Priority Queue

Add(x)

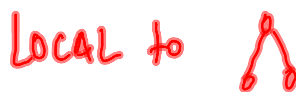
Delete_max()

Heap : implementation of priority queues

Binary tree that is

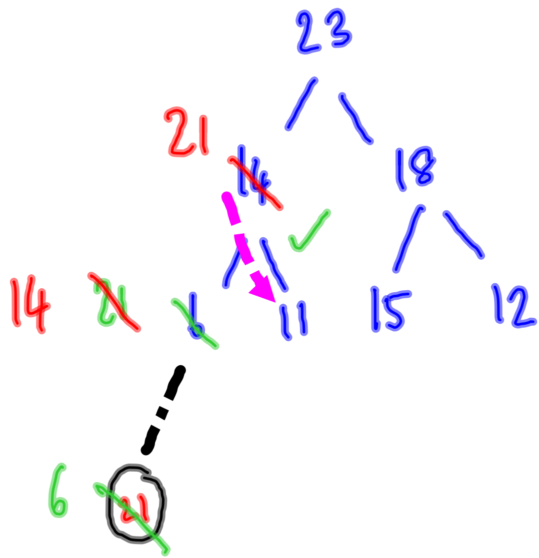
(a) Filled level by level, left to right

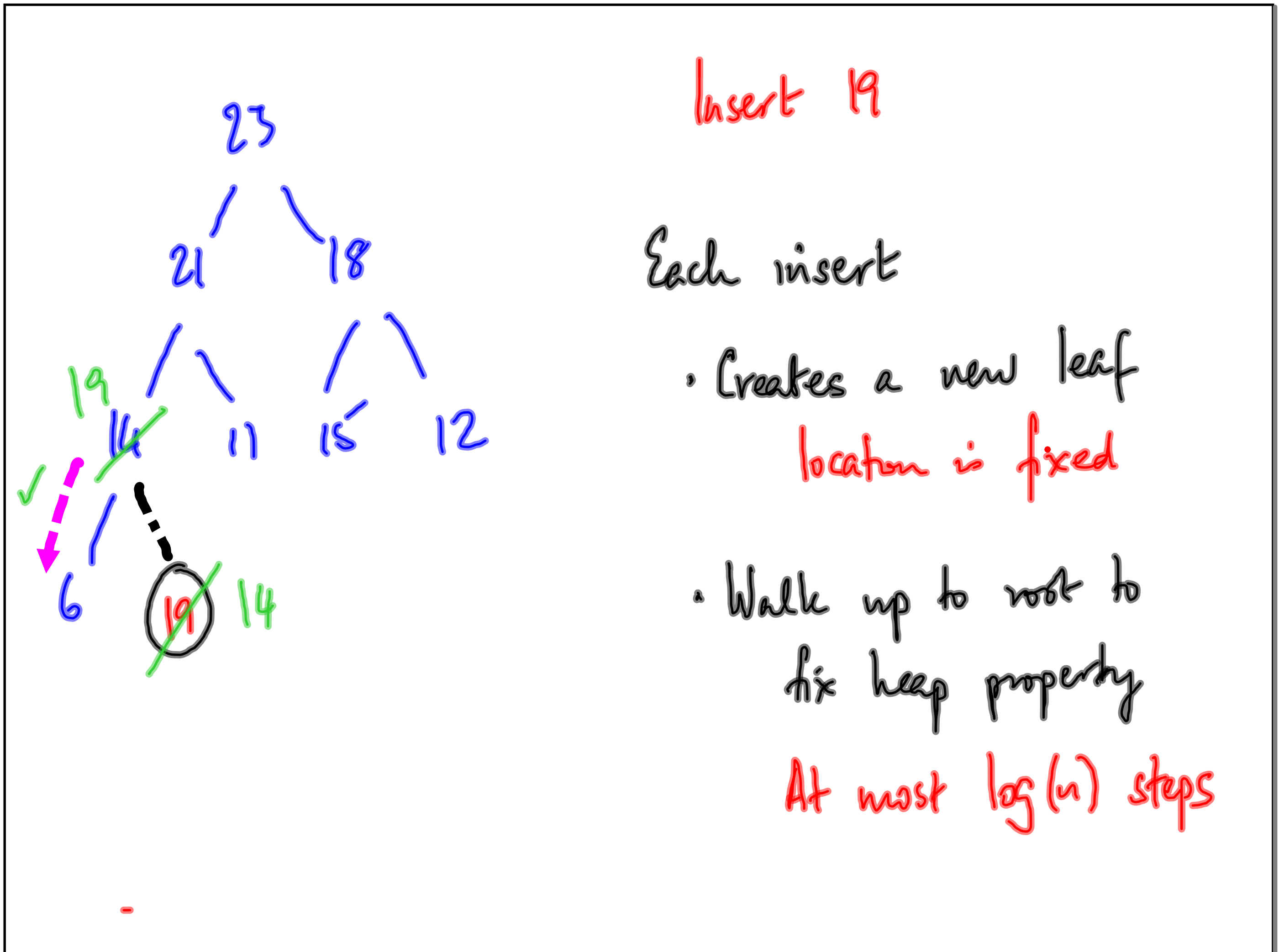
(b) For any node, value ≥ both children (if they exist)

