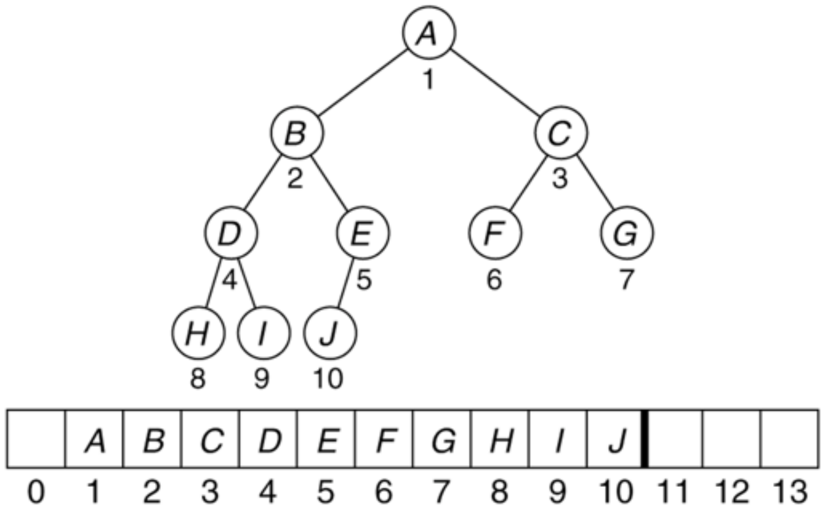



# CSSE 230 Day 20

## Priority Queues
## Heaps

After this lesson, you should be able to …
… apply the binary heap insertion and deletion algorithms by hand
… implement the binary heap insertion and deletion algorithms

# Exam 3

- Format same as Exam 1 (written and programming)
  - One 8.5x11 sheet of paper (one side) for written part
  - Same resources as before for programming part
- Topics: weeks 1-7
  - Through day 21, HW7, and EditorTrees milestone 3
  - Especially Binary trees, including BST, AVL, indexed (EditorTrees), Red-black
  - Traversals and iterators, size vs. height, rank
  - Recursive methods, including ones that should only touch each node once for efficiency (like sum of heights from HW5 and isHeightBalanced)
  - Hash tables
  - Heaps – basic concepts (we won't ask you to write code yet)
- Practice exam posted in Moodle and code in repos

# Priority Queue ADT

Basic operations
Implementation options

# Priority Queue operations

- Each element in the PQ has an associated **priority**
  - Could be specified (extra parameter) when an item is inserted into the PQ: insert(item, priority)
  - More commonly, priority is inferred from the comparable type (in our examples, an integer).
- Operations:
  - insert(item) (also called add/offer)
  - findMin()
  - deleteMin( ) (also called remove/poll)
  - isEmpty() …

# Priority queue implementation

- Can we reasonably implement PQ using data structures that we already know about?
  - Array?
  - Sorted array?
  - AVL?
- One efficient approach uses a binary heap
  - A complete binary tree that is ordered in a special way
- Questions we'll ask:
  - How can we efficiently represent a complete binary tree?
  - Can we add and remove items efficiently without destroying the "heapness" of the structure?

