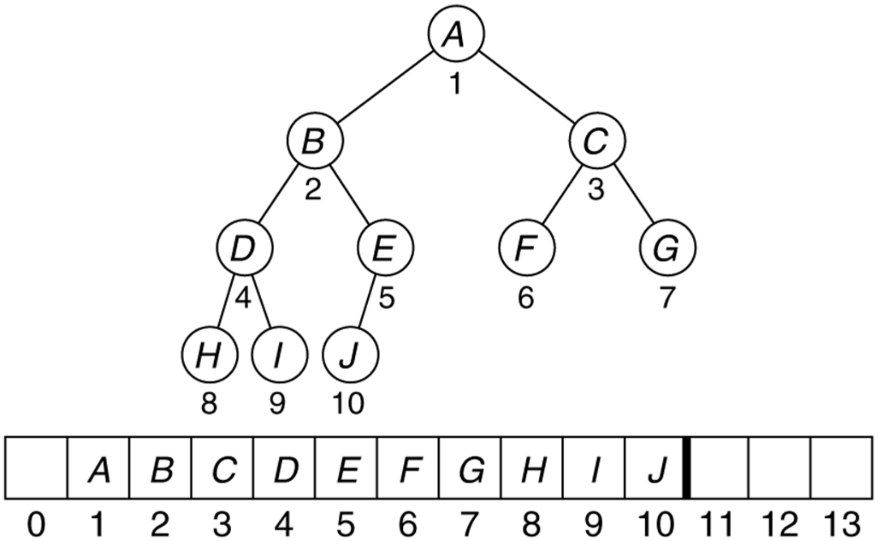


CSSE 230 Days 20-21

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

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 (but we won't ask you to write code yet)
- Practice exam posted in Moodle and code in repos

Abstract Data Type Review

- An ADT has an interface, and an implementation based on an underlying data structure.
- Review some ADTs that we have seen, and add two new ones

ADT	Underlying Data Structure
ArrayList	Growable Array, plus integers *capacity* and *size*
LinkedList	ListNodes, containing *data* and *next* pointer
BinaryTree	BinaryNodes, containing data, left, right pointers
HashTable	Growable Array (and possibly ListNodes)

- We will see how to implement PriorityQueue and BinaryHeap in terms of these underlying structures.

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):
  - insert(item, priority) (also called add,offer)
  - findMin()
  - deleteMin( ) (also called remove or poll)
  - isEmpty() …

Priority queue implementation

- What can we use for the internal representation of the abstract heap using data structures that we already know about?
  - Array?
  - Sorted array?
  - AVL?
- 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)

