Heapsort and Quicksort (Revisited)

O(n log n) Sorting

- sorting in O(n log n) time requires a divide an conquer approach
- although each pass requires n steps as all elements are examined, only log n passes are required
- or n passes, but each pass only takes log n steps

Three Sorting Algorithms

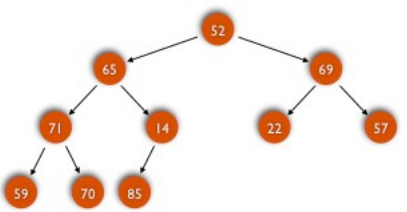
- mergesort sorts in O(n log n) time, but requires 2n space
- heapsort sorts in O(n log n) time, in place, but has a constant factor of about 2.5x.
- quicksort sorts in O(n log n) time, in place, but has bad worst case performance — $O(n^2)$

Heapsort

- array to be sorted interpreted as a binary tree
- first, rearrange elements to form a heap — O(n log n) time
- then, repeatedly swap top (largest element) with bottom, remove bottom from tree, then makeHeap — another O(n log n)
- makeHeap algorithm is same as leaving priority queue

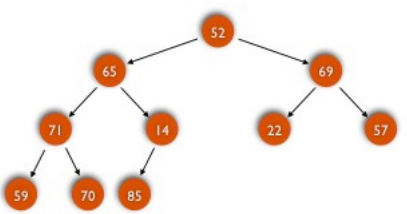
Step 1: Interpret Array as a Binary Tree

52
65
69
71
14
22
57
59
70
85



Step 1: Interpret Array as a Binary Tree

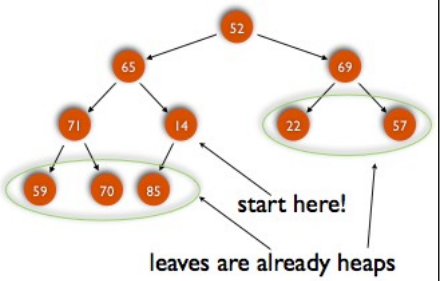
52
65
69
71
14
22
57
59
70
85



makeHeap algorithm (like leaving a priority queue) takes as input a tree that is not a heap but whose children are heaps, and turns it into a heap

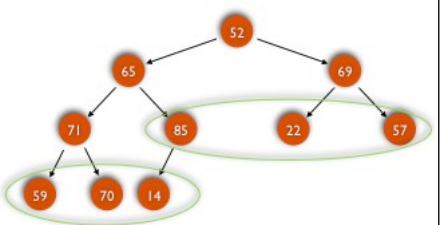
Step 1: Interpret Array as a Binary Tree