# Binary Heap

An efficient implementation of the PriorityQueue ADT

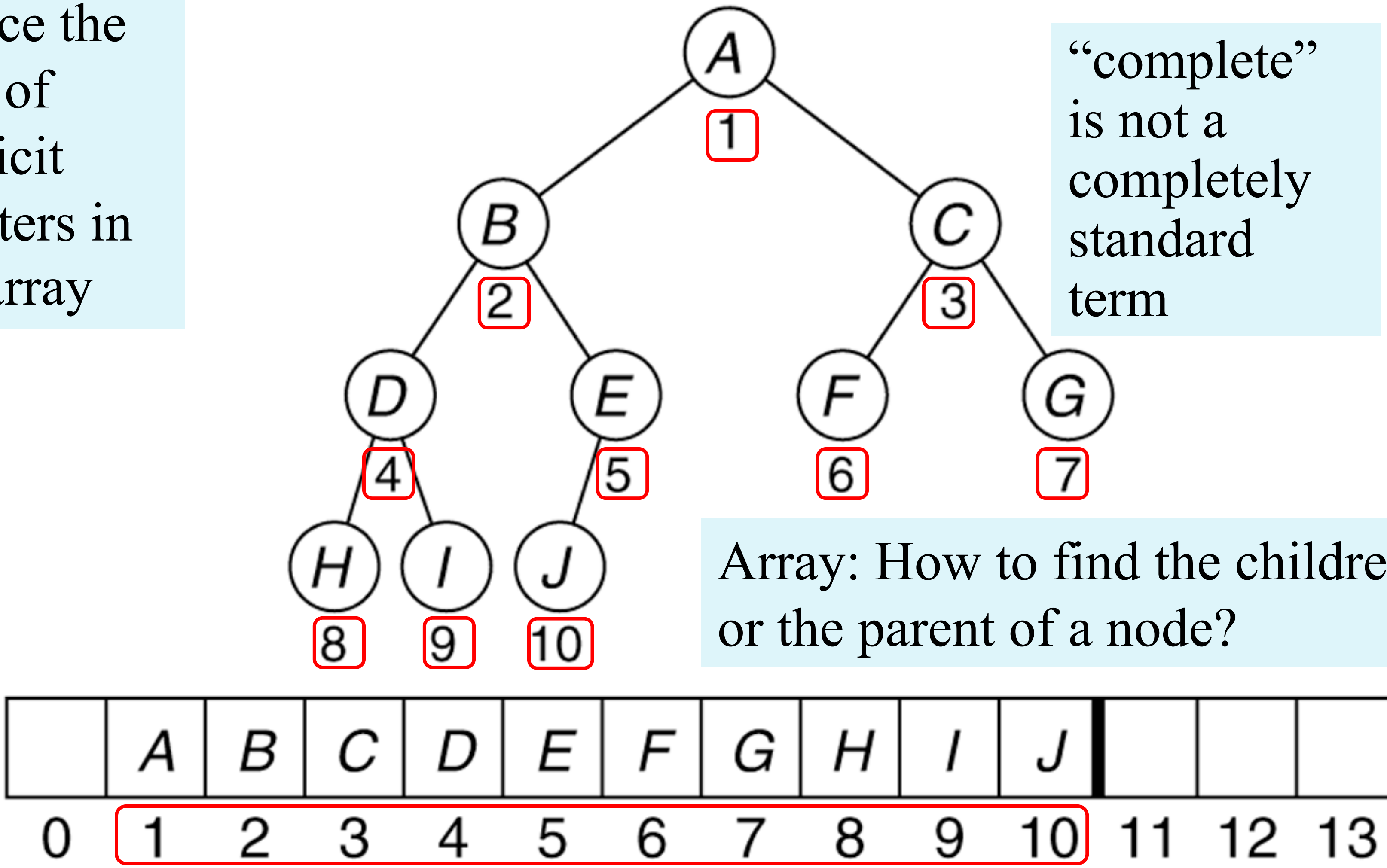
Storage (an array)

Algorithms for insertion and deleteMin

# Figure 21.1

A *complete binary tree* and its array representation

Notice the lack of explicit pointers in the array




“complete” is not a completely standard term

Array: How to find the children or the parent of a node?

One "wasted" array position (0)

# The (min) heap-order property: every node's value is ≤ its childrens' values


$P \leq X$

A **Binary** (min) **Heap** is a complete Binary Tree (using the array implementation, as on the previous slide) that has the heap-order property everywhere.