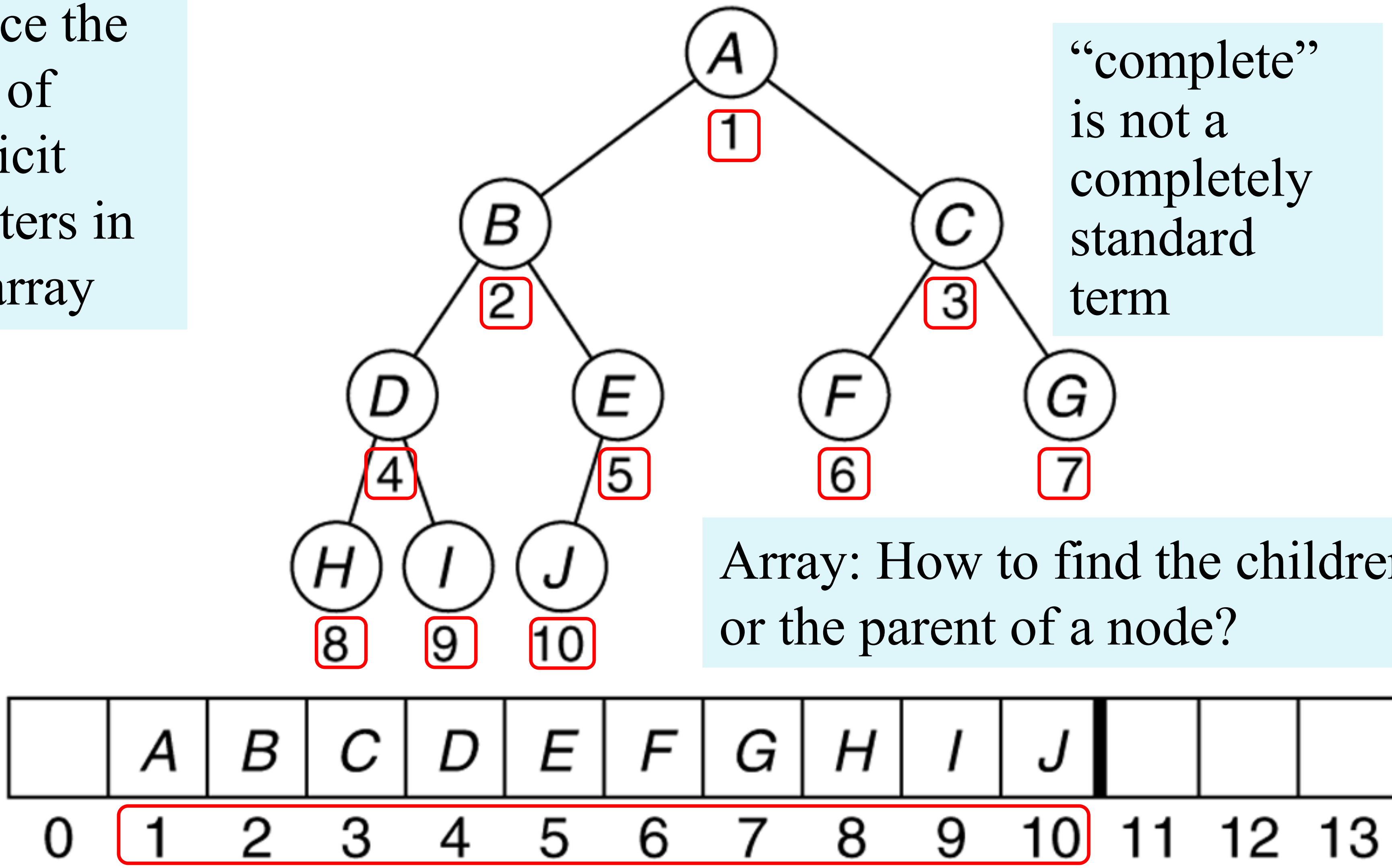
Algorithms for insertion and deleteMin

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



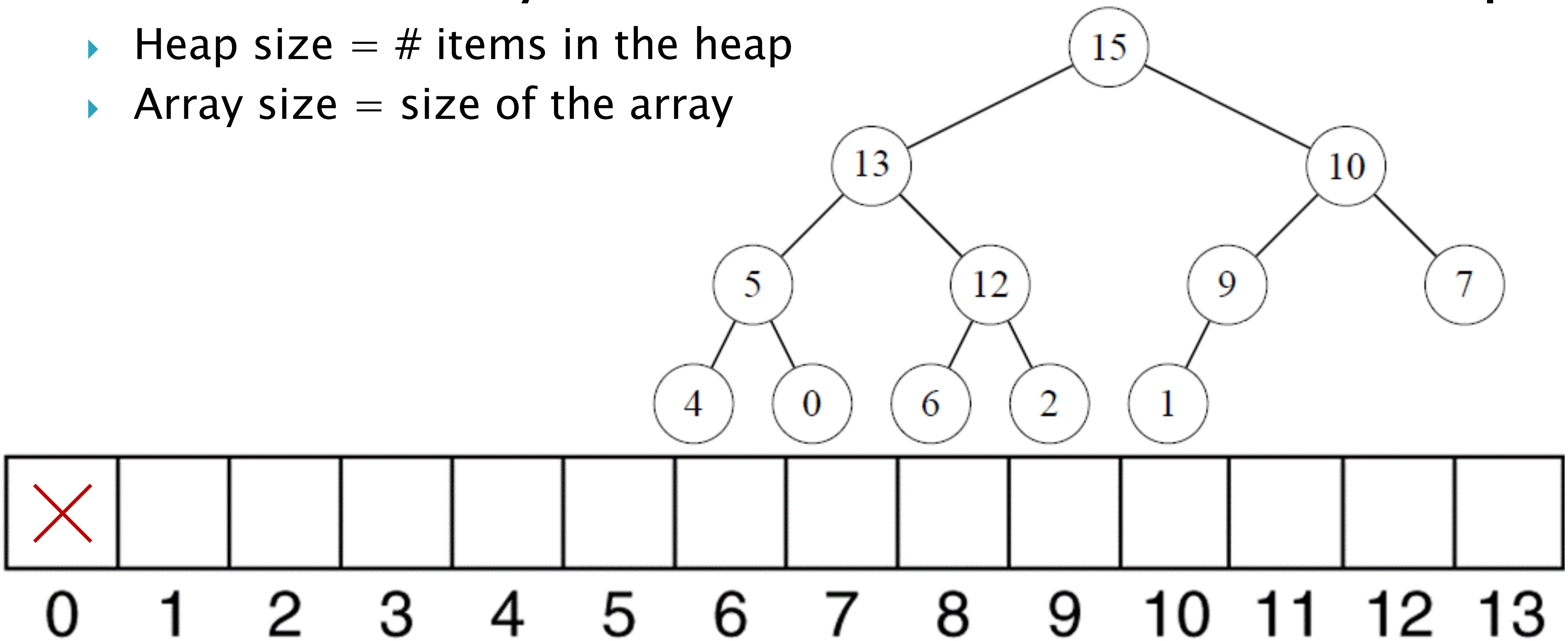
Array: How to find the children or the parent of a node?

One "wasted" array position (0)

Correspondence – Abstract Heap to Array Rep

Fill in the array with values from the *max* heap

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



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.

