

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

CSSE 230 Day 21

Heapsort

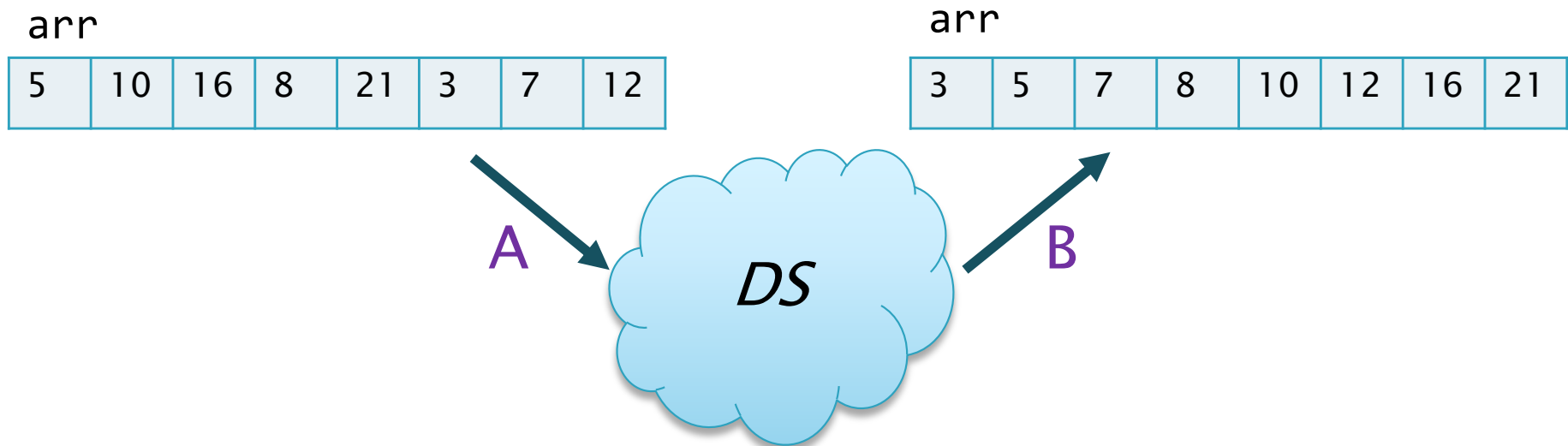
After this lesson, you should be able to ...
 ... explain how and why you can build a heap in $O(n)$ time
 ... implement heapsort

Sorting Problem



- Given array `arr` of Comparables, sort `arr`.

Idea: Using an auxiliary data structure for sorting



- Start with an empty *auxiliary data structure*, DS
- **Step A.** Insert each item from the unsorted array into DS
- **Step B.** Copy the items from DS (selecting the **most extreme item first**, then the next most extreme, etc.) one at a time, back into the original array
- What data structures work for DS ?
 - BST? Hash set? PQ/heap?

Naïve Heapsort

- Start with empty heap
- **Step A.** Insert each array element into heap, being sure to maintain the heap property after each insert
- **Step B.** Repeatedly run **deleteMin** on the heap, copying elements back into array.

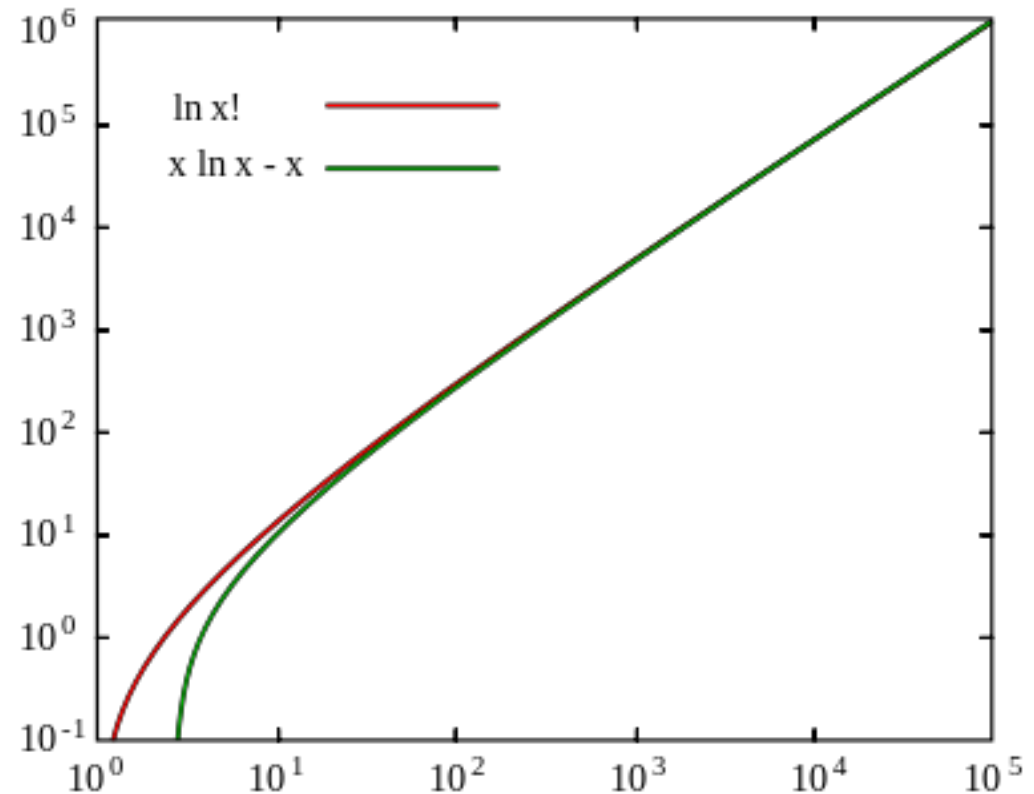
- Analysis?