52
65
69
71
14
22
57
59
70
85



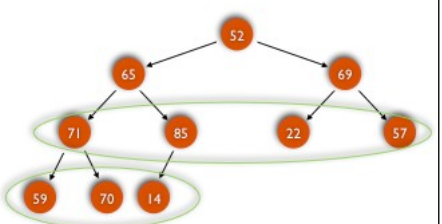
Step 2: Form a Heap

52
65
69
71
85
22
57
59
70
14



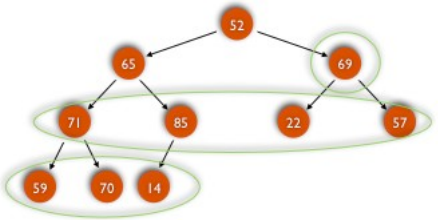
Step 2: Form a Heap

52
65
69
71
85
22
57
59
70
14



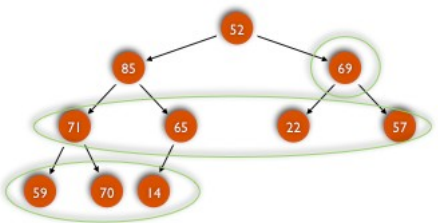
Step 2: Form a Heap

52
65
69
71
85
22
57
59
70
14



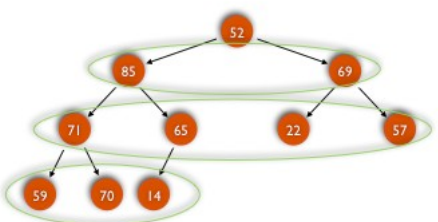
Step 2: Form a Heap

52
85
69
71
65
22
57
59
70
14



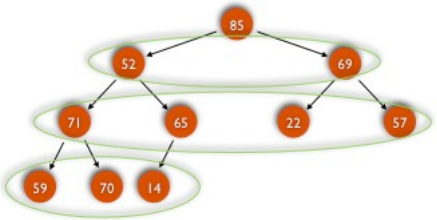
Step 2: Form a Heap

52
85
69
71
65
22
57
59
70
14



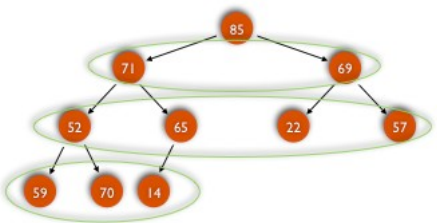
Step 2: Form a Heap

85
52
69
71
65
22
57
59
70
14



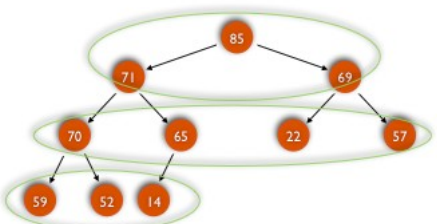
Step 2: Form a Heap

85
71
69
52
65
22
57
59
70
14



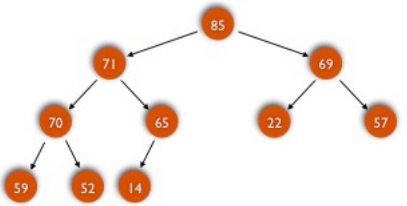
Step 2: Form a Heap

85
71
69
70
65
22
57
59
52
14



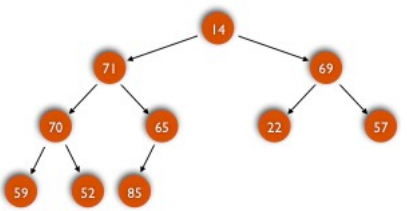
Step 3: Repeatedly Swap Top and Bottom, "Remove" Bottom, makeHeap Top

85
71
69
70
65
22
57
59
52
14



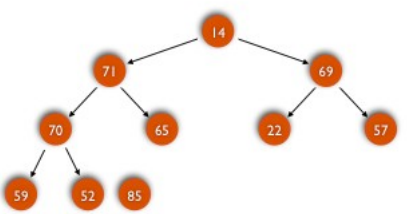
Step 3: Repeatedly Swap Top and Bottom, "Remove" Bottom, makeHeap Top

14
71
69
70
65
22
57
59
52
85



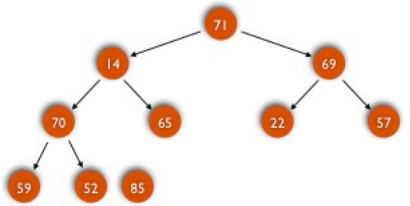
Step 3: Repeatedly Swap Top and Bottom, "Remove" Bottom, makeHeap Top

14
71
69
70
65
22
57
59
52
85



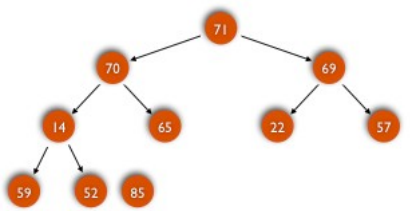
Step 3: Repeatedly Swap Top and Bottom, "Remove" Bottom, makeHeap Top