In a binary heap, where do we find

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

Insert and DeleteMin

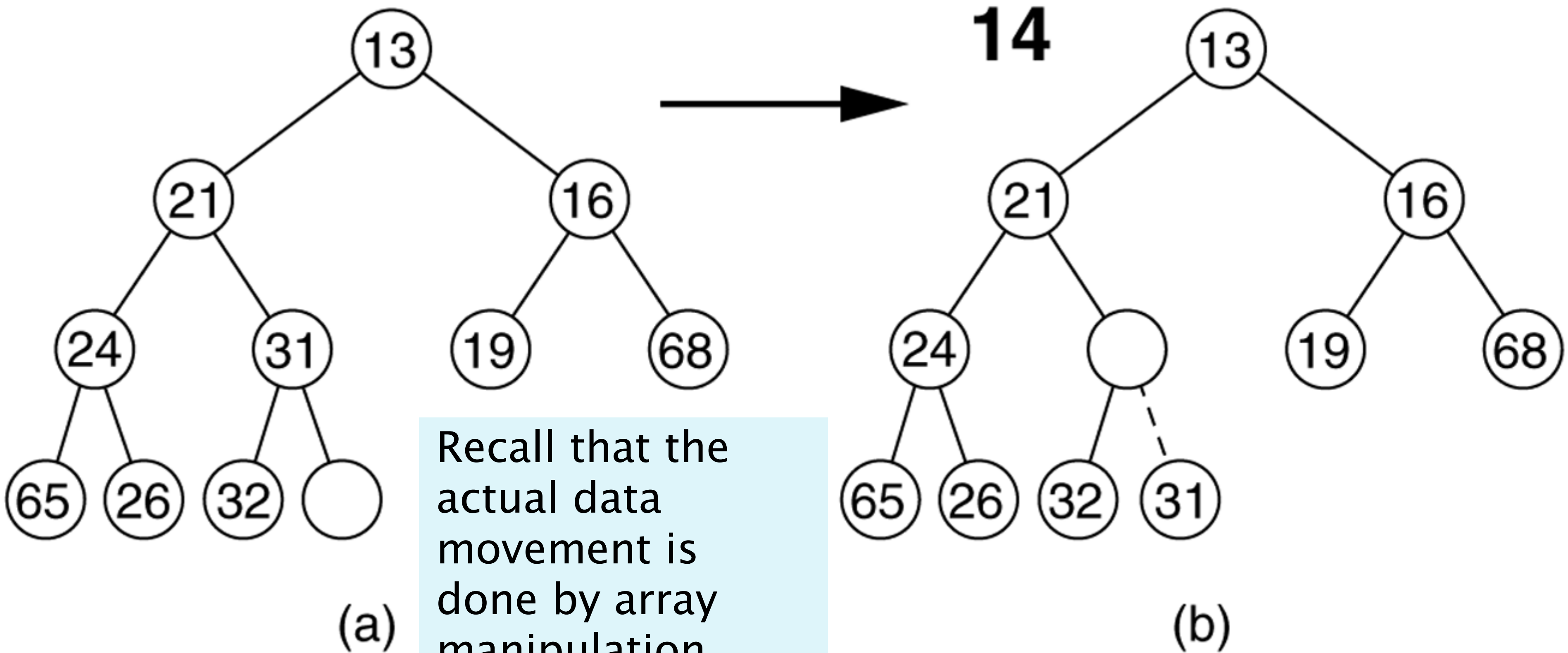
- Idea of each:
  1. Get the **structure** right first
     - Insert at end (bottom of tree)
     - Move the last element to the root after deleting the root
  2. Restore the heap-order property by percolating (swapping an element/child pair)
     - Insert by percolating *up*: swap with parent
     -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



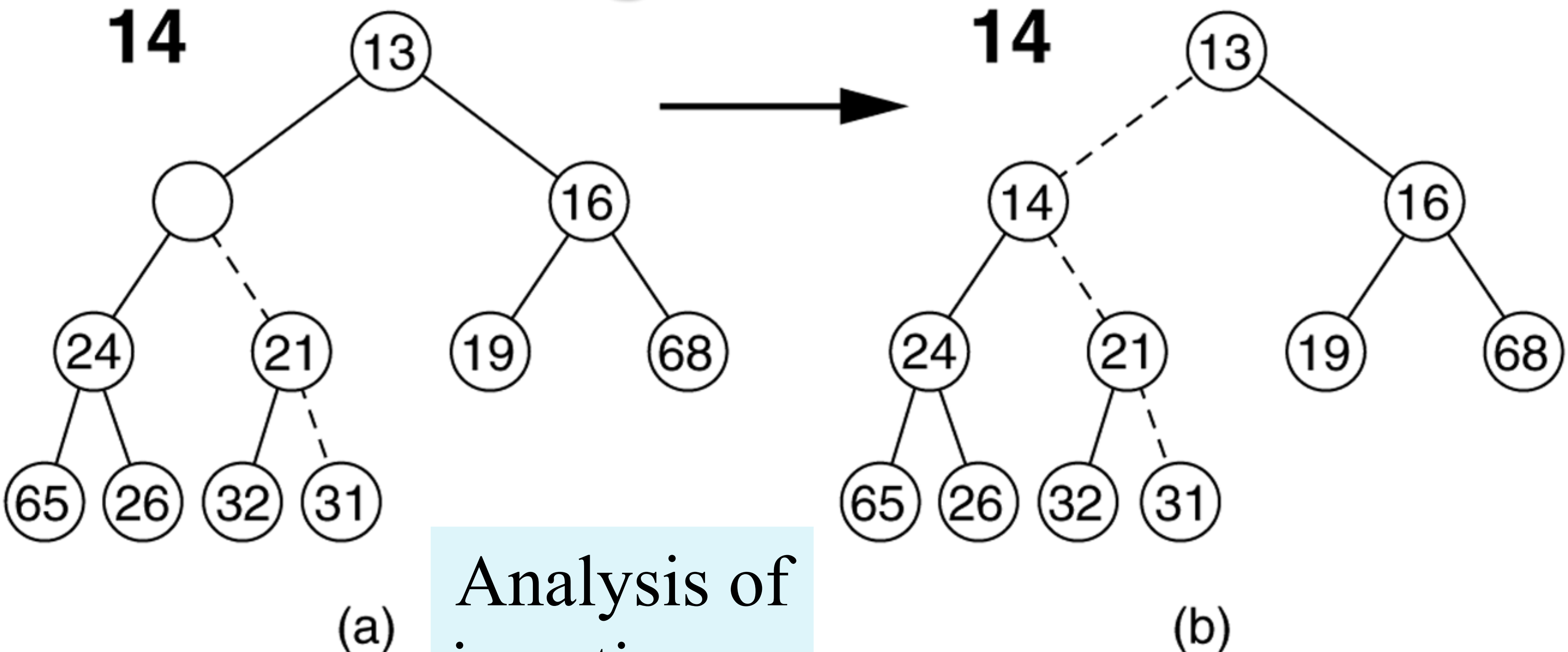
Recall that the actual data movement is done by array manipulation

