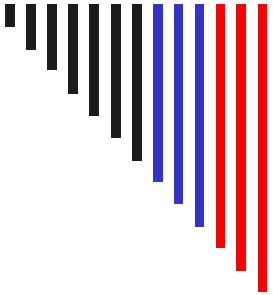


Heap

Objectives

- Define and implement heap structures
- Understand the operation and use of the heap ADT

9-1 Basic Concepts

•A heap is a binary tree whose left and right subtrees have values less than their parents. The root of a heap is guaranteed to hold the largest node in the tree.

•Both the left and the right branches of the tree have the same properties.

•Heaps are often implemented in an array rather than a linked list. When arrays are used, we are able to calculate the location of the left and the right subtrees. Conversely, we can calculate the address of it's parent.

•The tree is complete or nearly complete. The key value of each node is greater than or equal to the key value in each of its descendents. This structure is also called max- heap.

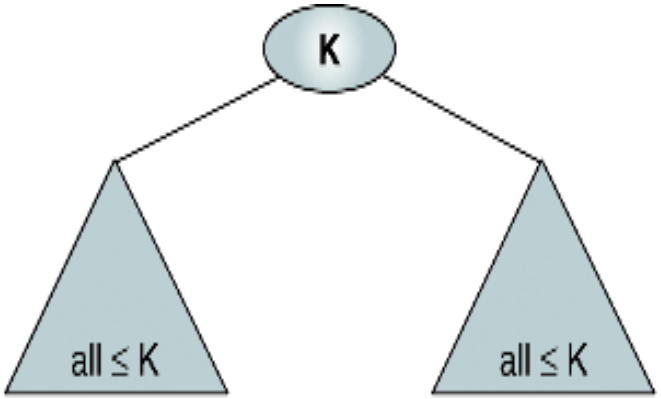
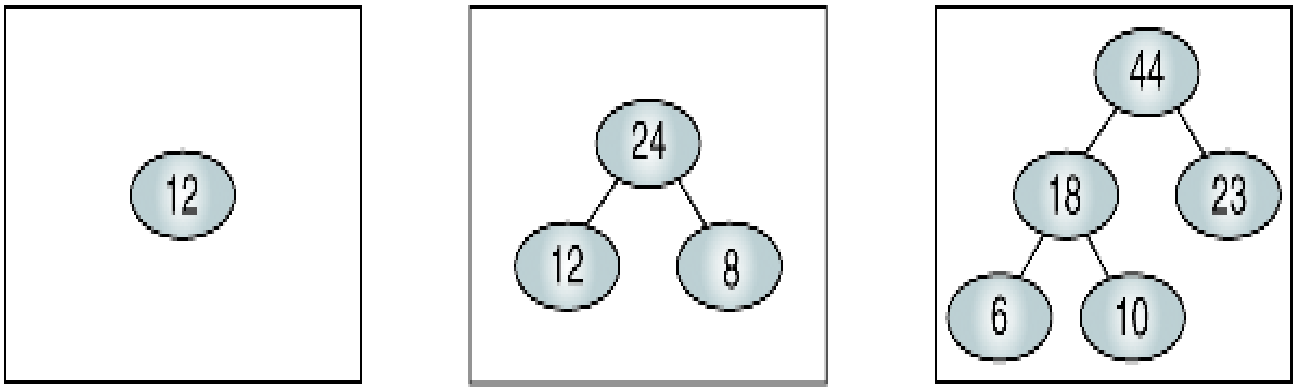


FIGURE 9-1 Heap



(a) Root-only heap (b) Two-level heap (c) Three-level heap

FIGURE 9-2 Heap Trees

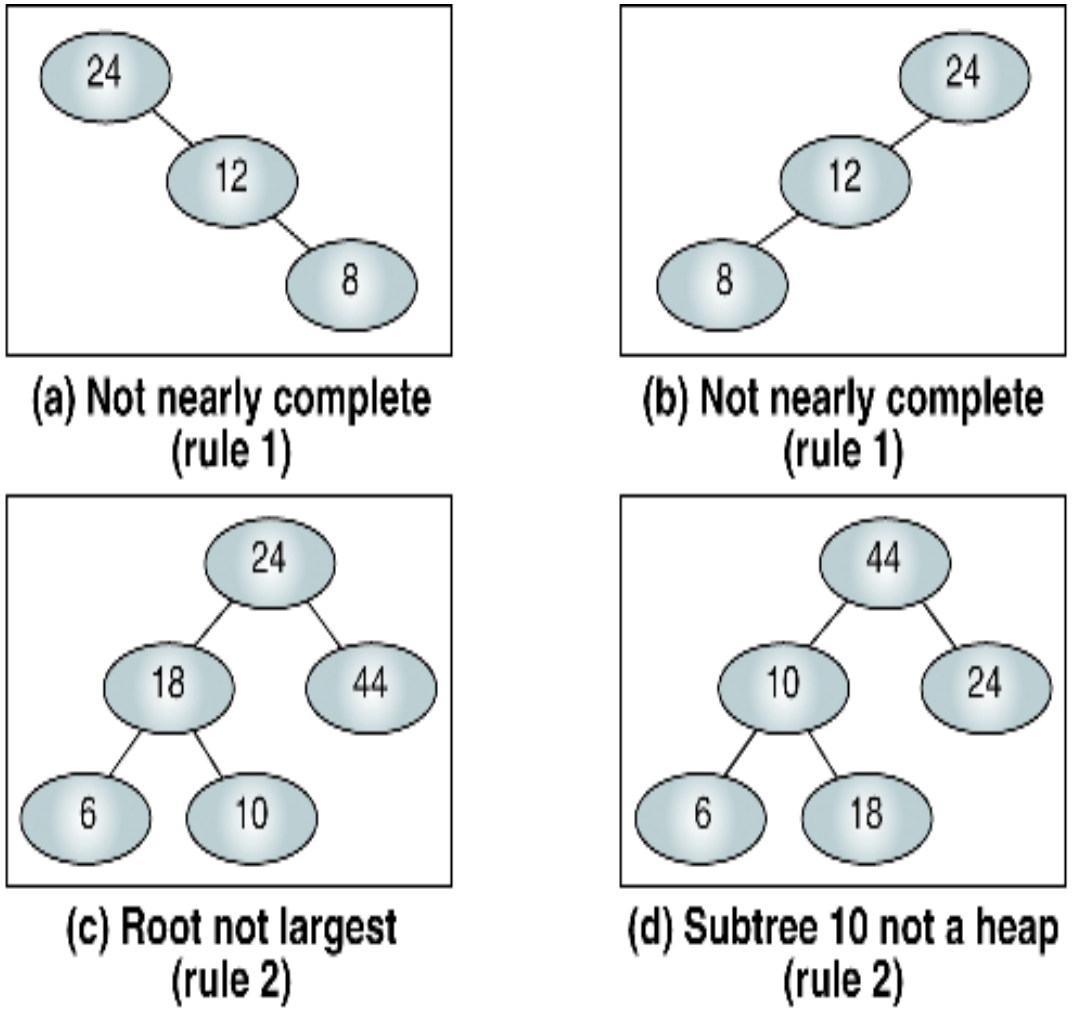


FIGURE 9-3 Invalid Heaps

Maintenance Operations

Two basic maintenance operations are performed on a heap.

- **Insert a heap**
- **Delete a heap**
- **Two basic algorithms are – Reheap Up and Reheap Down**

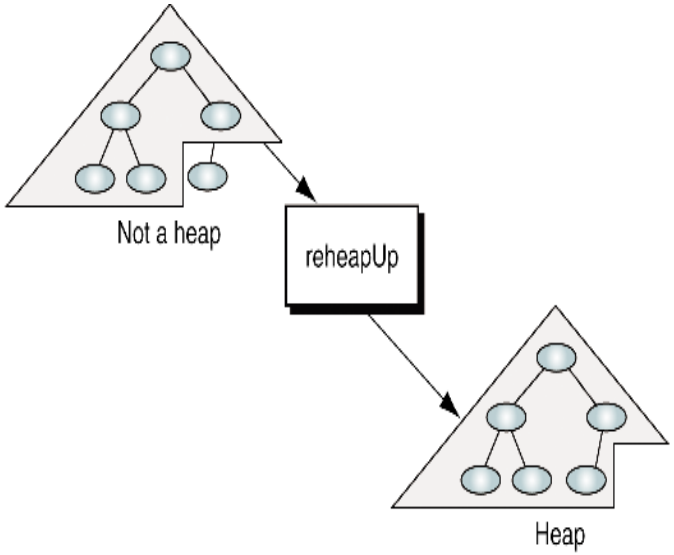
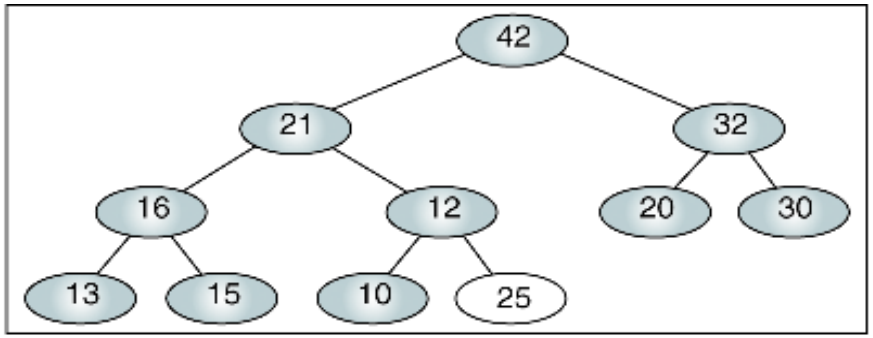
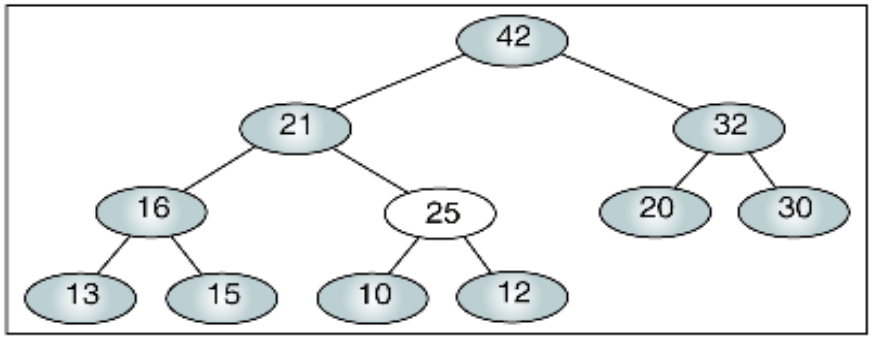


FIGURE 9-4 Reheap Up Operation

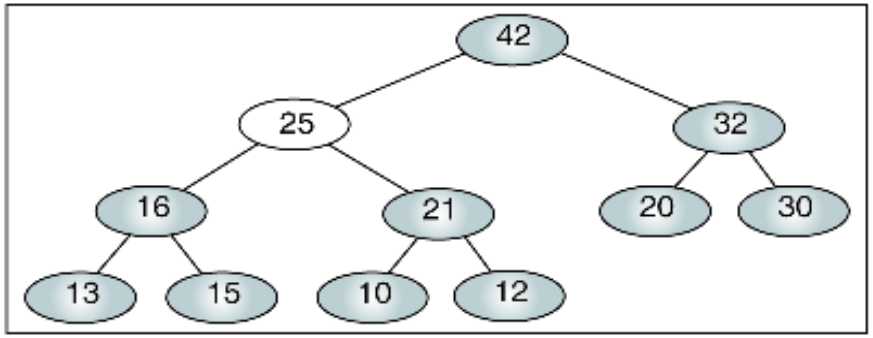
- **The reheap up operation repairs the structure so that it is a heap by floating the last element up the tree until that element is in its correct location.**
- **Insertion takes place at a leaf at the first empty position. This may create a situation where the new node's key is larger than that of its parent. If it is, the node is floated up the tree by exchanging the child and parent keys and data.**