Analysis of naïve heapsort

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

Use **Stirling's approximation**:
[Wikipedia link](#)

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



Analysis of naïve heapsort

- Add the elements to the heap
 - Repeatedly call insert $O(n \log n)$
- Copy the elements back to the array in order
 - 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

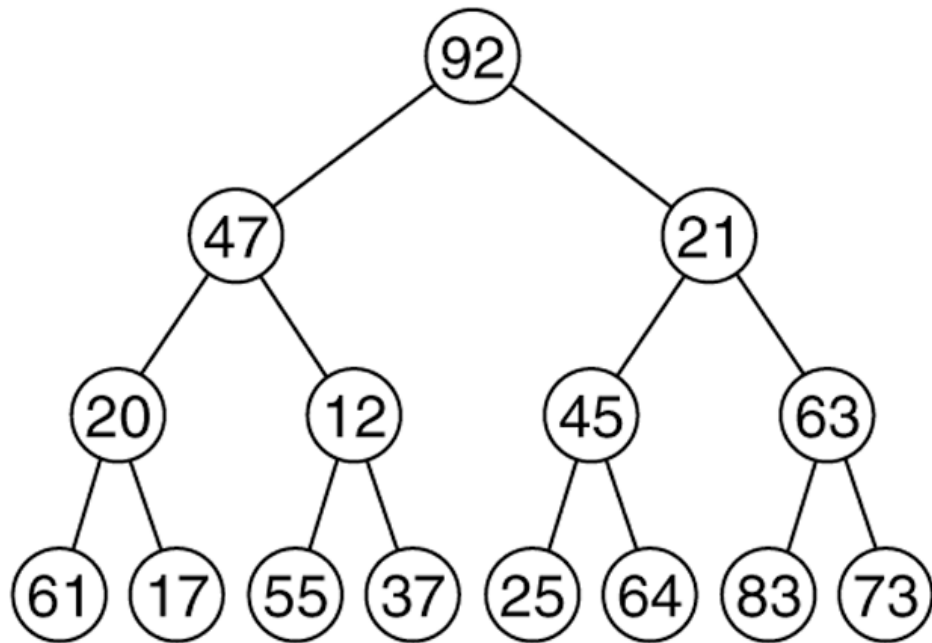
```
/**
 * Establish heap order property from an arbitrary
 * arrangement of items. Runs in linear time.
 */
private void buildHeap( )
{
    for( int i = currentSize / 2; i > 0; i-- )
        percolateDown( i );
}
```

Why this starting point?