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

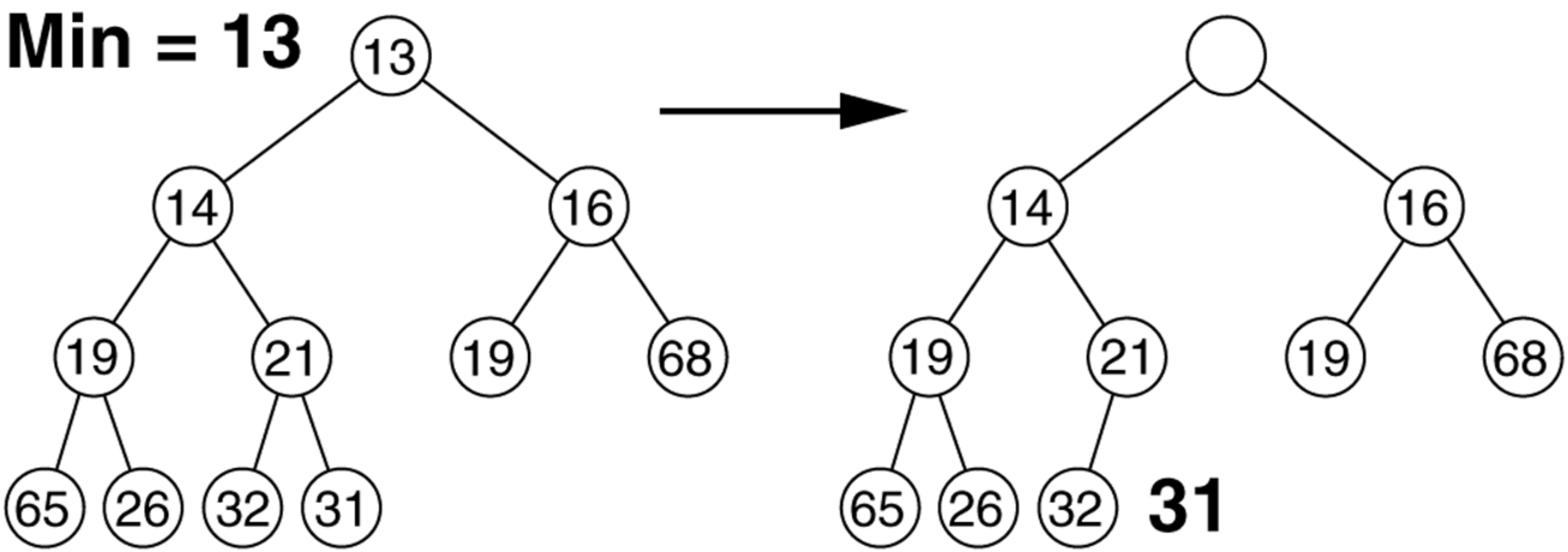
The add method

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

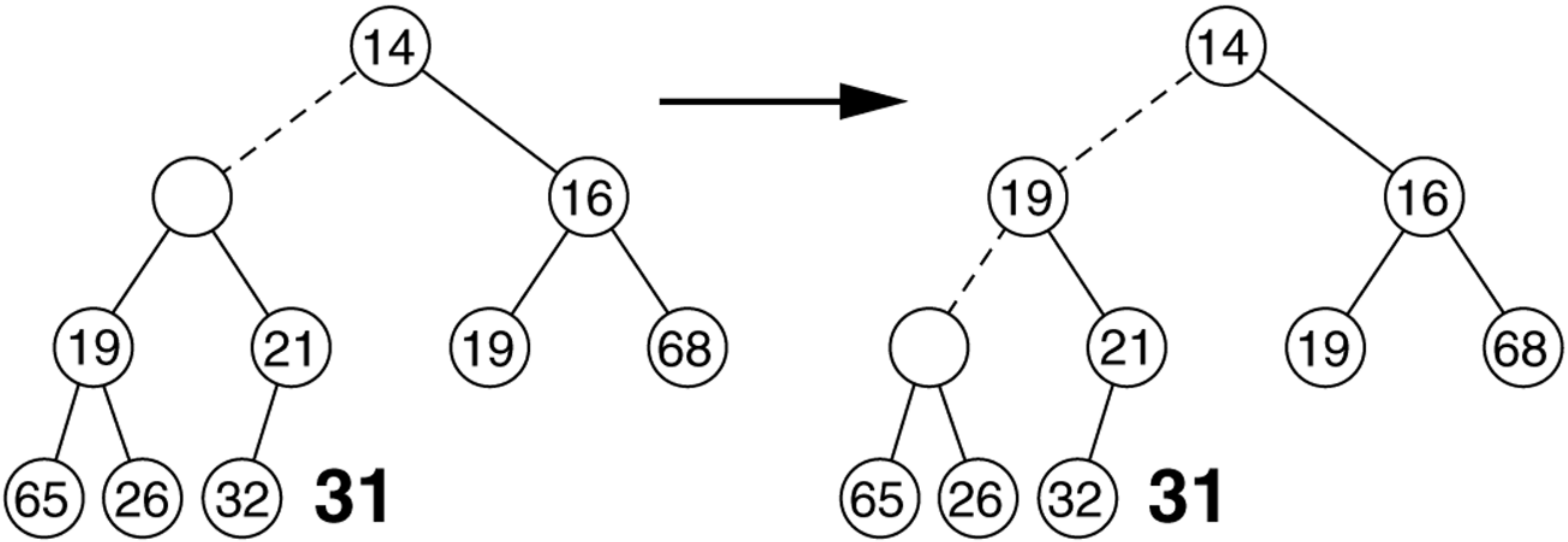
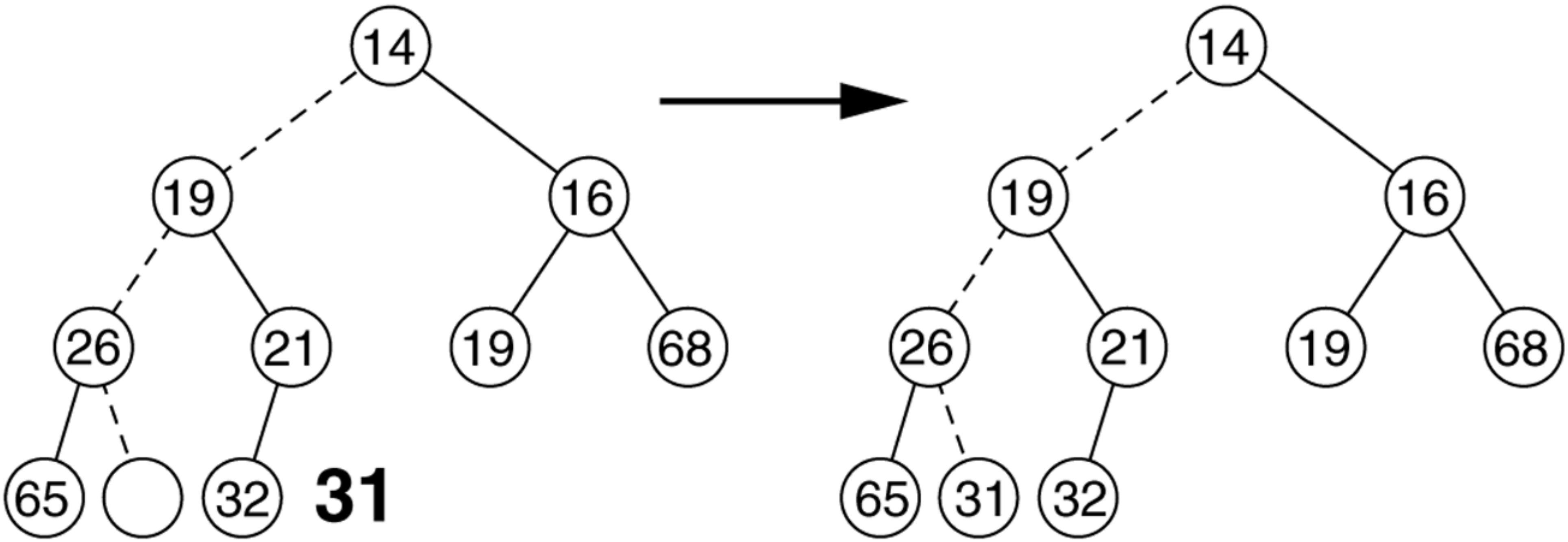


