Priority Queues and Heaps

CMPT 225

Objectives

- Define the ADT priority queue
- Define the partially ordered property
- Define a heap
- Implement a heap using an array
- Implement the heap sort algorithm

Priority Queue ADT

ADT Priority Queue

- Items in a priority queue have a *priority*
  - Not necessarily numerical
  - Could be lowest first or highest first
- The highest priority item is removed first
- Priority queue operations
  - Insert
  - Remove in priority queue order
  - Both operations should be performed in at most $O(\log n)$ time

Implementing a Priority Queue

- Items have to be removed in priority order
  - This can only be done efficiently if the items are ordered in some way
- One option would be to use a balanced binary search tree
  - Binary search trees are fully ordered and insertion and removal can be implemented in $O(\log n)$ time
    - Some operations (e.g. removal) are complex
    - Although operations are $O(\log n)$ they require quite a lot of structural overhead
- There is a much simpler binary tree solution

Heap ADT

A complete, partially ordered, binary tree

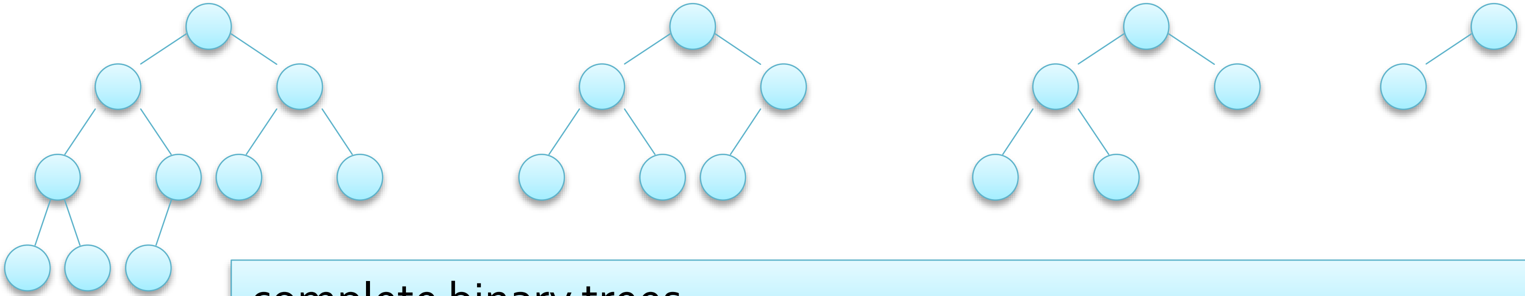
Tree Summary

- A *tree* is a connected graph made up of nodes and edges
    - With exactly one less edge than the number of nodes
- A tree has a root
    - The first node in the tree
- A tree has leaves
    - Nodes that have no children
- A binary tree is a tree with at most two children per node

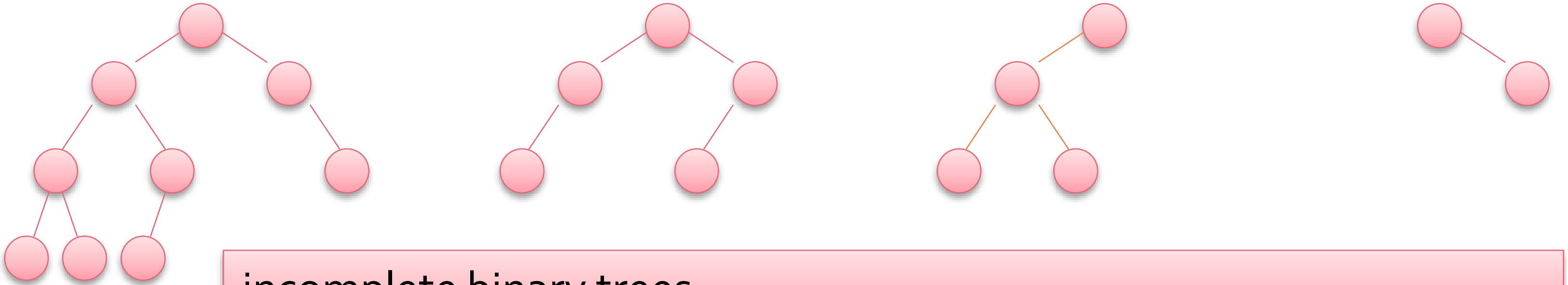
Heaps

- A *heap* is binary tree with two properties
- Heaps are *complete*
    - All levels, except the bottom, must be completely filled in
    - The leaves on the bottom level are as far to the left as possible
- Heaps are *partially ordered*
    - For a *max* heap – the value of a node is at least as large as its children's values
    - For a *min* heap – the value of a node is no greater than its children's values

