CS2010 – Data Structures and Algorithms II

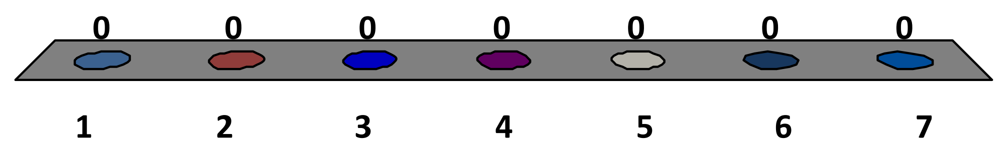
Lecture 04 – Heaps of Fun

stevenhalim@gmail.com

NUS
National University of Singapore
School of Computing

How many stripes that a woman has to see in “home pregnancy test kit” to confirm that she is pregnant?

1. 0
2. 1
3. 2
4. 3
5. 4
6. 7?
7. What stripes?

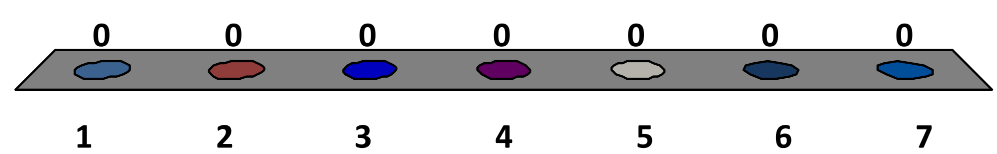


0 of 120

PS1 (Already open for 1+ week), I…

1. Have not even read it ☹
2. Have read, but confused
3. Have solved Subtask 1
4. Have solved Subtask 2
5. Have solved Subtask 3
6. Have solved Subtask 4
7. Have solved R-option ☺

0 of 120



Outline

- What are you going to learn in this lecture?
  - Motivation: Abstract Data Type: **PriorityQueue**
  - **Heap** data structure
  - Building Heap from a set of n numbers in **O(n)**
  - **Heap sort**
  - CS2010 PS2: “Scheduling Deliveries Problem”

- Reference in CP 2.5 book: Page 44-46 + 144-146

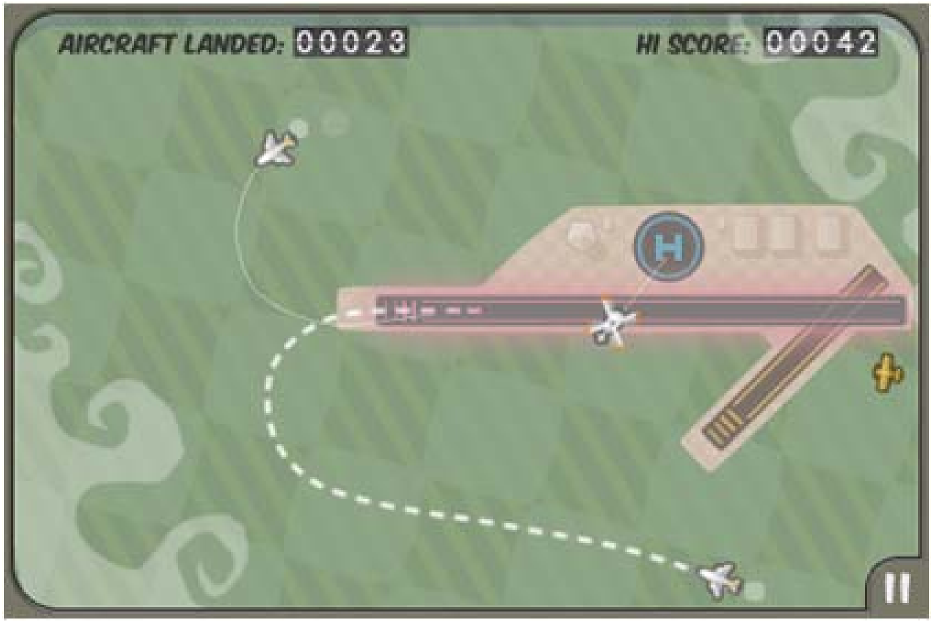
Regarding Today's Topic

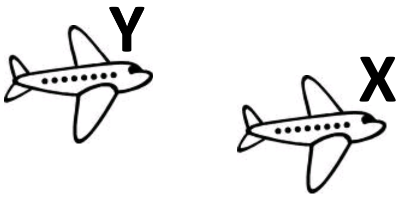
1. I do **not** know heap data structure yet, please teach me some *basic stuffs*
2. I already know heap, please teach me *more*

0 of 120

Abstract Data Type: PriorityQueue (1)

- Imagine that you are the Air Traffic Controller:
  - You have scheduled the next **aircraft X** to land in the **next 3 minutes**, and **aircraft Y** to land in the **next 6 minutes**
  - Both have enough fuel for at least the next **15 minutes** and both are just **2 minutes** away from your airport





The next slide is hidden…

- Attend the lecture to figure out

Abstract Data Type: PriorityQueue

- Important Basic Operations:
  - Enqueue(x)
    - Put a new item x in the priority queue PQ (in some order)
  - y $\leftarrow$ Dequeue()
    - Return an item y that has the **highest priority** (key) in the PQ
    - If there are more than one item with highest priority, return the one that is inserted first (FIFO)

Few Points To Remember

- Data Structure (DS) is…
  - A way to **store** and **organize data** in order to support efficient insertions, searches, deletions, queries, and/or updates
- Most data structures have propert(ies)
  - Each operation on that data structure has to **maintain** that propert(ies)

PriorityQueue Implementation (1)

- **Array-Based** Implementation (Strategy 1)
  - Property: The content of array is always in correct order
  - Enqueue(x)
    - Find the **correct insertion point**, O(n)
  - y ← Dequeue()
    - Return the **front-most item** which has the highest priority, O(1)

Index	0 (front)	1 (back)
Key	Aircraft X*	Aircraft Y*

Aircraft Z**

Index	0 (front)	1	2 (back)
Key	Aircraft Z**	Aircraft X*	Aircraft Y*

PriorityQueue Implementation (2)

- **Array-Based** Implementation (Strategy 2)
  - Property: dequeue() operation returns the correct item
  - Enqueue(x)
    - Put the new item at the **back of the queue**, O(1)
  - y ← Dequeue()
    - Scan the whole queue, return **first item with highest priority**, O(n)

Index	0	1 (back)
Key	Aircraft X*	Aircraft Y*

Aircraft Z**

Index	0	1	2 (back)
Key	Aircraft X*	Aircraft Y*	Aircraft Z**

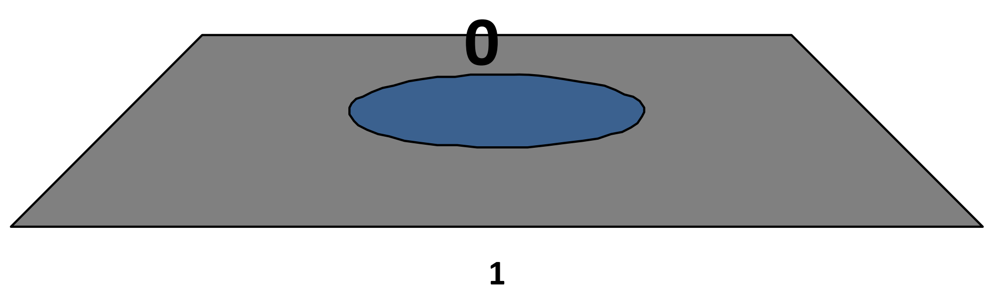
PriorityQueue Implementation (3)

- If we just stop at CS1020 (or first half of CS2020) knowledge level:

Strategy	Enqueue	Dequeue
Array-Based PQ (1)	O(N)	O(1)
Array-Based PQ (2)	O(1)	O(N)
Can we do better?	O(?)	O(?)

Can we do better?

1. Can ☺
   I have seen the answer in the middle of this lecture notes



0 of 120

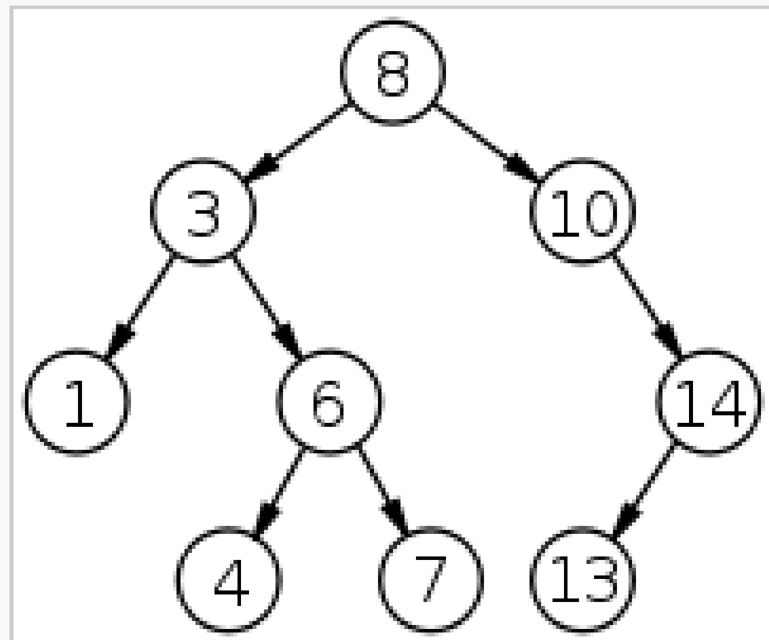
Visualization:

www.comp.nus.edu.sg/~stevenha/visualization/heap.html

INTRODUCING HEAP DATA STRUCTURE

Quick Review

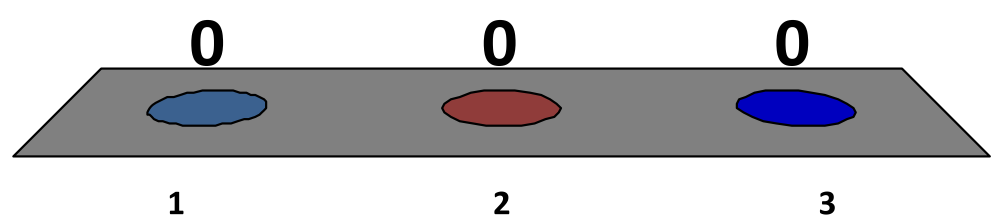
- Heap is similar to what you already know: *Binary Search Tree (BST)*
  - Vertex/Node/Item
  - Edge
  - Root
  - Internal Nodes
  - Leaves
  - Binary Tree
  - Left/Right Sub-Tree
  - The **BST Property**…



A binary search tree of size 9 and depth 3, with root 8 and leaves 1, 4, 7 and 13

The BST Property Is...