Figure 21.12
The last two steps in the deleteMin operation

DeleteMin Slide 3




```java
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-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.

Stop here for today

Heapsort

Use a binary heap to sort an array.

Using data structures for sorting

- Start with an unsorted array of data and a separate *other* empty data structure
- Insert each item from the unsorted array into the other data structure
- Copy the items from the other data structure (selecting the most extreme item first, then the next most extreme, etc.) one at a time, back into the original array, overwriting its unsorted data

- What data structures work for the other structure in this scheme?
  - BST? (Do it) Hash set? Priority queue, Heap?
- What is the runtime?

Using a Heap for sorting

- Start with empty heap
- Insert each array element into heap, being sure to maintain the heap property after each insert
- Repeatedly do deleteMin on the heap, copying elements back into array.
- Analysis?
  - **Next slide …**

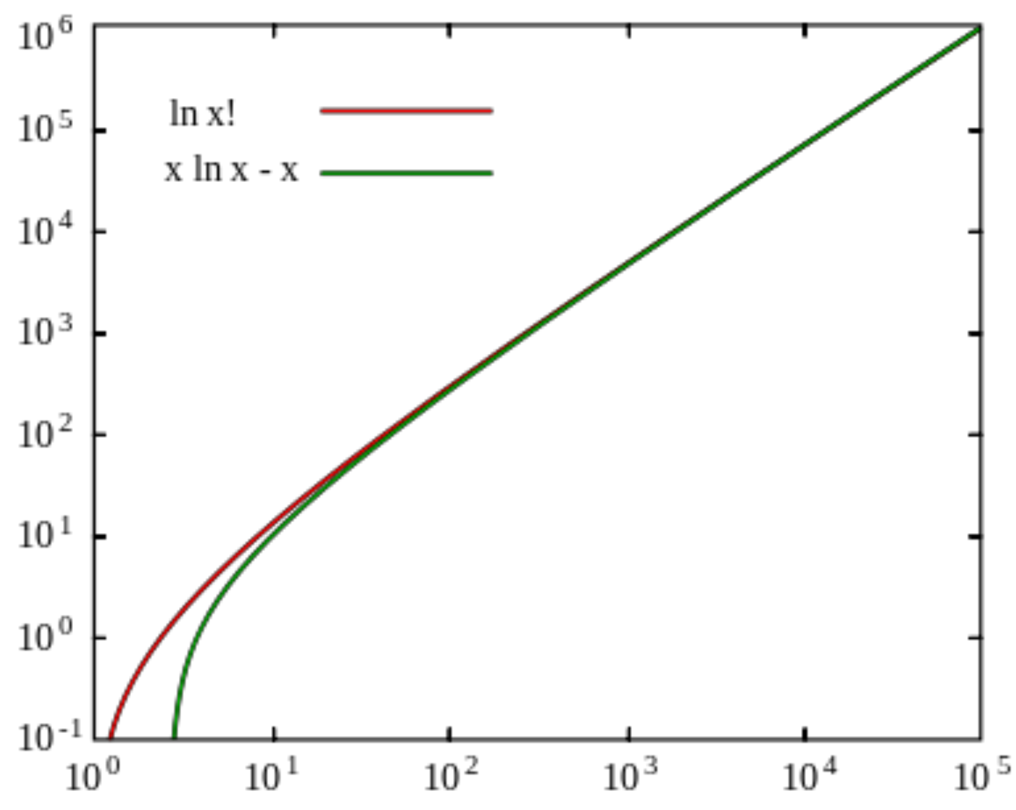
Analysis of simple heapsort

- Claim. $\log 1 + \log 2 + \log 3 + \cdots + \log N$ is $\Theta(N \log N)$.

Use Stirling's approximation:

$$\ln n! = n \ln n - n + O(\ln(n))$$

http://en.wikipedia.org/wiki/Stirling%27s_approximation



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, we don't need it to be a heap until we are ready to start deleting.
  - 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 );
}
```