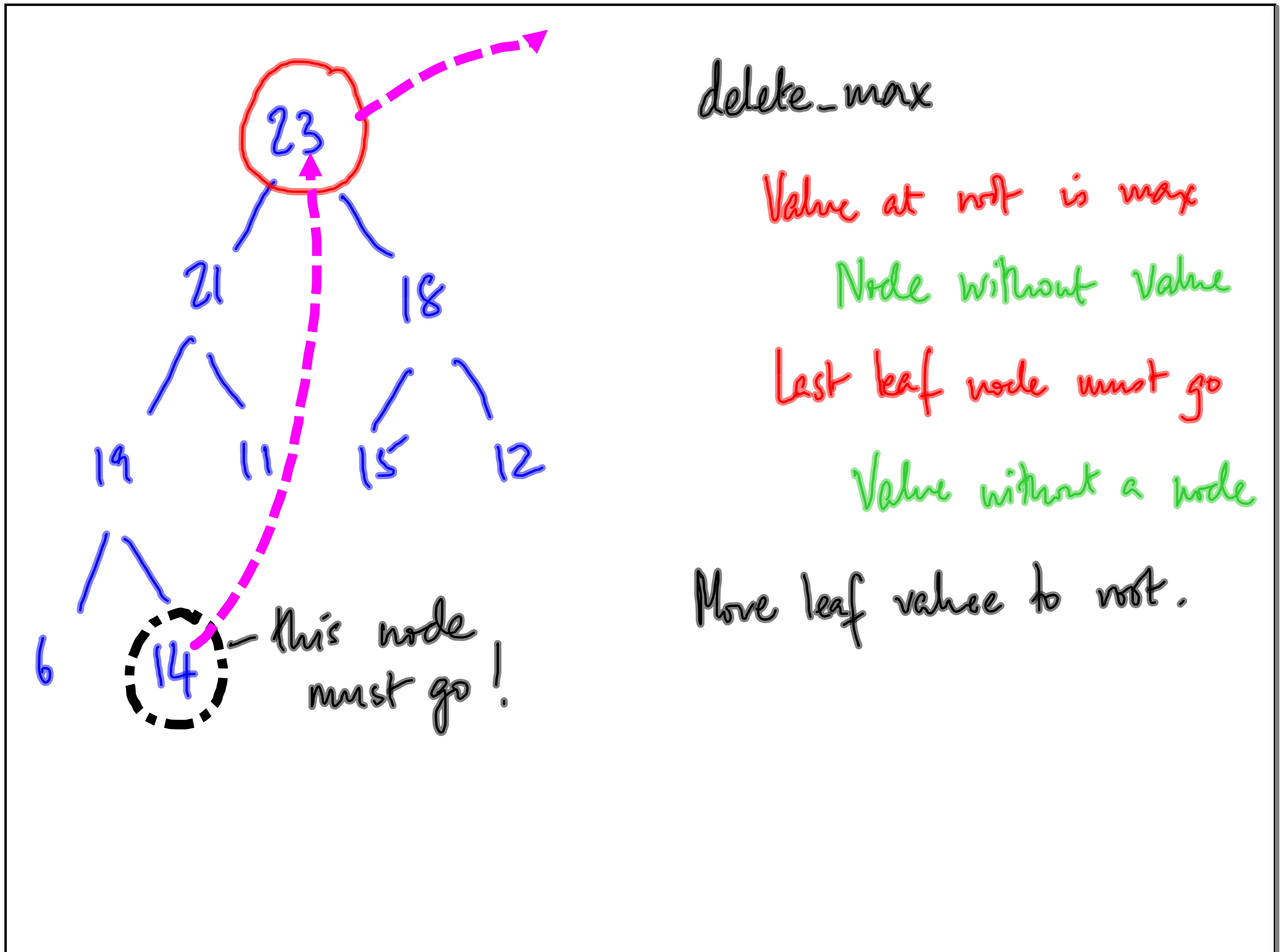
LOCAL to 

