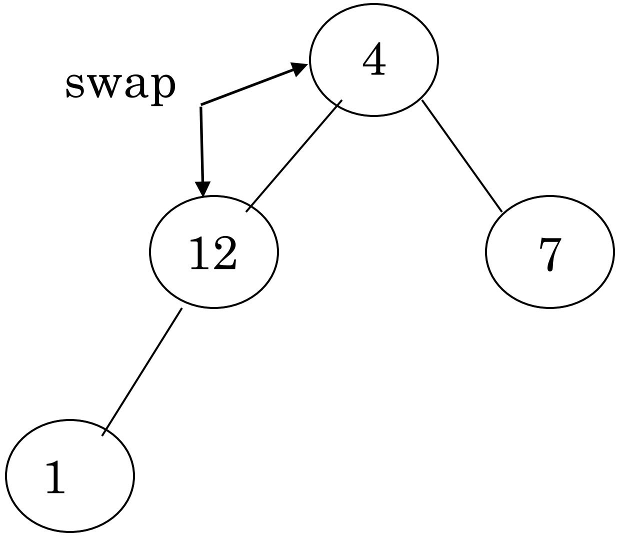
Array A

4	12	7	1

Sorted:

16	19

HEAPIFY()

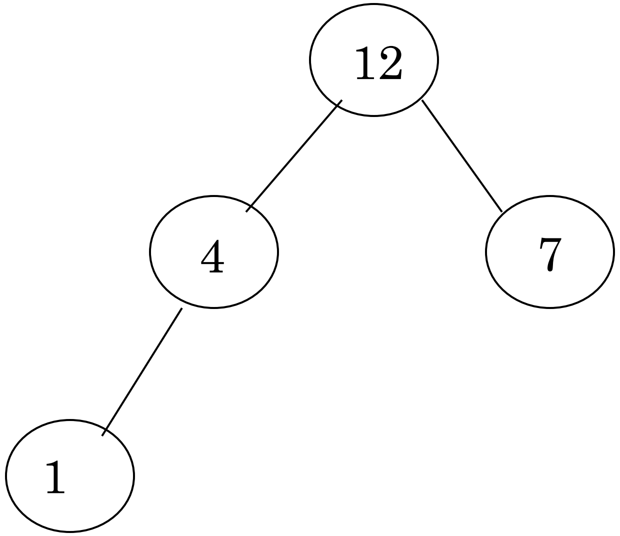


Array A

4	12	7	1

Sorted:

