CSE 373
Data Structures and Algorithms

Lecture 15: Priority Queues (Heaps) III

Generic Collections

Generics and arrays

```
public class Foo<T> {
    private T myField;                 // ok


    public void method1(T param) {
        myField = new T();             // error
        T[] a = new T[10];             // error
    }
}
```

- You cannot create objects or arrays of a parameterized type.
  - Why not?

Generics/arrays, fixed

```
public class Foo<T> {
    private T myField;                          // ok


    public void method1(T param) {
        myField = param;                        // ok
        T[] a2 = (T[])(new Object[10]);         // ok
    }
}
```

- But you can declare variables of that type, accept them as parameters, return them, or create arrays by casting Object[].

The `compareTo` method

- The standard way for a Java class to define a comparison function for its objects is to define a `compareTo` method.
  - Example: in the String class, there is a method:

```
public int compareTo(String other)
```

- A call of `A.compareTo(B)` will return:

a value	<	0	if A comes "before" B
a value	>	0	if A comes "after" B
or		0	if A and B are "equal”

Comparable

```
public interface Comparable<E> {
    public int compareTo(E other);
}
```

- A class can implement the `Comparable` interface to define a natural ordering function for its objects.

- A call to the `compareTo` method should return:

  a value < 0 if the `other` object comes "before" `this` one

  a value > 0 if the `other` object comes "after" `this` one

  or 0 if the `other` object is considered "equal" to `this`

`Comparable` template

```
public class name implements Comparable<name> {

    ...

    public int compareTo(name other) {
        ...
    }
}
```

- Exercise: Add a `compareTo` method to the `PrintJob` class such that `PrintJobs` are ordered according to their priority (ascending – lower priorities are more important than higher ones).

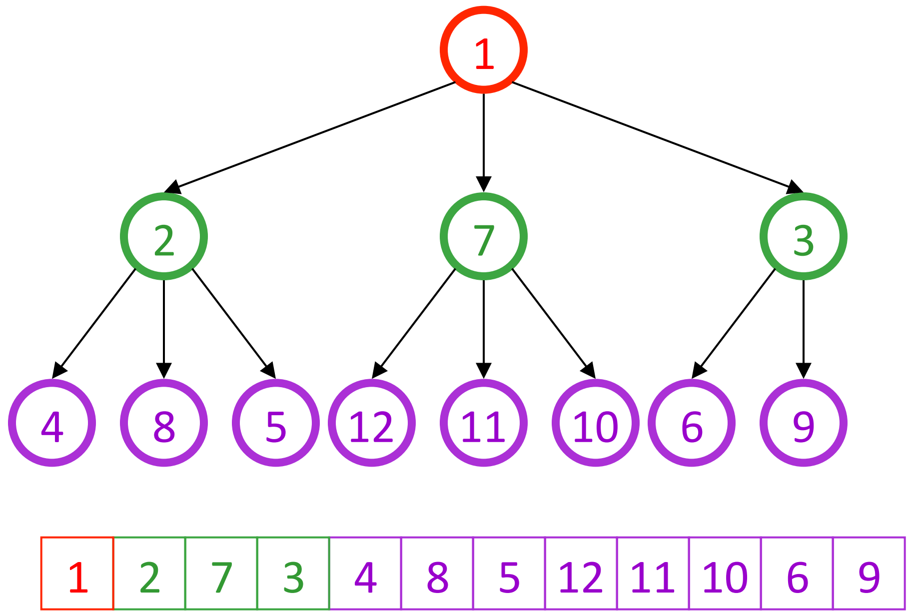
Comparable example

```
public class PrintJob implements Comparable<PrintJob> {
    private String user;
    private int number;
    private int priority;

    public PrintJob(int number, String user, int priority) {
        this.number = number;
        this.user = user;
        this.priority = priority;
    }

    public int compareTo(PrintJob otherJob) {
        return priority - otherJob.priority;
    }

    public String toString() {
        return number + " (" + user + "):" + priority;
    }
}
```

d-Heaps

Generalization: d-Heaps

- Each node has *d* children
- Still can be represented by array
- Good choices for *d* are a power of 2
  - Only because multiplying and dividing by powers of 2 is fast on a computer
- How does height compare to binary heap?



Operations on d-Heap

- insert: runtime = depth of tree decreases, $\Theta(\log_d n)$

- remove: runtime = bubbleDown requires more comparisons to find min, $\Theta(d \log_d n)$

- Does this help insert or remove more?

