CS2040S – Data Structures and Algorithms

Lecture 9 – Heaps of Fun

chongket@comp.nus.edu.sg

National University of Singapore

School of Computing

Outline

What are you going to learn in this lecture?

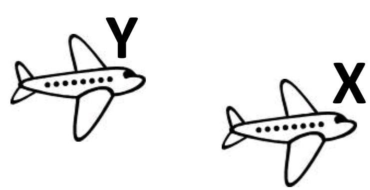
- Motivation: Abstract Data Type: **PriorityQueue**
- With major help from VisuAlgo Binary Heap Visualization
  - **Binary Heap** data structure and its operations
  - Creating a Heap from a set of **N** numbers in **O(N)**
  - **Heap Sort** in **O(N log N)**

Reference in CP4 book 1: Page 78-80

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 few slides are hidden…

(in public copy)

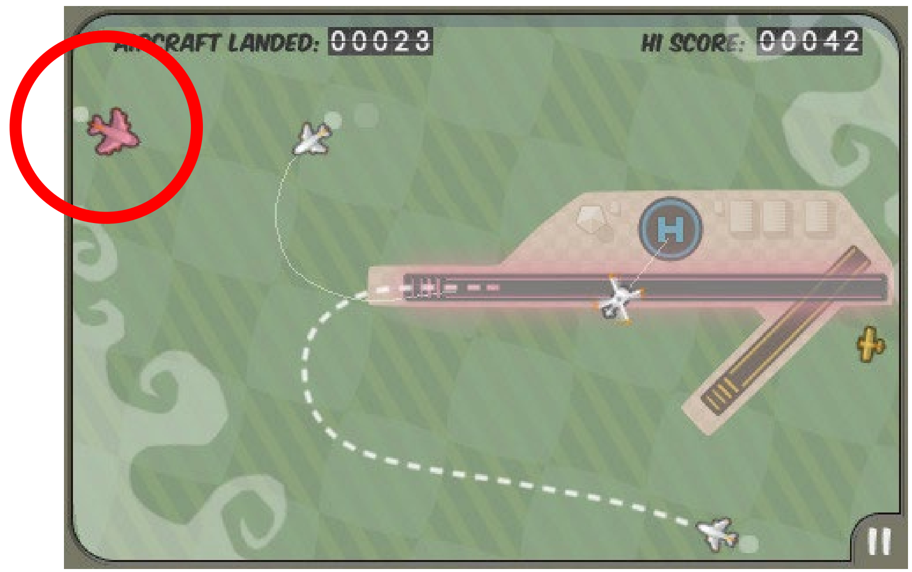
Attend the lecture to figure out

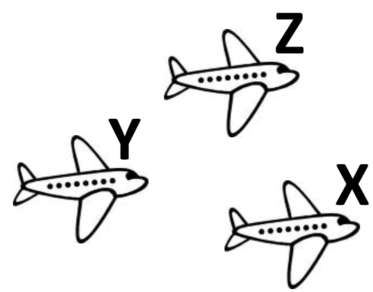
There will be two options presented and <u>you will have to decide</u>

- Raise **AND** wave your hand if you choose option 1
- Raise your hand but do **NOT** wave it if you choose option 2
- Do nothing if you are not sure what to do

Abstract Data Type: PriorityQueue (2)

- Suddenly, you receive an urgent SOS message that another **aircraft Z** is running out of fuel and request to land soon
- The pilot of **aircraft Z** estimates that he only have **3 minutes of flying time** and approximately **3 minutes** away from airport……………..
- You…





You...

1. Let aircraft Z lands first...
2. Stick with the original plan...

Avianca Flight 52

You...

