CS2020 – Data Structures and Algorithms Accelerated

Lecture 06 – Heaps of Fun

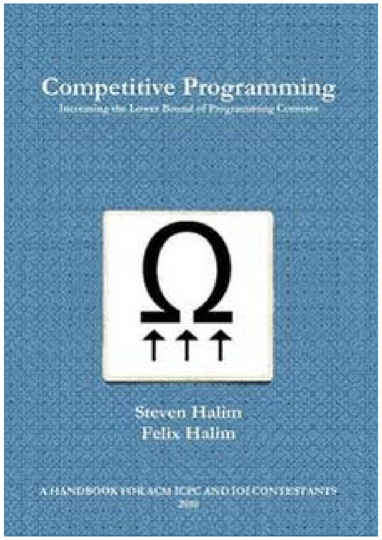
stevenhalim@gmail.com

National University of Singapore
School of Computing

About Me (1)

- The 2nd lecturer of CS2020:
  - Dr Steven Halim
    - Call me as: Steven
- Website:
  - http://www.comp.nus.edu.sg/~stevenha
- How to reach me:
  - Email: stevenhalim@gmail.com
    (+ Facebook ☺)
  - My office: COM2-03-37
  - Office Number: 6516-7361

About Me (2)



- I also teach:
  - CS3233 – Competitive Programming
- I co-author^ “Competitive Programming” book
  - http://www.lulu.com/product/paperback/competitive-programming/12110025
  - https://sites.google.com/site/stevenhalim
  - ~15 copies are available at 20 SGD/copy
- I am the coach for:
  - NUS ACM ICPC teams
    - International Collegiate Programming Contest
  - Singapore IOI team*
    - International Olympiad in Informatics

Special Note

- Steven will be lecturing mostly during the 2nd half of CS2020
- This is a "one off" lecture to cover a data structure that will be used again during the 2nd half of the class

Outline

- What are you going to learn in this lecture?
  - Motivation: Abstract Data Type: **PriorityQueue**
  - **Heap** data structure
  - **Heap sort**

Abstract Data Type: PriorityQueue

- Important Basic Operations:
  - Enqueue(x)
    - Put a new item x in the priority queue PQ (in some order)
  - y ← 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 is…
  - A particular way of **storing** and **organizing data** in a computer so that it can be used efficiently
- Most data structure 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 place**, 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**

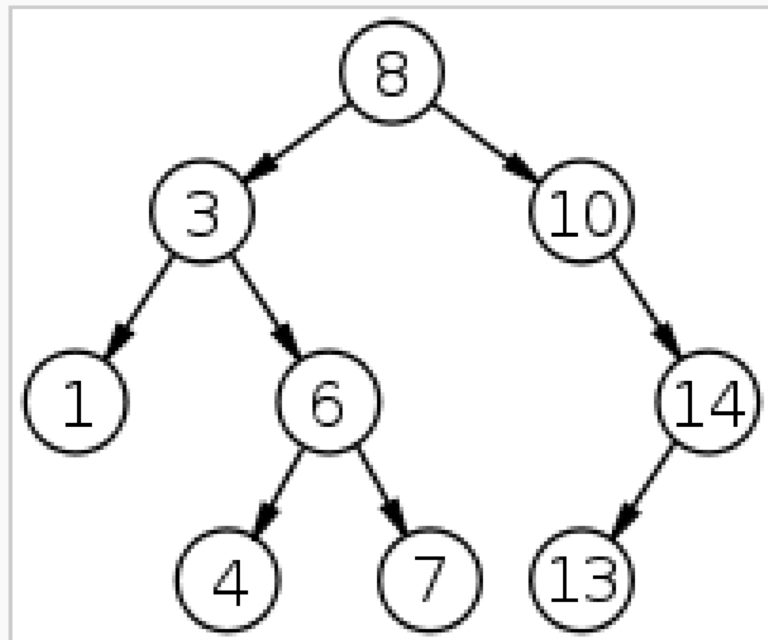
PriorityQueue Implementation (3)

Strategy	Enqueue	Dequeue
Array-Based PQ (1)	O(N)	O(1)
Array-Based PQ (2)	O(1)	O(N)
We can do better!	O(?)	O(?)

INTRODUCING HEAP DATA STRUCTURE

Quick Review

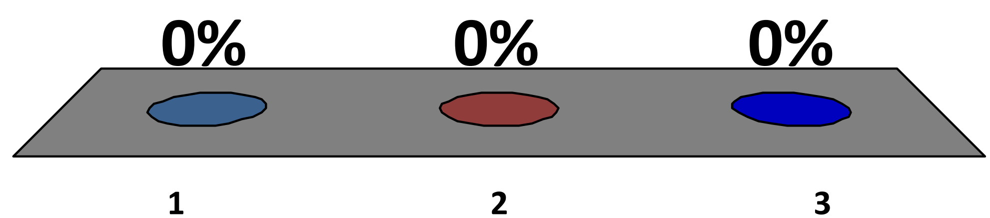
- Heap is similar to what you already know: *Binary Search Tree (BST, from previous two lectures)*
  - Vertex/Node/Item
  - Edge
  - Root
  - Internal Nodes
  - Leaves
  - Binary Tree
  - Left/Right Sub-Tree
  - The **BST Property**…



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

The BST Property Is...

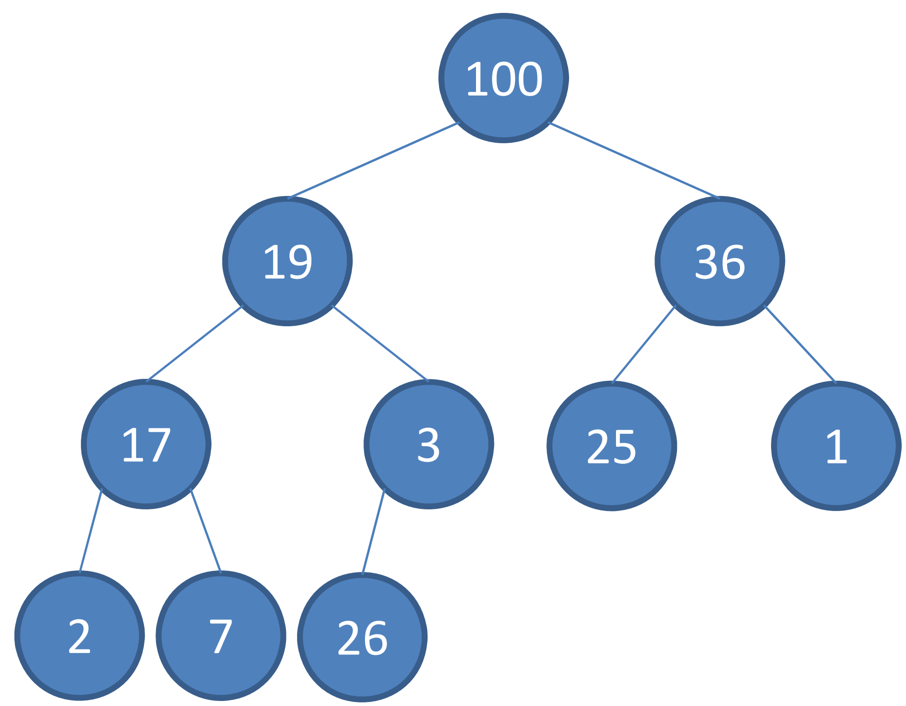
1. key[x] < key[left[x]] < key[right[x]]
2. key[left[x]] < key[x] < key[right[x]]
3. key[right[x]] < key[left[x]] < key[x]



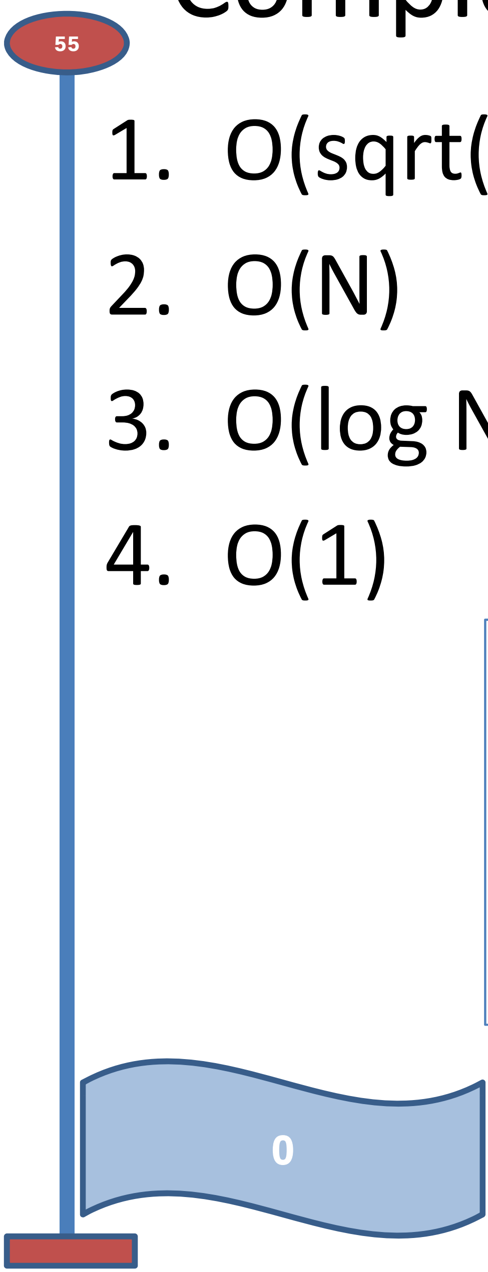
0 of 55

Complete Binary Tree

- Introducing 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**?