71
14
69
70
65
22
57
59
52
85



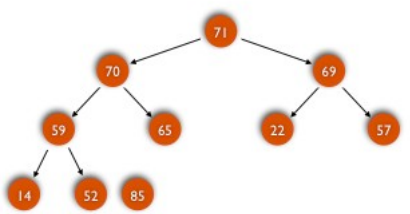
Step 3: Repeatedly Swap Top and Bottom, "Remove" Bottom, makeHeap Top

71
70
69
14
65
22
57
59
52
85



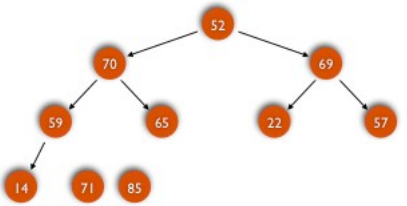
Step 3: Repeatedly Swap Top and Bottom, "Remove" Bottom, makeHeap Top

71
70
69
59
65
22
57
14
52
85



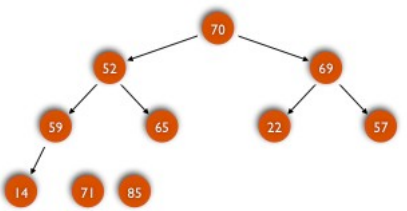
Step 3: Repeatedly Swap Top and Bottom, "Remove" Bottom, makeHeap Top

52
70
69
59
65
22
57
14
71
85



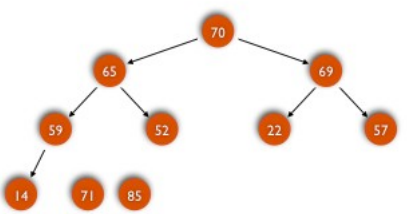
Step 3: Repeatedly Swap Top and Bottom, "Remove" Bottom, makeHeap Top

70
52
69
59
65
22
57
14
71
85



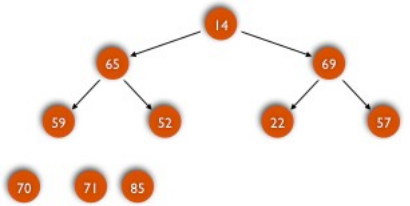
Step 3: Repeatedly Swap Top and Bottom, "Remove" Bottom, makeHeap Top

70
65
69
59
52
22
57
14
71
85



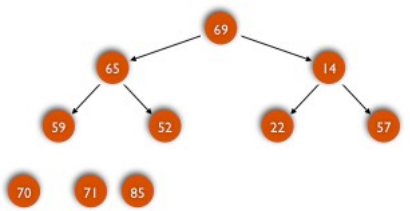
Step 3: Repeatedly Swap Top and Bottom, "Remove" Bottom, makeHeap Top

14
65
69
59
52
22
57
70
71
85



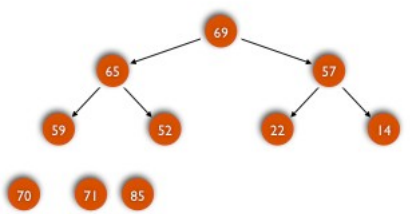
Step 3: Repeatedly Swap Top and Bottom, "Remove" Bottom, makeHeap Top

69
65
14
59
52
22
57
70
71
85



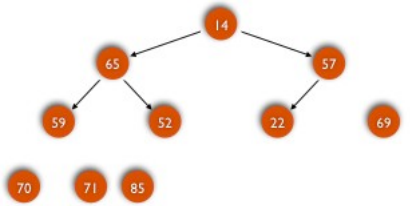
Step 3: Repeatedly Swap Top and Bottom, "Remove" Bottom, makeHeap Top

69
65
57
59
52
22
14
70
71
85



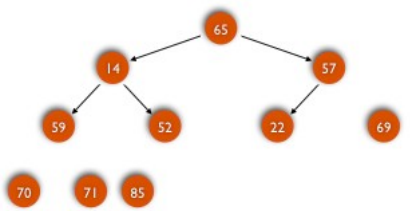
Step 3: Repeatedly Swap Top and Bottom, "Remove" Bottom, makeHeap Top