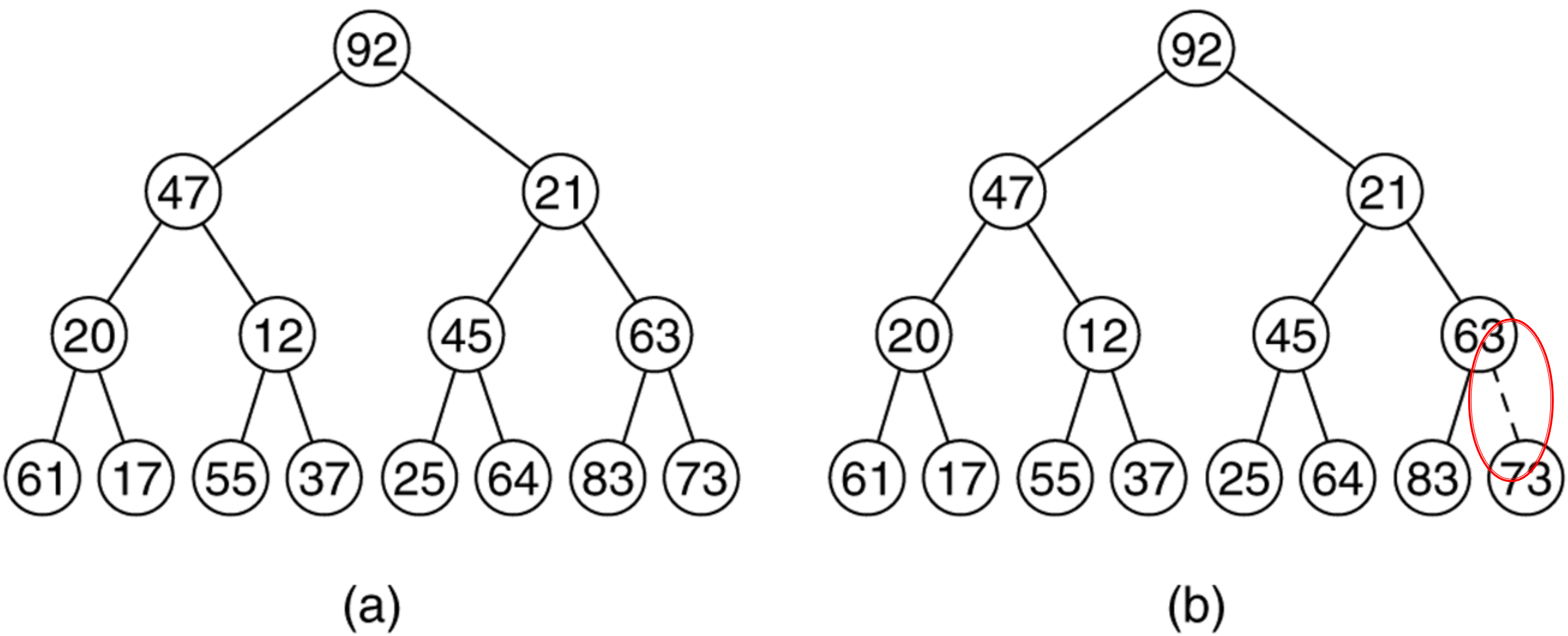


Figure 21.18

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