Complete Binary Trees

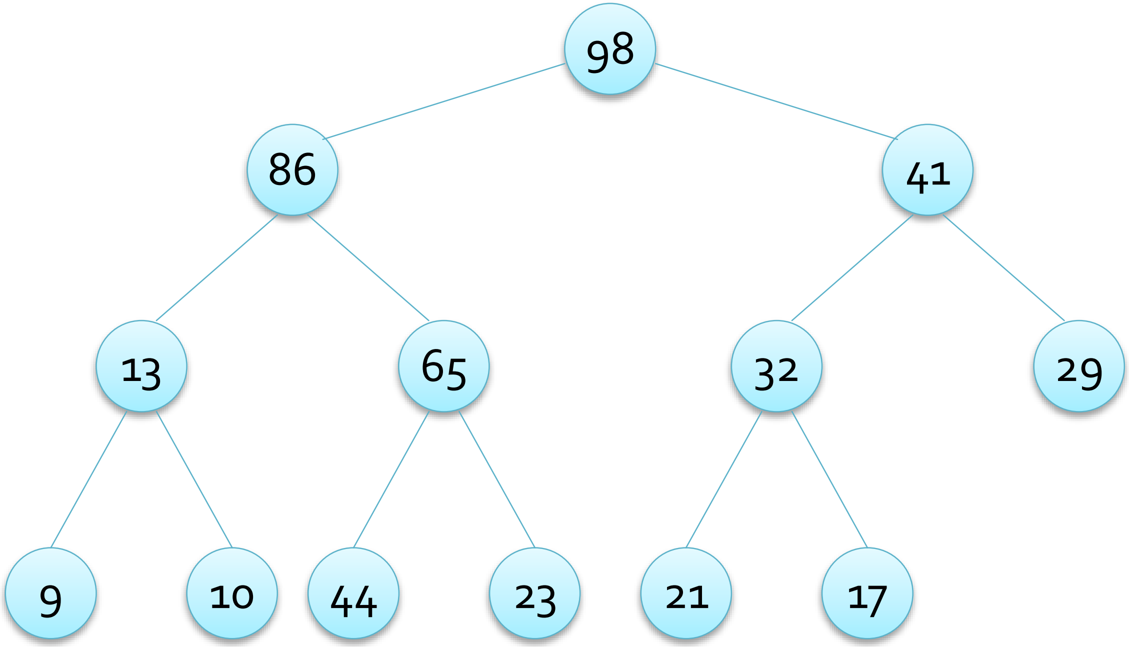


complete binary trees



incomplete binary trees

Partially Ordered Tree – max heap



Heaps are *not* fully ordered, an in order traversal would result in

9, 13, 10, 86, 44, 65, 23, 98, 21, 32, 17, 41, 29

Priority Queues and Heaps

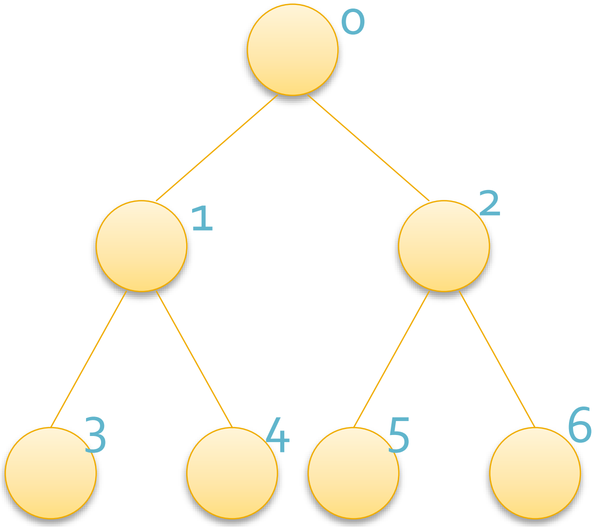
- A heap can be used to implement a priority queue
- The item at the top of the heap must always be the highest priority value
  - Because of the partial ordering property
- Implement priority queue operations:
  - Insertions – insert an item into a heap
  - Removal – remove and return the heap's root
  - For both operations preserve the heap property

Priority Queue Implementation

Using an Array to Implement a Heap

Heap Implementation

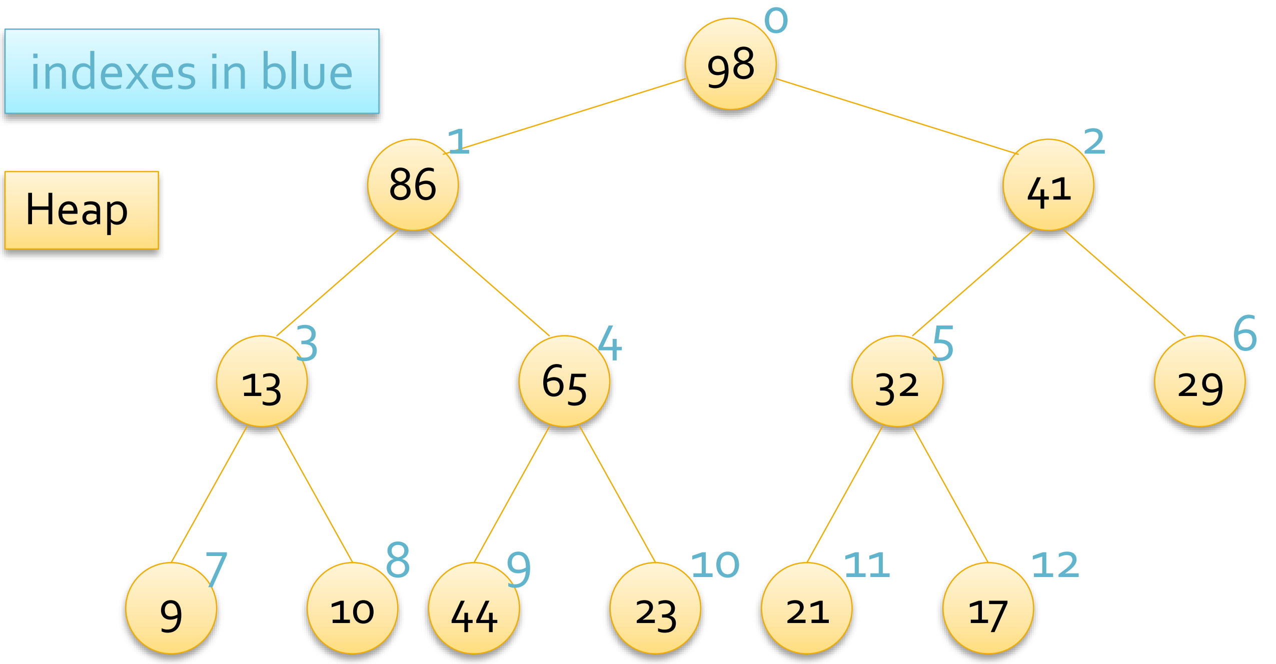
- Heaps can be implemented using *arrays*
- There is a natural method of indexing tree nodes
  - Index nodes from top to bottom and left to right
  - Because heaps are *complete* binary trees there can be no gaps in the array



Referencing Nodes

- It will be necessary to find the index of the parents of a node
  - Or the children of a node
- The array is indexed from 0 to $n - 1$
  - Each level's nodes are indexed from:
    - $2^{\text{level}} - 1$ to $2^{\text{level}+1} - 2$ (where the root is level 0)
  - The children of a node i, are the array elements indexed at $2i + 1$ and $2i + 2$
  - The parent of a node i, is the array element indexed at $(i - 1) / 2$

