CS350: Data Structures

Heaps and Priority Queues

James Moscola
Department of Physical Sciences
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**

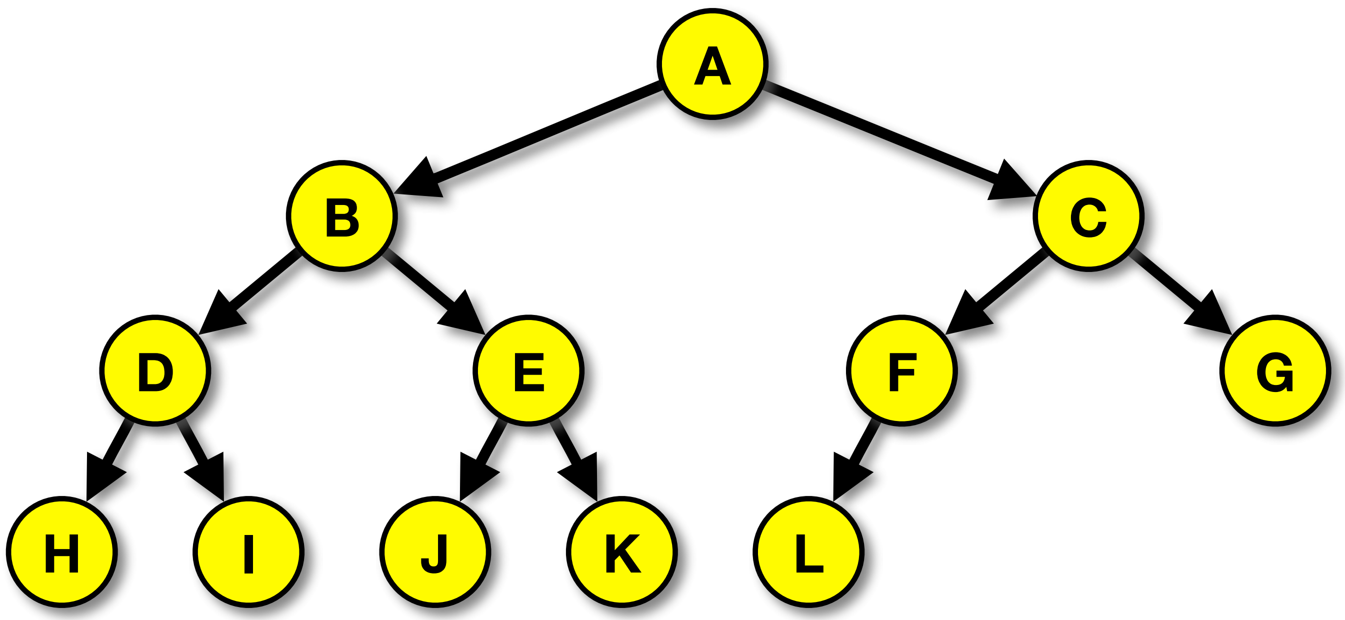
- **A binary heap can be either a Min Heap or a Max Heap**
    - Min Heap - smallest key values have highest priority
    - Max Heap - largest key values have highest priority

Complete Binary Tree

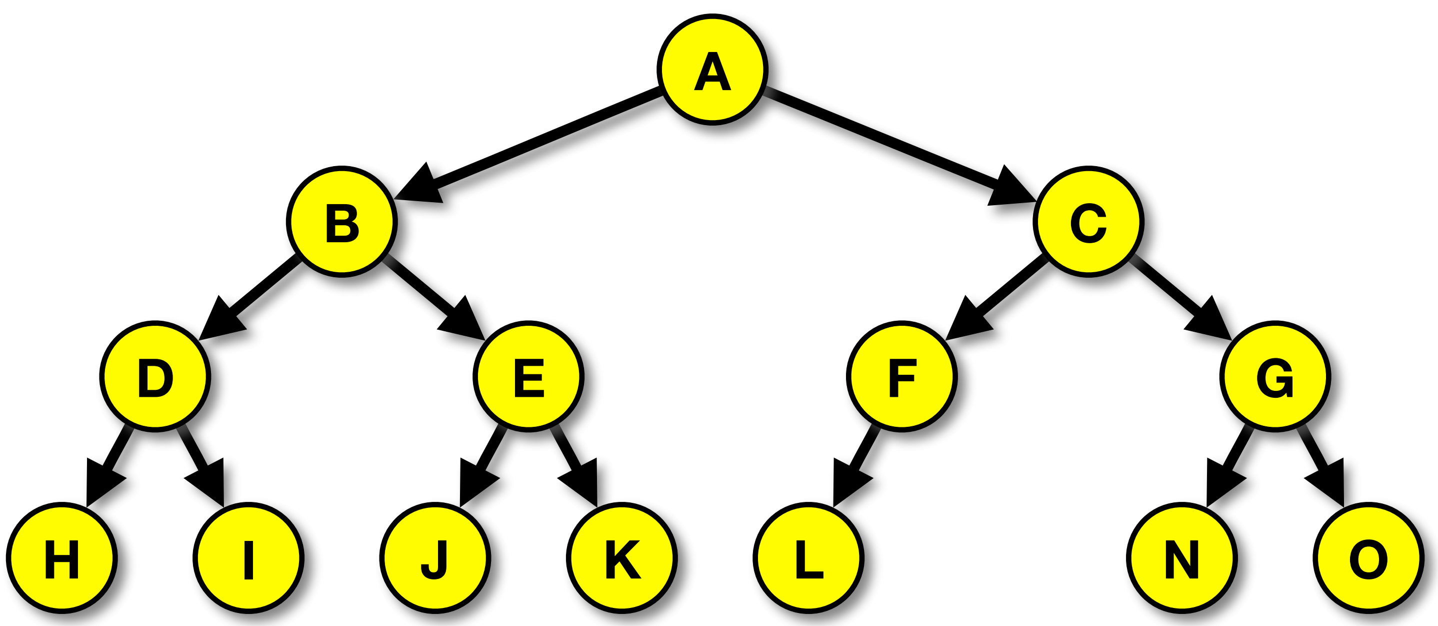
- **A heap is implemented as a *complete binary tree* which can be stored efficiently in an array**

- **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



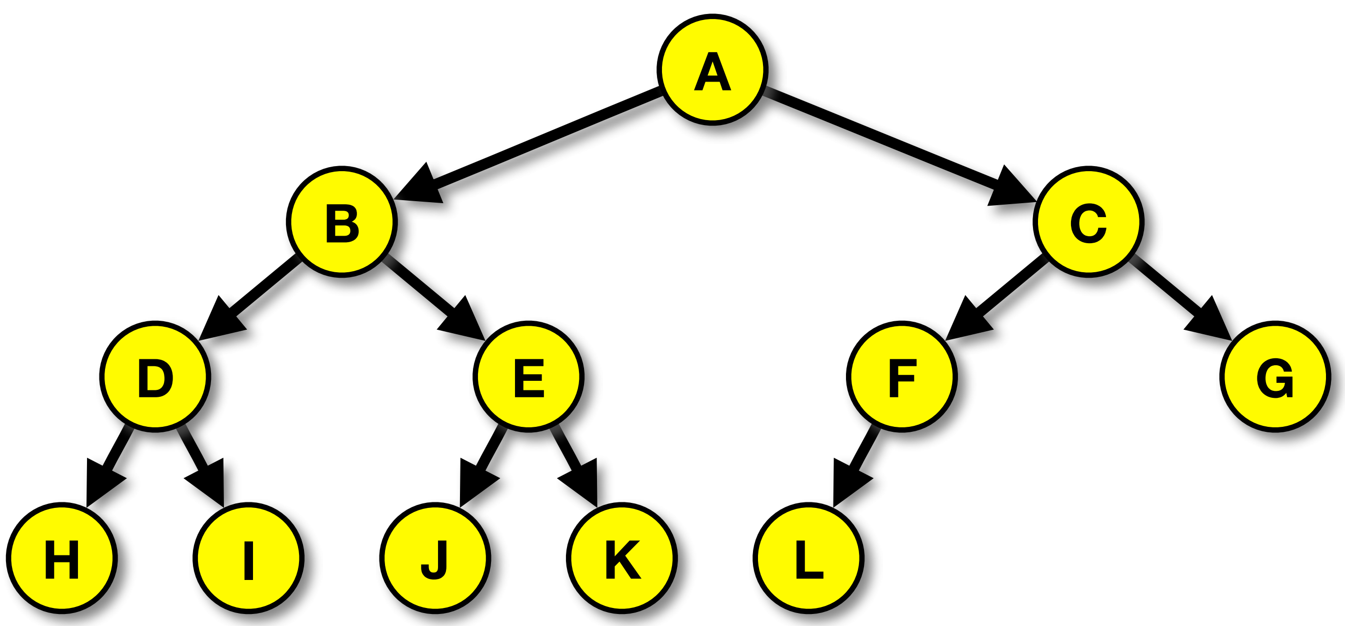
This IS a *complete binary tree*



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**



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

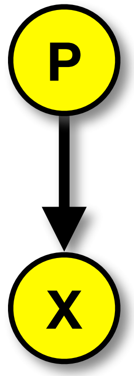
left child = 2 * i
right child = 2 * i + 1

parent = $\lfloor i / 2 \rfloor$

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)



$P \leq X$

Binary Heap Operations: insert

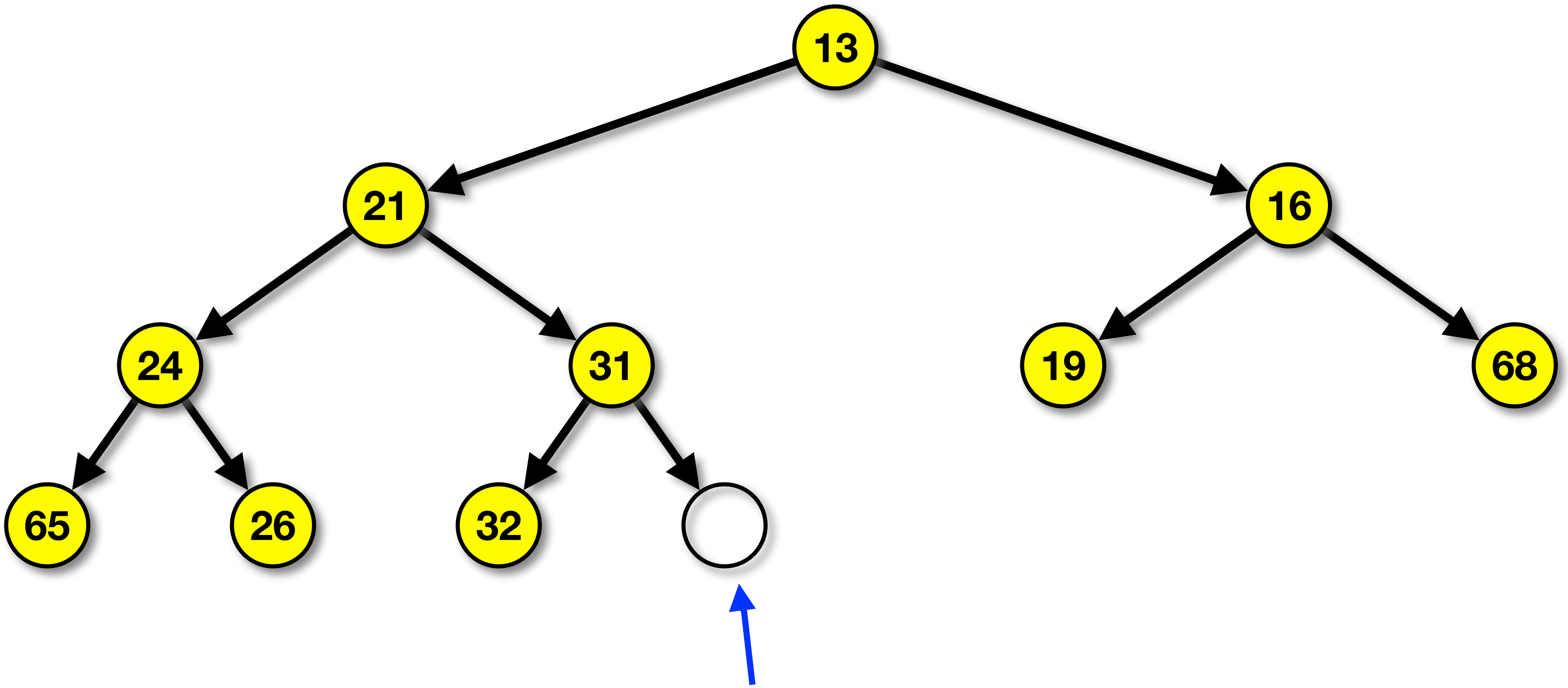
- **The basic idea for insertion:**
  - Create a ‘hole’ (an empty node) in the next available complete tree location
  - If the new element can be inserted into the hole without violating the heap-order property, then insert the new element into the hole
  - If inserting the new element into the hole will violate the heap-order property, then move the hole up the heap until a location is found where the new element can be inserted without violating 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**

