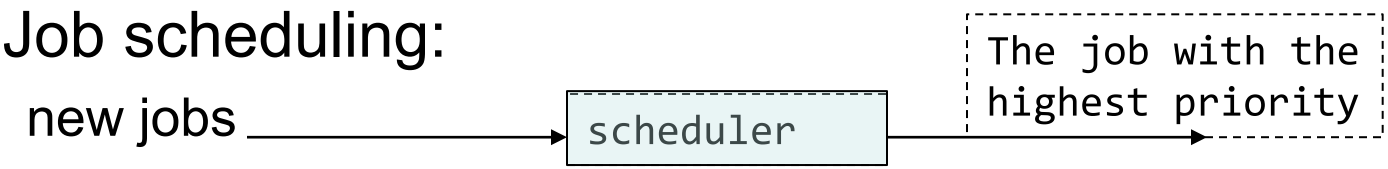
EE2331 Data Structures and Algorithms

Heap

Compare two applications

- Dictionary:
  - Look-up (very frequent)
  - Insertion of a new word (not so frequent)
  - Print-all, deletion
- Job scheduling:



  - Insert new job (frequently)
  - Extract the most urgent job (frequently)
  - Delete the completed job (frequently)

Compare several applications

- Examples:
- Printer; CPU(O.S.); routes; I/O; embedded systems

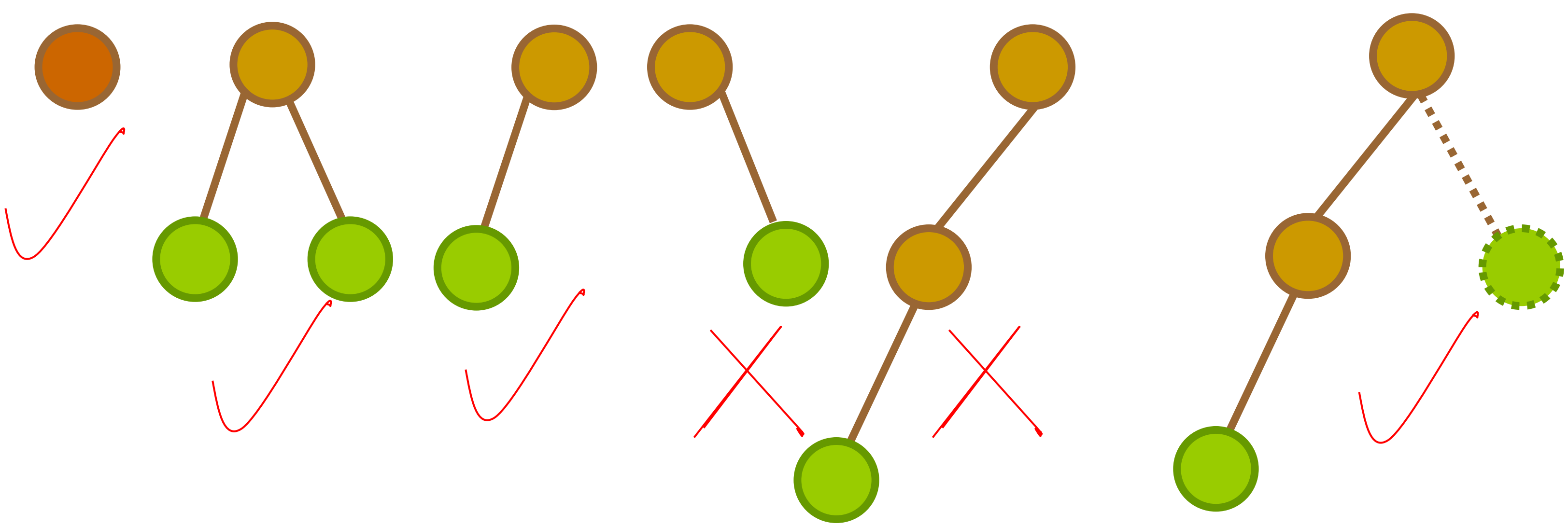
	Balanced BST	Sorted array	Unordered array
Find max	O(log(n))	O(1)	O(n)
insert	O(log(n))	O(n)	O(1)
Delete (given a key)	O(log(n))	O(n)	O(n)
search	O(log(n))	O(log(n))	O(n)

What is a Heap?

- Efficiently support Insertion / FindMax / Extraction
- A max tree is a tree in which the key value in each node is no smaller than the key values in its children (if any).
- Similarly, a min tree is a tree in which the key value in each node is no bigger than the key values in its children (if any).
- A max heap (descending heap) is a complete binary tree that is also a max tree.
- A min heap (ascending heap) is a complete binary tree that is also a min tree.

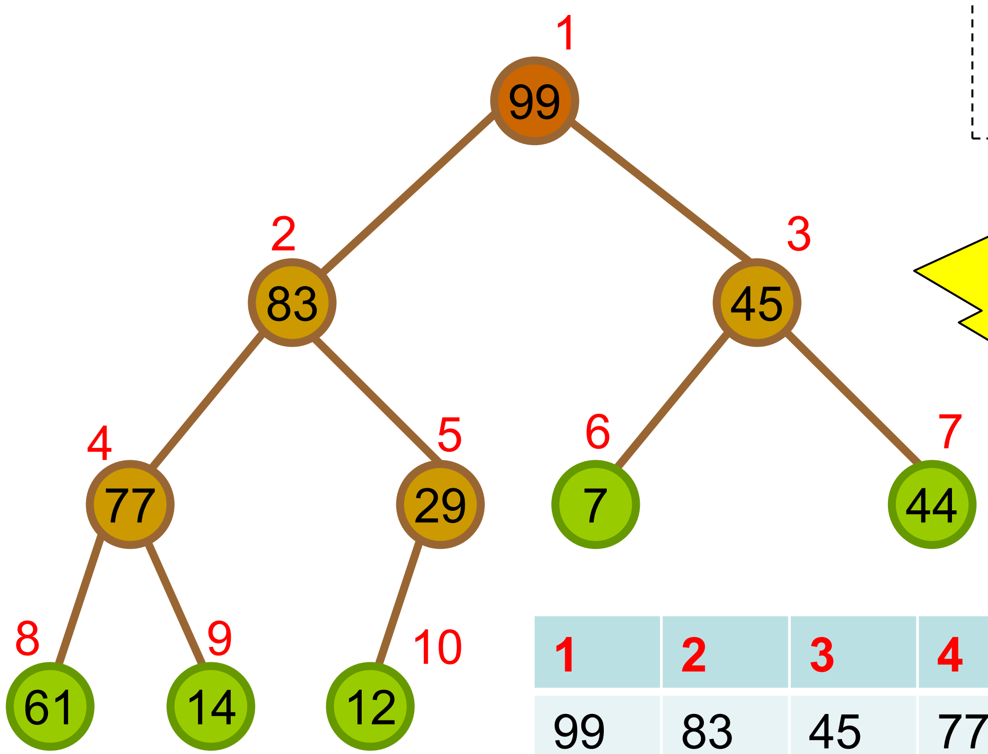
Review of complete binary tree

- **Complete binary tree** is a tree in which every level, **except possibly the last**, is <u>completely</u> filled, and all nodes are **as far left as possible**.
- It can have between 1 and 2^{m-1} nodes at the last level m.
- Exercise: Which is complete binary tree?



