Heap Sort

Sorting with Heaps

- Heaps can be used to sort data
  - Observation 1: Removal of a node from a heap can be performed in $O(\log n)$ time
  - Observation 2: Nodes are removed in order
  - Conclusion: Removing all of the nodes one by one would result in sorted output
  - Analysis: Removal of *all* the nodes from a heap is a $O(n*\log n)$ operation

But …

- A heap can be used to return sorted data
  - In $O(n*\log n)$ time
- However, we can't assume that the data to be sorted just happens to be in a heap!
  - **Aha!** But we can *put* it in a heap.
  - Inserting an item into a heap is a $O(\log n)$ operation so inserting n items is $O(n*\log n)$
- But we can do better than just repeatedly calling the insertion algorithm

Heapifying Data

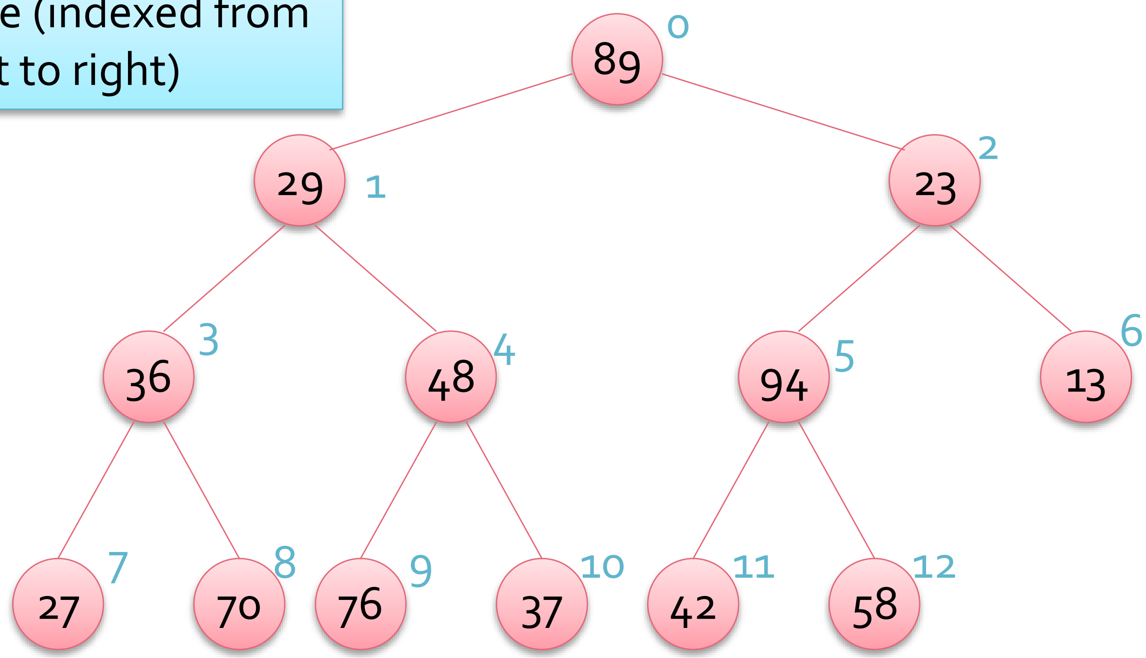
- To create a heap from an unordered array repeatedly call *bubbleDown*
  - Any subtree in a heap is itself a heap
  - Call *bubbleDown* on elements in the upper ½ of the array
  - Start with index $(n-2)/2$ and work up to index 0
    - i.e. from the last non-leaf node to the root

parent index = $(i-1) / 2$

- *bubbleDown* does not need to be called on the lower half of the array (the leaves)
  - Since *bubbleDown* restores the partial ordering from any given node down to the leaves

