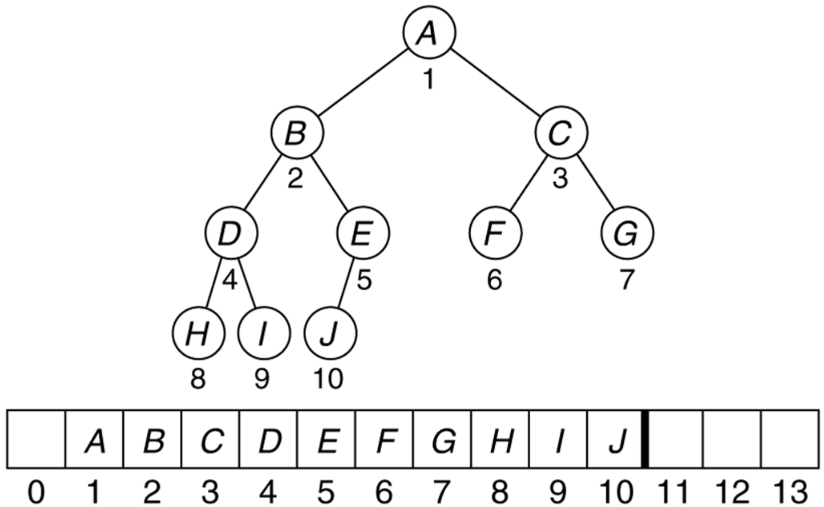


CSSE 230 Day 26

Priority Queues
Heaps
Heapsort

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

… explain why you can build a heap in O(n) time

… implement heapsort

Reminders

- Complete the **Doublets partner(s) evaluation** by tonight.

- Choose a partner today for last program: **SortingRaces** using signup sheet

Priority Queue ADT

Basic operations
Implementation options

Priority Queue operations

- Each element in the PQ has an associated **priority**, which is a value from a comparable type (in our examples, an integer).
- Operations (may have other names):
  - findMin()
  - insert(item, priority) (also called add,offer)
  - deleteMin( ) (also called remove or poll)
  - isEmpty() …

Priority queue implementation

- How could we implement it using data structures that we already know about?
  - Array?
  - Queue?
  - List?
  - BinarySearchTree?
- One efficient approach uses a binary heap
  - A somewhat-sorted complete binary tree
- 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

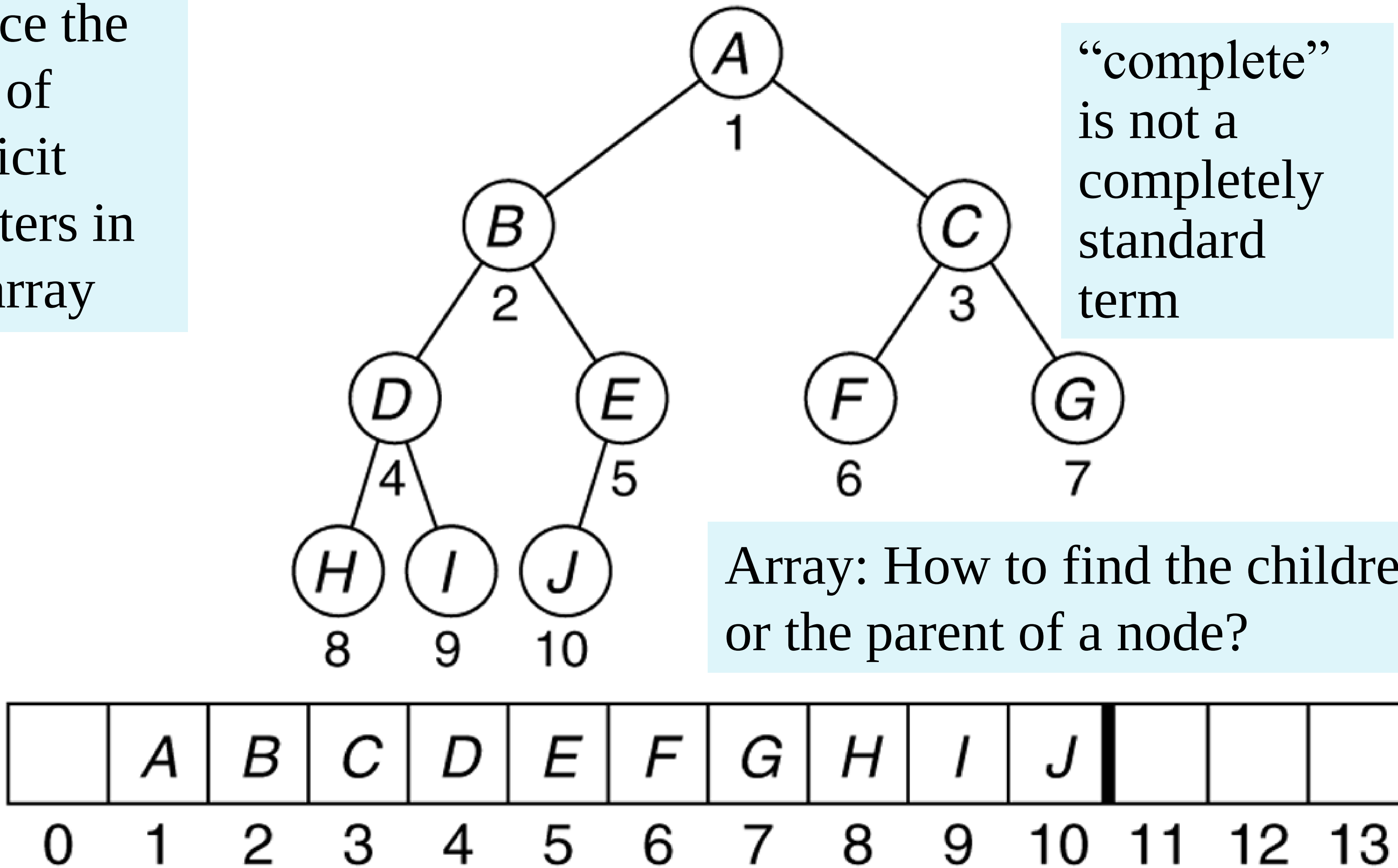
An efficient implementation of the PriorityQueue ADT