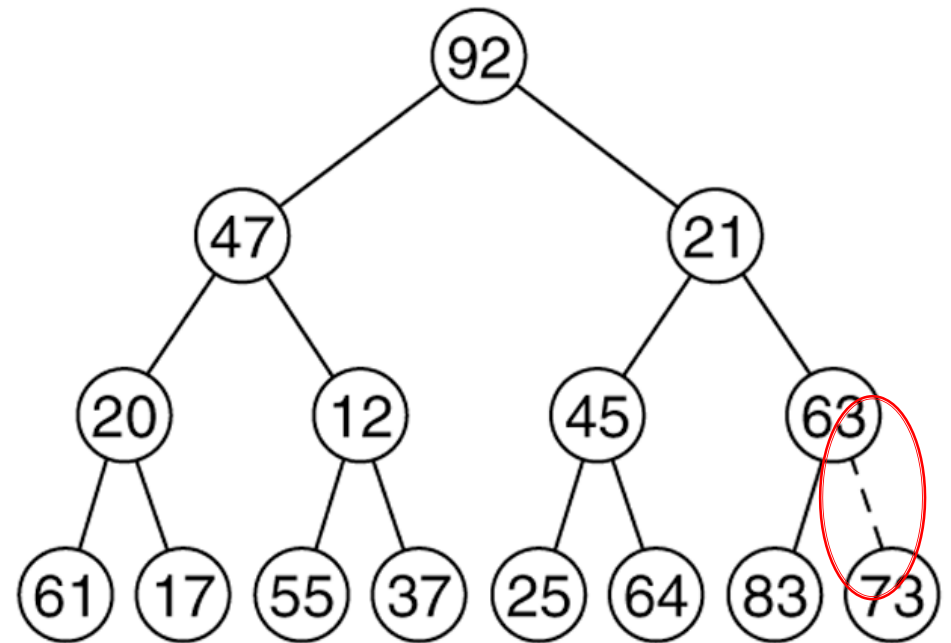


Figure 21.17 Implementation of the linear-time buildHeap method

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



(a)

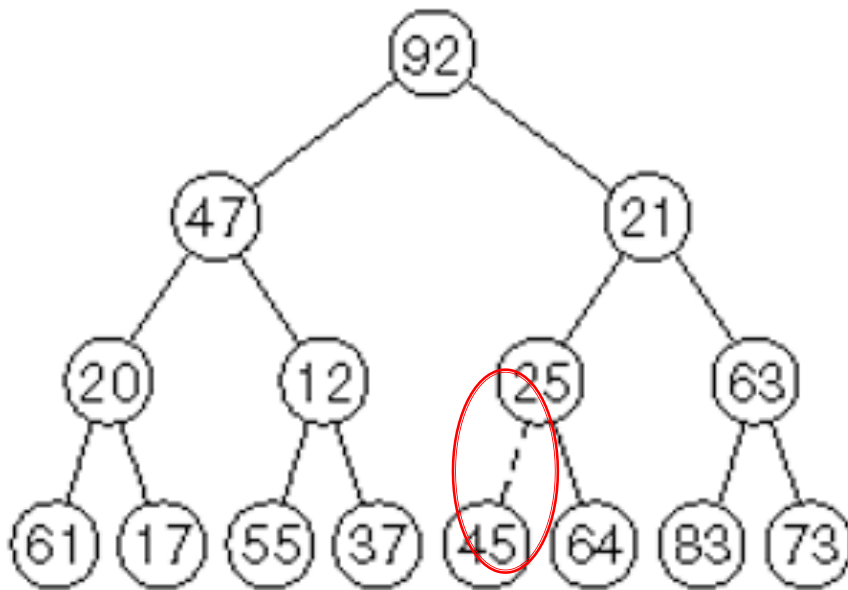


(b)

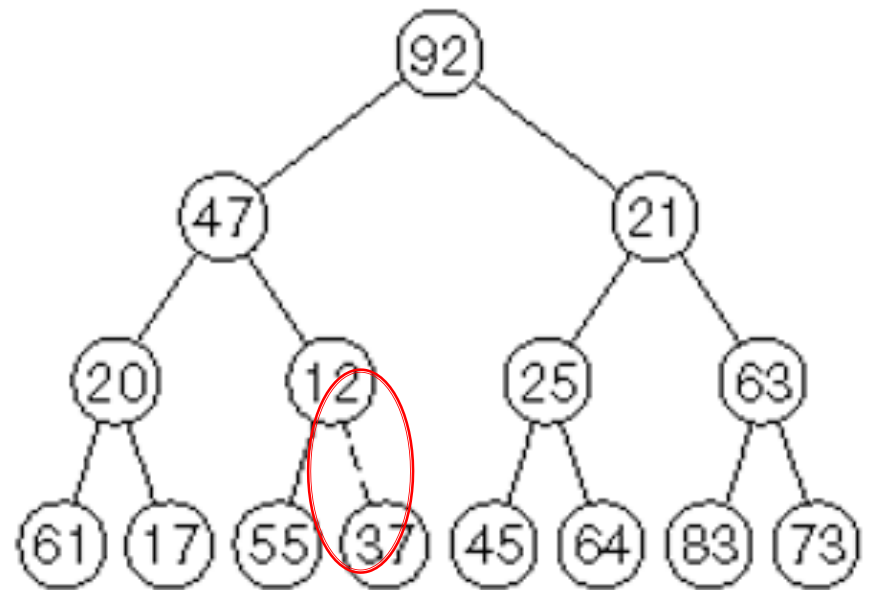
Figure 21.18

(a) After percolateDown(6);