(a) Original tree: not a heap



(b) Last element (25) moved up



(c) Moved up again: tree is a heap

FIGURE 9-5 Reheap Up Example

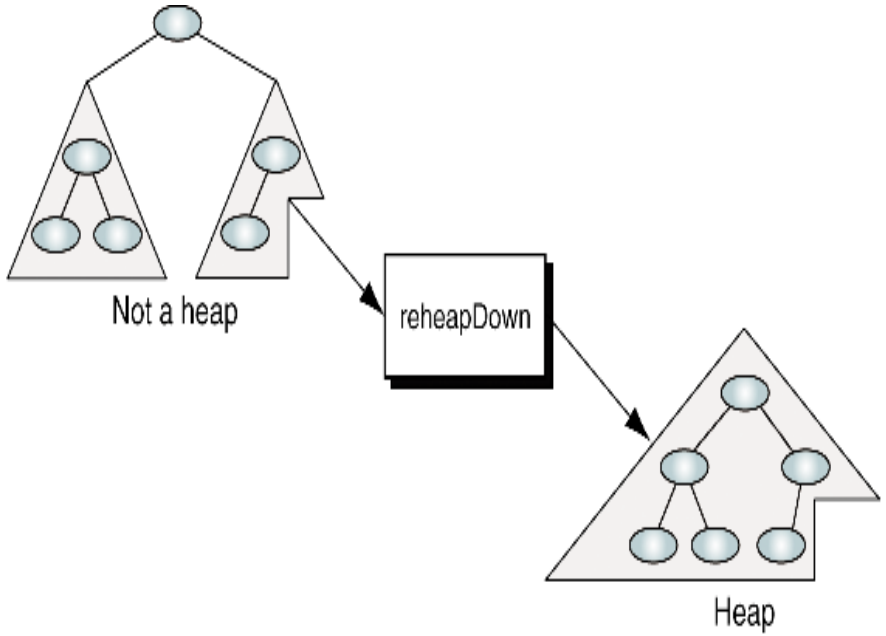
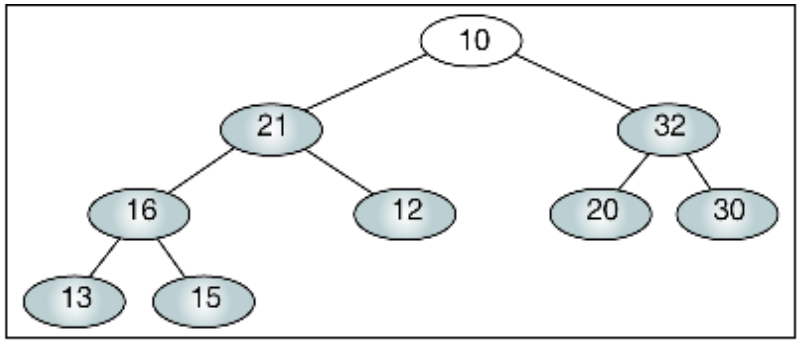
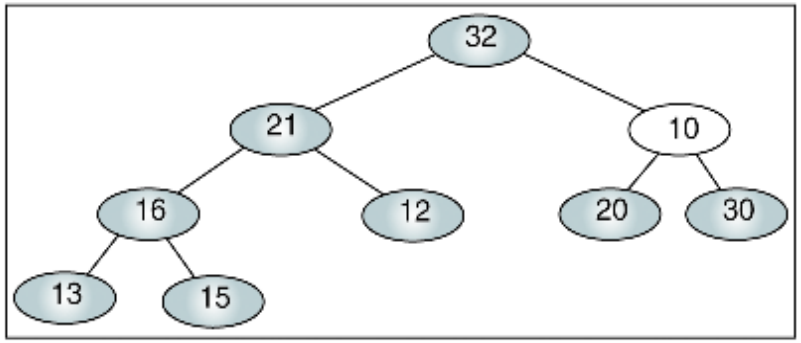


FIGURE 9-6 Reheap Down Operation

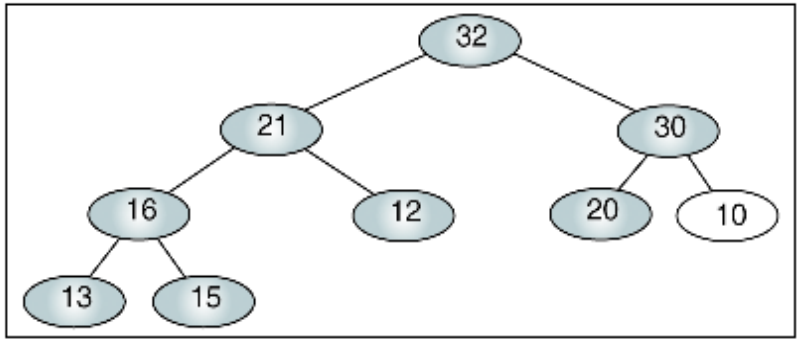
When we have a nearly complete binary tree that satisfies heap order property except in the root position. Reheap down operation reorders a broken heap by pushing the root down the tree until it is in correct position at the heap.



(a) Original tree: not a heap



(b) Root moved down (right)



(c) Moved down again: tree is a heap

FIGURE 9-7 Reheap Down Example

Implementatation

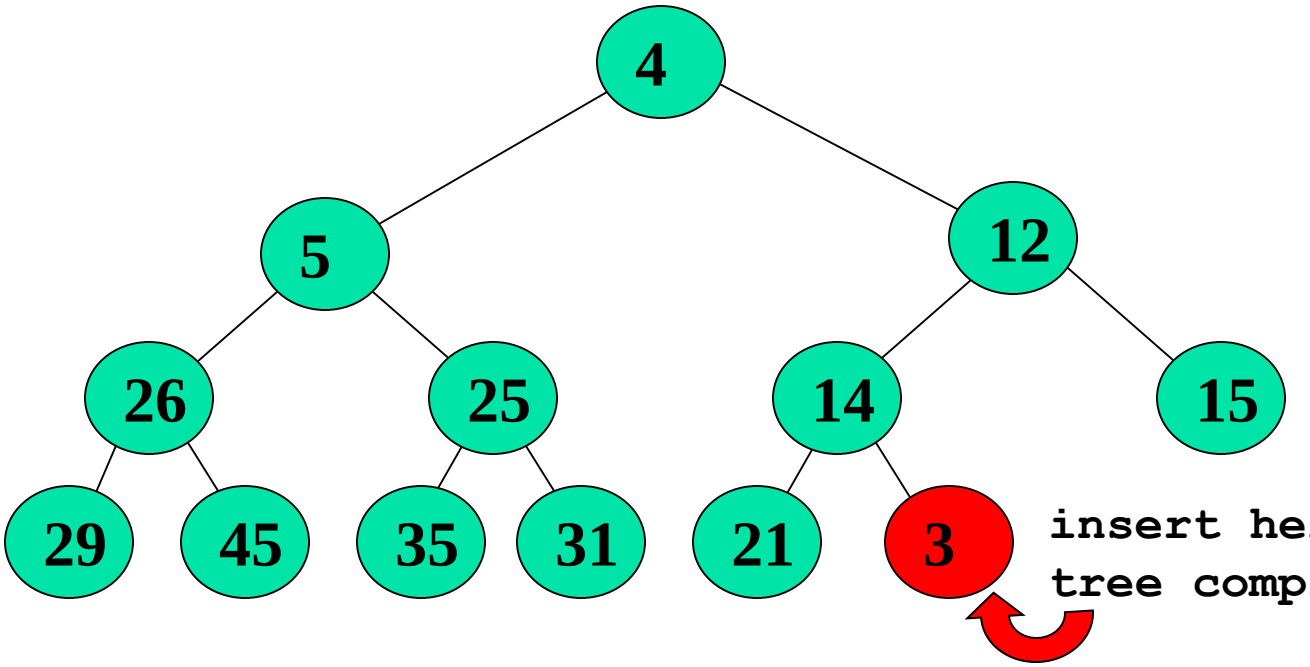
- Heap must be a *complete* tree

• all leaves are on the lowest two levels

• nodes are added on the lowest level, from left to right

• nodes are removed from the lowest level, from right to left

Inserting a Value

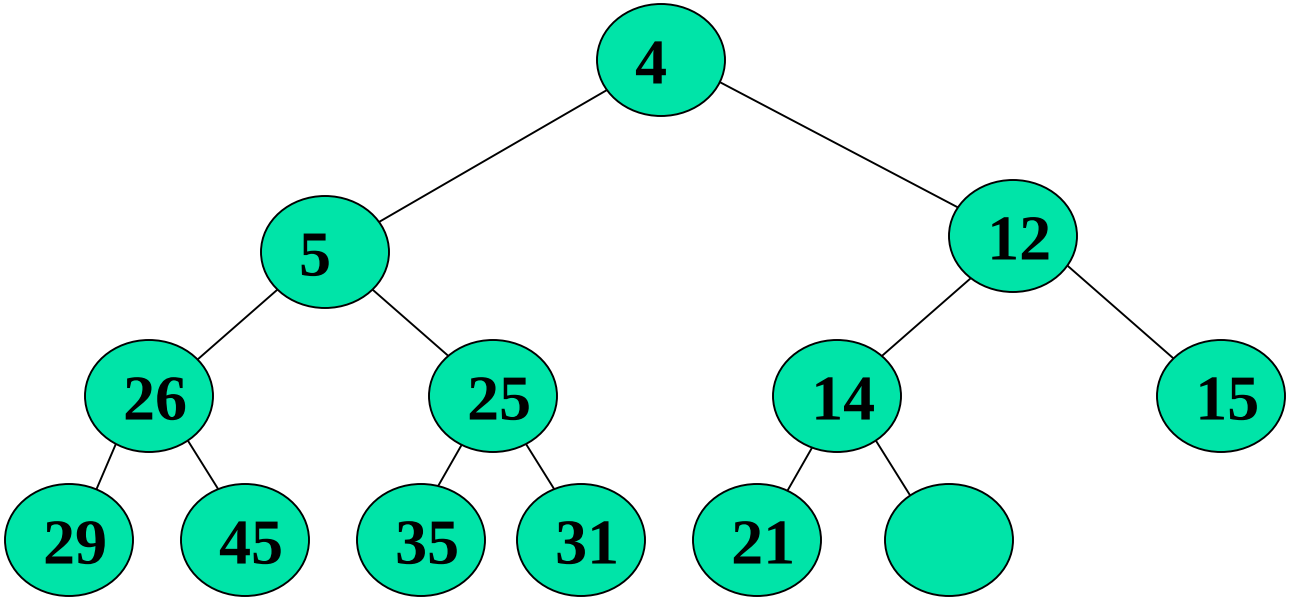


i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
array	__	4	5	12	26	25	14	15	29	45	35	31	21	3	__	__

currentsize = 13

Insert 3

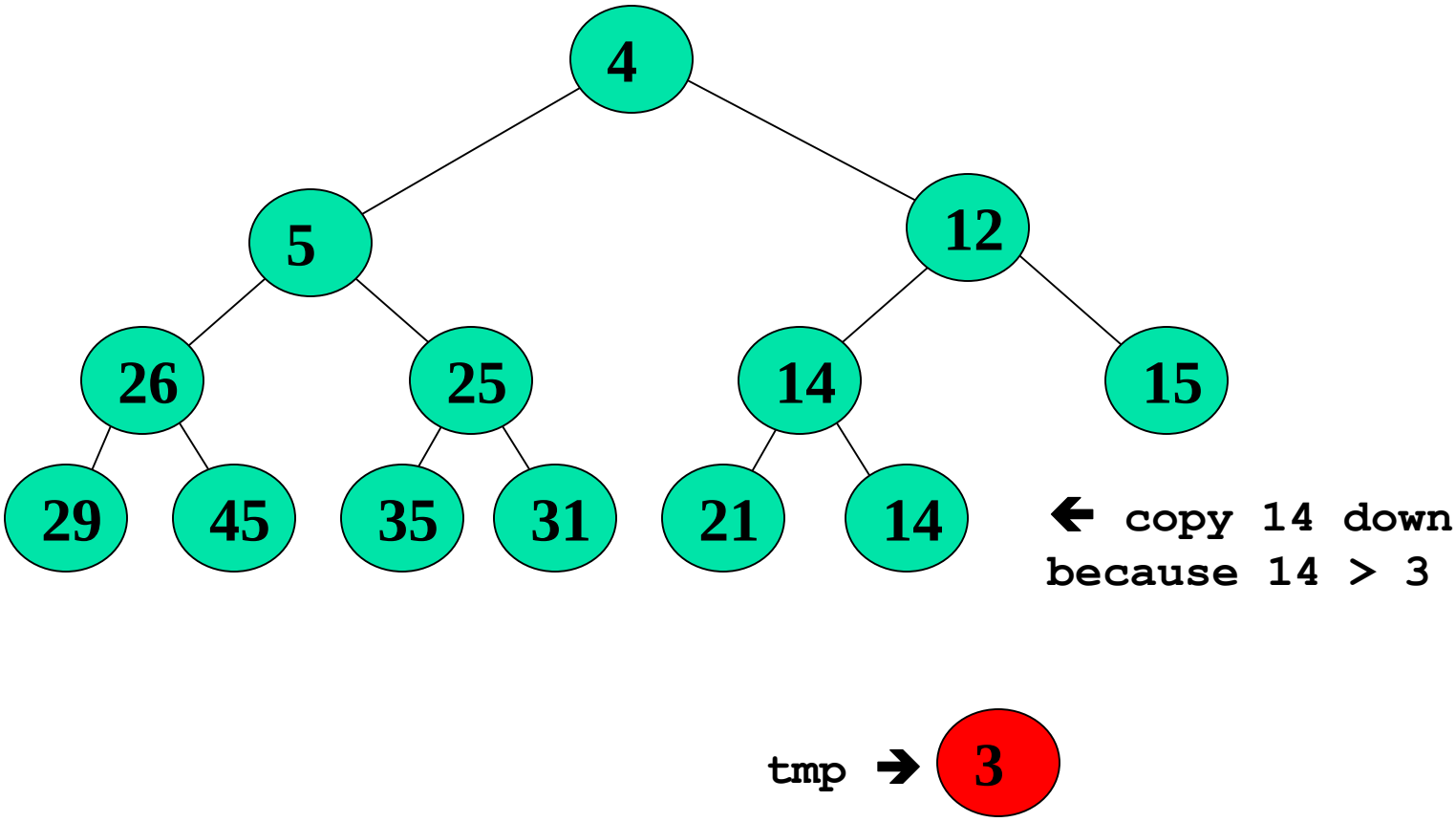
Inserting a Value




```
save new value in a
temporary location: tmp ➔
```