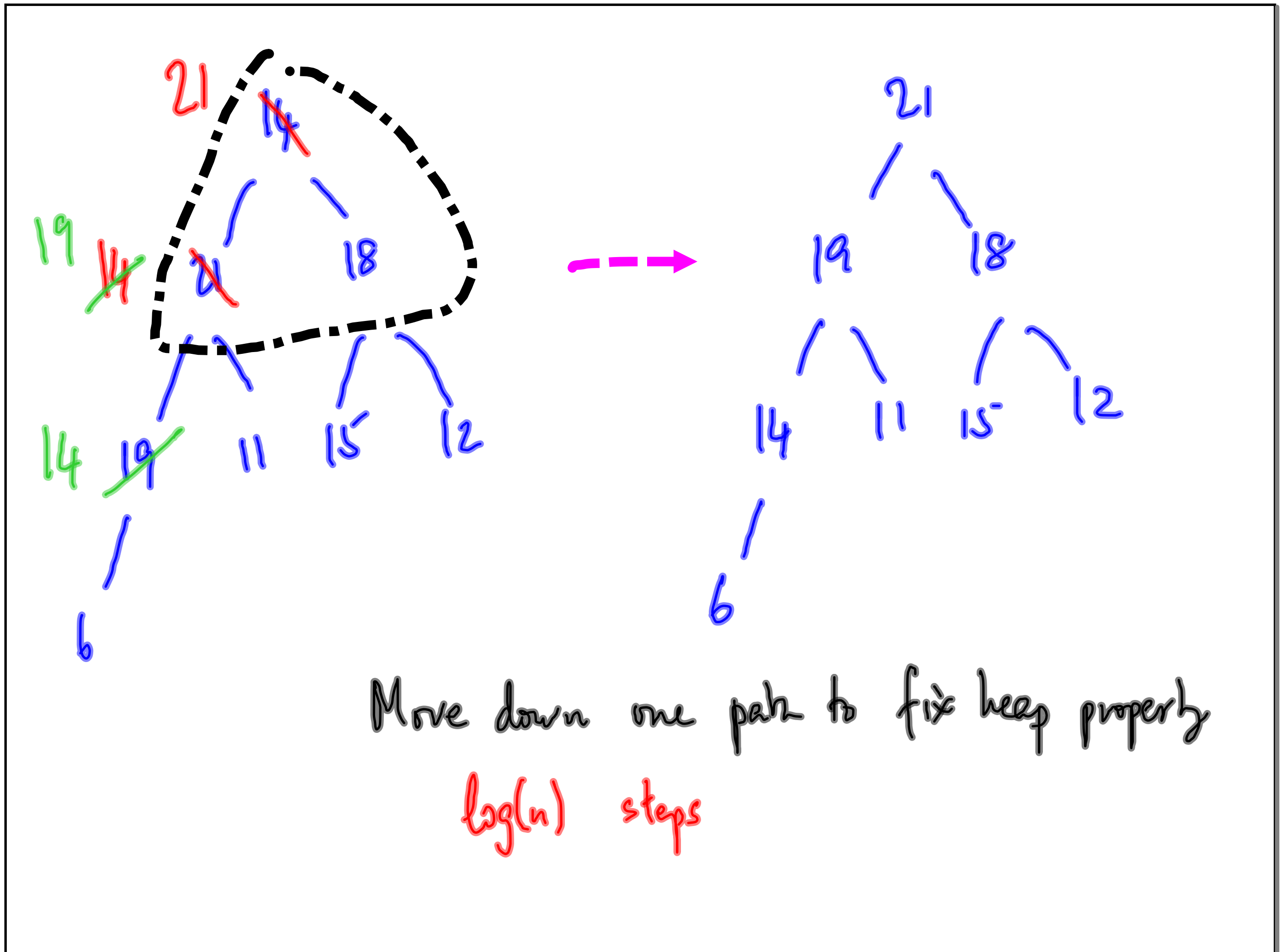
Insert into a heap:

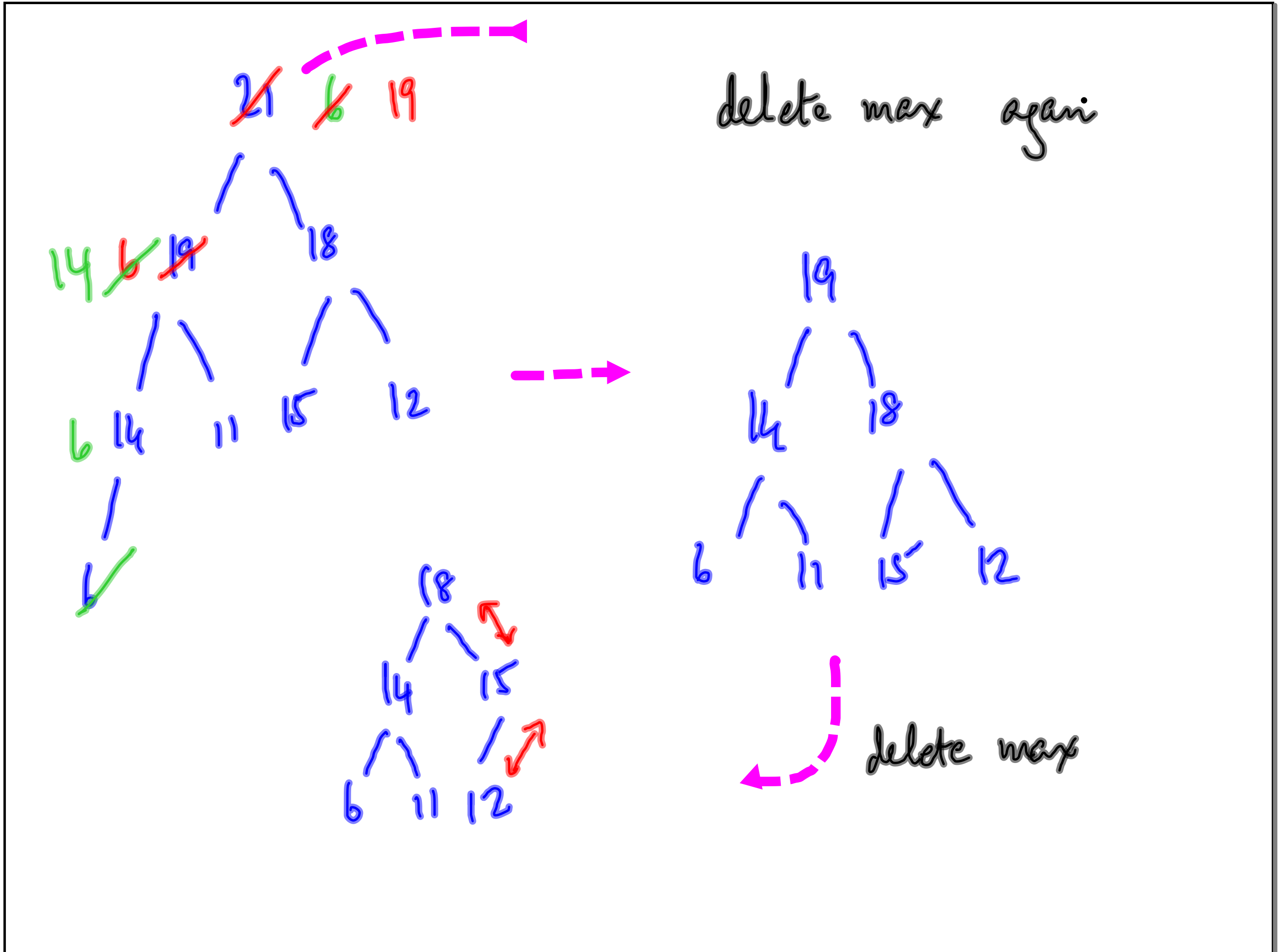
Insert 21




Insert 19
23
21
18
19
14
11
15
12
6
19
14
Each insert
• Creates a new leaf
location is fixed
• Walk up to root to
fix heap property
At most log(n) steps

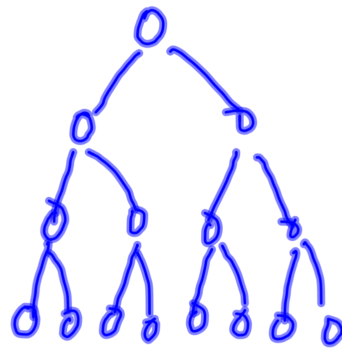

delete_max
Value at root is max
Node without value
Last leaf node must go
Value without a node
Move leaf value to root.
23
21
18
19
11
15
12
6
14
this node must go!


21
14
19
14
21
18
14
19
11
15
12
6
21
19
18
14
11
15
12
6
Move down one path to fix heap property
log(n) steps