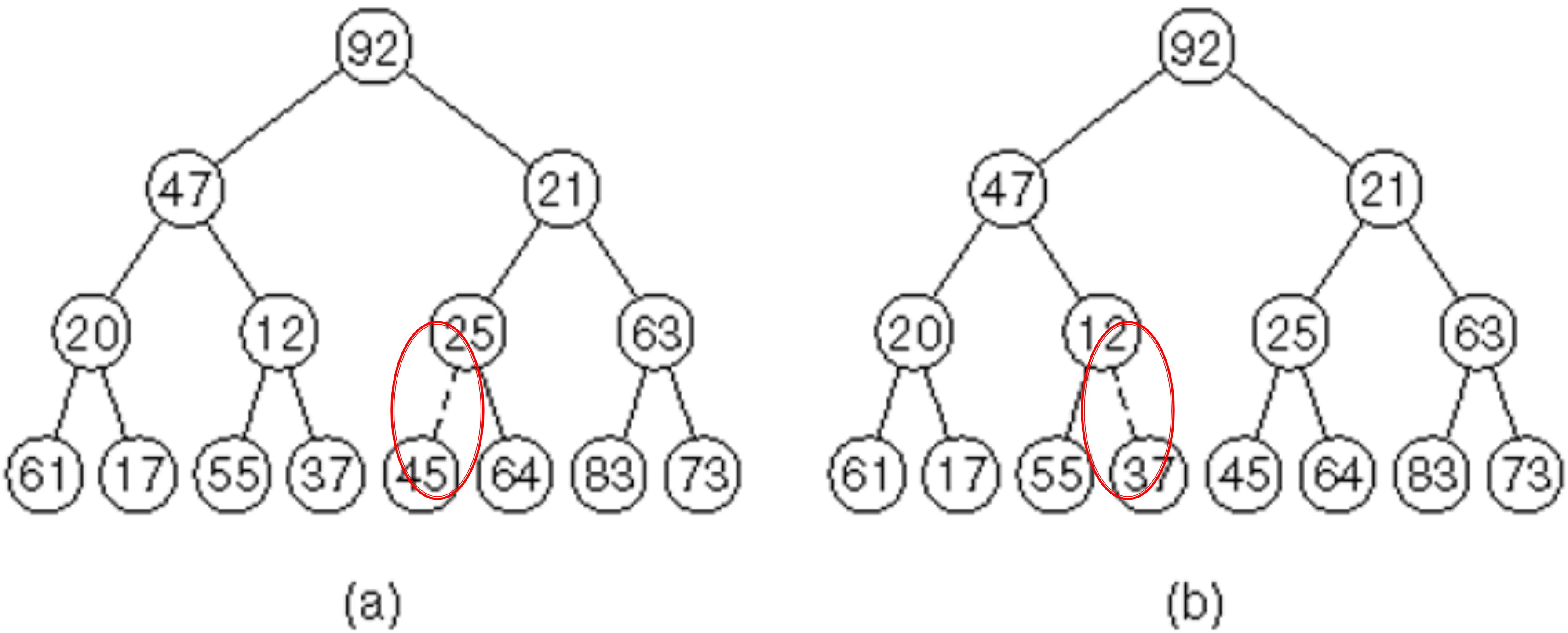
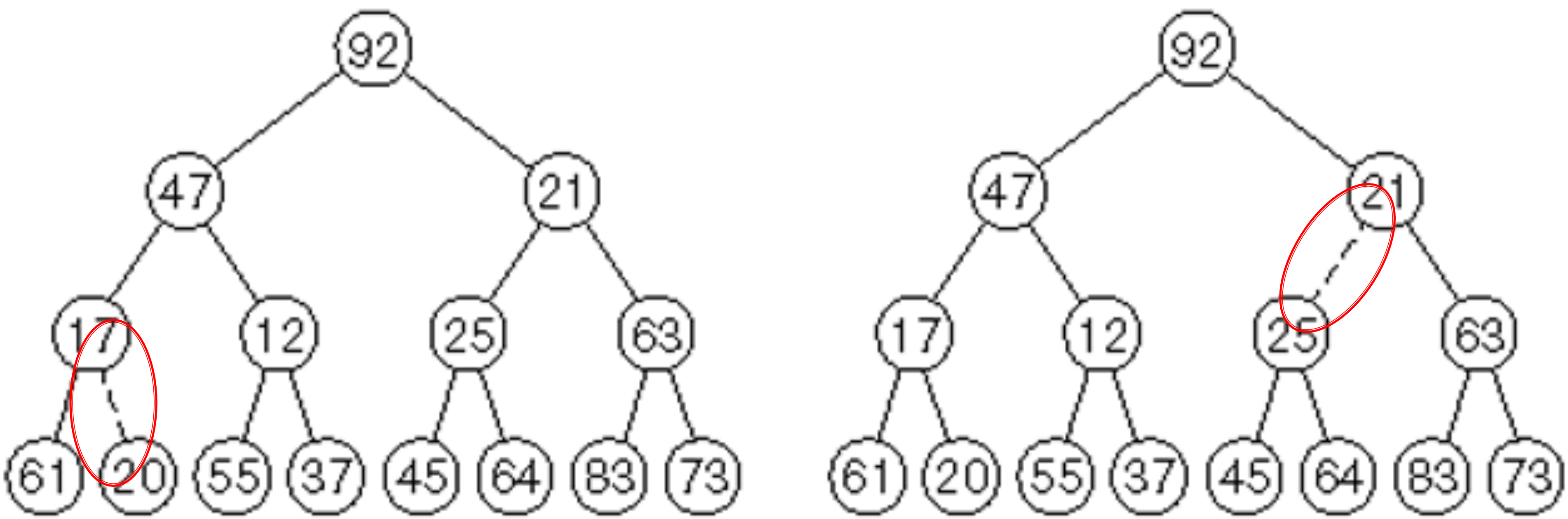