Other Priority Queue Operations

More Min-Heap Operations

- **`decreasePriority`**: reduce the priority value of an element in the queue
  - Solution: change priority and ______________________________
- **`increasePriority`**: increase the priority value of an element in the queue
  - Solution: change priority and ______________________________
- How do we find the element in the queue? What about duplicates?
  - Need a reference to the element!

More Min-Heap Operations

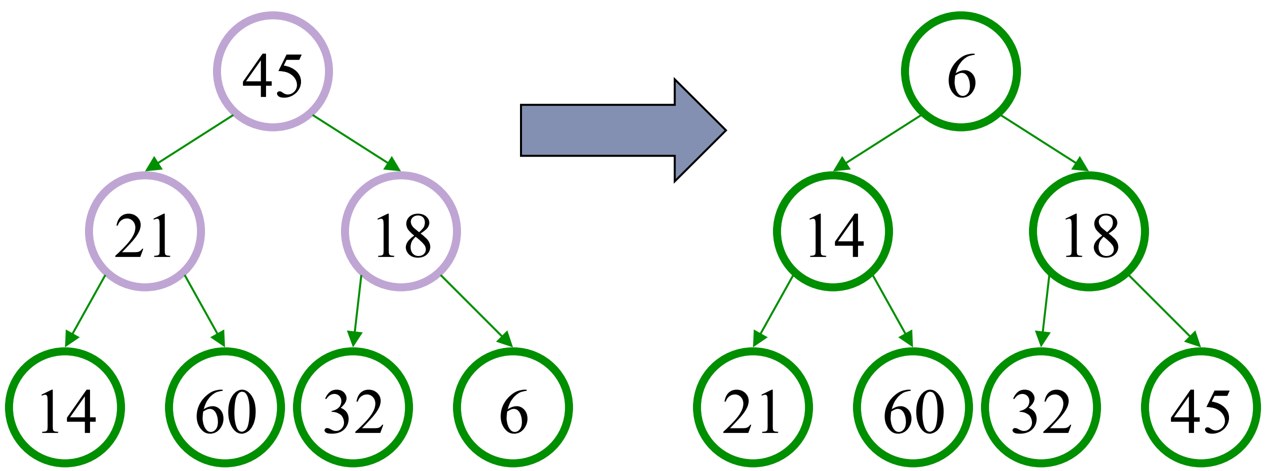
- **`remove`**: given a reference to an object in the queue, remove the object from the queue
  - Solution: set priority to negative infinity, percolate up to root and `deleteMin`
- **`findMax`**
  - Solution: Can look at all leaves, but not really the point of a min-heap!

Building a Heap

- Given a list of numbers, how would you build a heap?
- At every point, the new item may need to percolate all the way through the heap
- Adding the items one at a time is $\Theta(n \log n)$ in the worst case
- A more sophisticated algorithm does it in $\Theta(n)$