1. Let aircraft Z lands first...
2. Stick with the original plan...

And you need to make such
“priority change” decision FAST

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)

Note: We can always define highest priority = higher number or it's opposite: highest priority = lower number

A 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 some properties

- Each operation on that data structure has to **maintain** those properties

PriorityQueue Implementation (1)

The array is circular: We just manipulate front+back pointers to define the active part of array

(Circular) Array-Based Implementation (Strategy 1)

- Property: The content of array is always in correct order
- Enqueue(**x**)
  - Find the **correct insertion point**, O(**N**) – recall insertion sort
- **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*

We do not need to close the gap, just advance the front pointer, O(**1**)

PriorityQueue Implementation (2)

(Circular) 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 (front)	1 (back)
Key	Aircraft X*	Aircraft Y*
		Aircraft Z**

We may need to close the gap if this operation causes it, also O(**N**)

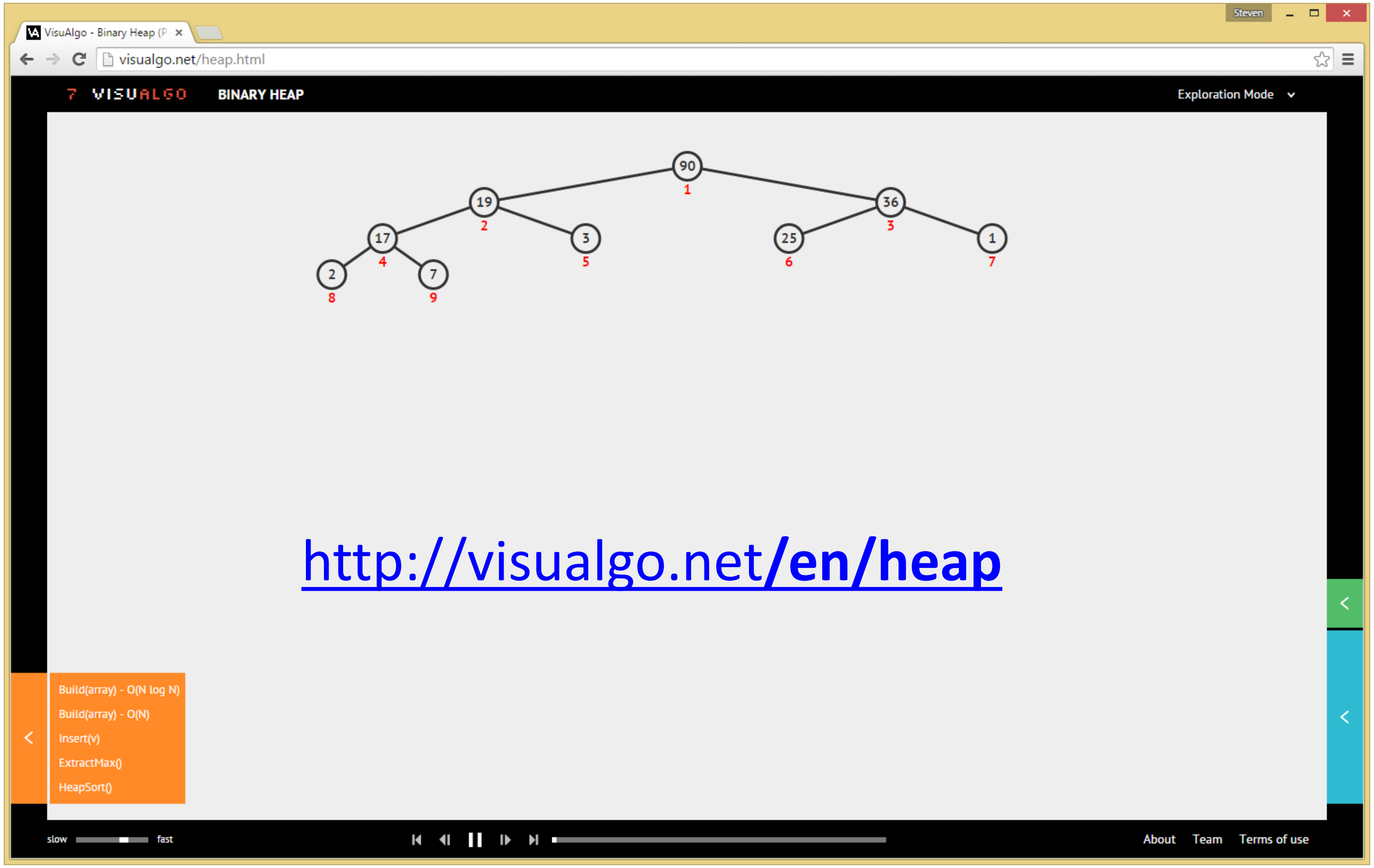
Index	0 (front)	1	2 (back)
Key	Aircraft X*	Aircraft Y*	Aircraft Z**