Binary Heap Operations: insert Example

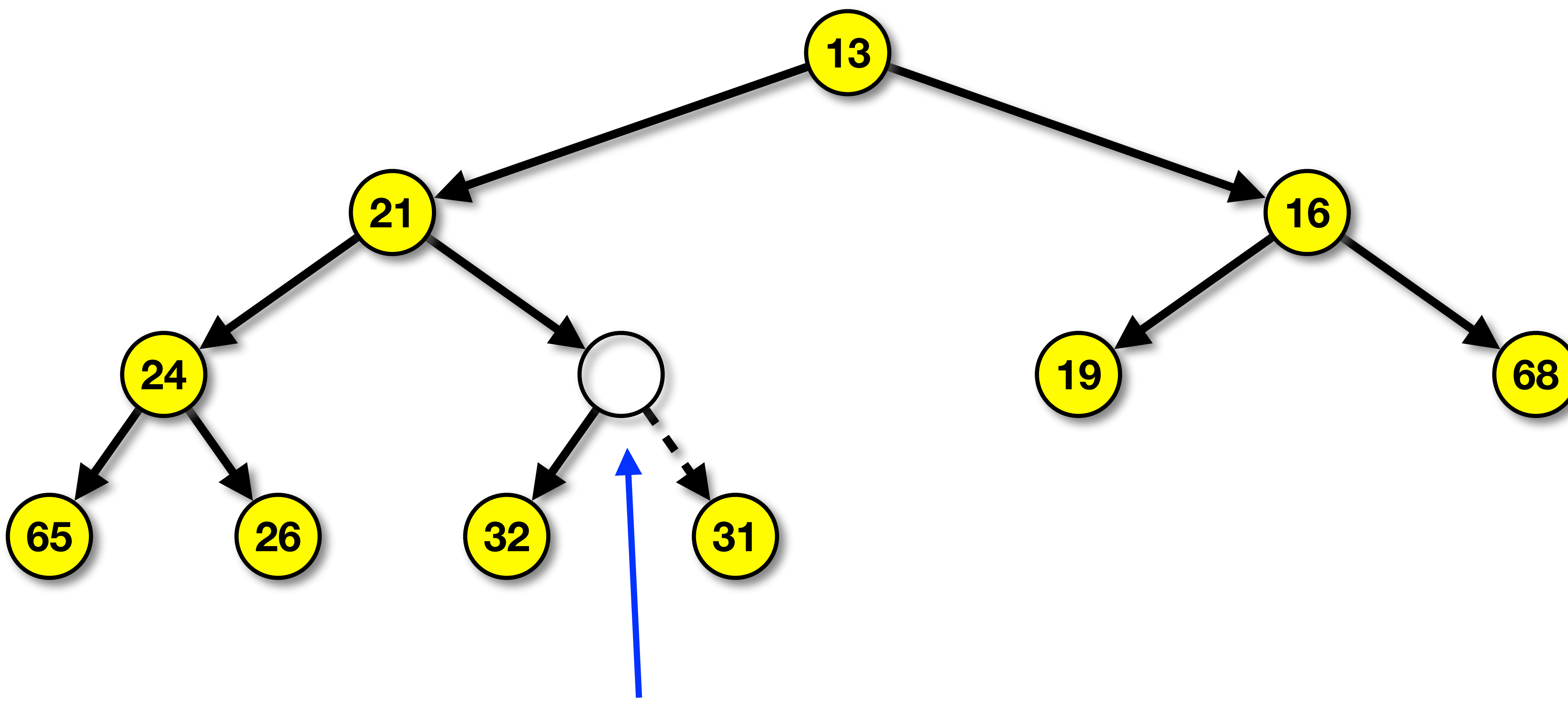
Insert node 14
First create a hole in the next available heap location



Can 14 be inserted into new hole without violating heap the property?

Binary Heap Operations: insert Example

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

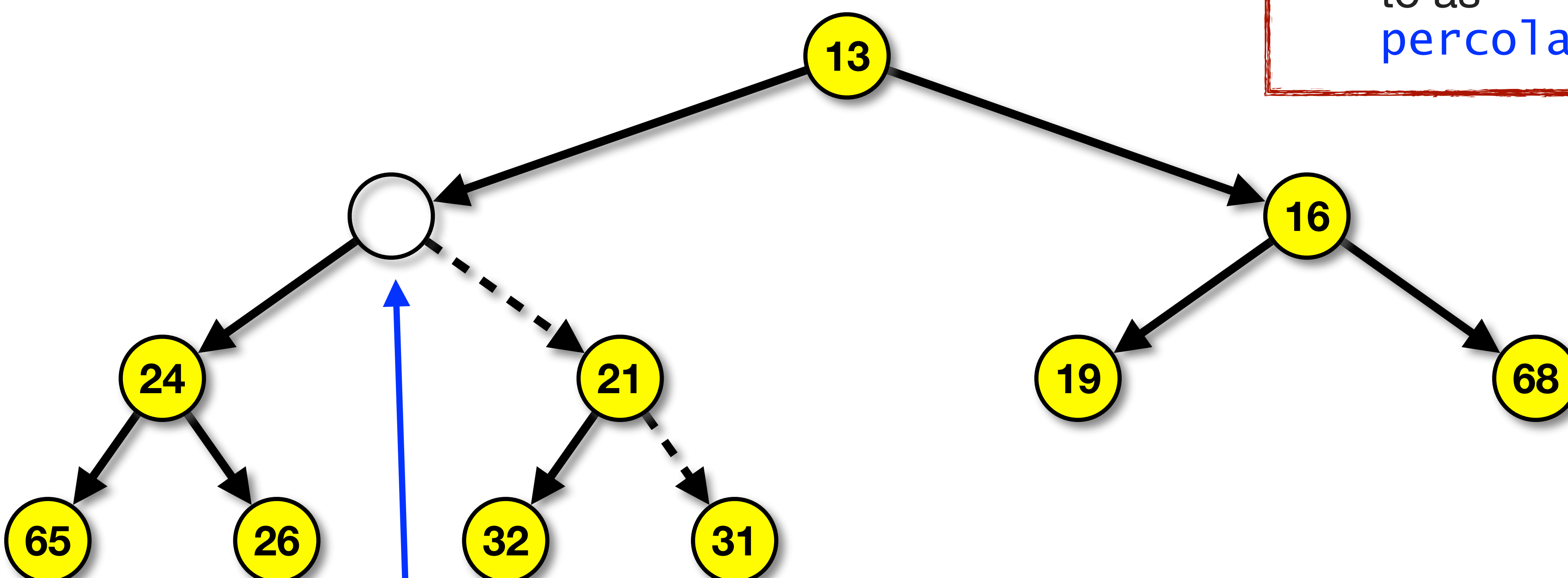


Can 14 be inserted into this location without violating heap the property?

Binary Heap Operations: insert Example

Insert node 14
Continually swap the hole with its parent to move the hole up the heap until a valid location is found to insert the new node

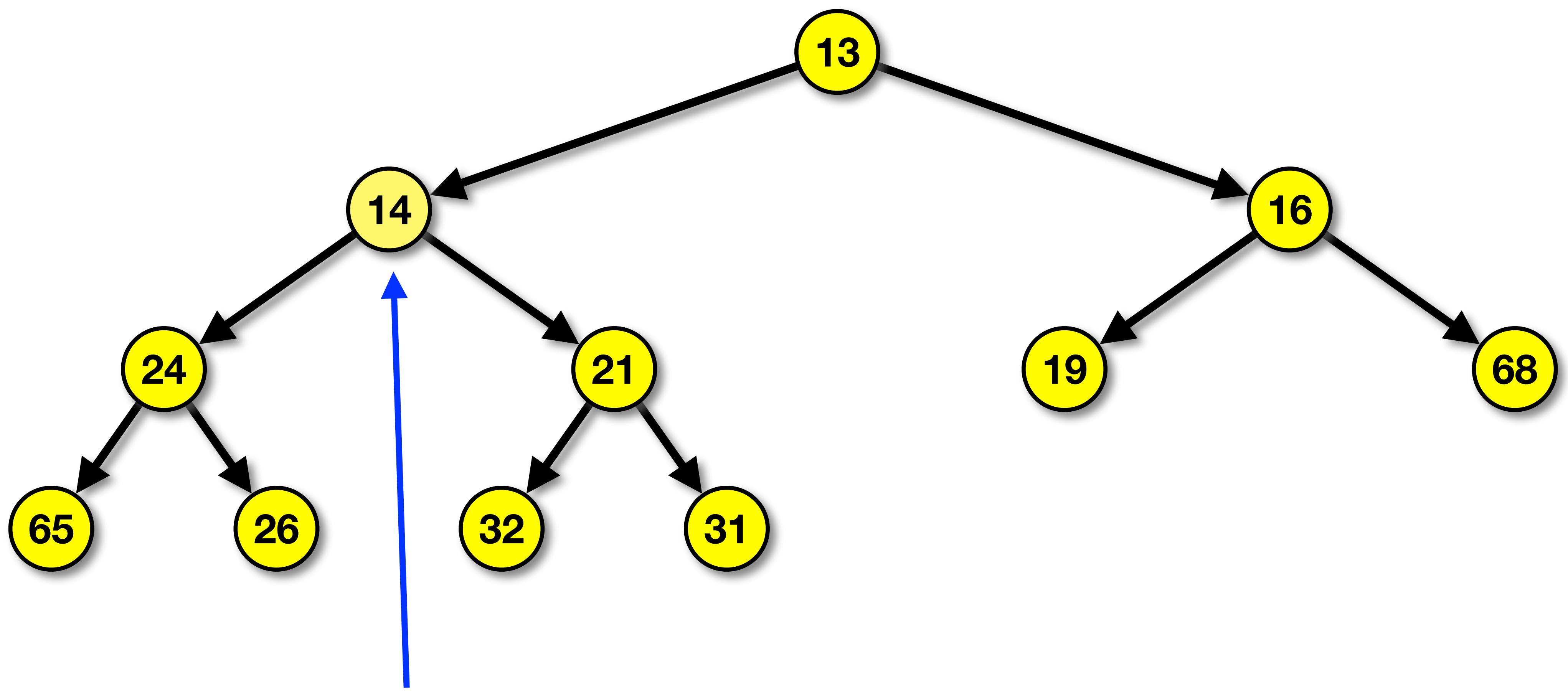
This is typically referred to as `percolateUp`



Can 14 be inserted into this location without violating heap the property?

