CS350: Data Structures

Heaps and Priority Queues

James Moscola
Department of Engineering & Computer Science
York College of Pennsylvania

Priority Queue

- **An abstract data type of a queue that associates a priority with each of the elements inserted**

- **Elements are enqueued with some priority**

- **Elements are dequeued in priority order**
  - Highest priority elements are dequeued first

Binary Heaps

- **Great for an implementation of a priority queue**
- **Similar in structure to a binary search tree**
- **Sorting of elements in a heap is much weaker than in a BST, however it is sufficient to implement a priority queue**

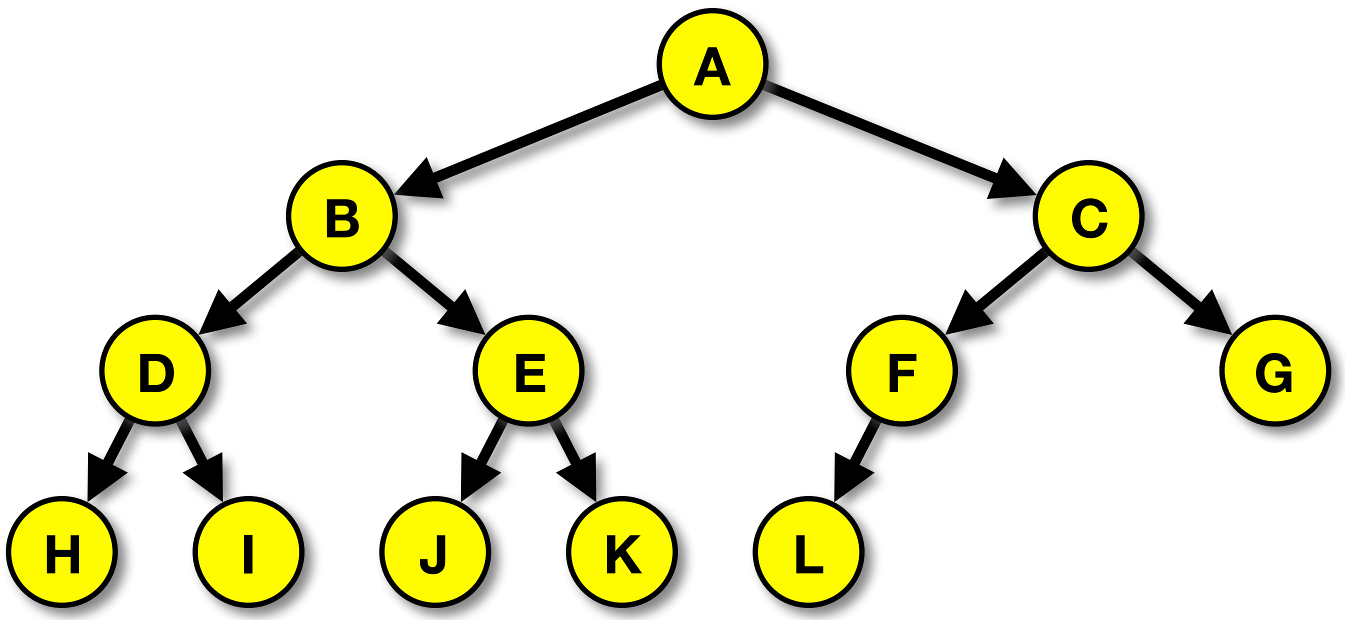
- **A binary heap can be either a Min Heap or a Max Heap**
  - Min Heap - smallest key values have highest priority
  - Max Heap - largest key values have highest priority

Complete Binary Tree

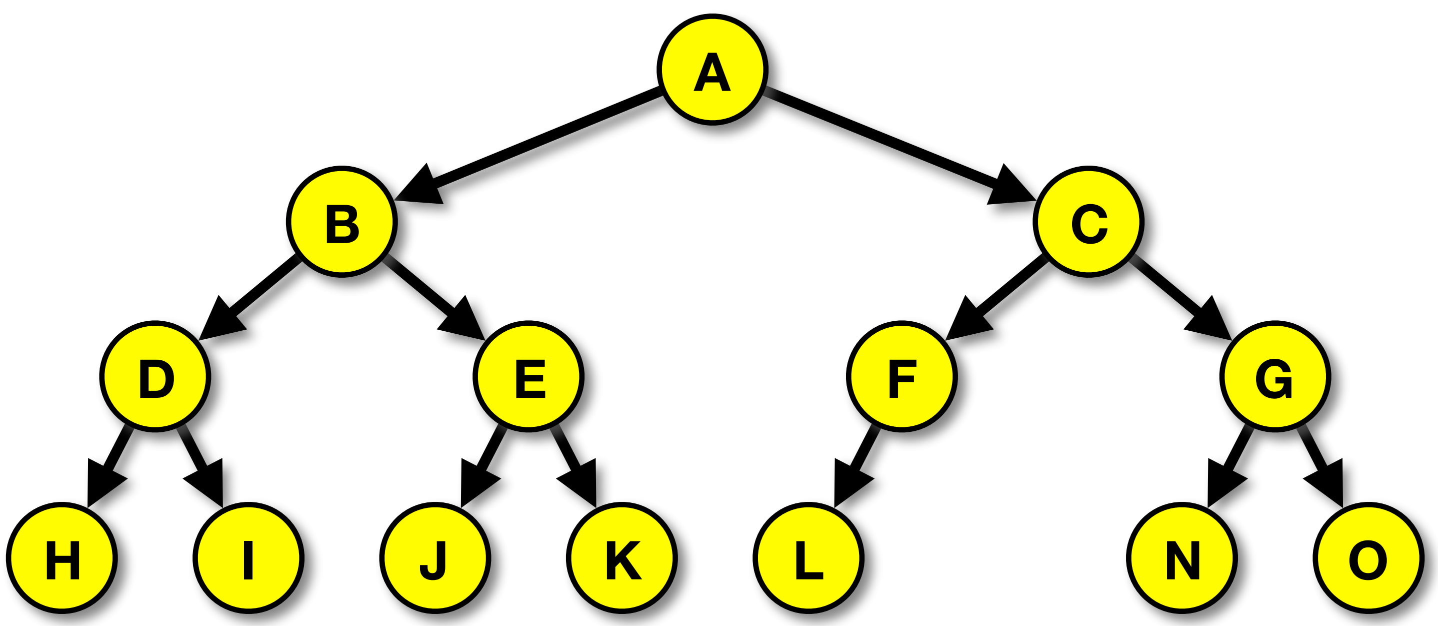
- **A heap is implemented as a *complete binary tree* which can be stored efficiently in an array**

- **A complete binary tree has the following properties:**
  - Tree is filled in level-order from left to right
  - Tree has the maximum number of nodes at every level except for possibly the bottom level
  - There are no holes allowed in the tree

Complete Binary Tree



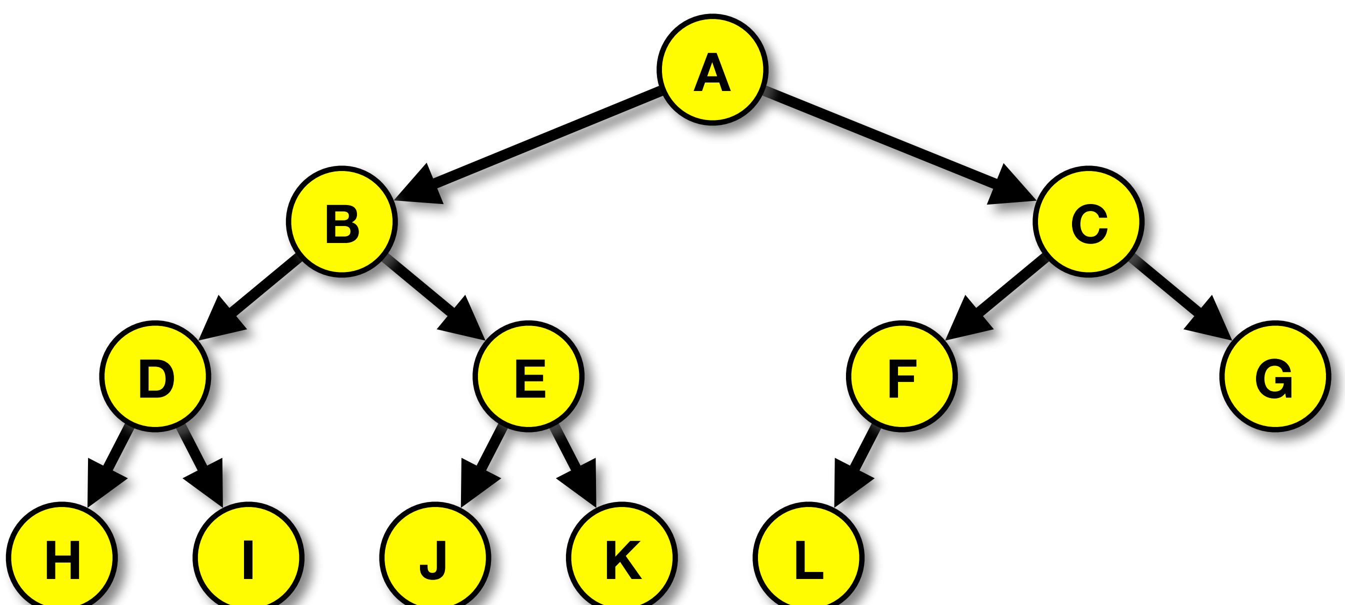
This IS a *complete binary tree*



This IS NOT a *complete binary tree*

Complete Binary Tree

- **Using a complete binary tree to represent a heap makes traversing the heap easy**
- **The complete binary tree can easily be stored in an array**



	A	B	C	D	E	F	G	H	I	J	K	L			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

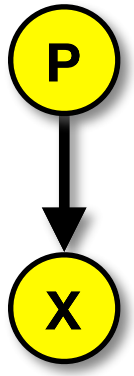
```
left child = 2 * i
right child = 2 * i + 1

parent = ⌊i / 2⌋
```

Binary Heap Properties

- **Just as with all other data structures, there are properties that must be maintained for the binary heap**

- **Elements in the heap must maintain the heap-order property**
  - Heap-order property (for a Min Heap):
    - In a heap, for every node X with parent P, the key in P is less than or equal to the key in X (i.e. the parent's key is less than a child's key)



$P \le X$

Binary Heap Operations: `insert`

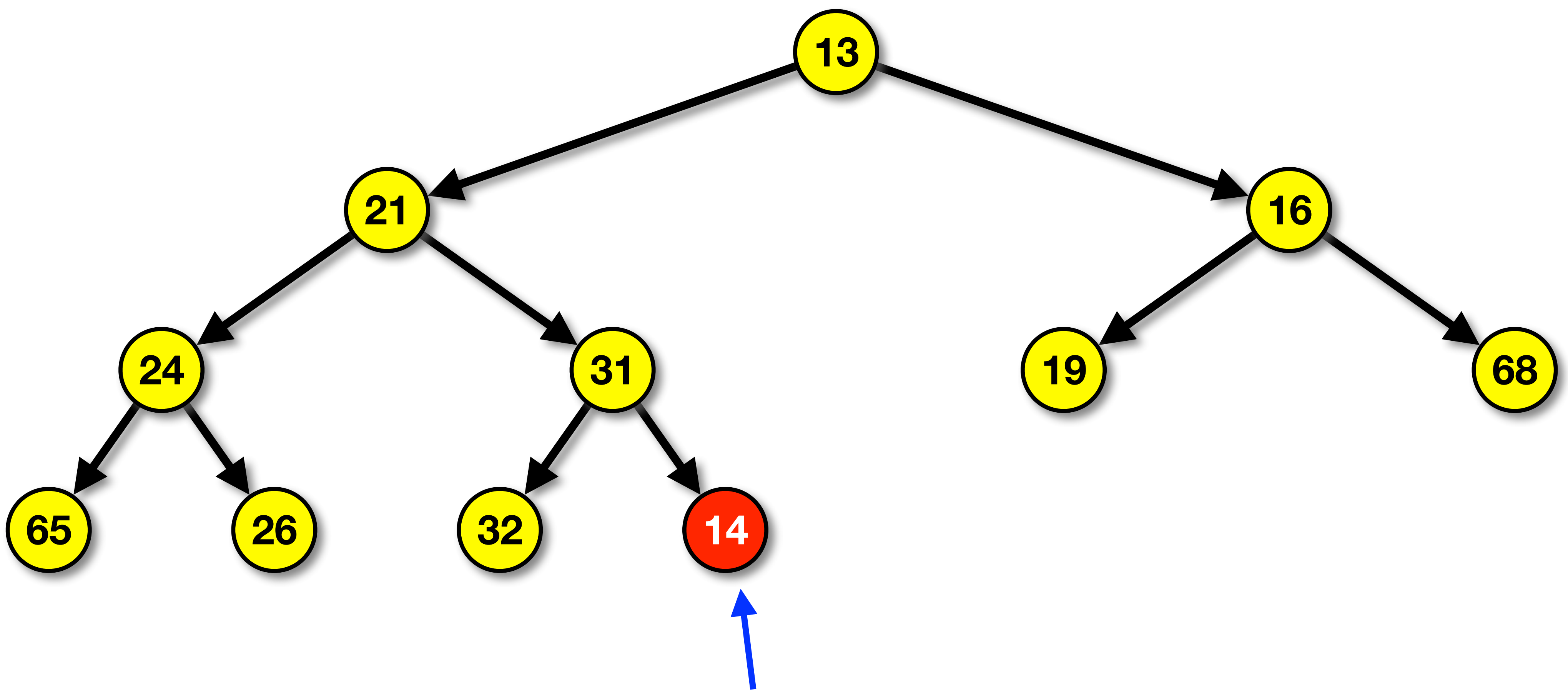
- **The basic idea for `insert`:**
  - Initially, insert the new element into the next available complete tree location (it may not stay here)
  - If the new element ***does not*** violate the heap-order property, then leave the new element in that location; insert is done
  - If the new element ***does*** violate the heap-order property, then move the element up the heap until a location is found where the new element does not violate the heap-order property
    - This operation is called `percolateUp`