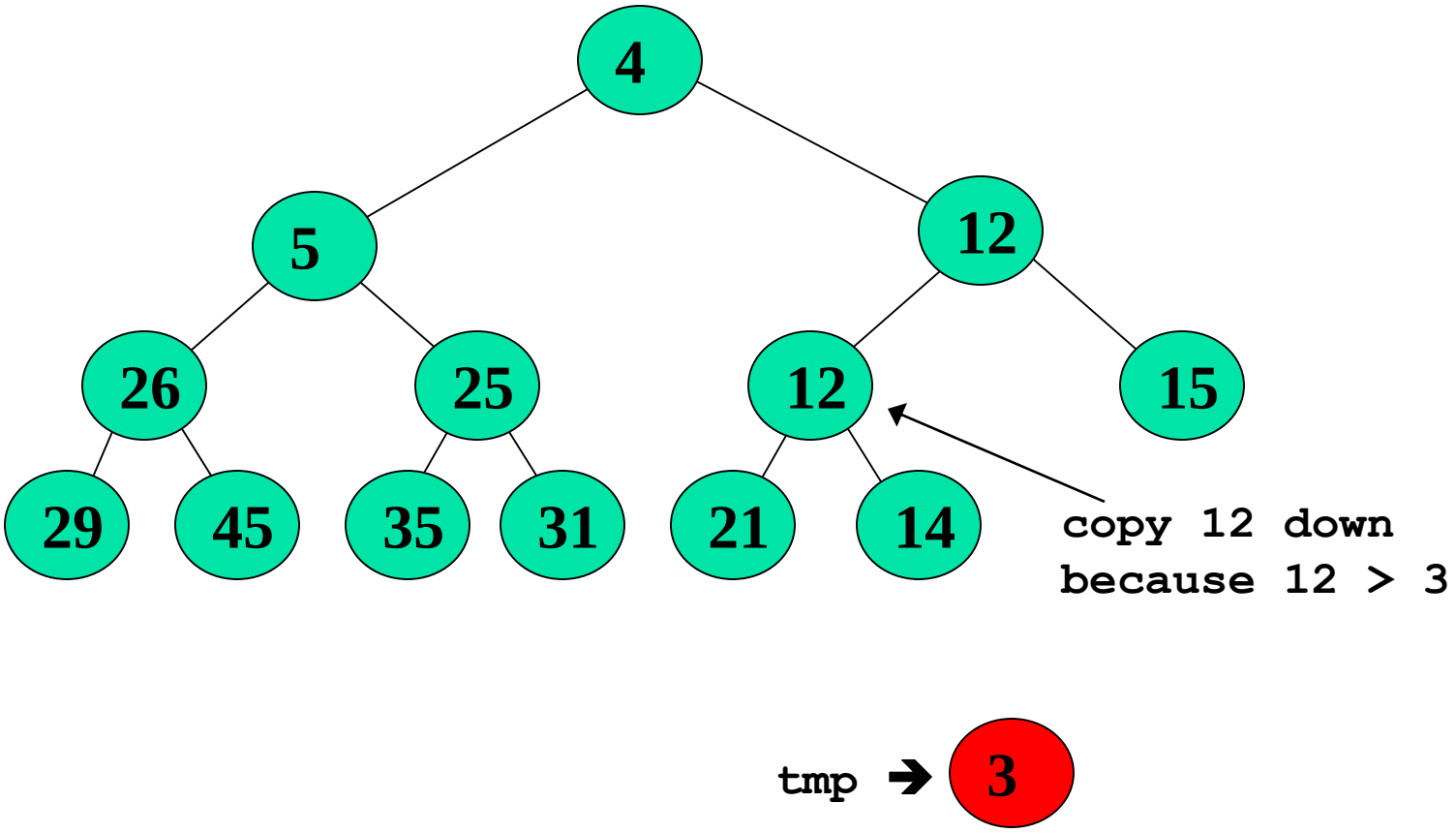
Insert 3

Inserting a Value



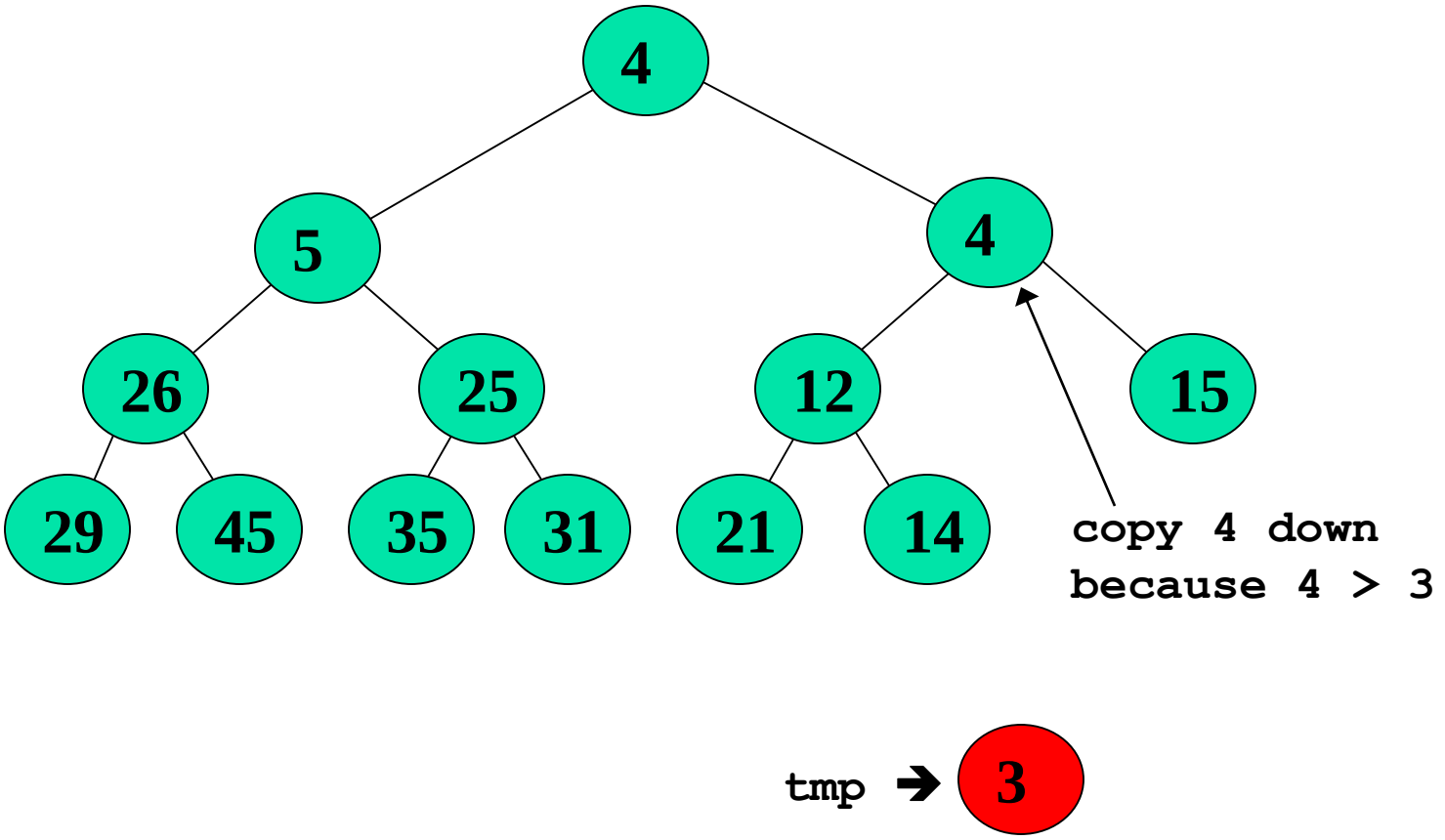
Insert 3

Inserting a Value



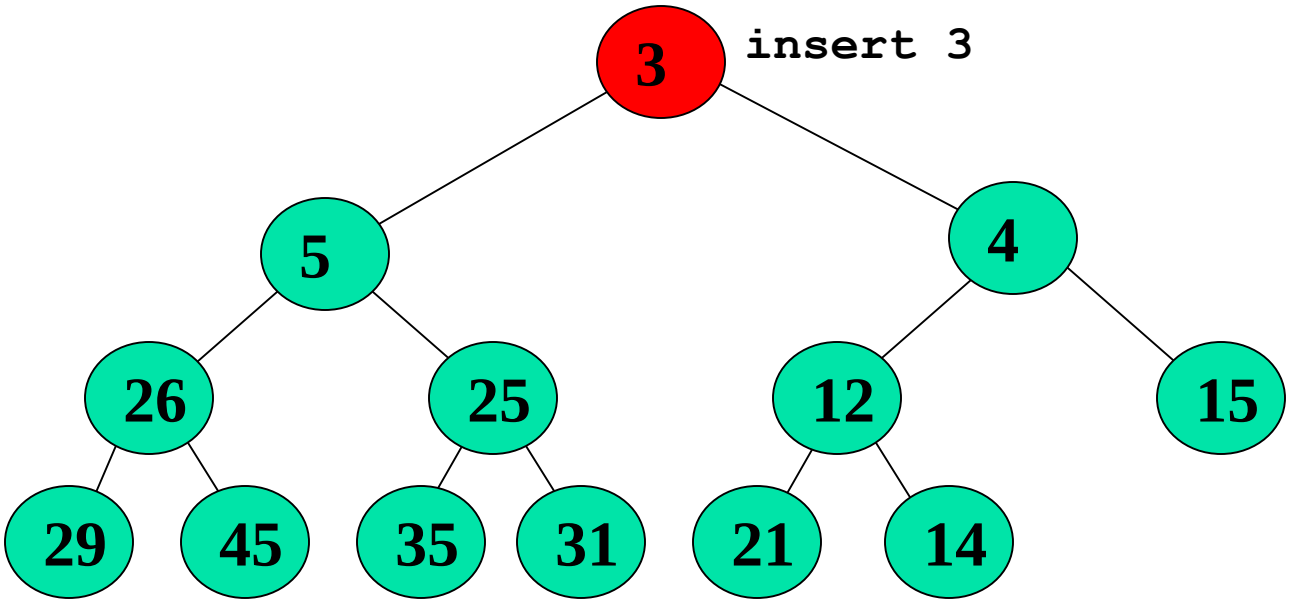
Insert 3

Inserting a Value



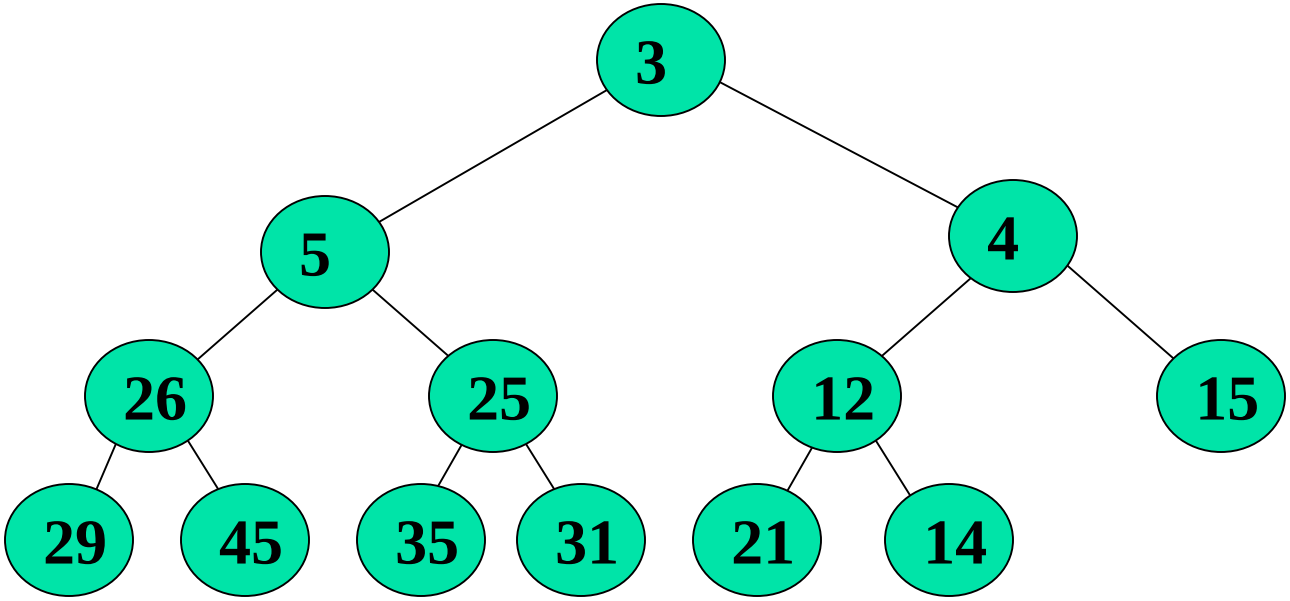
Insert 3

Inserting a Value

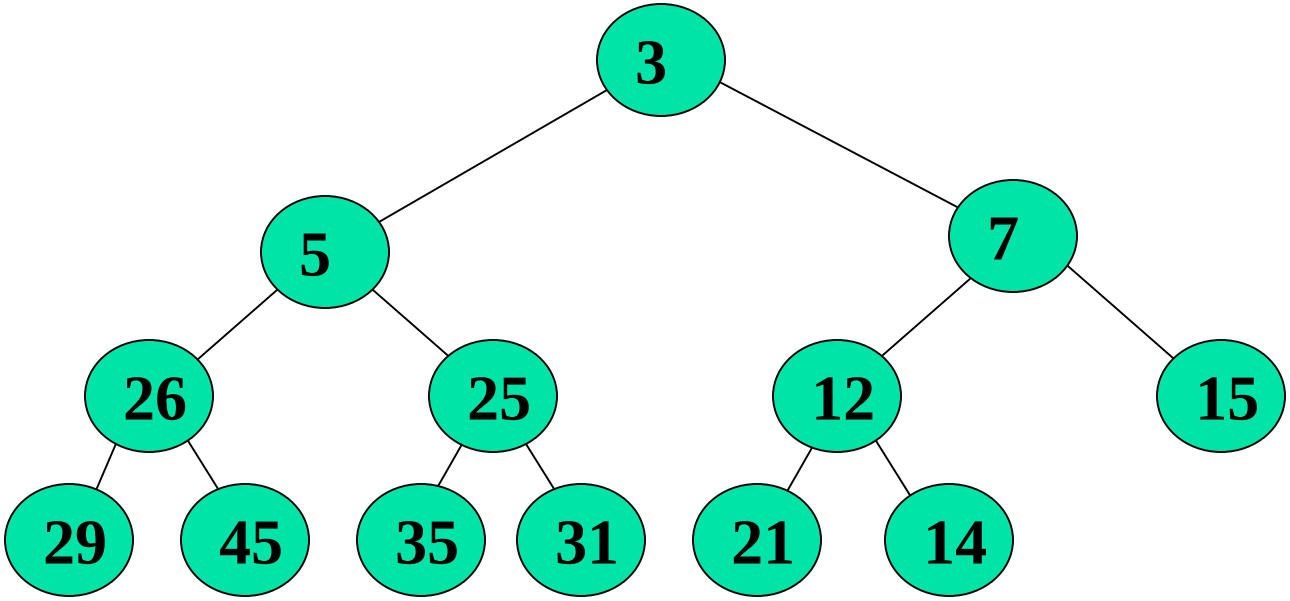


Insert 3

Heap After Insert