Binary Heap Operations: insert Example

Insert node 14
Done with insertion



Insert node 14 into 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)
  - Create a ‘hole’ in the location where the root node was ... this hole will eventually be filled with the last node in the heap (the rightmost node on the bottom level)
  - Move the hole down the tree until a location is found that permits the last node in the heap to get moved while still maintaining the heap-order property
    - This operation is called `percolateDown`

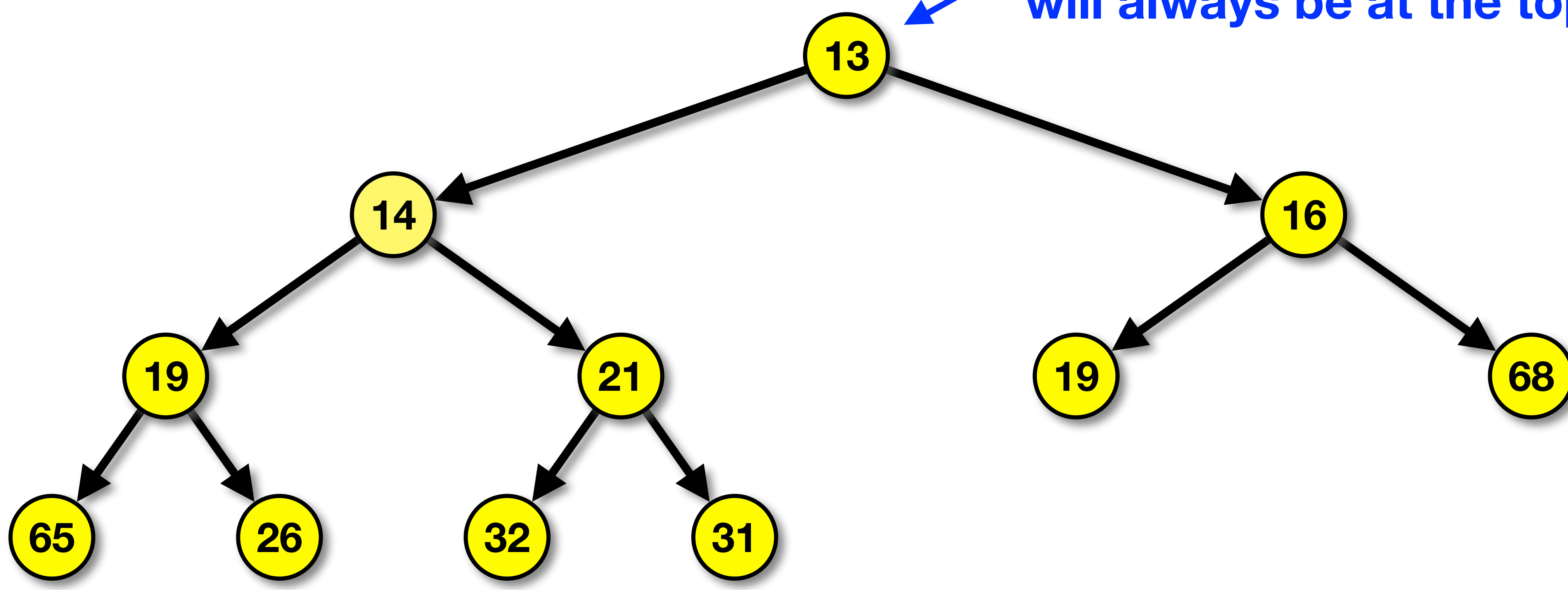
Binary Heap Operations: percolateDown

- **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

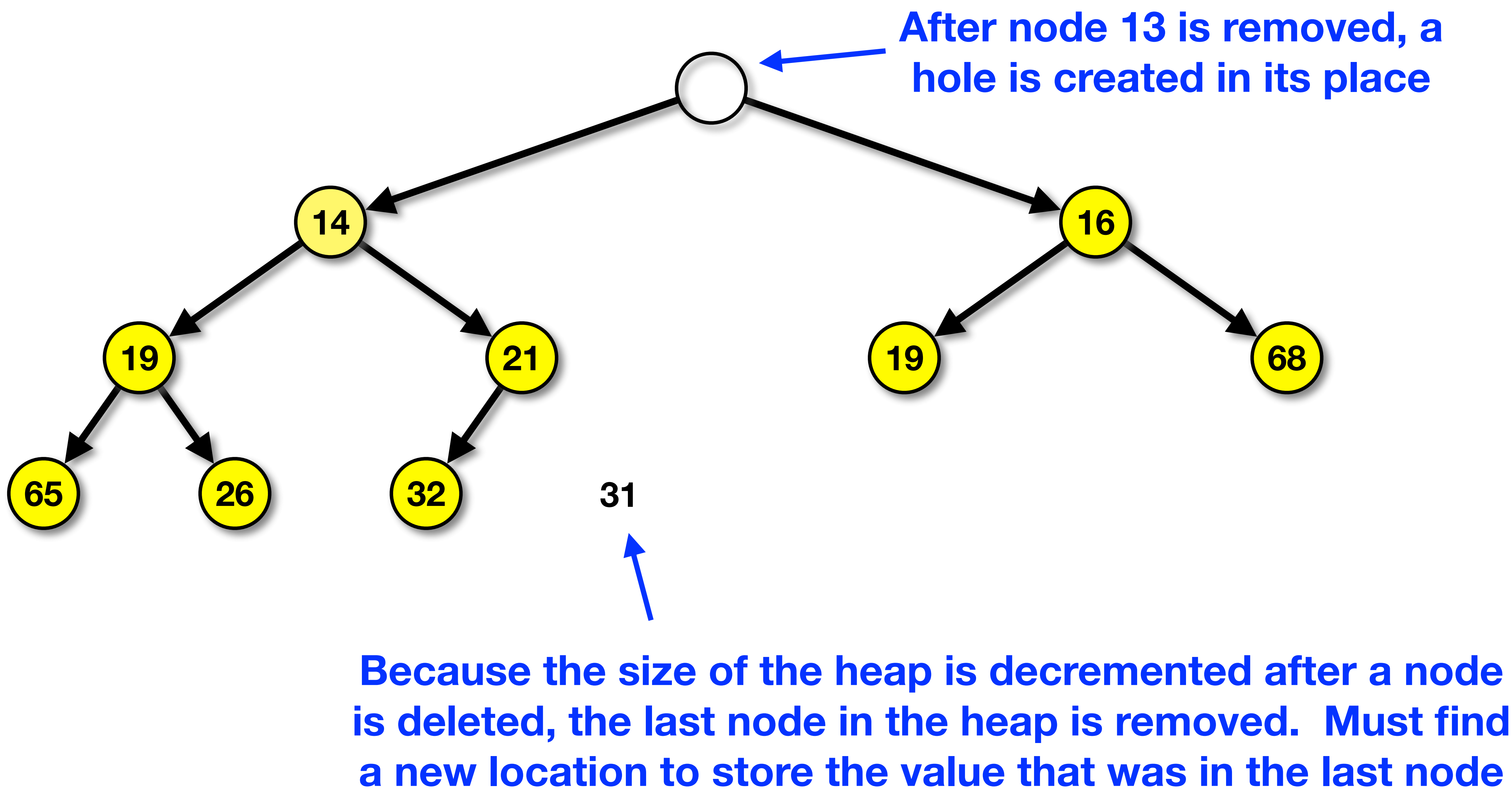
Delete the minimum node
The minimum node is node 13

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



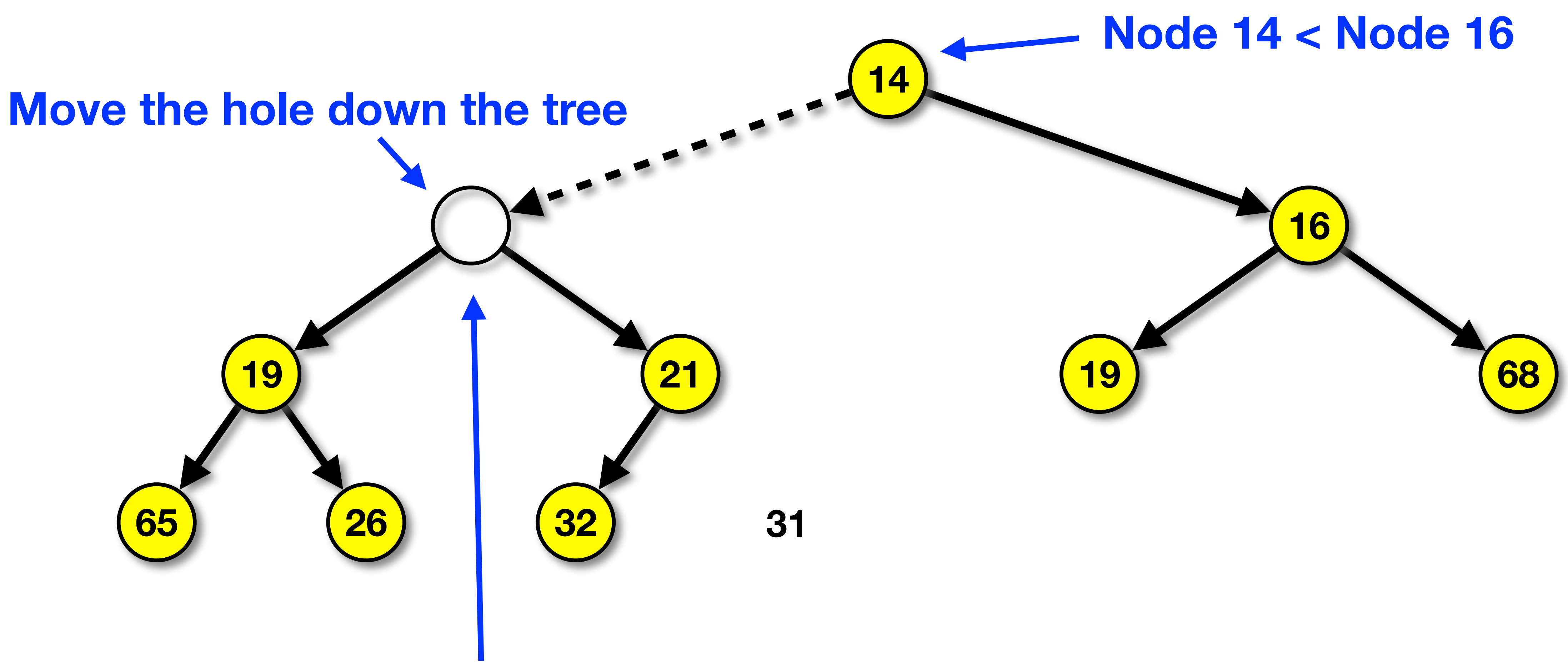
Binary Heap Operations: deleteMin Example

Delete the minimum node
A hole is created where the minimum value was removed
The size of the heap decreases



Binary Heap Operations: deleteMin Example

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

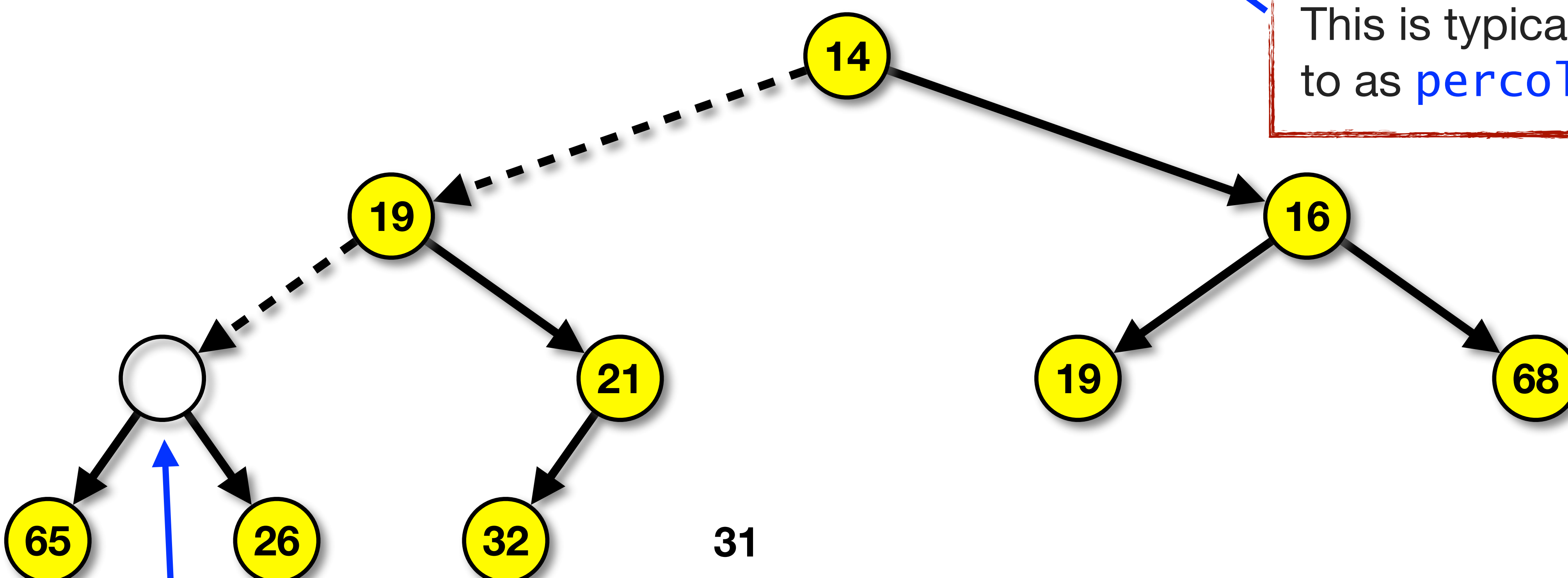


Can the 31 we're trying to place be moved to this location without violating the heap property?