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

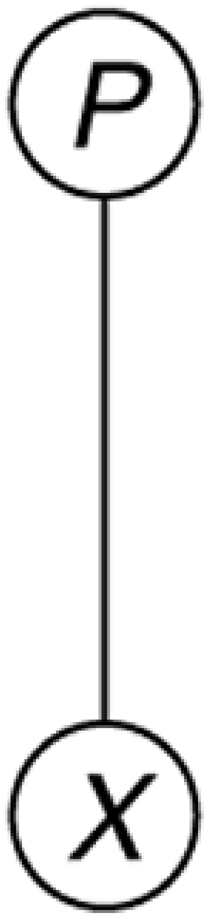


“complete” is not a completely standard term

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



$$P \le X$$

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.

In a binary heap, where do we find

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

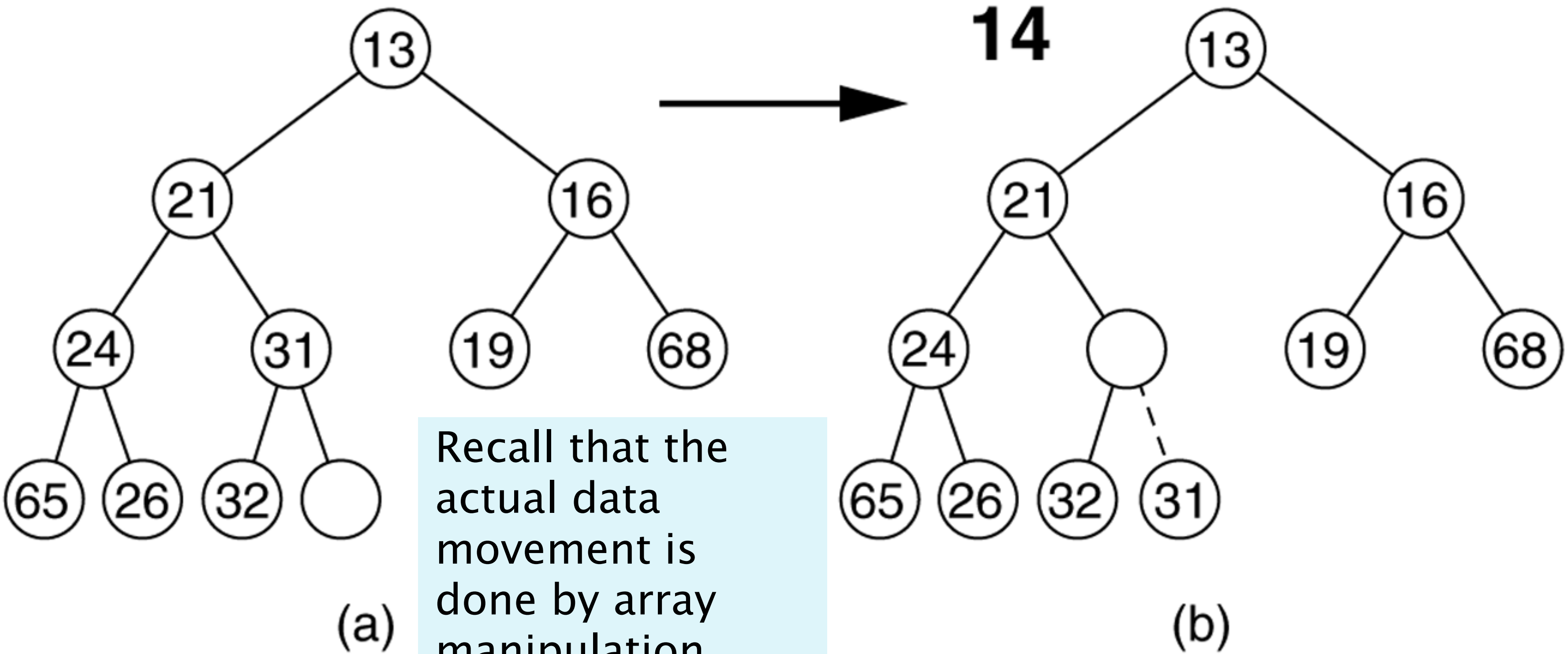
Insert and DeleteMin

- 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
     - Delete 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