14
65
57
59
52
22
69
70
71
85



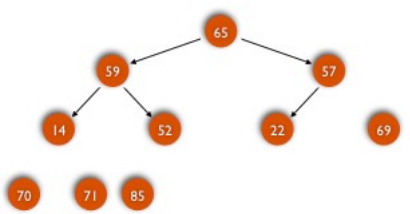
Step 3: Repeatedly Swap Top and Bottom, "Remove" Bottom, makeHeap Top

65
14
57
59
52
22
69
70
71
85

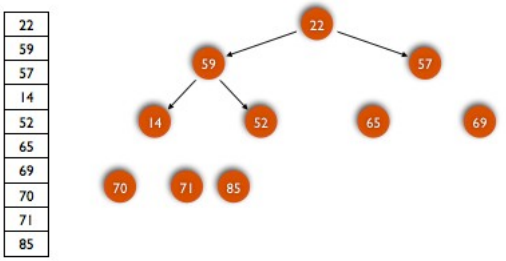


Step 3: Repeatedly Swap Top and Bottom, "Remove" Bottom, makeHeap Top

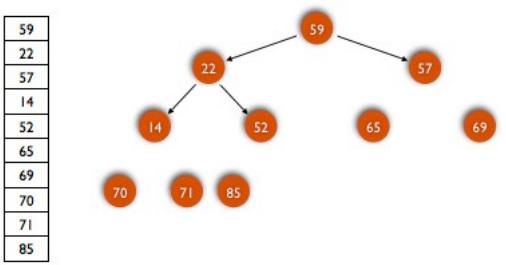
65
59
57
14
52
22
69
70
71
85



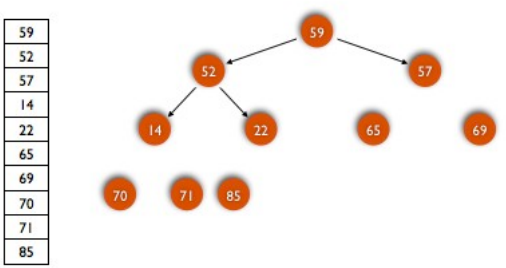
Step 3: Repeatedly Swap Top and Bottom, "Remove" Bottom, makeHeap Top



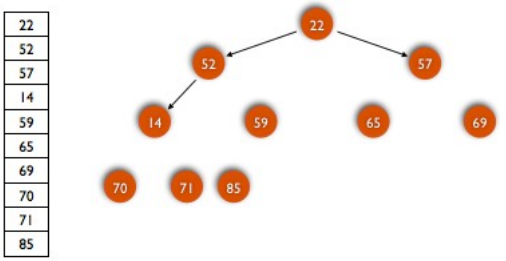
Step 3: Repeatedly Swap Top and Bottom, "Remove" Bottom, makeHeap Top



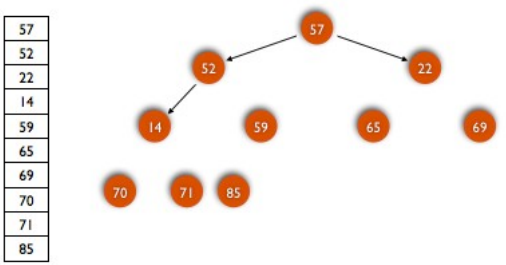
Step 3: Repeatedly Swap Top and Bottom, "Remove" Bottom, makeHeap Top



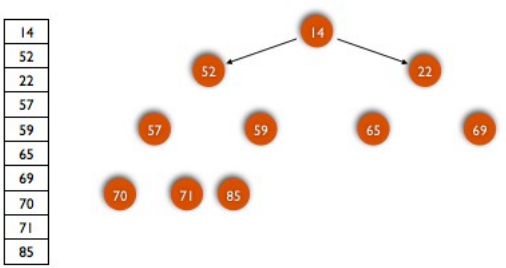
Step 3: Repeatedly Swap Top and Bottom, "Remove" Bottom, makeHeap Top