Binary Heap Operations: deleteMin Example

Delete the minimum node
Continually swap the hole with its smaller child to move the hole down the heap until a valid location is found to place the last value

This is typically referred to as `percolateDown`

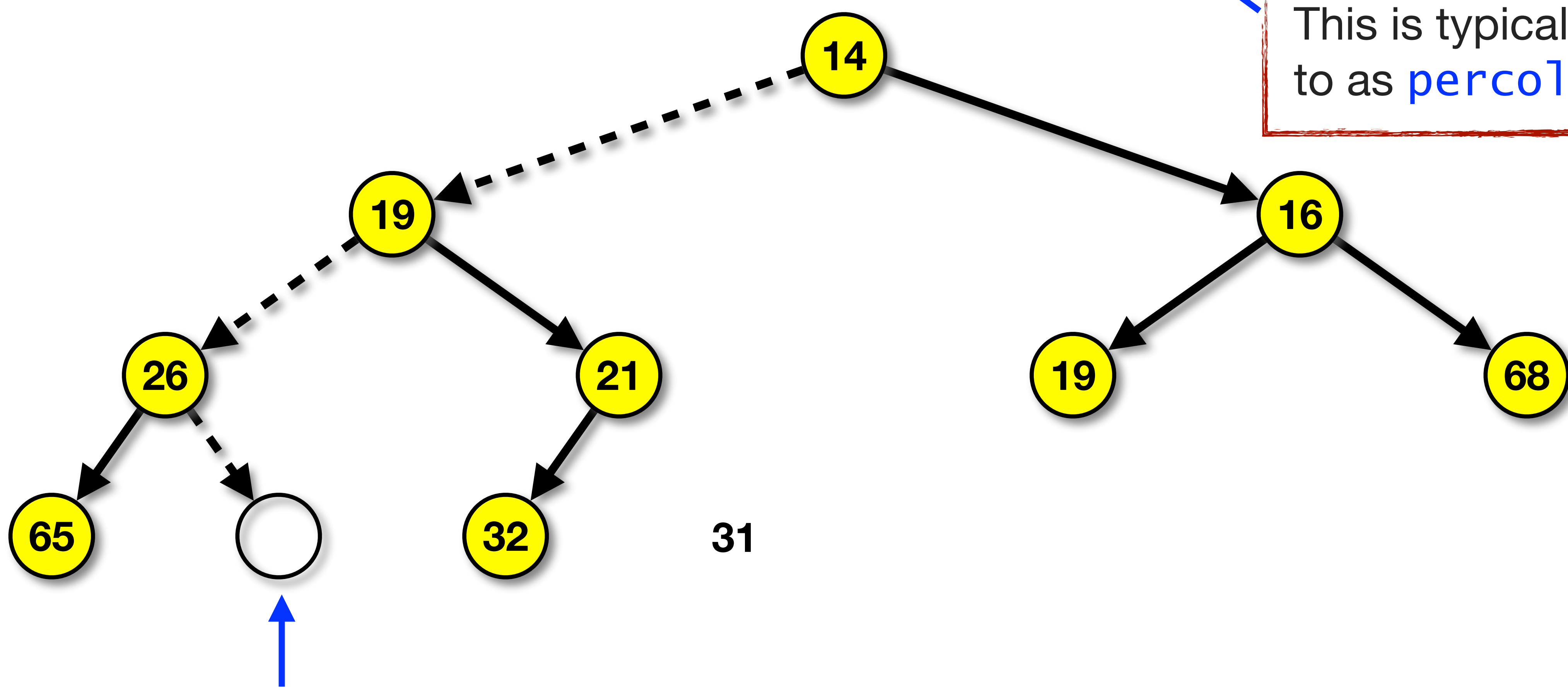


Can the 31 we're trying to place be moved to this location without violating the heap property?

Binary Heap Operations: deleteMin Example

Delete the minimum node
Continually swap the hole with its smaller child to move the hole down the heap until a valid location is found to place the last value

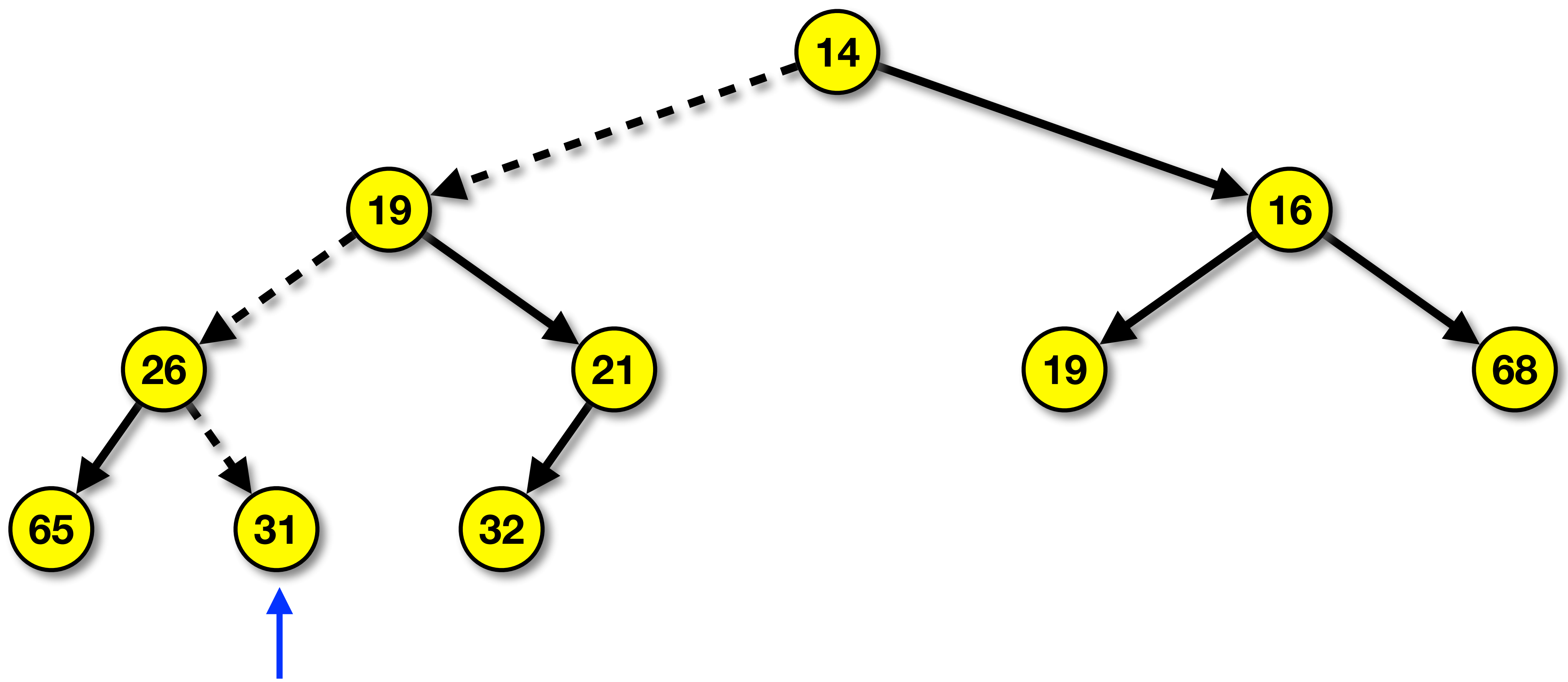
This is typically referred to as `percolateDown`



Can the 31 we're trying to place be moved to this location without violating the heap property?

Binary Heap Operations: deleteMin Example

Delete the minimum node
Once a location is found for the 31, fill the hole with that value



Fill the hole with the value that was last in the heap prior to the deleteMin operation

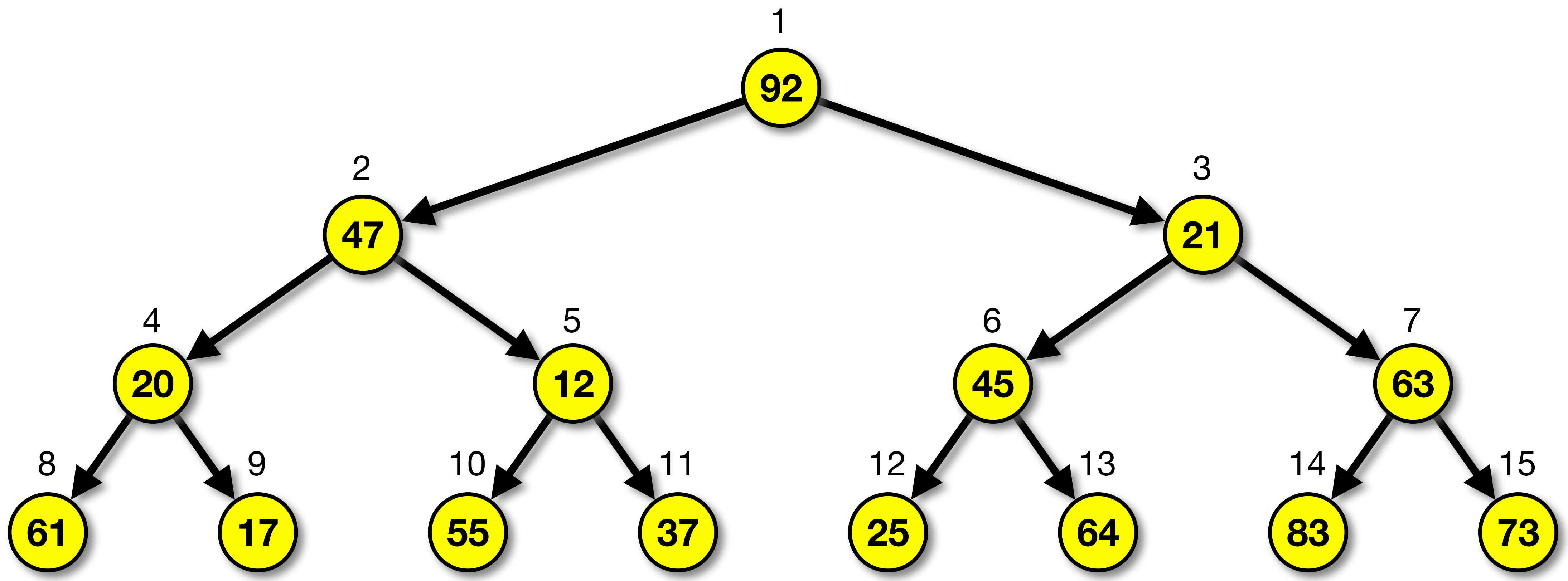
Binary Heap Operations: buildHeap

- **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 ⌊`currentSize/2`⌋
    - 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

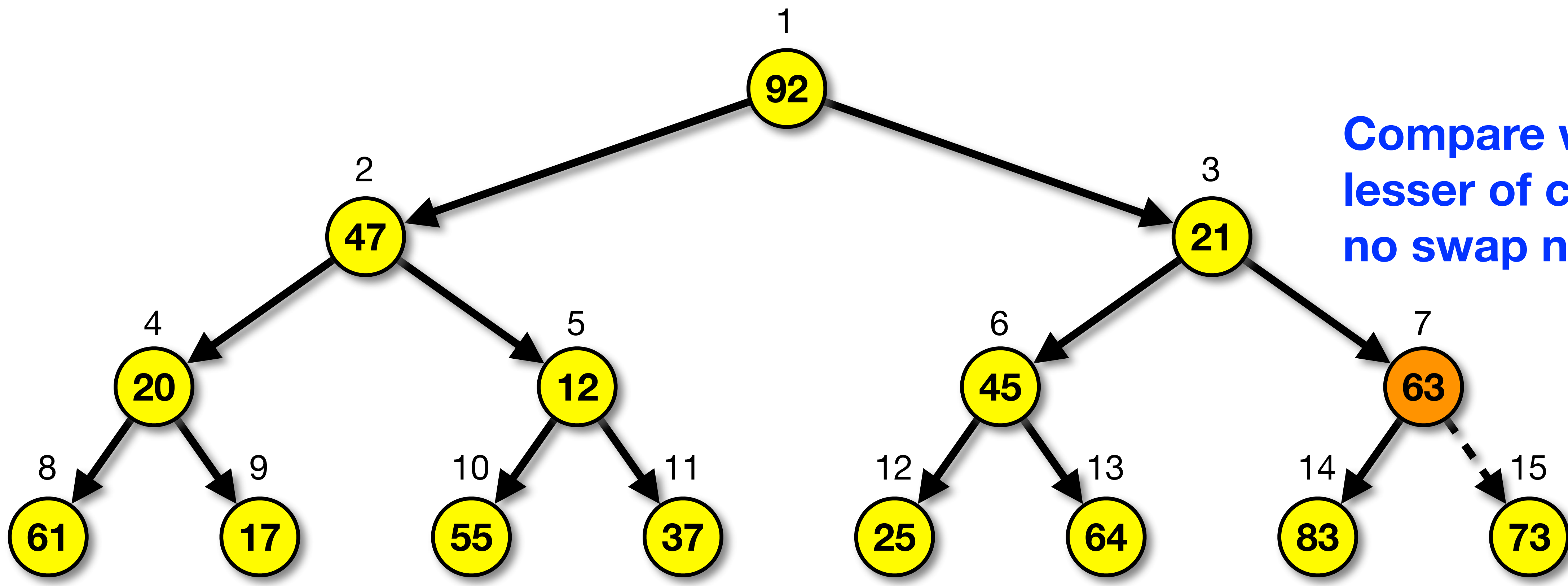
Fixing the heap order
No need to call percolateDown on the leaf nodes