How to store a complete binary tree?

- With node and tree structure
- With array



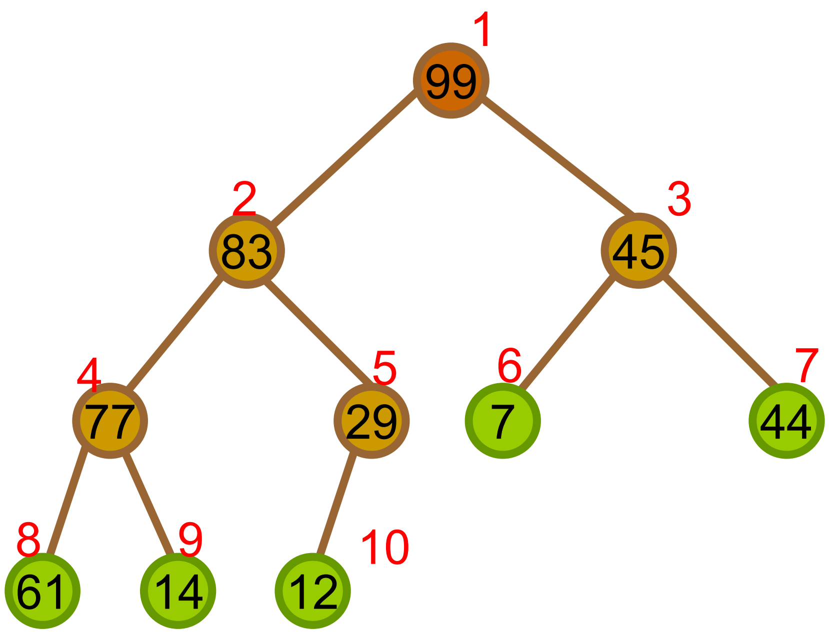
Heap has two views:
Tree view and array view.
You should know both.

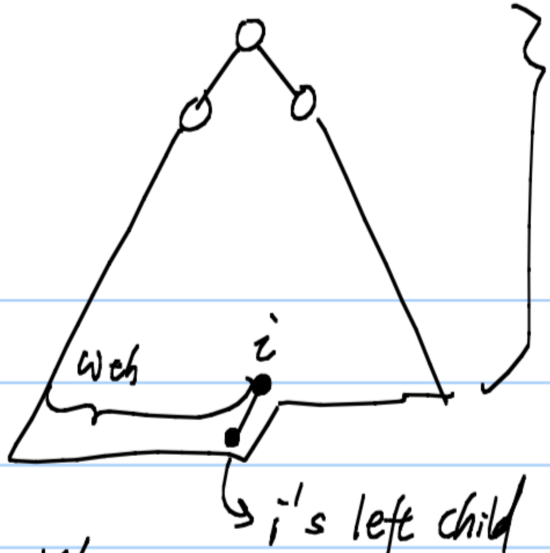
Alarm:
We save the tree in an array
with starting index 1

1	2	3	4	5	6	7	8	9	10
99	83	45	77	29	7	44	61	14	12

Node index in array

- There is a strong connection between the node index of parent-child node-pair
- Index of each left child = 2* index of its parent
- Index of each right child = 2* index of its parent +1



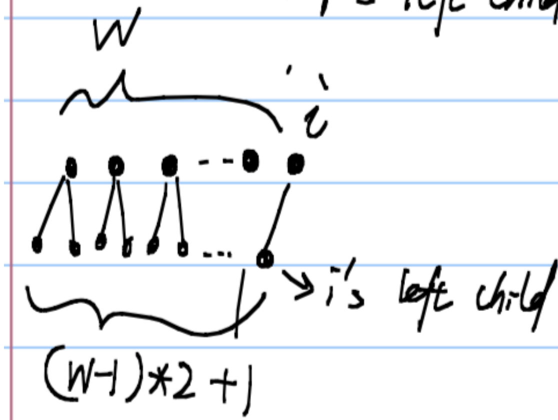


index of node i in the heap array

$$\underbrace{2^0+2^1+2^2+\cdots+2^{d-1}}+W = \boxed{\text{index of } i}$$

$$\underbrace{a_1+a_1q+a_1q^2+\cdots+a_1q^n}_{n+1 \text{ items}} = \frac{a_1(1-q^{n+1})}{1-q}$$

$$= \frac{1(1-2^d)}{1-2}+W = \boxed{2^d-1+W}$$

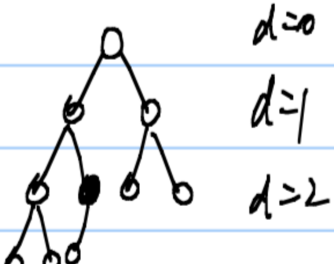


$\boxed{\text{index of i's left child:}}$

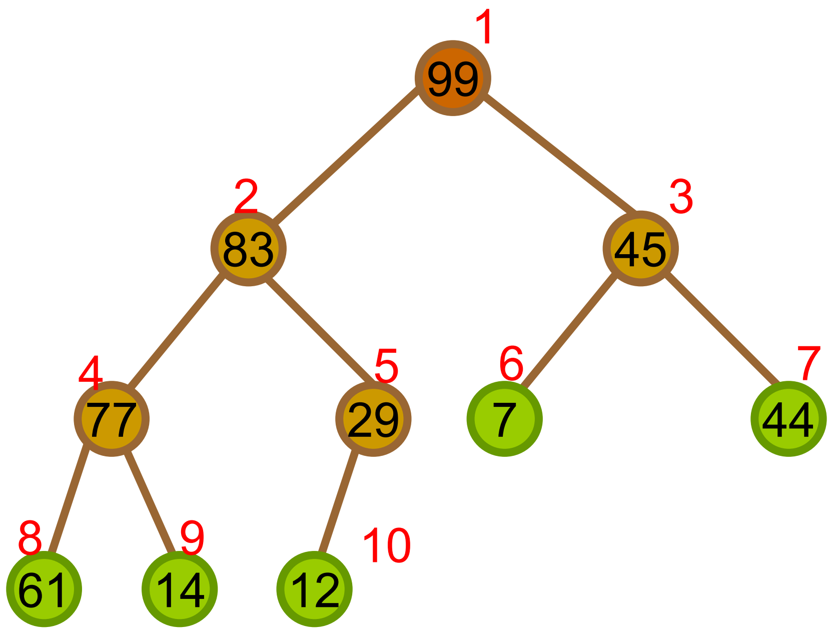
$$\underbrace{2^0+2^1+\cdots+2^d}_{d+1 \text{ items}}+(W-1)*2+1$$