Heapify Example

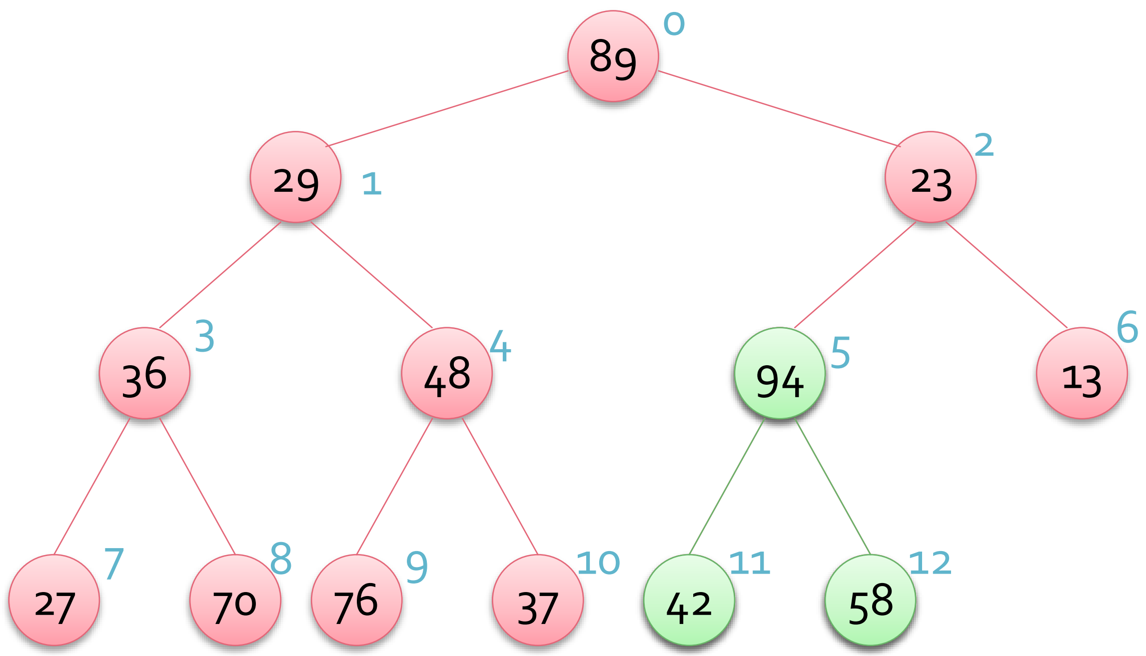
Assume unsorted input is contained in an array as shown here (indexed from top to bottom and left to right)



Heapify Example

n = 13, (n-2) / 2 = 5

bubbleDown(5)