PriorityQueue Implementation (3)

If we just stop at CS2040S 1st half knowledge level:

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

If N is large, our queries are slow…



INTRODUCING BINARY HEAP DATA STRUCTURE

Complete Binary Tree

Introducing a few concepts:

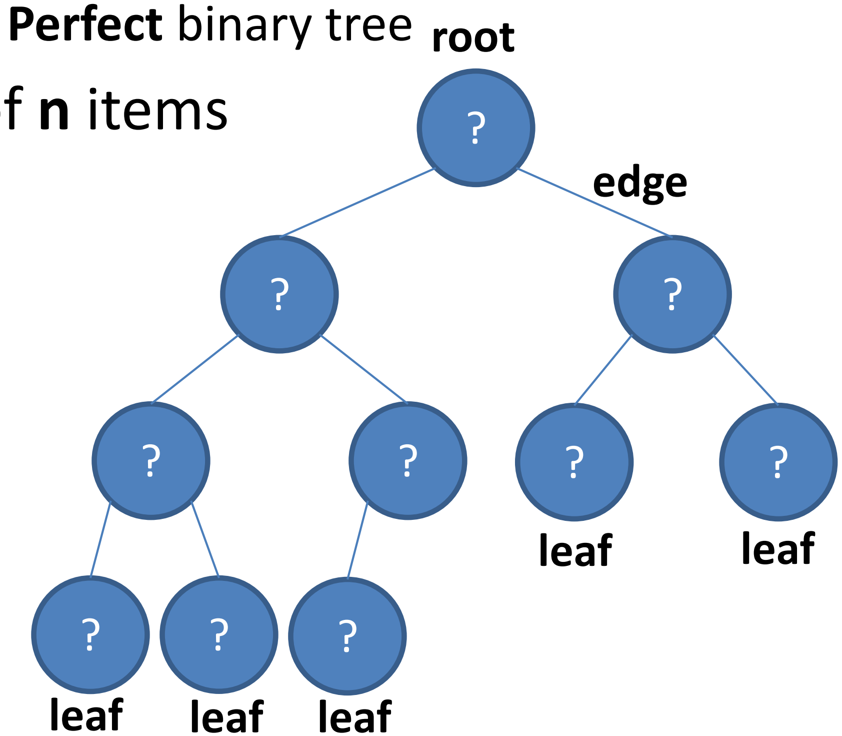
- **Complete** Binary Tree
  - Binary tree in which every level, *except possibly the last*, is completely filled, and all nodes are as far left as possible
  - If every level including last is filled → **Perfect** binary tree
- **Height** of a complete binary tree of **n** items

  = number of levels-1

  = *max edges from root to deepest leaf*

given some complete bin tree with at least one node in last level, the formula for finding the exact height is height = floor(log n)

Internal vertices = Every node other than leaves & root



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)

Memorize this answer!
We will need that for *nearly all* time complexity analysis of binary heap operations

Storing a Complete Binary Tree

Q: Why not 0-based?