Binary Heap Operations: buildHeap Example

Heap size is 15, so first non-leaf node is $\lfloor 15/2 \rfloor = 7$

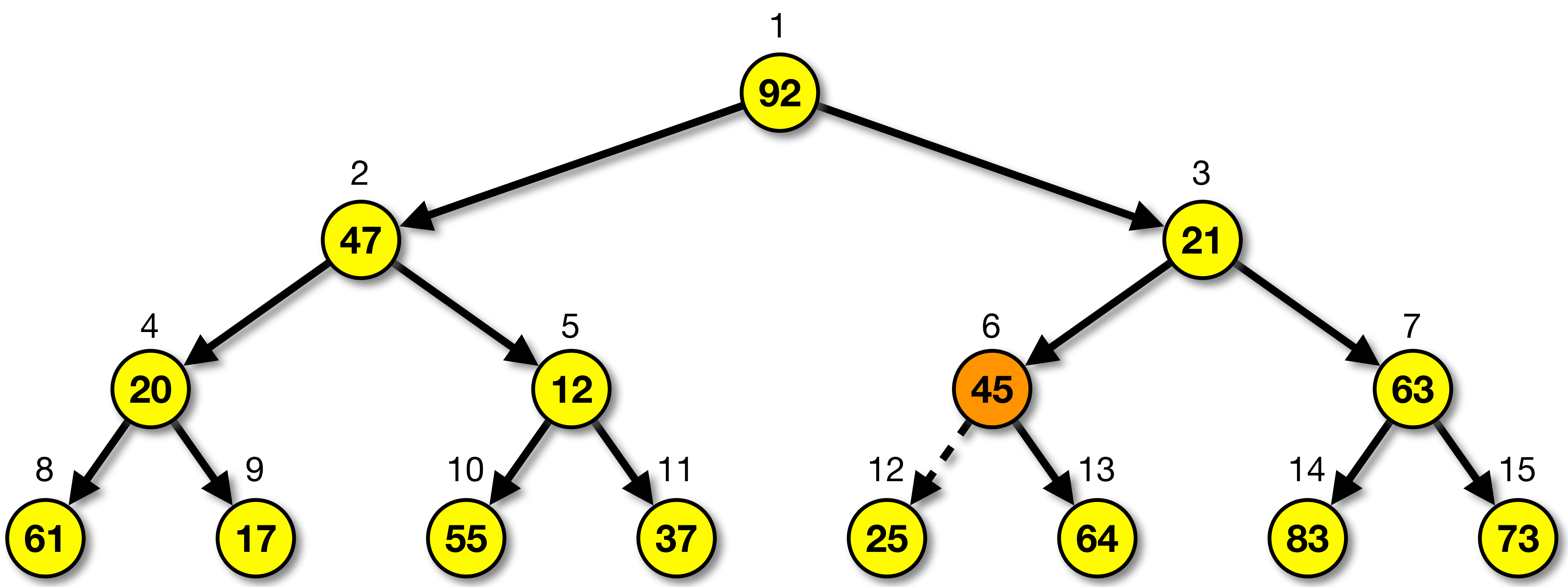
Processing node at array index 7



Compare with lesser of children, no swap needed

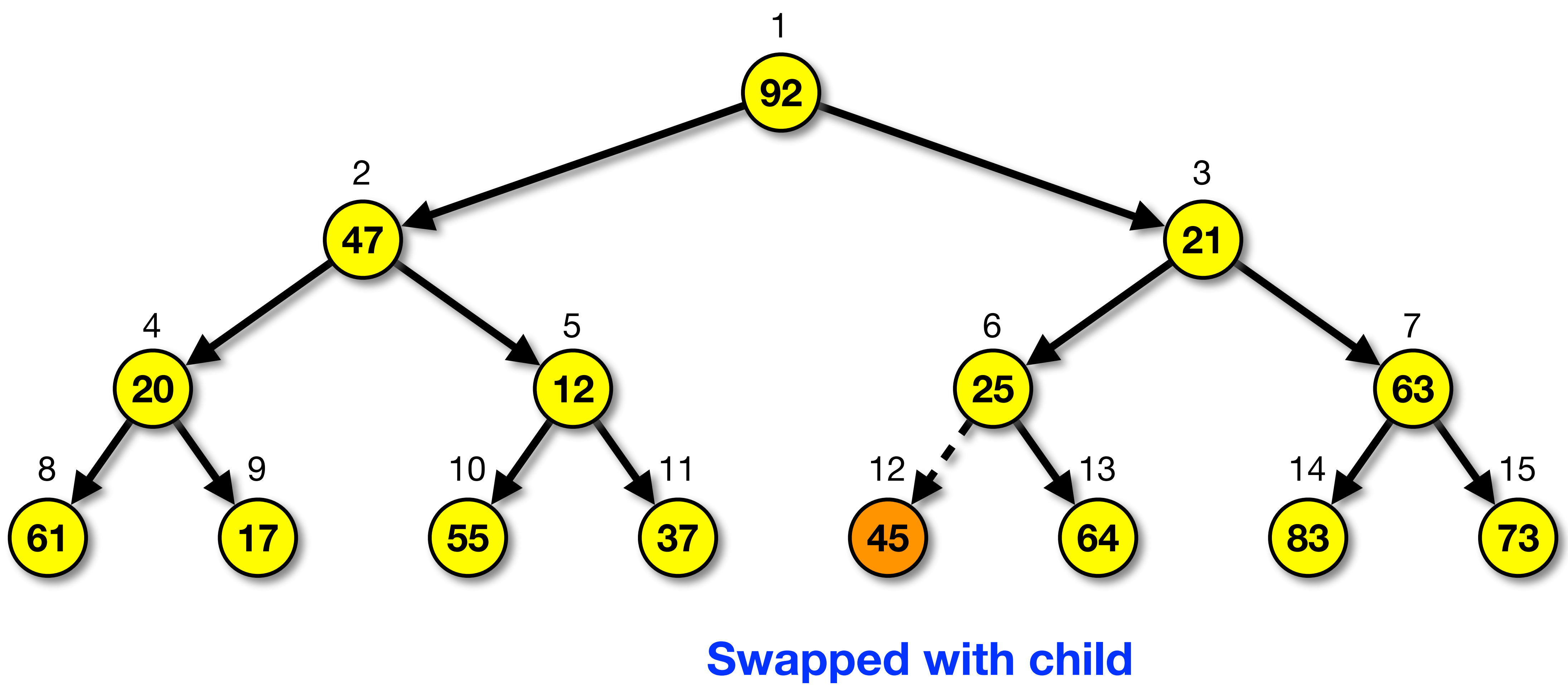
Binary Heap Operations: buildHeap Example

Processing node at array index 6



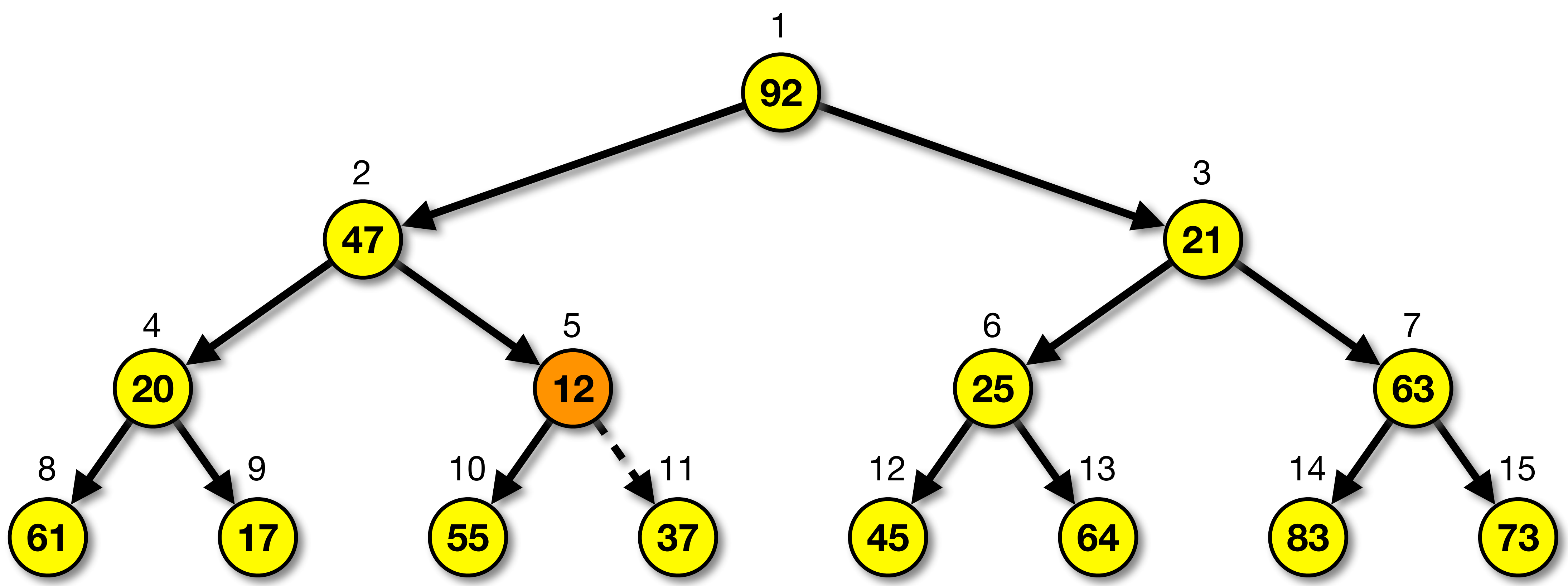
Compare with lesser of children, need to swap