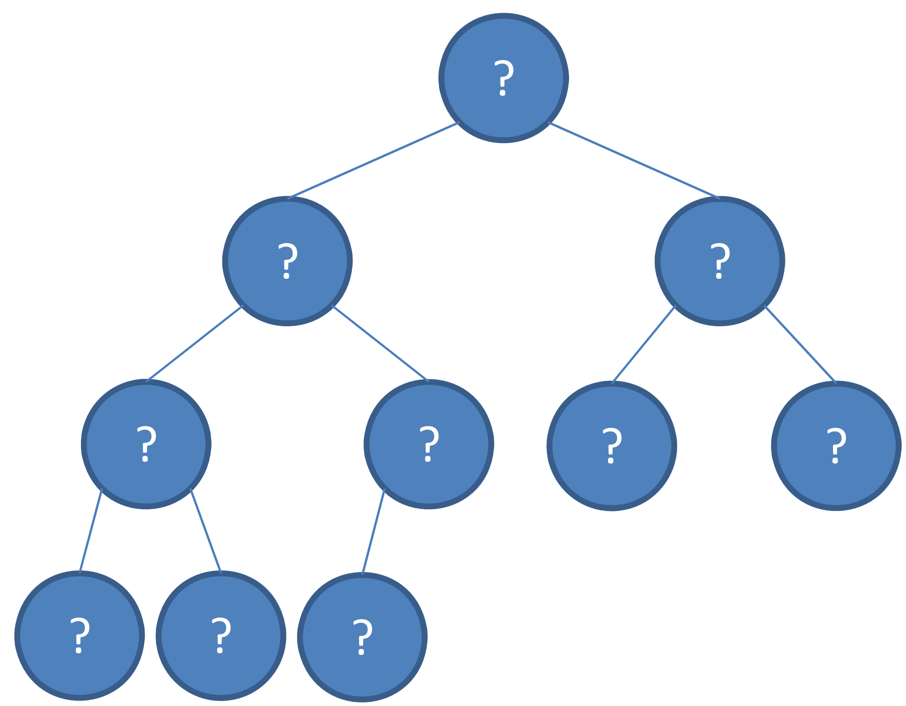
1. x.key < x.left.key < x.right.key
2. x.right.key < x.left.key < x.key
3. x.left.key < x.key < x.right.key



0 of 120

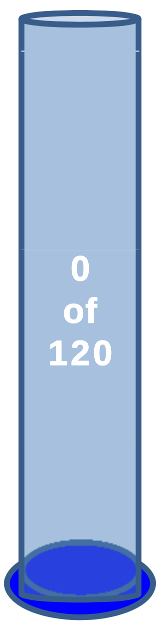
Complete Binary Tree

- Introducing a few more concepts:
  - **Complete** Binary Tree
    - Binary tree in which every level, *except possibly the last*, is completely filled, and all nodes are as far left as possible
  - If you have a complete binary tree of N items, what will be **the height of it**?

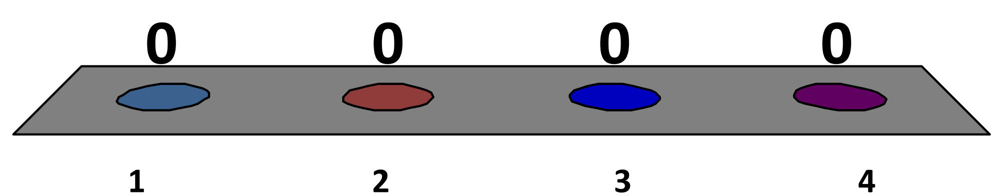


The Height of a Complete Binary Tree of N Items is...

1. O(N)
2. O(sqrt(N))
3. O(log N)
4. O(1)



Now, memorize this answer! We will need that for nearly all time complexity analysis of heap operations



Storing a Complete Binary Tree

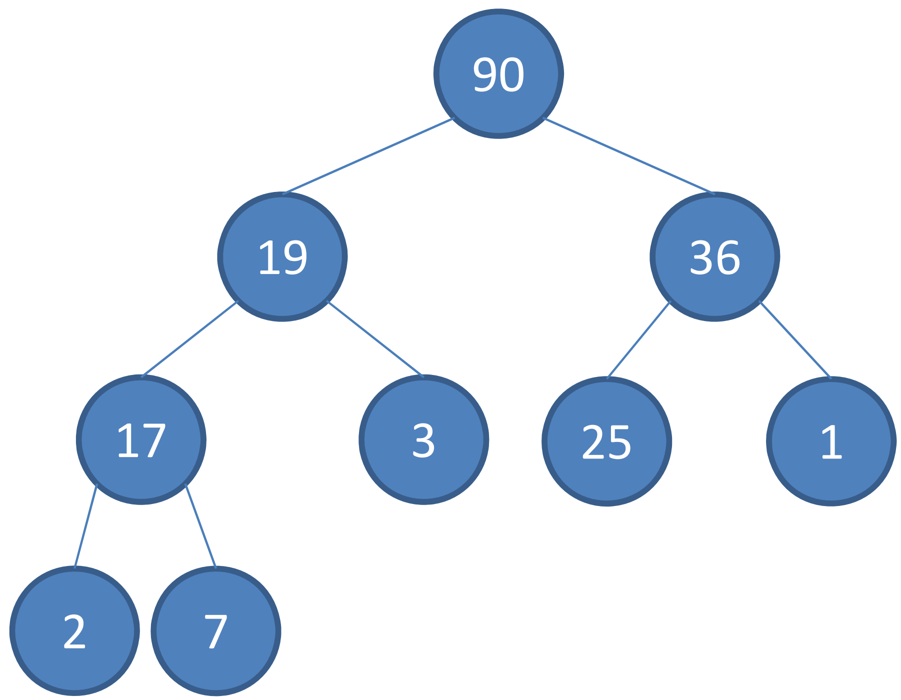
- As a **1-based** compact array: `A[1..size(A)]`

`size(A)`

0	1	2	3	4	5	6	7	8	9	10	11
NIL	90	19	36	17	3	25	1	2	7	-	-

`heapsize ≤ size(A)`

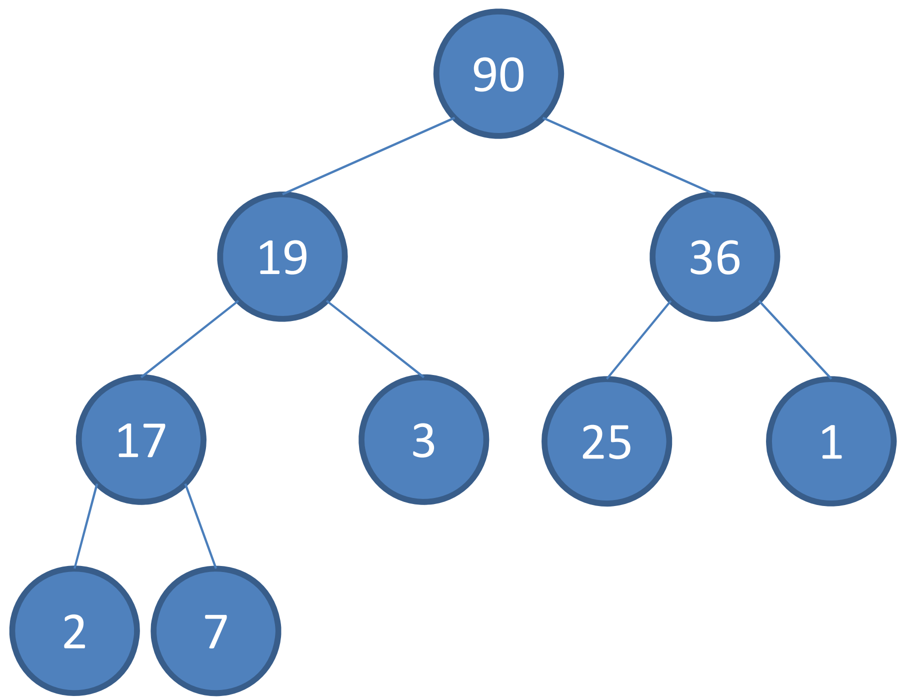
- Navigation operations:
  - `parent(i) = floor(i/2)`
    - Except for `i = 1 (root)`
  - `left(i) = 2*i`
  - `right(i) = 2*i + 1`
    - No left/right child when:
    - `left(i) > heapsize`
    - `right(i) > heapsize`



Q: Why not 0-based?

The Heap Property

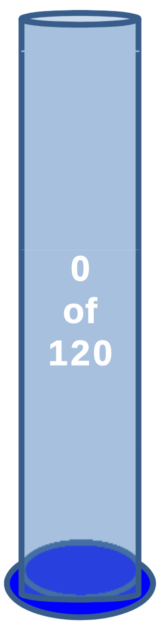
- The **Heap property** (except for root)
  - `A[parent(i)]` $\geq$ `A[i]` **(max heap)**
  - `A[parent(i)]` $\leq$ `A[i]` **(min heap)**
- Without loss of generality, we will use "**max heap**" for all examples in this lecture

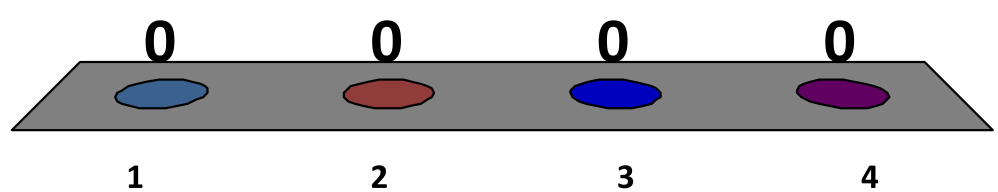


Q: Can we write max heap property as:
`A[i]` $\geq$ `A[left(i)] &&`
`A[i]` $\geq$ `A[right(i)]?`