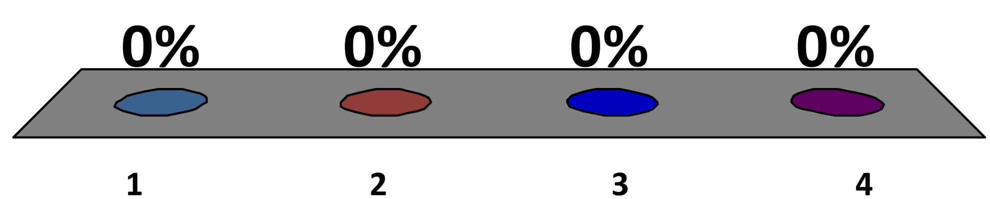


The Height of a Complete Binary Tree of N Items is...



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

Now, memorize this answer, we will need that for all the time complexity analysis of heap operations



Storing Complete Binary Tree

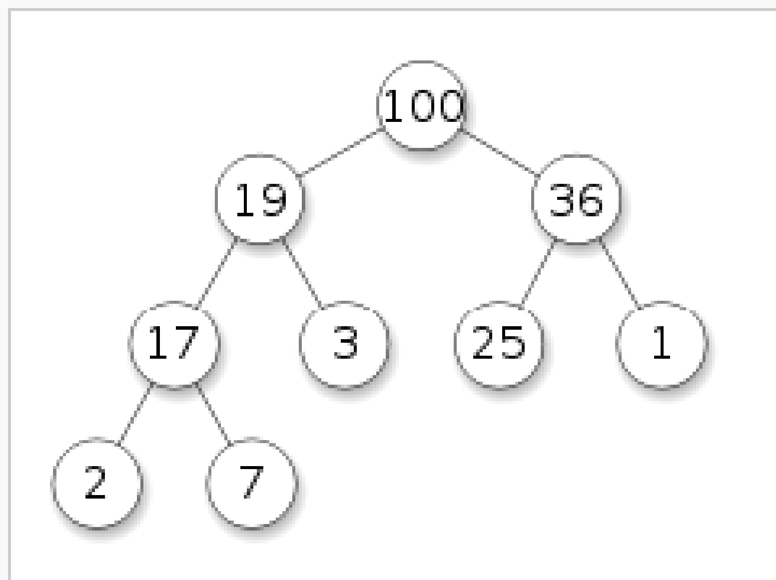
- As an 1-based compact array: `A[1..size(A)]`

size(A)

0	1	2	3	4	5	6	7	8	9	10	11
NIL	100	19	36	17	3	25	1	2	7	-	-

heapsize ≤ size(A)

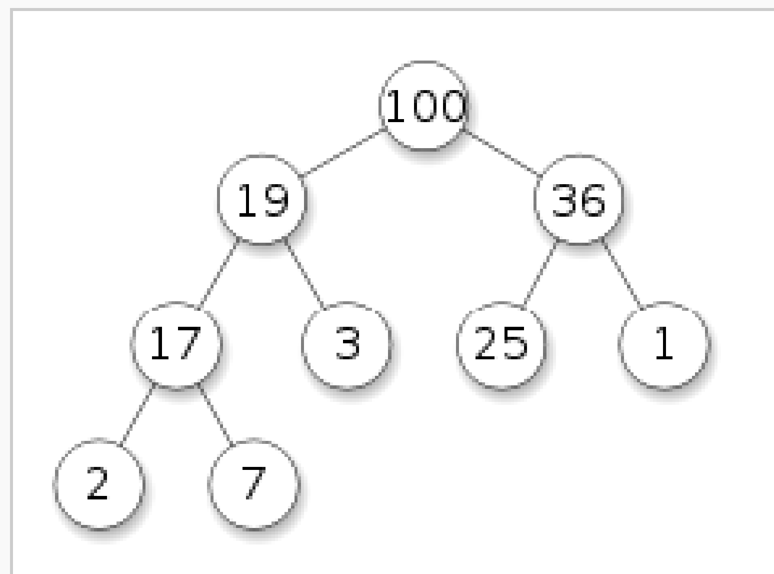
- Navigation operations:
  - `Parent(i) = floor(i/2)`
    - Except for `i = 1`
  - `Left(i) = 2*i`
  - `Right(i) = 2*i + 1`
    - No left/right child when:
    - `Left(i) > heapsize`
    - `Right(i) > heapsize`



Example of a complete binary max-heap

The Heap Property

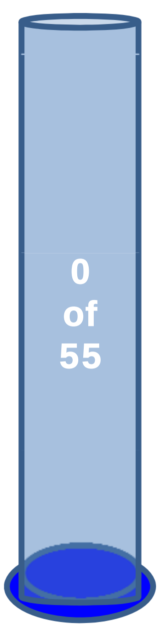
- The **Heap property**
  - `A[parent(i)] ≥ A[x]` **(max heap)**
  - `A[parent(i)] ≤ A[x]` **(min heap)**
- Without loss of generality,
  I will use "**max heap**"
  for all examples
  in this lecture

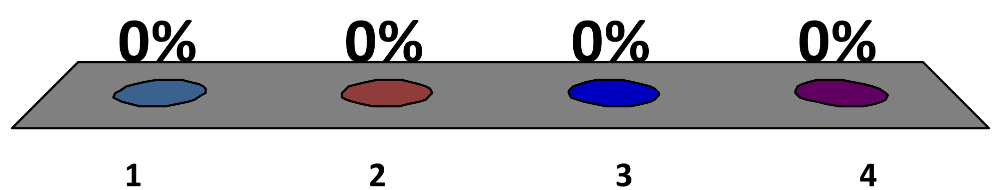


Example of a complete binary max-heap

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

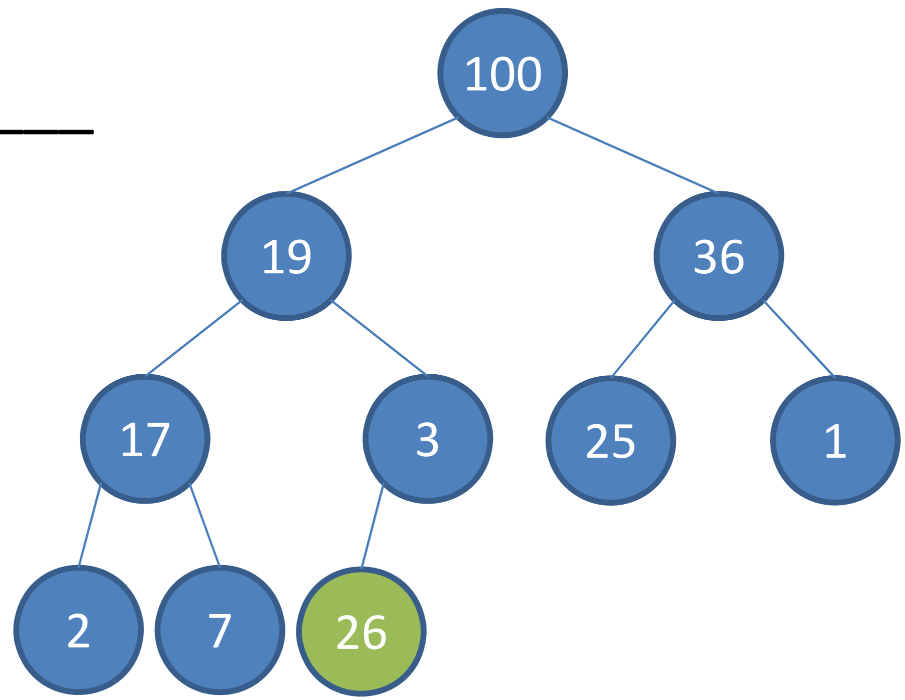
1. One of the leaves
2. One of the internal nodes
3. Can be anywhere in the heap
4. Must be at the root