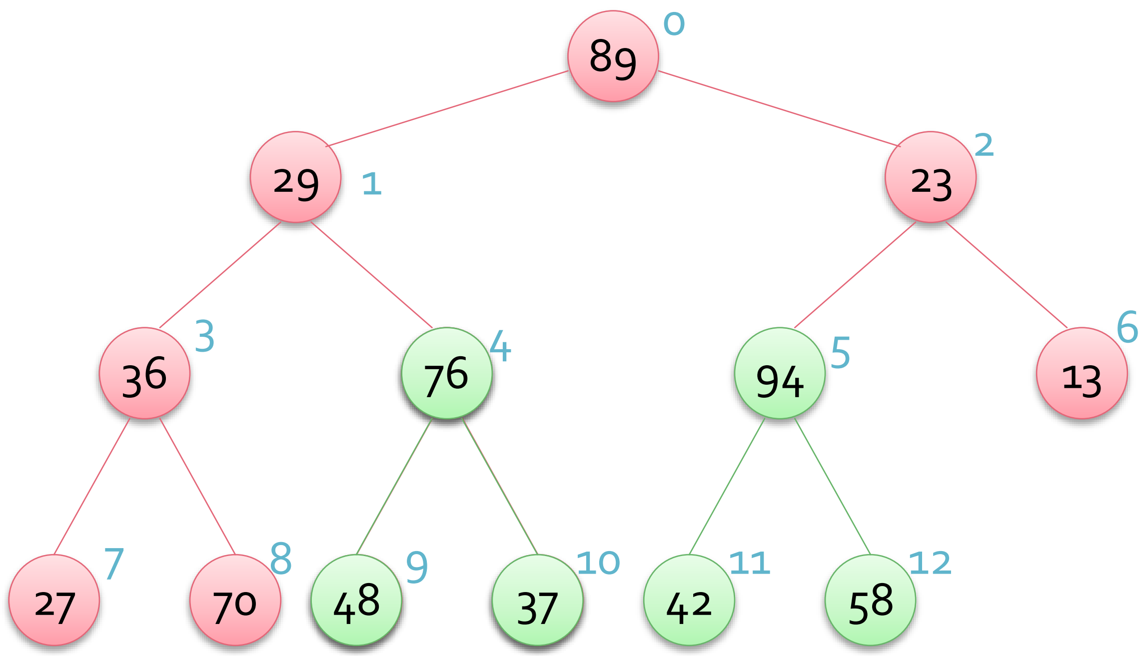


Heapify Example

n = 13, (n-2) / 2 = 5

bubbleDown(5)

bubbleDown(4)



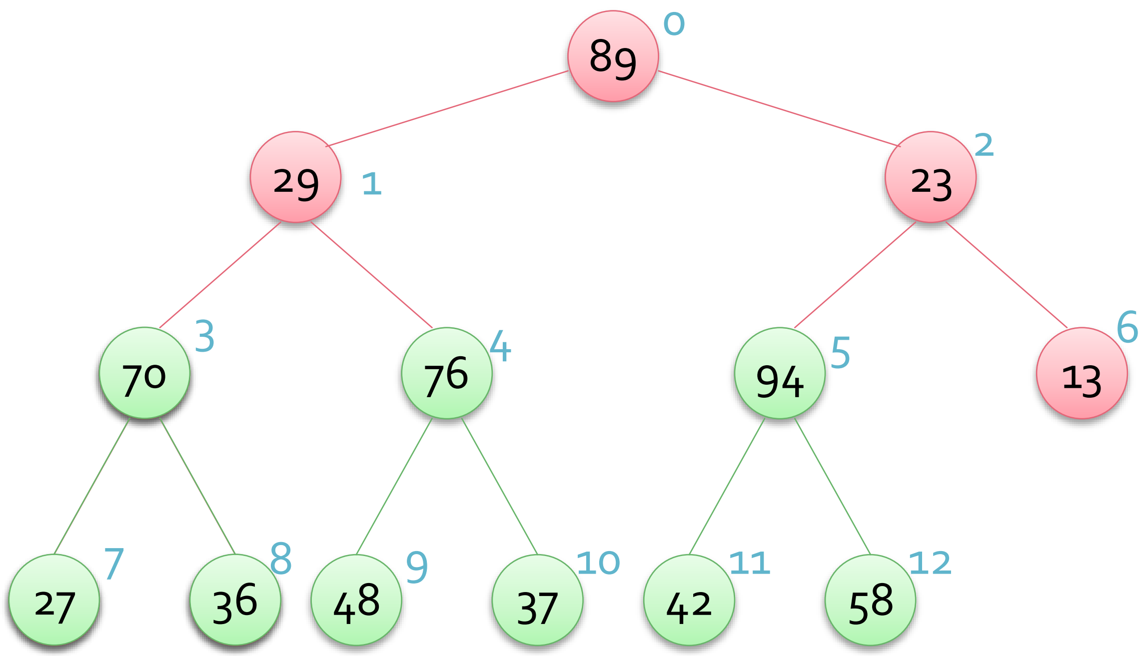
Heapify Example

n = 13, (n-2) / 2 = 5

bubbleDown(5)

bubbleDown(4)

bubbleDown(3)