Heap Array Example



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

The heap is represented by an array

Heap Insertion

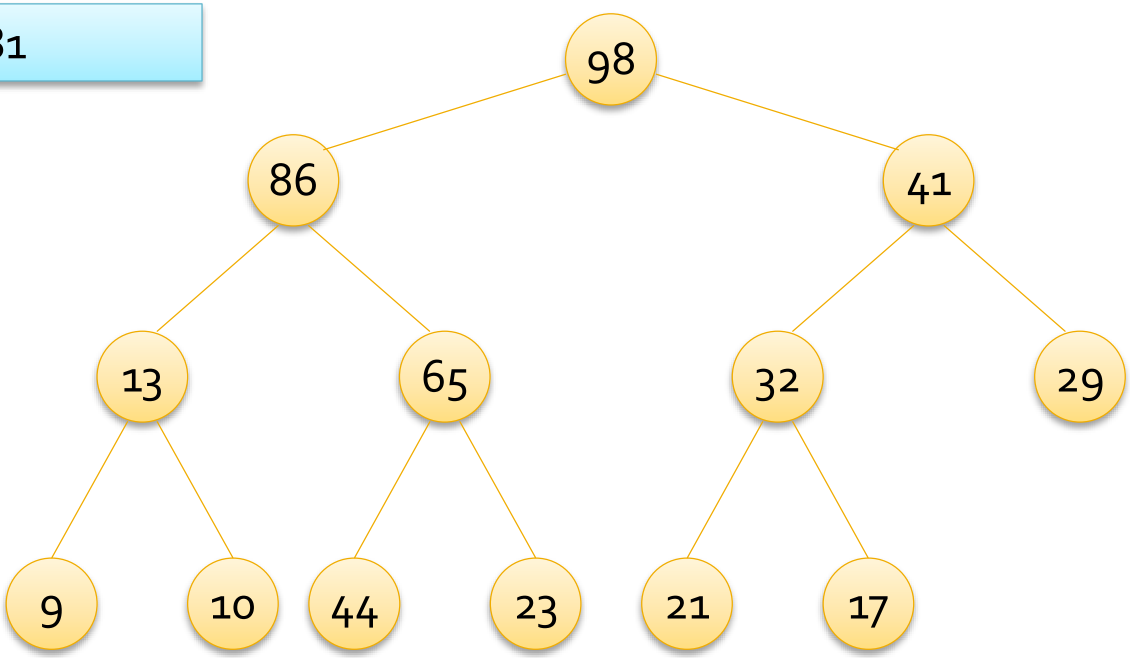
- On insertion the heap properties have to be maintained, remember that
  - A heap is a complete binary tree and
  - A partially ordered binary tree
- There are two general strategies that could be used to maintain the heap properties
  - Make sure that the tree is complete and then fix the ordering or
  - Make sure the ordering is correct first
  - Which is better?

Heap Insertion Sketch

- The insertion algorithm first ensures that the tree is complete
    - Make the new item the first available (left-most) leaf on the bottom level
    - i.e. the first free element in the underlying array
- Fix the partial ordering
    - Compare the new value to its parent
    - Swap them if the new value is greater than the parent
    - Repeat until this is not the case
        - Referred to as *bubbling up*, or *trickling up*

Heap Insertion Example

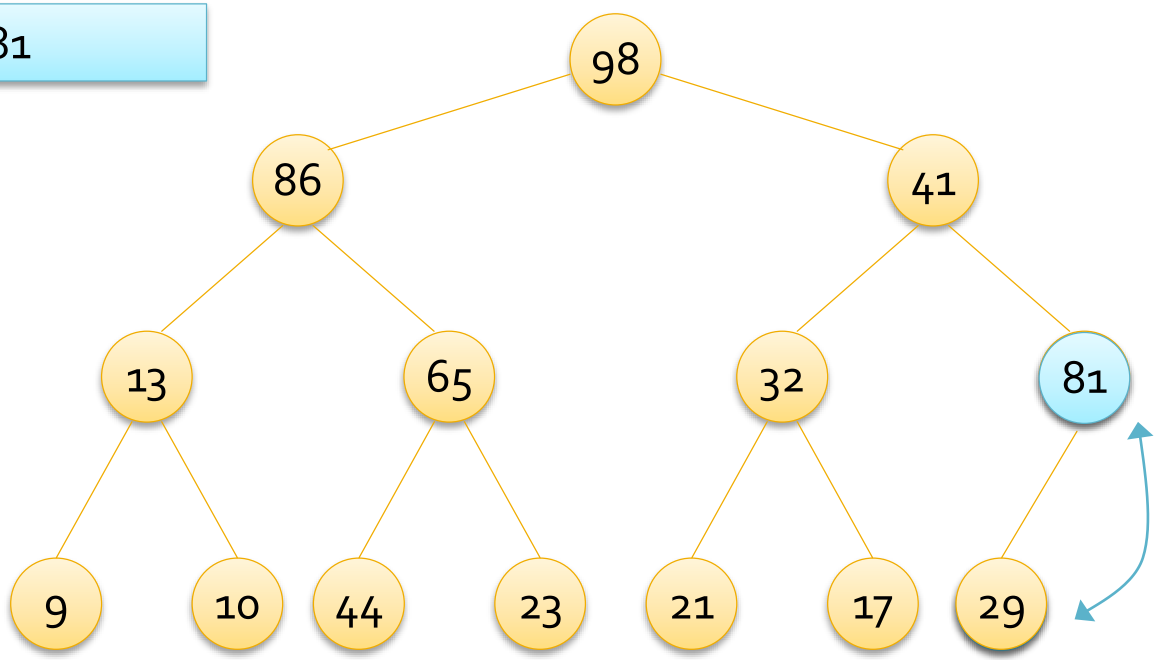
Insert 81



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

Heap Insertion Example

Insert 81

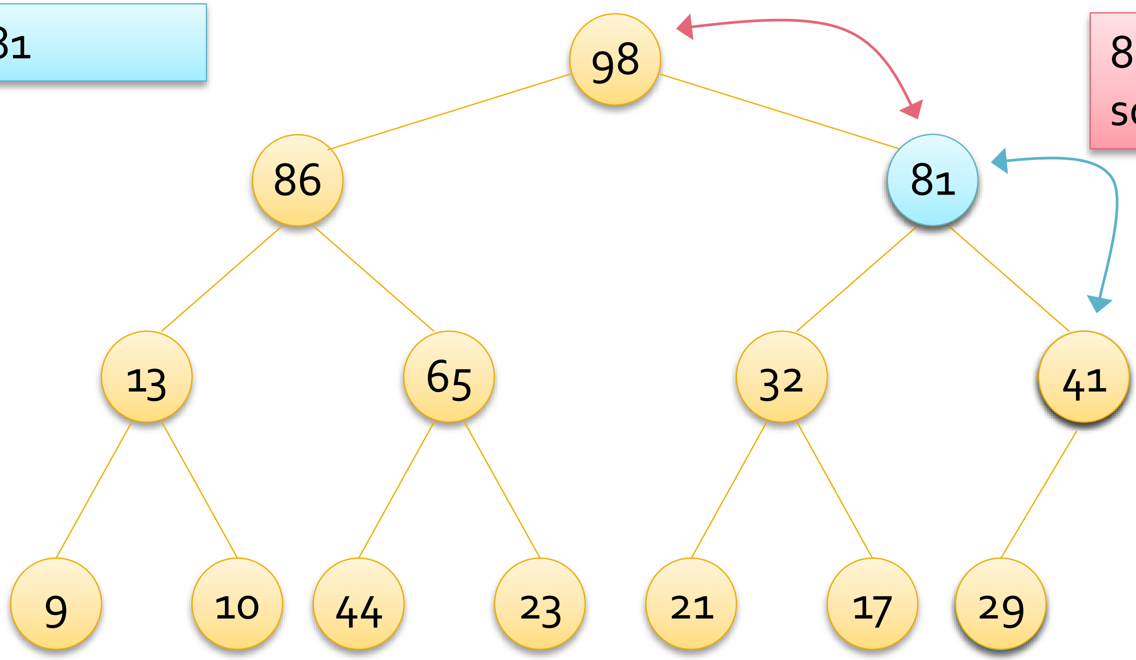


(13-1)/2 = 6

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
value	98	86	41	13	65	32	81	9	10	44	23	21	17	29