Insertion to Existing Heap

- The most appropriate insertion point to 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	100	19	36	17	3	25	1	2	7		

Heap_Insert – Pseudo Code

```
Heap_Insert(key)
  heapsize = heapsize + 1; // extend O(1)
  A[heapsize] = key // insert at the back O(1)
  shiftUp(heapsize) // fix the heap property
                    // in O(?)

// Preliminary analysis:
// Heap_Insert(A, key) depends on shiftUp(A, i)
```

shiftUp – Pseudo Code

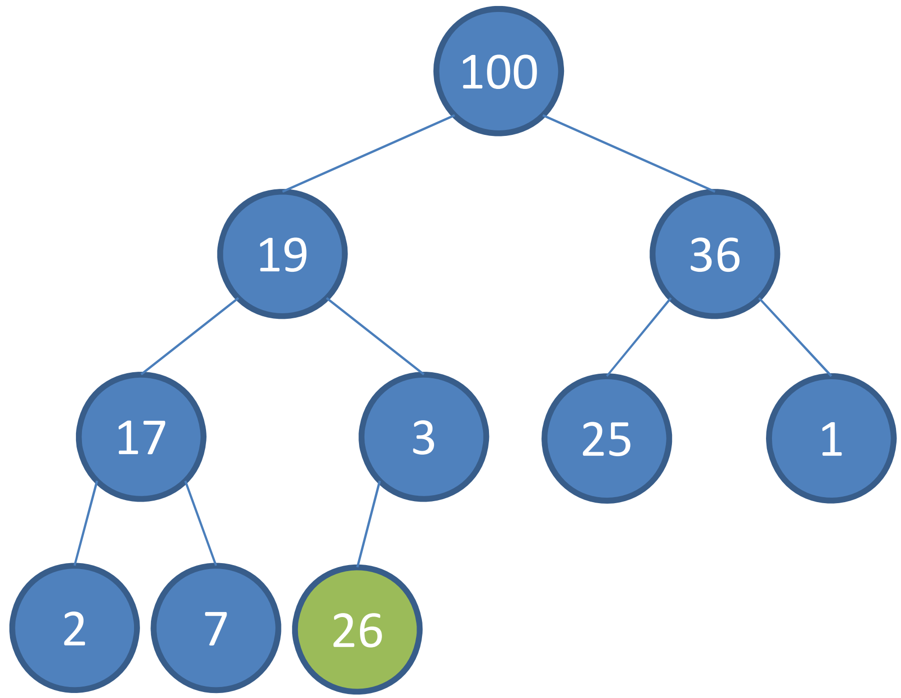
- Name is not unique: shiftUp/bubbleUp/HeapIncreaseKey/etc

“not root” “violates max heap property”

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

Animation (1)

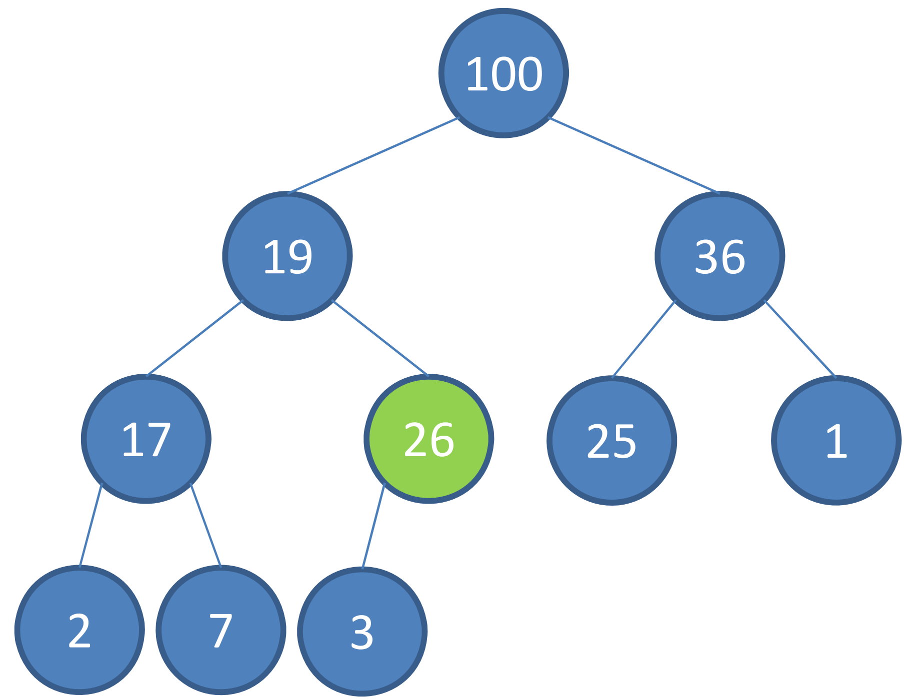
```
shiftUp(A, i)
  while i > 1 and A[parent(i)] < A[i]
    swap(A[i], A[parent(i)])
    i = parent(i)
```



0	1	2	3	4	5	6	7	8	9	10	11
0	100	19	36	17	3	25	1	2	7	26	

Animation (2)

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

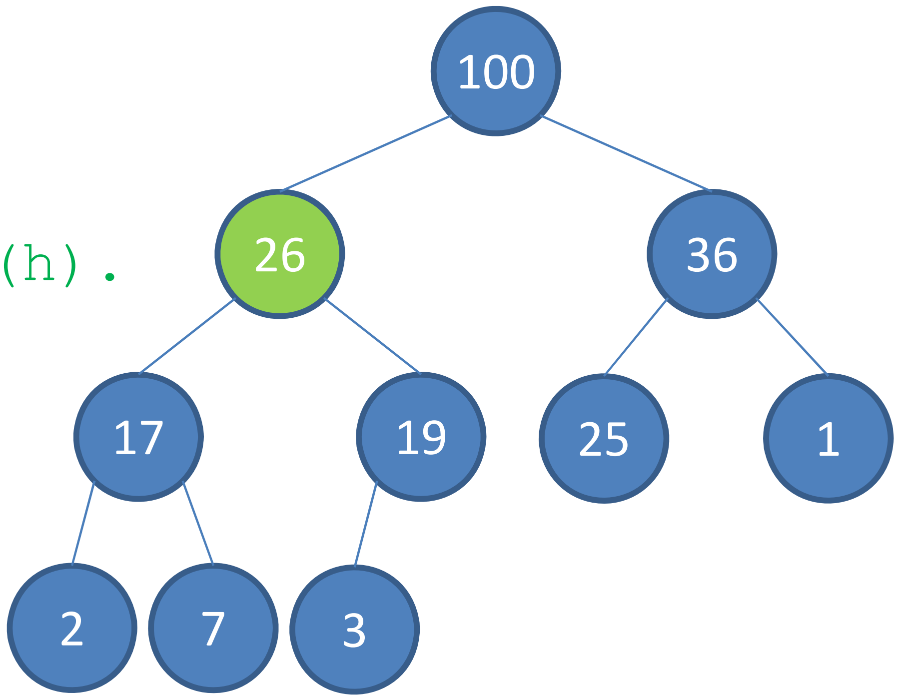


0	1	2	3	4	5	6	7	8	9	10	11
0	100	19	36	17	26	25	1	2	7	3	

Animation (3)

```
shiftUp(A, 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 deepest leaf to root O(h).
// In a complete binary tree,
// this h is just log N.
// Thus, shiftUp AND
// Heap_Insert runs in
// O(log N)
```



0	1	2	3	4	5	6	7	8	9	10	11
0	100	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 don't 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)`