indexes change and navigation operations dont work anymore

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

`size(A)`

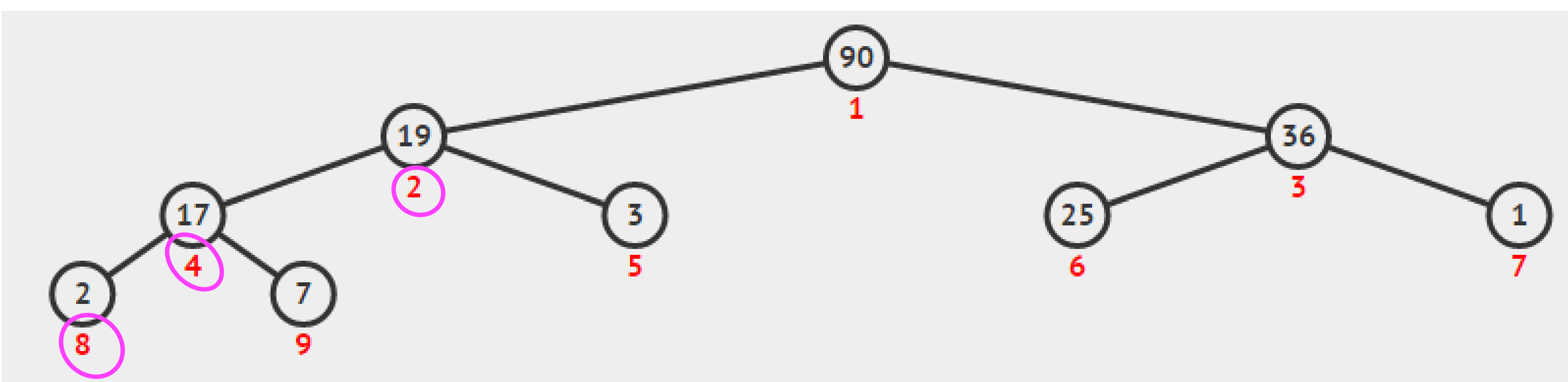
0	1	2	3	4	5	6	7	8	9	10	11
NIL	90	19	36	17	3	25	1	2	7	-	-

`heapsize ≤ size(A)`

heapsize is index of last element in the array

Navigation operations:

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



Binary Heap Property

2 properties defined for binary heap -> one for min heap and one for max heap

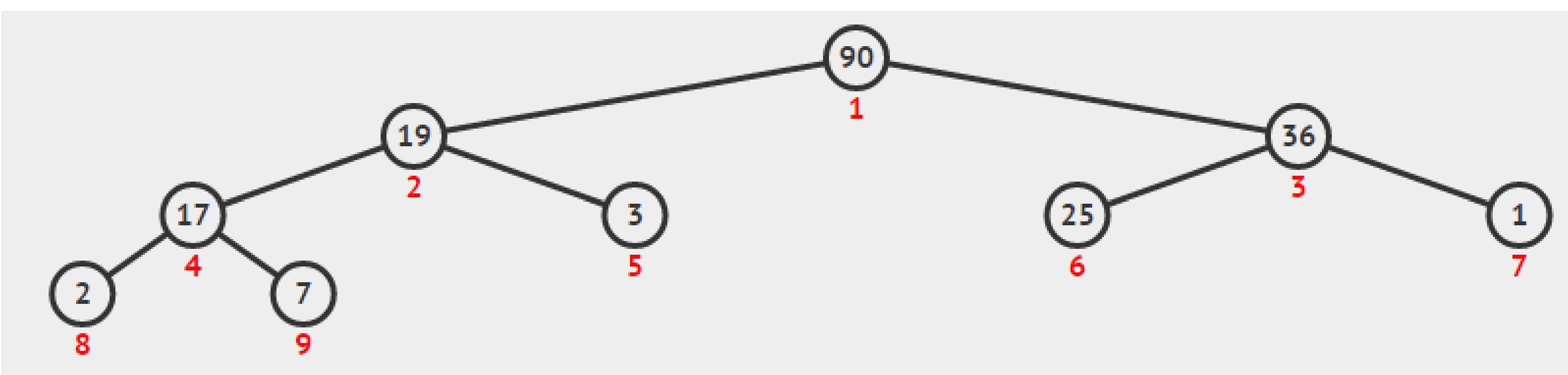
Binary Heap property (except root)