Q: In a binary heap, where do we find

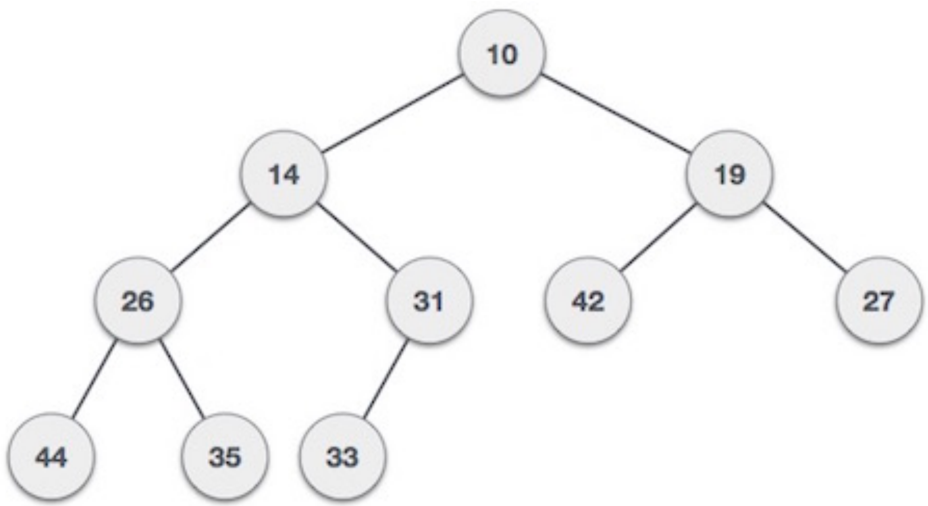
- The smallest element?
- 2nd smallest?
- 3rd smallest?

# Correspondence – Abstract Heap to Array Rep

## Fill in the array with values from the min-heap

- Heap size = # items in the heap
- Array capacity = size of the array




# Insert and DeleteMin

Idea of each:

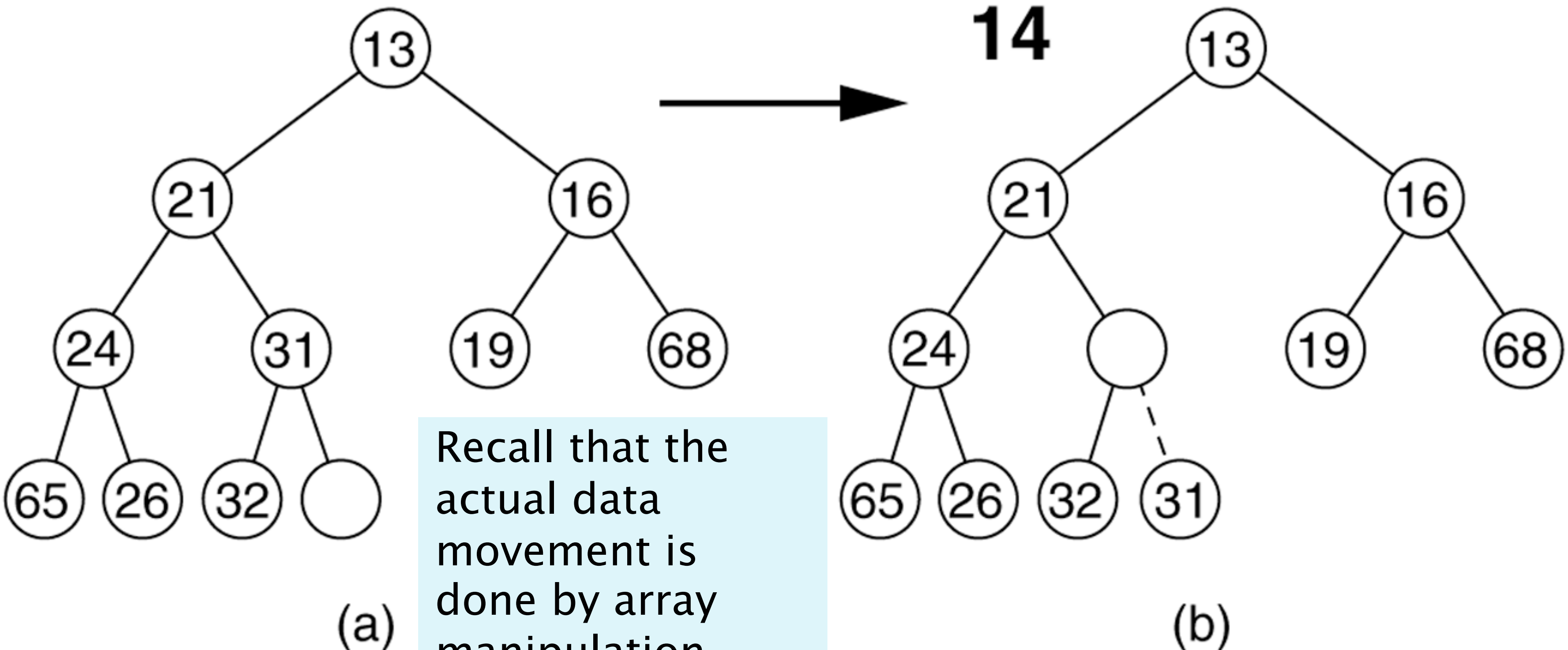
1. Get the **structure** right first
   - Insert at end (bottom of tree)
   - Move the last element to the root after deleting the root
2. Restore the heap-order property by percolating (swapping an element/child pair)
   - Insert by percolating *up*: swap with parent
   - DeleteMin by percolating *down*: swap with child with min value