The largest element in a **max heap** is stored at…

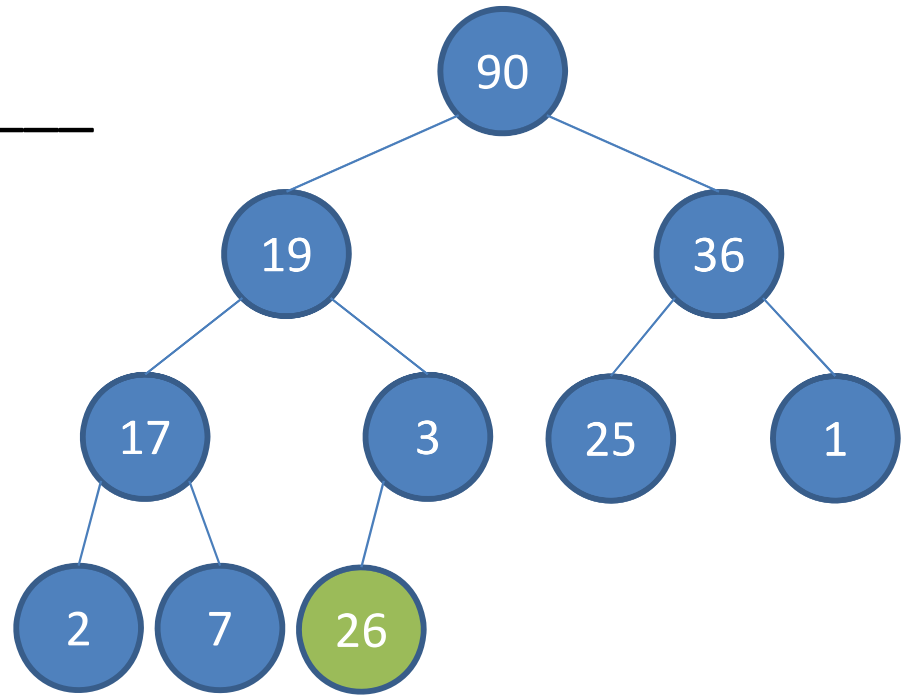
1. One of the leaves
2. One of the internal nodes
3. Can be anywhere in the heap
4. The root





Insertion to an Existing Max Heap

- The most appropriate insertion point into an existing heap is the **bottom-most**, **right-most new leaf**
- Why?
  - ______________________________
- But the Heap property can still be violated?
  - No problem,
    we use `ShiftUp(i)`
    to fix the heap property



0	1	2	3	4	5	6	7	8	9	10	11
0	90	19	36	17	3	25	1	2	7		

Insert(v) – Pseudo Code

```
Insert(v)
  heapsize = heapsize + 1; // extend, O(1)
  A[heapsize] = v // insert at the back, O(1)
  ShiftUp(heapsize) // fix the heap property
                    // in O(?)

// Preliminary analysis:
// Insert(v) depends on ShiftUp(i)
```

ShiftUp – Pseudo Code

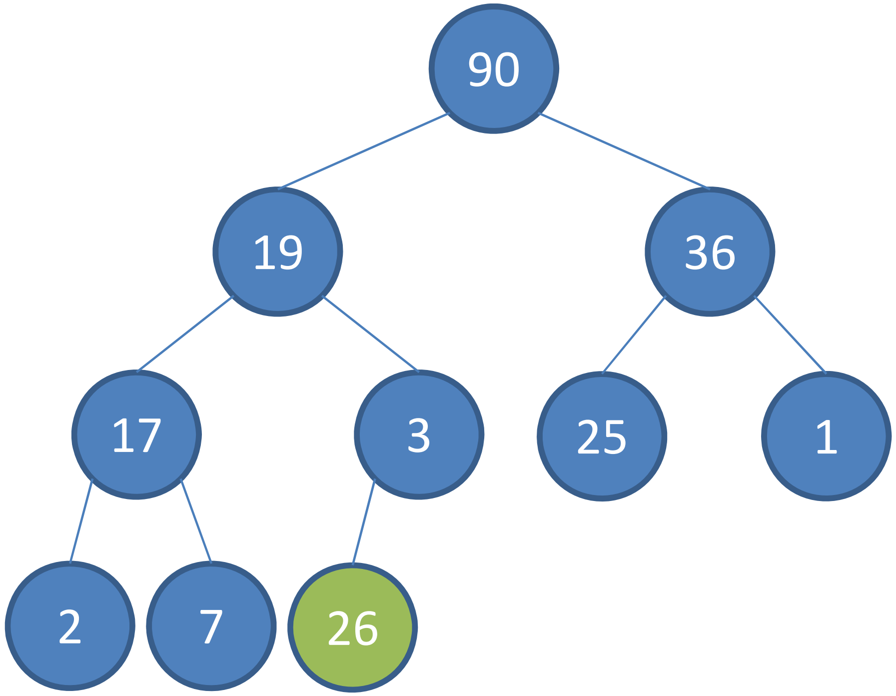
- Name is not unique: ShiftUp/BubbleUp/IncreaseKey/etc

“not root” “violates max heap property”

```
ShiftUp(i)
  while i > 1 and A[parent(i)] < A[i]
    swap(A[i], A[parent(i)])
    i = parent(i)
```

Animation (1)

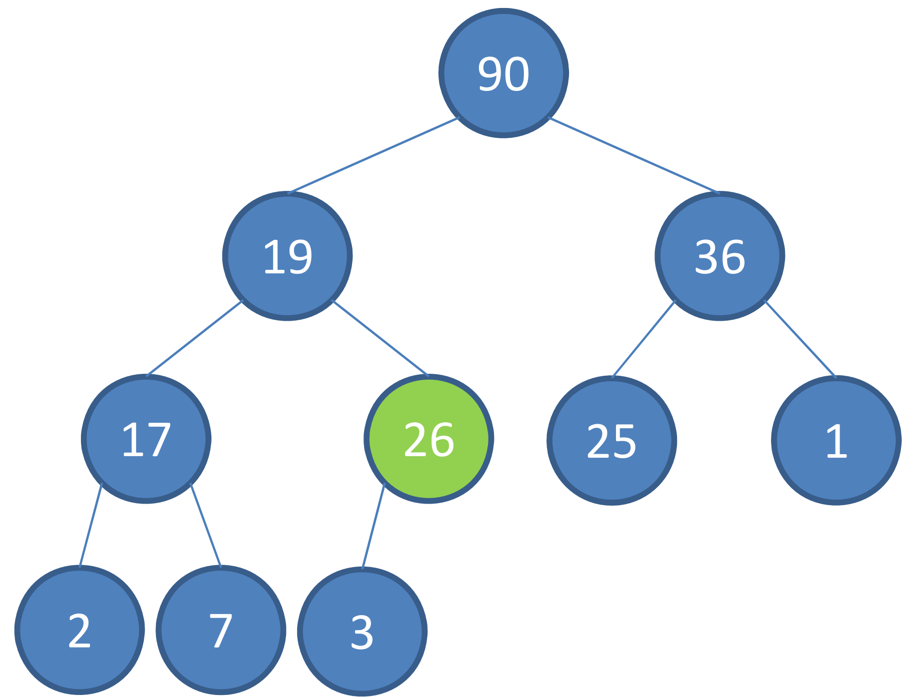
```
ShiftUp(i)
  while i > 1 and A[parent(i)] < A[i]
    swap(A[i], A[parent(i)])
    i = parent(i)
```



0	1	2	3	4	5	6	7	8	9	10	11
0	90	19	36	17	3	25	1	2	7	26	

Animation (2)

```
ShiftUp(i)
  while i > 1 and A[parent(i)] < A[i]
    swap(A[i], A[parent(i)])
    i = parent(i)
```

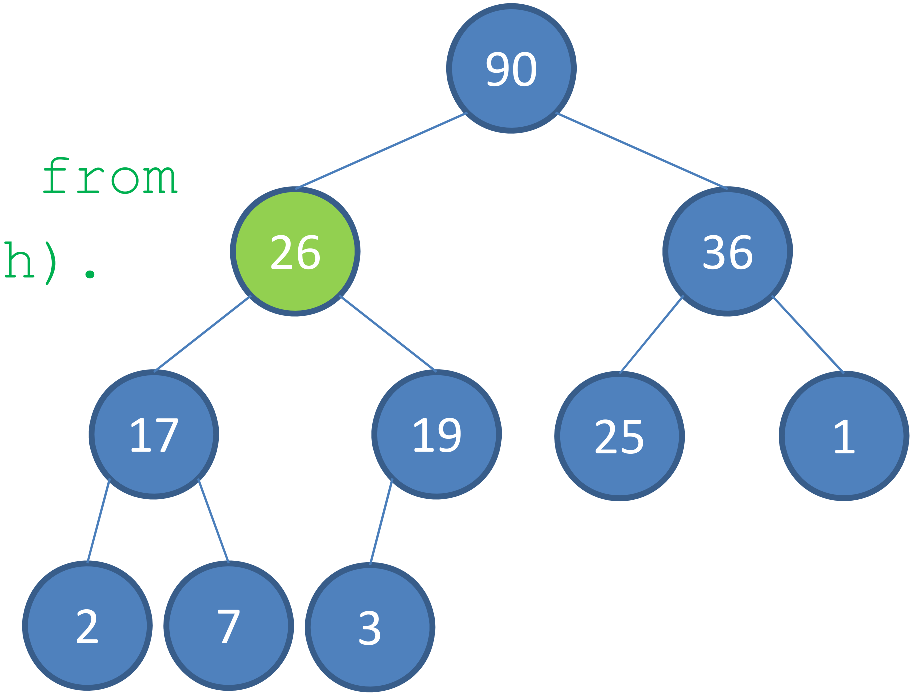


0	1	2	3	4	5	6	7	8	9	10	11
0	90	19	36	17	26	25	1	2	7	3	

Animation (3)

```
ShiftUp(i)
  while i > 1 and A[parent(i)] < A[i] // see below
    swap(A[i], A[parent(i)]) // O(1)
    i = parent(i) // O(1)

// Analysis: The worst case is from
// the deepest leaf to root O(h).
// In a complete binary tree,
// this h is just log n.
// Thus, ShiftUp AND
// Insert runs in
// O(log N)
```



0	1	2	3	4	5	6	7	8	9	10	11
0	90	26	36	17	19	25	1	2	7	3	

Deleting Max Element

- The max element of a max heap is at **the root**
- But simply taking the root out from a max heap will disconnect the complete binary tree ☹
- We do not want that…
- So, which node is the best candidate to **replace** the root yet still maintain complete binary tree property?
- Again the ______________________________ **existing leaf**
  - Which is again the last element in the compact array
- But the heap property can still be violated?
  - No problem, this time we call `ShiftDown(1)`

ExtractMax - Pseudocode

```
ExtractMax()
  maxV ← A[1] // O(1)
  A[1] ← A[heapsize] // O(1)
  heapsize = heapsize - 1 // O(1)
  ShiftDown(1) // O(?)
  return maxV

// Preliminary analysis:
// ExtractMax() depends on ShiftDown()
```