$$= \frac{1(1-2^{d+1})}{1-2}+(W-1)*2+1$$

$$= \boxed{2^{d+1}+2W-2}$$



Node index in array

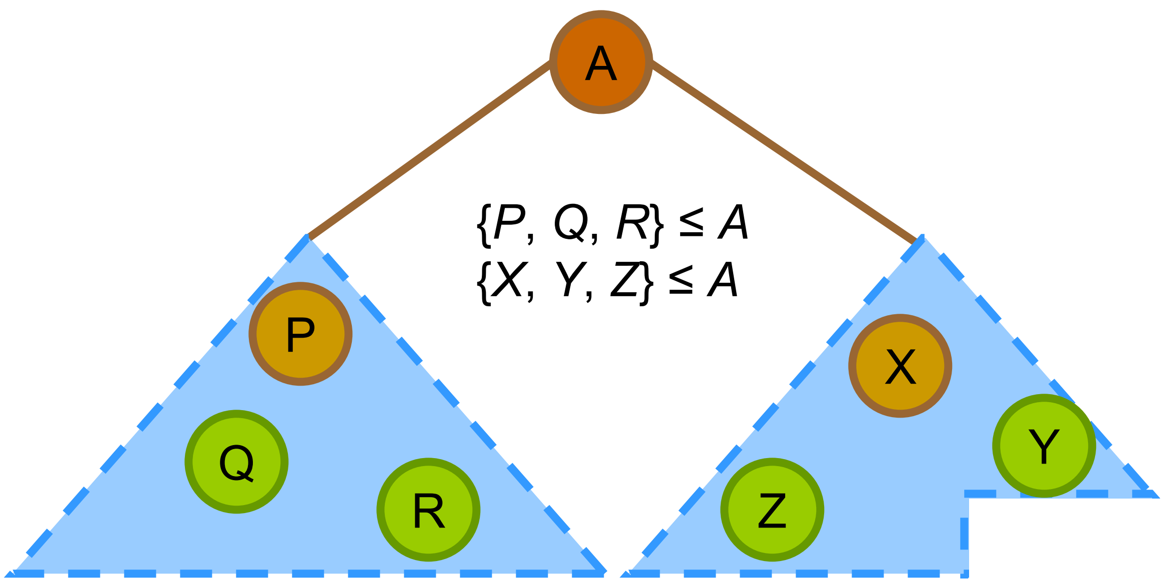


1	2	3	4	5	6	7	8	9	10
99	83	45	77	29	7	44	61	14	12

What can be done with these properties?

Using Heap as Priority Queue

- In Priority Queue, the element to be deleted (dequeue) is the one with the highest priority.
- Max Heap always has the largest element in root



Using Heap as Priority Queue

- Frequent operations in Priority Queue (Heap):
- A) Find the element with highest priority
- B) Insert new element and keep the property
- C) Delete the element with highest priority and keep the property
- D) Build a priority queue (heap) from a disordered array

Find Max

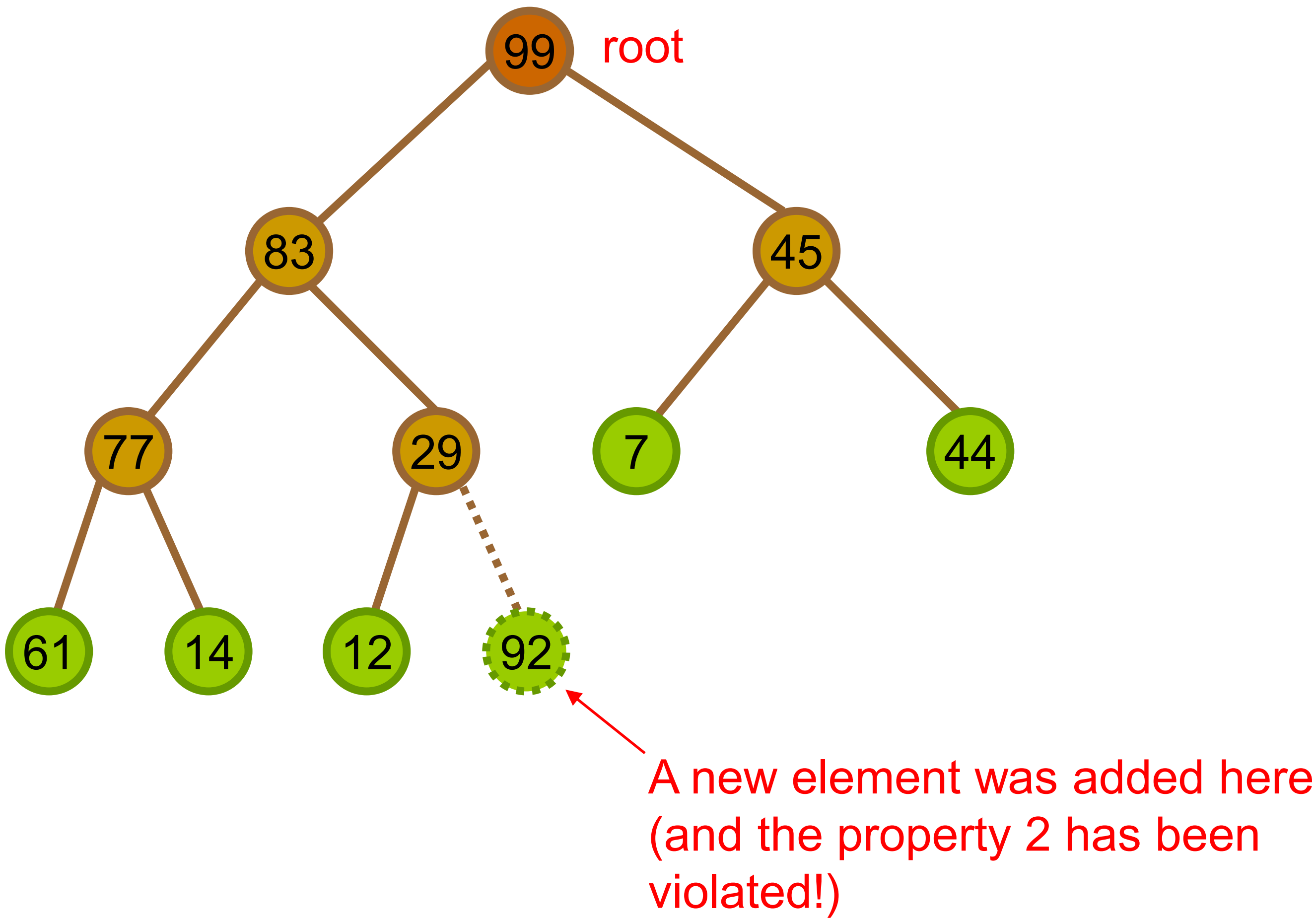
- The root of a heap always has the largest key (in maxheap)
- H[1] should be the maximum
- where H denote the heap, and 1 represent the index