Binary Heap Operations: `percolateUp`

- Select the node that is to be percolated up the heap
- Compare the node with its parent
- If the node is less than it's parent, then swap the node with its parent
- Compare the node to its new parent
- Continue moving the node up the tree until the node is greater than or equal to its parent node (i.e. it's parent is smaller)

Binary Heap Operations: `insert` Example

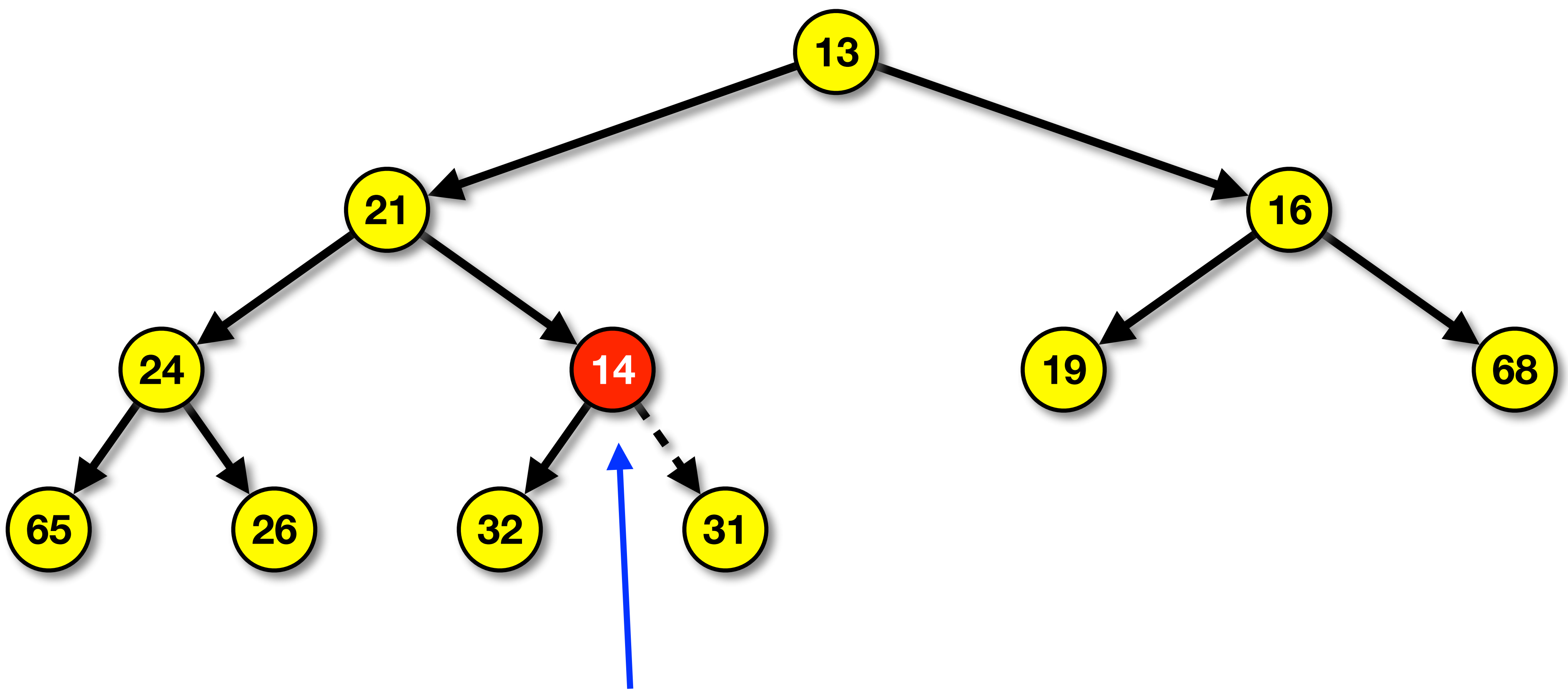
Insert node 14
First insert the new element in the next available heap location



Can 14 be inserted at this location without violating the heap-order property?

Binary Heap Operations: `insert` Example

Insert node 14
Swap the element with its parent to move it up the heap

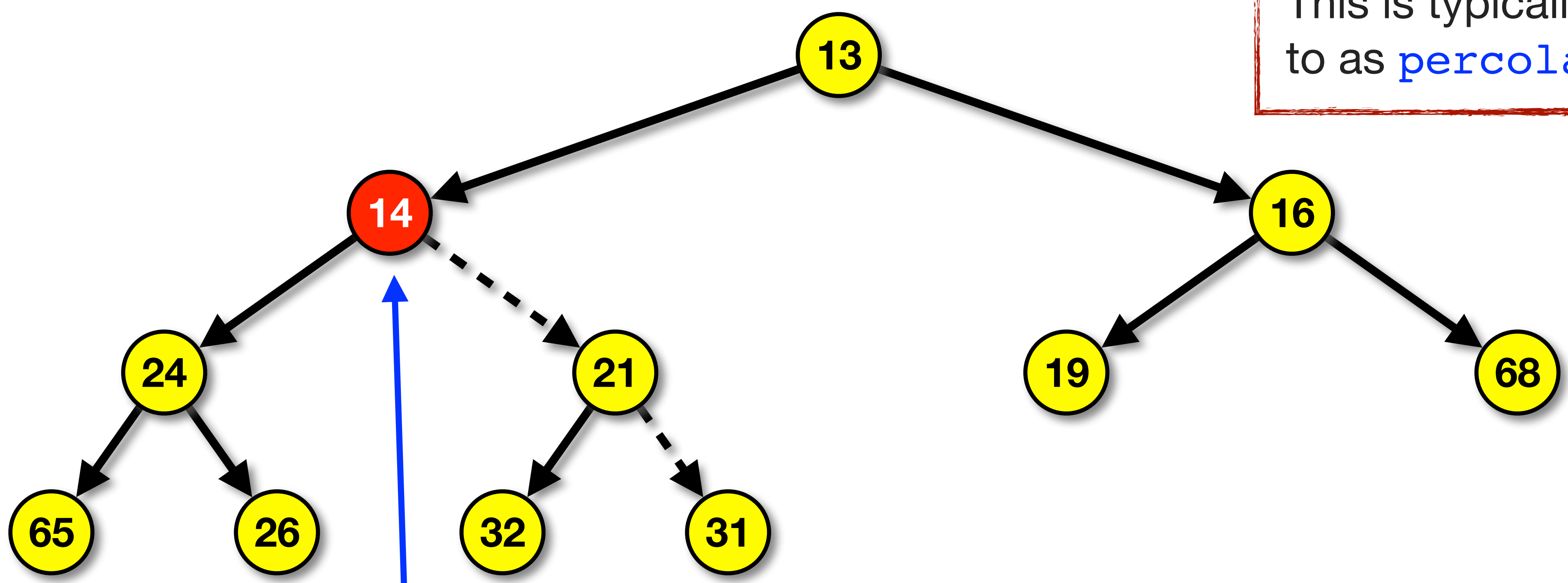


Can 14 be inserted at this location without violating the heap-order property?

Binary Heap Operations: `insert` Example

Insert node 14
Continually swap the element with its parent to move it up the heap until a valid location is found

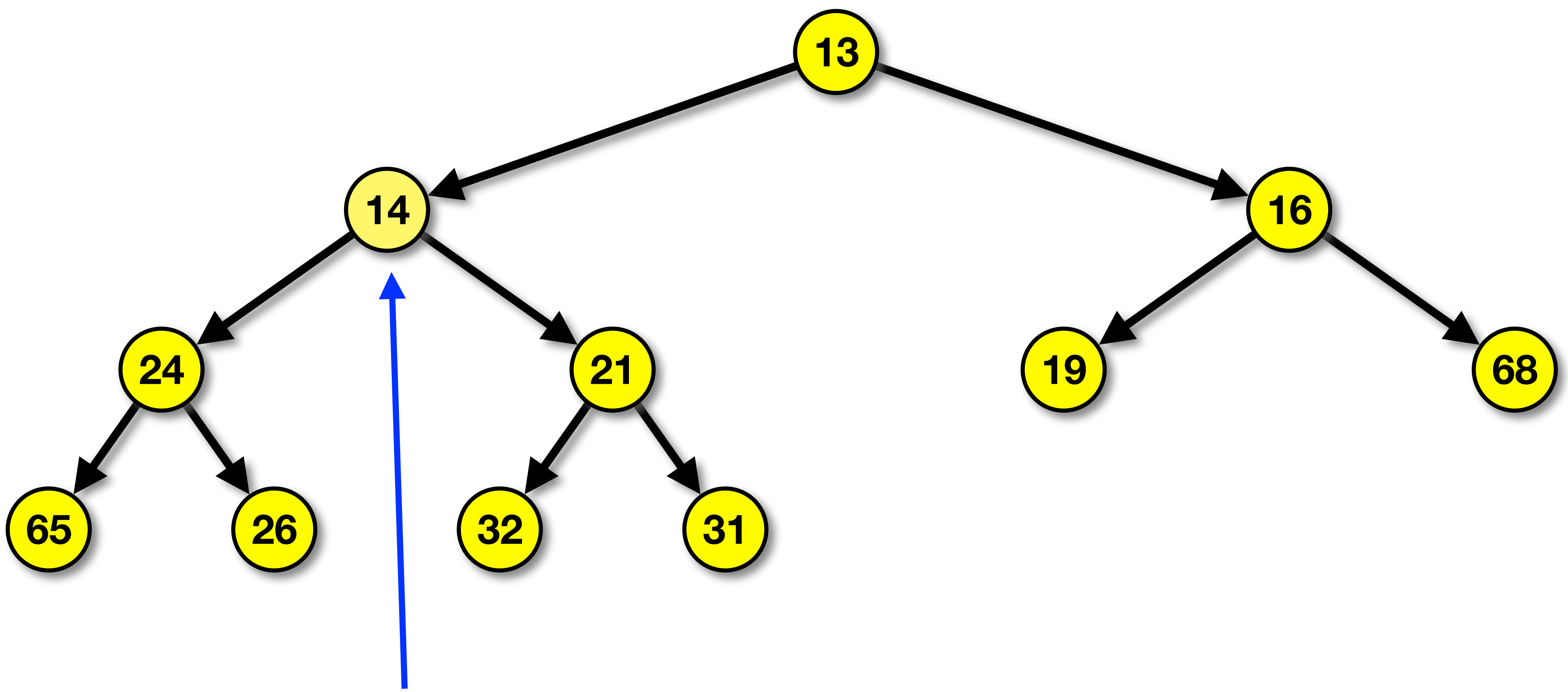
This is typically referred to as `percolateUp`



Can 14 be inserted at this location without violating the heap-order property?