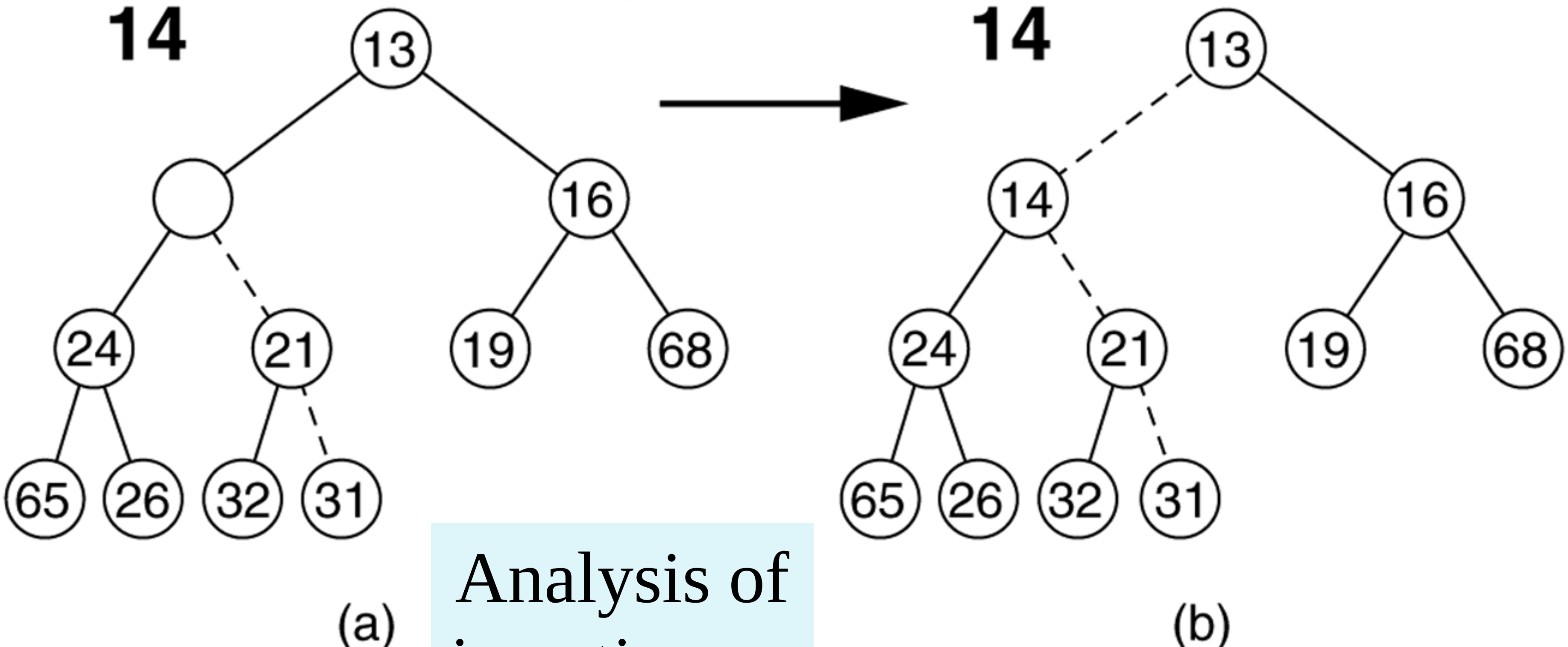


Recall that the actual data movement is done by array manipulation

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

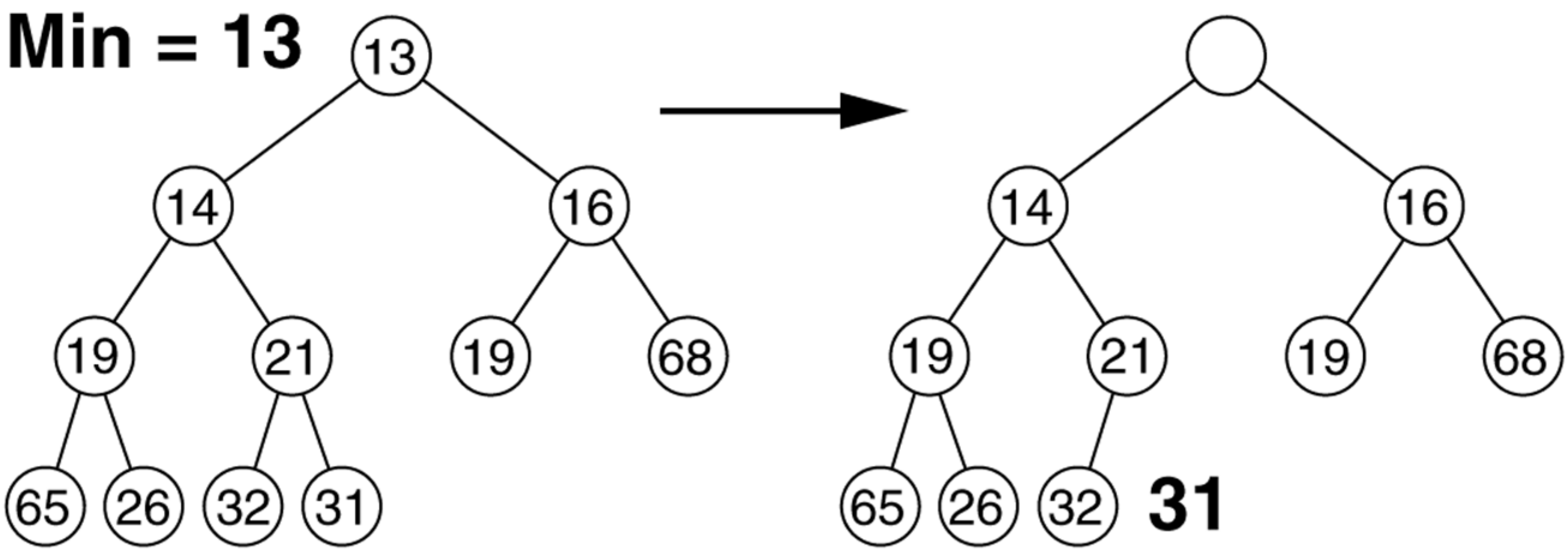
The add method

Your turn: 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