Binary Heap Operations: buildHeap Example



Binary Heap Operations: buildHeap Example

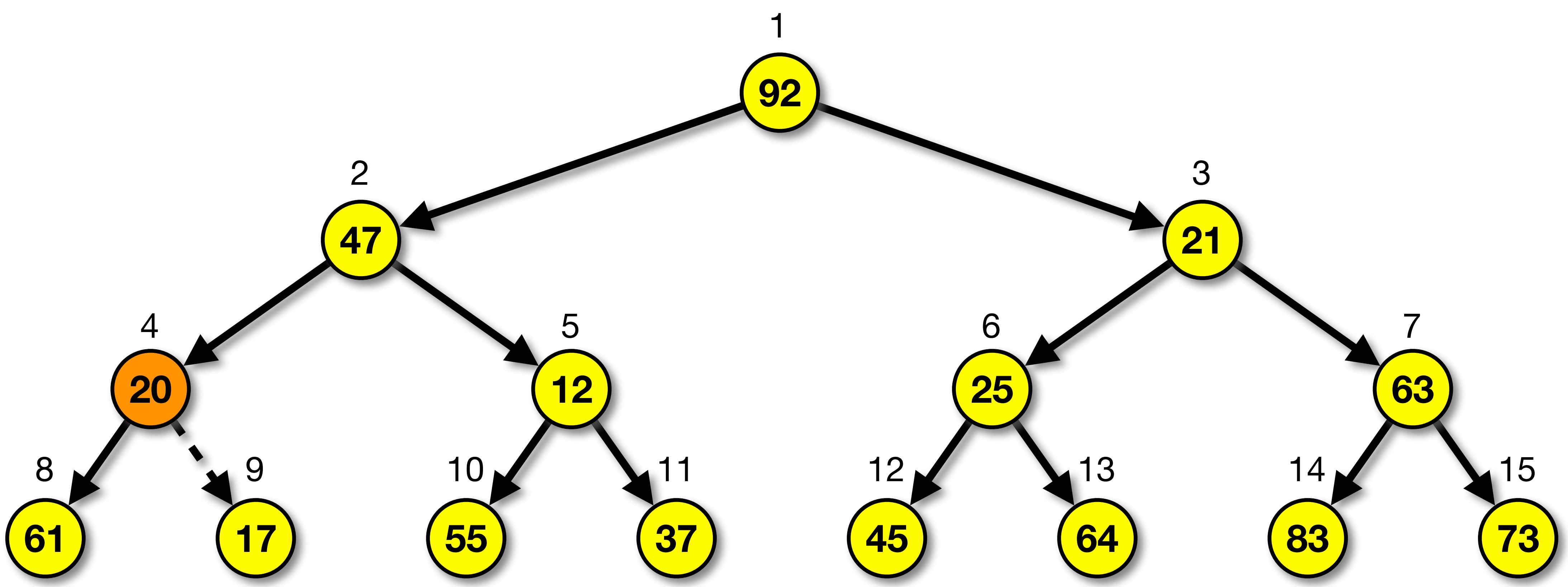
Processing node at array index 5



Compare with lesser of children, no swap needed

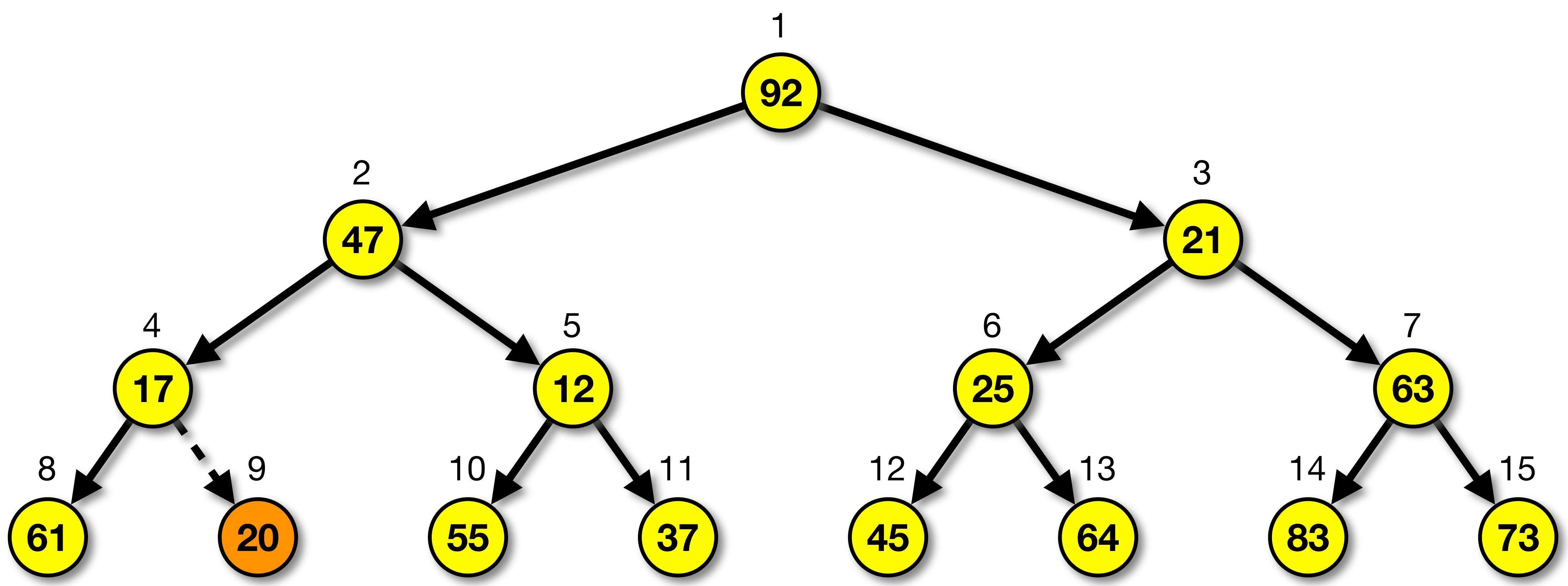
Binary Heap Operations: buildHeap Example

Processing node at array index 4



Compare with lesser of children, need to swap

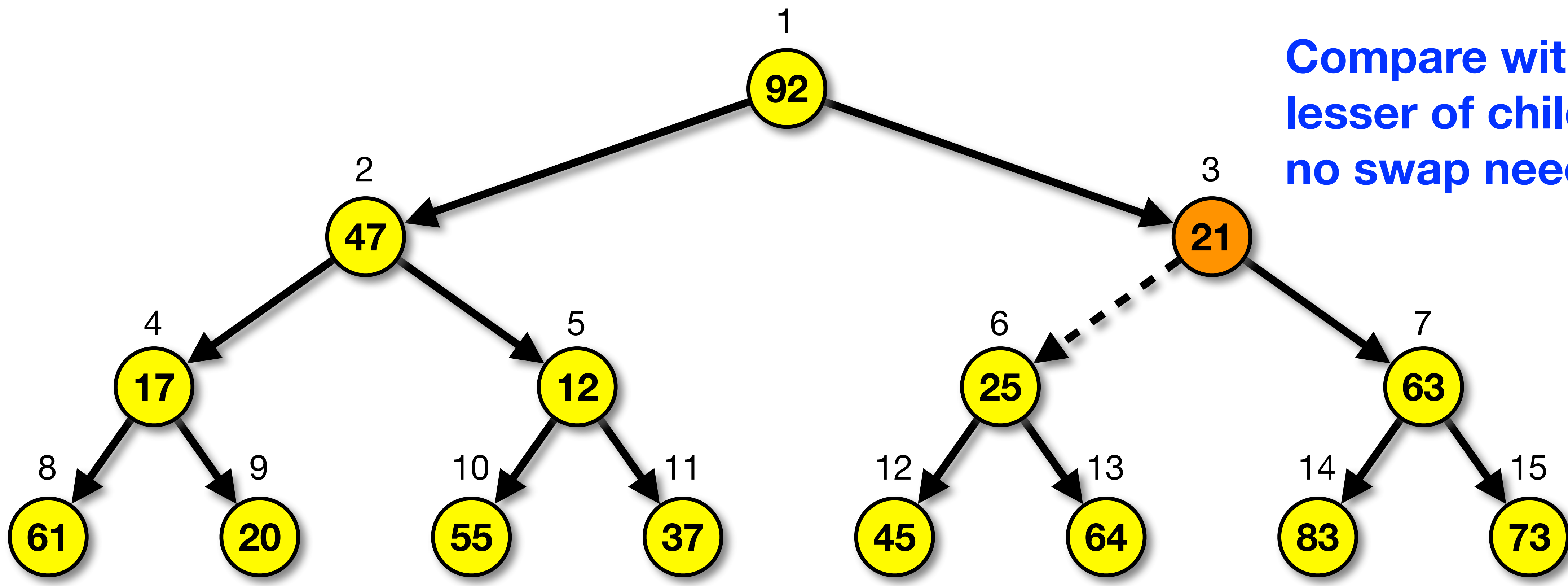
Binary Heap Operations: buildHeap Example



Swapped with child

Binary Heap Operations: buildHeap Example

Processing node at array index 3

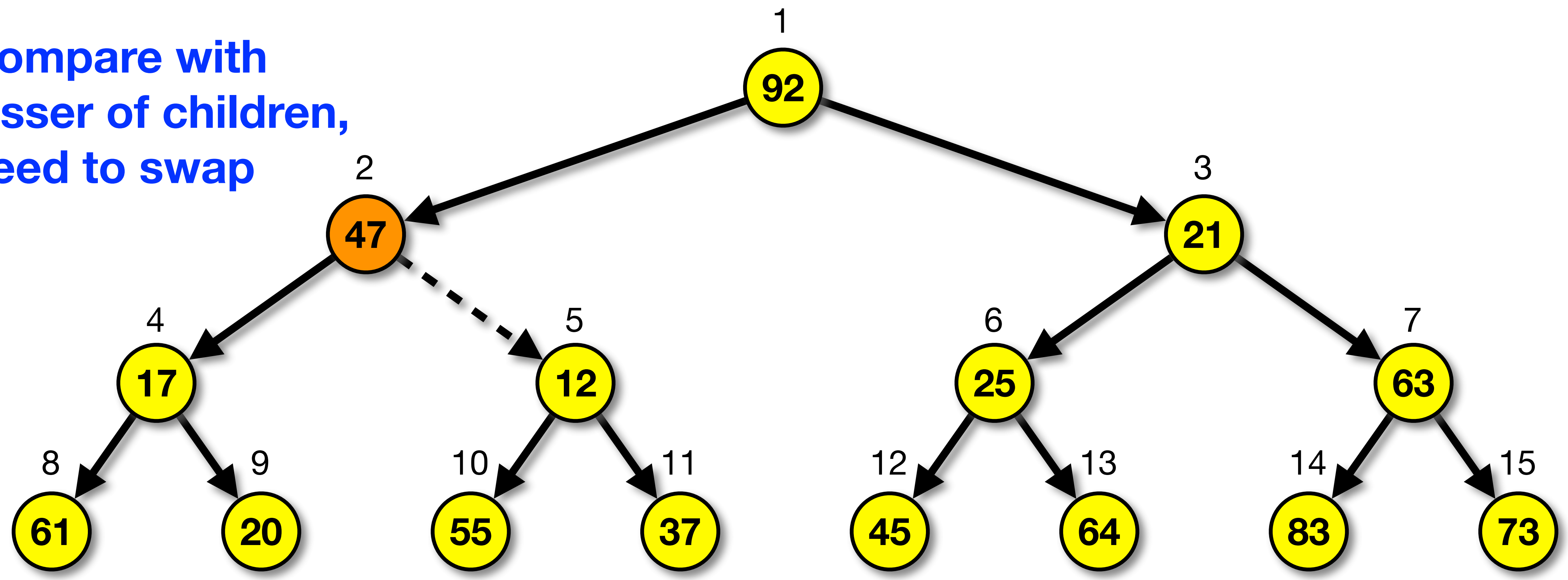


Compare with lesser of children, no swap needed

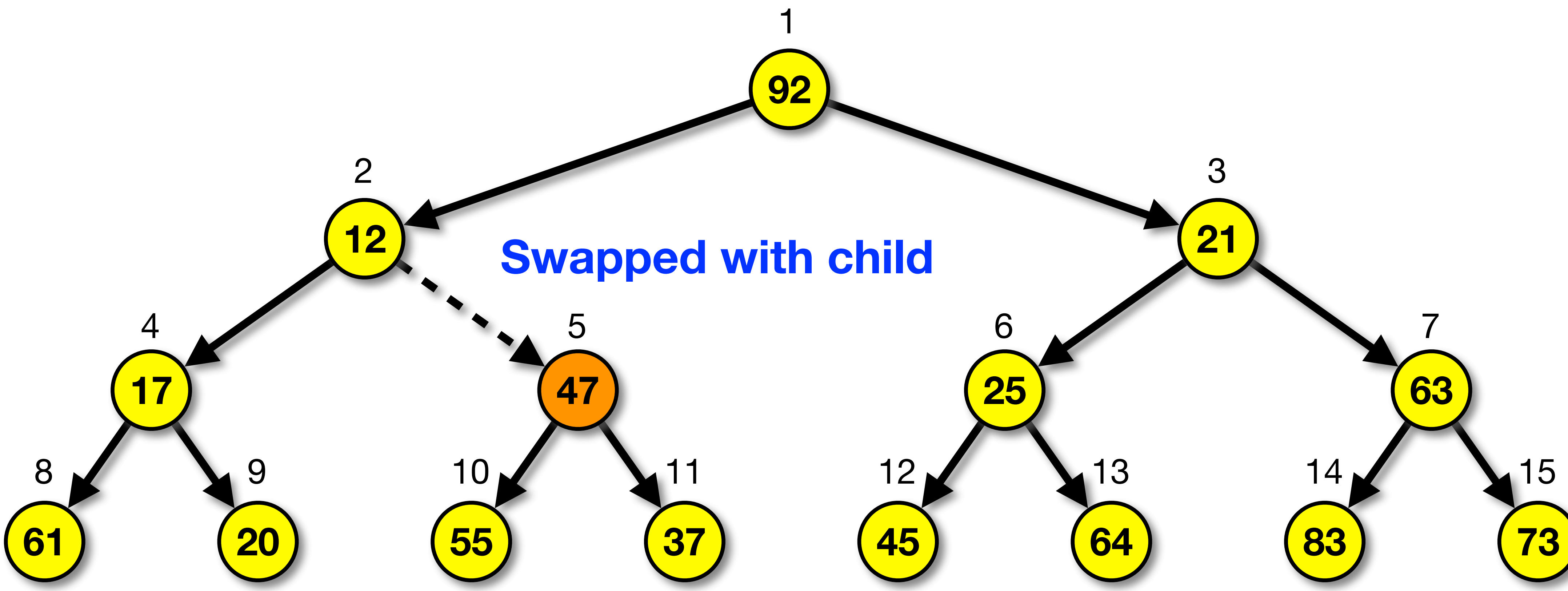
Binary Heap Operations: buildHeap Example