Binary Heap Operations: `insert` Example

Insert node 14
Done with insertion



Node 14 satisfies the heap-order property in this location
Done with insertion since `(14 > 13)`

Binary Heap Operations: `insert`

- **Time required to do insertion could be as much as O(log N) if the value getting inserting is the new minimum value in the heap**
  - A newly inserted value that is the minimum value must percolate all the way up the tree

Binary Heap Operations: `deleteMin`

- **The basic idea for `deleteMin`:**
  - Delete the root node of the heap (will be the minimum node)
    - Not really "deleting", but value will be overwritten
  - Move the last node in the heap (the rightmost node on the bottom level) to the location where the root node was located
    - The last node becomes the root node, but may not stay there
  - Move the new root node down the tree until a location is found where that node does not violate the heap-order property
    - This operation is called `percolateDown`

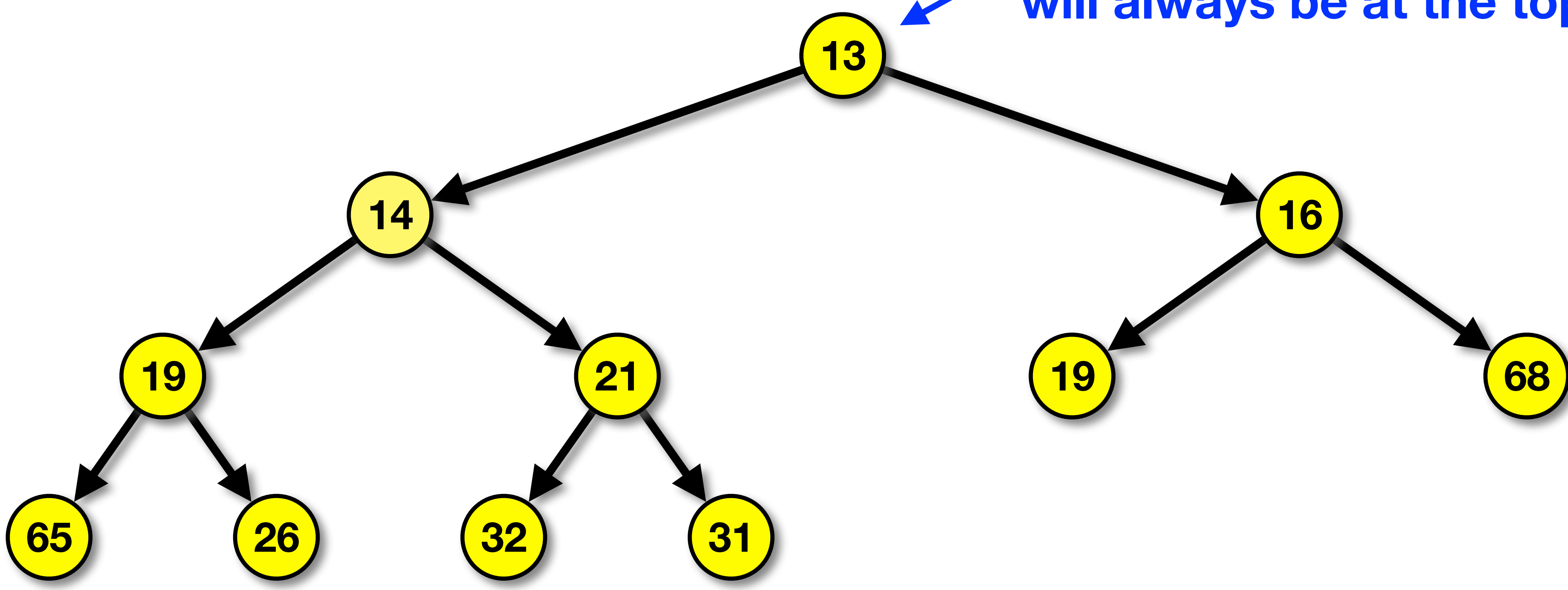
Binary Heap Operations: `percolateDown`

- Select the node that is to be percolated down the heap
- Compare the node the lesser of its two children
- If the node is greater than the lesser of its two children, then swap the node with that child
- Compare the node to its new children
- Continue moving the node down the tree until the node is less than or equal to both of its children

Binary Heap Operations: `deleteMin` Example

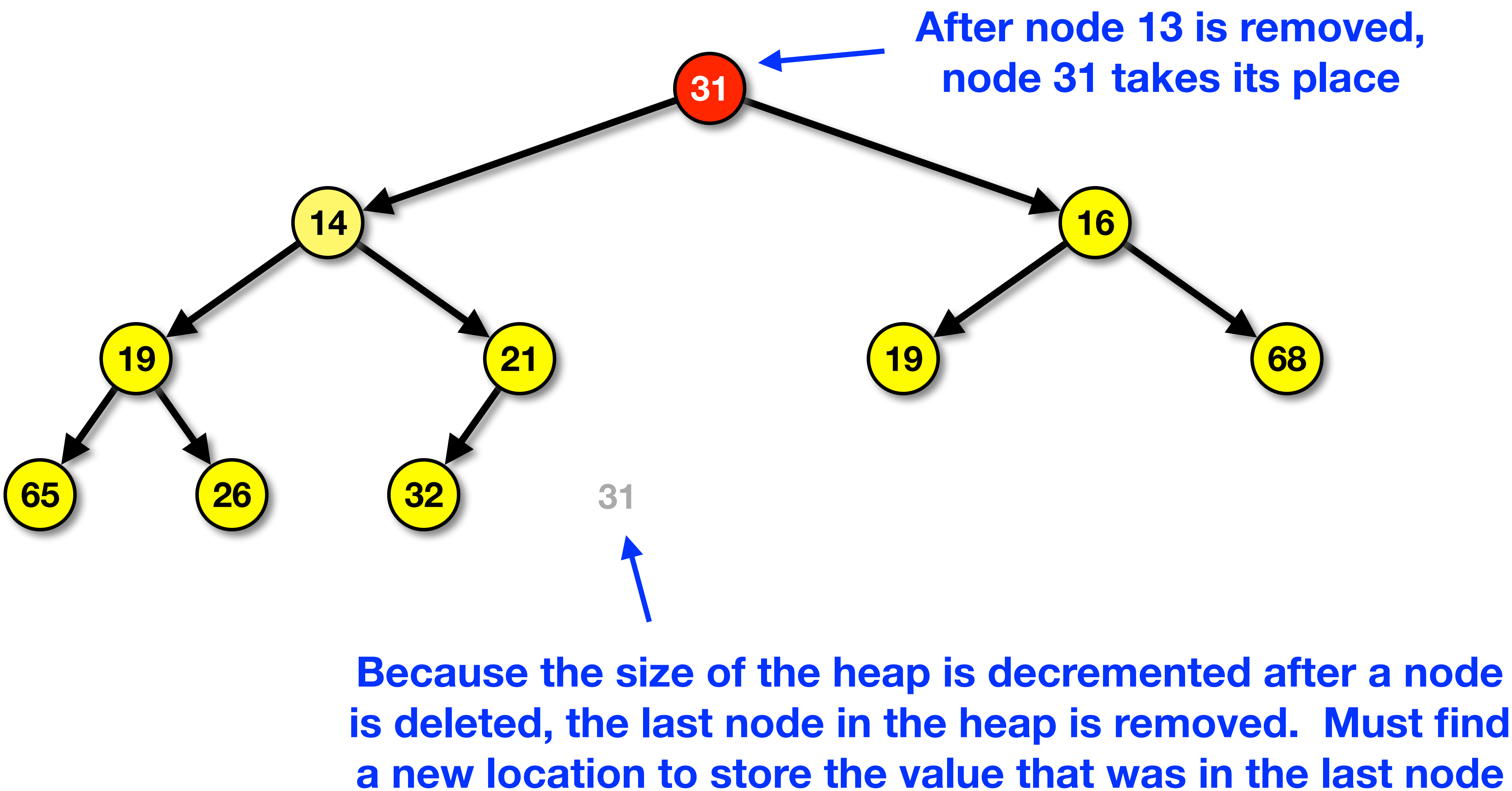
Want to delete the minimum node
The minimum node is node 13

In a minHeap, the minimum node will always be at the top of the heap



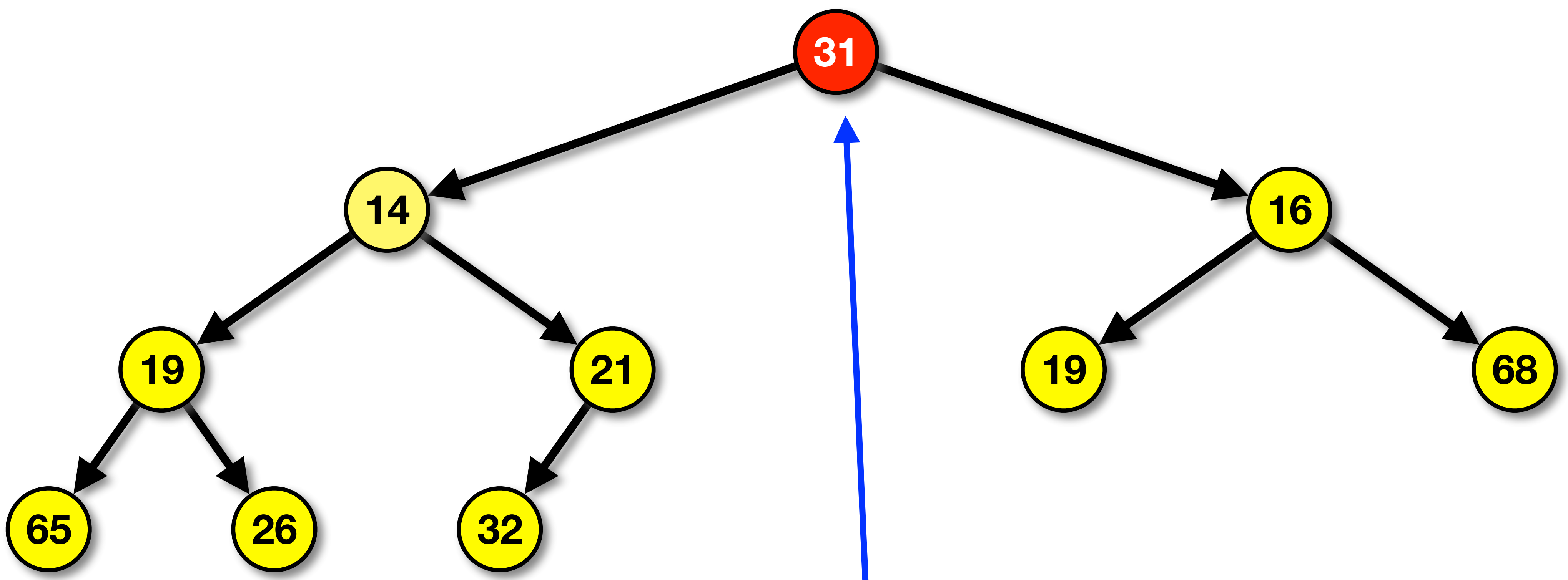
Binary Heap Operations: `deleteMin` Example

Delete the minimum node (the root node)
Replace the old root with the last node in the heap
The size of the heap decreases



Binary Heap Operations: `deleteMin` Example

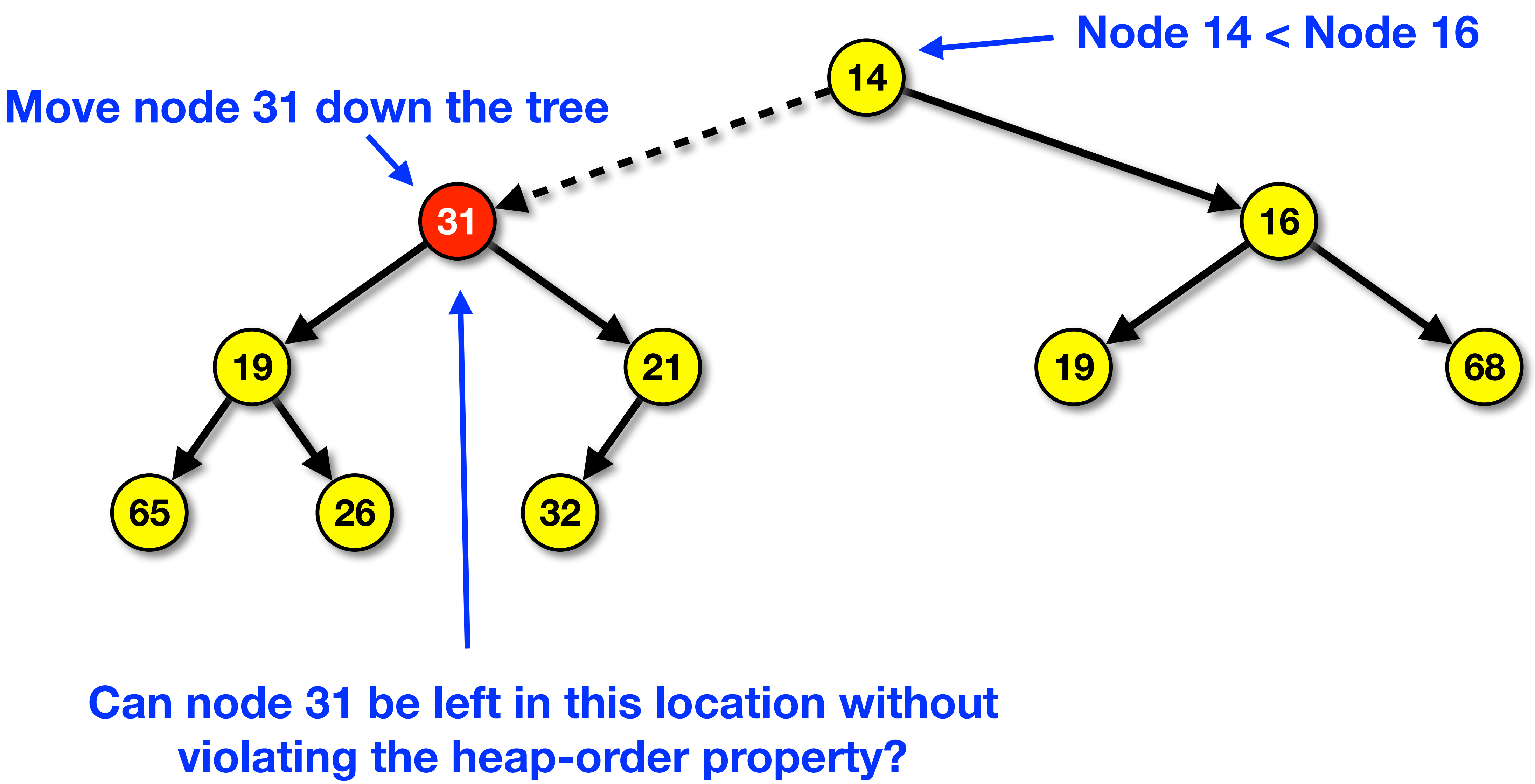
Check for violations of the the heap-order property



Can node 31 be left in this location without violating the heap-order property?