- Time ~ O(1)

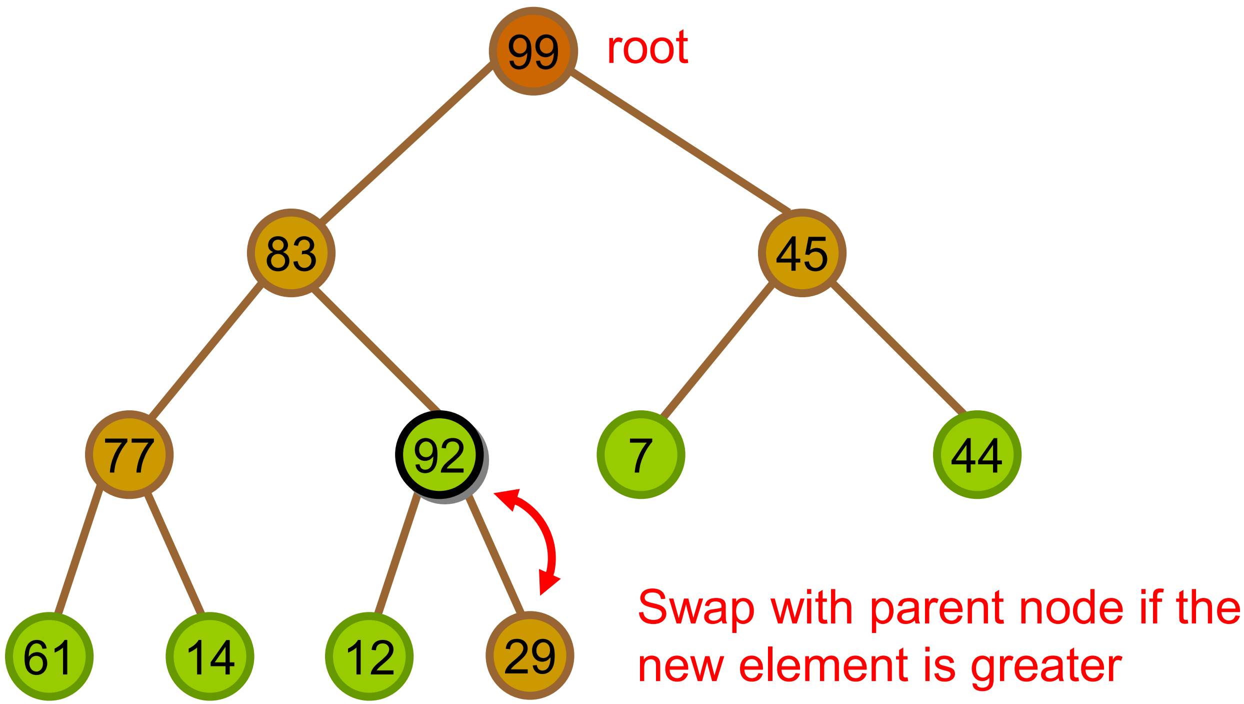
Insert Node Into Heap

- Step 1) push back the new element to the array (what is this position in the tree?)
- Step 2) **Percolate up**
  - Swap with its parent node recursively until it satisfies the property of heap

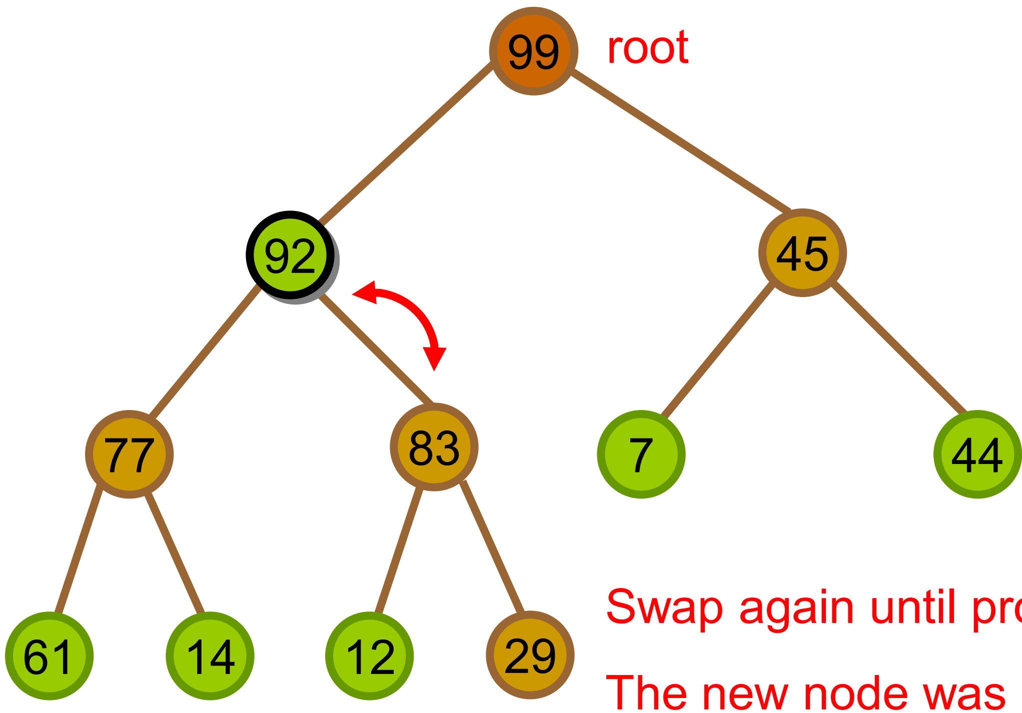
Example: Percolate Up



Example: Percolate Up



Example: Percolate Up



Swap again until property 2 is preserved!

The new node was propagated upward!