ShiftDown – Pseudo Code

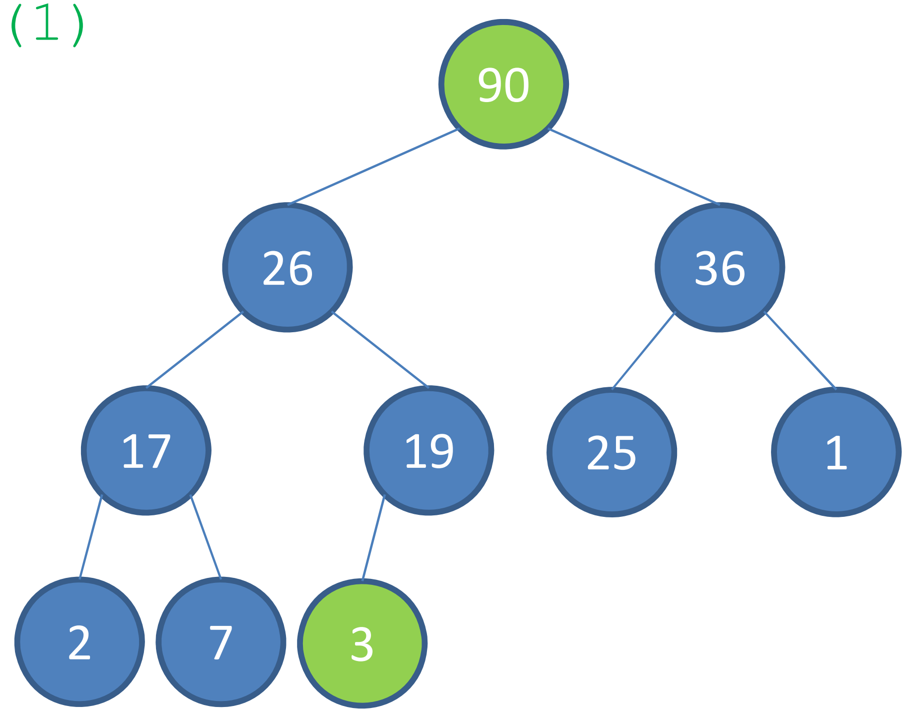
Again, name is not unique:
ShiftDown/BubbleDown/Heapify/etc

```
ShiftDown(i)
  while i <= heapsize
    maxV ← A[i]; max_id = i;
    if left(i) <= heapsize and maxV < A[left(i)]
      maxV ← A[left(i)]; max_id ← left(i)
    if right(i) <= heapsize and maxV < A[right(i)]
      maxV ← A[right(i)]; max_id ← right(i)

    if (max_id != i)
      swap(A[i], A[max_id])
      i = max_id;
    else
      break;
```

Animation (1)

```
ExtractMax()
  maxV ← A[1] // O(1)
  A[1] ← A[heapsize] // O(1)
  heapsize = heapsize - 1 // O(1)
  ShiftDown(1) // O(?)
  return maxV
```

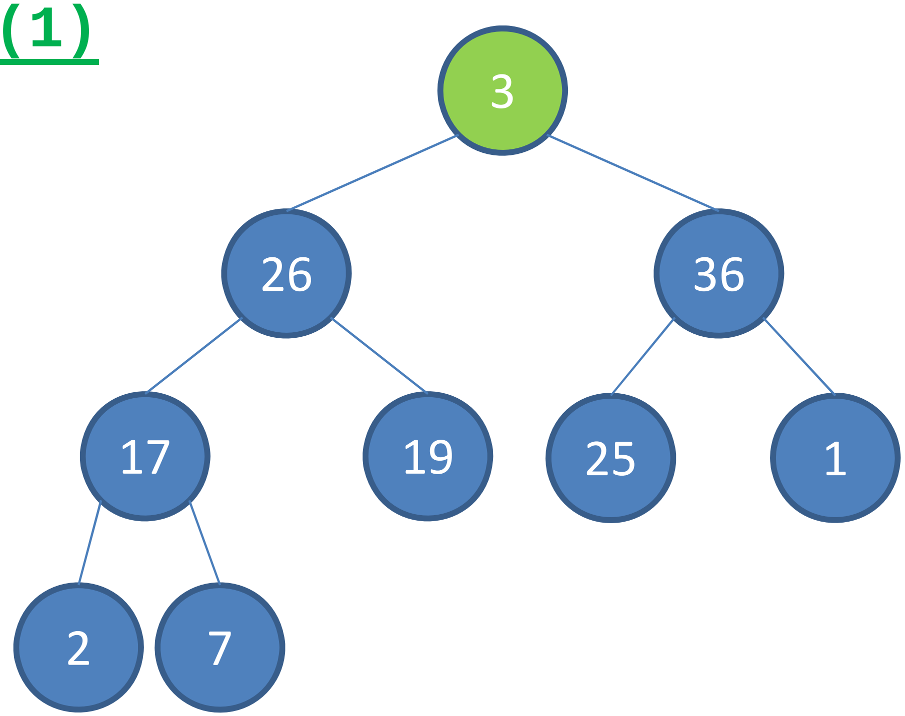


0	1	2	3	4	5	6	7	8	9	[10]	11
0	90	26	36	17	19	25	1	2	7	3	

Animation (2)

```
ExtractMax()
   maxV ← A[1] // O(1)
   A[1] ← A[heapsize] // O(1)
   heapsize = heapsize - 1 // O(1)
   ShiftDown(1) // O(?)
   return maxV
```

90 is stored at maxV and returned later after ShiftDown(1) is done

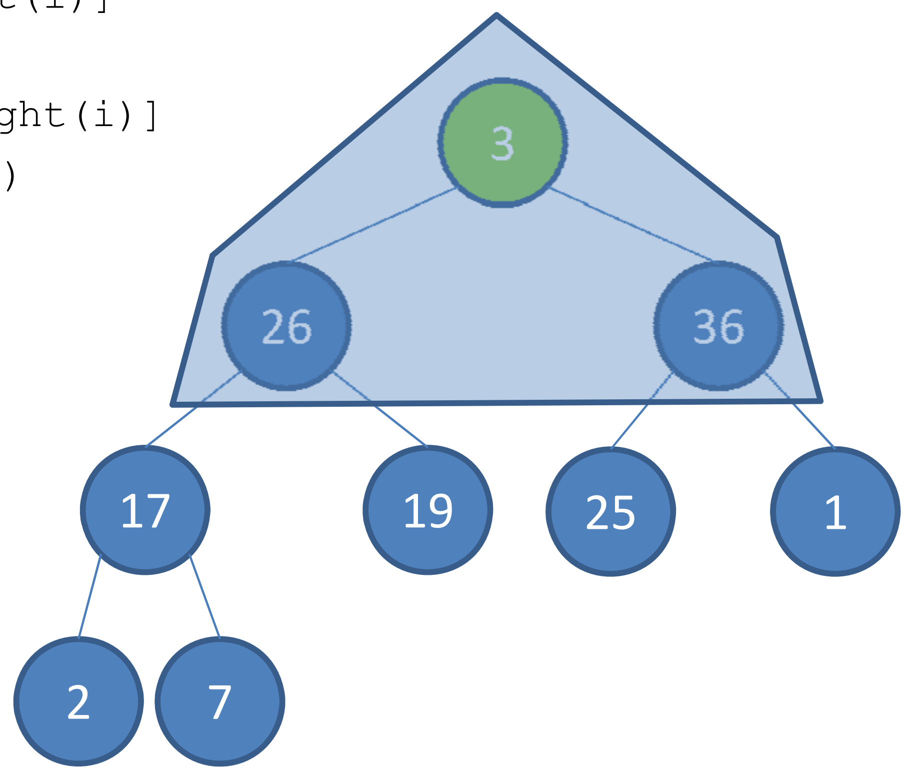


0	1	2	3	4	5	6	7	8	[9]	10	11
0	3	26	36	17	19	25	1	2	7		

Animation (3)

```
ShiftDown(i)
  while i <= heapsize
    maxV ← A[i]; max_id = i;
    if Left(i) <= heapsize and maxV < A[Left(i)]
      maxV ← A[Left(i)]; max_id ← Left(i)
    if Right(i) <= heapsize and maxV < A[Right(i)]
      maxV ← A[Right(i)]; max_id ← Right(i)

    if (max_id != i)
      swap(A[i], A[max_id])
      i = max_id;
    else
      break;
```

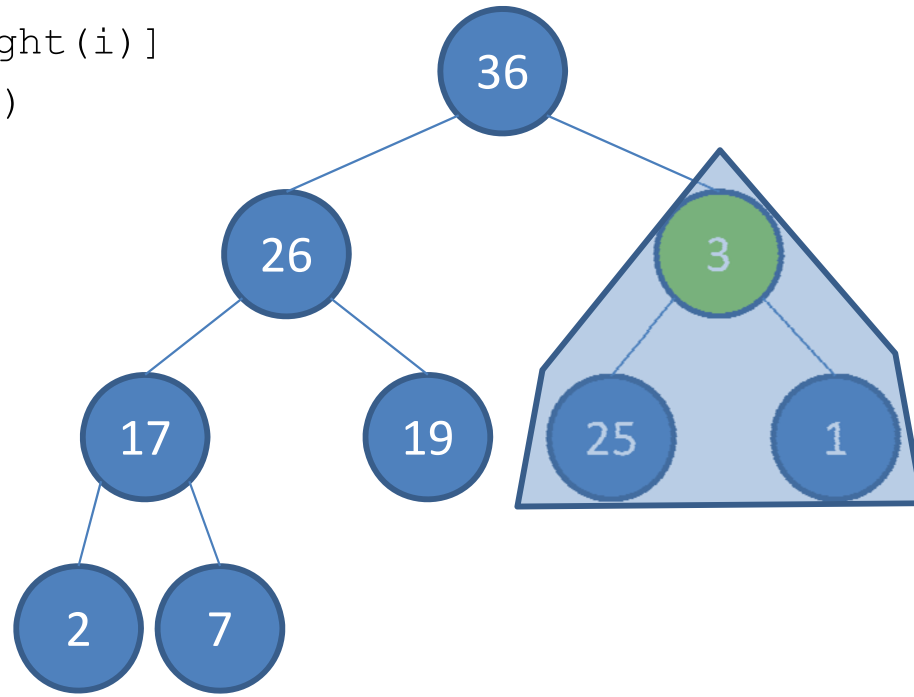


0	1	2	3	4	5	6	7	8	[9]	10	11
0	3	26	36	17	19	25	1	2	7		

Animation (4)

```
ShiftDown(i)
  while i <= heapsize
    maxV ← A[i]; max_id = i;
    if Left(i) <= heapsize and maxV < A[Left(i)]
      maxV ← A[Left(i)]; max_id ← Left(i)
    if Right(i) <= heapsize and maxV < A[Right(i)]
      maxV ← A[Right(i)]; max_id ← Right(i)

    if (max_id != i)
      swap(A[i], A[max_id])
      i = max_id;
    else
      break;
```