(b) after percolateDown(5)



(a)

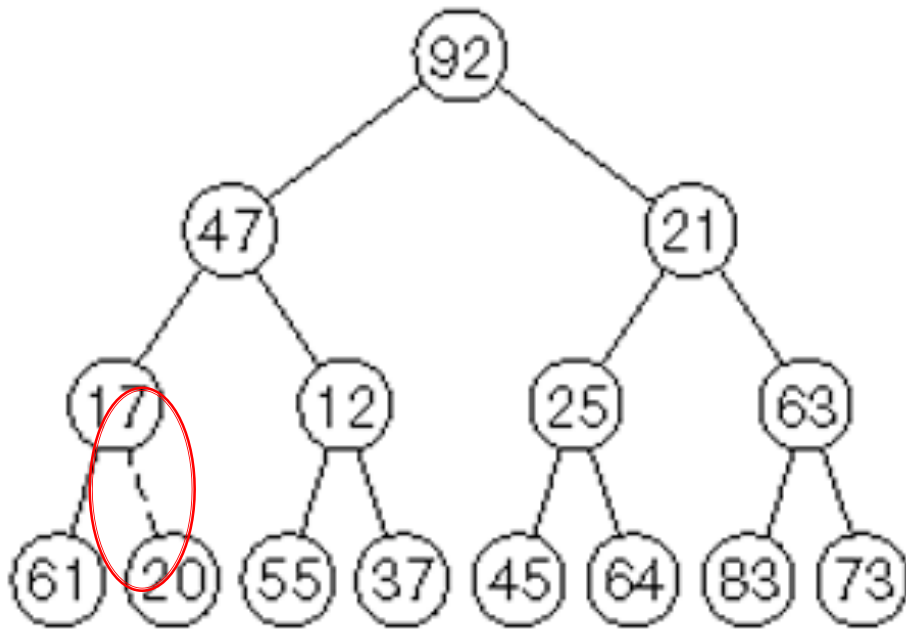


(b)

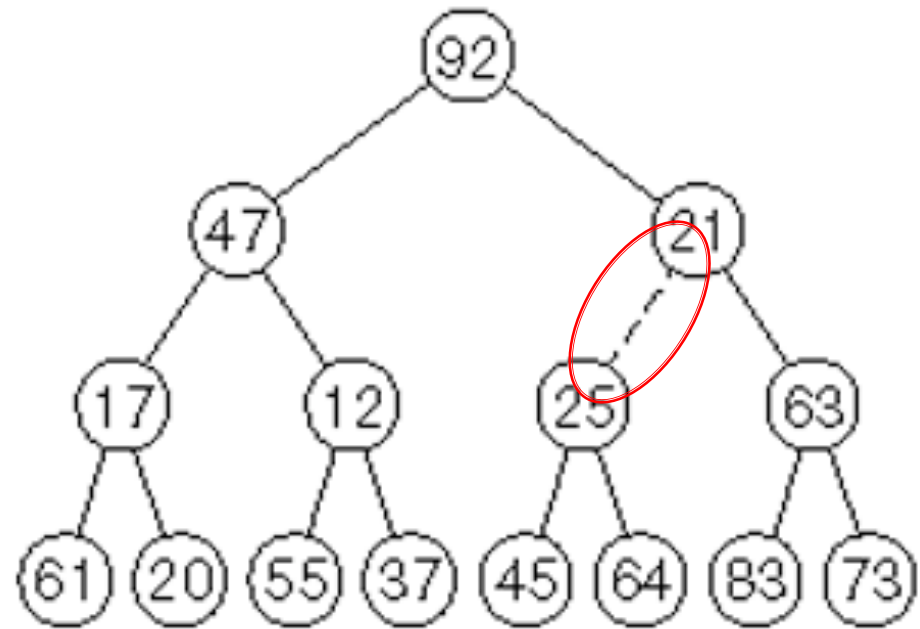
Figure 21.19

(a) After percolateDown(4);

(b) after percolateDown(3)



(a)