Binary Heap Operations: `deleteMin` Example

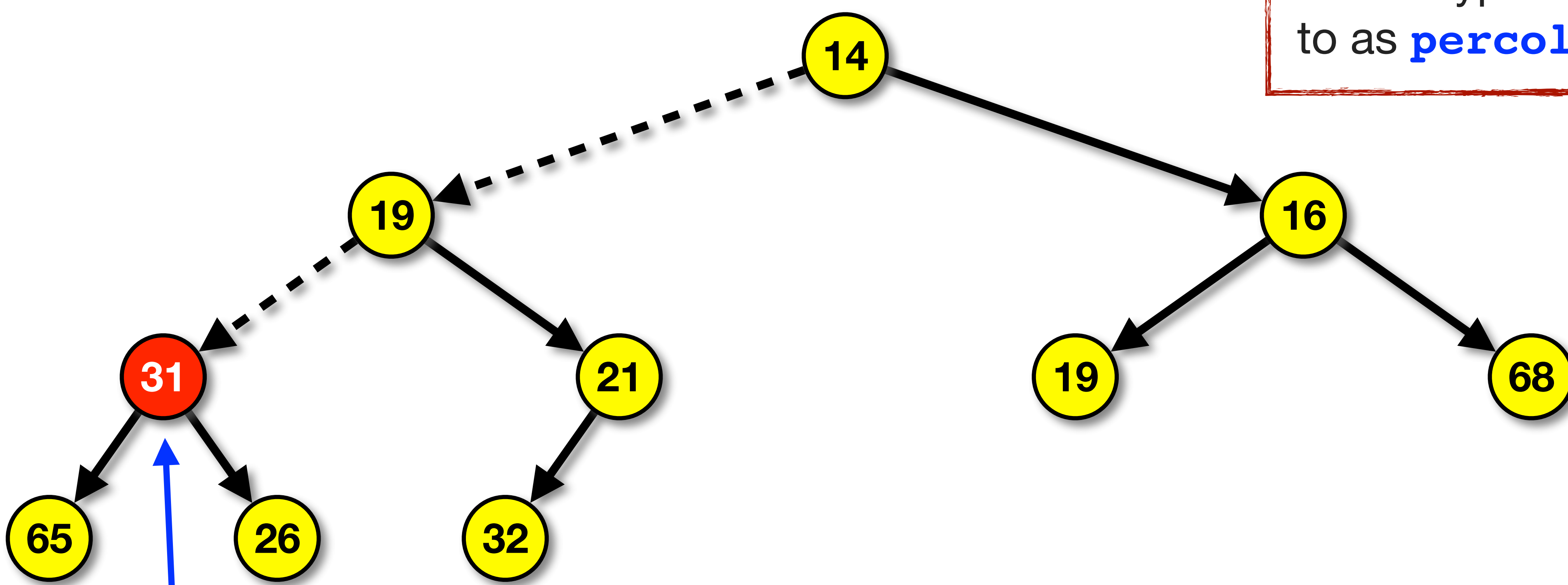
Swap the node with the smaller of its children to move it down the heap



Binary Heap Operations: `deleteMin` Example

Continually swap the node with its smaller child to move it down the heap until a valid location is found that does not violate the heap-order property

This is typically referred to as `percolateDown`

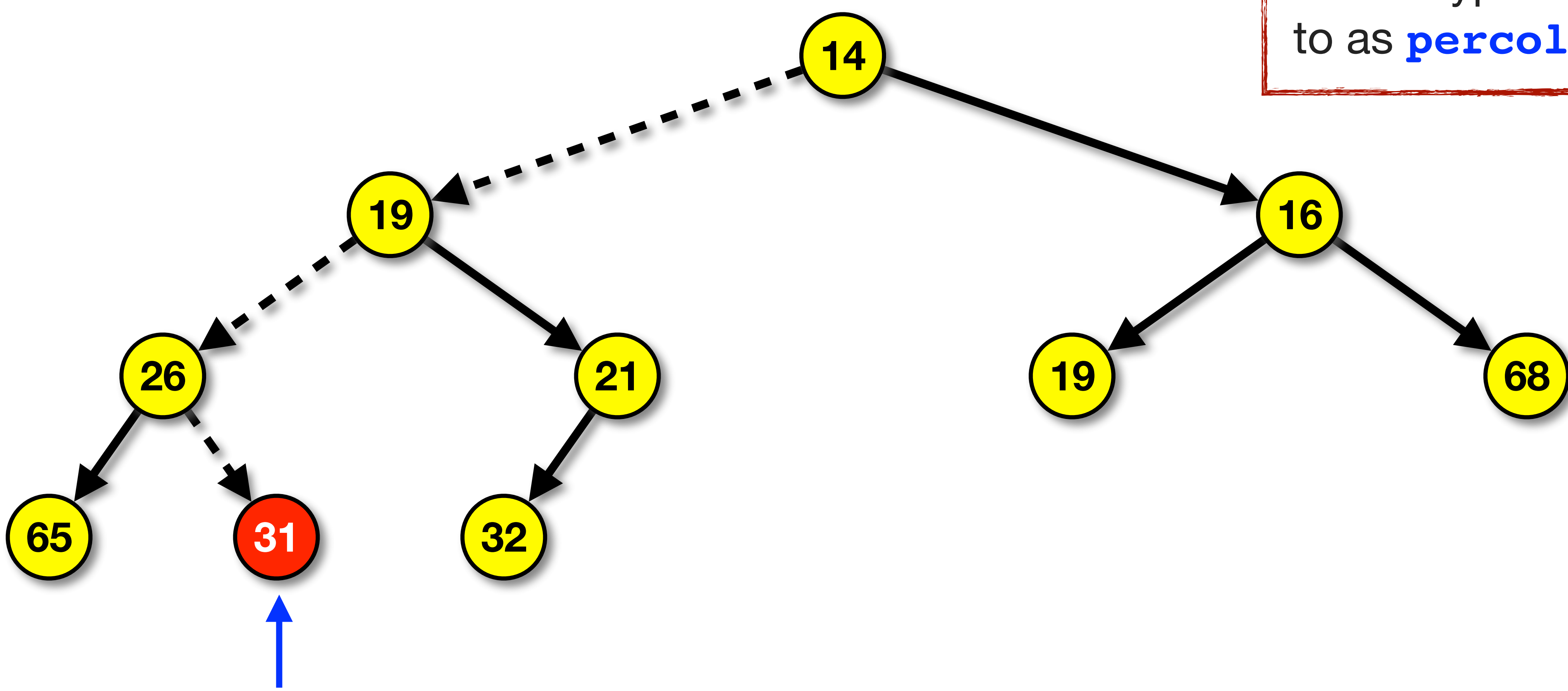


Can node 31 be left in this location without violating the heap-order property?

Binary Heap Operations: `deleteMin` Example

Continually swap the node with its smaller child to move it down the heap until a valid location is found that does not violate the heap-order property

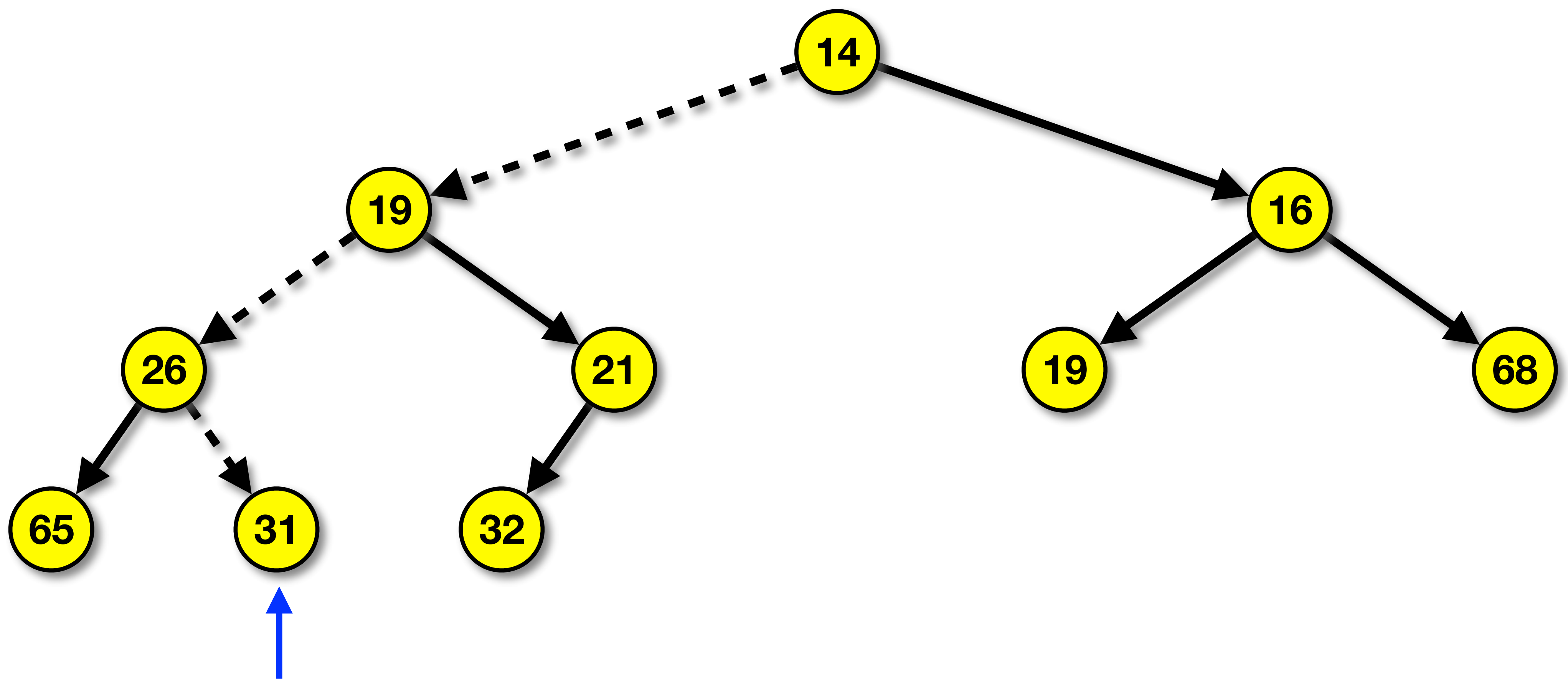
This is typically referred to as `percolateDown`



Can node 31 be left in this location without violating the heap-order property?

Binary Heap Operations: `deleteMin` Example

Once a location is found for node 31, the operation is done



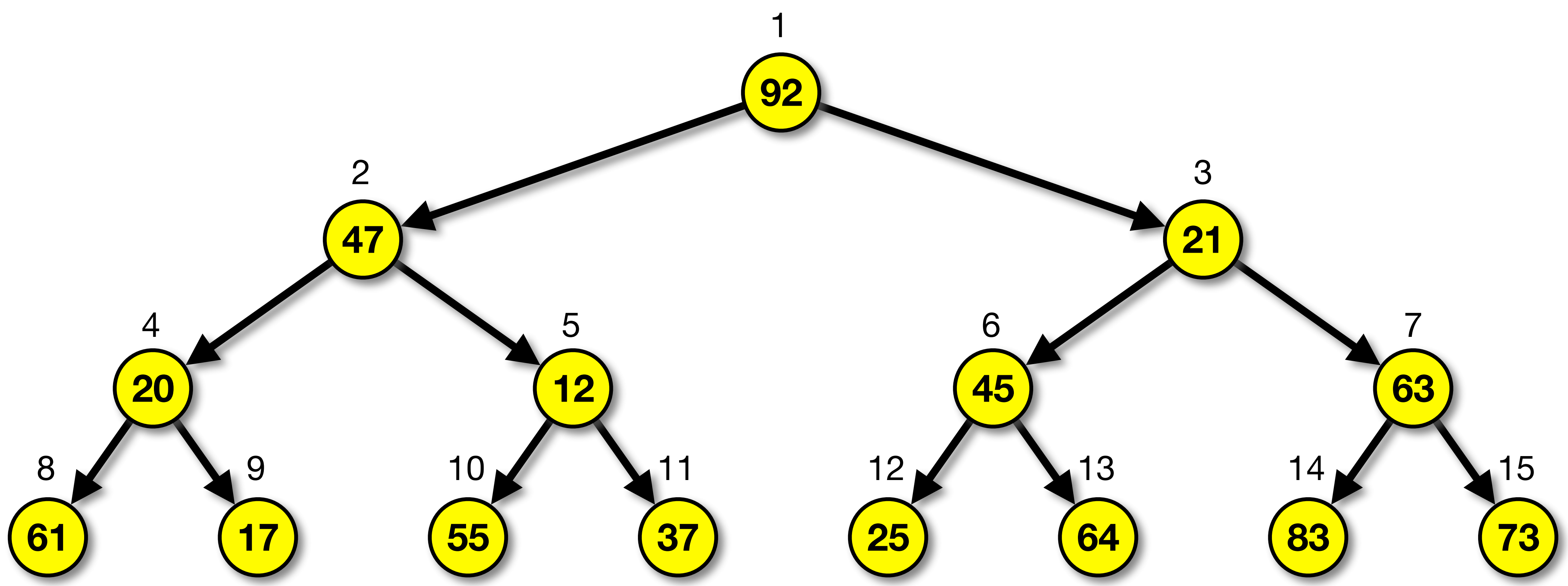
No violations of the heap-order property