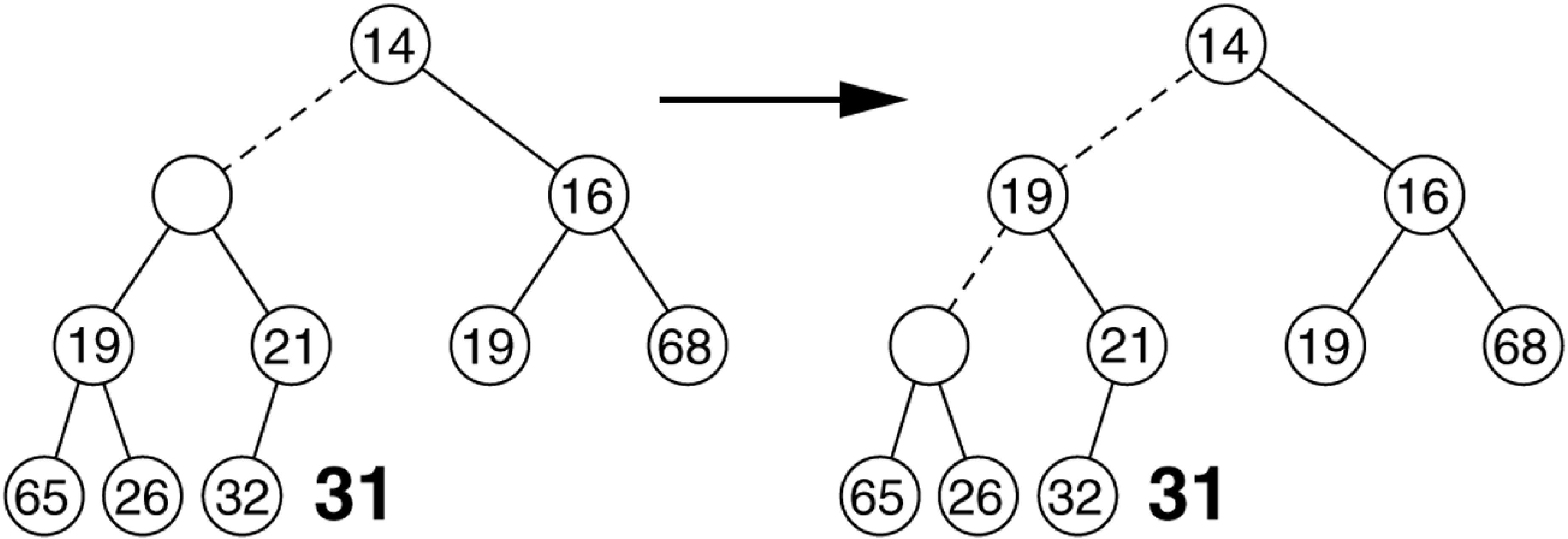
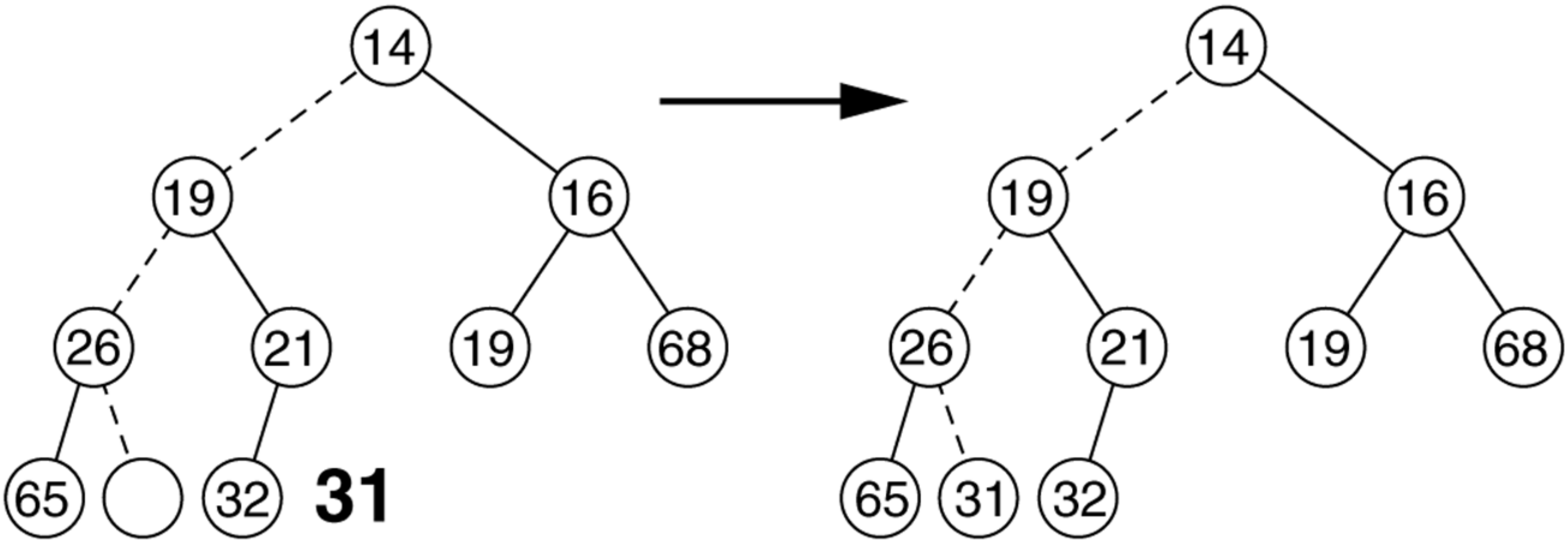


Figure 21.12
The last two steps in the deleteMin operation

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

Summary: Implementing a Priority Queue as a binary heap

- Worst case times:
  - findMin: O(1)
  - insert: O(log n)
  - deleteMin O(log n)
- big-oh times for insert/delete are the same as in the balanced BST implementation, but ..
  - Heap operations are much simpler to write.
  - A heap doesn't require additional space for pointers or balance codes.