Heap_ExtractMax - Pseudocode

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

// Preliminary analysis:
// Heap_ExtractMax() depends on shiftDown()
```

ShiftDown – Pseudo Code

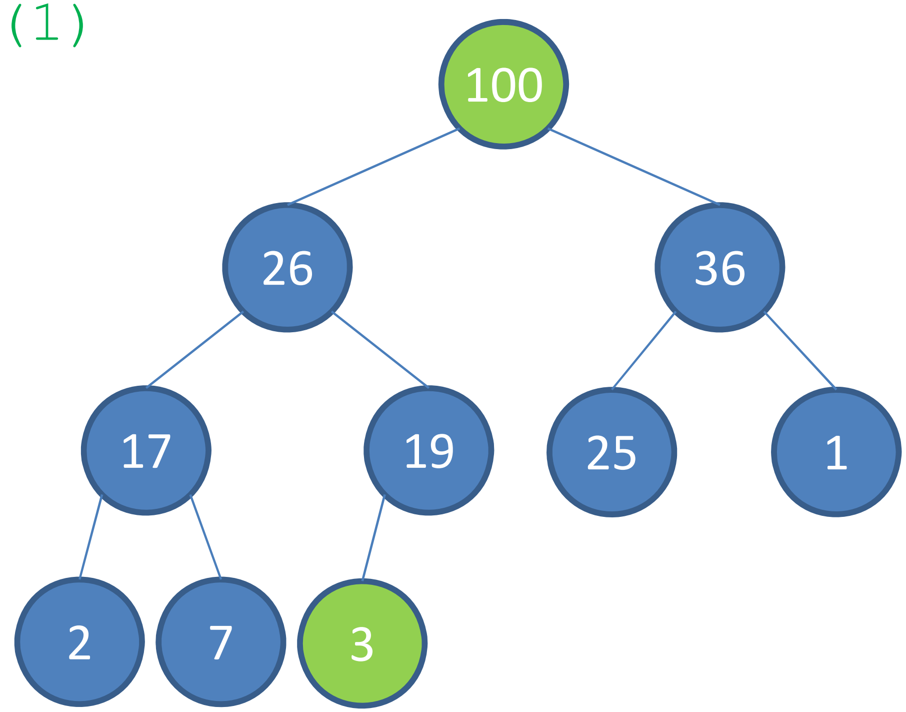
Again, name is not unique:
shiftDown/bubbleDown/Heapify/etc

```
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)

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

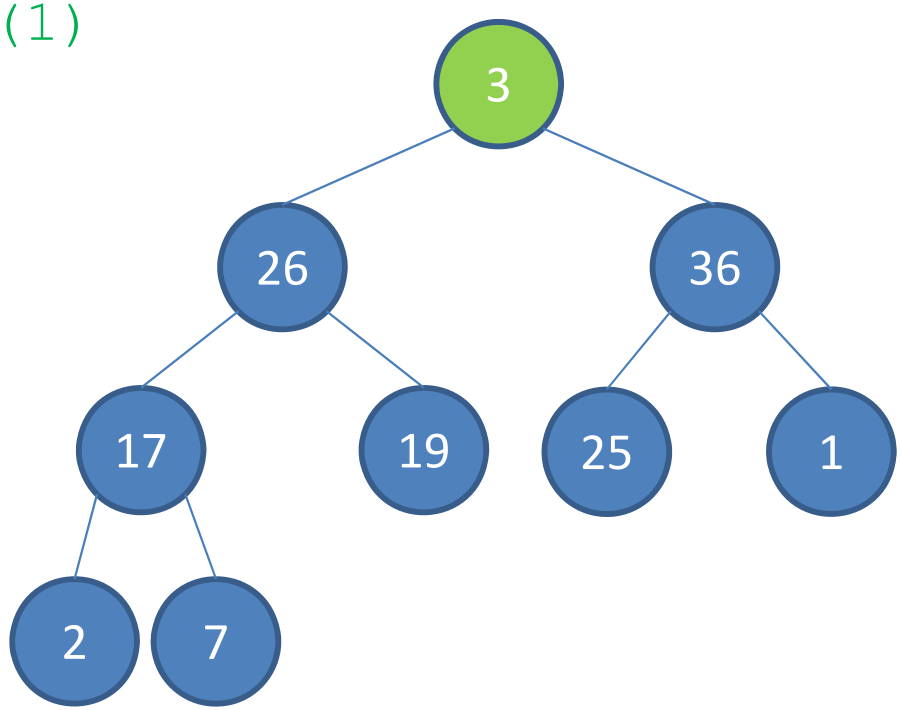


0	1	2	3	4	5	6	7	8	9	[10]	11
0	100	26	36	17	19	25	1	2	7	3	

Animation (2)

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

100 is stored at maxV and returned later after shiftDown(1) is done

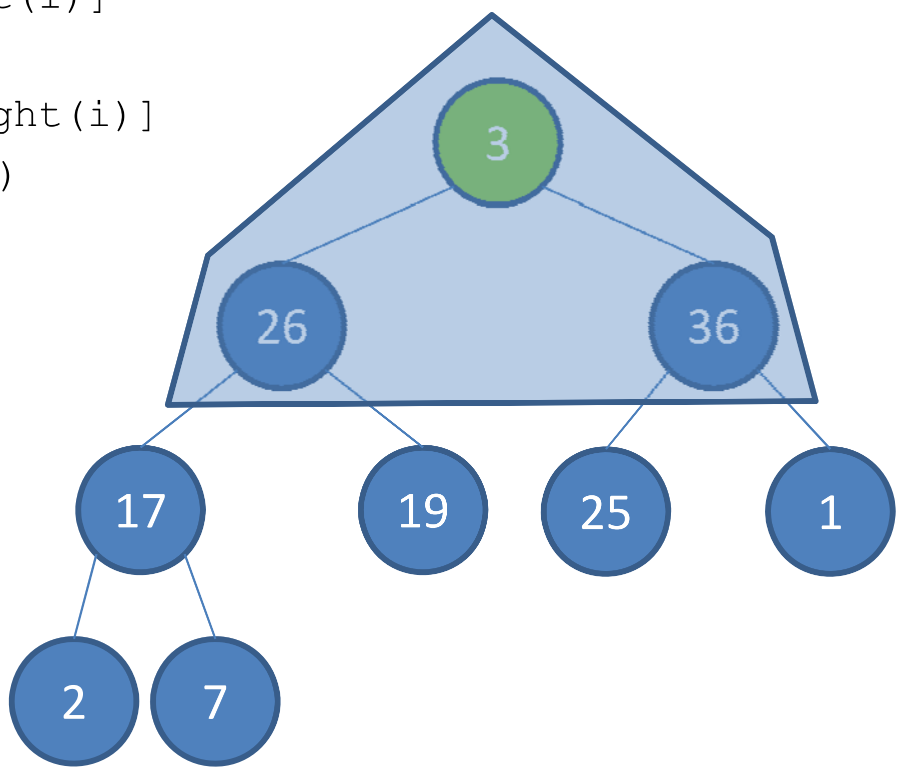


0	1	2	3	4	5	6	7	8	[9]	10	11
0	3	26	36	17	19	25	1	2	7		

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;
```

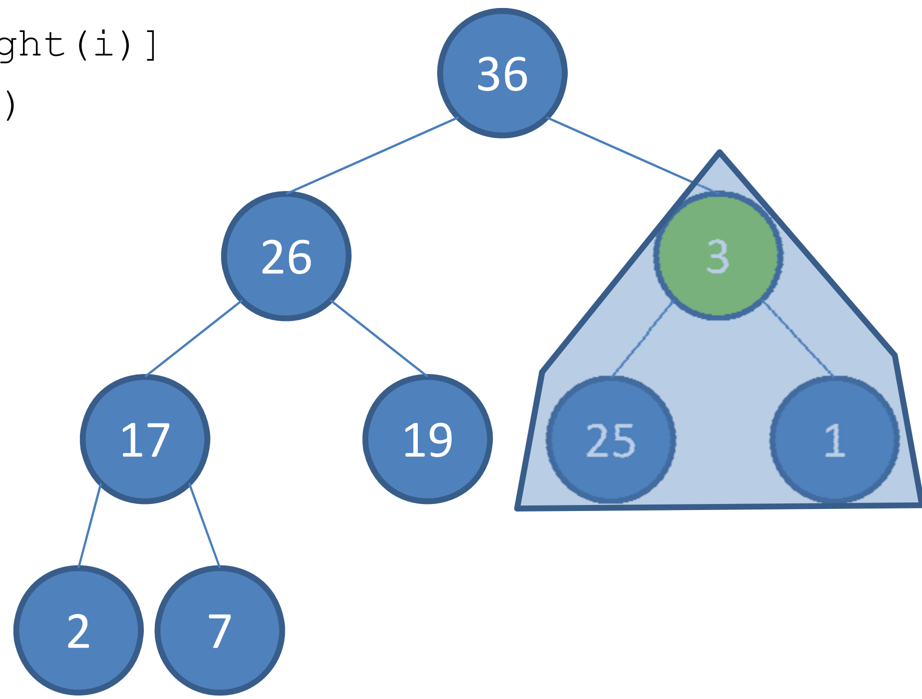


0	1	2	3	4	5	6	7	8	[9]	10	11
0	3	26	36	17	19	25	1	2	7		

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;
```