Binary Heap Operations: `buildHeap`

- **The `buildHeap` operation takes a tree that is not in heap-order and puts it into heap-order**
- **Idea:**
  - Call `percolateDown` on all non-leaf nodes in reverse level-order
    - No need to call `percolateDown` on leaf nodes since they cannot be moved down
    - The first non-leaf node is located in the array index $\lfloor \texttt{currentSize}/2 \rfloor$
    - Easily implemented by iteratively visiting each node in the heap array in reverse order starting at the first non-leaf node

Binary Heap Operations: `buildHeap` Example

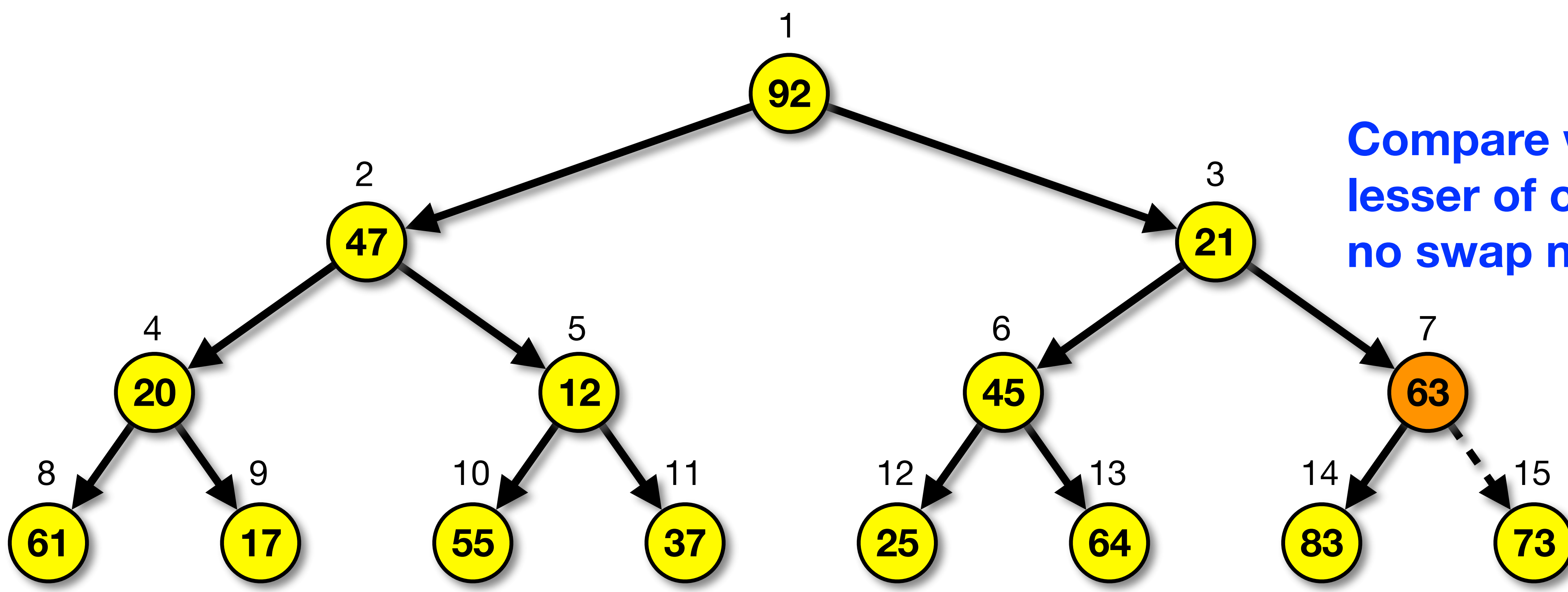
Fixing the heap order
No need to call percolateDown on the leaf nodes



Binary Heap Operations: **`buildHeap`** Example

Heap size is 15, so first non-leaf node is ⌊15/2⌋ = 7

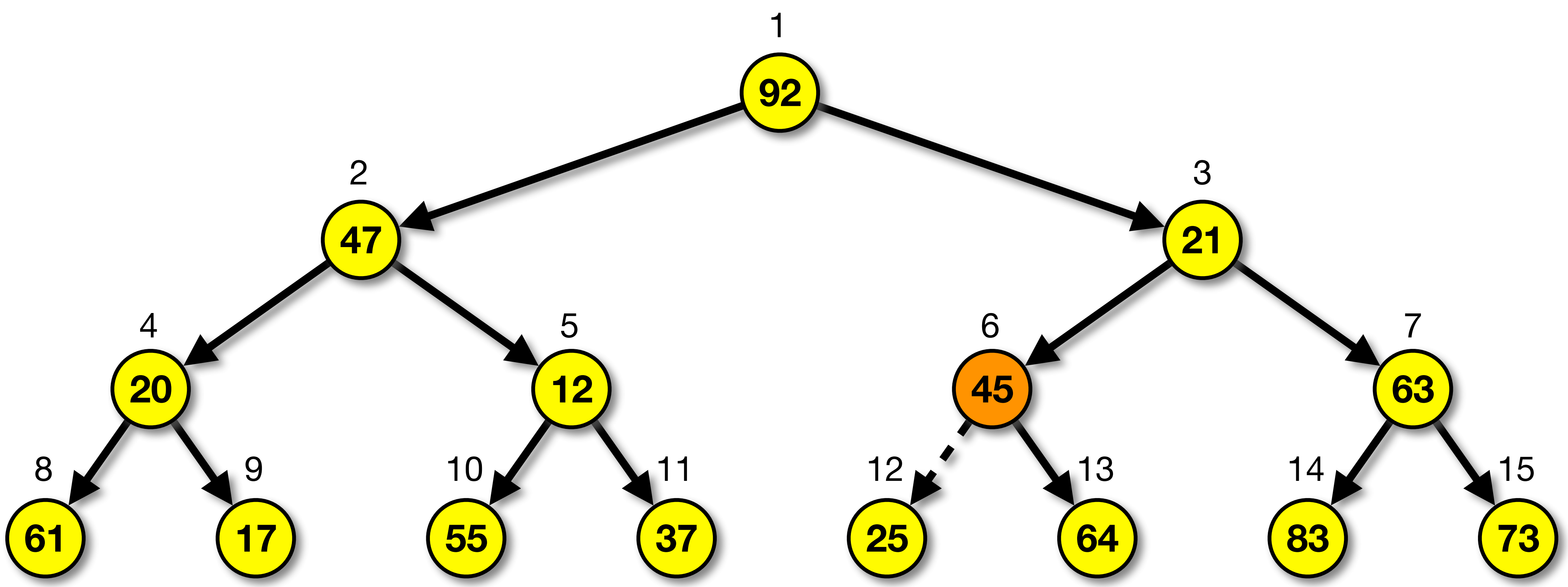
Processing node at array index 7



Compare with lesser of children, no swap needed

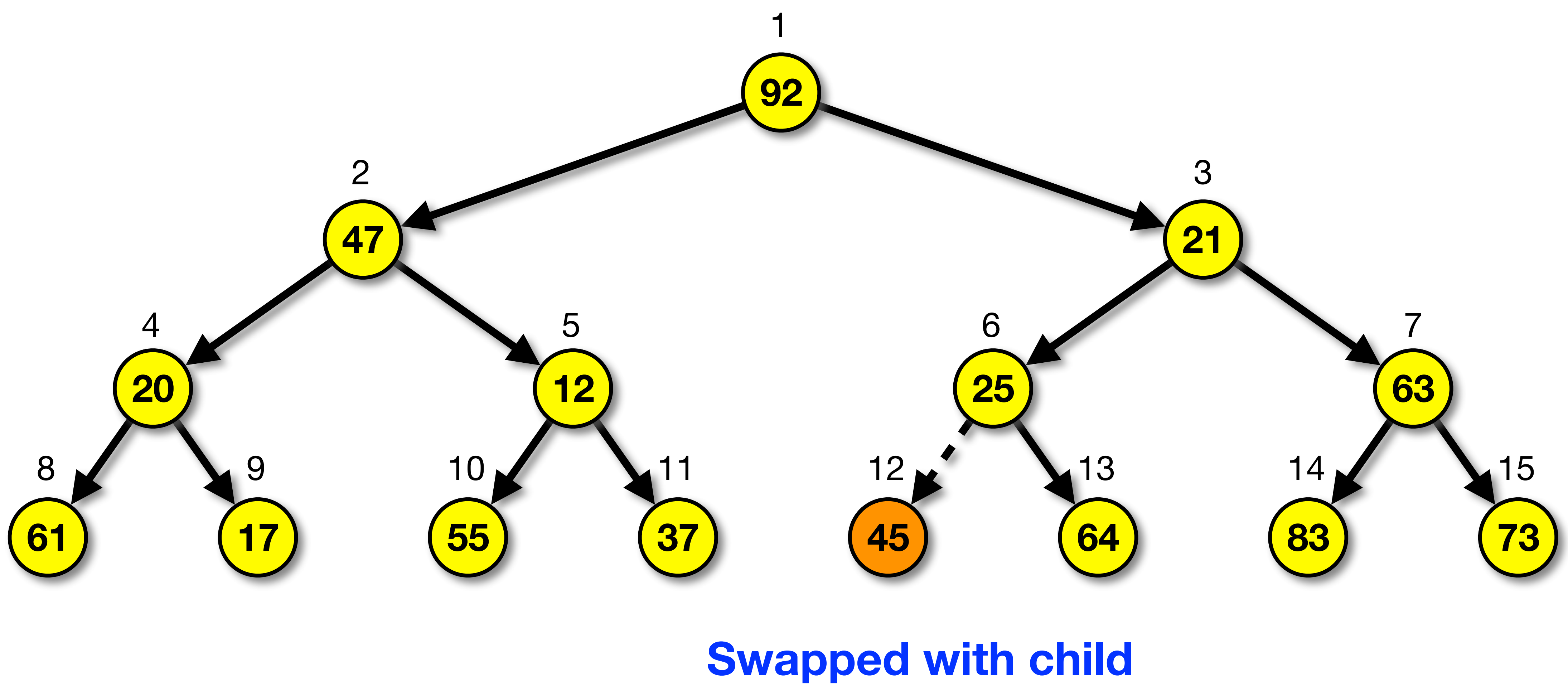
Binary Heap Operations: `buildHeap` Example

Processing node at array index 6



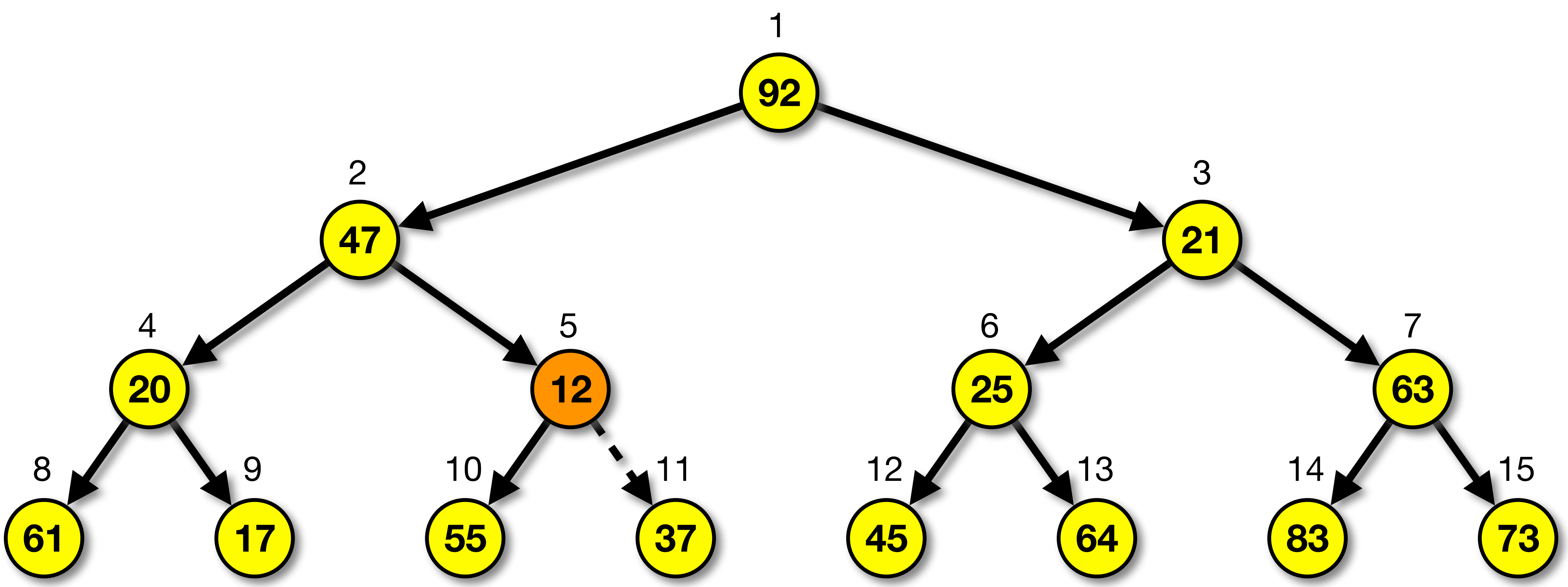
Compare with lesser of children, need to swap

