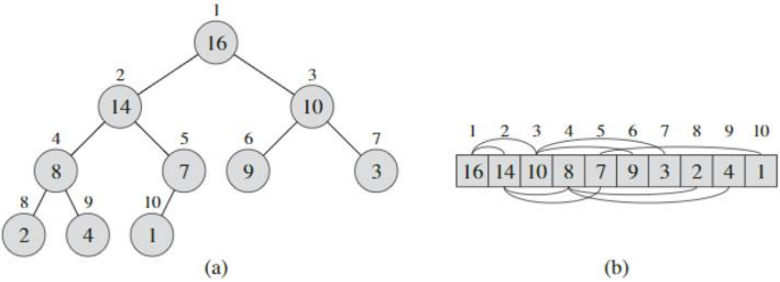
# Heap Sort

Sheng Li
229221

April 6, 2022

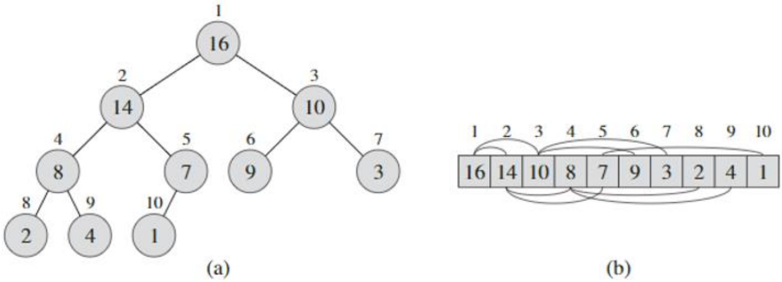
# Overview

- An array, visualized as a nearly complete binary tree.
- Lay out the nodes of the tree in breadth-first order.
- Max-Heap Property: The key of a node is $\geq$ than the keys of its children for any given node.




A max-heap viewed as (a) a binary tree and (b) an array

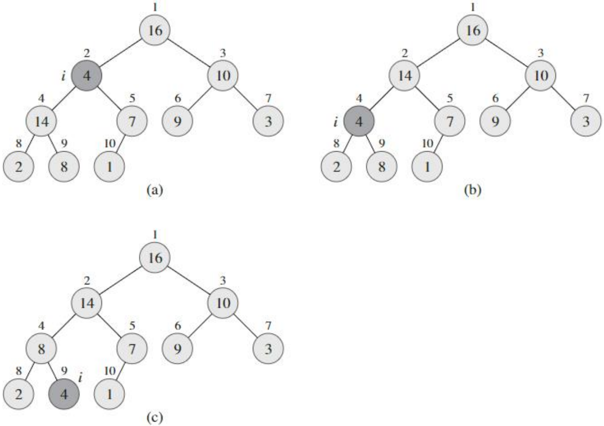
root of tree: first element in the array, corresponding to i = 1.
parent(i) = $\lfloor i/2 \rfloor$: returns index of node's parent.
left(i) = 2i: returns index of node's left child.
right(i)= 2i + 1: returns index of node's right child.
A.length: number of elements in the array A.
A.heap-size: elements of the heap stored in the array A.($\leq$ A.length)




A max-heap viewed as (a) a binary tree and (b) an array

*Max_Heapify*$(A, i)$: make the subtree rooted at index i a max-heap by assuming that the binary trees rooted at LEFT(i) and RIGHT(i) are max-heaps.
*Build_Max_Heap*$(A)$: produce a max-heap from an unordered array A.

# *Max_Heapify*$(A, i)$(Example)




The action of *Max_Heapify*$(A, 2)$ where A.heap-size $=$ 10. (a) The initial configuration with 4 at node i$=$2. (b)Restore the max-heap for node 2 by exchanging A[2] with A[4] and recursively call *Max_Heapify*$(A, 4)$. (c)node 4 is fixed up, the recursive call *Max_Heapify*$(A, 9)$ yields no further change

# *Max_Heapify*$(A, i)$ Pseudocode

```
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)
```

# *Max_Heapify*$(A, i)$ Time Complexity Analysis

The running time of *Max_Heapify*$(A, i)$ is $\theta(1)$ time to compare, plus the time to run *Max_Heapify* on a subtree (assuming that the recursive call occurs).

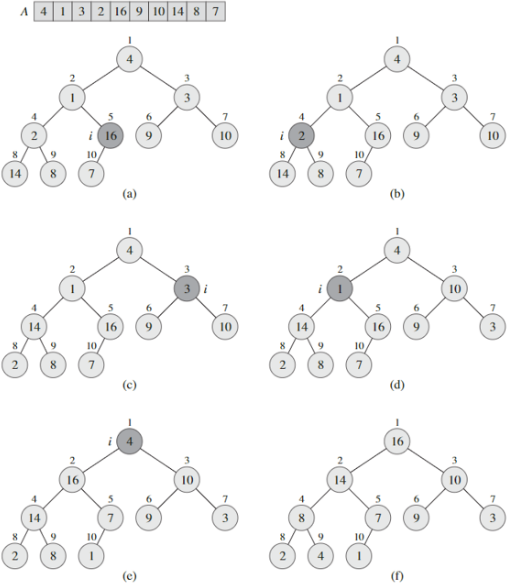
Worst case: occurs when the bottom level of the tree is exactly half full (The subtrees of children each have size at most 2n/3)

Running time of *Max_Heapify* by the recurrence:

$$T(n) \leq T(2n/3) + \theta(1)$$

According to master theorem, $T(n) = O(logn)$.

# $Build\_Max\_Heap(A)$ (Example & Pseudocode )




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

Observation:
1. *Max_Heapify* takes O(1) time for nodes that are 1 level above the leaves, and in general, O(k) for the nodes that are k levels above the leaves.
2. We have n/4 nodes with level 1, n/8 with level 2, and so on till we have the root node that is lg n levels above the leaves. Total amount of work in the for loop can be summed as:

$$T(n) = n/4(1c) + n/8(2c) + \cdots + 1(lgnc) = cn/2(1/2^1 + 2/2^2 + \cdots + (k+1)/2^{k+1})$$

$$\sum_{h=1}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

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

# HeapSort Pseudocode

```
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)
```

Running time:
The call to *Build_Max_Heap*$(A)$ takes time O(n).
After (n-1) iterations the Heap is empty. Every iteration involves a swap and a *Max_Heapify*$(A, i)$ operation which takes O(log n) time.
Overall, the HeapSort procedure takes time $O(nlogn)$.