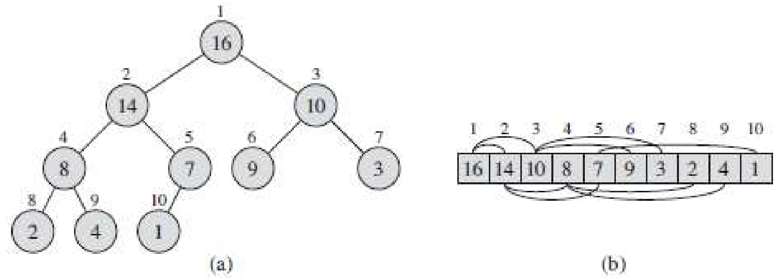
Heaps

Sorting problem:
Input: A sequence of n numbers <a1, a2, …, an>.
Output: A permutation (reordering) <a1’, a2’, …, an’> of the input sequence such a1’<= a2’<= …<= an’

The (binary) heap data structure is an array object that we can view as a nearly complete binary tree as shown below.



0 <= A.heap-size <= A.length; A[1] is the root

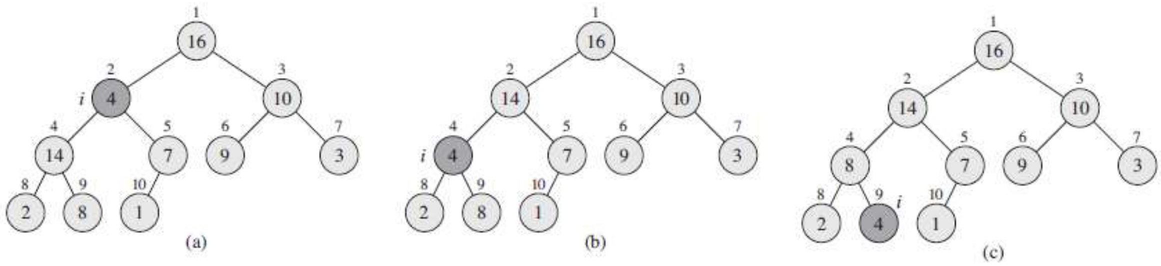
```
PARENT(i)
  return ⌊i/2⌋

LEFT(i)
  return 2i

RIGHT(i)
  return 2i + 1
```

There are two types of heaps: max-heap and min-heaps
Max-heap has the following property: A[Parent(i)] >=A[i] => the largest element is stored at the root.
Max-heap has the following property: A[Parent(i)] <= A[i] => the smallest element is stored at the root.
Height of a node = number of edges from the node to a leaf. Heigh of a heap = $\Theta(\lg n)$.

Q1: What are the minimum and maximum numbers of elements in a heap of height h?
Q2: Show that, with the array representation for storing an n-element heap, the leaves are the nodes indexed by floor(n/2)+1, floor(n/2)+2, ..., n.

Maintaining the heap property

```
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)
```



$T(n) \le T(2n/3) + \Theta(1)$. By the master method, T(n) =O(lgn)

Q: Why 2n/3 ?
Q: What is the effect of calling MAX-HEAPIFY(A,i) / for i > A.heap-size/2?

Building a heap

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

The running time of Build-Max-Heap is T(n)= n/2 * Max-Heapify = n/2 * O(lgn)= O(nlgn). But this is not asymptotically tight.

T(n) = O(n) // Each node does not need O(lgn) time which heap's height. Some nodes need 1,2, etc. In general, for a node with height *h* Max-Heapify needs O(h).

At most ceil(n/2^(h+1)) nodes of any height *h.*

Q: Why?