Binary Heap Operations: `buildHeap` Example



Binary Heap Operations: `buildHeap` Example

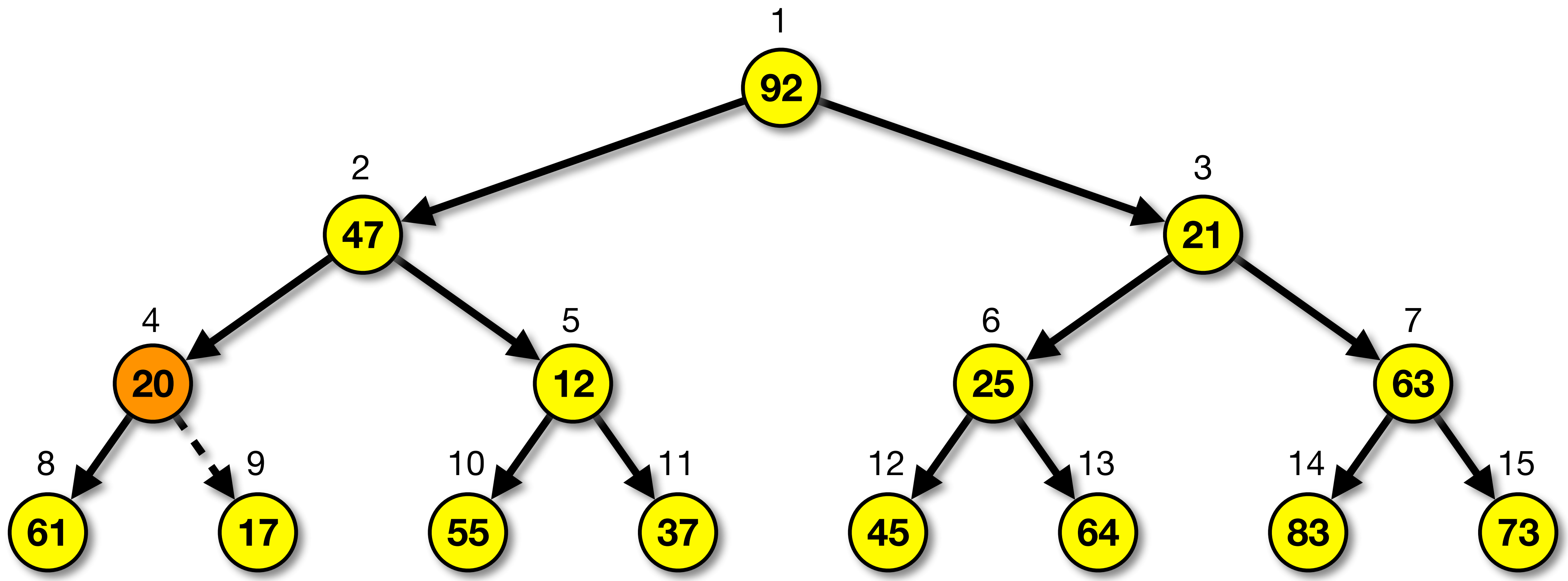
Processing node at array index 5



Compare with lesser of children, no swap needed

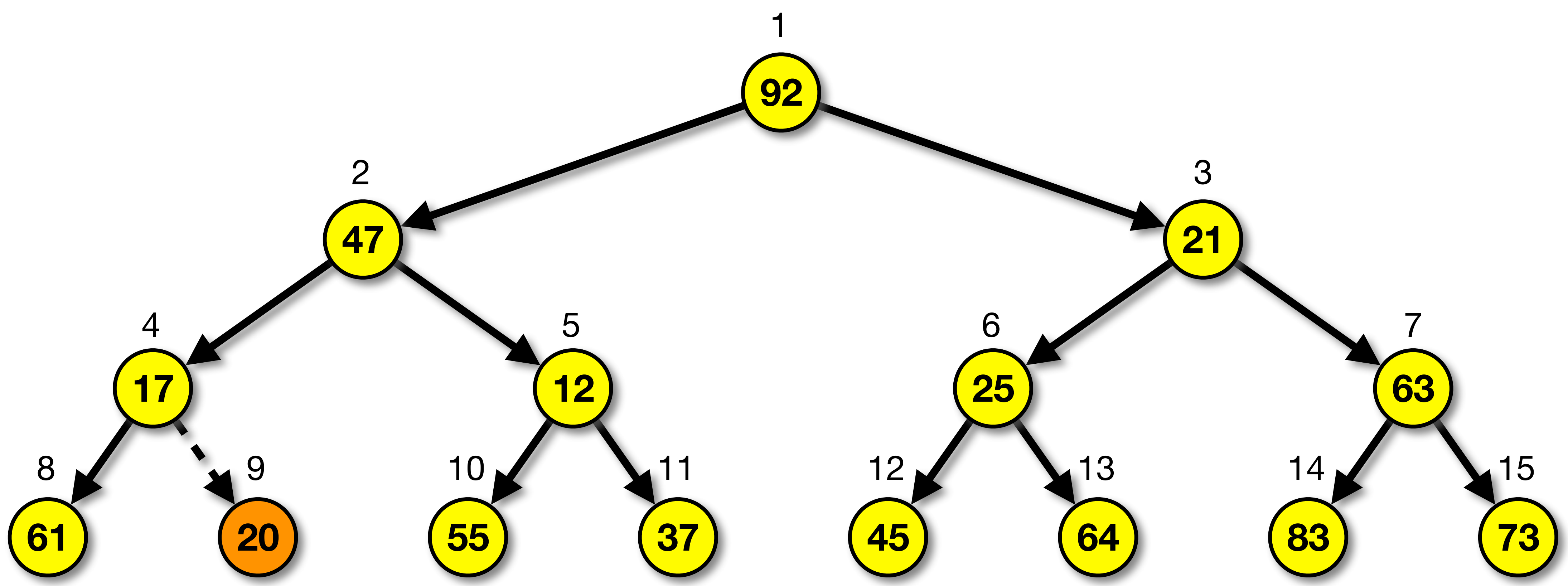
Binary Heap Operations: `buildHeap` Example

Processing node at array index 4



Compare with lesser of children, need to swap

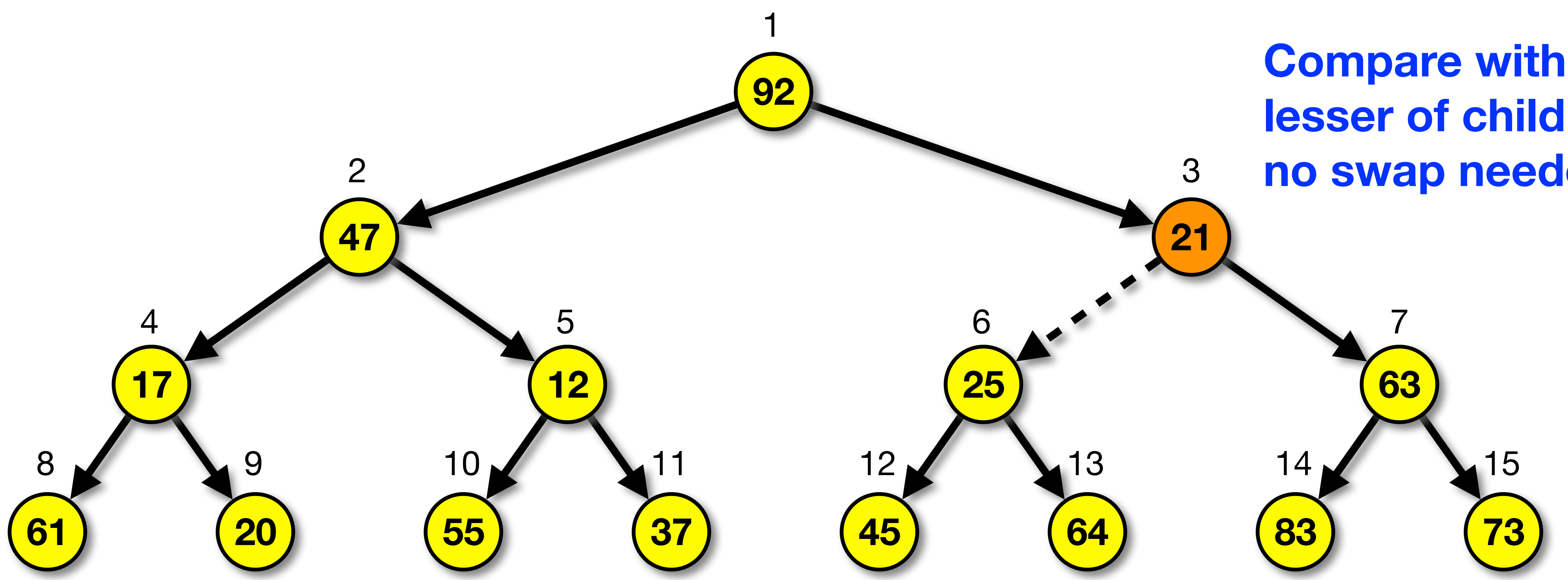
Binary Heap Operations: **`buildHeap`** Example