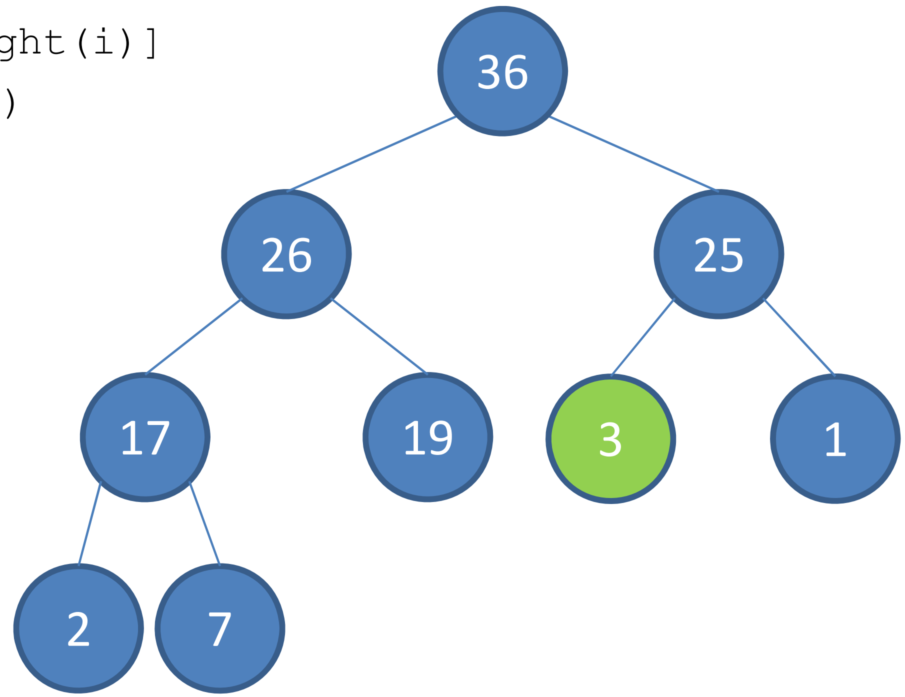
0	1	2	3	4	5	6	7	8	[9]	10	11
0	36	26	3	17	19	25	1	2	7		

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
// Heap_ExtractMax runs in
// O(h) = O(log N) time
```



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)

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

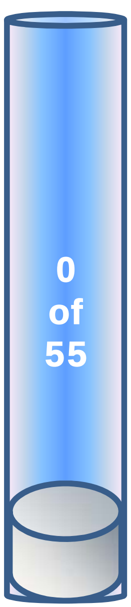
Next Items:

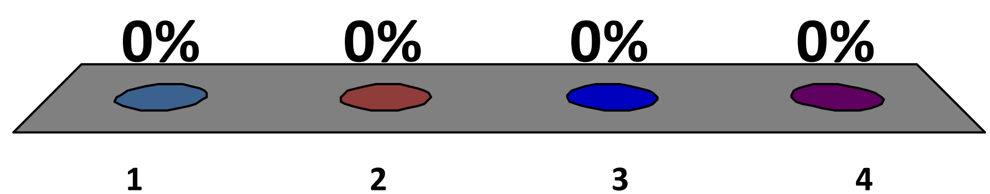
- Heap Sort
- Building Max Heap from an ordinary Array
- Java Implementation of Max Heap

10 MINUTES BREAK

Review: We have seen MergeSort in Lect2. It can sort N items in…

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





Heap_Sort Pseudo Code

- With a max heap, we can do sorting too ☺
  - Just call Heap_ExtractMax N times
  - If we don't have a max heap yet, simply build one!

```
Heap_Sort(Array)
  Build_Heap(Array) // O(?)
  N ← size(Array)
  for i from 1 to N // O(N)
    A[N - i + 1] ← Heap_ExtractMax() // O(log N)
  return A

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

Build_Heap (Version 1)

```
Build_Heap(Array)
  N ← size(Array)
  A[0] ← 0 // dummy entry
  for i = 1 to N // O(N)
    Heap_Insert(Array[i]) // O(log N)

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

- Can we do better?

Build_Heap v1, can we do better?

1. Yes, you must have some more trick
2. No, this is already good enough

0 of 55

Build_Heap (Version 2)

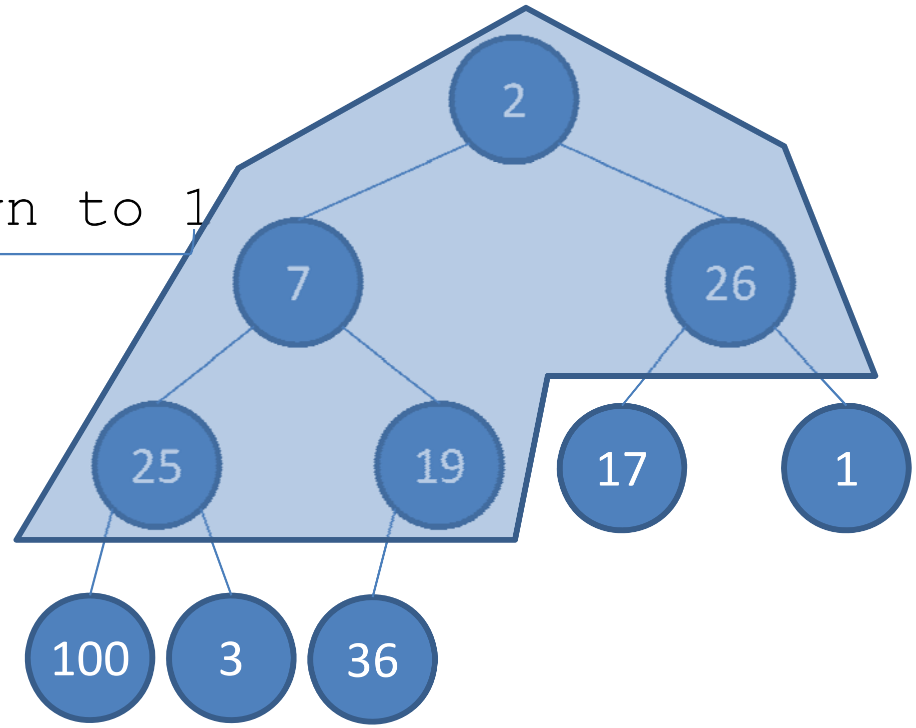
```
Build_Heap(Array)
  heapsize ← size(Array)
  A[0] ← 0 // dummy entry
  for i = 1 to heapsize // copy the content O(N)
    A[i] ← Array[i]
  for i = Parent(heapsize) down to 1 // O(N/2)
    shiftDown(i) // O(log N)

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

Animation (1)

```
Build_Heap(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)
```

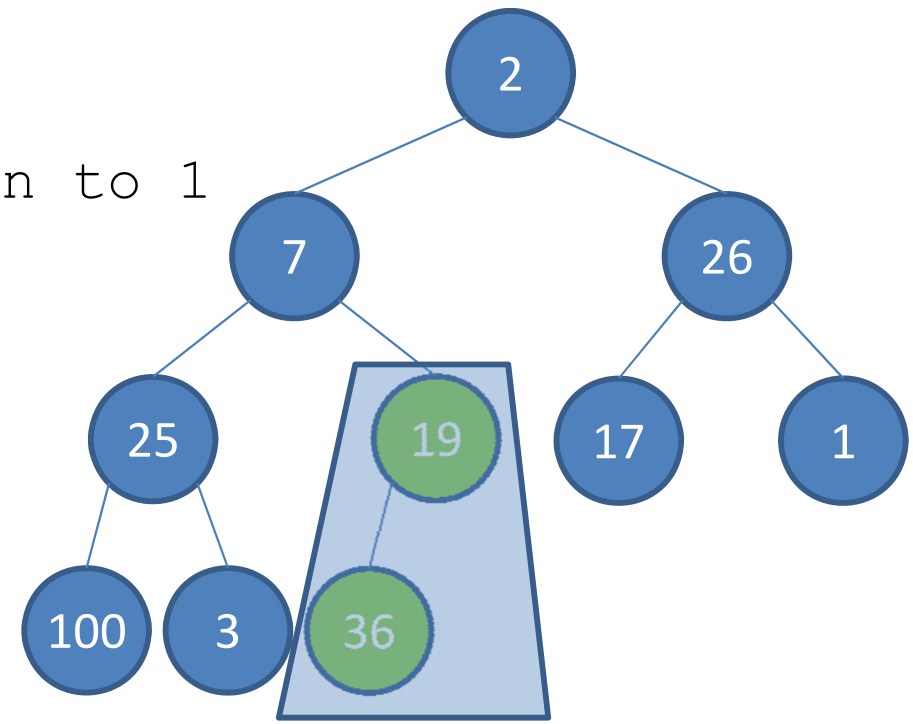
Internal Nodes Only!