Processing node at array index 2

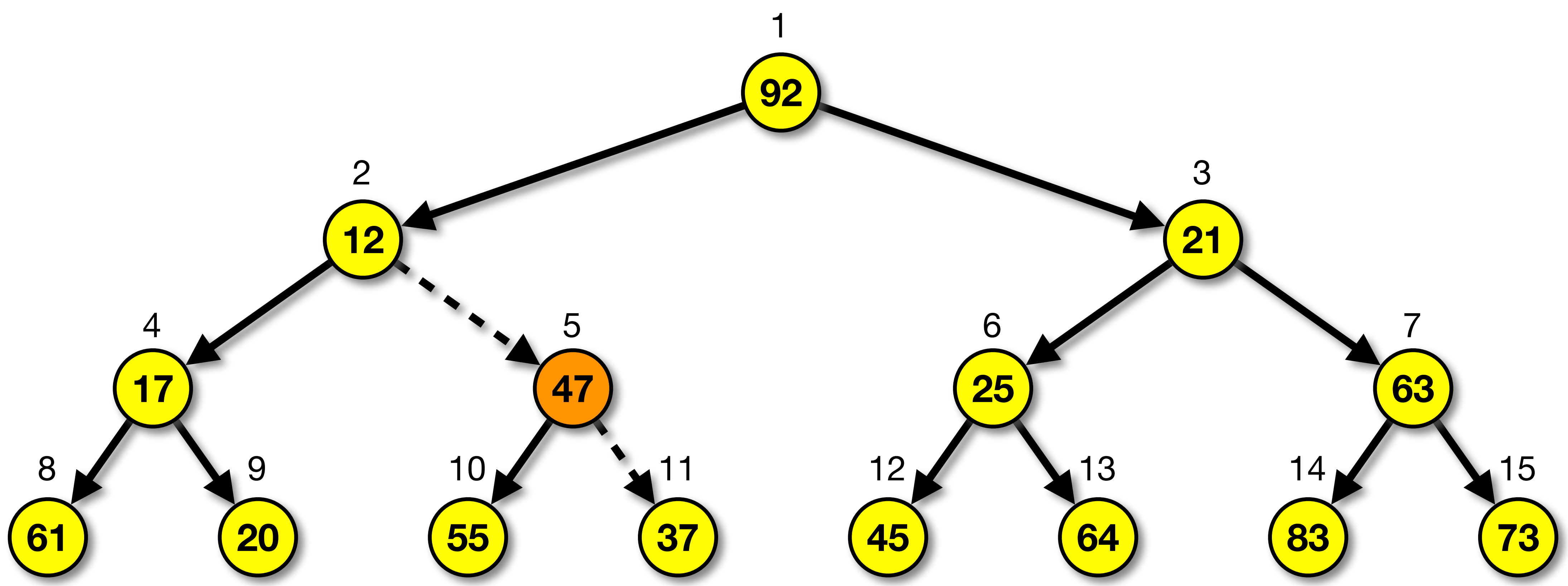
Compare with lesser of children, need to swap



Binary Heap Operations: buildHeap Example

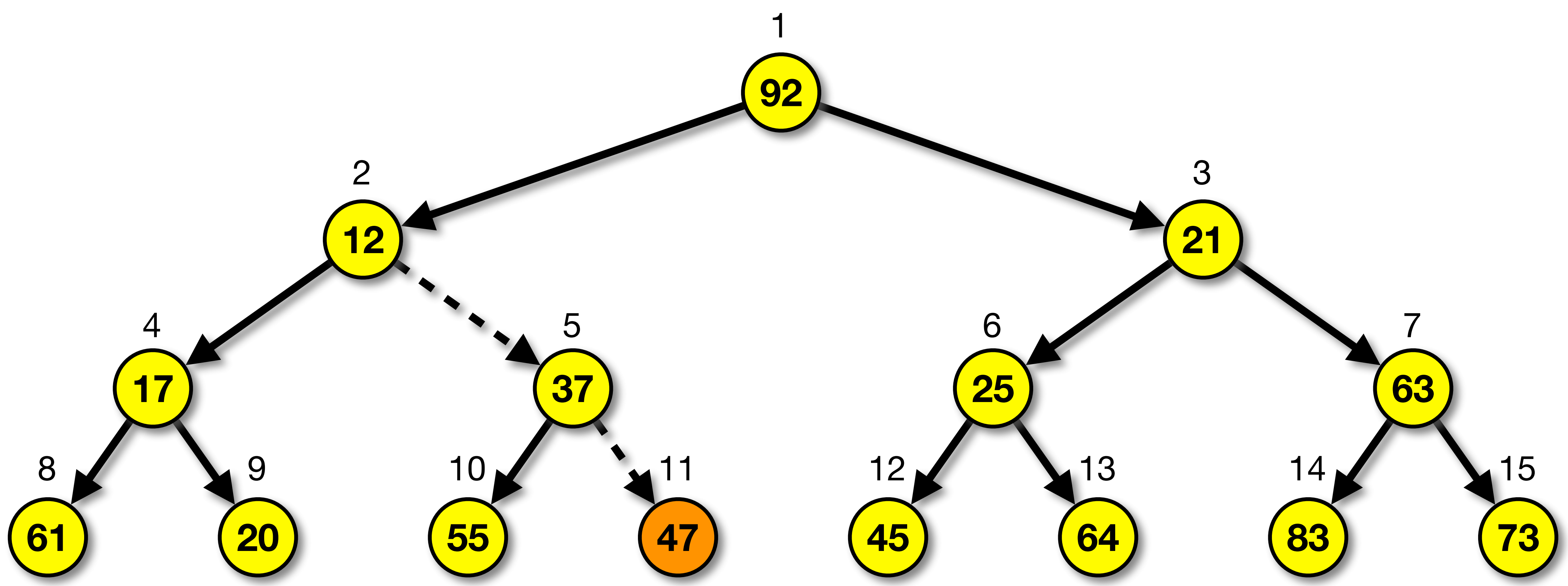


Binary Heap Operations: buildHeap Example



Continue to compare with lesser of children, Need another swap

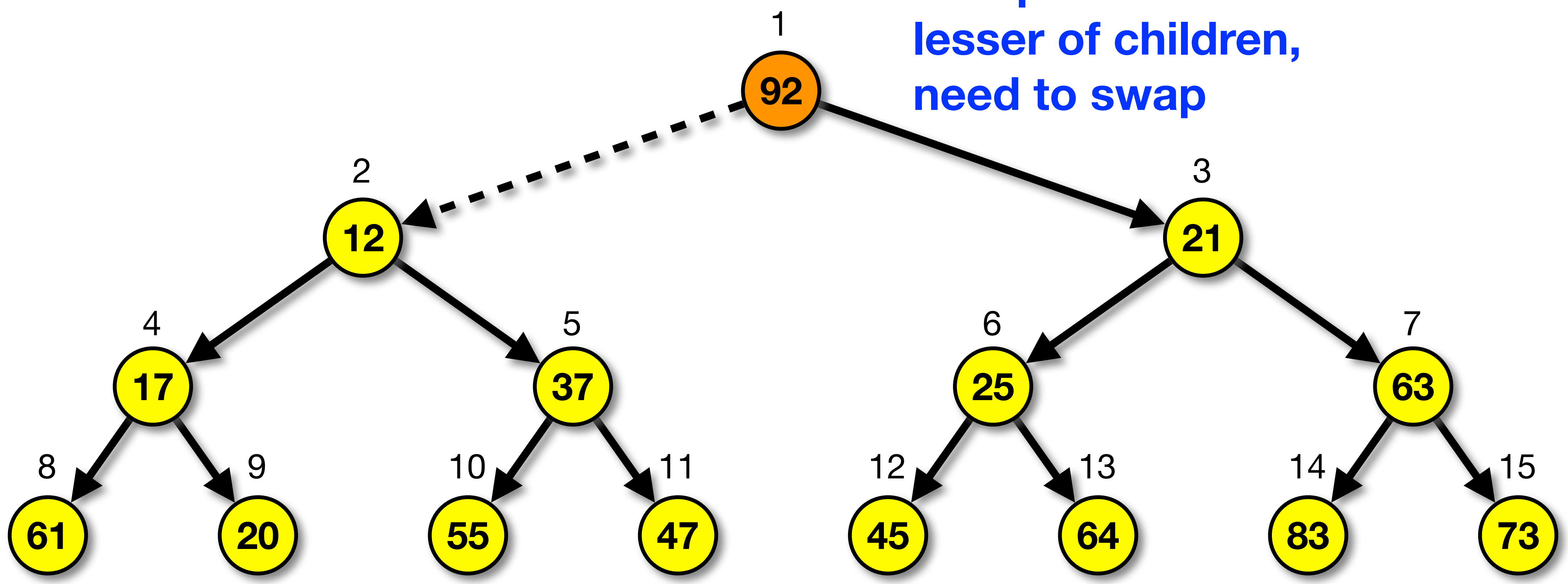
Binary Heap Operations: buildHeap Example