Swapped with child

Binary Heap Operations: `buildHeap` Example

Processing node at array index 3

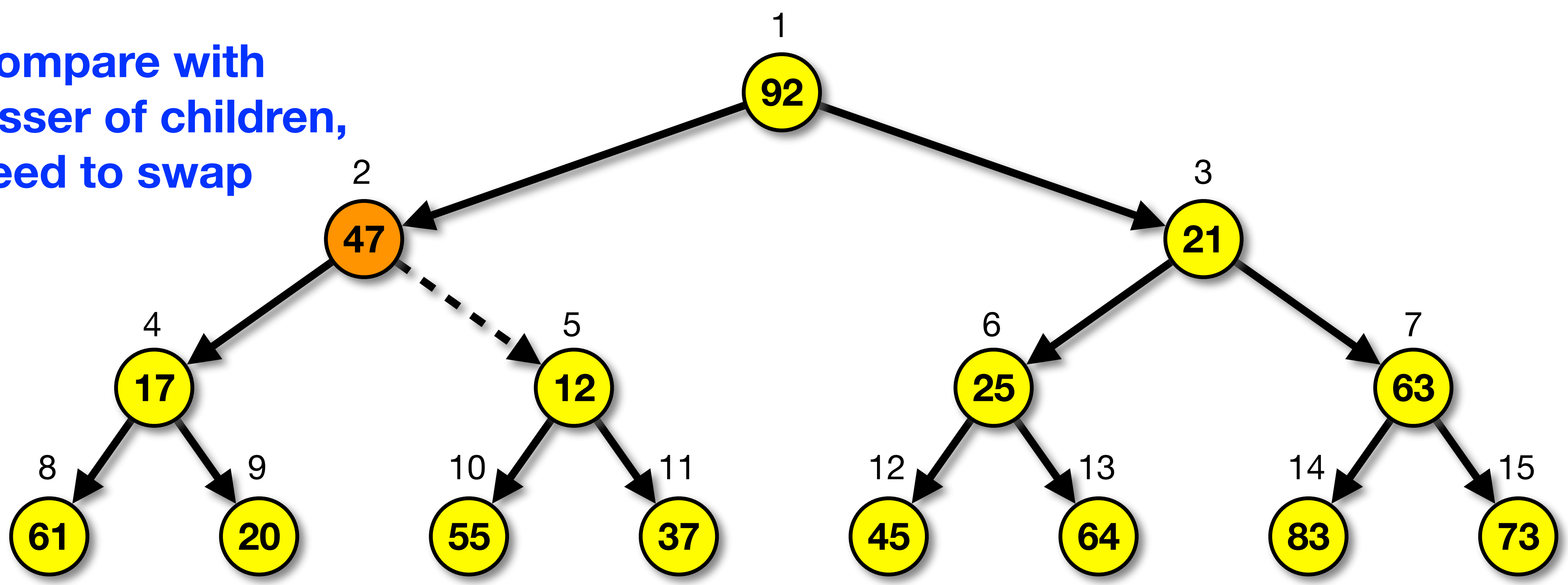


Compare with lesser of children, no swap needed

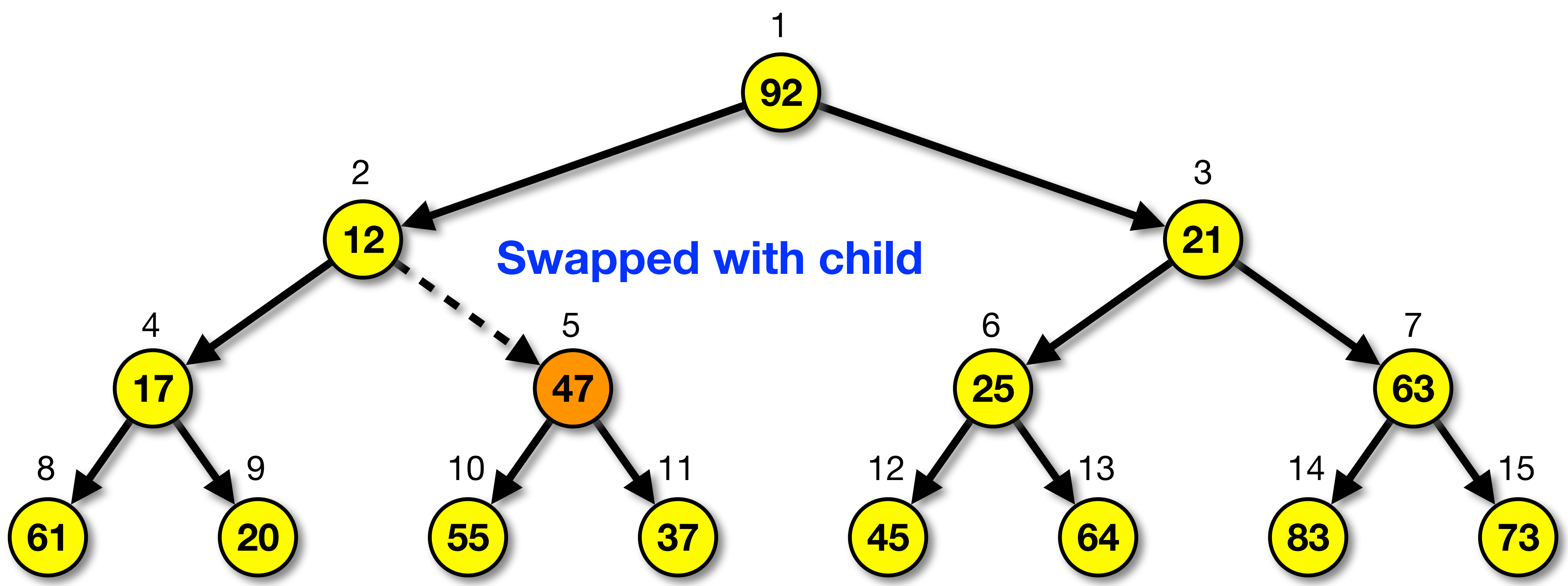
Binary Heap Operations: `buildHeap` Example

Processing node at array index 2

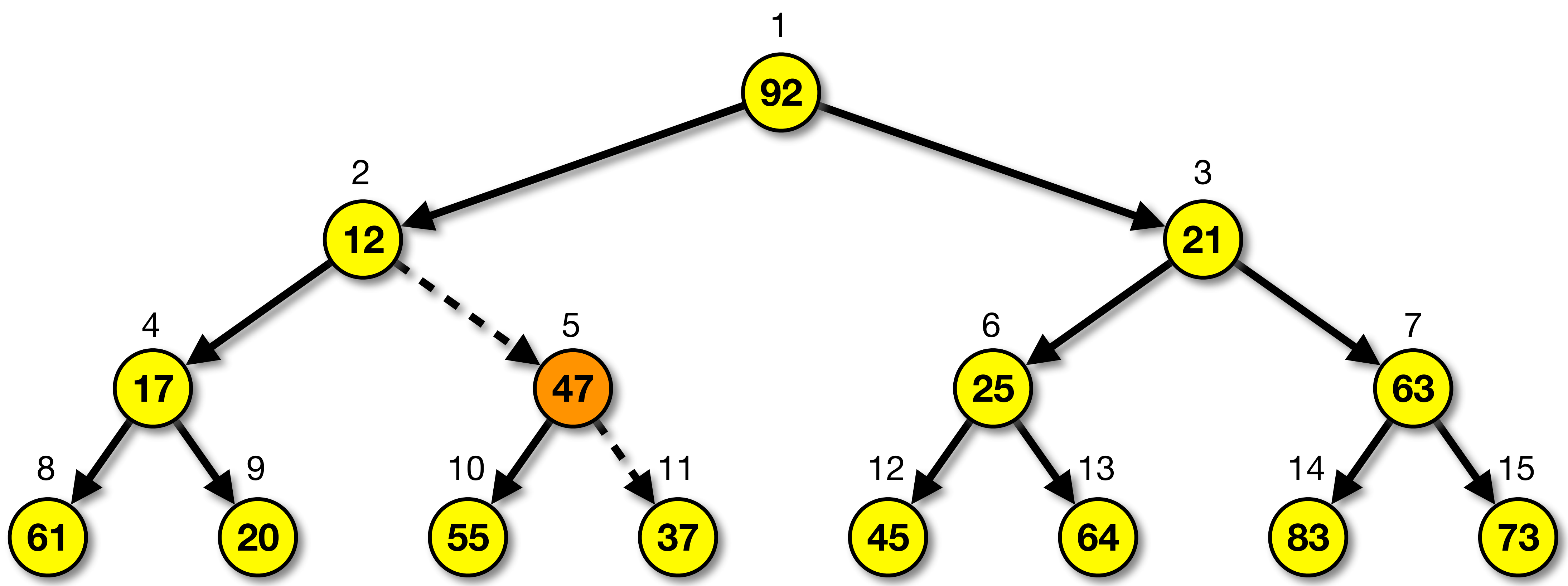
Compare with lesser of children, need to swap



Binary Heap Operations: `buildHeap` Example

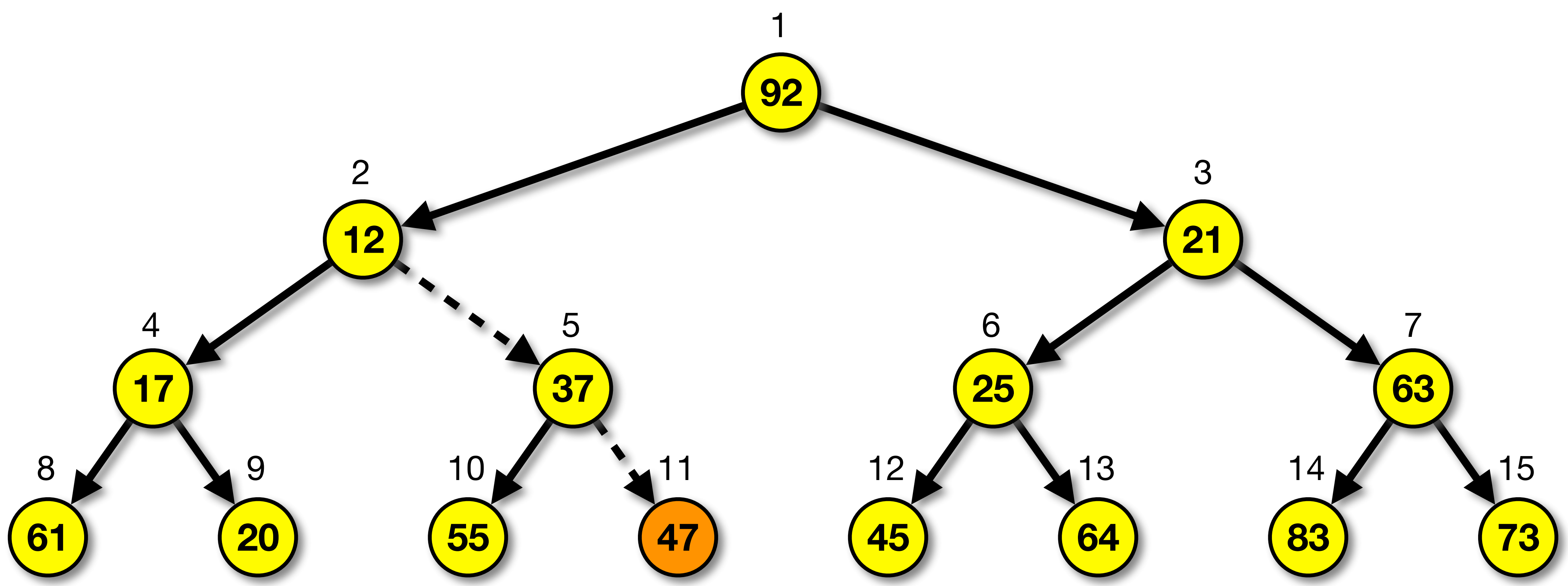


Binary Heap Operations: `buildHeap` Example



Continue to compare with lesser of children, Need another swap

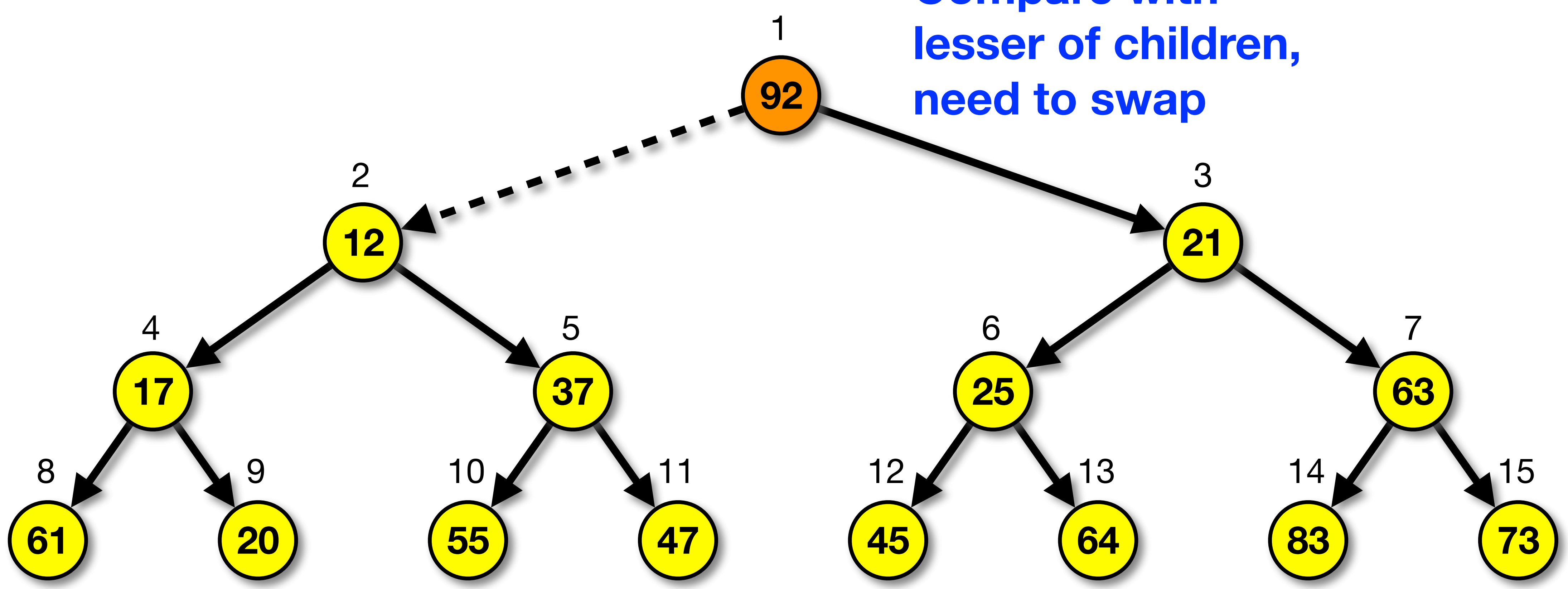
Binary Heap Operations: `buildHeap` Example



Swapped with child

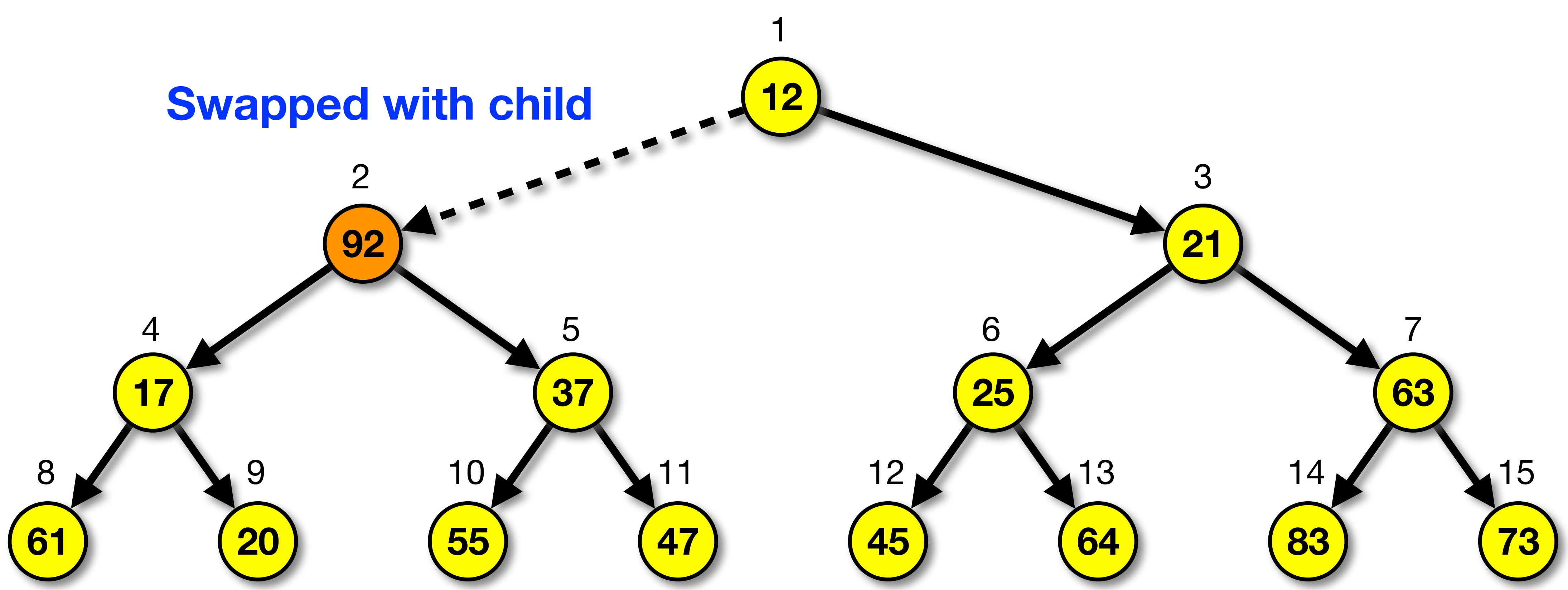
Binary Heap Operations: `buildHeap` Example