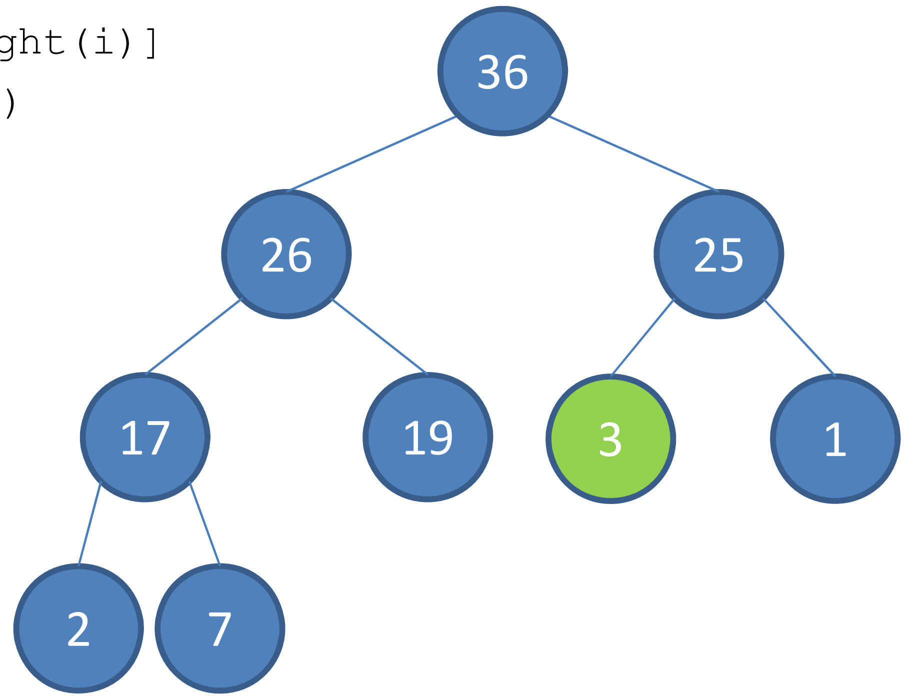
0	1	2	3	4	5	6	7	8	[9]	10	11
0	36	26	3	17	19	25	1	2	7		

Animation (5)

```
ShiftDown(i)
  while i <= heapsize // at most root to leaf! O(h) = O(log N)
    maxV ← A[i]; max_id = i;
    if Left(i) <= heapsize and maxV < A[Left(i)]
      maxV ← A[Left(i)]; max_id ← Left(i)
    if Right(i) <= heapsize and maxV < A[Right(i)]
      maxV ← A[Right(i)]; max_id ← Right(i)

    if (max_id != i)
      swap(A[i], A[max_id])
      i = max_id;
    else
      break;

// In overall, ShiftDown AND ExtractMax
// runs in O(h), which is just O(log N)
// in a complete binary tree
```



0	1	2	3	4	5	6	7	8	[9]	10	11
0	36	26	25	17	19	3	1	2	7		

PriorityQueue Implementation (4)

- Now, with new knowledge of *non linear* DS:

Strategy	Enqueue	Dequeue
Array-Based PQ (1)	O(N)	O(1)
Array-Based PQ (2)	O(1)	O(N)
Binary-Heap	Insert(key) O(log N)	ExtractMax() O(log N)

Summary so far:
Heap data structure is an efficient data structure -- O(log N) operations for enqueue/dequeue -- to implement ADT priority queue where ‘key’ represent the ‘priority’ of each item

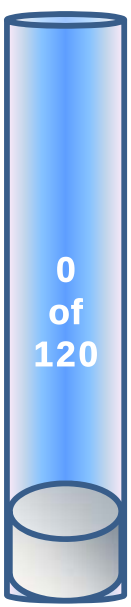
Next Items:

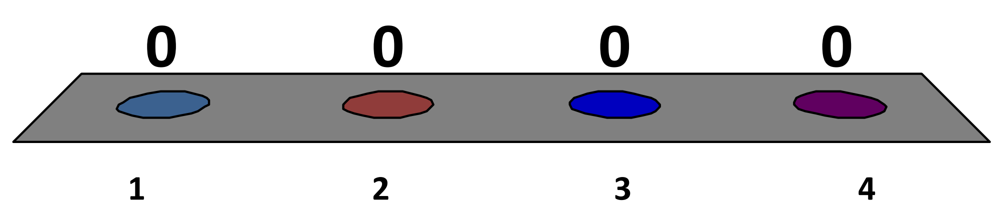
- Building Max Heap from an ordinary Array, slow one, O(n log n)
- And the faster one, O(n)
- Heap Sort
- Java Implementation of Max Heap

5 MINUTES BREAK

Review: We have seen MergeSort. It can sort N items in...

1. $O(N^2)$
2. $O(N \log N)$
3. $O(N)$
4. $O(\log N)$





HeapSort Pseudo Code

- With a max heap, we can do sorting too ☺
  - Just call ExtractMax() N times
  - If we do not have a max heap yet, simply build one!

```
HeapSort(array)
  BuildHeap(array) // O(?)
  N ← size(array)
  for i from 1 to N // O(N)
    A[N - i + 1] ← ExtractMax() // O(log N)
  return A


// Preliminary analysis:
// HeapSort runs in O(? + N log N)
```

BuildHeap (Version 1)

```
BuildHeapSlow(array) // naïve version
  N ← size(array)
  A[0] ← 0 // dummy entry
  for i = 1 to N // O(N)
    Insert(array[i]) // O(log N)

// Analysis: This clearly runs in O(N log N)
```

- Can we do better?

BuildHeap (Version 2)

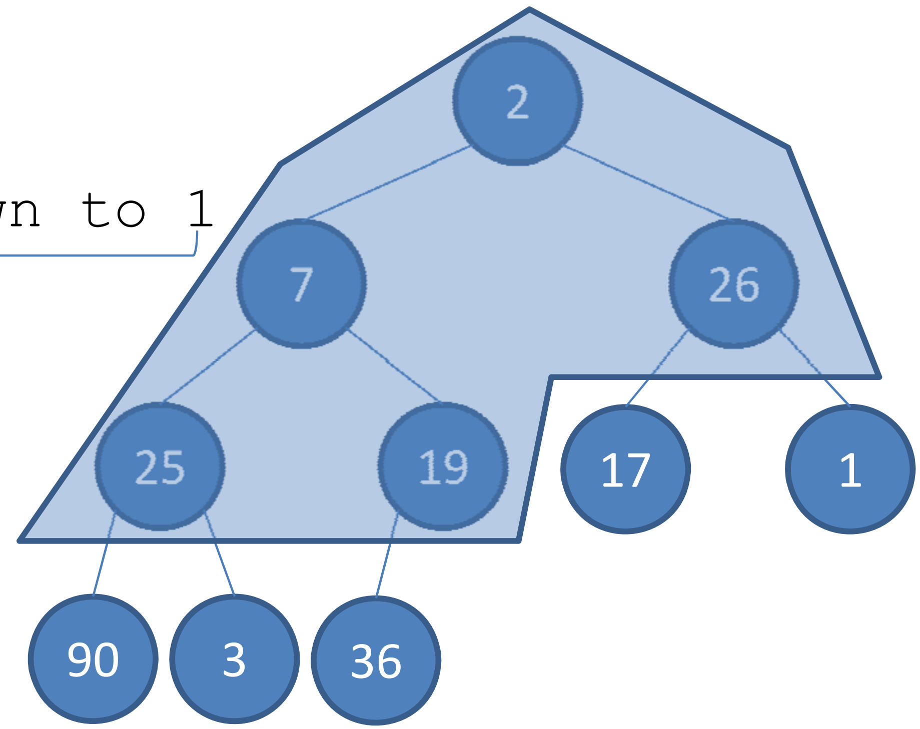
```
BuildHeap(array)
  heapsize ← size(array)
  A[0] ← 0 // dummy entry
  for i = 1 to heapsize // copy the content O(N)
    A[i] ← array[i]
  for i = parent(heapsize) down to 1 // O(N/2)
    ShiftDown(i) // O(log N)

// Analysis: Is this also O(N log N) ??
```

Animation (1)

```
BuildHeap(array)
   heapsize ← size(array)
   A[0] ← 0
   for i = 1 to heapsize
      A[i] ← array[i]
   for i = parent(heapsize) down to 1
      ShiftDown(i)
```

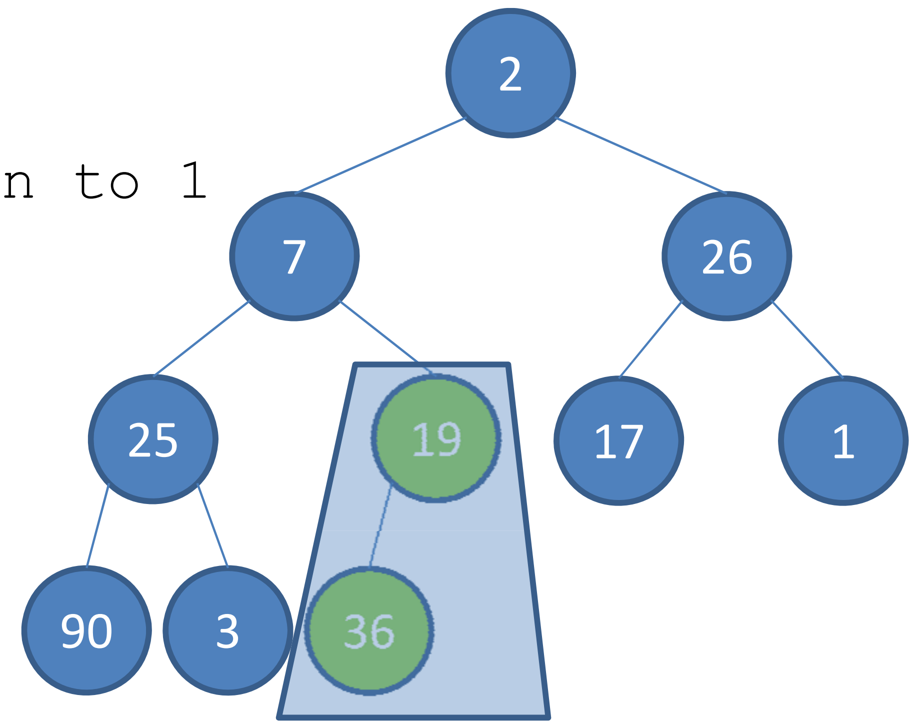
Internal Nodes Only!