delete max again
delete max

Why is the height of a heap $\log(n)$?

Complete binary tree



2^0 nodes

2^1 nodes

2^2 nodes

2^3 nodes

h levels: $\leq 2^h$ leaves

Total nodes $\leq 2^h + (2^h - 1)$

$2^{h+1} - 1$

3	2	1	0	
1	0	0	0	2^3
0	1	1	1	$2^3 - 1$

h levels $\leq 2^{h+1}-1$ nodes

Add a level.

Every leaf node adds upto 2 children

h+1 levels $2^{h+1}-1 + \underbrace{2(2^h)}_{\text{new leaves}}$

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

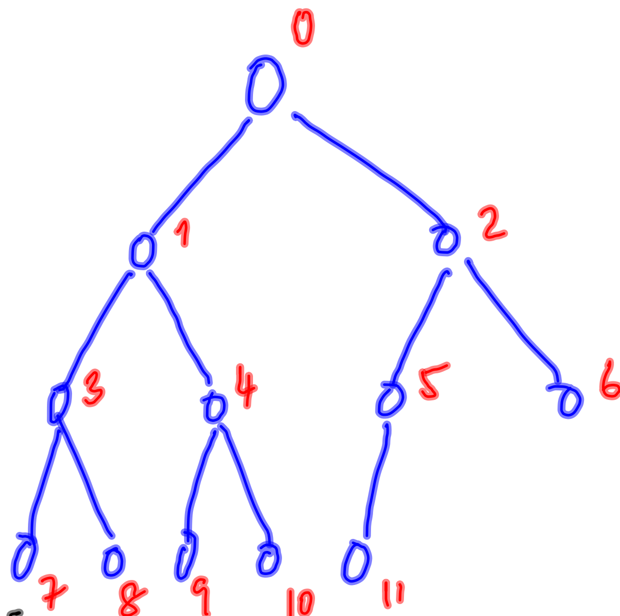
How to implement this?