Processing node at array index 1



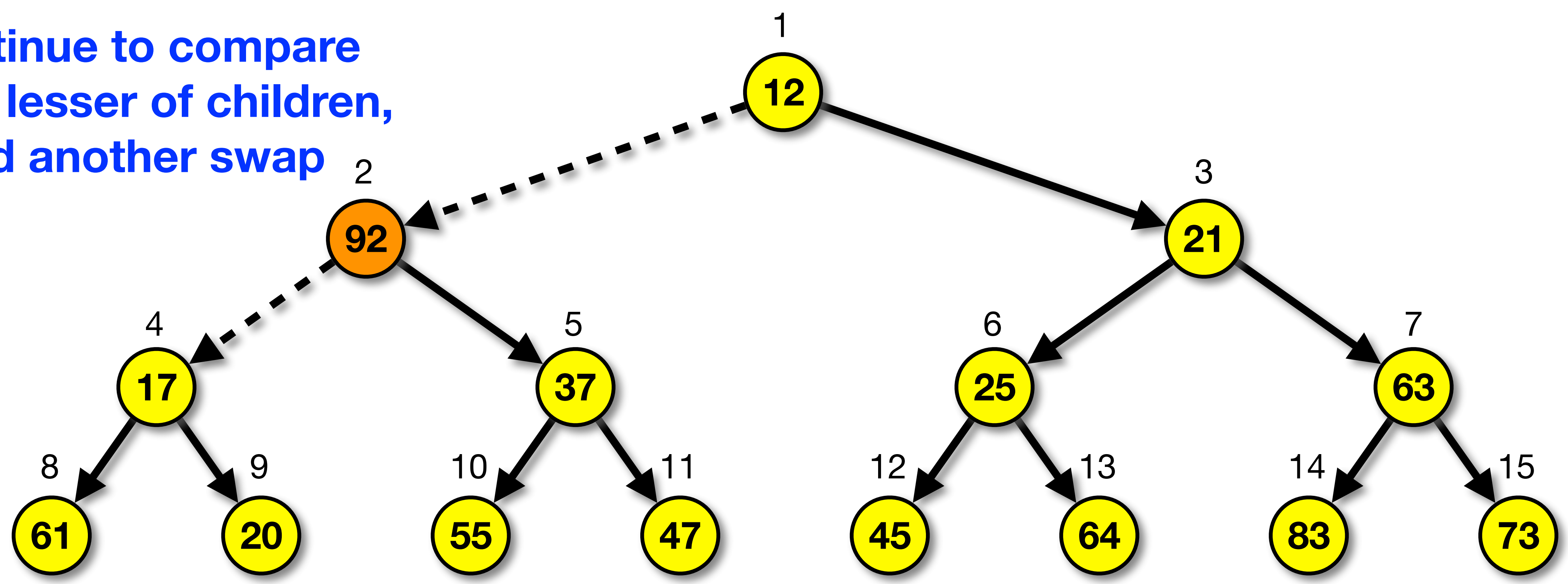
Compare with lesser of children, need to swap

Binary Heap Operations: `buildHeap` Example



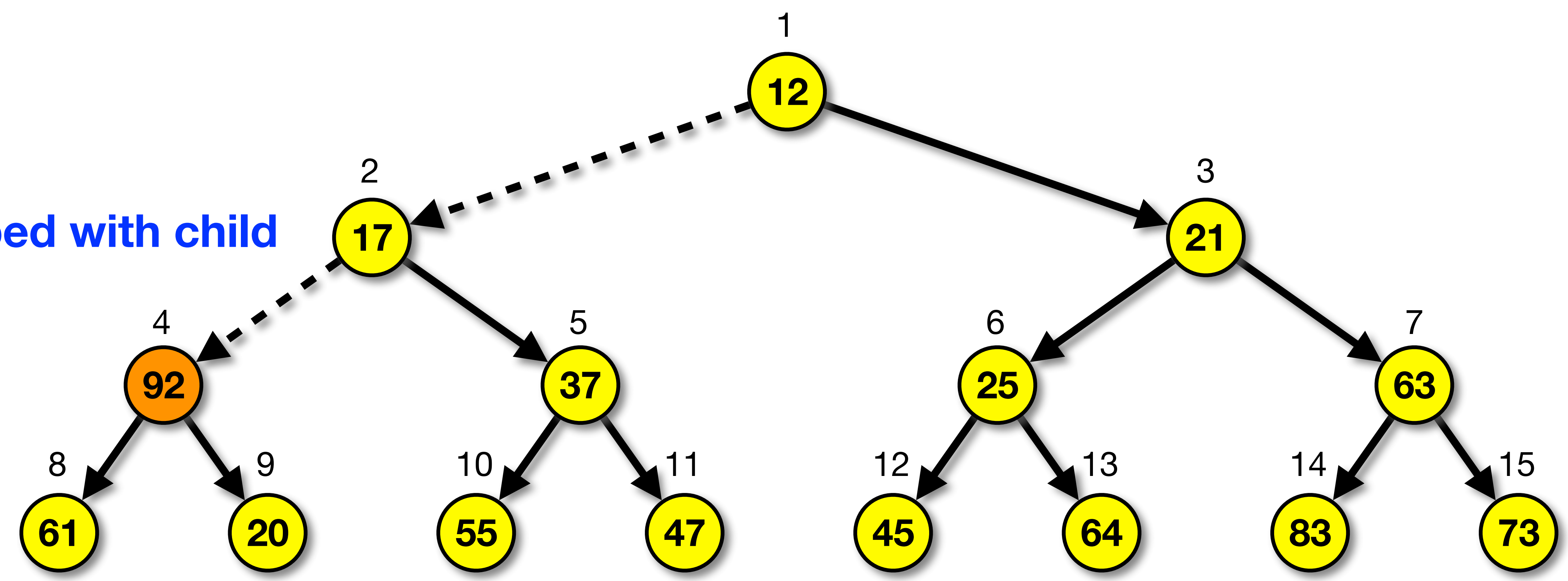
Binary Heap Operations: `buildHeap` Example

Continue to compare with lesser of children, Need another swap

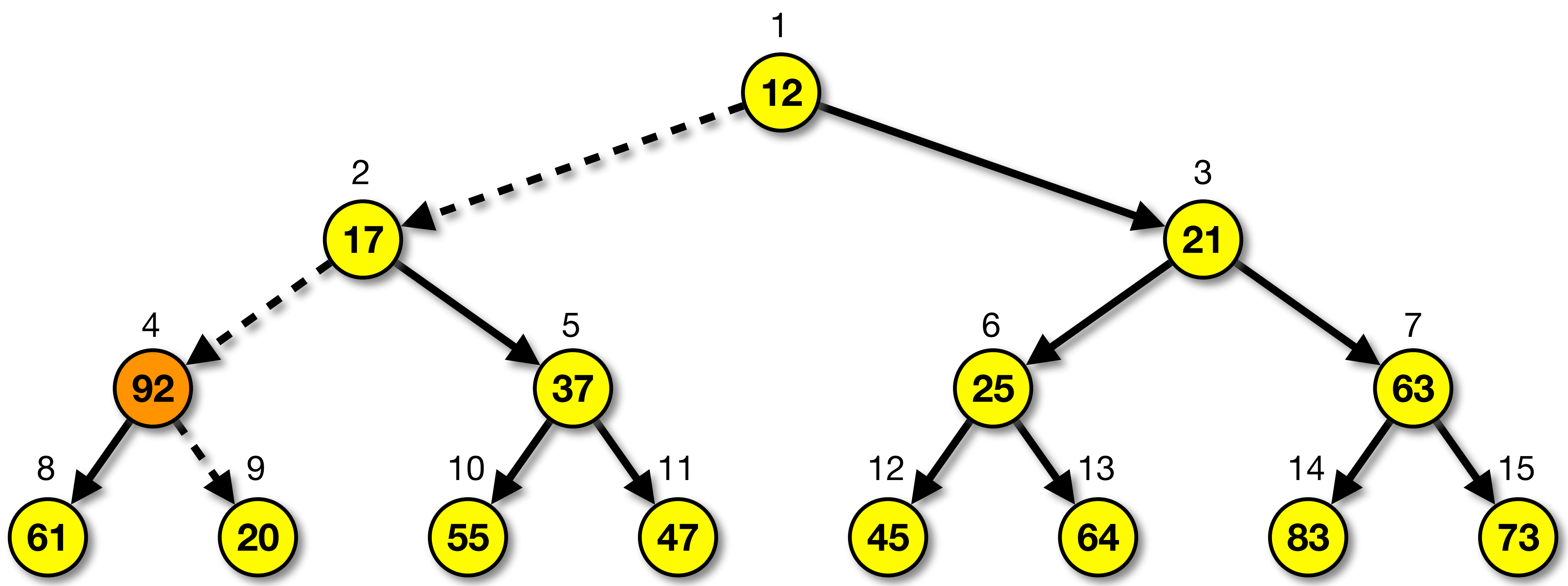


Binary Heap Operations: `buildHeap` Example

Swapped with child

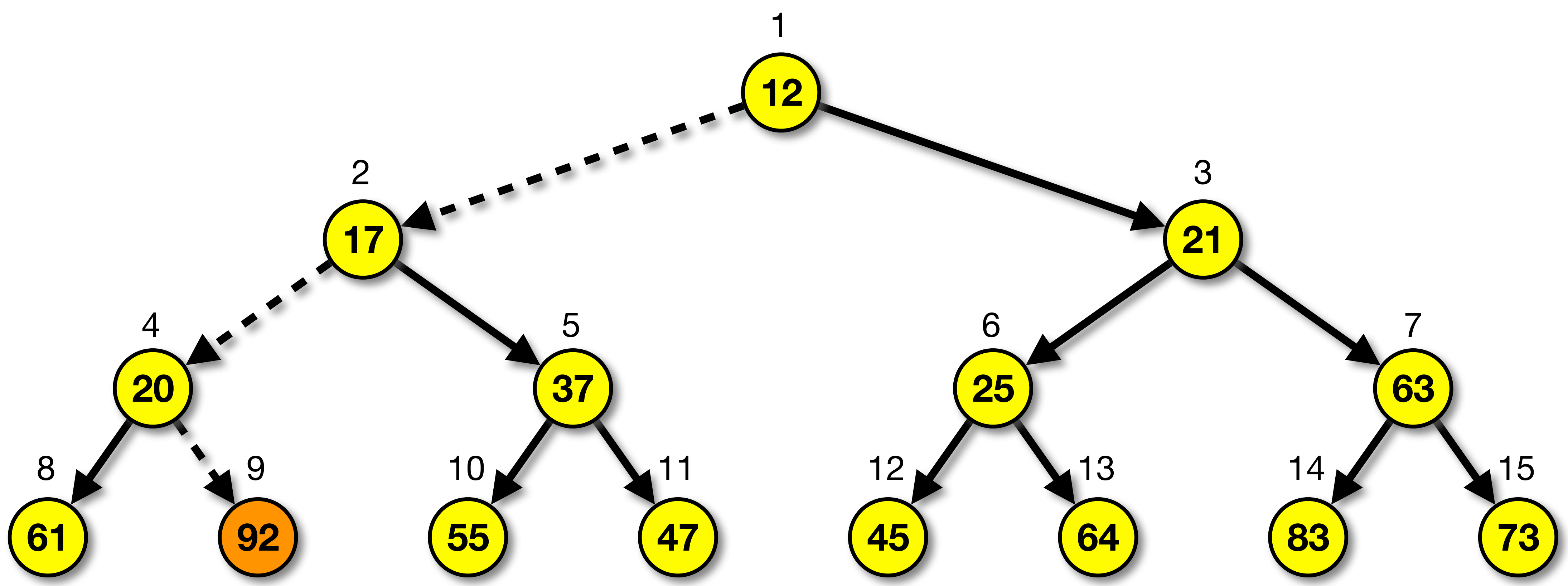


Binary Heap Operations: `buildHeap` Example



Continue to compare with lesser of children, Need another swap

Binary Heap Operations: `buildHeap` Example



Swapped with child

Binary Heap Operations: `buildHeap` Example

Fixing the heap order
Now in heap order