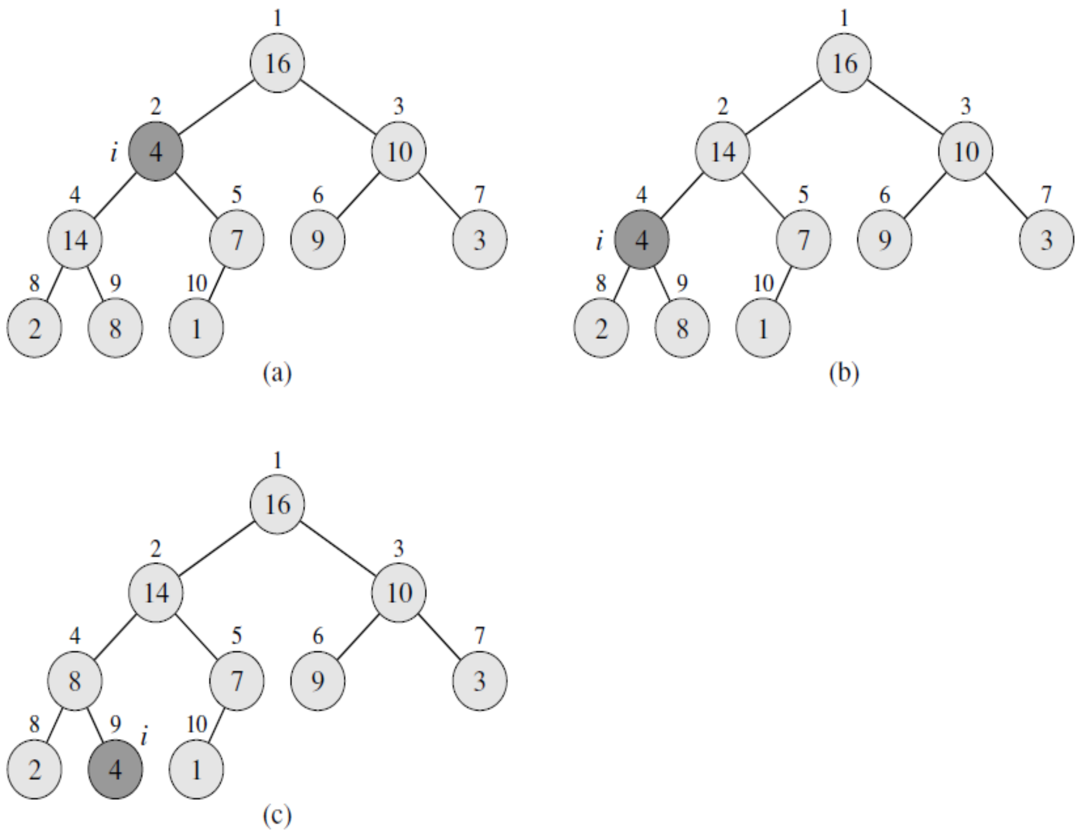
Maintaining the heap property

- MAX-HEAPIFY is important procedure for manipulating max-heaps. It is used to maintain the max-heap property.
  Inputs: an array A and an index i into the array.
- When MAX-HEAPIFY is called, it is assumed that the binary trees rooted at LEFT(i) and RIGHT(i) are max-heaps, but that A[i] may be smaller than its children, thus violating the max-heap property.
  Output: After MAX-HEAPIFY, subtree rooted at i is a max-heap.

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

Following diagram illustrates the action of MAX-HEAPIFY.

At each step, the largest of the elements A[i], A[LEFT(i)], and A[RIGHT(i)] is determined, and its index is stored in *largest*.



The action of MAX-HEAPIFY(A, 2), where heap-size[A] = 10.

(a) The initial configuration, with A[2] at node i = 2 violating the max-heap property since it is not larger than both children.
(b) by exchanging A[2] with A[4], which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY(A, 4) now has i = 4.
(c) After swapping A[4] with A[9], node 4 is fixed up, and the recursive call MAX-HEAPIFY(A, 9) yields no further change to the data structure.