Nice demo:
http://www.cs.usfca.edu/~galles/visualization/Heap.html

Figure 21.7

Attempt to insert 14, creating the hole and bubbling the hole up

# Insertion algorithm



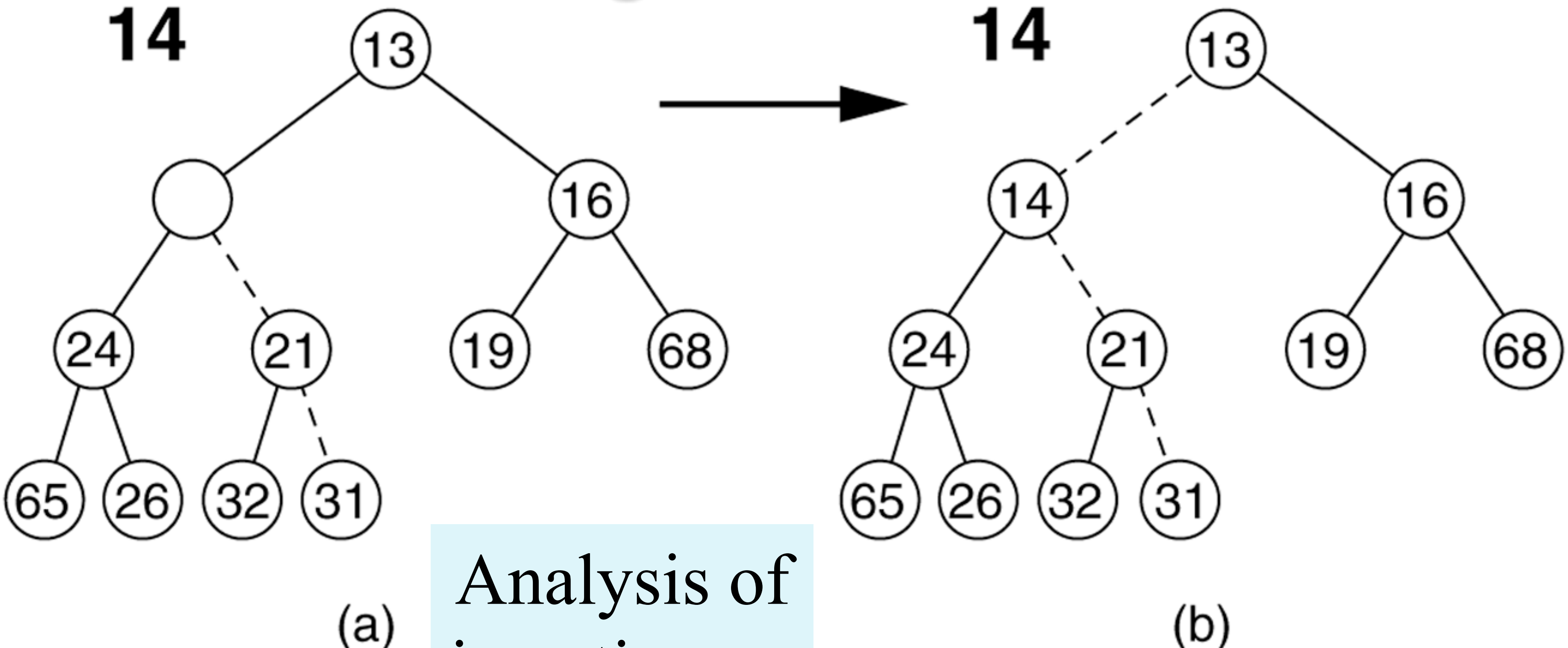

Recall that the actual data movement is done by array manipulation

Create a "hole" where 14 can be inserted. Percolate up!

Figure 21.8
The remaining two steps required to insert 14 in the original heap shown in Figure 21.7

# Insertion Algorithm continued




Analysis of insertion …

# Code for Insertion

```
/**
 * Adds an item to this PriorityQueue.
 * @param x any object.
 * @return true.
 */
public boolean add( AnyType x )