16	19

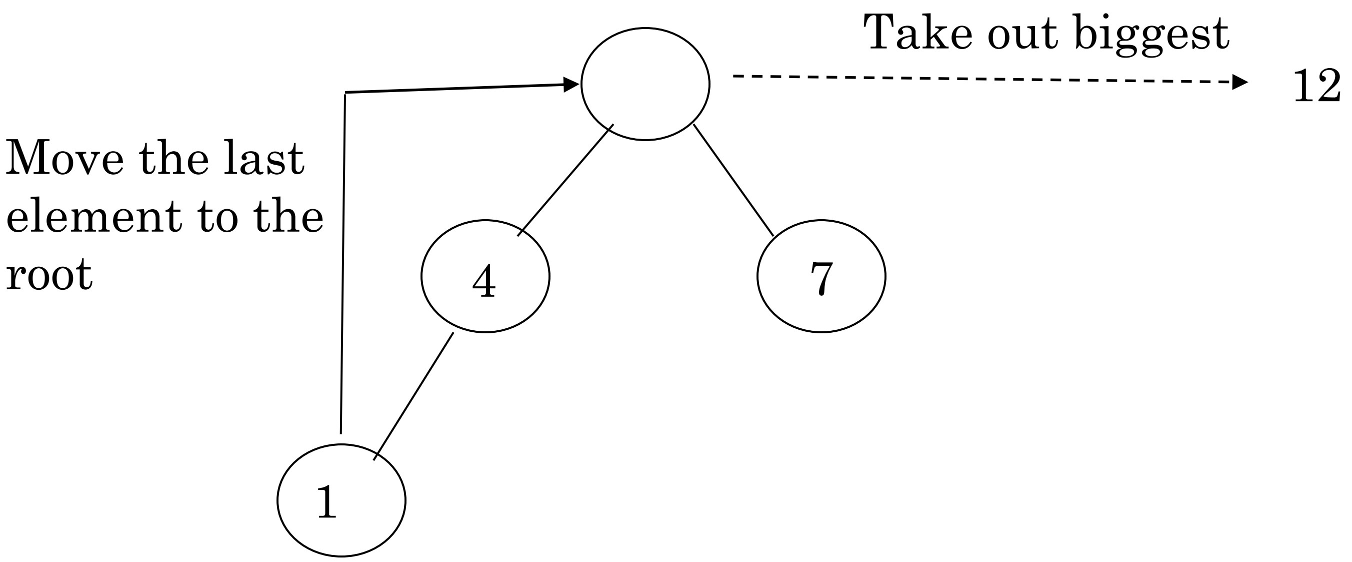


Array A

12	4	7	1

Sorted:

16	19

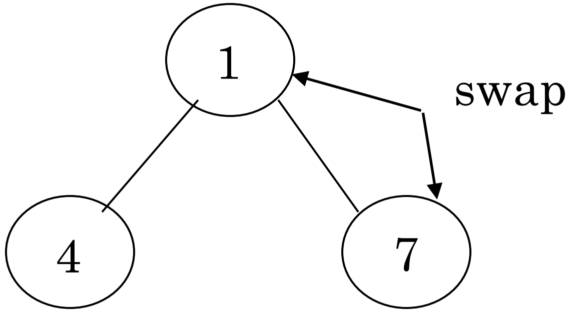


Array A

4	7	1

Sorted:

12	16	19

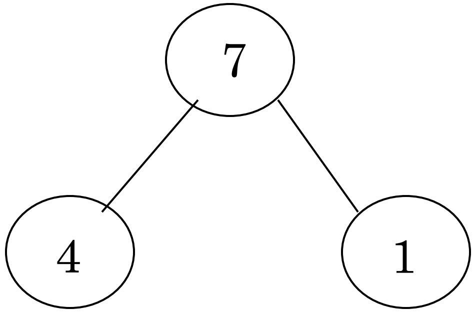


Array A

1	4	7

Sorted:

12	16	19

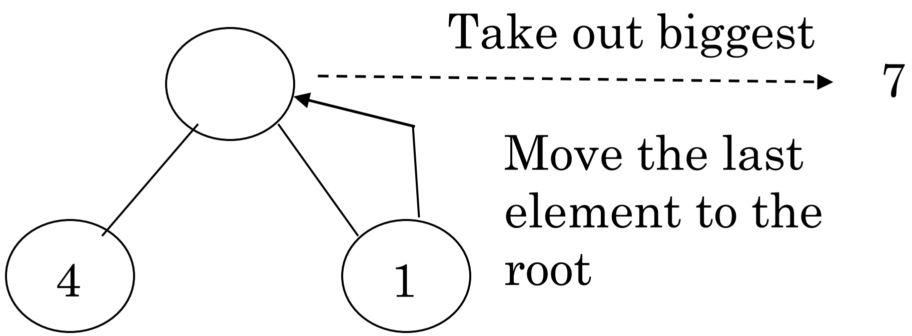


Array A

7	4	1

Sorted:

12	16	19



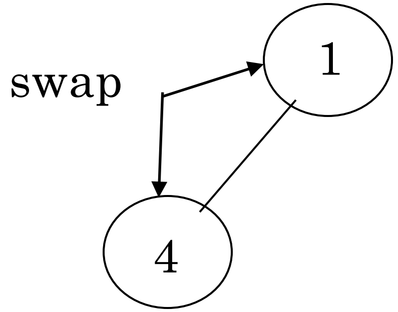
Array A

1	4

Sorted:

7	12	16	19

HEAPIFY()



Array A

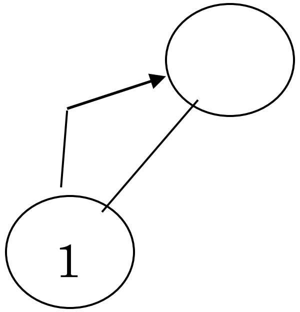
4	1

Sorted:

7	12	16	19

Take out biggest

Move the last element to the root



4

Sorted:

Array A

1

4	7	12	16	19

1

Take out biggest

Array A

Sorted:

1	4	7	12	16	19

Sorted:

1	4	7	12	16	19

اثبات الگوریتم heap sort

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size - 1
    MAX-HEAPIFY(A, 1)
```

مستقل از حلقه

امتیازی! ۳ دقیقه فرصت

صف اولویت یا priority queue

A ***priority queue*** is a data structure for maintaining a set S of elements, each with an associated value called a ***key***. A ***max-priority queue*** supports the following operations:

$\boxed{\text{INSERT}(S, x)}$ inserts the element x into the set S, which is equivalent to the operation $S = S \cup \{x\}$.

$\boxed{\text{MAXIMUM}(S)}$ returns the element of S with the largest key.

$\boxed{\text{EXTRACT-MAX}(S)}$ removes and returns the element of S with the largest key.

$\boxed{\text{INCREASE-KEY}(S, x, k)}$ increases the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value.

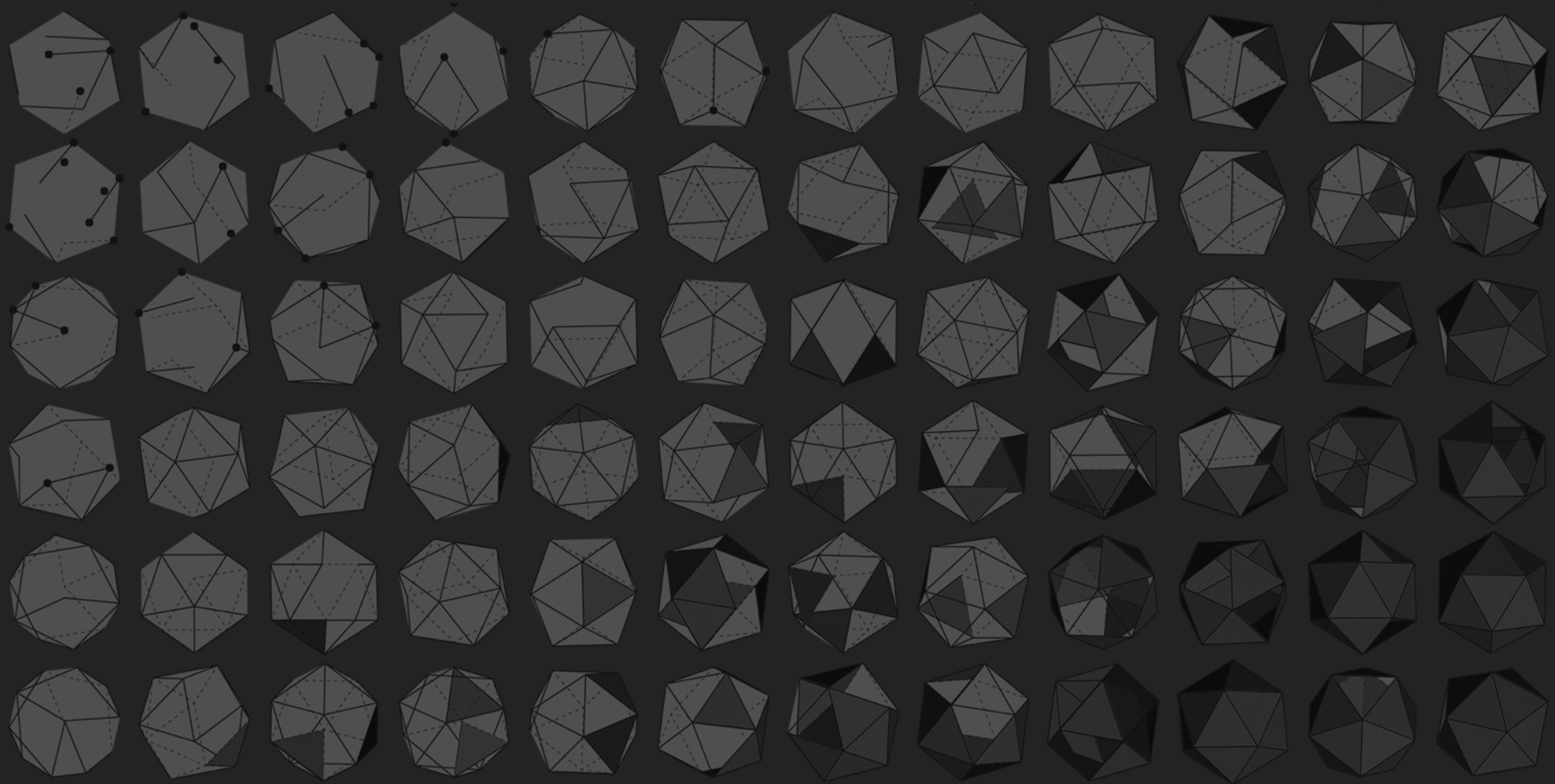
عملیات‌های روی priority queue