0	1	2	3	4	5	6	7	8	9	10	11
0	2	7	26	25	19	17	1	100	3	36	

Animation (2)

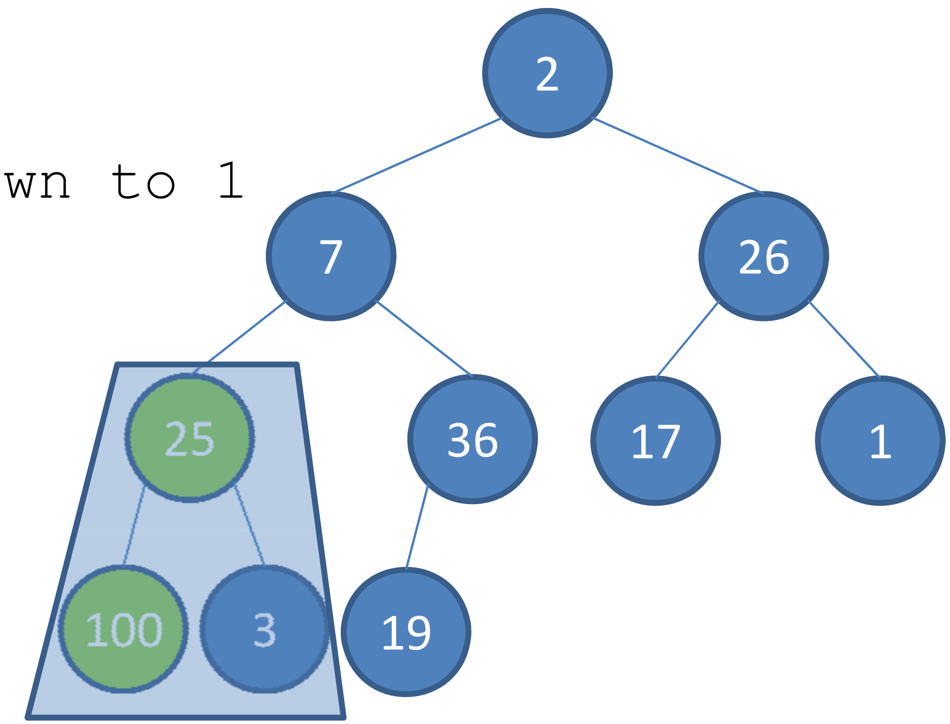
```
Build_Heap(Array)
  heapsize ← size(Array)
  A[0] ← 0
  for i = 1 to heapsize
    A[i] ← Array[i]
  for i = Parent(heapsize) down to 1
    shiftDown(i)
```



0	1	2	3	4	5	6	7	8	9	10	11
0	2	7	26	25	19	17	1	100	3	36	

Animation (3)

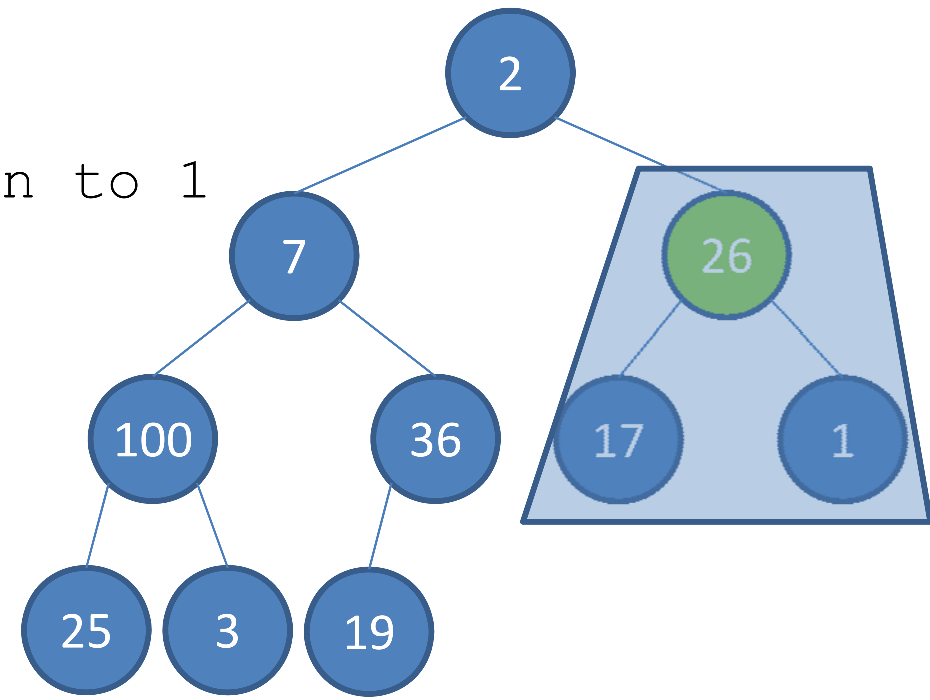
```
Build_Heap(Array)
  heapsize ← size(Array)
  A[0] ← 0
  for i = 1 to heapsize
    A[i] ← Array[i]
  for i = Parent(heapsize) down to 1
    shiftDown(i)
```



0	1	2	3	4	5	6	7	8	9	10	11
0	2	7	26	25	36	17	1	100	3	19	

Animation (4)

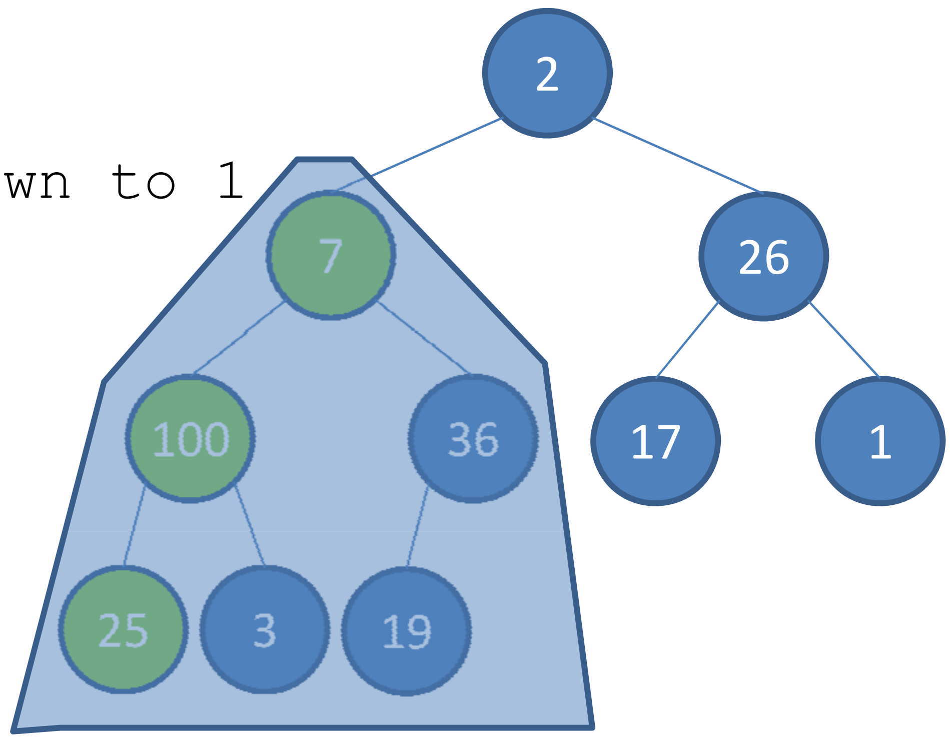
```
Build_Heap(Array)
  heapsize ← size(Array)
  A[0] ← 0
  for i = 1 to heapsize
    A[i] ← Array[i]
  for i = Parent(heapsize) down to 1
    shiftDown(i)
```



0	1	2	3	4	5	6	7	8	9	10	11
0	2	7	26	100	36	17	1	25	3	19	

Animation (5)