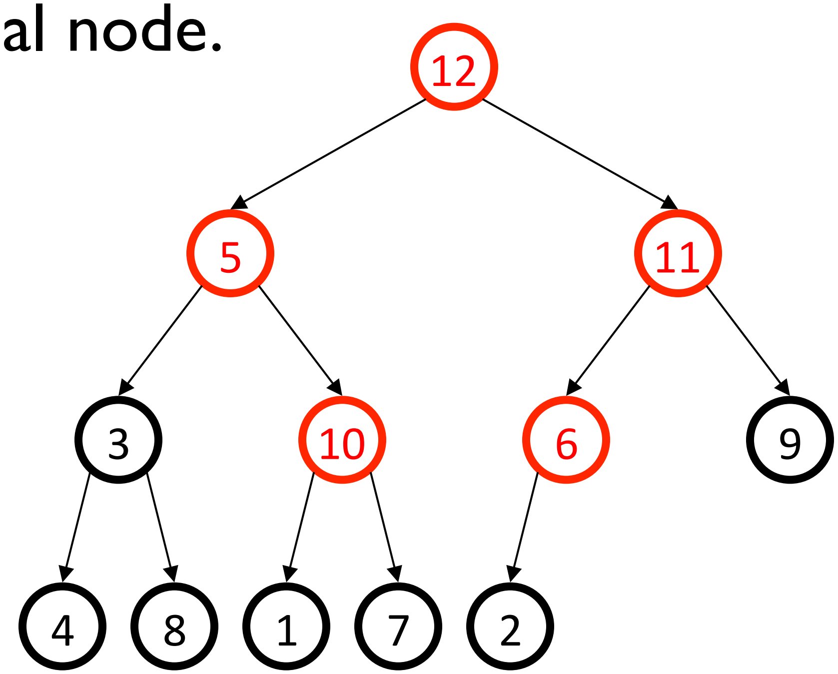
O(N) buildHeap

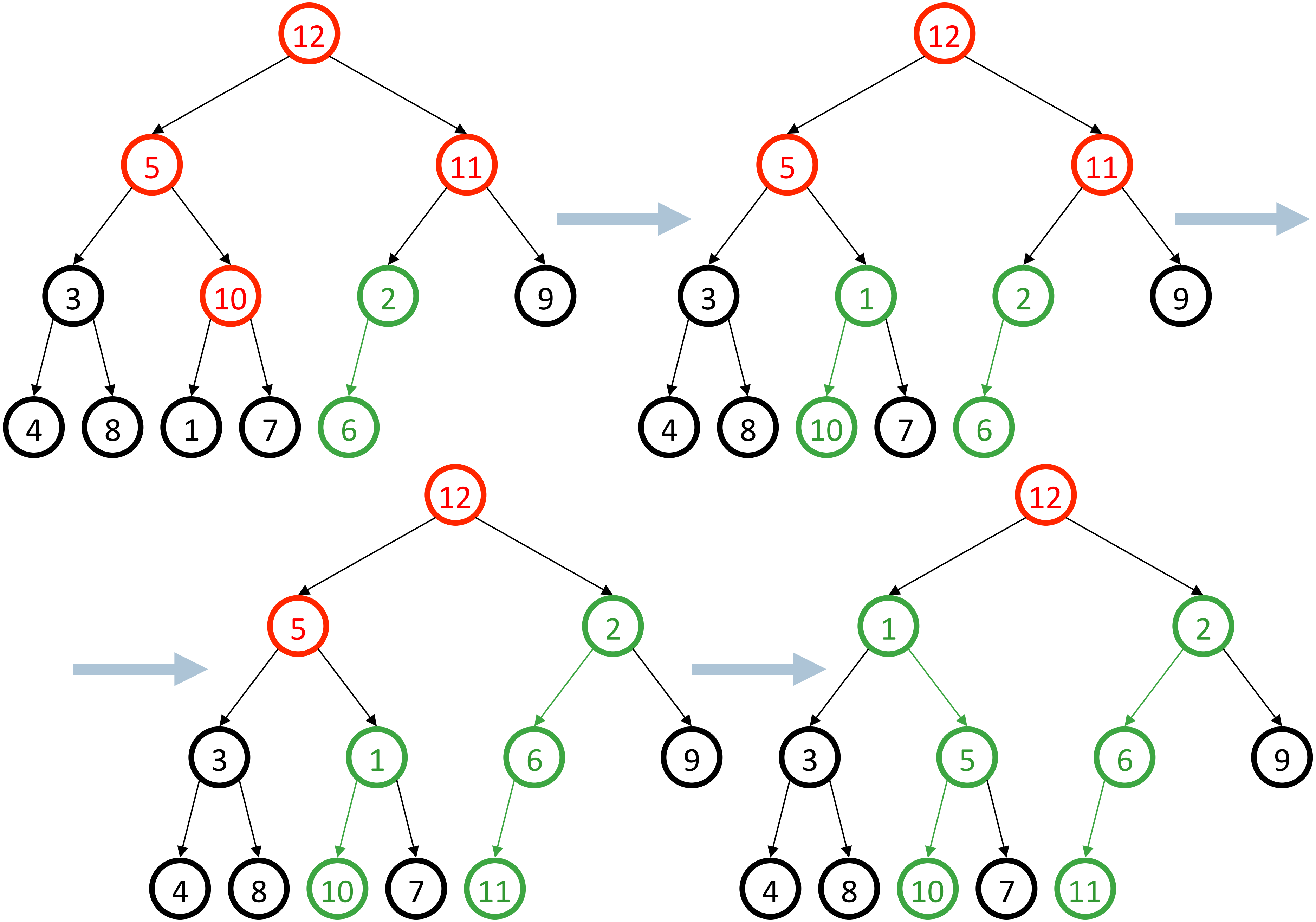
- First, add all elements arbitrarily maintaining the completeness property
- Then fix the heap order property by performing a "bubble down" operation on every node that is not a leaf, starting from the rightmost internal node and working back to the root