Heap Insertion Example

Insert 81



81 is less than 98 so finished

(6-1)/2 = 2

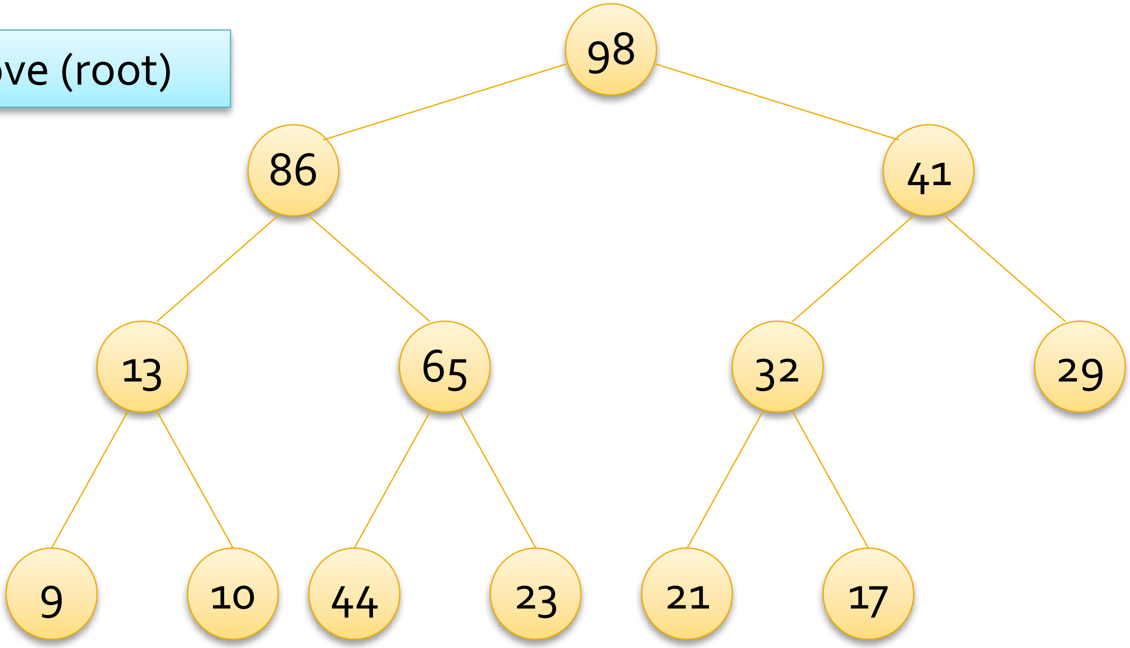
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
value	98	86	81	13	65	32	41	9	10	44	23	21	17	29

Heap Removal

- Make a temporary copy of the root’s data
- Similarly to the insertion algorithm, first ensure that the heap remains complete
  - Replace the root node with the right-most leaf
  - i.e. the highest (occupied) index in the array
- Swap the new root with its largest valued child until the partially ordered property holds
  - i.e. *bubble down*
- Return the root’s data

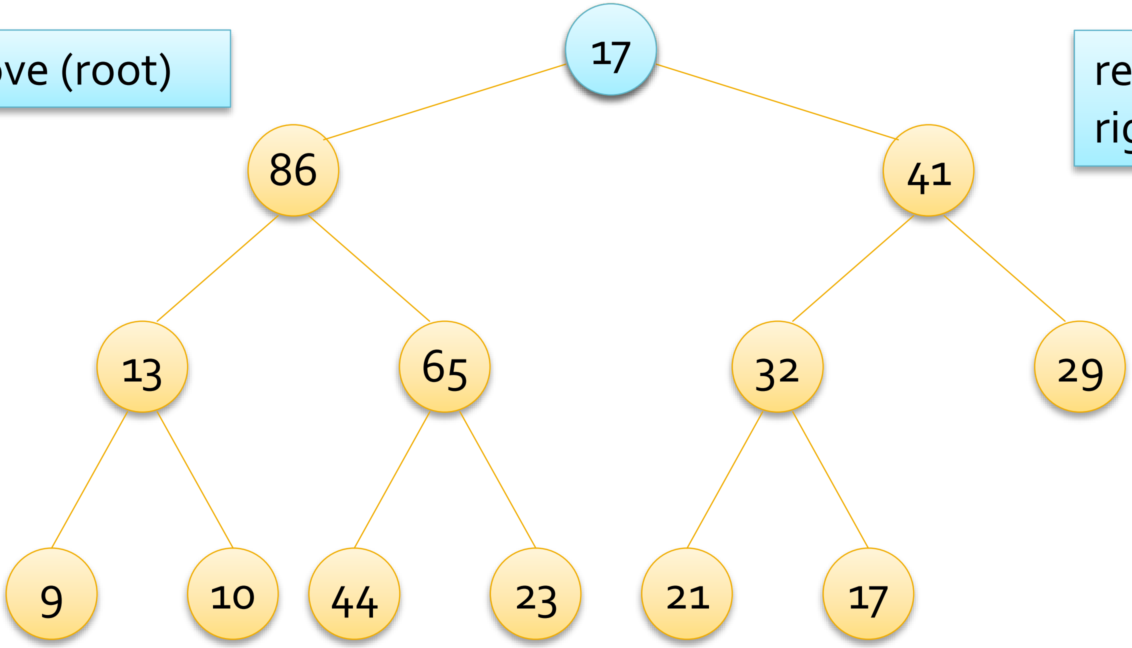
Heap Array Example

Remove (root)



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 Array Example



Remove (root)