Insert Node Into max-Heap

```
Def Heap_insert(H,key)
        Heap_size = Heap_size(H)+1       //increase array size
(the implementation of this step depends on what data type you
use, e.g. array, linked list, or vector)
        i = Heap_size(H)         //i is the index of new element
        while i > 1 and H[parrent(i)] < key
                H[i] = H[parent(i)]
                i = parent(i)    //parent(i)=int(i/2)
        H[i] = key
```

- Time ~ O(log(n))
- n: number of nodes

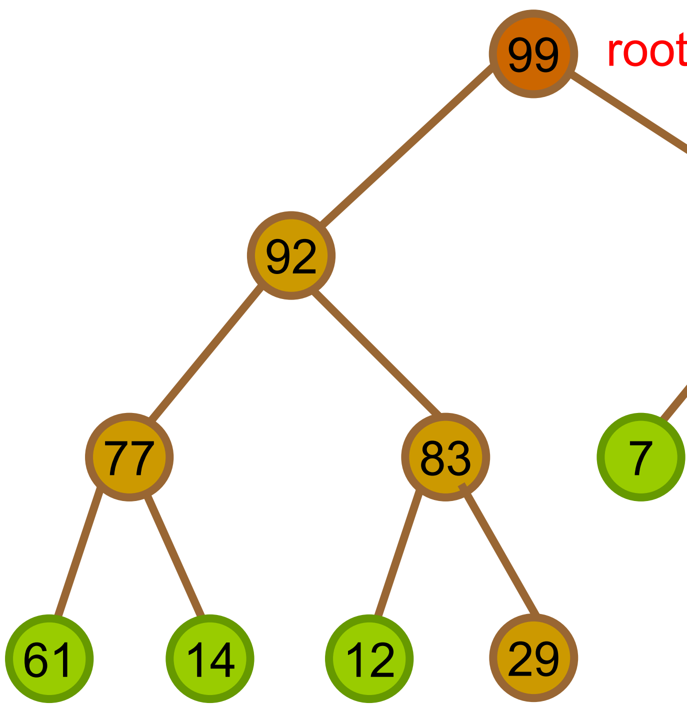
In-class exercise:
Insert 95 into heap H={100, 90, 40, 50, 55, 20, 19}. Apply Heap_insert(H,95) and show H in each iteration of the while loop and the final H.

This is the pseudocode, don't directly copy this code. It won't work.

Does this part ring a bell? (similar to sth. We learned before?)

Remove Node From Heap

In heap, remove the largest element. What is your algorithm/strategy?