Explanation

- Since the running time $T(n)$ is analysed by the number of elements in the tree n

- the worst case of Max-heapify occurs when we start at the root of a heap and recurse all the way to a leaf

⇒ the procedure makes a choice of recursing to its left subtree or right subtree

⇒ when the procedure recurses it cuts the work done by some fraction of the work it had before

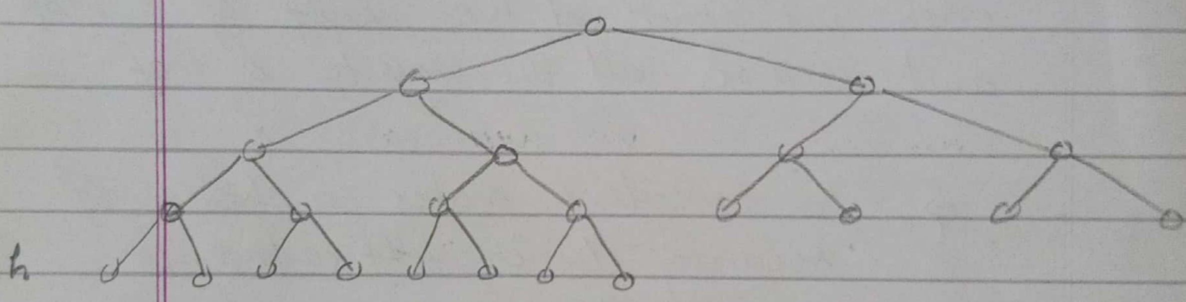
⇒ we have to show that this fraction is at most $2/3$.

- From the inspection we see that the worst ratio occurs when we are at the root ~~node of a heap of height h and~~ the left subtree is a complete binary tree of height $h-1$ and right subtree is a complete binary tree of height $(h-2)$.

Scanned with CamScanner

ok

we have to understand that the maximum number of elements in a subtree happens for the left subtree of a tree that has the last level half full.



h = height of heap

no of nodes in left subtree $= (2^h - 1)$

—— " —— right subtree $= (2^{h-1} - 1)$

total nodes $= 1 + (2^h - 1) + (2^{h-1} - 1)$

ratio of $\dfrac{\text{size of left subtree}}{\text{size of entire heap}} = \dfrac{2^h - 1}{1 + (2^h - 1) + (2^{h-1} - 1)}$

$$= \frac{2^h - 1}{2^h + 2^{h-1} - 1}$$

$$= \frac{2^h - 1}{2^{h-1}(2+1) - 1}$$

$$= \frac{2^h - 1}{3 \cdot 2^{h-1} - 1}$$

$$= \frac{2 \cdot 2^{h-1} - 1}{3 \cdot 2^{h-1} - 1}$$

Now for obtaining an upper bound on the ratio, we will take $h \to \infty$

Therefore, $\lim_{h \to \infty} \frac{2 \cdot 2^{h-1} - 1}{3 \cdot 2^{h-1} - 1}$

$$= \frac{2}{3}$$

so

$$T(n) = T\left(\frac{2}{3}n\right) + \Theta(1)$$

or

$$T(n) \leq T\left(\frac{2n}{3}\right) + \Theta(1)$$

so $T(n) = \Theta(\lg n)$

$$\left\{ \begin{array}{l} \text{Here } a = 1 \\ b = 3/2 \\ n^{\log_b a} = n^0 = 1 \end{array} \right\}$$

classmate
Date
Page

Q 6.2.3

No effect, just return

because the if statement on line 8 will be false

6.2.4

if $i > \frac{heap\text{-}size[A]}{2}$

then l and r both will exceed heap-size[A]

so the if statement conditions on lines 3 & 6 of the procedure will never be satisfied. Therefore largest = i, so the re

6.2.5

```
Max-heapify(A, i)
    while ( i < heapsize[A])
        l ← left(i)
        r ← right(i)
        largest ← i
        if l ≤ heap-size[A] and A[l] > A[i]
            then largest ← l
        if r ≤ heap-size[A] and A[r] > A[i]
            then largest ← r
        if largest ≠ i
            exchange A[i] ⟷ A[largest]
                i ← largest
        else return
```