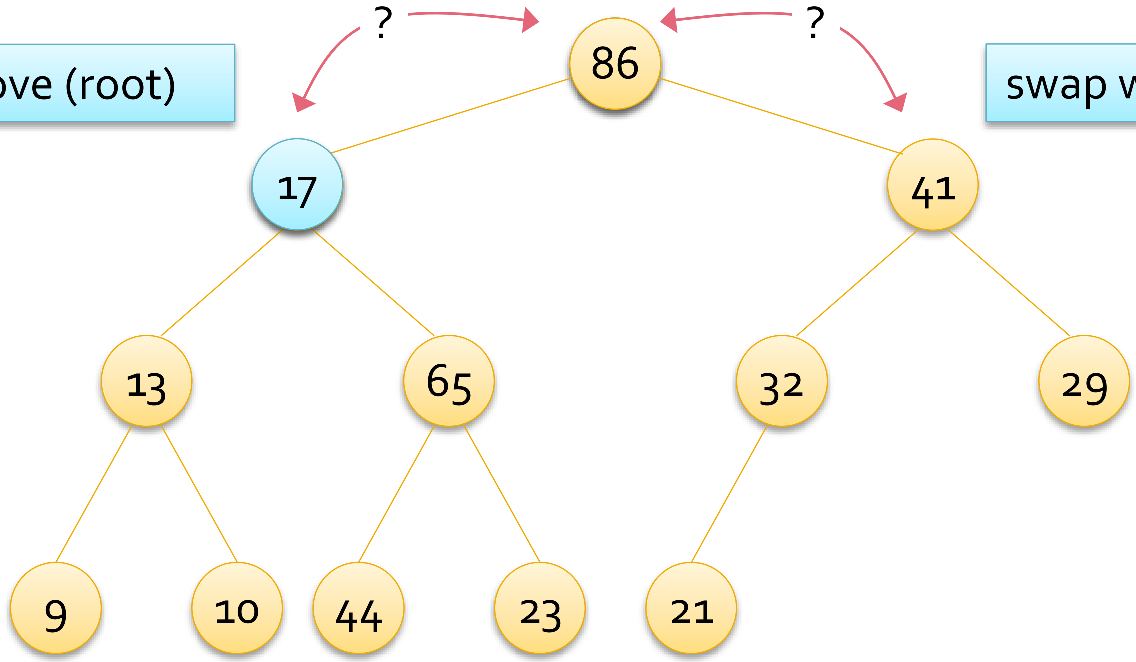
replace root with right-most leaf

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

Heap Array Example

Remove (root)

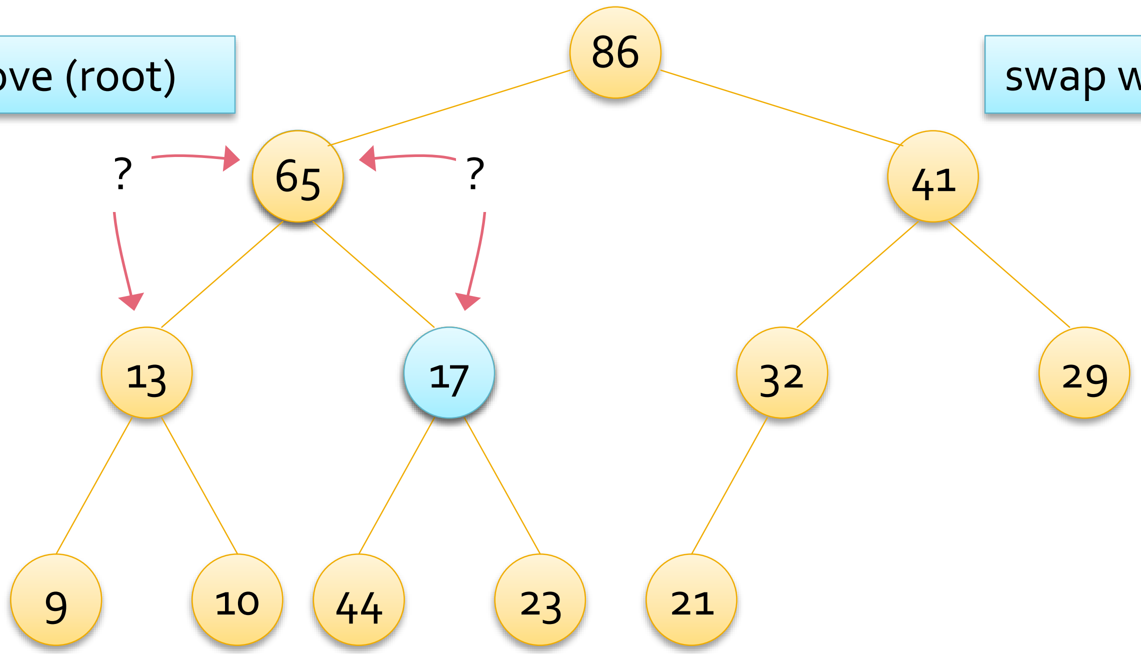
swap with larger child



children of root: 2*0+1, 2*0+2 = 1, 2

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

Heap Array Example



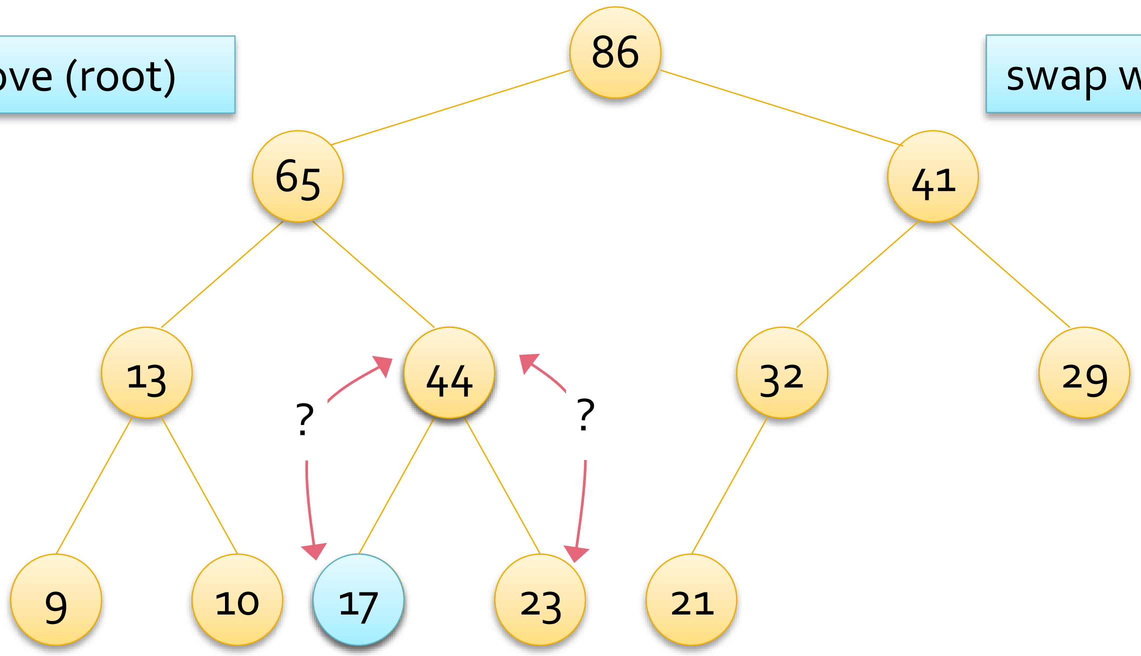
Remove (root)

swap with larger child

children of 1: 2*1+1, 2*1+2 = 3, 4

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

Heap Array Example



Remove (root)

swap with larger child

children of 4: 2*4+1, 2*4+2 = 9, 10

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	

Bubble Up and Bubble Down

- Helper functions are usually written for preserving the heap property
  - *bubbleUp* ensures that the heap property is preserved from the start node up to the root
  - *bubbleDown* ensures that the heap property is preserved from the start node down to the leaves
- These functions may be implemented recursively or iteratively