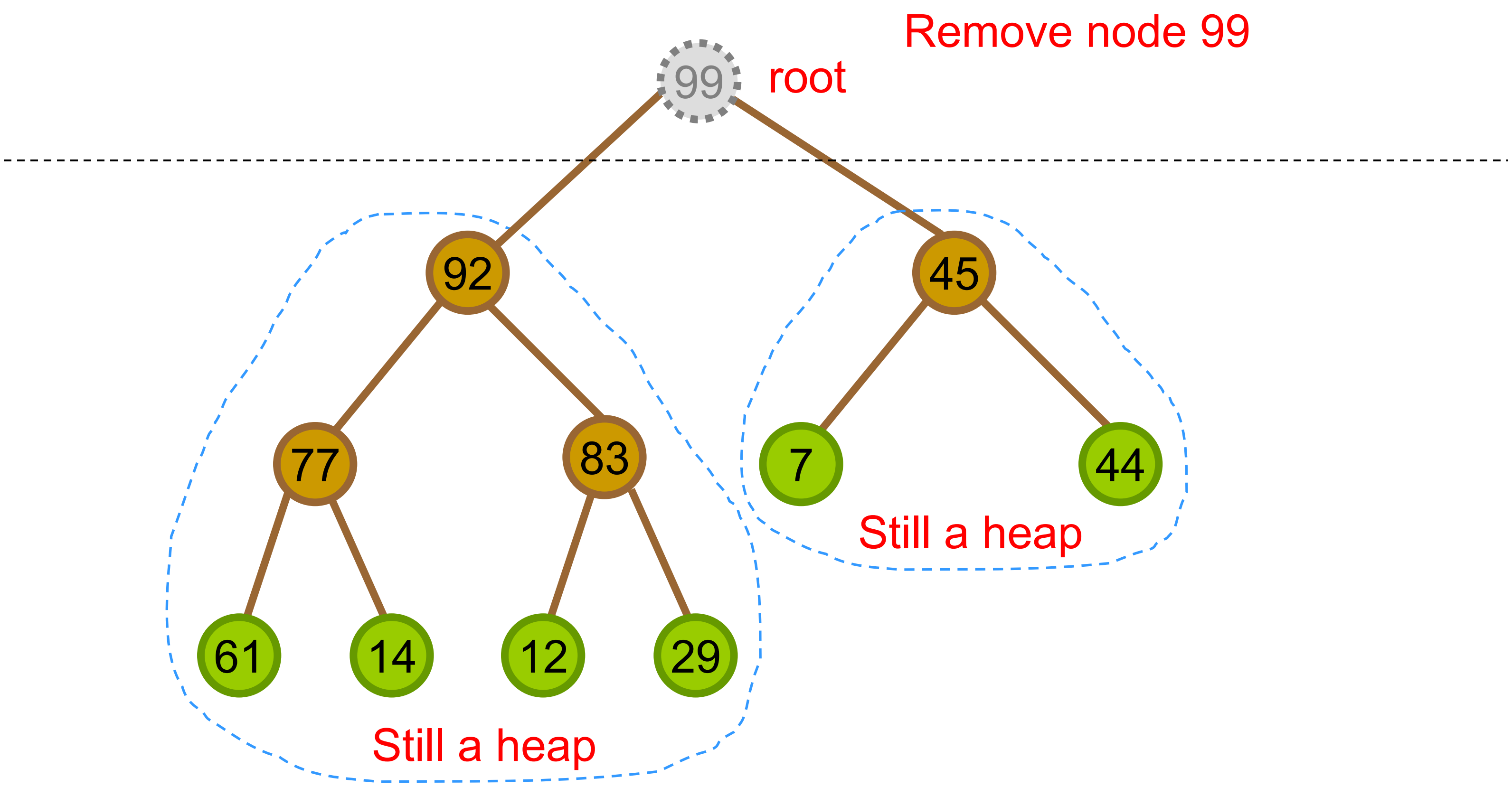


Remove Node From Heap

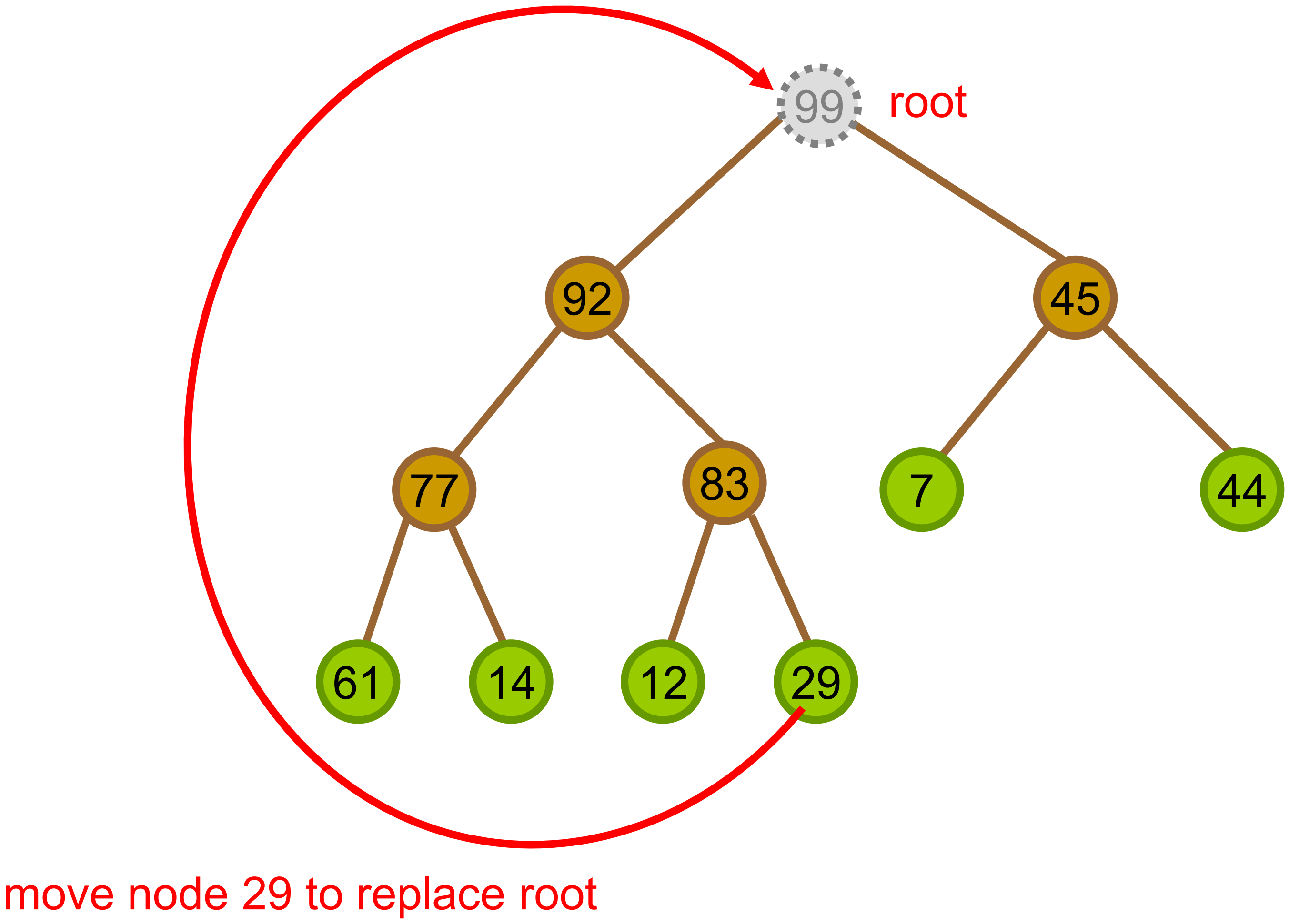
In heap, remove the largest element. What is your algorithm/strategy?

- Step 1) Replace the root node with the bottom rightmost element
- Step 2) **Percolate down(Heapify)**
  - Swap with the its greater child node recursively until it satisfies the heap property