Binary Heaps worktime

Read Heaps and heapsort instructions

You may leave early if you finish the heap implementation. Otherwise aim to finish before next class

Tomorrow: heapsort

Reminder: Doublets evals due today at midnight.

Heapsort

Use a binary heap to sort an array.

Using data structures for sorting

- Start with an empty structure.
- Insert each item from the unsorted array into the data structure
- Copy the items from the data structure, one at a time, back into the array, overwriting the unsorted data.

- What data structures work in this scheme?
- What is the runtime?

Using a Heap for sorting

- Start with empty heap
- Insert each array element into heap
- Repeatedly do deleteMin, copying elements back into array.
- One alternative for space efficiency:
  - We can save space by doing the whole sort in place, using a "maxHeap" (i.e. a heap where the maximum element is at the root instead of the minimum)
  - http://www.cs.usfca.edu/~galles/visualization/HeapSort.html
- Analysis?
  - **Next slide …**

Analysis of simple heapsort

- Add the elements to the heap
  - Repeatedly call insert O(n log n)
- Remove the elements and place into the array
  - Repeatedly call deleteMin O(n log n)
- Total O(n log n)
- Can we do better for the insertion part?
  - Yes, insert all the items in arbitrary order into the heap's internal array and then use **BuildHeap** (next)

BuildHeap takes a complete tree that is not a heap and exchanges elements to get it into heap form

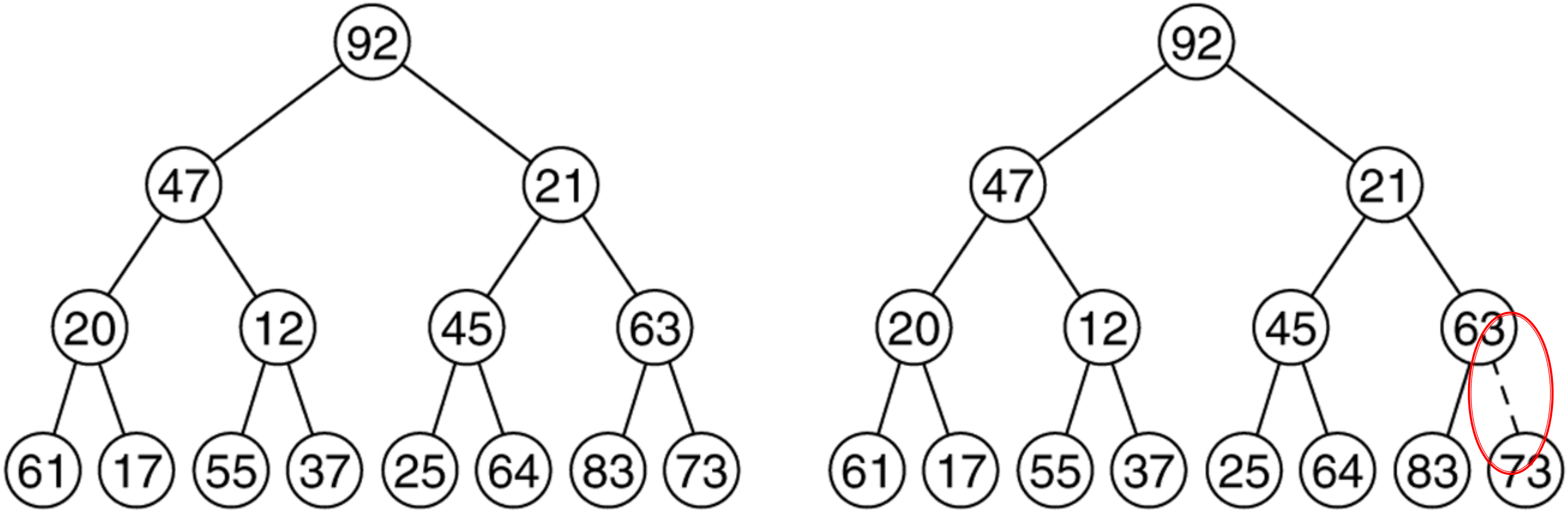
At each stage it takes a root plus two heaps and "percolates down" the root to restore "heapness" to the entire subtree

```
/**
 * Establish heap order property from an arbitrary
 * arrangement of items. Runs in linear time.
 */
private void buildHeap( )
{
    for( int i = currentSize / 2; i > 0; i-- )
        percolateDown( i );
}
```

Why this starting point?