BubbleUp Algorithm

```
void bubbleUp(int i){
  int parent = (i - 1) / 2;
  if (i > 0 && arr[i] > arr[parent]){
    int temp = arr[i];
    arr[i] = arr[parent];
    arr[parent] = temp;
    bubbleUp(parent);
  }
  // no else - implicit base case
}
```

Insertion Algorithm

```
void insert(int x){
  arr[size] = x;
  bubbleUp(size);
  size++;
}
```

Heap Efficiency

- Both insertion and removal into a heap involve at most *height* swaps
  - For insertion at most *height* comparisons
    - To bubble up the array
  - For removal at most *height* * 2 comparisons
    - To bubble down the array (have to compare two children)
- Height of a complete binary tree is $\lfloor \log_2(n) \rfloor$
  - Both insertion and removal are therefore $O(\log n)$

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 …

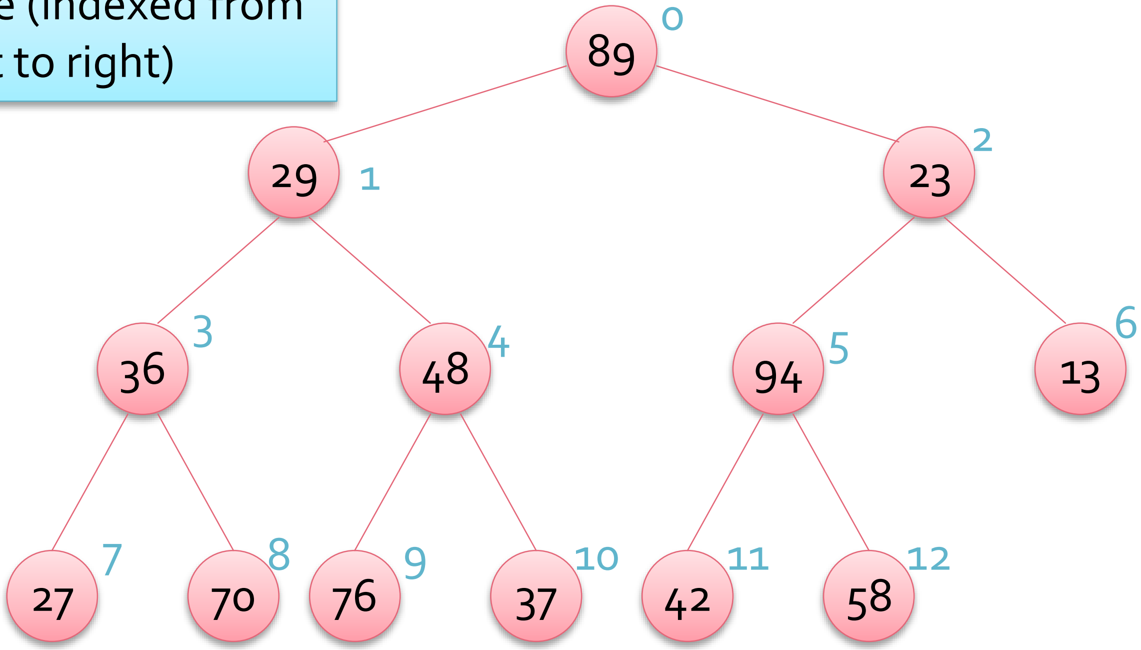
- A heap can be used to return sorted data
  - In $O(n*\log n)$ time
- However, we can't assume that the data to be sorted just happens to be in a heap!
  - **Aha!** But we can *put* it in a heap.
  - Inserting an item into a heap is a $O(\log n)$ operation so inserting *n* items is $O(n*\log n)$
- But 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 and work up to index 0
    - i.e. from the last non-leaf node to the root
- *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

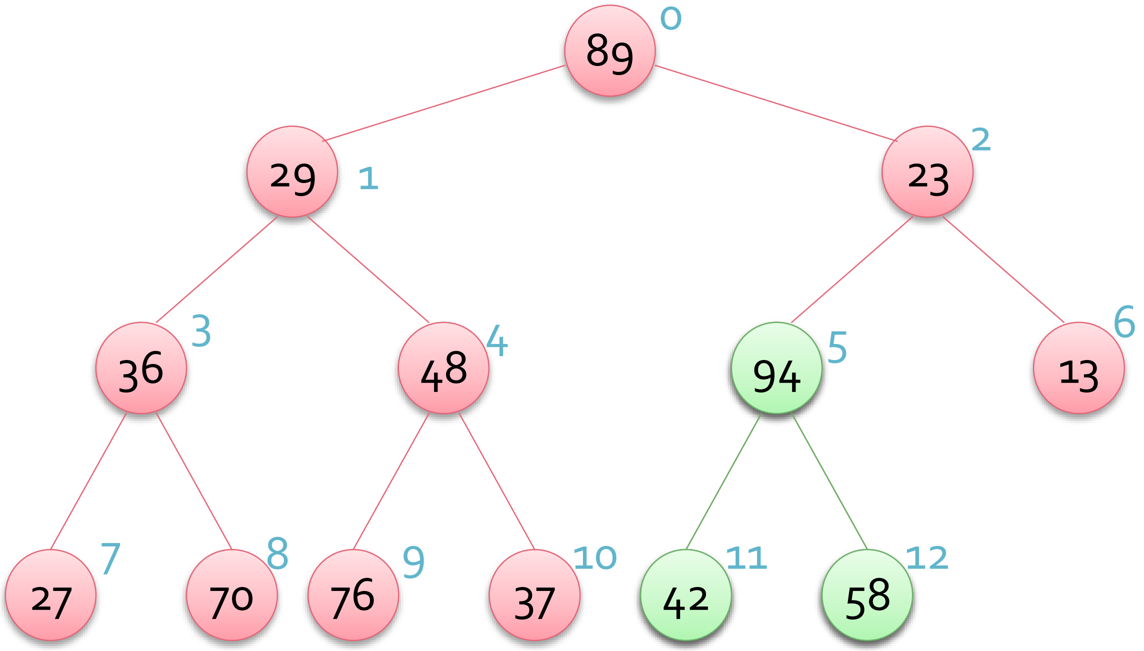
Assume unsorted input is contained in an array as shown here (indexed from top to bottom and left to right)