Heapify Example

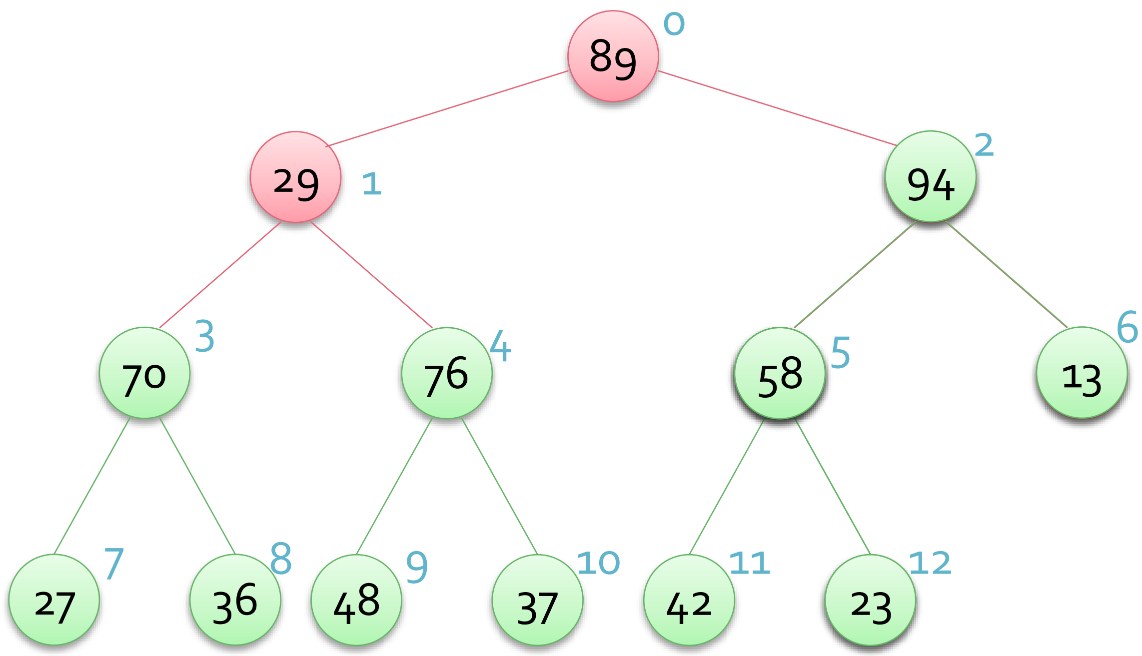
n = 13, (n-2) / 2 = 5

bubbleDown(5)

bubbleDown(4)

bubbleDown(3)

bubbleDown(2)



Heapify Example

n = 13, (n-2) / 2 = 5

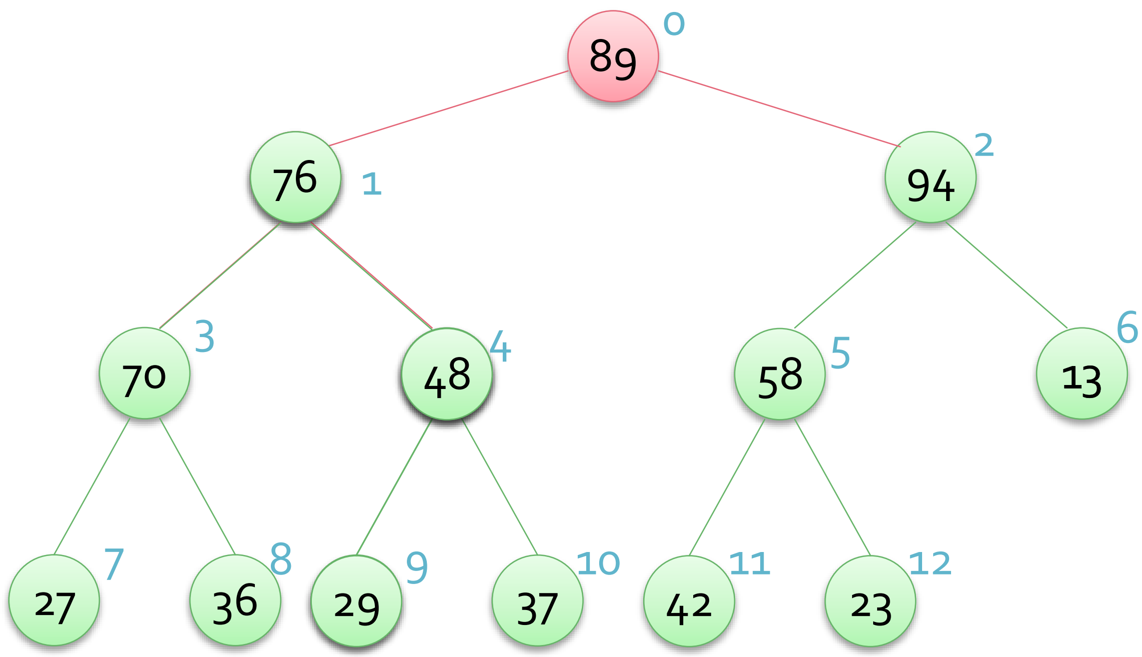
bubbleDown(5)