value stored at parent of i must be >= value stored at i

- `A[parent(i)] ≥ A[i]` **(Max Heap)**
- `A[parent(i)] ≤ A[i]` **(Min Heap)**

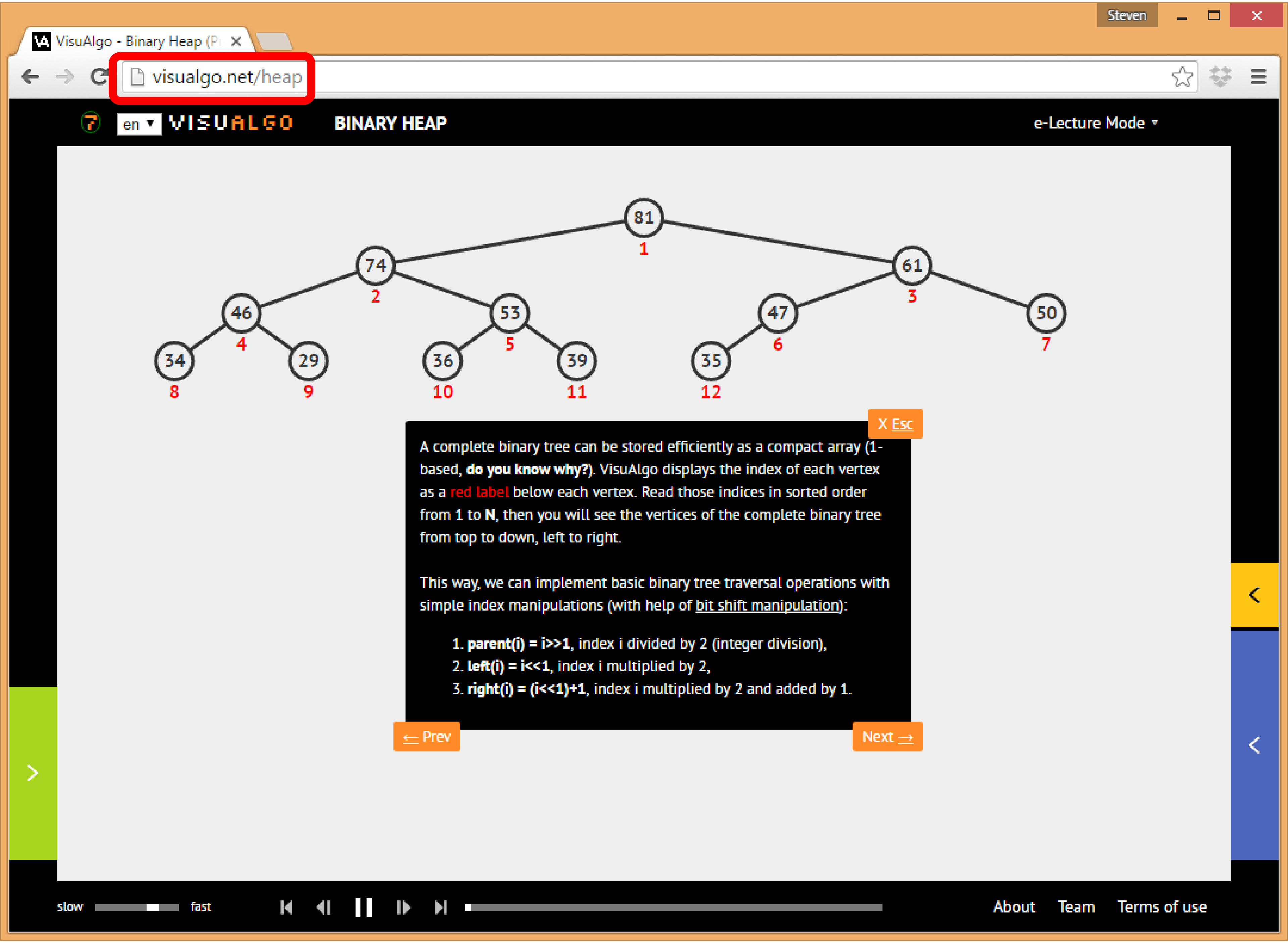
Q: Can we write Binary Max Heap property as:
`A[i] ≥ A[left(i)]`
`&&`
`A[i] ≥ A[right(i)]`
?