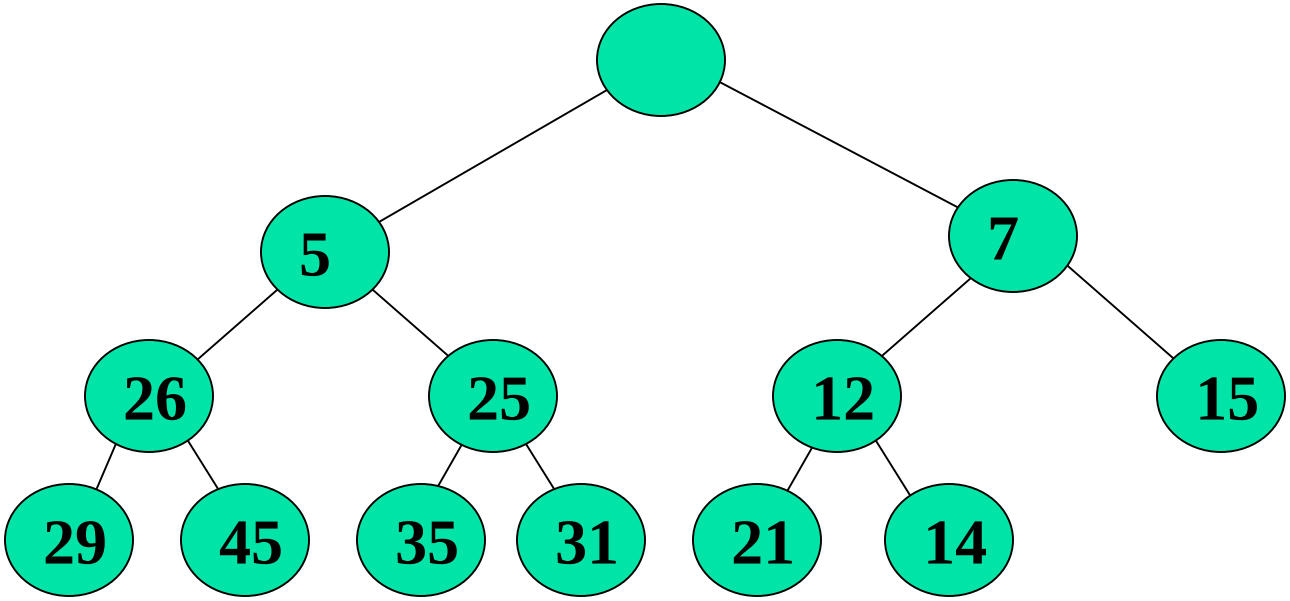


Deleting a Value (note new tree!)



Delete 3

Deleting a Value



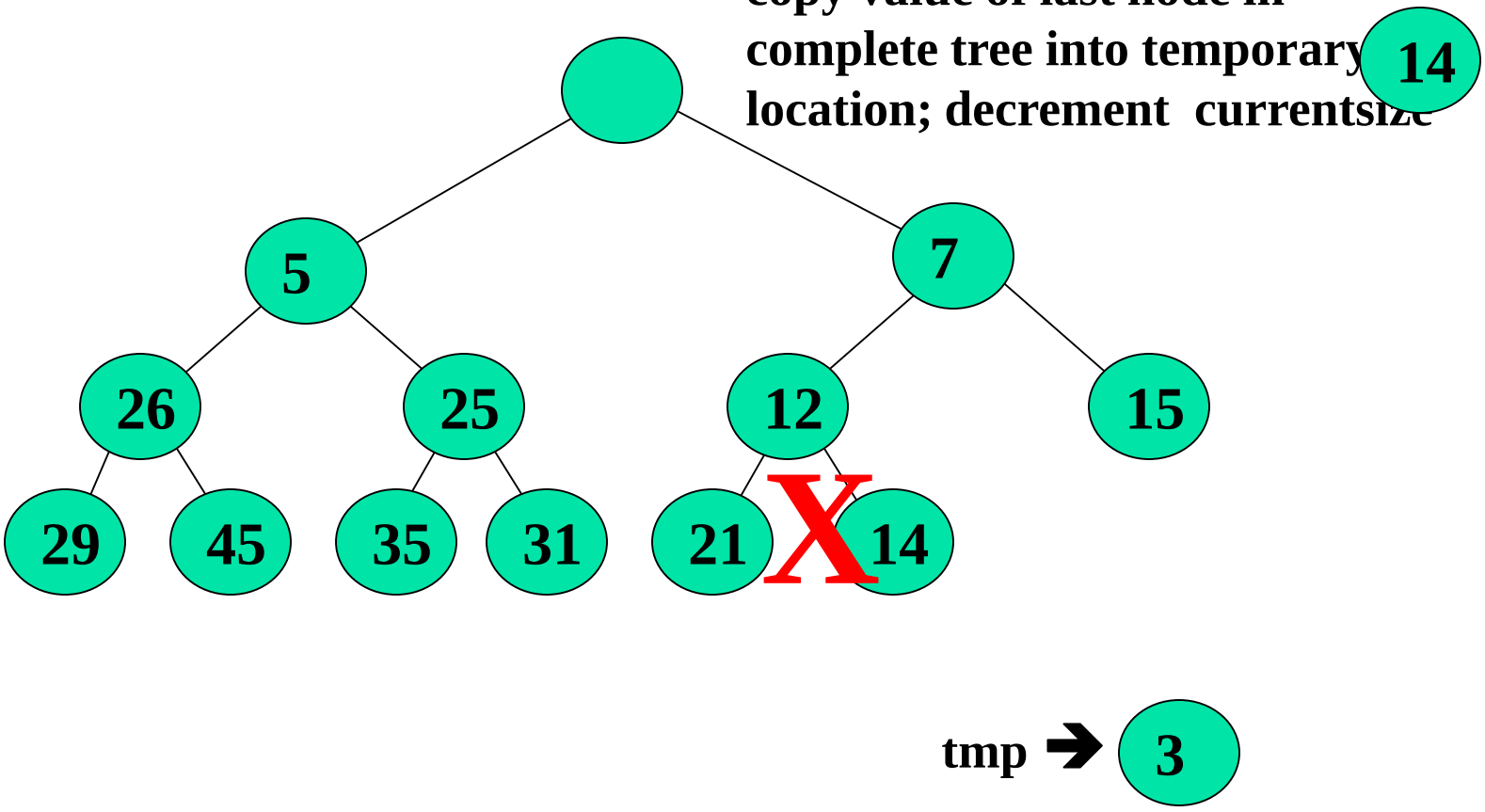
save root value … tmp **3**

➔

Delete 3

Deleting a Value

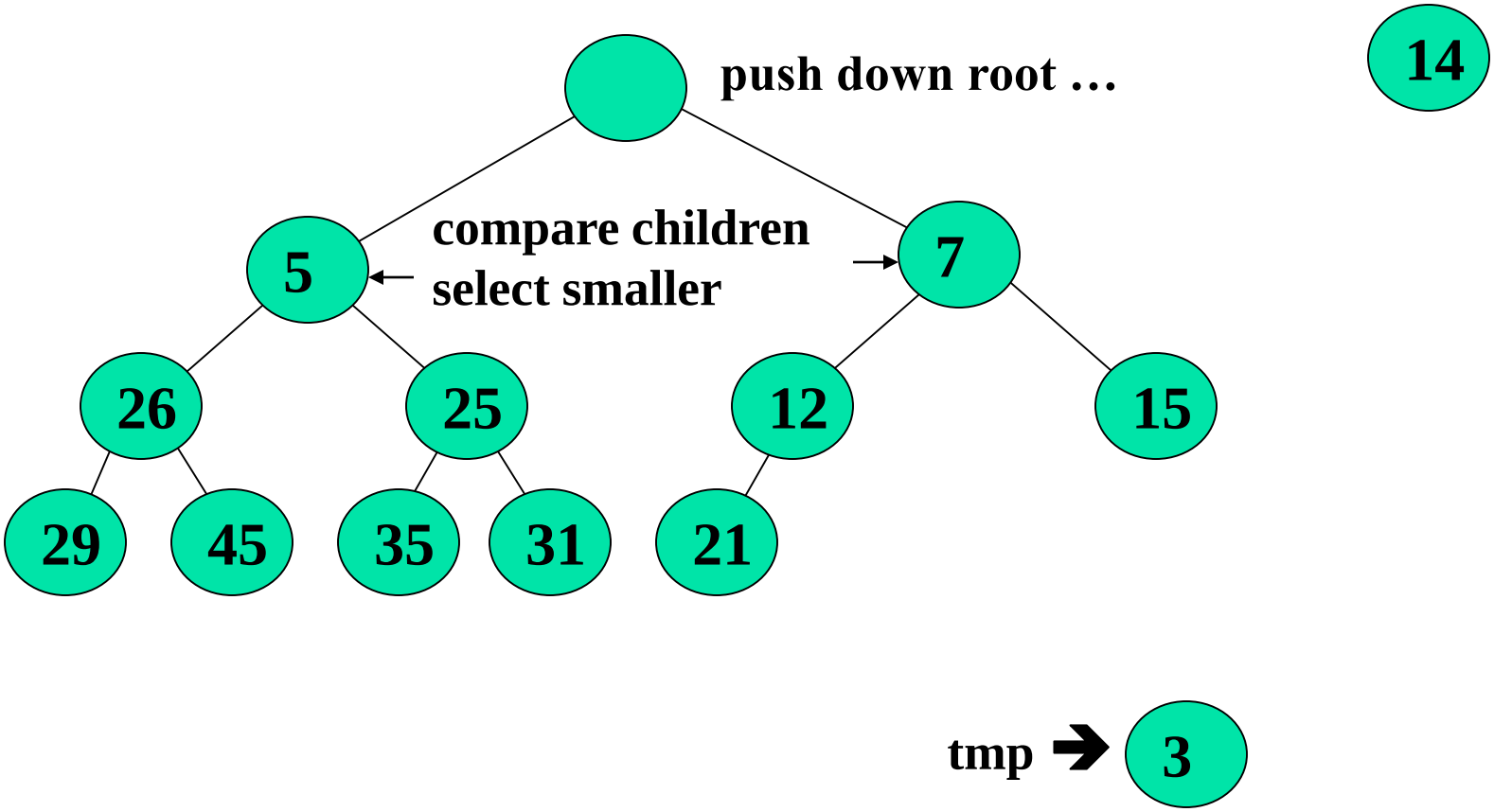
copy value of last node in complete tree into temporary location; decrement currentsize