Heapify Example

n = 12, (n-1)/2 = 5

bubbleDown(5)

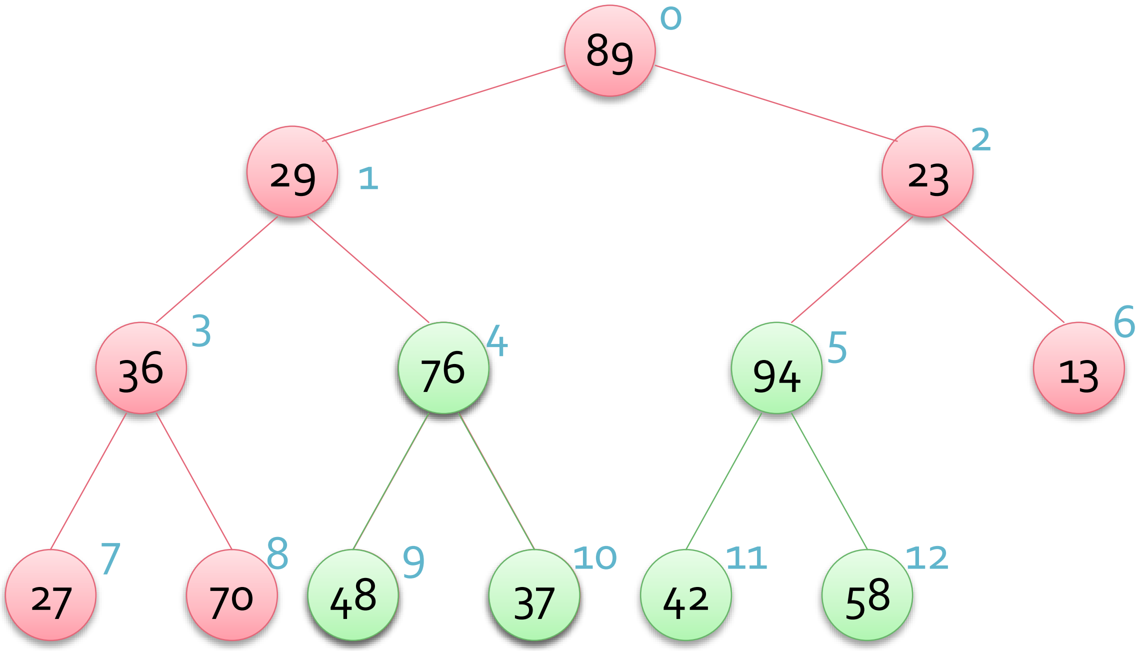


Heapify Example

n = 12, (n-1)/2 = 5

bubbleDown(5)

bubbleDown(4)



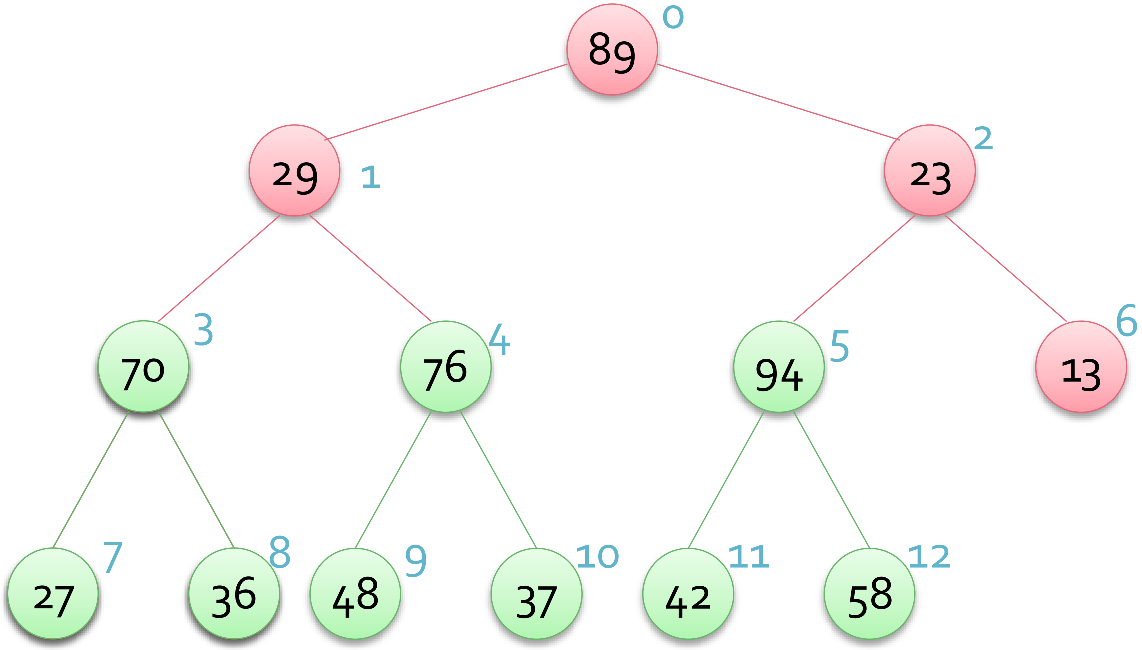
Heapify Example

n = 12, (n-1)/2 = 5

bubbleDown(5)

bubbleDown(4)

bubbleDown(3)