Without loss of generality, we will use **(Binary Max) Heap** for all examples in this lecture, and it will store only distinct integer values



The largest element in a **Binary Max Heap** is stored at…

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



VisuAlgo - Binary Heap (P
Steven
visualgo.net/heap
en
VISUALGO
BINARY HEAP
e-Lecture Mode
81
1
74
2
61
3
46
4
53
5
47
6
50
7
34
8
29
9
36
10
39
11
35
12
X Esc
A complete binary tree can be stored efficiently as a compact array (1-based, do you know why?). VisuAlgo displays the index of each vertex as a red label below each vertex. Read those indices in sorted order from 1 to N, then you will see the vertices of the complete binary tree from top to down, left to right.
This way, we can implement basic binary tree traversal operations with simple index manipulations (with help of bit shift manipulation):
1. parent(i) = i>>1, index i divided by 2 (integer division),
2. left(i) = i<<1, index i multiplied by 2,
3. right(i) = (i<<1)+1, index i multiplied by 2 and added by 1.
← Prev
Next →
slow
fast
About
Team
Terms of use

Insert(v) – Pseudo Code

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

// Preliminary analysis:
// Time complexity Insert(v) depends on time
// complexity of ShiftUp(i)
```

ShiftUp – Pseudo Code

This name is <u>not unique</u>, the alternative names are:

ShiftUp/BubbleUp/IncreaseKey/etc

“not root” “violates max heap property”

```
ShiftUp(i)
  while i > 1 and A[parent(i)] < A[i] // swap
    swap(A[i], A[parent(i)])
    i = parent(i)
// Analysis: ShiftUp() runs in __O(log n)__
```

at most starts at last leaf and goes to root so it traverses entire height

ExtractMax - Pseudocode

max value is at root

```
ExtractMax()
  maxV ← A[1] // O(1)
            copy last leaf at index heapsize and copy it into the root
  A[1] ← A[heapsize] // O(1)
  heapsize ← heapsize-1 // O(1)
  ShiftDown(1) // O(?) O(log n)
  return maxV

// Preliminary analysis:
// Time complexity of ExtractMax() depends on
// time complexity of ShiftDown()
```

ShiftDown – Pseudo Code

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

```
ShiftDown(i)
    while curr <= heapsize not the leaf yet
  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)
    // be careful with the implementation
    if (max_id != i)
      swap(A[i], A[max_id])
      i ← max_id;
    else
      break; // Analysis: ShiftDown() runs in O(log n)
```

PriorityQueue Implementation (4)

Now, with knowledge of *non linear* DS:

Strategy	Enqueue	Dequeue
Circular-Array-Based PQ (1)	O(**N**)	O(**1**)
Circular-Array-Based PQ (2)	O(**1**)	O(**N**)
Binary-Heap (actually uses array too)	Insert(key) O(log **N**)	ExtractMax() O(log **N**)

Summary so far:

Heap data structure is an efficient data structure ***-- O(log N) enqueue/dequeue operations --*** to implement ADT priority queue where the 'key' represent the 'priority' of each item

`CreateHeap(arr)`, O(N log N) Version

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

// Analysis: This clearly runs in O(N log N)
// Can we do better?
```