tmp ➔ 3

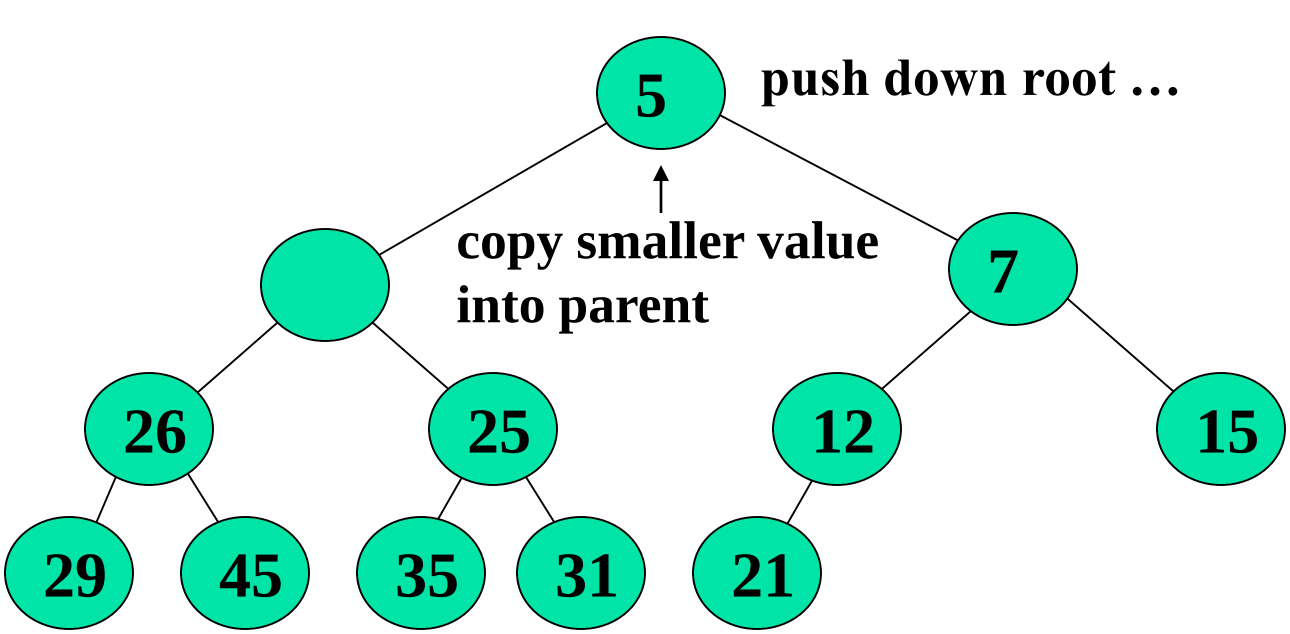
Delete 3

Deleting a Value



Delete 3

Deleting a Value

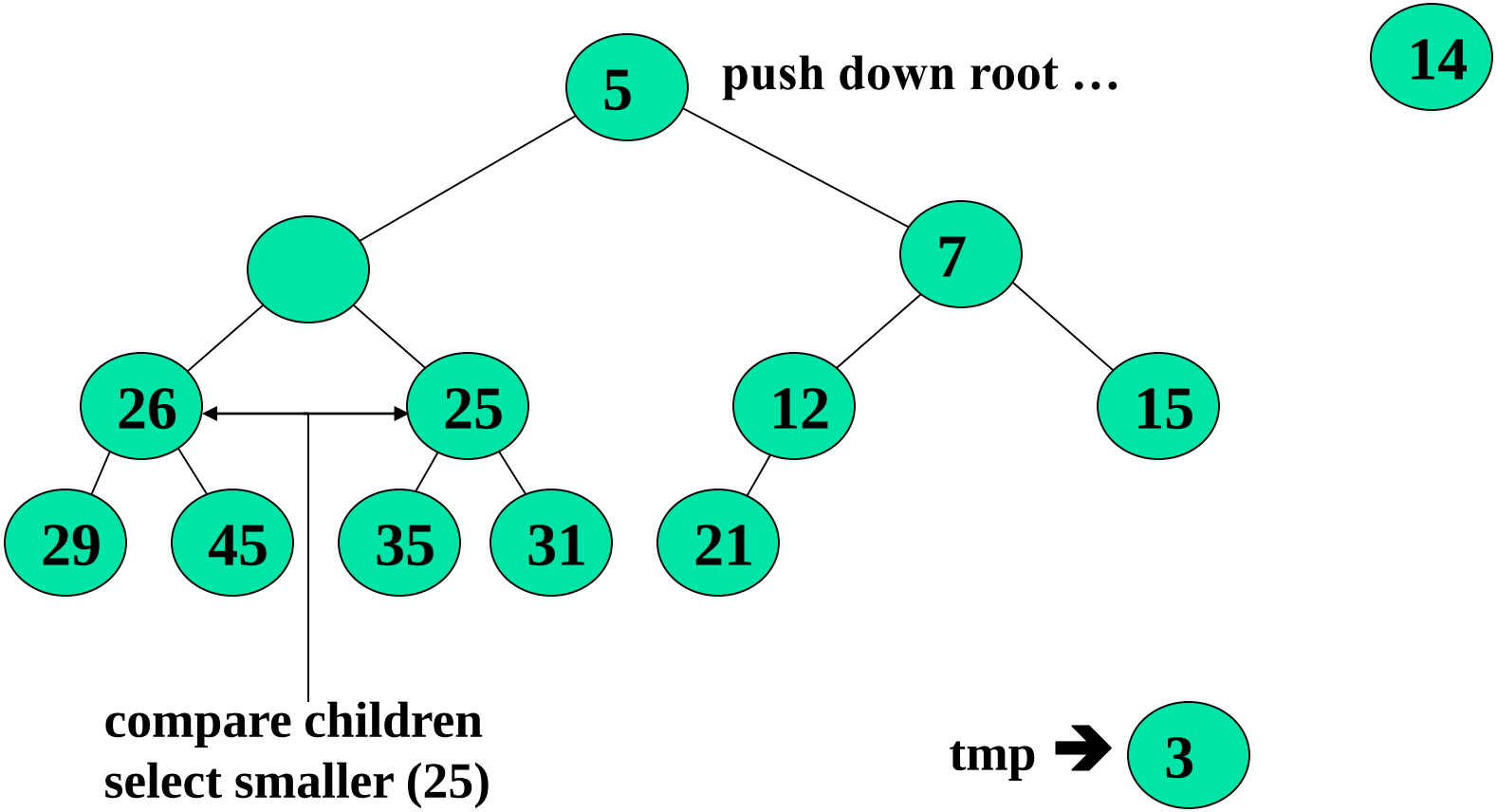


14

tmp ➔ 3

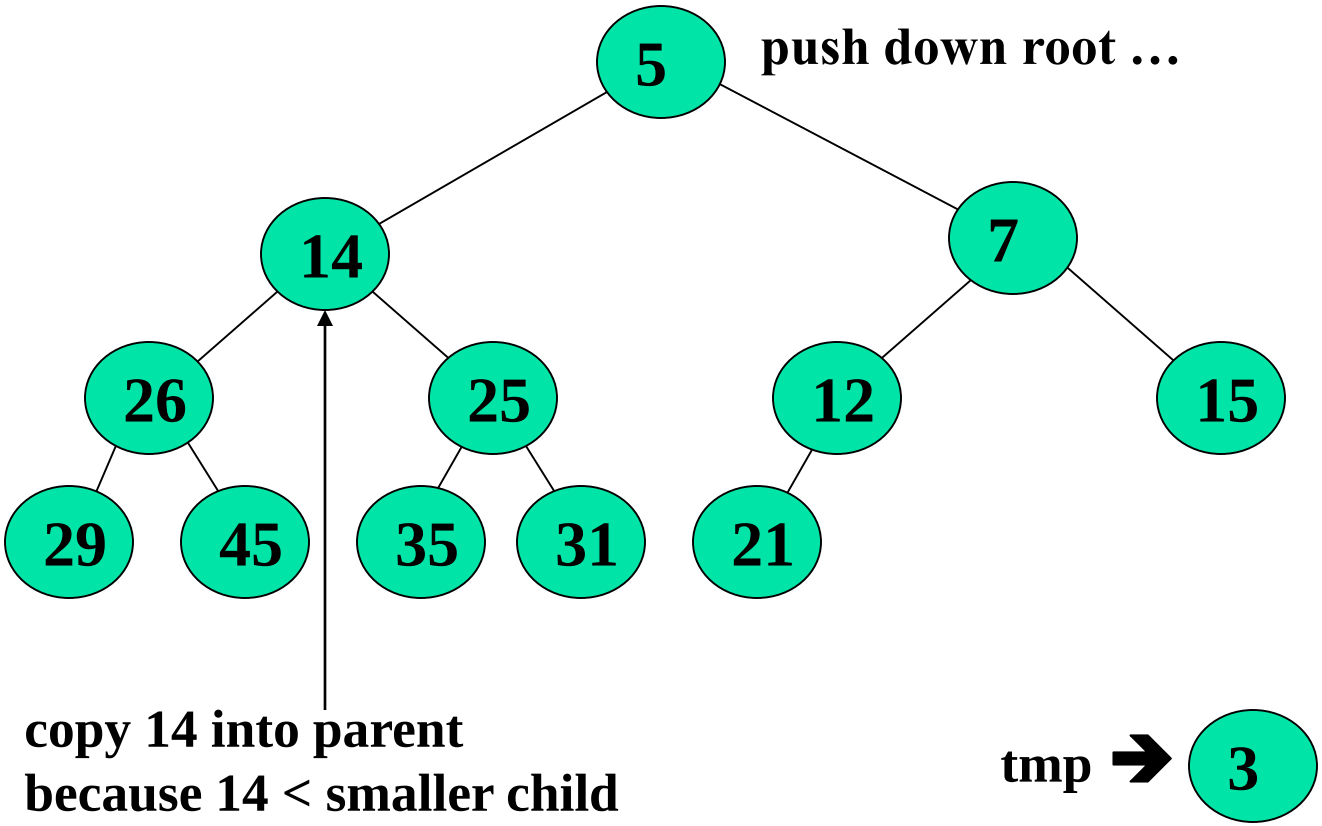
Delete 3

Deleting a Value



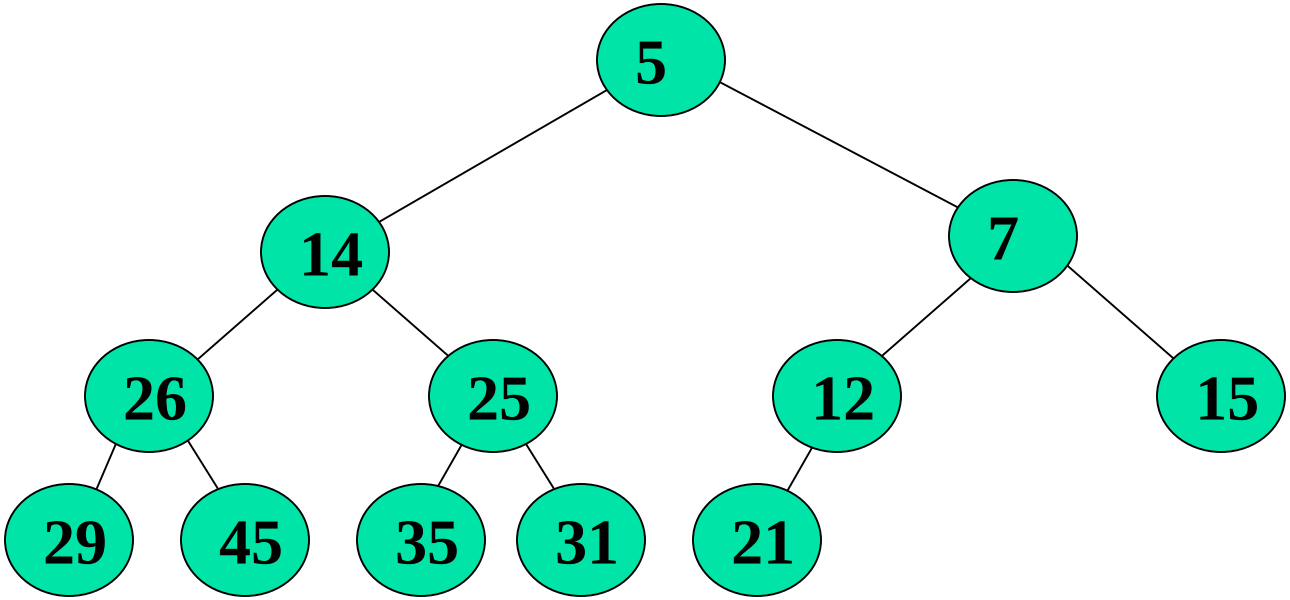
Delete 3

Deleting a Value



Delete 3

Deleting a Value

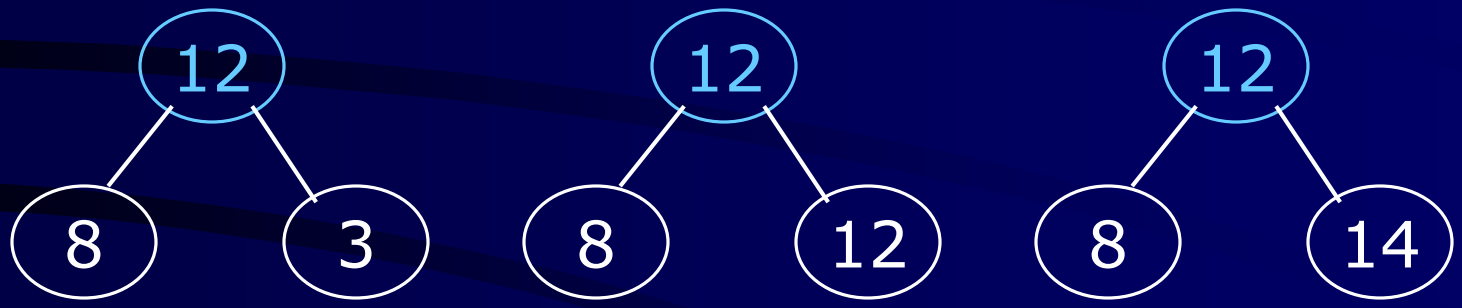


return 3

Delete 3

The heap property

- A node has the heap property if the value in the node is as large as or larger than the values in its children



Blue node has heap property

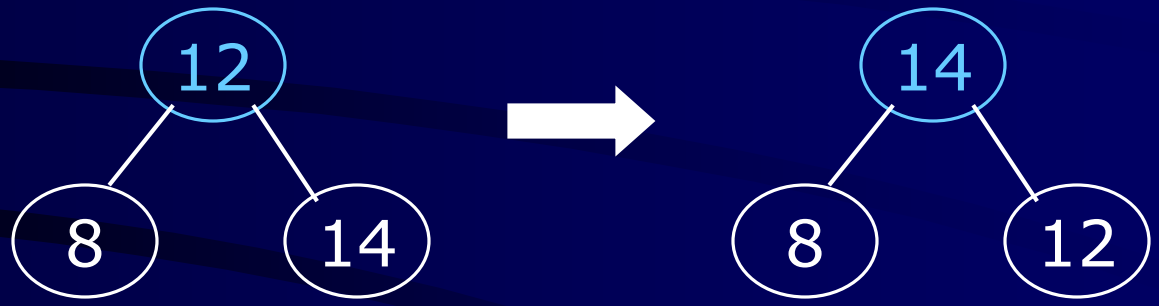
Blue node has heap property

Blue node does not have heap property

- All leaf nodes automatically have the heap property
- A binary tree is a heap if *all* nodes in it have the heap property

siftUp

- Given a node that does not have the heap property, you can give it the heap property by exchanging its value with the value of the larger child



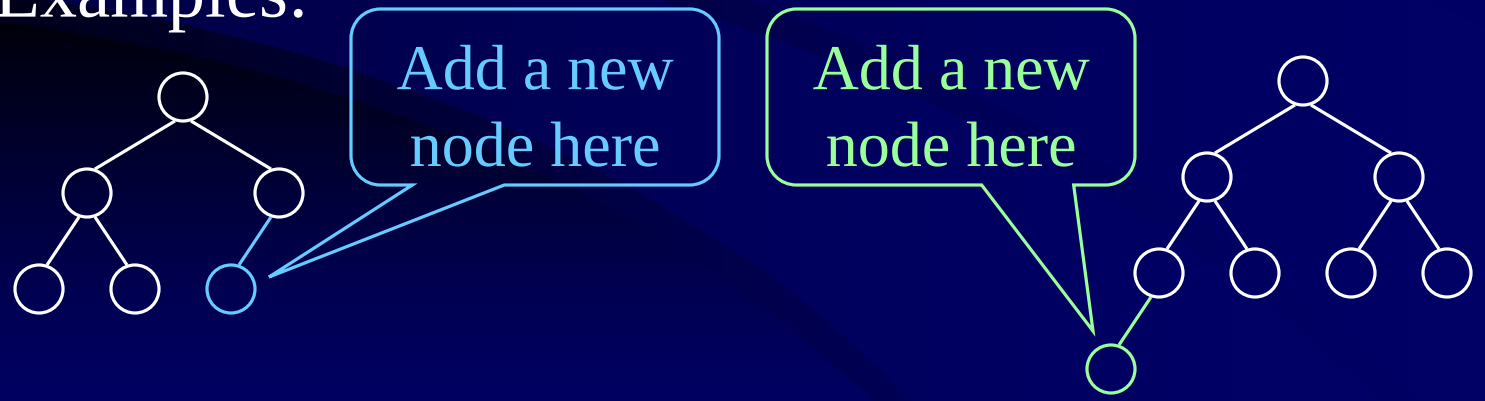