{
    if( currentSize + 1 == array.length )
        doubleArray( );

        // Percolate up
    int hole = ++currentSize;
    array[ 0 ] = x;

    for( ; compare( x, array[ hole / 2 ] ) < 0; hole /= 2 )
        array[ hole ] = array[ hole / 2 ];
    array[ hole ] = x;

    return true;
}
```

**figure 21.9**

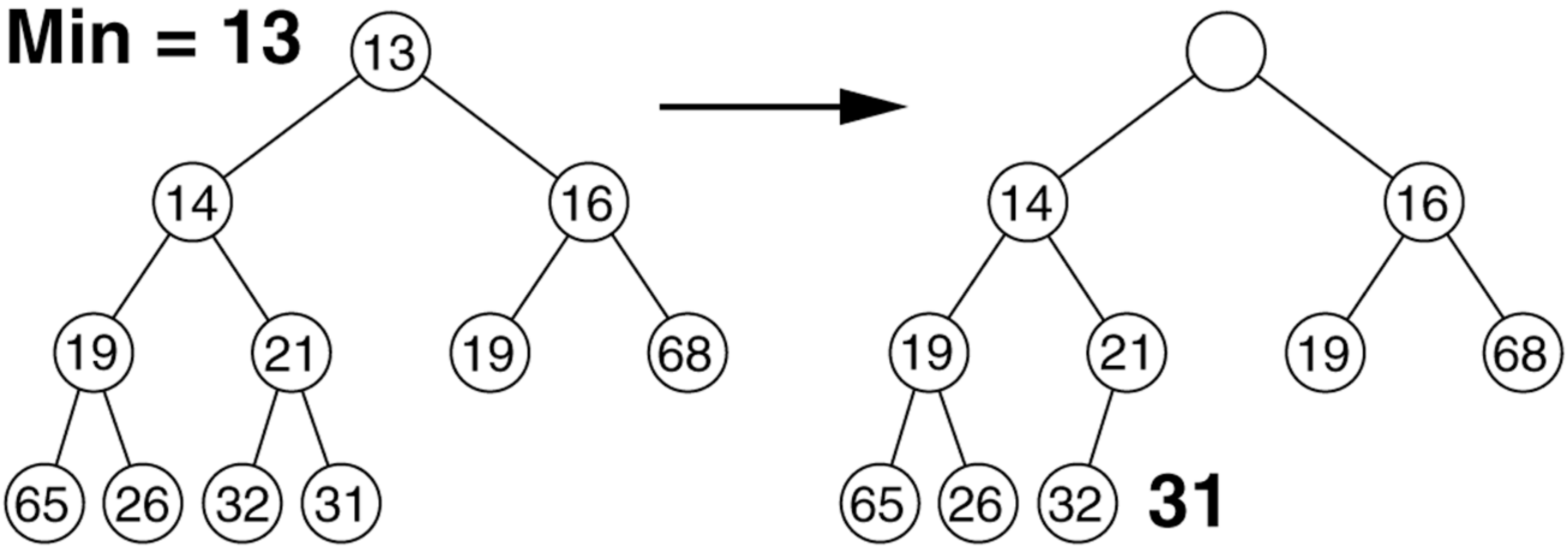
The add method

Your turn:

1. Draw an empty array representation
2. Insert into an initially empty heap: 6 4 8 1 5 3 2 7

# DeleteMin algorithm

The *min* is at the root. Delete it, then use the **percolateDown** algorithm to find the correct place for its replacement.