Example: Percolate Down

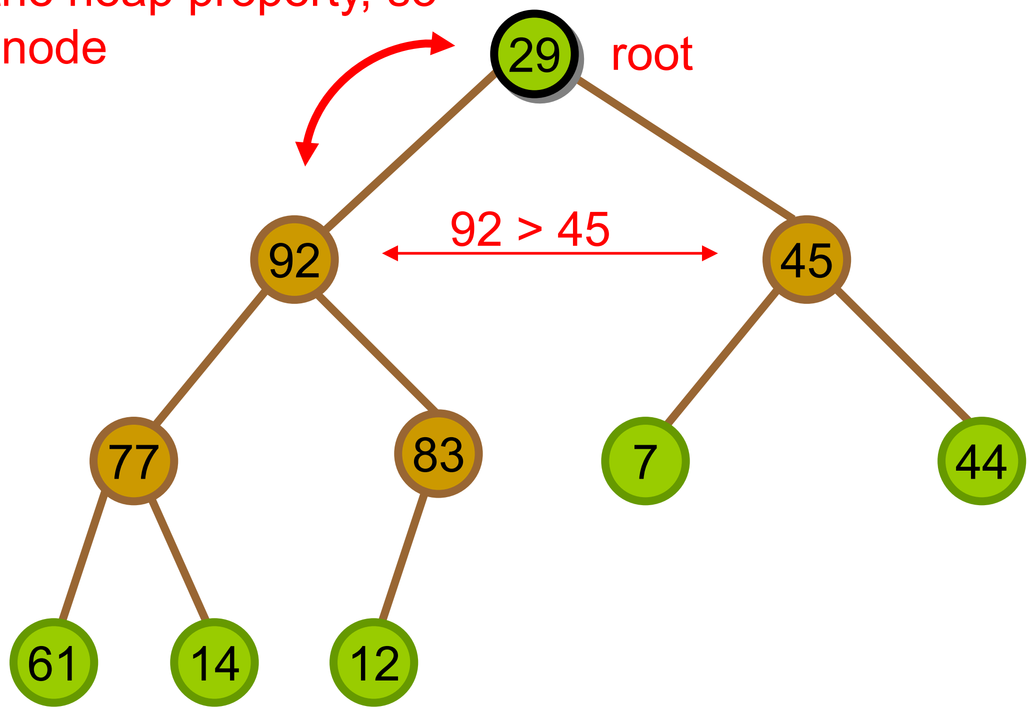


Example: Percolate Down



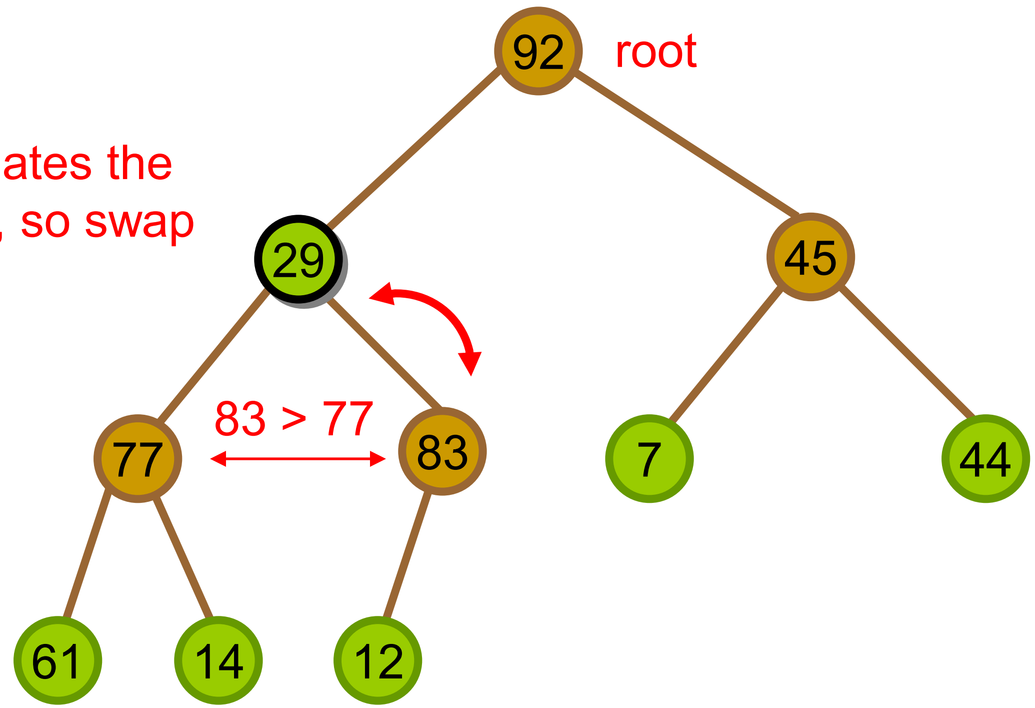
Example: Percolate Down

Now it violates the heap property, so swap with child node



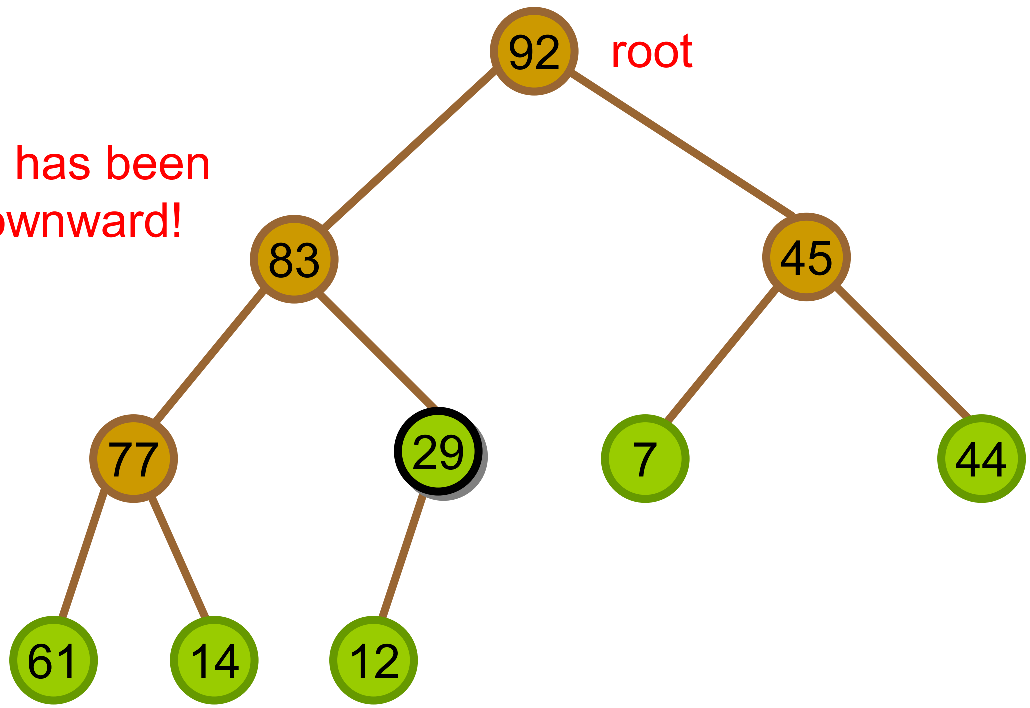
Example: Percolate Down

Now it still violates the heap property, so swap again



Example: Percolate Down

The new node has been
propagated downward!

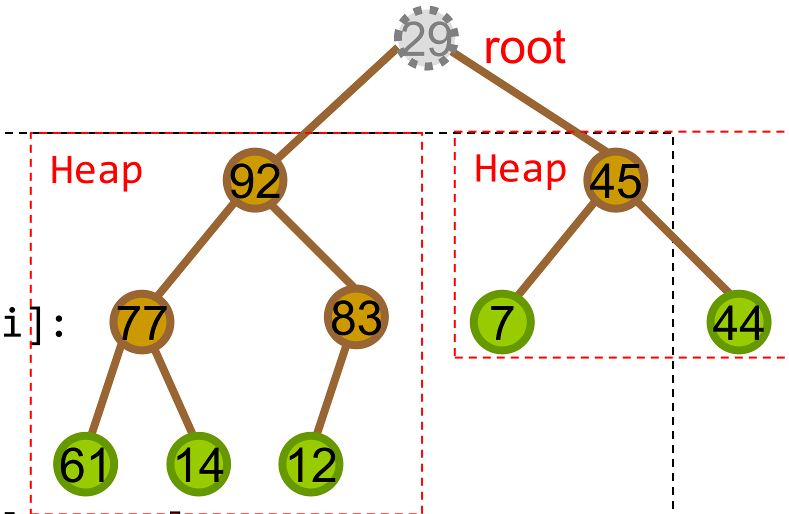


Remove root Node From Heap

```
Def Heap_Extraction(H)
        if heap_size(H)<=0:
                throw Error ”heap_underflow”
        max = H[1]
        H[1] = H[Heapsize(H)]    //move last element to root
        Heap_size(H)-=1          //Heap_size minus 1
        Heapify(H,1)
        return max
```

- Heapify(H,i) : Given an array with left(i) and right(i) being heaps, make i a heap
- left(i) = 2*i,   right(i) = 2*i+1
- Time ~ Time(Heapify(H,1))