Indexing of elements

n elements

list of length n

h[0], h[1], .., h[n-1]



Children of h[i] are at h[2i+1], h[2i+2]

Parent of h[j] is at h[(j-1)//2]

Swap h[i] with its left child

(h[i], h[2i+1]) = (h[2i+1], h[i])

Typically

We start with an empty heap

Each add(n) inserts an element

Each delete_max() removes a value

Over a course of n add + delete_max ops

$O(n \log(n))$ work

Symmetric treatment for delete_min()

Adjust second heap property

Value $\leq$ both children

Smallest value rises to root

Max heap vs Min heap

delete_max delete_min

Sorting with a heap

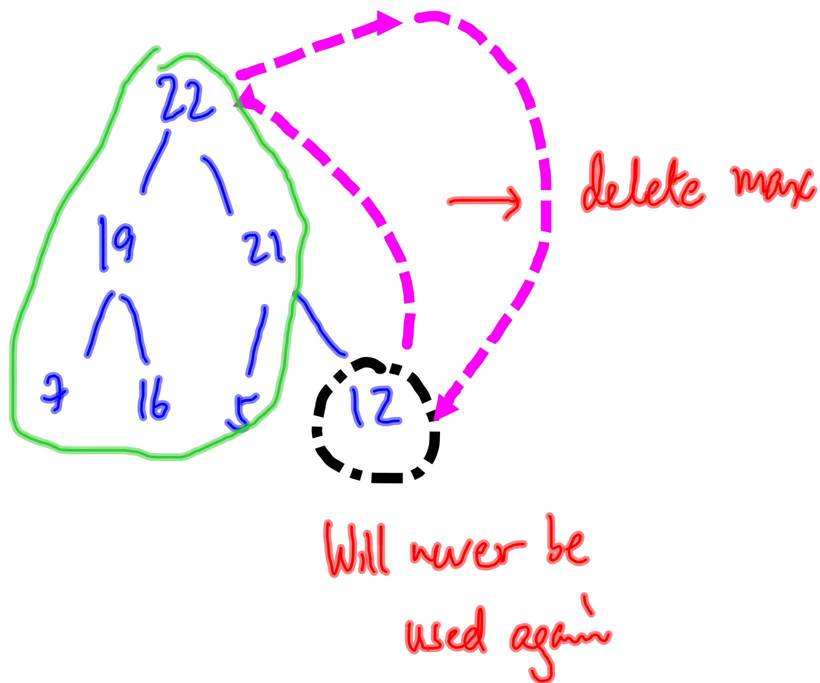
List of n elements

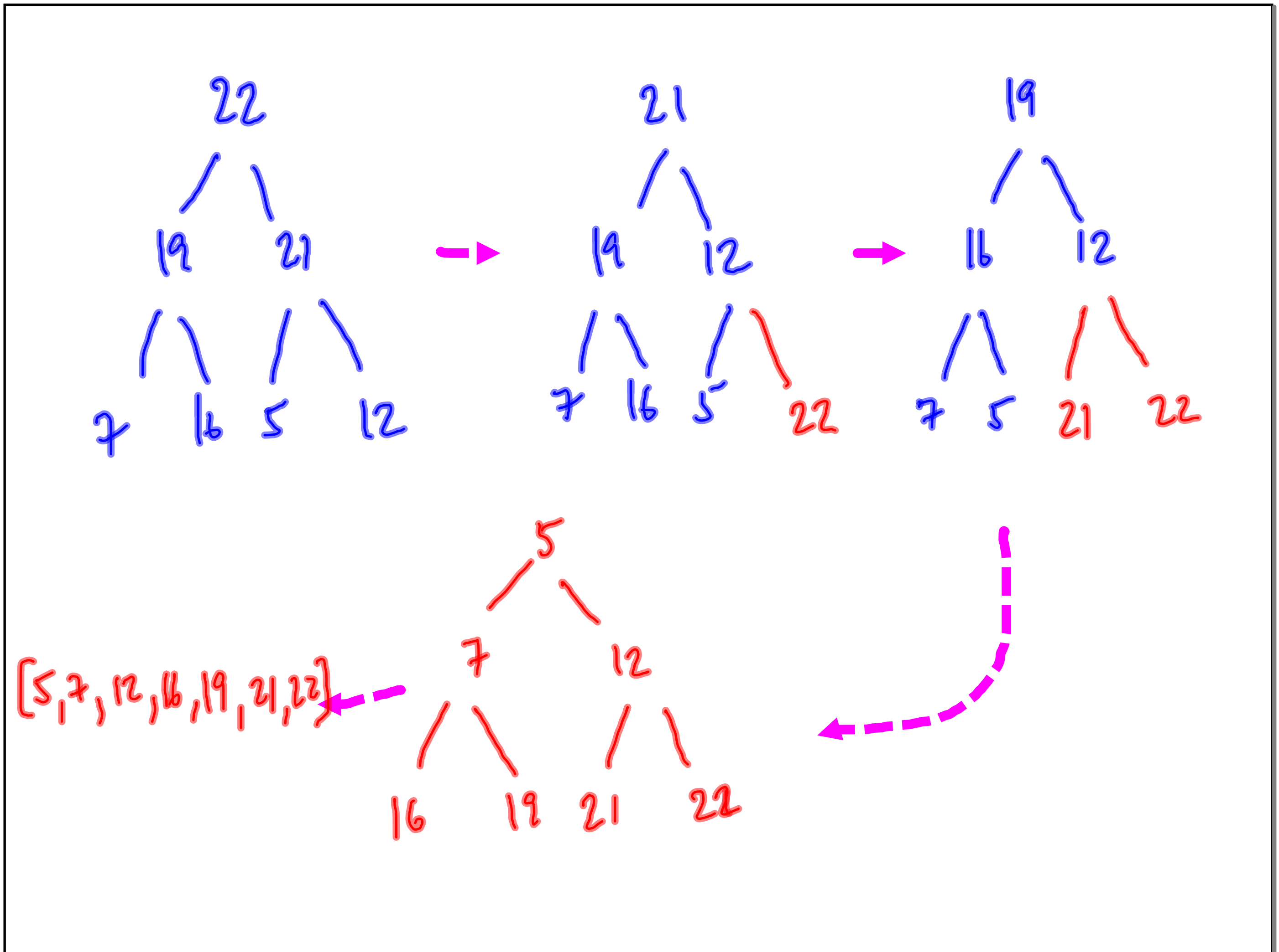
$\downarrow$ $n \log(n)$

Heap of n elements

$\downarrow$ $n \log(n)$

Extract in descending order


Can we avoid having 2 copies of the list?
22
19
21
7
16
5
12
delete max
Will never be used again


22
19
21
7
16
5
12
21
19
12
7
16
5
22