Blue node does not have heap property

Blue node has heap property

- This is sometimes called sifting up
- Notice that the child may have *lost* the heap property

Constructing a heap I

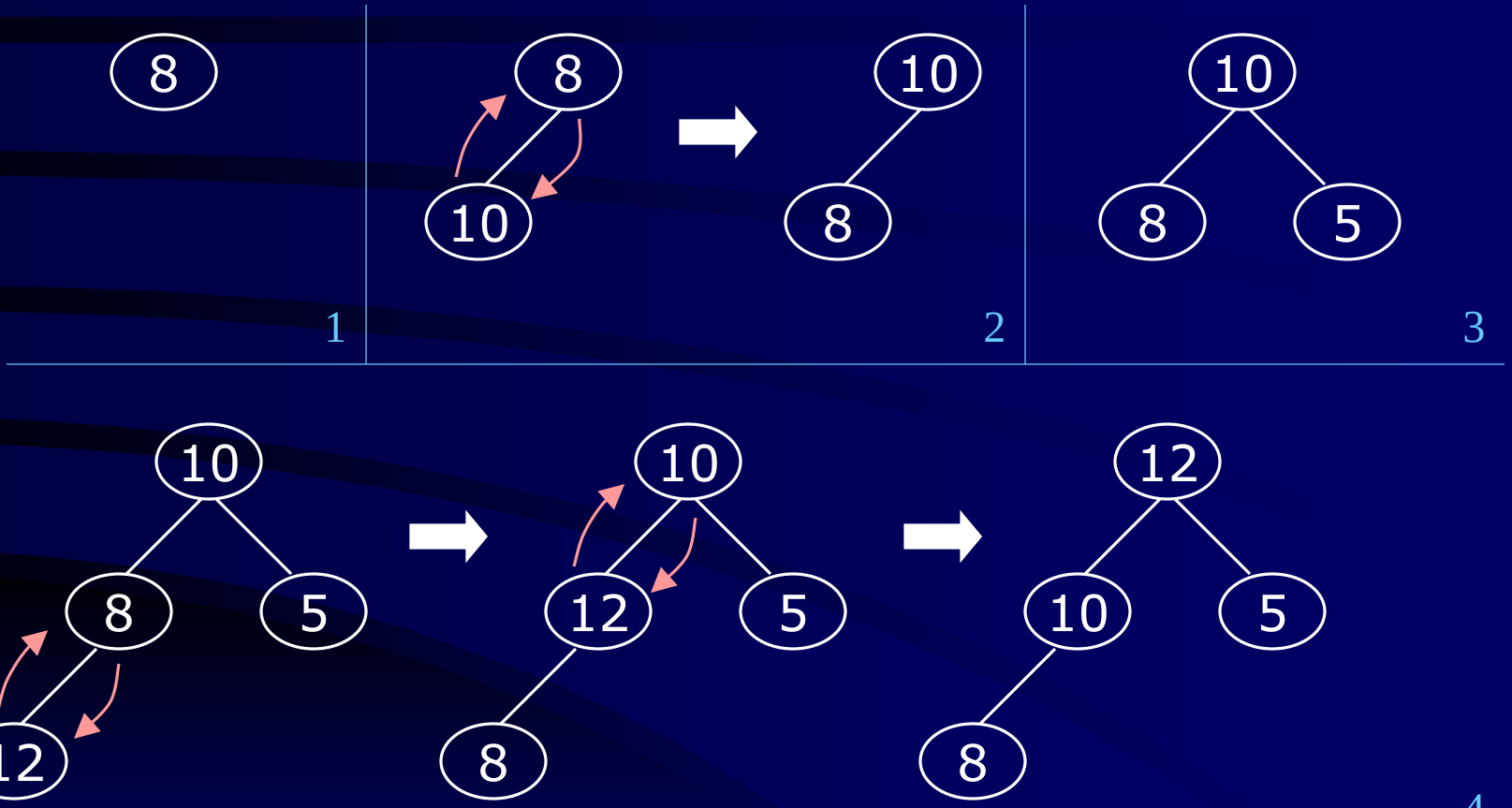
- A tree consisting of a single node is automatically a heap
- We construct a heap by adding nodes one at a time:
  - Add the node just to the right of the rightmost node in the deepest level
  - If the deepest level is full, start a new level
- Examples:



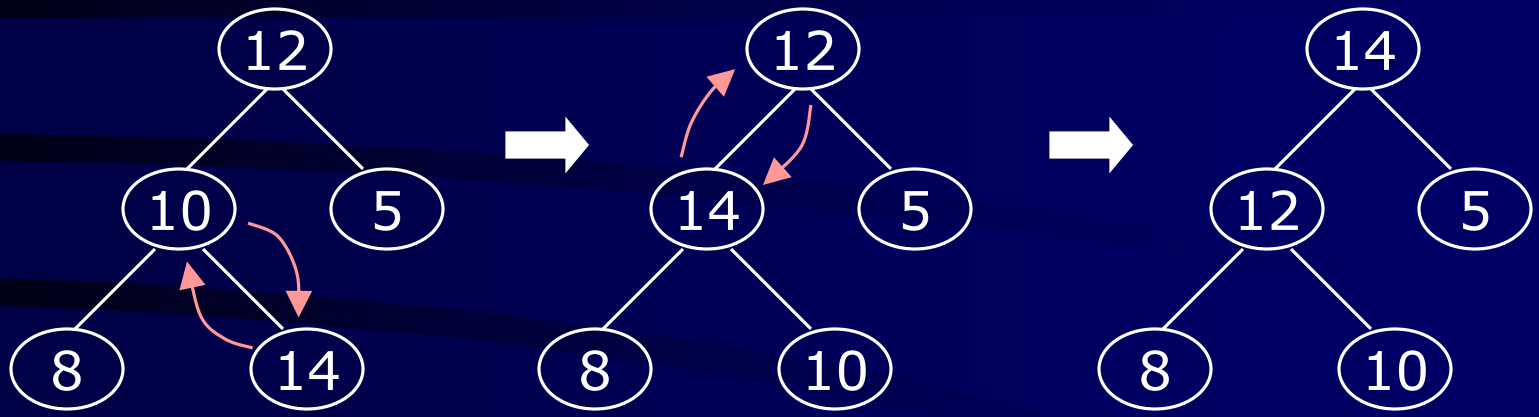
Constructing a heap II

- Each time we add a node, we may destroy the heap property of its parent node
- To fix this, we sift up
- But each time we sift up, the value of the topmost node in the sift may increase, and this may destroy the heap property of *its* parent node
- We repeat the sifting up process, moving up in the tree, until either
  - We reach nodes whose values don't need to be swapped (because the parent is *still* larger than both children), or
  - We reach the root