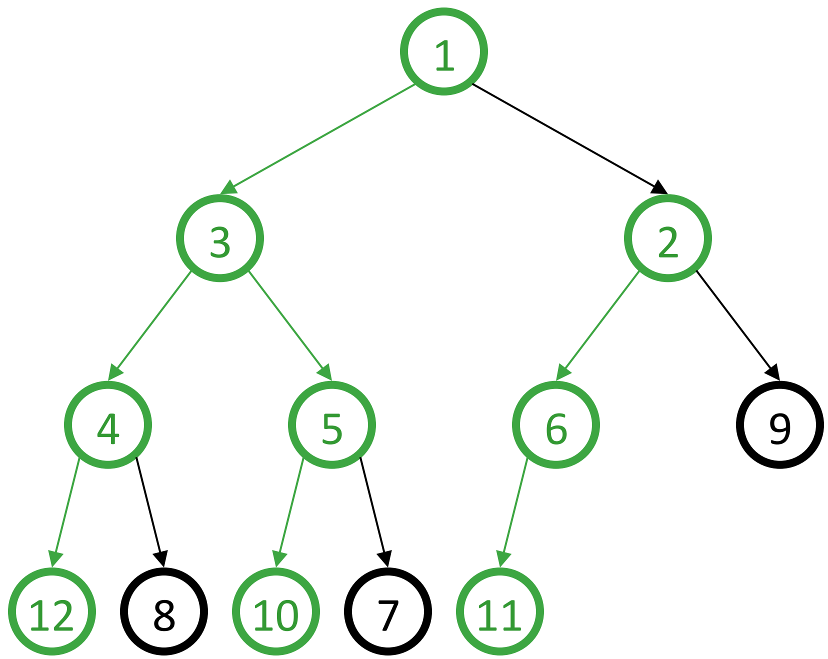
buildHeap practice problem

- Each element in the list [12, 5, 11, 3, 10, 6, 9, 4, 8, 1, 7, 2] has been inserted into a heap such that the completeness property has been maintained.
- Now, fix the heap's order property by "bubbling down" every internal node.



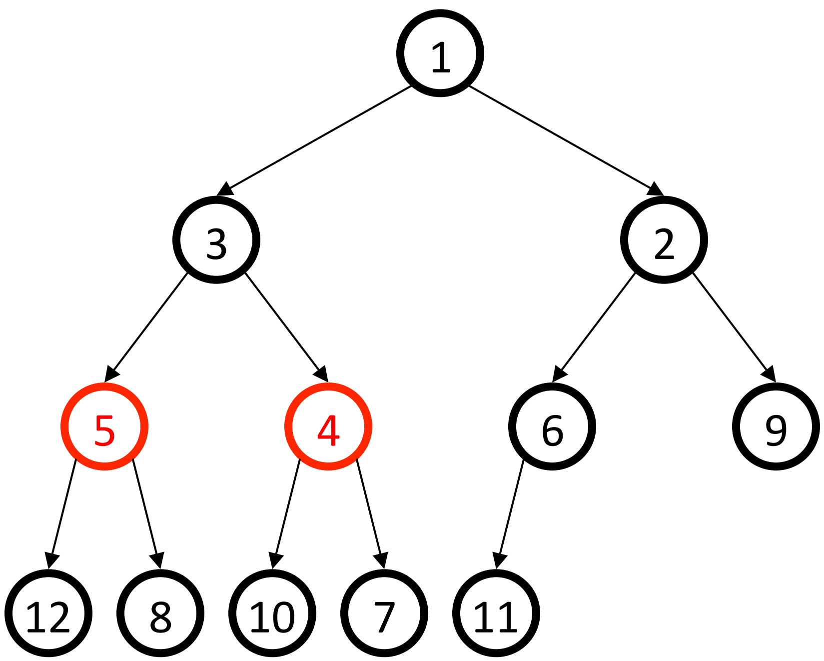

12
5 11
3 10 2 9
4 8 1 7 6
12
5 11
3 1 2 9
4 8 10 7 6
12
5 2
3 1 6 9
4 8 10 7 11
12
1 2
3 5 6 9
4 8 10 7 11