19
16
12
7
5
21
22
5
7
12
16
19
21
22
[5, 7, 12, 16, 19, 21, 22]

"Heapify"ing a list

1. Insert n elements into a heap, one at a time

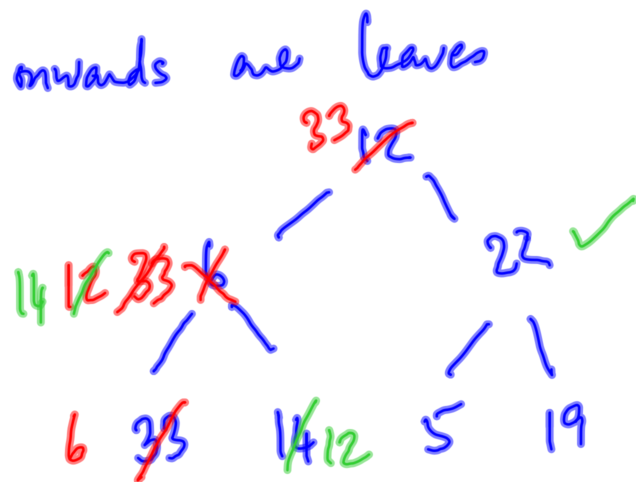
2. If full list is available:

   Positions n/2 onwards are leaves

0 1 2 3 4 5 6

[12, 6, 22, 33, 14, 5, 19]

Takes O(n) time



Names & values

```
def f(l):
    l ...
```

f(m) ← here l is not a "visible" name

What if we want f() to directly modify a name outside f?

"global" names ("global" variables)

```
def f():
    global m
    :
```

Any reference to m inside f affects the name m outside f

```
def f():
    y = x
    y = 2*y
    print(y)

x = 7
f()
print(x)
```

If x is not reset in f, pick up global value

global x

also local x, undefined

```
y = x
x = 2*y
print(x)
```

x becomes local

Can define functions inside fns

```
def f():
    def g():
    def h():
```

functions used within f

≡ Can only call f(), not g(), h()