0	1	2	3	4	5	6	7	8	9	10	11
0	2	7	26	25	19	17	1	90	3	36	

Animation (2)

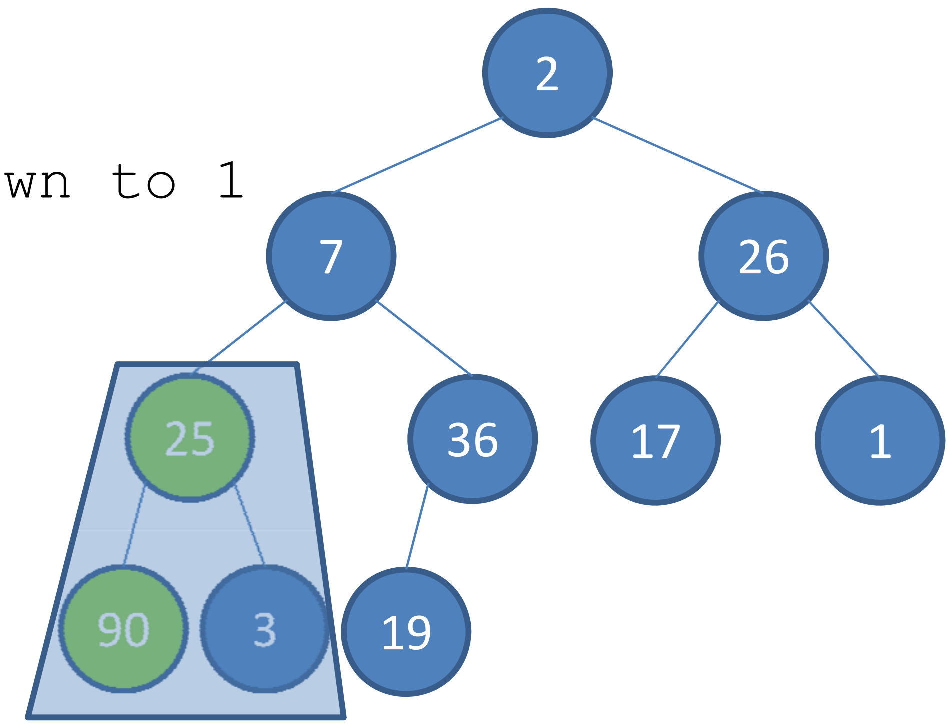
```
BuildHeap(array)
  heapsize ← size(array)
  A[0] ← 0
  for i = 1 to heapsize
    A[i] ← array[i]
  for i = parent(heapsize) down to 1
    ShiftDown(i)
```



0	1	2	3	4	5	6	7	8	9	10	11
0	2	7	26	25	19	17	1	90	3	36	

Animation (3)

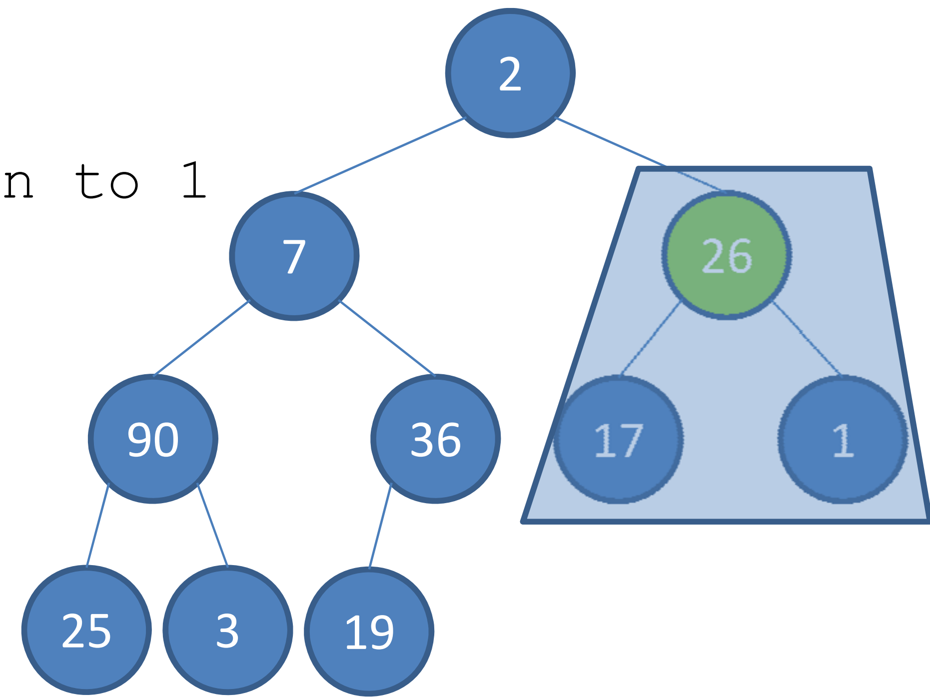
```
BuildHeap(array)
  heapsize ← size(array)
  A[0] ← 0
  for i = 1 to heapsize
    A[i] ← array[i]
  for i = parent(heapsize) down to 1
    ShiftDown(i)
```



0	1	2	3	4	5	6	7	8	9	10	11
0	2	7	26	25	36	17	1	90	3	19	

Animation (4)

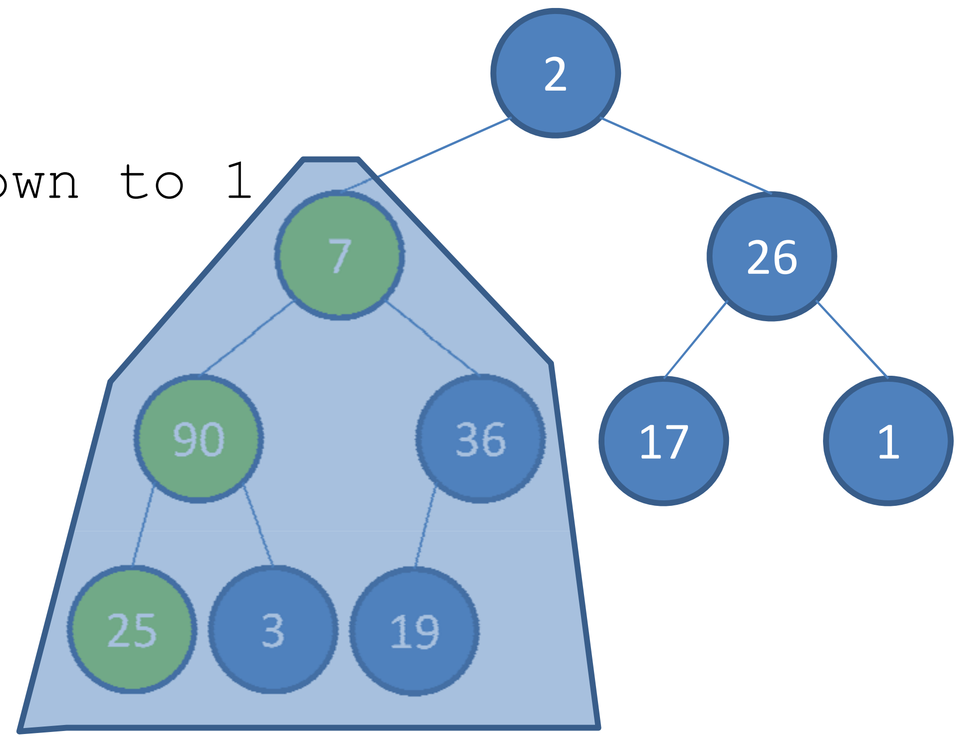
```
BuildHeap(array)
  heapsize ← size(array)
  A[0] ← 0
  for i = 1 to heapsize
    A[i] ← array[i]
  for i = parent(heapsize) down to 1
    ShiftDown(i)
```



0	1	2	3	4	5	6	7	8	9	10	11
0	2	7	26	90	36	17	1	25	3	19	

Animation (5)

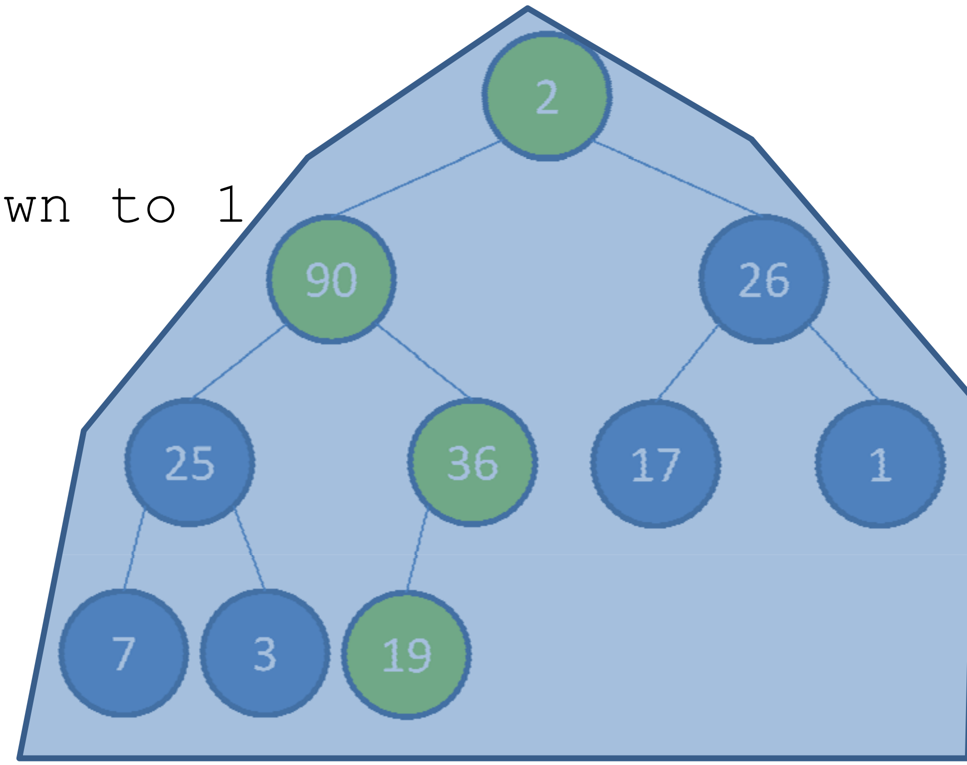
```
BuildHeap(array)
  heapsize ← size(array)
  A[0] ← 0
  for i = 1 to heapsize
    A[i] ← array[i]
  for i = parent(heapsize) down to 1
    ShiftDown(i)
```



0	1	2	3	4	5	6	7	8	9	10	11
0	2	7	26	90	36	17	1	25	3	19	

Animation (6)