Heapify Example

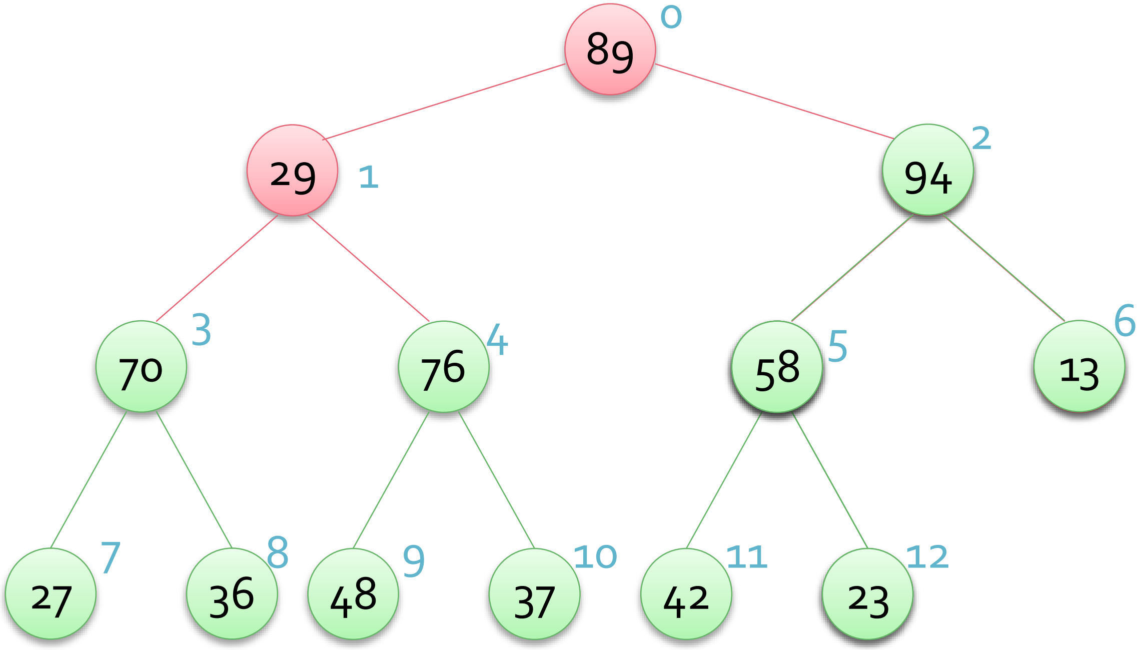
n = 12, (n-1)/2 = 5

bubbleDown(5)

bubbleDown(4)

bubbleDown(3)

bubbleDown(2)



Heapify Example

n = 12, (n-1)/2 = 5

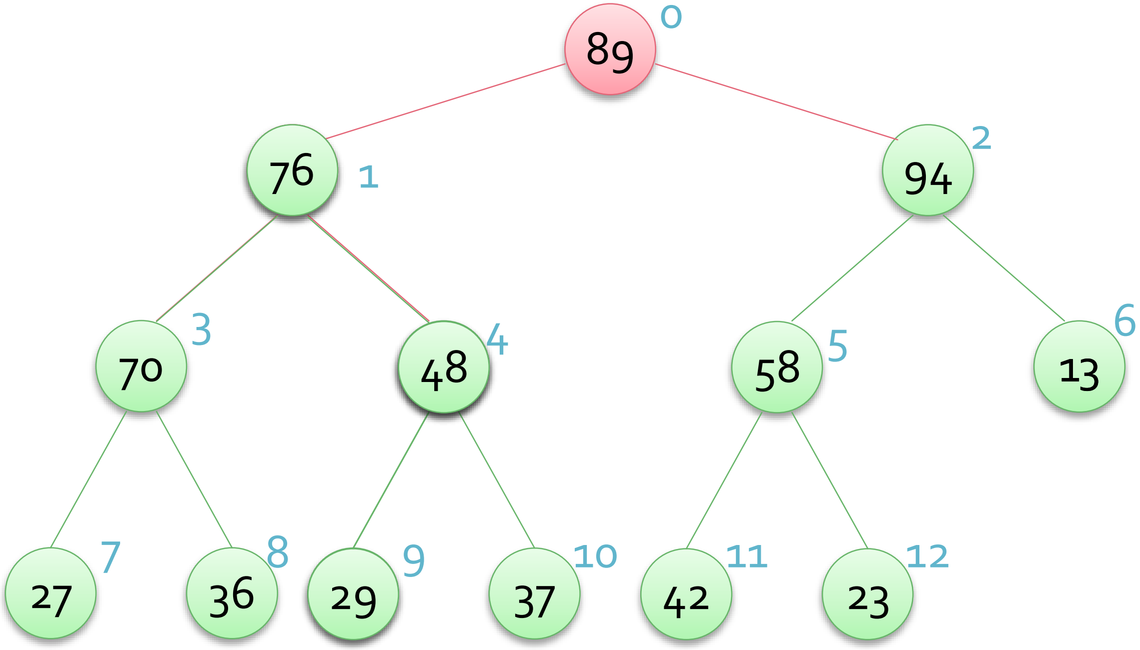
bubbleDown(5)

bubbleDown(4)

bubbleDown(3)

bubbleDown(2)

bubbleDown(1)



Heapify Example

n = 12, (n-1)/2 = 5

bubbleDown(5)

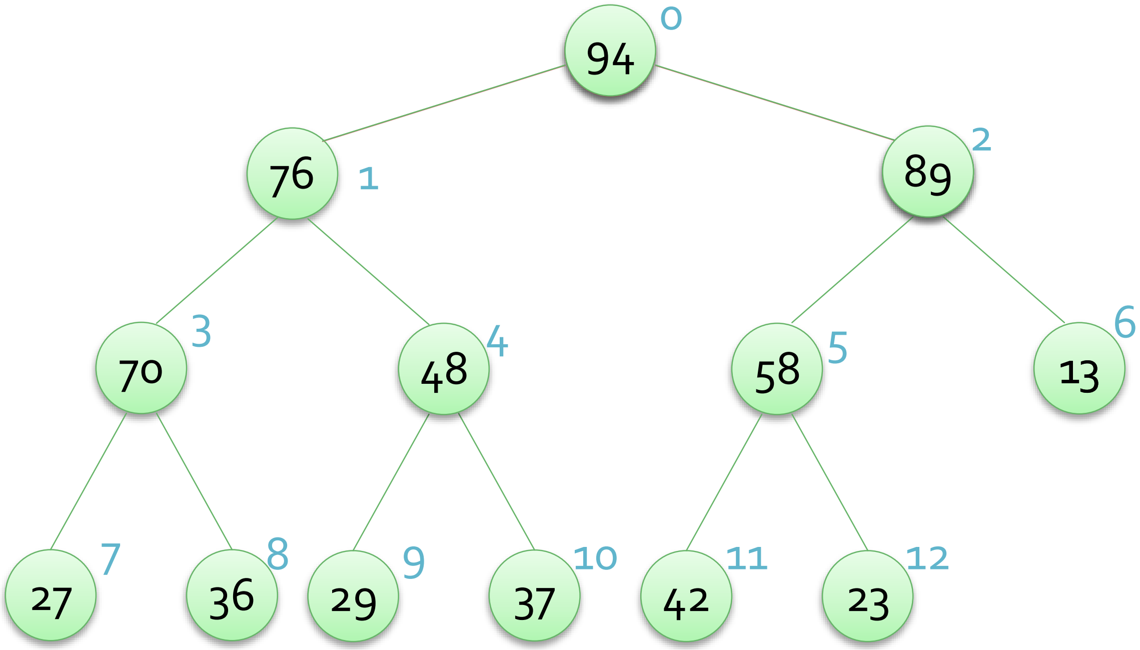
bubbleDown(4)

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
  - *bubbleDown* is only called on ½*n* nodes
  - and mostly on sub-trees

HeapSort Algorithm Sketch

- Heapify the array
- Repeatedly remove the root
    - After each removal swap the root with the last element in the tree
    - The array is divided into a heap part and a sorted part
- At the end of the sort the array will be sorted in reverse order

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