Constructing a heap III



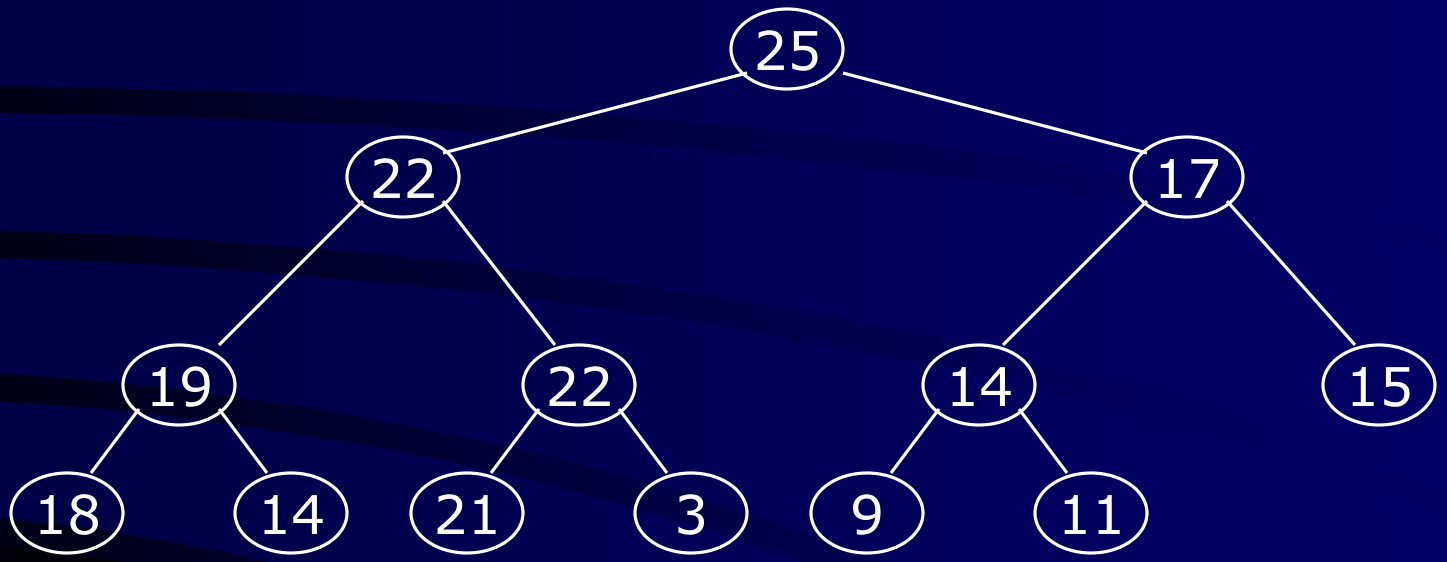
Other children are not affected



- The node containing 8 is not affected because its parent gets larger, not smaller
- The node containing 5 is not affected because its parent gets larger, not smaller
- The node containing 8 is still not affected because, although its parent got smaller, its parent is still greater than it was originally

A sample heap

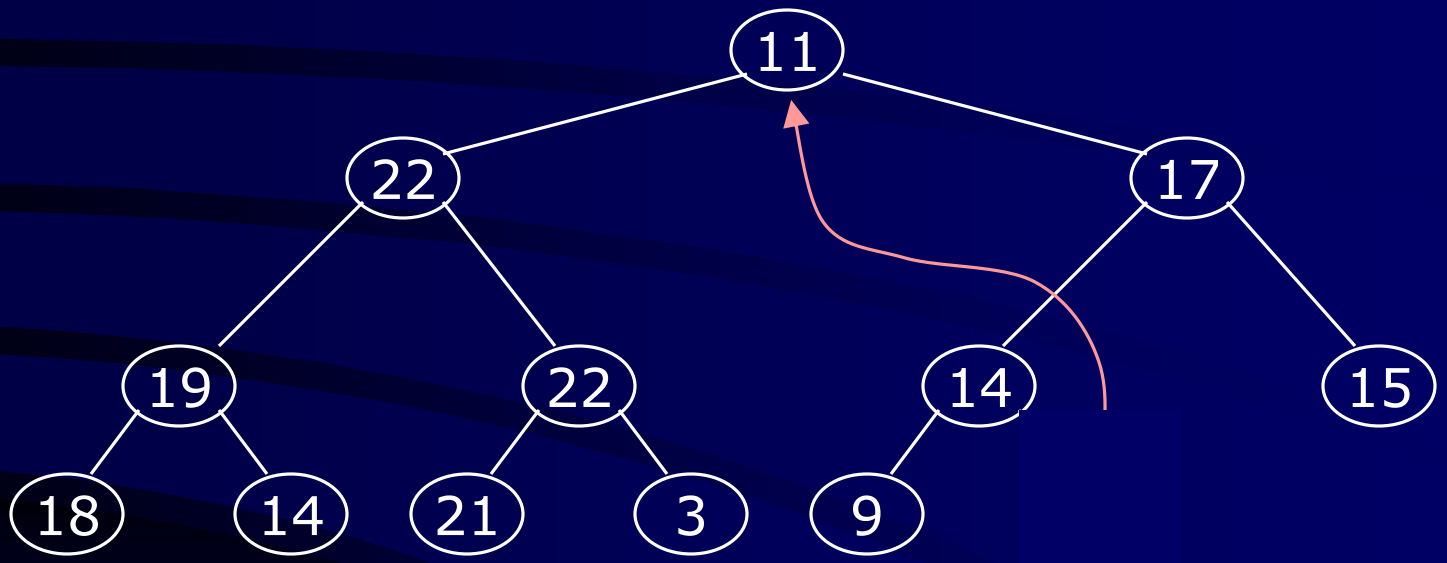
- Here's a sample binary tree after it has been heapified



- Notice that heapified does *not* mean sorted
- Heapifying does *not* change the shape of the binary tree; this binary tree is balanced and left-justified because it started out that way

Removing the root

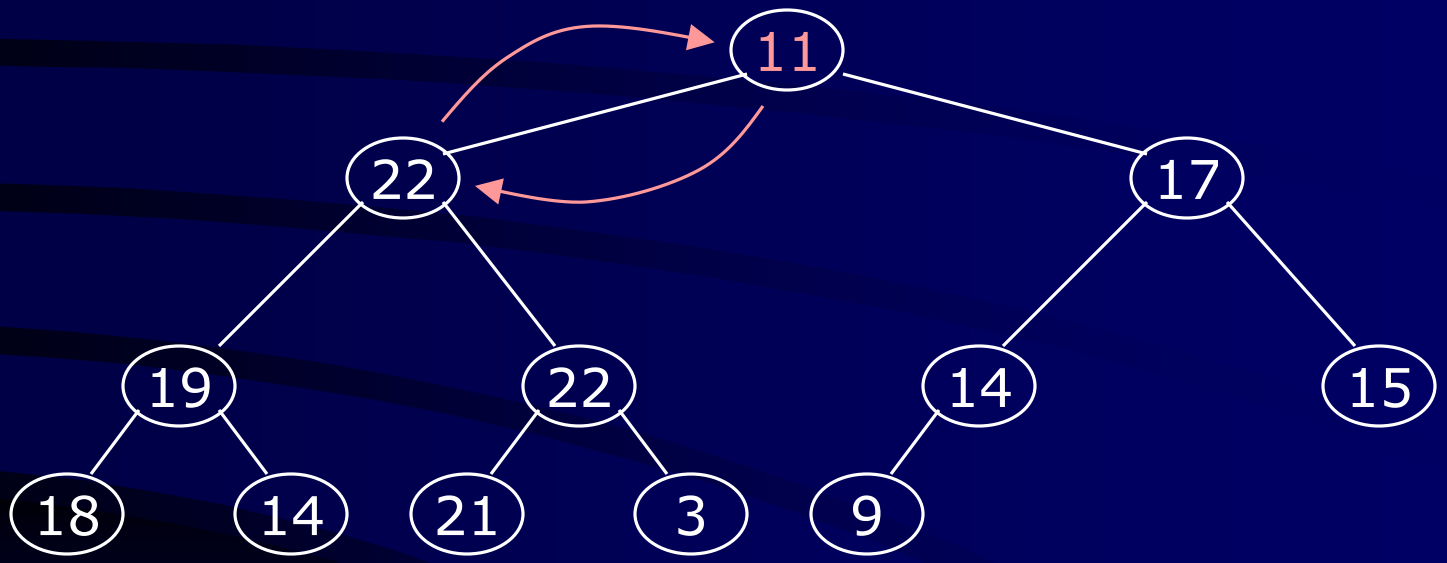
- Notice that the largest number is now in the root
- Suppose we *discard* the root:



- How can we fix the binary tree so it is once again *balanced and left-justified?*
- Solution: remove the rightmost leaf at the deepest level and use it for the new root

The reHeap method I

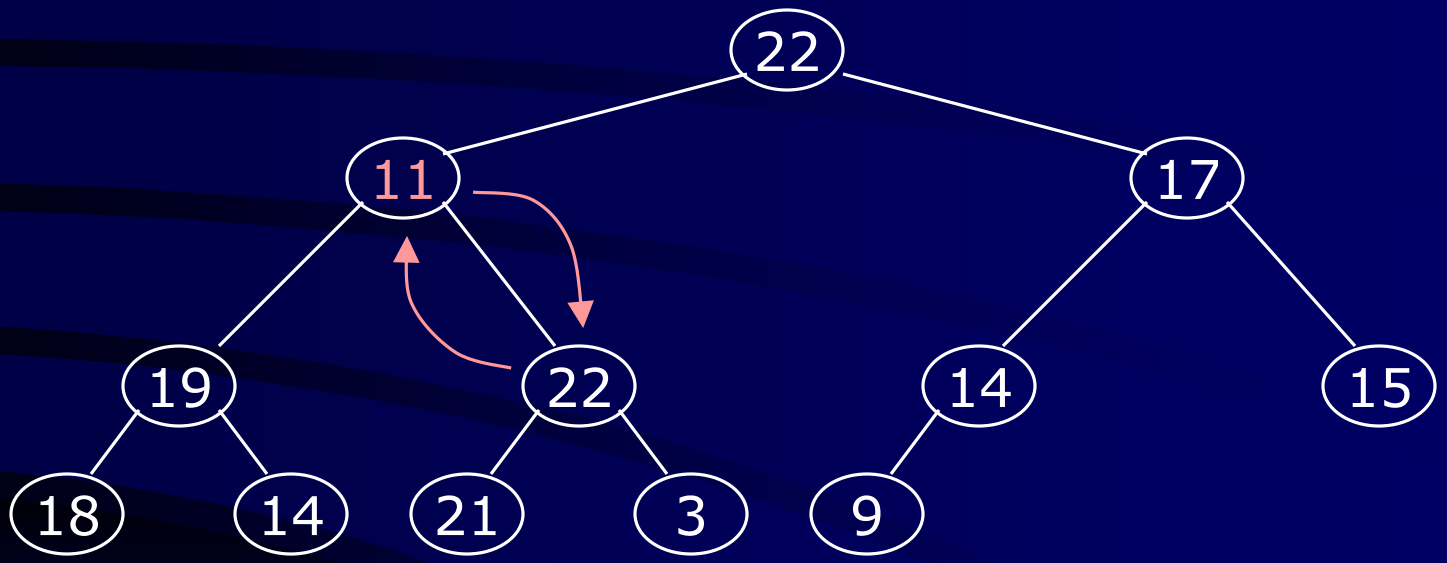
- Our tree is balanced and left-justified, but no longer a heap
- However, *only the root* lacks the heap property



- We can siftUp() the root
- After doing this, one and only one of its children may have lost the heap property

The reHeap method II

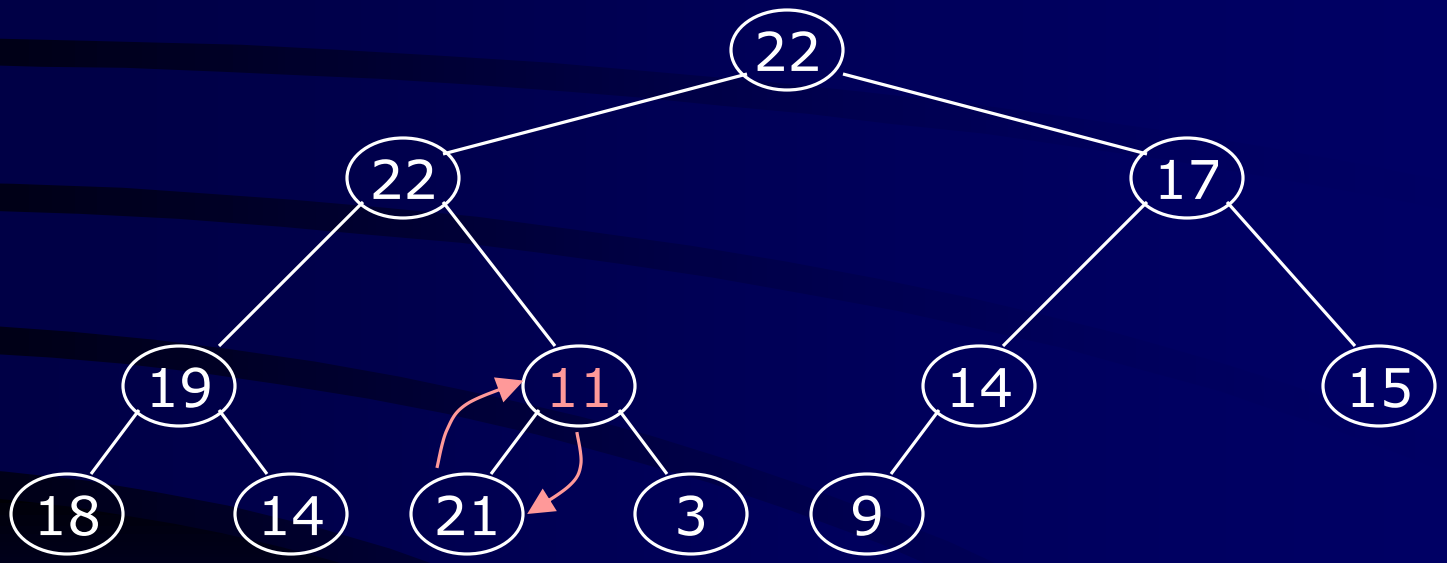
- Now the left child of the root (still the number `11`) lacks the heap property



