Rushil Chintojan

Heap Sort

The heapSort is similar to the selectionSort, in that we look for the maximum value and swap it to the end. However, a maxHeap finds the maximum faster than the selectionSort, because the maximum is always at the root, for a performance of O(1). Now the heap needs to be re-formed, which we can do by repeated swapping down. (So the heapSort is also similar to the bubble sort, but works on a tree.) The reheap expense is O(log n). Since we must do this for each item, the heap sort is O(n log n).

Previous O(n log n) sorts were the MergeSort and the QuickSort. Let's compare all three sorts.

The **MergeSort** gives stable performances. It usually requires temporary storage space.

The **QuickSort**'s performance is "unstable" because it can degrade to $O(n^2)$ when you choose a bad pivot. Despite that problem, the QuickSort is actually faster than the other two sorts when sorting random arrays. The QuickSort was invented in 1962.

The **HeapSort** is stable and does not use any temporary storage space. It is not good for small n, because of the "overhead" of extra time it needs to rearrange random elements into the heap order. The HeapSort was invented in 1964.

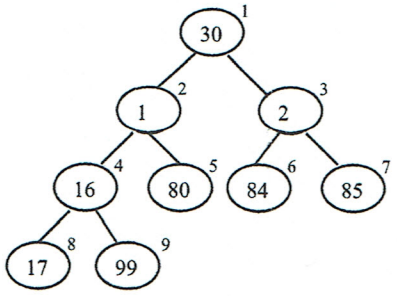
This table summarizes the characteristics of all our O(n log n) sorts:

	best	average	worst	Extra Space	Stable	Overhead
MergeSort	O(n log n)	O(n log n)	O(n log n)	yes	yes	no
QuickSort	O(n log n)	O(n log n)	$O(n^2)$	no	no	no
HeapSort	O(n log n)	O(n log n)	O(n log n)	no	yes	yes

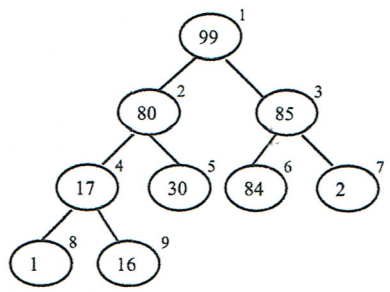
Coding the HeapSort

If you begin with random numbers, then the heap sorting takes 2 distinct phases. First, make the array into a maxHeap (this phase is the "overhead"). Second, sort the heap.

First phase. Transform the random array into a maxHeap. The basic idea is to visit each subtree and form each subtree into a heap. The most efficient algorithm starts at the middle of the array, does heapDown on that subtree, moves to the left in the array, does heapDown, and so on. How clever! Using the algorithm, show how the data moves as it turns the random array (on the left) into a maxheap (on the right):



	30	1	2	16	80	84	85	17	99
0	1	2	3	4	5	6	7	8	9



	99	80	85	17	30	84	2	1	16
0	1	2	3	4	5	6	7	8	9

Assume you have a working `heapDown` method. Write the code for `makeHeap`:

```
def makeHeap(array, size):
    for h in range(size // 2, 0, -1)
        heapDown(array, h, size)
```

Phase 2. Now the elements are in a heap! Let's sort it. You know that the greatest element is at the root. Swap the root and the last element. Then walk **down** the heap from the top, swapping with the larger child, until it is either in place or at the end. (You may do this either iteratively or recursively.) Reduce the last index by one, swap, and reheap down. After N-1 iterations, the array is ordered.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	
	99	80	85	17	30	84	2	16	1	start
	1	80	85	17	30	84	2	16	99	swap first and last
	85	80	1	17	30	84	2	16	99	heapDown
	85	80	84	17	30	1	2	16	99	heapDown continued
	16	80	84	17	30	1	2	85		swap first and last
	84	80	16	17	30	1	2			heapDown
	2	80	16	17	30	1	84			swap first and last
	80	2	16	17	30	1				heapDown
	80	30	16	17	2	1				heapDown continued
	1	30	16	17	2	80				swap first and last
	30	1	16	17	2					heapDown
	30	17	16	1	2					heapDown continued
	2	17	16	1	30					swap first and last
	17	2	16	1						heapDown
	1	2	16	17						swap first and last
	16	2	1							heapDown
	1	2	16							swap first and last
	2	1								heapDown
	1	2								swap first and last -- Done

Ordering this array took 18 steps. By the O(n*log n) formula, $9 * \log_2 9 = 28.53$.