Figure 21.17 Implementation of the linear-time buildHeap method

```
private void buildHeap( )
{
    for( int i = currentSize / 2; i > 0; i-- )
        percolateDown( i );
}
```



(a) (b)

Figure 21.18

(a) After percolateDown(6);
(b) after percolateDown(5)

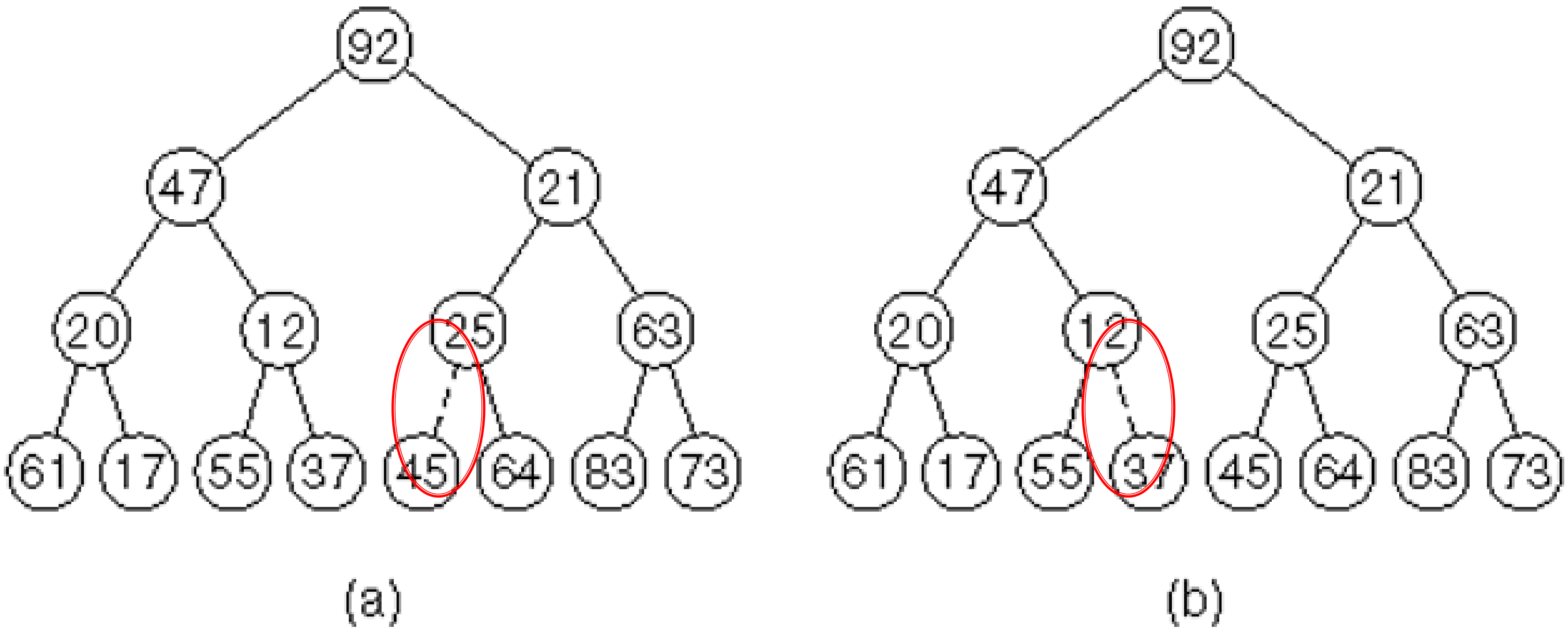


Figure 21.19

(a) After percolateDown(4);
(b) after percolateDown(3)