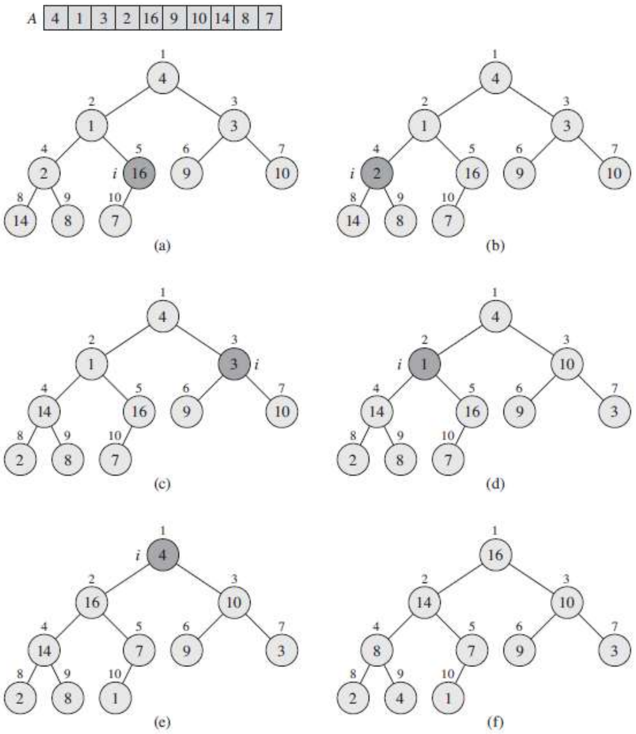
$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right)$$

$\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1}$	$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$	$\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}$ for $\lvert x \rvert < 1$.

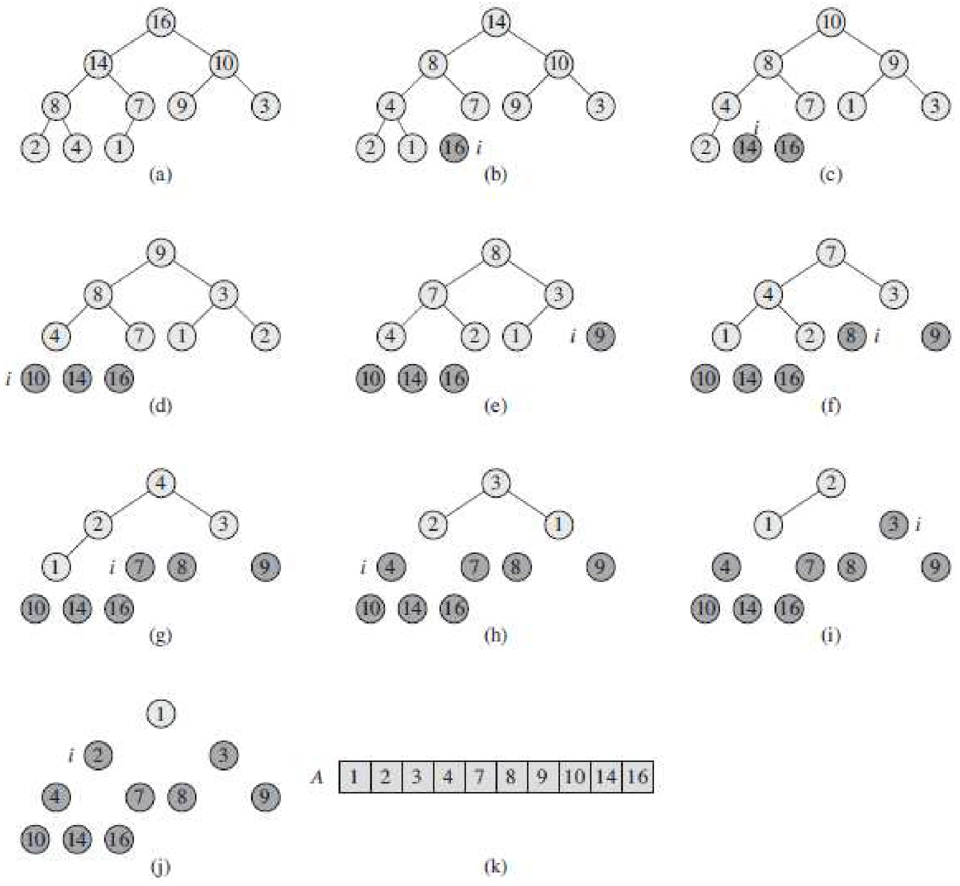
$$\begin{aligned}\sum_{h=0}^{\infty} \frac{h}{2^h} &= \frac{1/2}{(1-1/2)^2} \\ &= 2 .\end{aligned}$$

$$\begin{aligned}O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) \\ &= O(n) .\end{aligned}$$

Ex:



The heapsort algorithm



```
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)
```

Priority queues

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:

INSERT(S, x) inserts the element x into the set S, which is equivalent to the operation S U S {x}.

MAXIMUM(S) returns the element of S with the largest key.

EXTRACT-MAX(S) removes and returns the element of S with the largest key.

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.

```
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)
```