Lab Assignment

Part 1: Given a heap (the 9 numbers shown above), display it, heap sort it, and display it again. Use these headers:

```
def display(array)
def sort(array)
def swap(array, a, b)
def heapDown(array, k, size)
def isSorted(array)      # returns True/False
```

Part 2: Generate 100 random numbers between 1 and 100, formatted to 2 decimal places, make a heap, sort it, and display it. You will need to write the methods:

```
def createRandom(array)
def makeHeap(array, size)
```

File name: **LastName_FirstInitial_U1_L2.py**

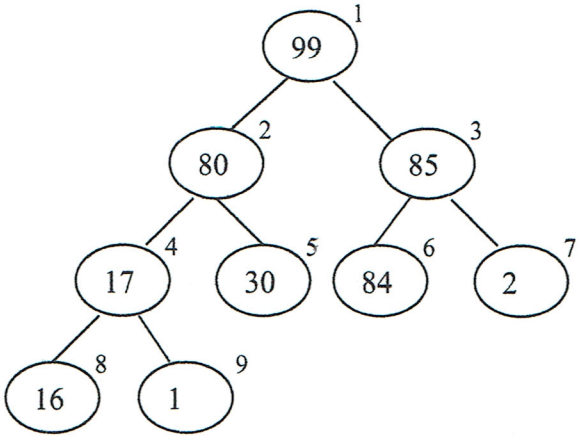
Kushal Chowdhary P2L

Heaps

A heap, or a maxheap, is a binary tree with two additional properties: 1) each node is larger than either of its children and 2) it is complete.

(A minheap is a heap in which each node is smaller than its children.)

Strangely enough, we do not code heaps using TreeNode. Rather, we represent the heap as an array! We do not use the zero cell. This way, any arbitrary node k has children in locations 2k and 2k+1, and a parent in k/2. Make sure these formulas work.



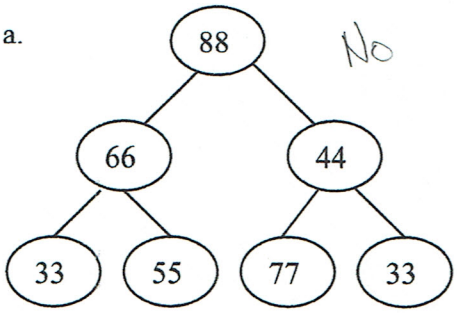
	99	80	85	17	30	84	2	16	1			
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]