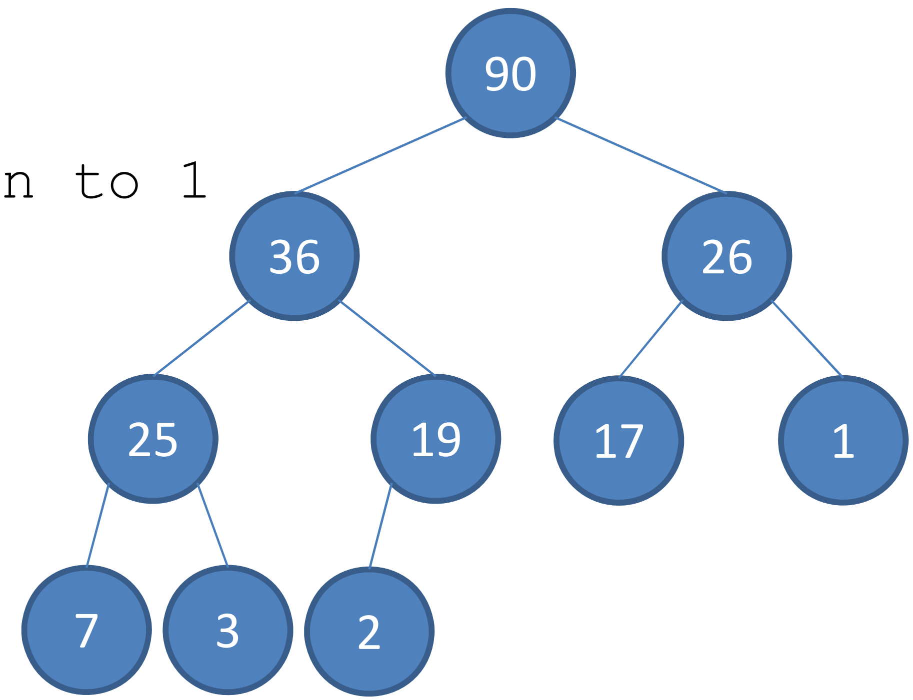
```
BuildHeap(array)
  heapsize ← size(array)
  A[0] ← 0
  for i = 1 to heapsize
    A[i] ← array[i]
  for i = parent(heapsize) down to 1
    ShiftDown(i)
```



0	1	2	3	4	5	6	7	8	9	10	11
0	2	90	26	25	36	17	1	7	3	19	

Animation (7)

```
BuildHeap(array)
  heapsize ← size(array)
  A[0] ← 0
  for i = 1 to heapsize
    A[i] ← array[i]
  for i = parent(heapsize) down to 1
    ShiftDown(i)
```



0	1	2	3	4	5	6	7	8	9	10	11
0	90	36	26	25	19	17	1	7	3	2	

`BuildHeap()` runs in O(N log N)?

1. Yes, obviously O(N log N)
2. No, it is ___________

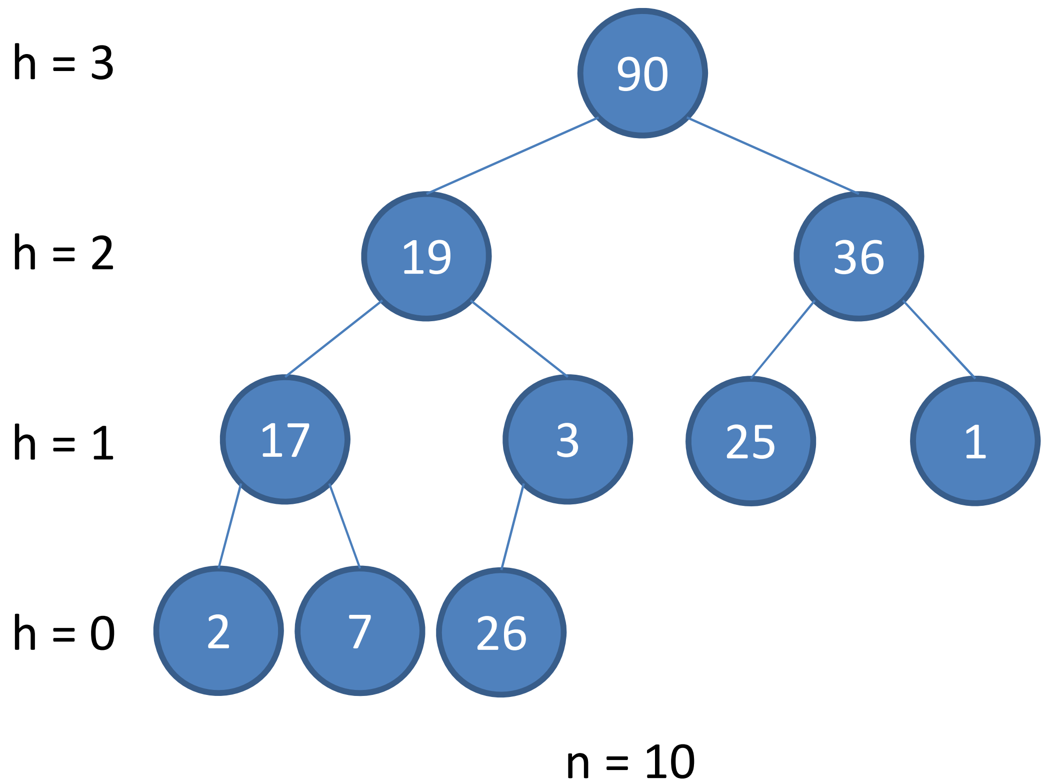
0 of 120

`BuildHeap()` Analysis… (1)

- Recall: How many levels (height) are there in a complete binary tree (heap) of size N? ___________
- Recall: What is the cost to run `shiftDown(i)`?

  ____
- Q: How many nodes are there at height **h** of a full binary tree?

  ____



`BuildHeap()` Analysis... (2)

- Cost of `BuildHeap()` is thus:

$$\underbrace{\sum_{h=0}^{\lfloor \lg(n) \rfloor}}_{\text{Sum over all levels}} \underbrace{\overbrace{\left\lceil \frac{n}{2^{h+1}} \right\rceil}^{\text{\# of nodes at height h}} \overbrace{O(h)}^{\text{Cost to Heapify a node at height h}}}_{\text{Cost for a level}} = \sum_{h=0}^{\lfloor \lg(n) \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil c*h = O\left(n \sum_{h=0}^{\lfloor \lg(n) \rfloor} \frac{h}{2^h}\right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(2n) = O(n)$$

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} = 2$$

$x = 1/2$

$$\frac{0}{2^0} + \frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \frac{4}{2^4} + \ldots = \frac{0}{1} + \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \ldots =$$

$$0 + 0.5 + 0.5 + 0.375 + 0.25 + 0.15625 + 0.09375 + \ldots \mathbf{< 2}$$

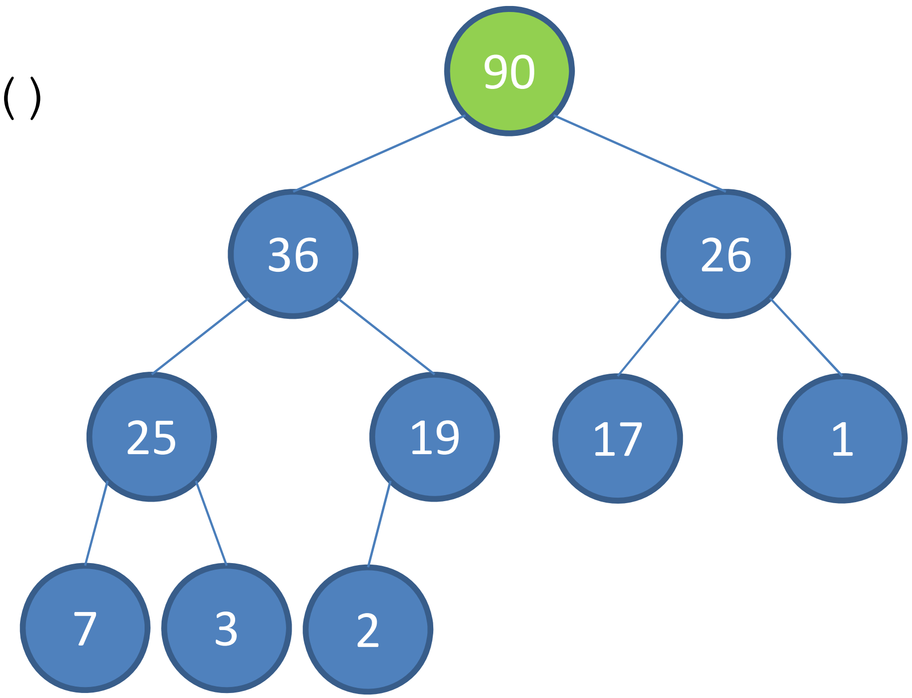
HeapSort Analysis

```
HeapSort(array)
  BuildHeap(array) // The best we can do is _______
  N ← size(array)
  for i from 1 to N // O(N)
    A[N - i + 1] ← ExtractMax() // O(log N)
  return A

// Analysis: Thus HeapSort runs in O(____________)
// Do you notice that we do not need extra array
// like merge sort to perform sorting?
// Thus heap sort is more memory friendly.
// This is called "in-place sorting"
// But HeapSort is not "cache friendly"
```

Animation (1)

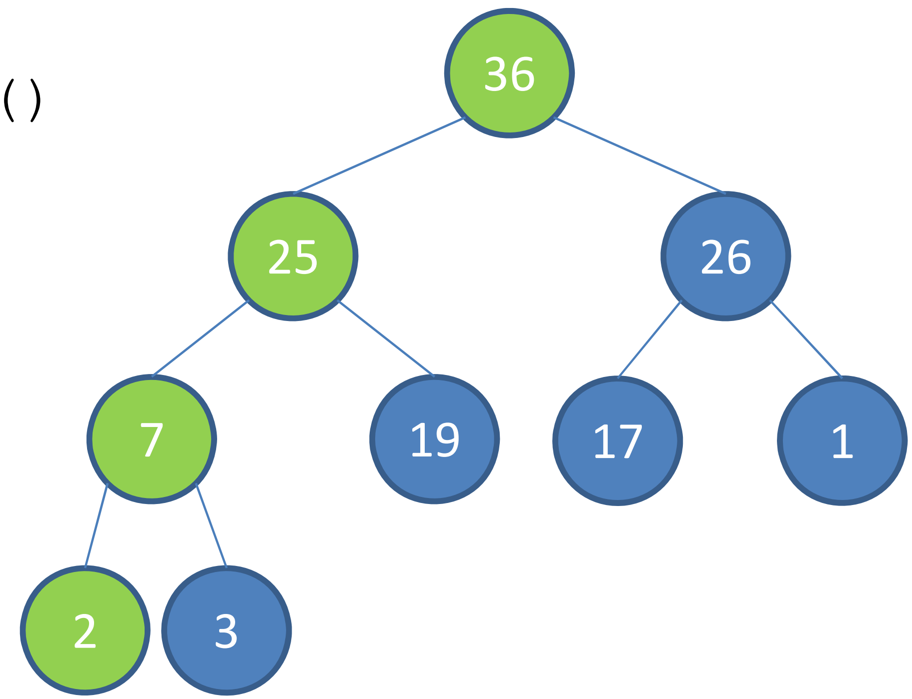
```
HeapSort(array)
  BuildHeap(array)
  N ← size(array)
  for i from 1 to N
    A[N - i + 1] ← ExtractMax()
  return A
```