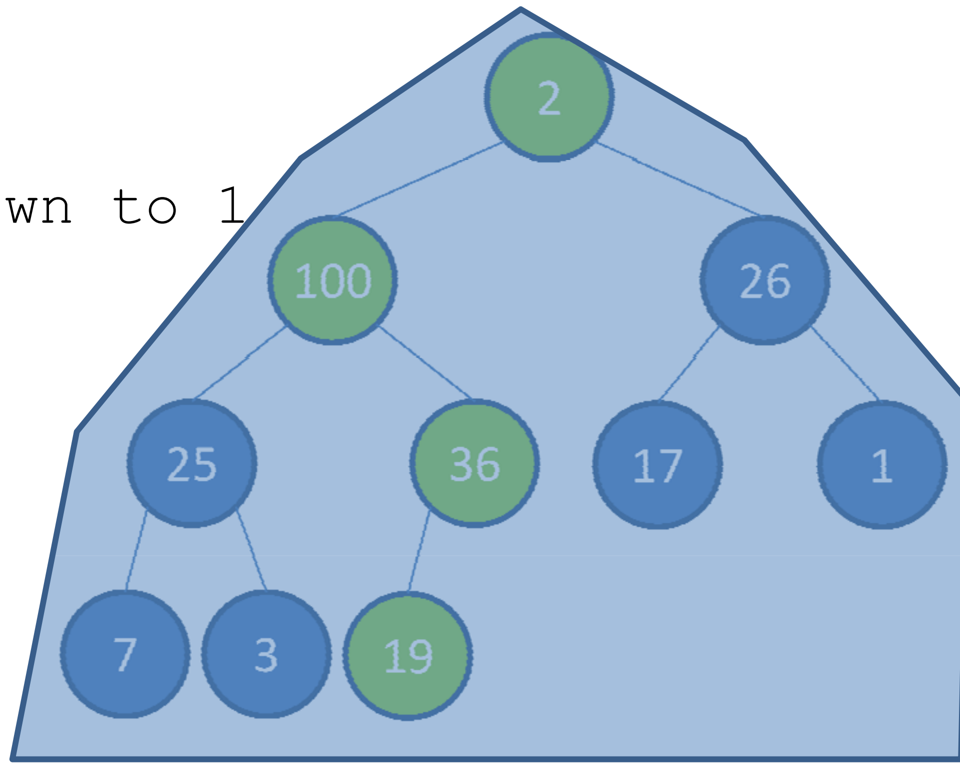
```
Build_Heap(Array)
  heapsize ← size(Array)
  A[0] ← 0
  for i = 1 to heapsize
    A[i] ← Array[i]
  for i = Parent(heapsize) down to 1
    shiftDown(i)
```



0	1	2	3	4	5	6	7	8	9	10	11
0	2	7	26	100	36	17	1	25	3	19	

Animation (6)

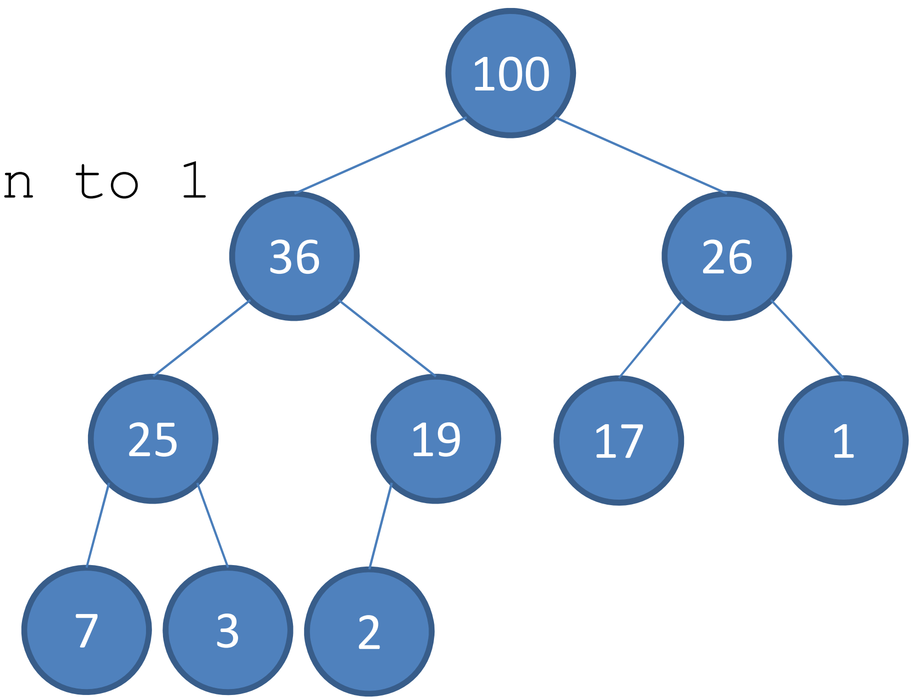
```
Build_Heap(Array)
  heapsize ← size(Array)
  A[0] ← 0
  for i = 1 to heapsize
    A[i] ← Array[i]
  for i = Parent(heapsize) down to 1
    shiftDown(i)
```



0	1	2	3	4	5	6	7	8	9	10	11
0	2	100	26	25	36	17	1	7	3	19	

Animation (7)

```
Build_Heap(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	100	36	26	25	19	17	1	7	3	2	

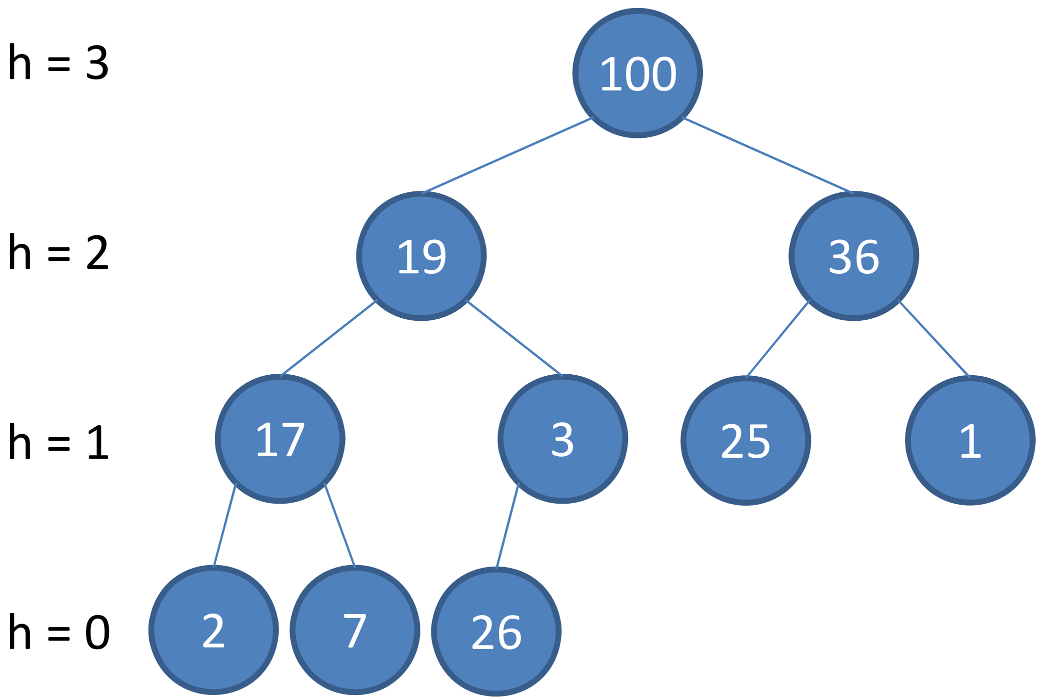
Build-Heap v2 runs in O(N log N)?

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

Build-Heap v2 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)?

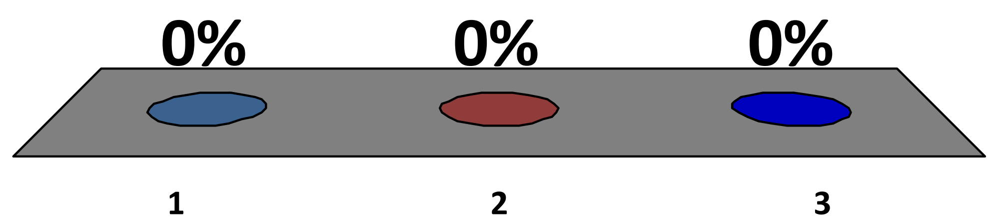
  ____
- How many nodes are there at height **h** of a complete binary tree?



Number of CBT nodes at height **h?**

1. ceil(n / (h + 1))
2. floor(2^(h + 1))
3. ceil(n / 2^(h + 1))

0 of 55



Build-Heap v2 Analysis… (2)

- Cost of Build-Heap v2 is thus:

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

Heap-Sort Analysis

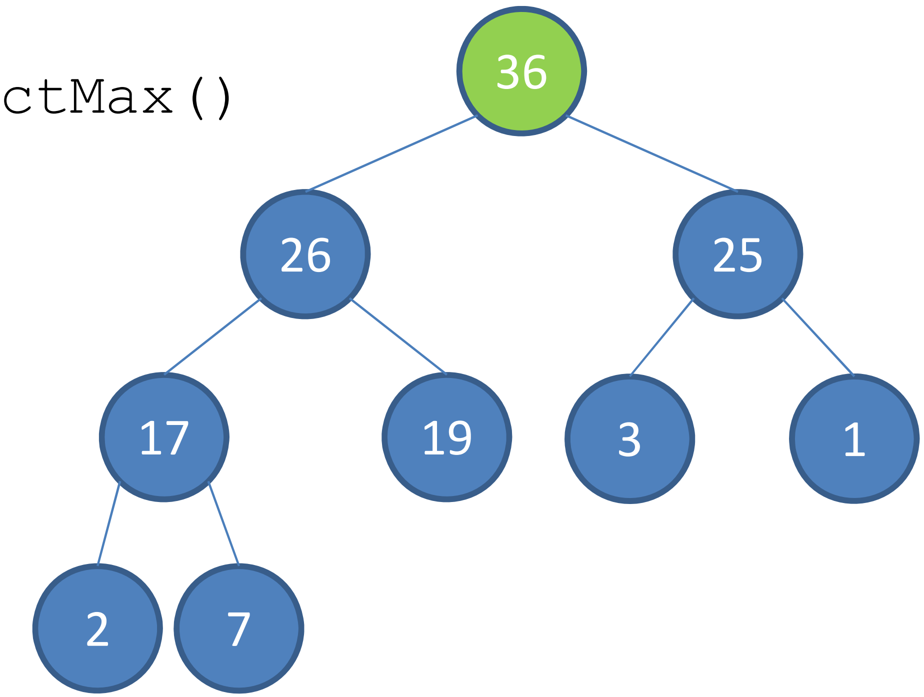
```
Heap_Sort(Array)
  Build_Heap(Array) // The best we can do is _______
  N ← size(Array)
  for i from 1 to N // O(N)
    A[N - i + 1] ← Heap_ExtractMax() // O(log N)
  return A