Why must the heap be a *complete* binary tree? So that the indices are unique

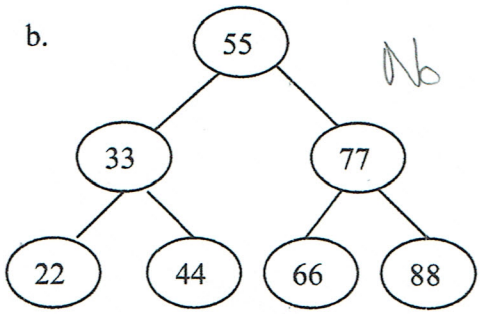
Exercises

1. Which of the following trees is a heap?

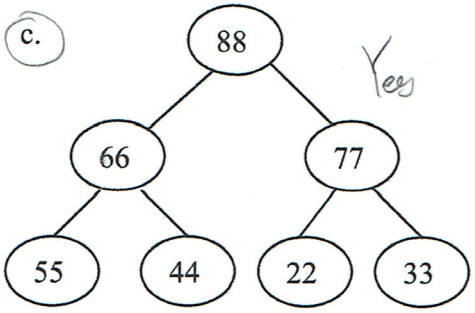
a. No



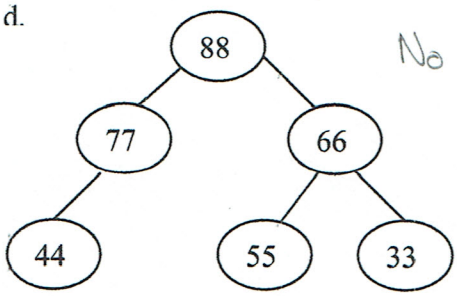
b. No



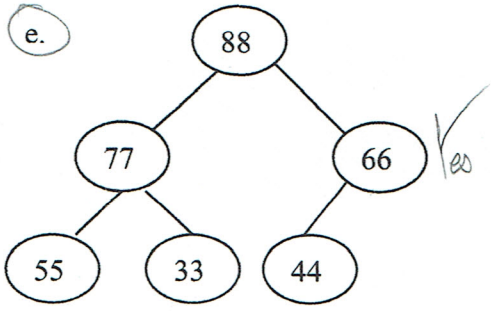
c. Yes



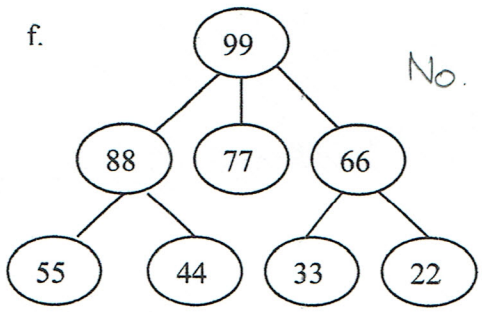
d. No



e. Yes



f. No



2. Which of the following arrays have the heap property?

a.

	99	88	66	44	33	55	77	33
0	1	2	3	4	5	6	7	8

No

b.

	99	88	77	66	55	44	33	22
0	1	2	3	4	5	6	7	8

Yes

c.

	99	88	44	77	22	33	55	66
0	1	2	3	4	5	6	7	8

No

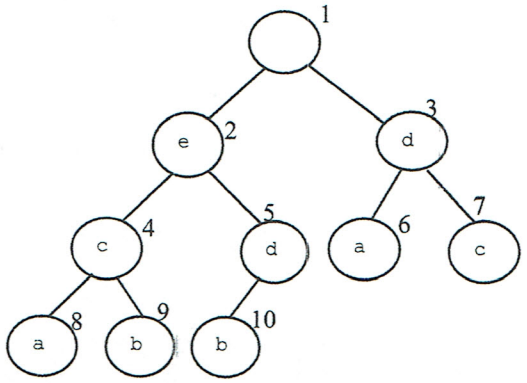
d.

	99	88	66	77	22	33	44	55
0	1	2	3	4	5	6	7	8

Yes

heapDown

`heapUp` and `heapDown` are the two basic heap operations. Whenever you add to or remove from a heap, the new item has to rise or fall to its proper place so that the heap is kept in heap order. Each method works in O(log n).



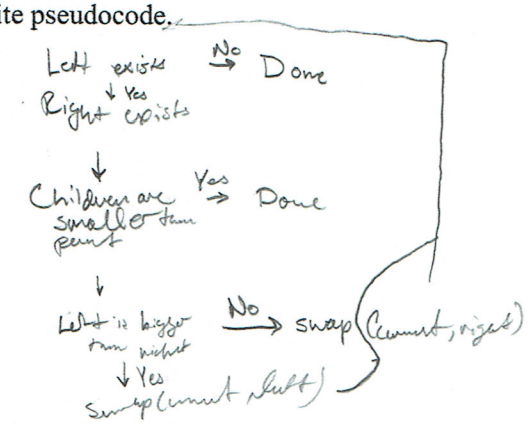
Let's do `heapDown`. For variety, we will use `char` data. Let's just put in a new value at the root, for example, put in `'b'`. `heapDown` swaps, if necessary, that value with its largest child, and keeps swapping, until the value is in place, i.e, the heap order has been restored. Trace the movement if you put a new `'b'` at the root:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	
null	'b'	e	d	c	d	a	c	a	b	b	insert 'b'
null	e	b	d	c	d	a	c	a	b	b	swap with largest child
null	e	d	d	c	b	a	c	a	b	b	swap with largest child

Go back to the original heap. Insert an `'f'` at the root. What happens? There is no swapping, because `'f'` is larger than its children; it has found its proper place, which is one base case. Going off the end of the array is another base case.

The `heapDown` algorithm is easy to say, but several things are going on: swap the current value with its largest child, then recur. If it helps, draw a flowchart. Then write pseudocode.

While node is not bigger than children:
    swapNode = max(left, right)
    swap (swapNode, node)



The header below is for the recursive version. (`heapDown` can also be done iteratively.) Assume that you have a working `swap` method. `size` in this case is just the length of the array (later, in heapSort, `size` will be reduced by 1 during each recursive call).

```
def heapDown(array, k, size):
```