bubbleDown(4)

bubbleDown(3)

bubbleDown(2)

bubbleDown(1)



Heapify Example

n = 13, (n-2) / 2 = 5

bubbleDown(5)

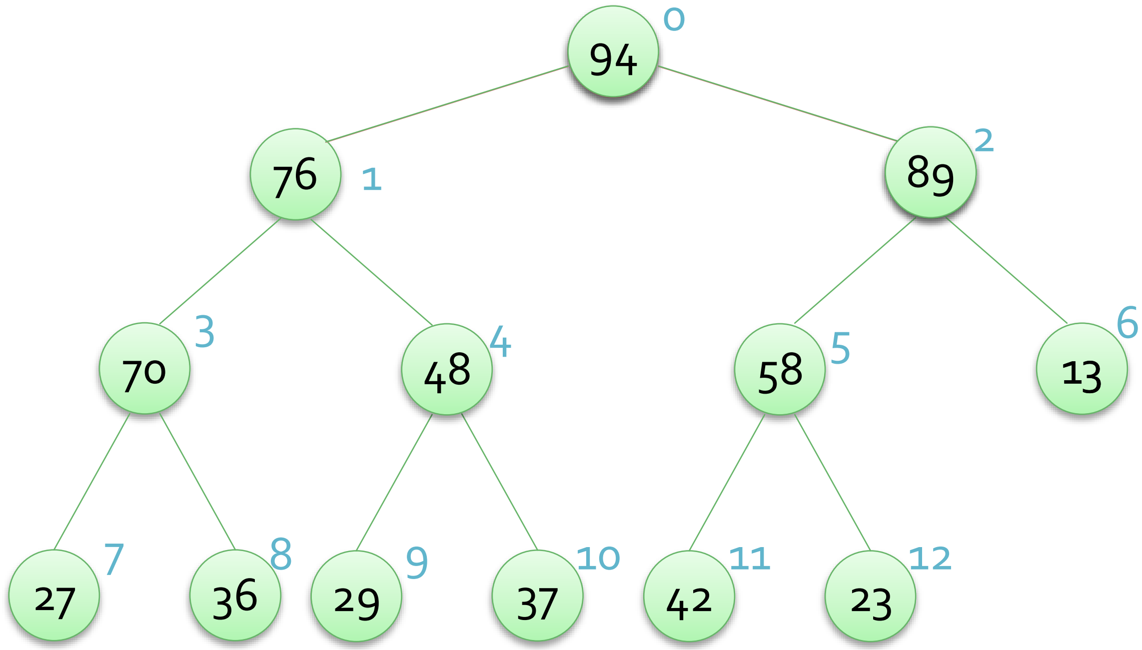
bubbleDown(4)

bubbleDown(3)

bubbleDown(2)

bubbleDown(1)

bubbleDown(0)