```
HEAP-MAXIMUM(A)
  return A[1]
```

```
HEAP-EXTRACT-MAX(A)
  if A.heap-size < 1
      error "heap underflow"
  max = A[1]
  A[1] = A[A.heap-size]
  A.heap-size = A.heap-size - 1
  MAX-HEAPIFY(A, 1)
  return max
```

```
HEAP-INCREASE-KEY(A, i, key)
  if key < A[i]
      error "new key is smaller than current key"
  A[i] = key
  while i > 1 and A[PARENT(i)] < A[i]
      exchange A[i] with A[PARENT(i)]
      i = PARENT(i)
```

```
MAX-HEAP-INSERT(A, key)
  A.heap-size = A.heap-size + 1
  A[A.heap-size] = -∞
  HEAP-INCREASE-KEY(A, A.heap-size, key)
```



دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

INTRODUCTION TO ALGORITHM | درس طراحی الگوریتم (ترم اول ۱۳۹۹)

صف اولویت و عملیات‌های آن

صف اولویت یا priority queue

A ***priority queue*** is a data structure for maintaining a set S of elements, each with an associated value called a ***key***. A ***max-priority queue*** supports the following operations:

$\boxed{\text{INSERT}(S, x)}$ inserts the element x into the set S, which is equivalent to the operation $S = S \cup \{x\}$.

$\boxed{\text{MAXIMUM}(S)}$ returns the element of S with the largest key.

$\boxed{\text{EXTRACT-MAX}(S)}$ removes and returns the element of S with the largest key.

$\boxed{\text{INCREASE-KEY}(S, x, k)}$ increases the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value.

عملیات‌های روی priority queue

```
HEAP-MAXIMUM(A)
return A[1]
```

```
HEAP-EXTRACT-MAX(A)
if A.heap-size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size − 1
MAX-HEAPIFY(A, 1)
return max
```

```
HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
    error "new key is smaller than current key"
A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```

```
MAX-HEAP-INSERT(A, key)
A.heap-size = A.heap-size + 1
A[A.heap-size] = −∞
HEAP-INCREASE-KEY(A, A.heap-size, key)
```