We must decide which child to promote, to make room for 31.

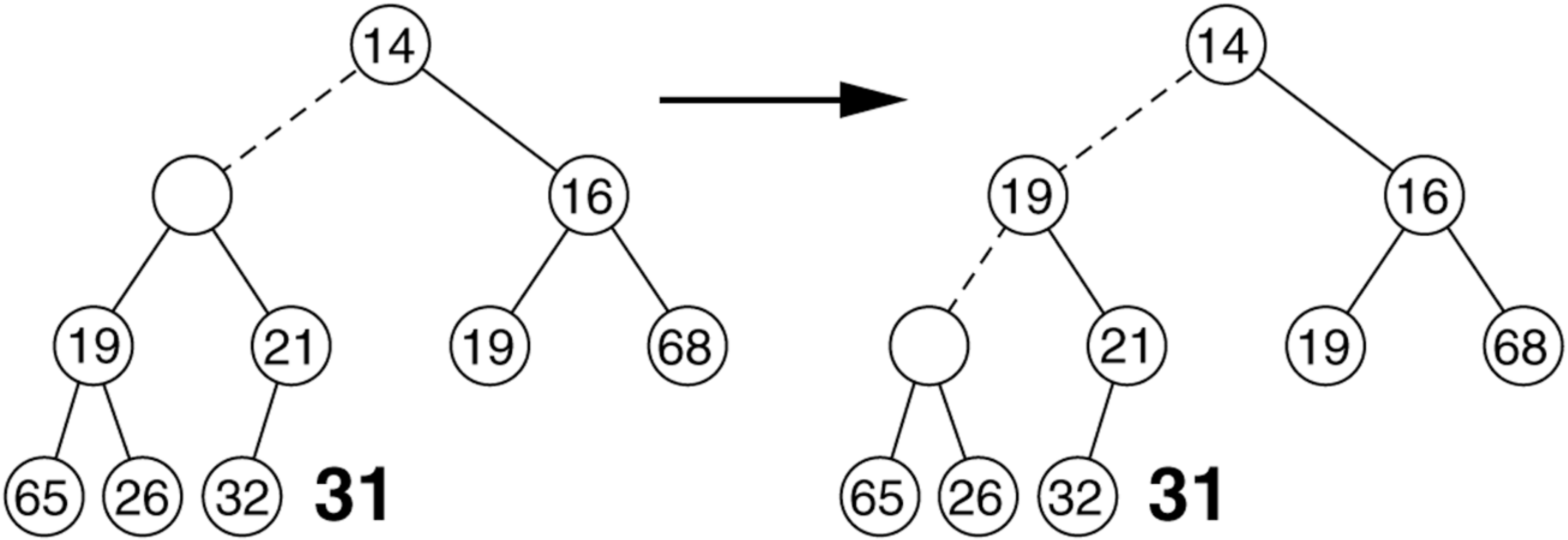
Figure 21.10 Creation of the hole at the root