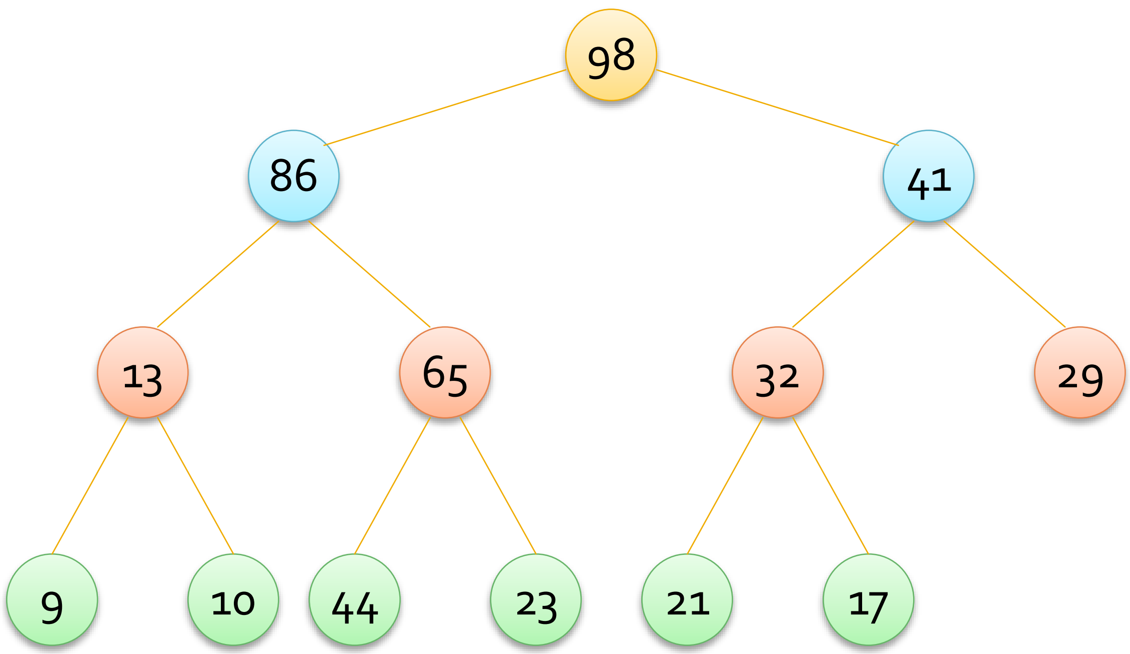
Cost to Heapify an Array

- *bubbleDown* is called on half the array
  - The cost for *bubbleDown* is *O*(*height*)
  - It would appear that heapify cost is *O*(*n**log*n*)
- In fact, the cost is *O*(*n*)
- The analysis is complex but

Beyond the scope of CMPT 225

  - *bubbleDown* is only called on ½*n* nodes
  - And mostly on sub-trees
    - And most of these are near the bottom of the tree and small

Heap Sort – Array After Heapify



colours represent tree levels

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	98	86	41	13	65	32	29	9	10	44	23	21	17

Heap Sort – In Place

- Step 1 – Heapify the array as a max heap
- Step 2 – Repeatedly remove the root
  - But what do we do with the removed root?
  - We could insert it in another array
    - But that requires additional main memory space
  - Note that we have a free array element at the end
    - So, insert it there – actually swap it

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	98	86	41	13	65	32	29	9	10	44	23	21	17

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	86	65	41	13	44	32	29	9	10	17	23	21	

98

Heap Sort – In Place

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	98	86	41	13	65	32	29	9	10	44	23	21	17

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	86	65	41	13	44	32	29	9	10	17	23	21	98

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	65	44	41	13	23	32	29	9	10	17	21	86	98

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	44	23	41	13	21	32	29	9	10	17	65	86	98

...

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	9	10	13	17	21	23	29	32	41	44	65	86	98

HeapSort Notes

- The algorithm runs in $O(n*\log n)$ time
  - Considerably more efficient than selection sort and insertion sort
  - The same (O) efficiency as MergeSort and QuickSort
- The sort can be carried out *in-place*
  - That is, it does not require that a copy of the array to be made
  - The original array is divided into a heap part and a sorted part