Final State of the Heap

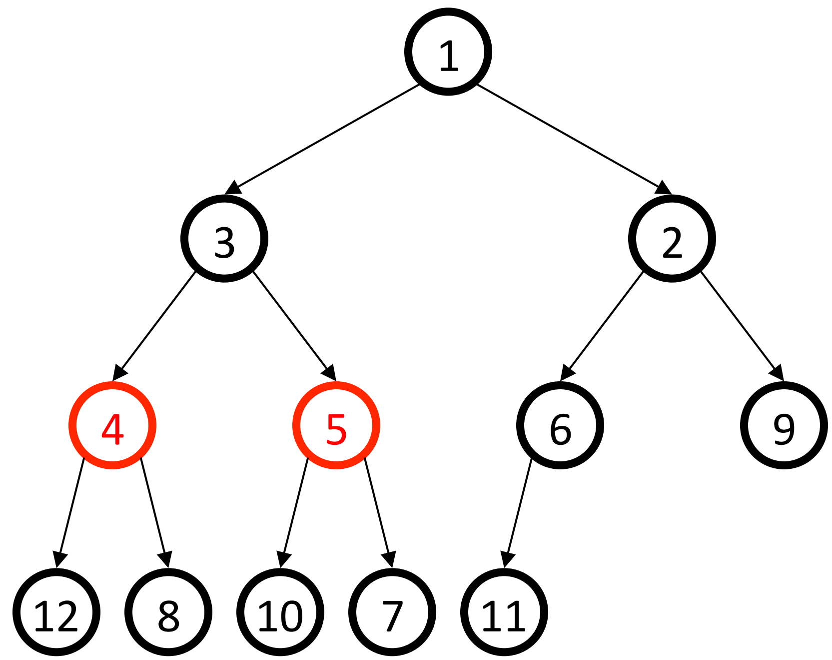


Different Heaps

Successive inserts $\Theta(n \log n)$:



buildHeap $\Theta(n)$:



But it doesn't matter because they are both heaps.

Heap Sort

Heap sort

- **heap sort**: an algorithm to sort an array of N elements by turning the array into a heap, then doing a `remove` N times
  - The elements will come out in sorted order!

- What is the runtime?

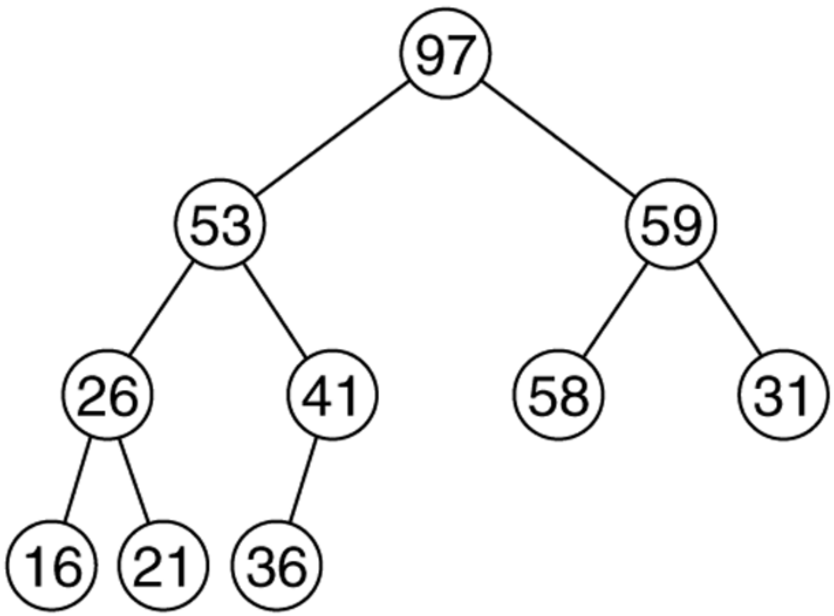
- This algorithm is not very space-efficient. Why not?

Improved heap sort

- The heap sort shown requires a second array
- We can use a max-heap to implement an improved version of heap sort that needs no extra storage
  - Useful on low-memory devices
  - Still only O($n \log n$) runtime
  - Elegant

Improved heap sort 1

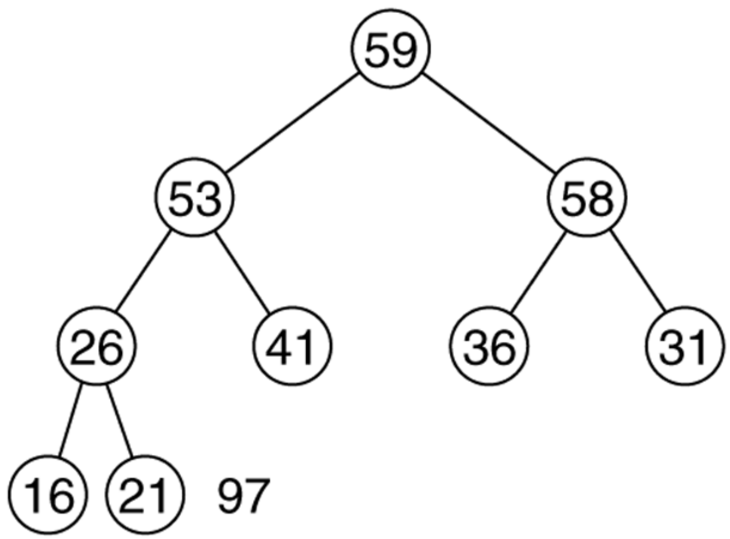
- Use an array heap, but with 0 as the root index
- max-heap state after `buildHeap` operation:



97	53	59	26	41	58	31	16	21	36				
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Improved heap sort 2

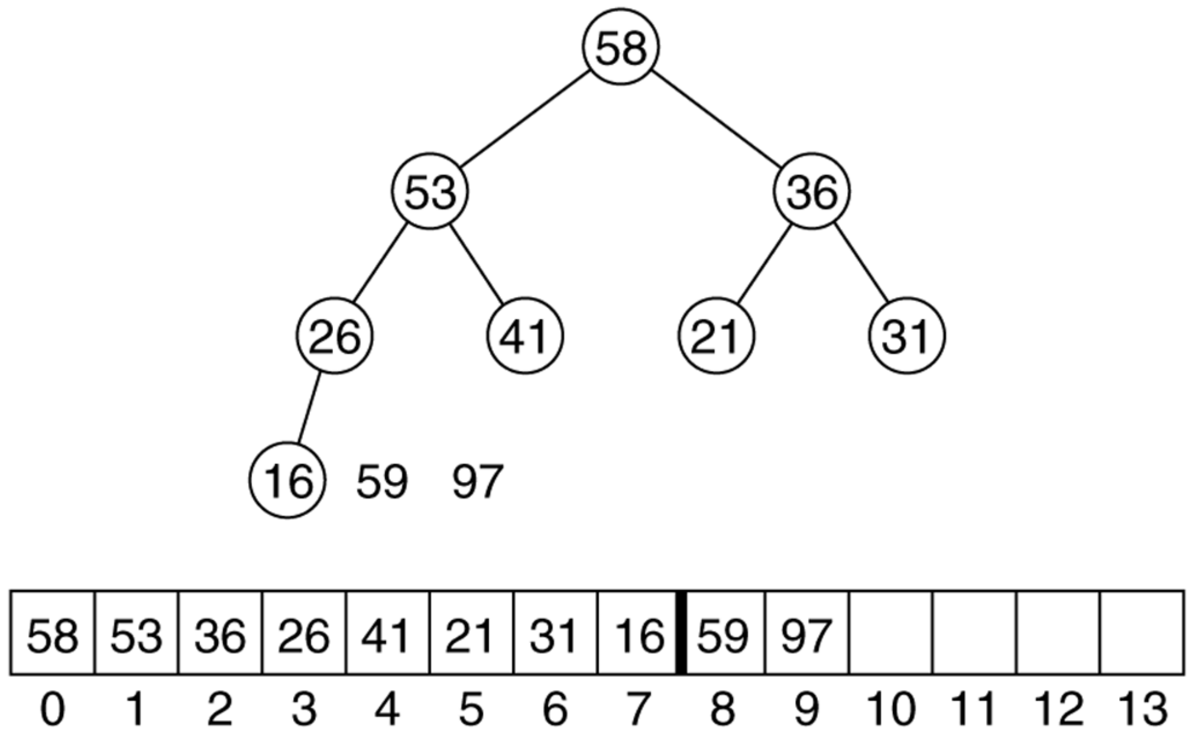
- State after one `remove` operation:
  - Modified `remove` that moves element to end



59	53	58	26	41	36	31	16	21	97				
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Improved heap sort 3

- State after two `remove` operations:
  - Notice that the largest elements are at the end (becoming sorted!)



Sorting algorithms review

	Best case	Average case (†)	Worst case
Bubble sort	n	n^2	n^2
Selection sort	n^2	n^2	n^2
Insertion sort	n	n^2	n^2
Mergesort	$n \log_2 n$	$n \log_2 n$	$n \log_2 n$
Heapsort	$n \log_2 n$	$n \log_2 n$	$n \log_2 n$
Quicksort	$n \log_2 n$	$n \log_2 n$	n^2

† According to Knuth, the **average growth rate** of Insertion sort is about 0.9 times that of Selection sort and about 0.4 times that of Bubble Sort. The **average growth rate** of Quicksort is about 0.74 times that of Mergesort and about 0.5 times that of Heapsort.