0	1	2	3	4	5	6	7	8	9	[10]	11
0	90	36	26	25	19	17	1	7	3	2	

Animation (2)

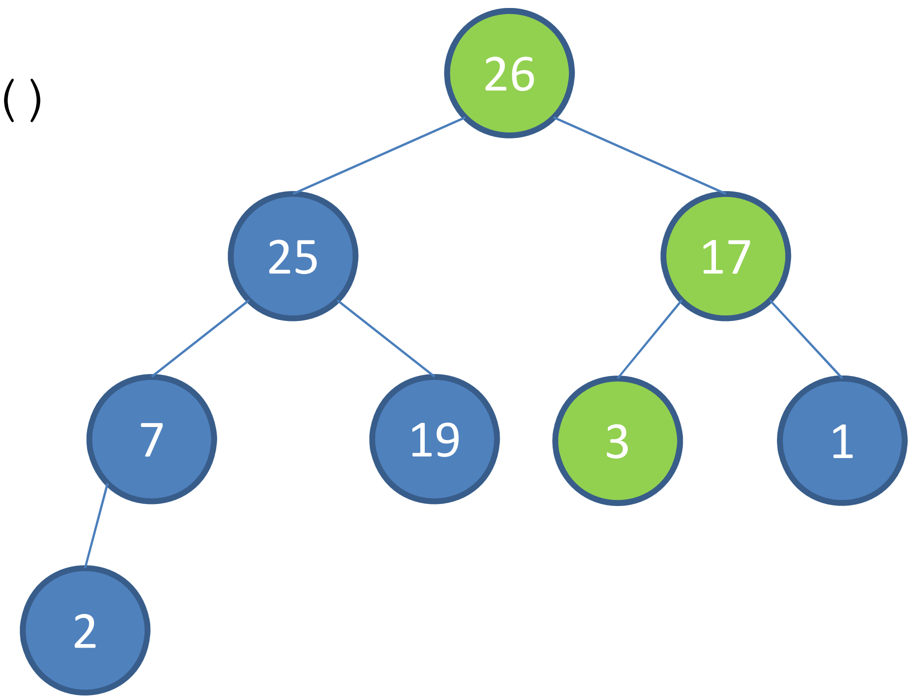
```
HeapSort(array)
  BuildHeap(array)
  N ← size(array)
  for i from 1 to N
    A[N - i + 1] ← ExtractMax()
  return A
```



0	1	2	3	4	5	6	7	8	[9]	10	11
0	36	25	26	7	19	17	1	2	3	90	

Animation (3)

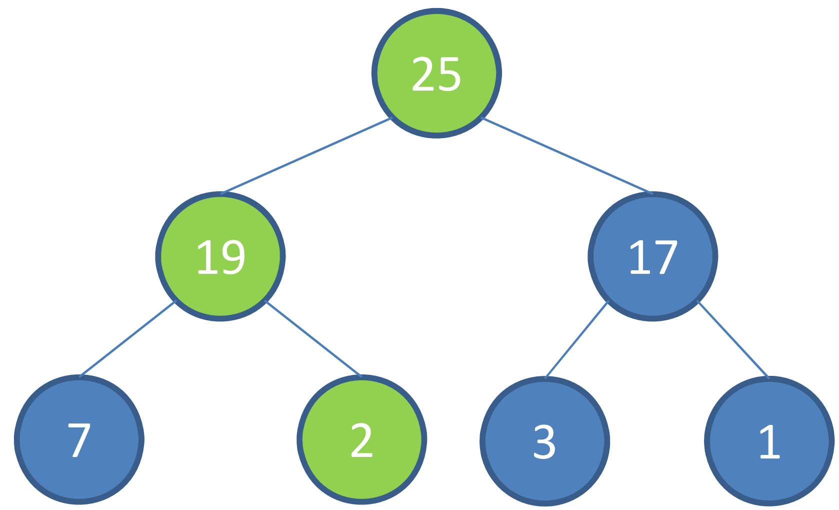
```
HeapSort(array)
  BuildHeap(array)
  N ← size(array)
  for i from 1 to N
    A[N − i + 1] ← ExtractMax()
  return A
```



0	1	2	3	4	5	6	7	[8]	9	10	11
0	26	25	17	7	19	3	1	2	36	90	

Animation (4)

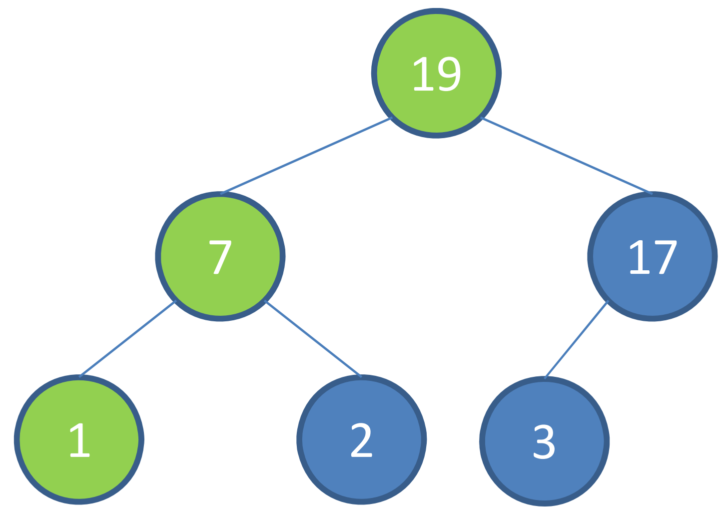
```
HeapSort(array)
  BuildHeap(array)
  N ← size(array)
  for i from 1 to N
    A[N – i + 1] ← ExtractMax()
  return A
```



0	1	2	3	4	5	6	[7]	8	9	10	11
0	25	19	17	7	2	3	1	26	36	90	

Animation (5)

```
HeapSort(array)
  BuildHeap(array)
  N ← size(array)
  for i from 1 to N
    A[N – i + 1] ← ExtractMax()
  return A
```



And so on until A[1..9] are sorted

0	1	2	3	4	5	[6]	7	8	9	10	11
0	19	7	17	1	2	3	25	26	36	90	

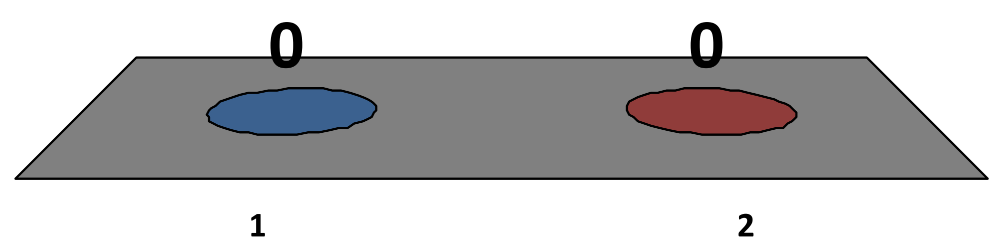
Java Implementation

- Priority Queue ADT
- Heap Class (Java file given, you can use it for PS2)
  - ShiftUp
  - Insert(v)
  - ShiftDown
  - ExtractMax
  - BuildHeapSlow(array) and BuildHeap(array)
  - HeapSort
- In OOP Style ☺

Pop Quiz (not in your copy):

Is a sorted array (descending) a Max Heap?

1. Yes
2. No

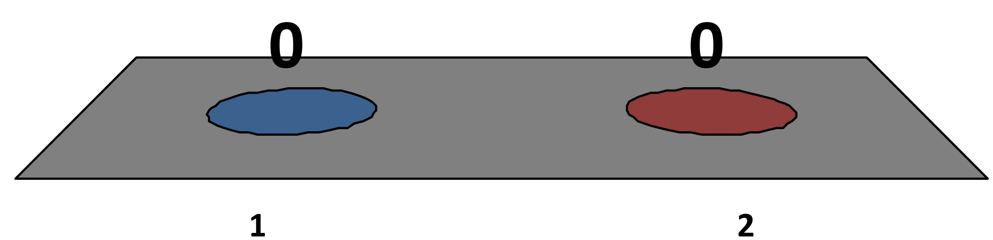


0 of 120

Pop Quiz (not in your copy):

Running ShiftDown(i) for i > heapsize/2 will

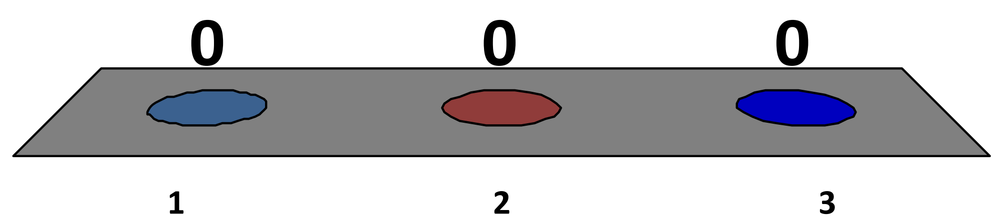
1. Have no effect to the Max Heap
2. Possibly change some items in the Max Heap



0 of 120

Quick Feedback

1. The Heap DS visualization is neutral
2. The Heap DS visualization is cool
3. Hey, I have tried the other graph visualizations too, they are also cool



0 of 120

Summary

- In this lecture, we have looked at:
  - Heap DS and its application for PriorityQueue
  - Storing heap as a compact array and its operations
    - Remember how we always try to maintain complete binary tree and heap property in all our operations!!!
  - Building a heap from a set of numbers in O(n) time
  - Simple application of Heap DS: HeapSort
- We will use BST/Heap again in the 2nd part of CS2010
- Play around with this max heap visualization to help strengthen your understanding of this DS
  - http://www.comp.nus.edu.sg/~stevenha/visualization/heap.html

Scheduling Deliveries Problem (PS2)

- This happens in the delivery suite (or surgery room for Caesarean section) of a hospital

PS2, the task

- Given a list of pregnant women,
  prioritize the ones who will give birth sooner over the one who will give birth later…
- Will be uploaded on Thursday, 06 Sep 2012
- Involving Priority Queue ☺

Help Session

- Current plan so far:
  - Saturday, 8 September 2012, 12.30-2.00pm
    Topic: BST + Balanced BST + Heap
  - Venue: NUS Business canteen
  - Who can attend: Preferably those who are struggling with this module so far