## Figure 21.11

The next two steps in the deleteMin operation

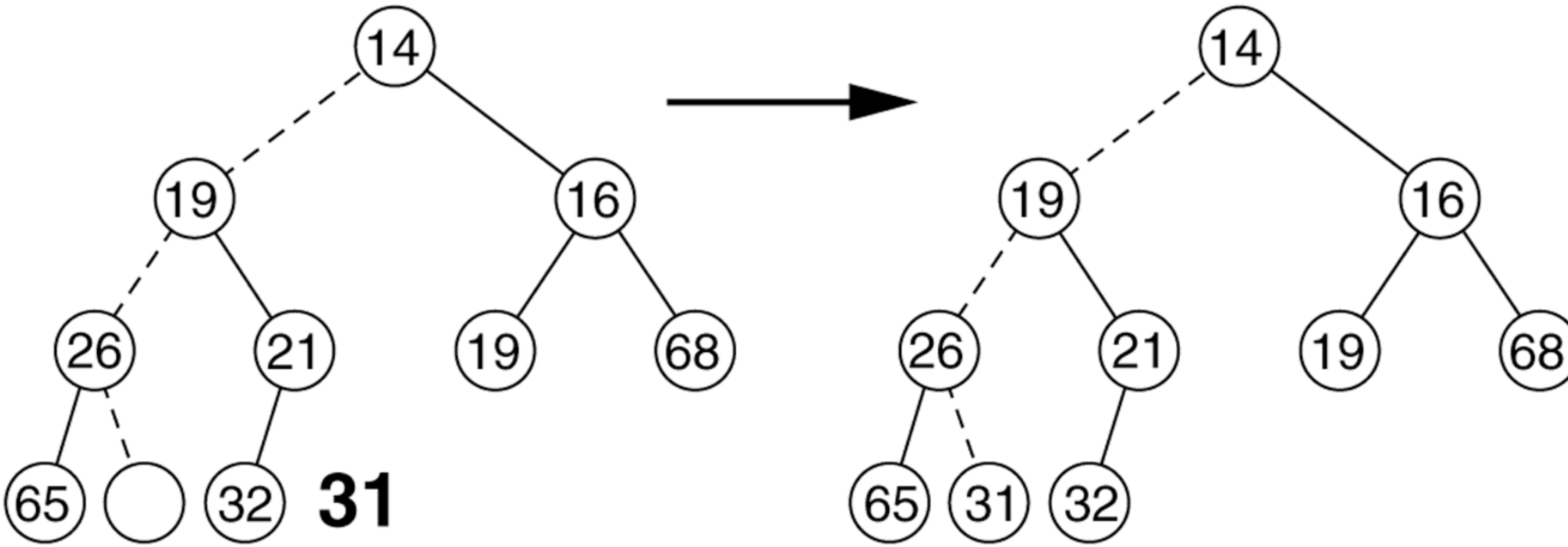
# DeleteMin Slide 2



# Figure 21.12
The last two steps in the deleteMin operation

## DeleteMin Slide 3



```
public Comparable deleteMin( )
{
    Comparable minItem = findMin( );
    array[ 1 ] = array[ currentSize-- ];
    percolateDown( 1 );

    return minItem;
}
private void percolateDown( int hole )
{
    int child;
    Comparable tmp = array[ hole ];

    for( ; hole * 2 <= currentSize; hole = child )
    {
        child = hole * 2;
        if( child != currentSize &&
                array[ child + 1 ].compareTo( array[ child ] ) < 0 )
            child++;
        if( array[ child ].compareTo( tmp ) < 0 )
            array[ hole ] = array[ child ];
        else
            break;
    }
    array[ hole ] = tmp;
}
```

Compare node to its children, moving root down and promoting the smaller child until proper place is found.

We'll re-use percolateDown in HeapSort

# Insert and DeleteMin commonalities

Idea of each:

1. Get the **structure** right first
   - Insert at end (bottom of tree)
   - Move the last element to the root after deleting the root
2. Restore the heap-order property by percolating (swapping an element/child pair)
   - Insert by percolating *up*: swap with parent
   - Delete by percolating *down*: swap with child with min value

# Summary: Implementing a Priority Queue as a binary heap

- Worst case times:
  - findMin:  O(1)
  - insert: amortized O(log n), worst O(n)
  - deleteMin O(log n)

- Big-O (amortized) times for insert/delete are the same as for balanced BSTs, but ..
  - Heap operations are *much* simpler to write.
  - A heap doesn't require additional space for pointers, balance codes or R-B colors, etc.
  - BinaryHeap findMin is faster.