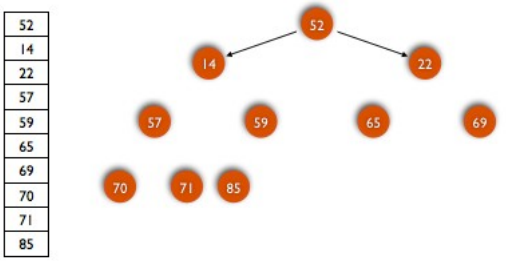
Step 3: Repeatedly Swap Top and Bottom, "Remove" Bottom, makeHeap Top



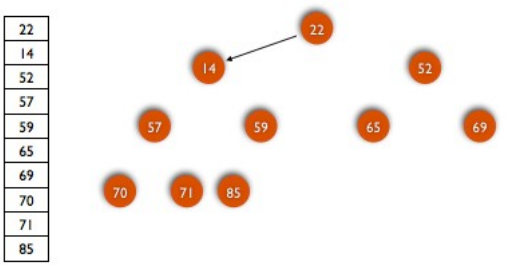
Step 3: Repeatedly Swap Top and Bottom, "Remove" Bottom, makeHeap Top



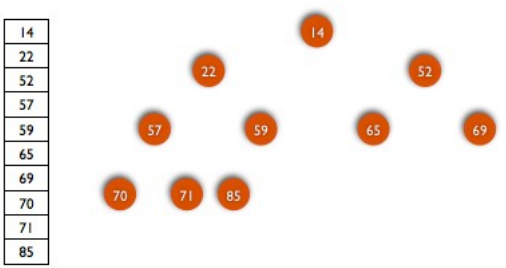
Step 3: Repeatedly Swap Top and Bottom, "Remove" Bottom, makeHeap Top



Step 3: Repeatedly Swap Top and Bottom, "Remove" Bottom, makeHeap Top



Step 3: Repeatedly Swap Top and Bottom, "Remove" Bottom, makeHeap Top



```
public class Array {
.
.
.
    private static int left (int i) {
        return 2 * i + 1;
    }

    private static int right (int i) {
        return 2 * i + 2;
    }

    private static int lastInternal (int out) {
        return (out / 2) - 1;
    }

    private static boolean isLeaf (int i, int out) {
        return i > lastInternal(out);
    }
.
.
.
}
```

```
public class Array {
.
.
    private void makeHeap (int top, int out) {
        makeHeap((Comparable) this.values[top], top, out);
    }

    private void makeHeap (Comparable v, int top, int out) {
        if (Array.isLeaf(top, out) {
            this.values[top] = v; return;
        }
        int larger = Array.right(top) < out &&
     ((Comparable) this.values[Array.right(top)]).compareTo(
                   this.values[Array.left(top)]) > 0 ?
            Array.right(top) :
            Array.left(top);
        if (v.compareTo(this.values[larger]) > 0) {
            this.values[top] = v; return;
        }
        this.values[top] = this.values[larger];
        makeHeap(v, larger, out);
    }
.
.
}
```

```
public class Array {
.
.
.
    public void heapsort () {
      int i;

      for (i = Array.lastInternal(this.length()); i >= 0; i--)
          this.makeHeap(i, this.length());

      for (i = this.length() - 1; i > 0; i--) {
          this.swap(0,i);
          this.makeHeap(0,i);
      }
    }
.
.
.
}
```

Heapsort's 2.5x Constant

- 0.5n log n to make initial heap — leaves are already heaps
- 2 log n for each of n passes — must compare node to each child
- but — heapsort's worst case is also O(n log n)

Quicksort's Worst Case

- in quicksort, each pass divides the array into two parts — less than pivot and greater or equal to pivot
- what if pivot is at the extreme? then two parts are size 1 and (n - 1)
- thus, worst case quicksort is $O(n^2)$
- if pivotIndex just returns top, then quicksorting a sorted array results in worst case
- improve quicksort by replacing pivotIndex with smarter selection criterion