CreateHeap(arr), O(N) version

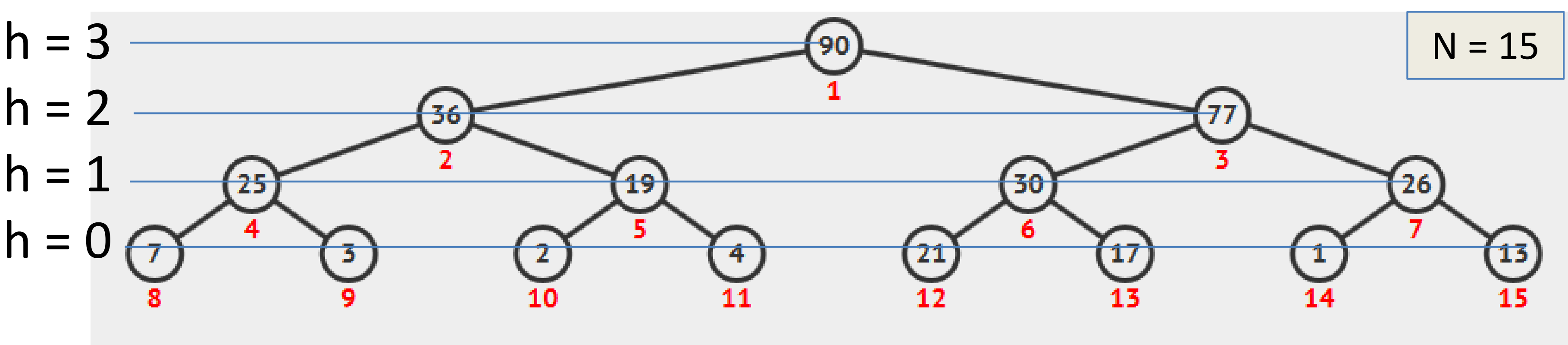
```
CreateHeap(arr)
  heapsize ← size(arr)
  A[0] ← 0 // dummy entry
  for i = 1 to heapsize // copy the content O(N)
    A[i] ← arr[i-1]
                     start from parent of the last leaf which is heapsize
  for i = parent(heapsize) down to 1 // O(N/2)
    ShiftDown(i) // O(log N)

// Analysis: Is this also O(N log N) ??
// No... soon, we will see that this is just O(N)

                    // Inventor: Robert W. Floyd
```

`CreateHeap(arr)` Analysis… (1)

Recall: What is the height of a complete binary tree (heap) of size **N**? floor(log n)____________

Recall: What is the cost to run `shiftDown(i)`? depends on level i is at_______

Q: How many nodes are there at height **h** of a perfect binary tree? ceil(N/2^n+1)_____________



`CreateHeap(arr)` Analysis... (2)

Cost of `CreateHeap(arr)` is thus:

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

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

$\boxed{x = 1/2}$

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

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

Can check WolframAlpha:
http://www.wolframalpha.com/input/?i=0%2F1+%2B+1%2F2+%2B+2%2F4+%2B+3%2F8+%2B+4%2F16+...

Review: We have already learnt MergeSort. It can sort **N** items in…

1. O(**N**^{2})
2. O(**N** log **N**) 
3. O(**N**)
4. O(log **N**)

HeapSort(arr) Pseudo Code

With a Binary (Max) Heap, we can do sorting too ☺

- Just call ExtractMax() **N** times
- If we do not have a Binary (Max) Heap yet, simply build one!

```
HeapSort(array)
  CreateHeap(array) // O(?) N
  N ← size(array)
  for i from 1 to N // O(N)
    A[N-i+1] ← ExtractMax() // ~O(log N)
  return A
// Inventor: John William Joseph Williams
```

`HeapSort(arr)` Analysis

```
HeapSort(arr)
  CreateHeap(arr) // The best we can do is ___O(n)___
  N ← size(arr)
  for i from 1 to N // O(N)
    A[N-i+1] ← ExtractMax() // O(log N)
  return A

// Analysis: Thus HeapSort runs in O(___O(N log n)___)
// 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"
```

Java Implementation

Priority Queue ADT

BinaryHeap Class (Java file given)

- `ShiftUp(i)` used in `Insert(key)`
- `ShiftDown(i)` used in `ExtractMax()`
- `CreateHeapSlow(arr)` and `CreateHeap(arr)`
- `HeapSort(arr)`

Testing/Training Binary Heap knowledge on Visualgo ☺

- Go to http://visualgo.net/training.html
- Click Binary Heap
- Set the question difficulty (go from easy to hard)
- Set the number of questions (try 5 to 10 questions)
- Set a suitable time limit (20 to 60 mins)

Summary

In this heavy VisuAlgo lecture, we have looked at:

- Heap DS and its application as efficient PriorityQueue
- Storing (max) heap as a compact array and its operations
  - Remember how we always try to maintain complete binary tree and (max) heap property in all our operations!
- Building a (max) heap from a set of numbers in O(**N**) time
- Simple application of Heap DS: O(**N** log **N**) HeapSort

We will still be using PriorityQueue later on in CS2040S