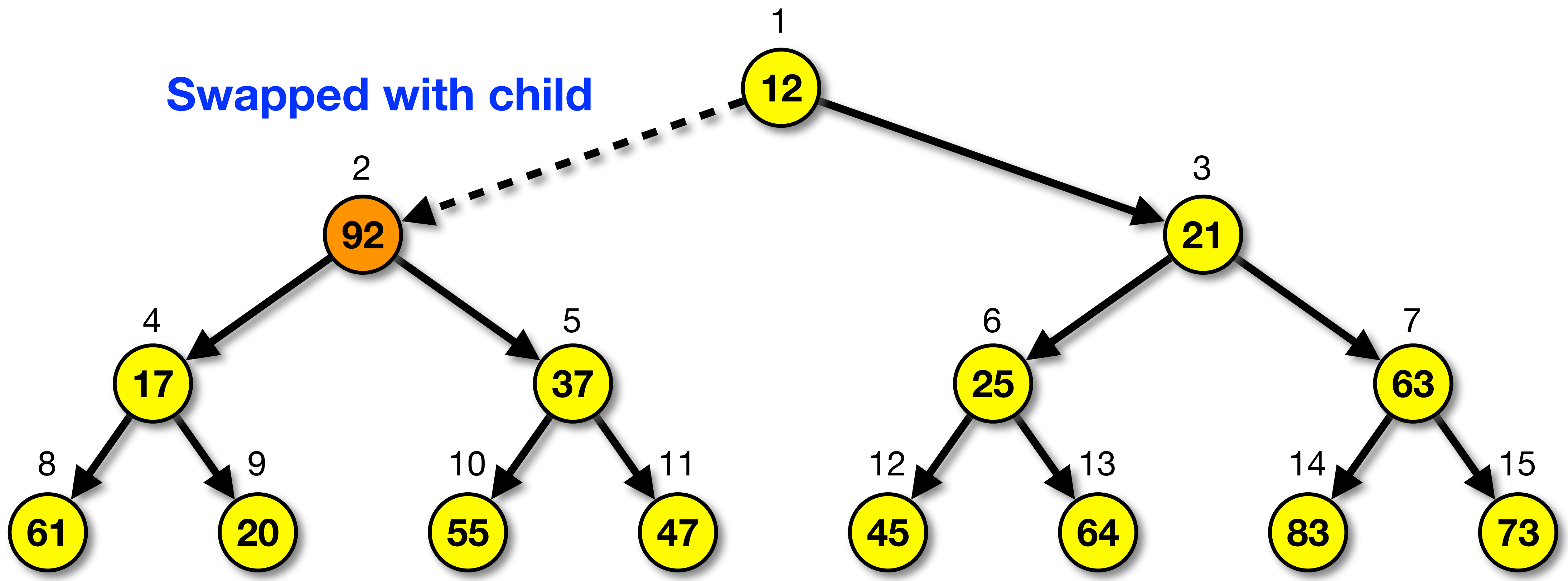
Swapped with child

Binary Heap Operations: buildHeap Example

Processing node at array index 1

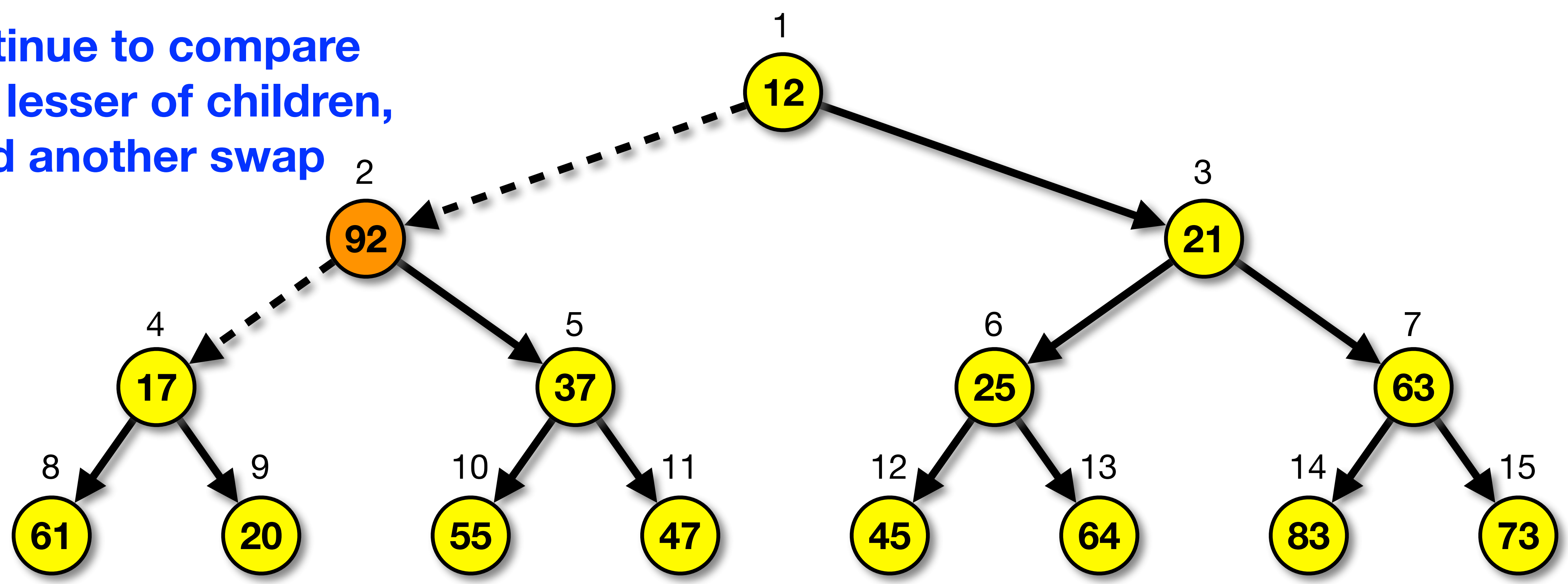


Binary Heap Operations: buildHeap Example

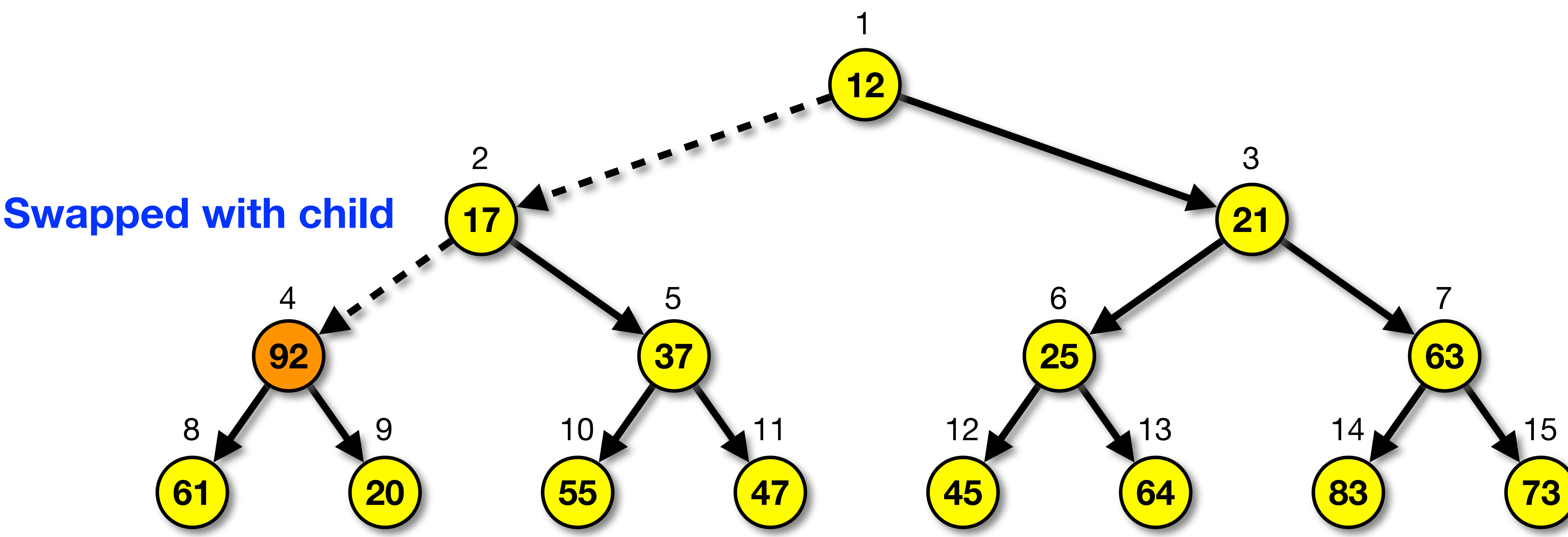


Binary Heap Operations: buildHeap Example

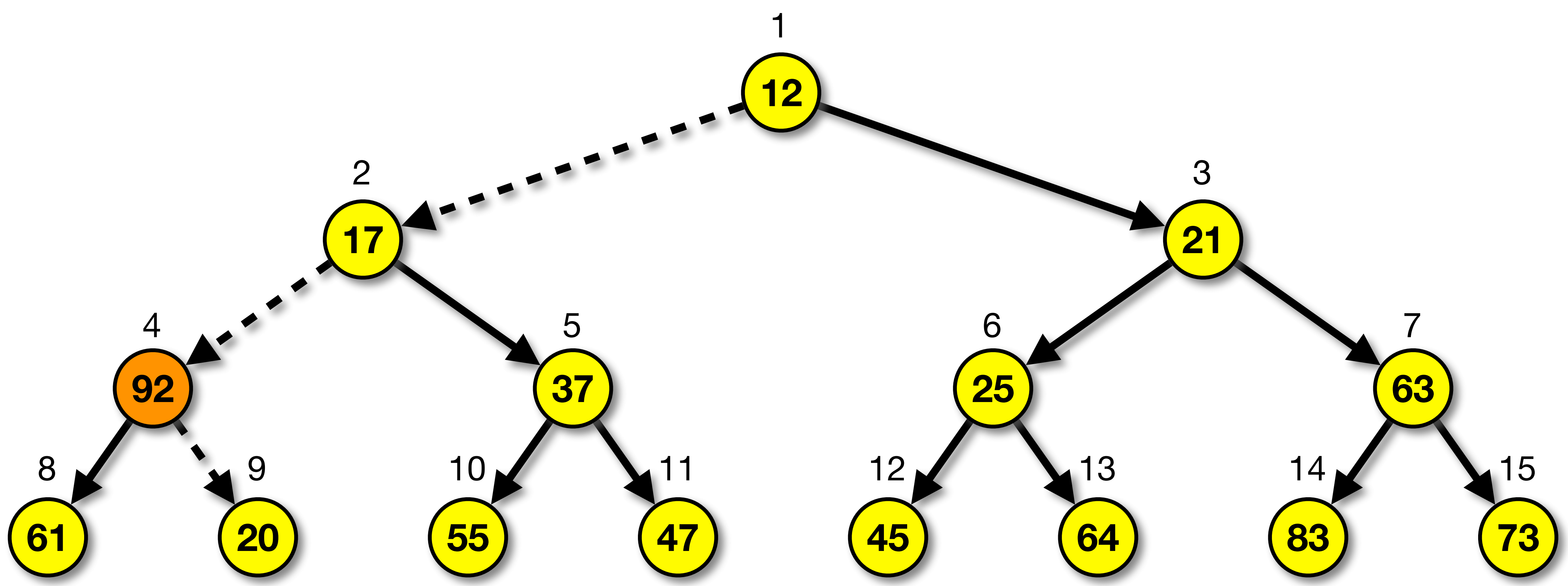
Continue to compare with lesser of children, Need another swap



Binary Heap Operations: buildHeap Example

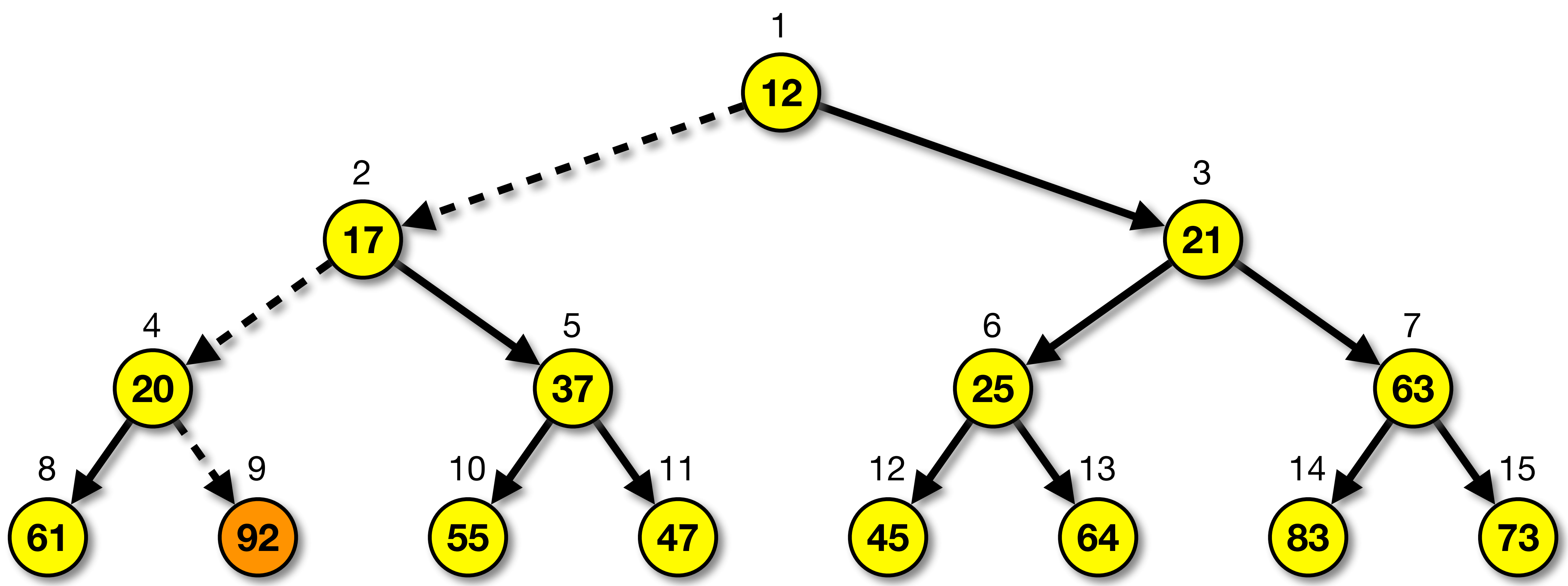


Binary Heap Operations: buildHeap Example



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