- We can siftUp() this node
- After doing this, one and only one of its children may have lost the heap property

The reHeap method III

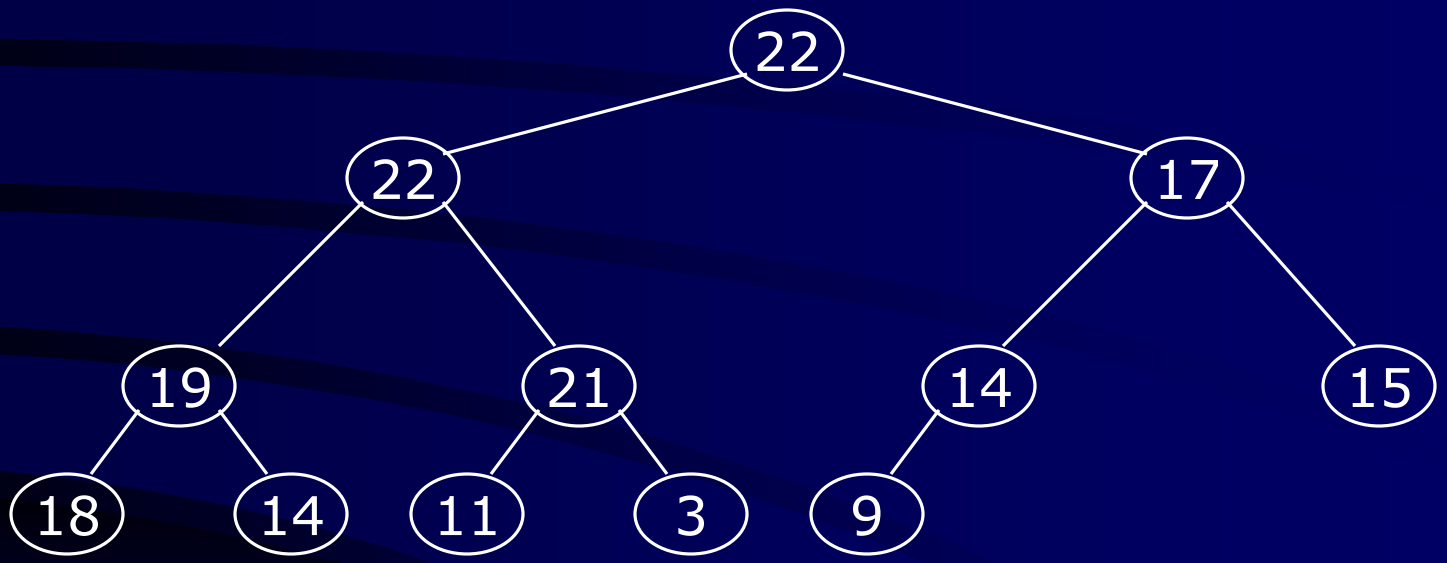
- Now the right child of the left child of the root (still the number 11) lacks the heap property:



- We can siftUp() this node
- After doing this, one and only one of its children may have lost the heap property —but it doesn't, because it's a leaf

The reHeap method IV

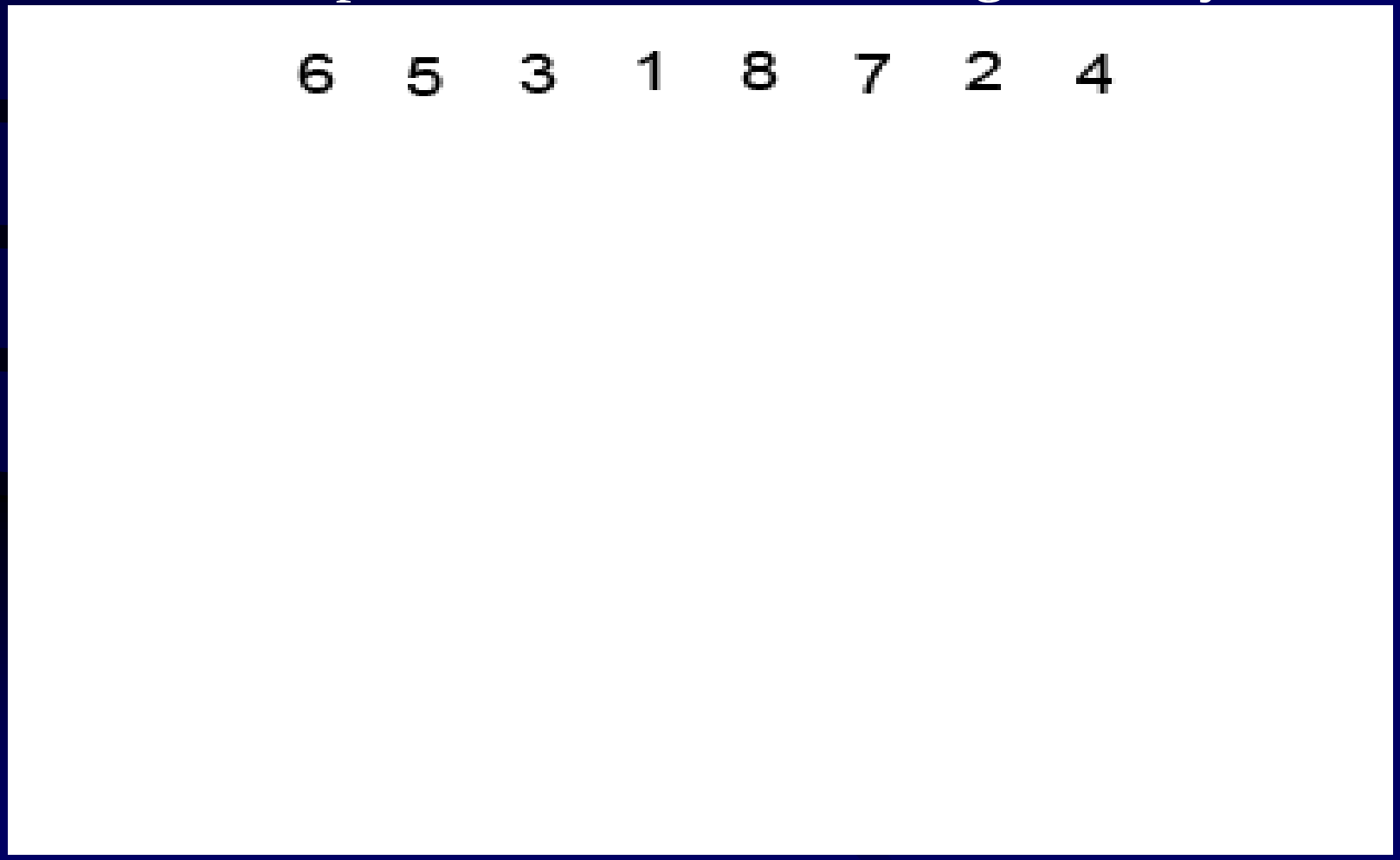
- Our tree is once again a heap, because every node in it has the heap property



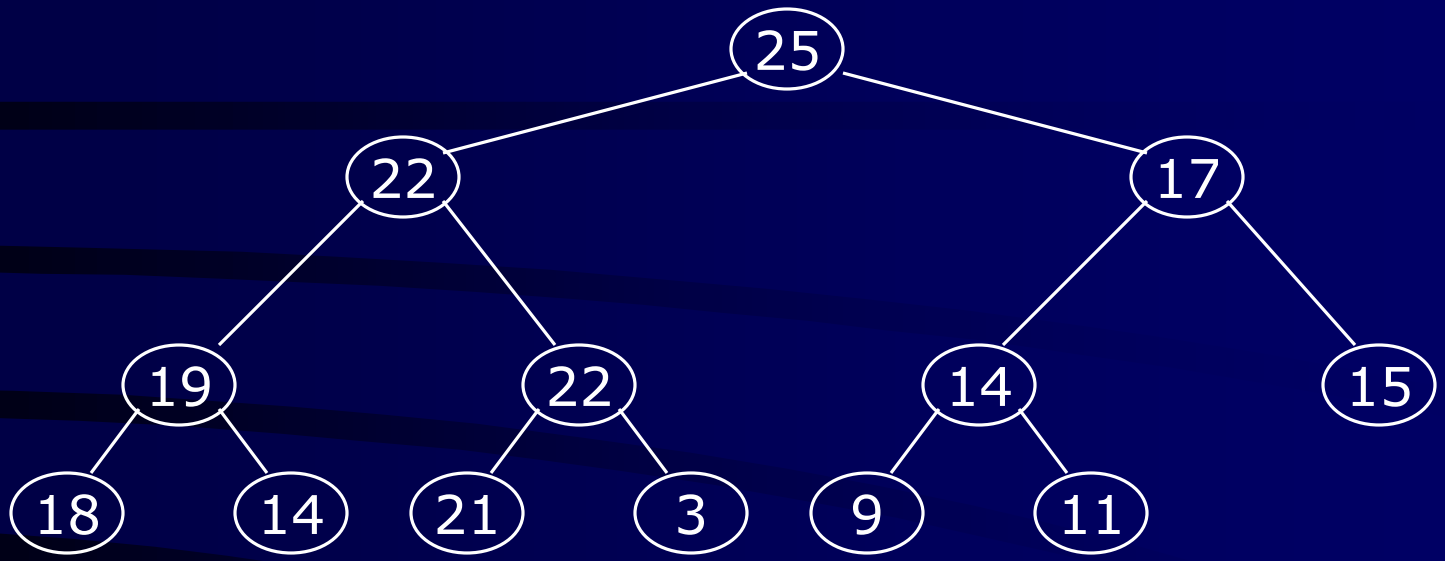
- Once again, the largest (or *a* largest) value is in the root
- We can repeat this process until the tree becomes empty
- This produces a sequence of values in order largest to smallest

Sorting

- What do heaps have to do with sorting an array?
-



Mapping into an array



0	1	2	3	4	5	6	7	8	9	10	11	12
25	22	17	19	22	14	15	18	14	21	3	9	11

- Notice:
  - The left child of index i is at index 2*i+1
  - The right child of index i is at index 2*i+2
  - Example: the children of node 3 (19) are 7 (18) and 8 (14)

Removing and replacing the root

- The “root” is the first element in the array
- The “rightmost node at the deepest level” is the last element
- Swap them...

- ...And pretend that the last element in the array no longer exists—that is, the “last index” is `11` (9)

Reheap and repeat

- Reheap the root node (index 0, containing 11)...

- ...And again, remove and replace the root node
- Remember, though, that the “last” array index is changed
- Repeat until the last becomes first, and the array is sorted!

Analysis I

- Here's how the algorithm starts:

  heapify the array;

- Heapifying the array: we add each of n nodes
  - Each node has to be sifted up, possibly as far as the root
    - Since the binary tree is perfectly balanced, sifting up a single node takes O(log n) time
  - Since we do this n times, heapifying takes n*O(log n) time, that is, O(n log n) time

Analysis II

- Here's the rest of the algorithm:

```
while the array isn't empty {
    remove and replace the root;
    reheap the new root node;
}
```

- We do the while loop `n` times (actually, `n-1` times), because we remove one of the `n` nodes each time
- Removing and replacing the root takes `O(1)` time
- Therefore, the total time is `n` times however long it takes the `reheap` method

Analysis III

- To reheap the root node, we have to follow *one path* from the root to a leaf node (and we might stop before we reach a leaf)
- The binary tree is perfectly balanced
- Therefore, this path is O(log n) long
  - And we only do O(1) operations at each node
  - Therefore, reheaping takes O(log n) times
- Since we reheap inside a while loop that we do n times, the total time for the while loop is n*O(log n), or O(n log n)

Analysis IV

- Here's the algorithm again:

```
heapify the array;
while the array isn't empty {
    remove and replace the root;
    reheap the new root node;
}
```

- We have seen that heapifying takes O(n log n) time
- The while loop takes O(n log n) time
- The total time is therefore O(n log n) + O(n log n)
- This is the same as O(n log n) time

The End