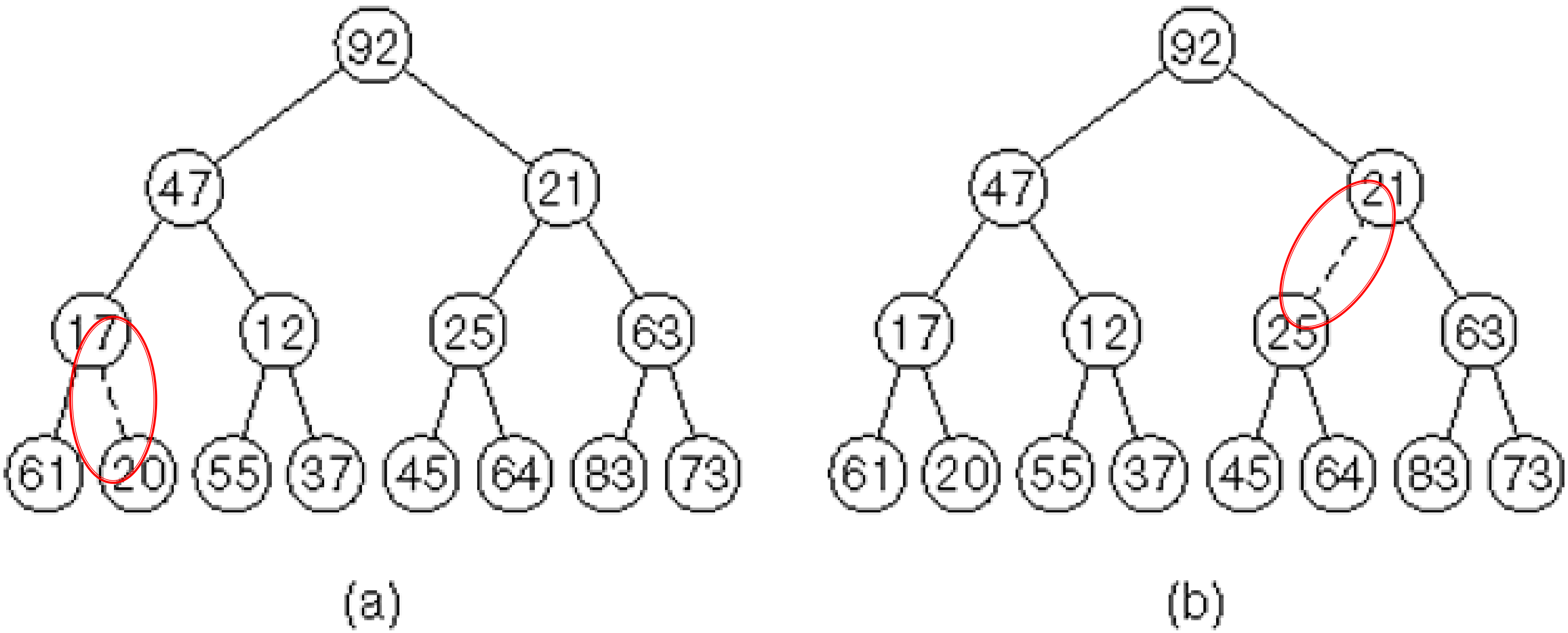
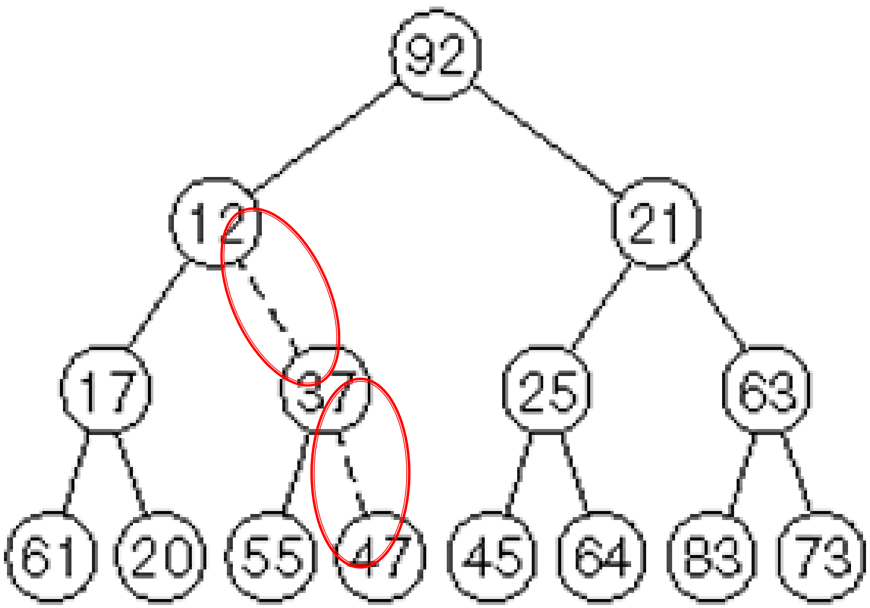


Figure 21.20
(a)After percolateDown(2);
(b) after percolateDown(1) and buildHeap terminates



(a) (b)

Analysis of BuildHeap

- Find a summation that represents the maximum number of comparisons required to rearrange an array of $N=2^{H+1}-1$ elements into a heap
- Can you find a summation and its value?

Analysis of better heapsort

- Add the elements to the heap
  - ~~Insert n elements into heap~~ (call buildHeap, faster)
- Remove the elements and place into the array
  - Repeatedly call deleteMin
- Total runtime?
  - **θ(n log n)**
  - We should expect no faster to sort! Why not?

Worktime now…