// Analysis: Thus Heap_Sort 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"
```

Animation (1)

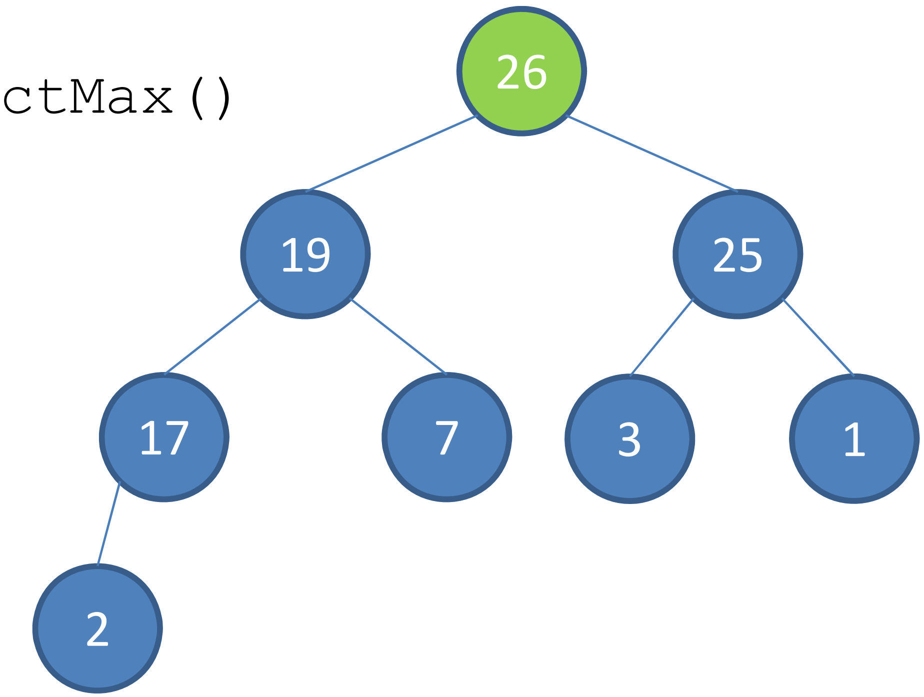
```
Heap_Sort(Array)
  Build_Heap(Array)
  N ← size(Array)
  for i from 1 to N
    A[N − i + 1] ← Heap_ExtractMax()
  return A
```



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

Animation (2)

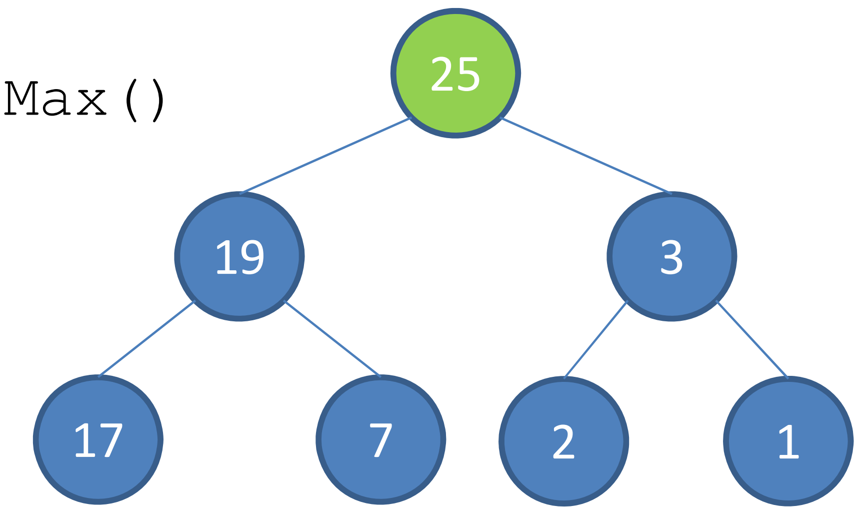
```
Heap_Sort(Array)
  Build_Heap(Array)
  N ← size(Array)
  for i from 1 to N
    A[N – i + 1] ← Heap_ExtractMax()
  return A
```



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

Animation (3)

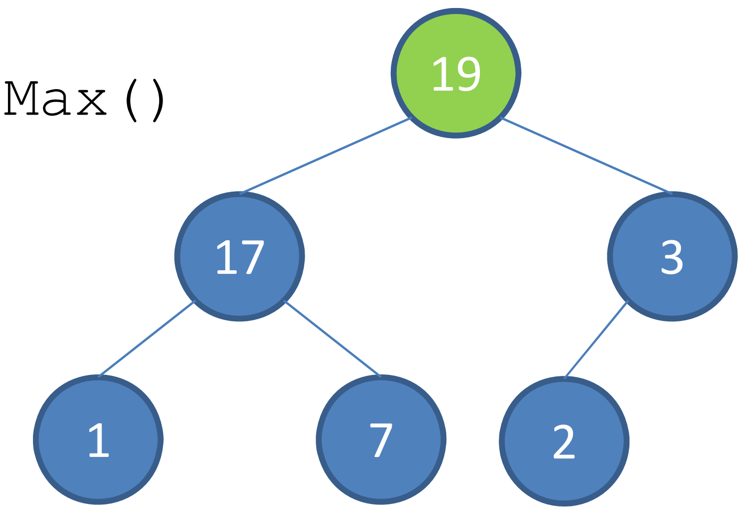
```
Heap_Sort(Array)
  Build_Heap(Array)
  N ← size(Array)
  for i from 1 to N
    A[N – i + 1] ← Heap_ExtractMax()
  return A
```



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

Animation (3)

```
Heap_Sort(Array)
  Build_Heap(Array)
  N ← size(Array)
  for i from 1 to N
    A[N – i + 1] ← Heap_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	17	3	1	7	2	25	26	36		

Java Implementation

- Priority Queue ADT
- Heap Class
  - shiftUp
  - Heap_Insert
  - shiftDown
  - Heap_ExtractMax
  - Build_Heap
  - Heap_Sort
- In OOP Style

PS4

- PS4 will only be released on Tuesday of Week05
- So, enjoy your CNY break ☺

Summary

- In this lecture we 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!!!
  - Simple application of Heap DS: Heap_Sort
- See you again in the 2nd half of CS2020
  - We will use Heap/PriorityQueue for this algorithm:
    - Dijsktra's algorithm for Single Source Shortest Paths Problem
- Note about Today's Recitation Classes:
  - Still with Dr Seth Gilbert, about Balanced Trees