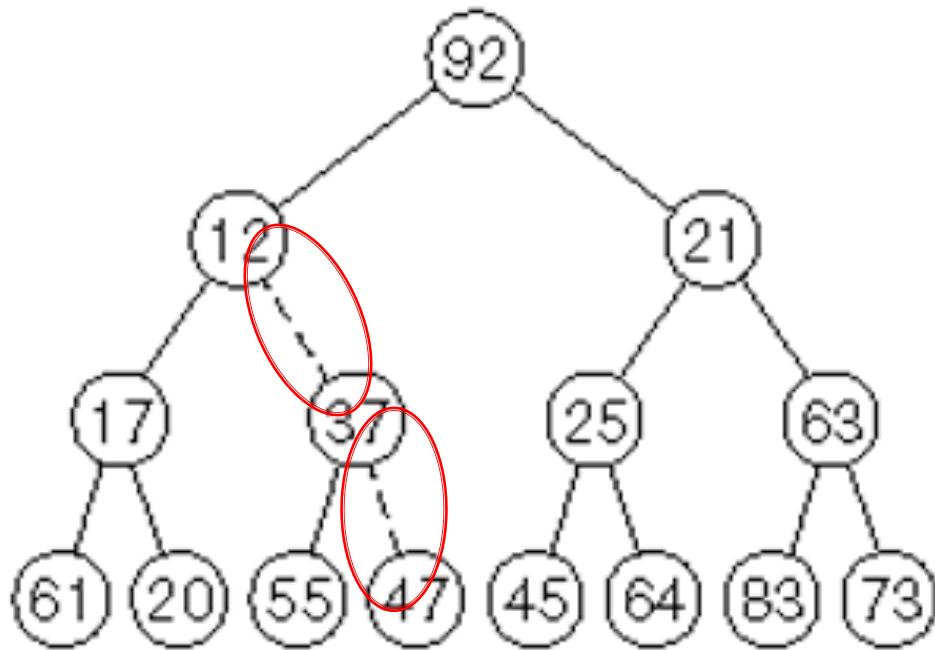


(b)

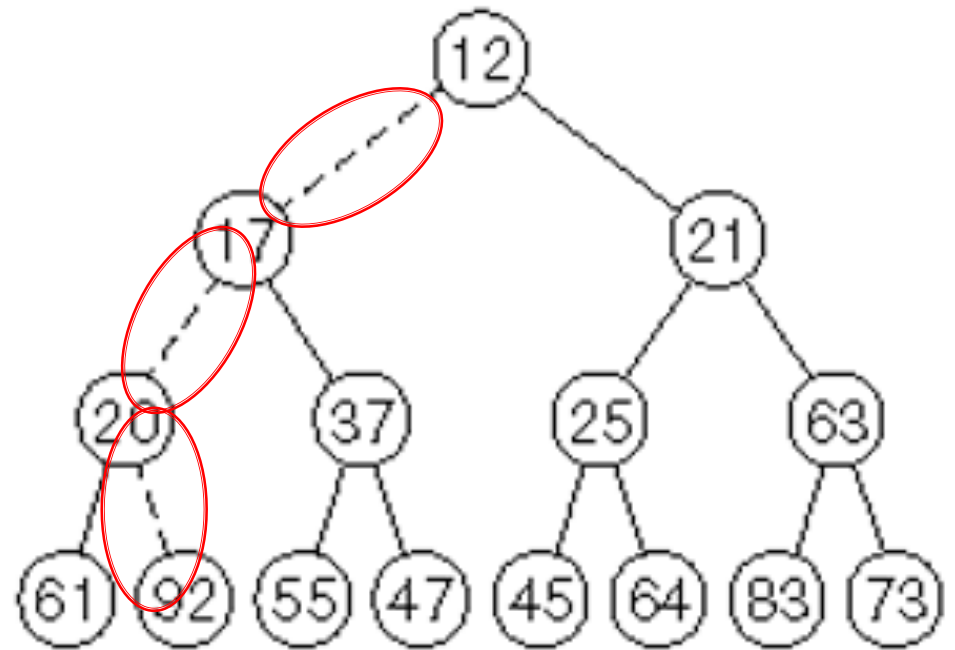
Figure 21.20

(a) After `percolateDown(2)`;

(b) after `percolateDown(1)` and `buildHeap` terminates



(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
 - How many comparisons? The sum of the heights.
- Can you find a summation and its value?
- In HW8, you'll do this.
- Conclusion: buildHeap is $O(N)$

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>