Figure 21.19

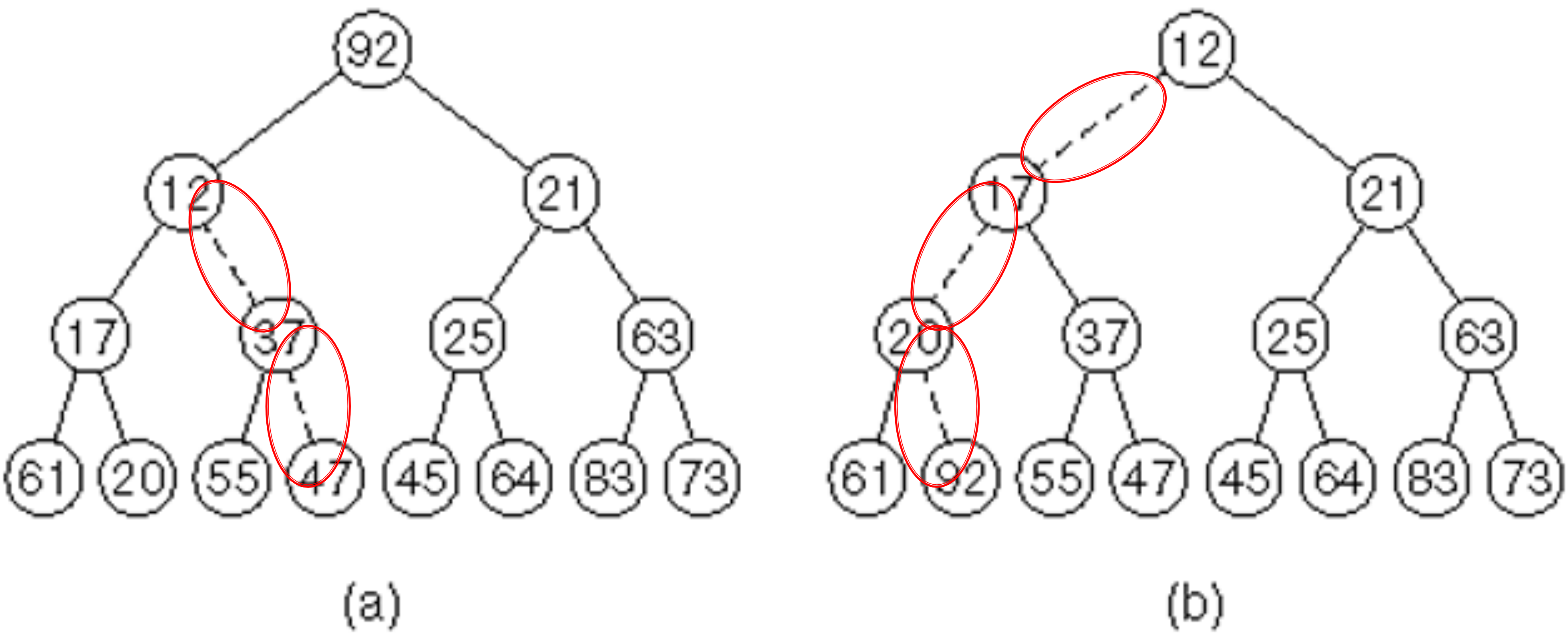
(a) After percolateDown(4);
(b) after percolateDown(3)



(a)

(b)

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



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
  - How many comparisons? The sum of the heights.
- Can you find a summation and its value?
- In HW8, you'll do this.

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
  - The summation is $\sum_{k=0}^{H} k\, 2^{H-k}$.

    and the sum is N – H – 1

- HW8: prove this formula by induction
  - Can do it strictly by the numbers
  - Simpler?: Do it based on the trees.

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

In-place heapsort

- With one final tweak, heapsort only needs O(1) extra space!

- Idea:
  - When we deleteMin, we free up space at the end of the heap's array.
  - Idea: write deleted item in just-vacated space!
  - Would result in a reverse-sort. Can fix in linear time, but better: use a max-heap. Then, comes out in order!

- http://www.cs.usfca.edu/~galles/visualization/HeapSort.html