Heapify

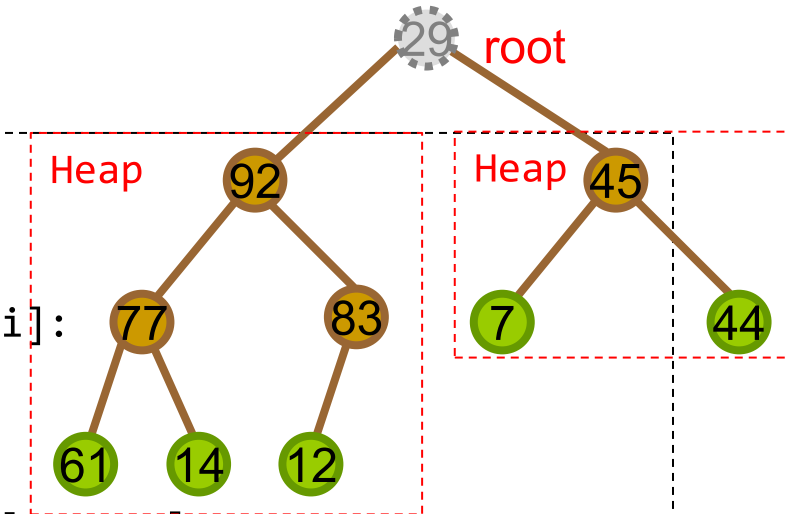



```
Def Heapify(H,i)
        l = left(i)
        r = right(i)
        if l<=Heap_size(H) and H[l]>H[i]:
                largest = l
        else:
                largest = i
        if r<=Heap_size(H) and H[r]>H[largest]:
                largest = r
        if largest!=i:
                swap(H[i],H[largest])
                Heapify(H,largest)
```

- Heapify(H,i) : Given an array with left(i) and right(i) being heaps, make i a heap
- Time ~ O(log(n))

Heapify

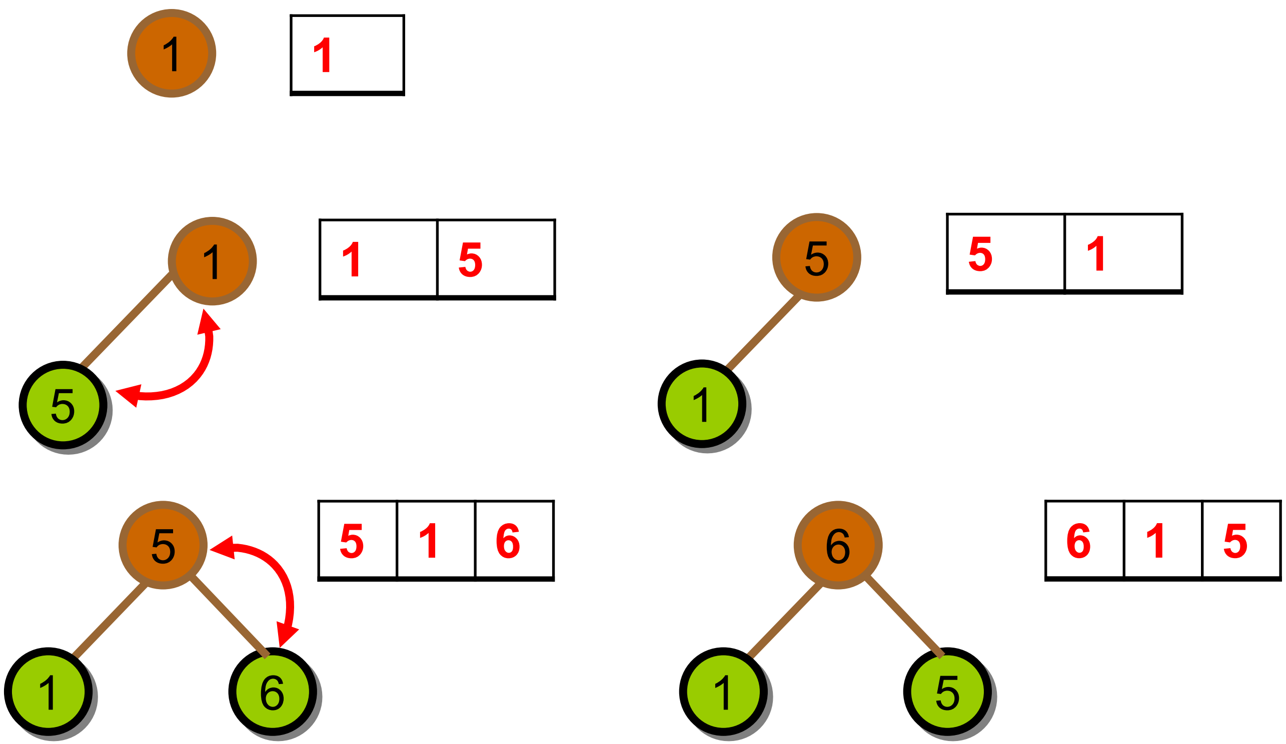
```
Def Heapify(H,i)
        l = left(i)
        r = right(i)
        if l<=Heap_size(H) and H[l]>H[i]:
                largest = l
        else:
                largest = i
        if r<=Heap_size(H) and H[r]>H[largest]:
                largest = r
        if largest!=i:
                swap(H[i],H[largest])
                Heapify(H,largest)
```



- Exercise: show H during the process of calling Heapify(H,1)

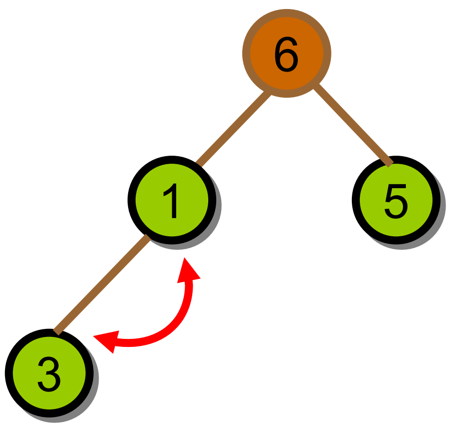
Build a Heap

- Convert an array into a heap using insertion. Input array: 1 5 6 3 2

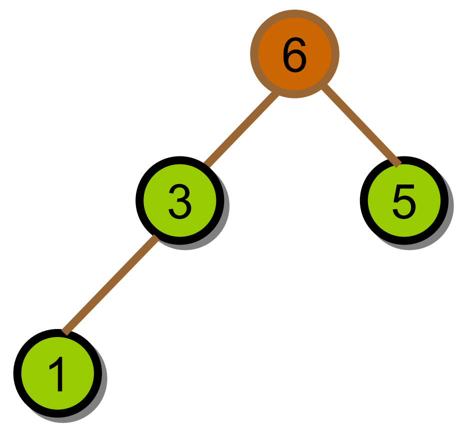


Build a Heap

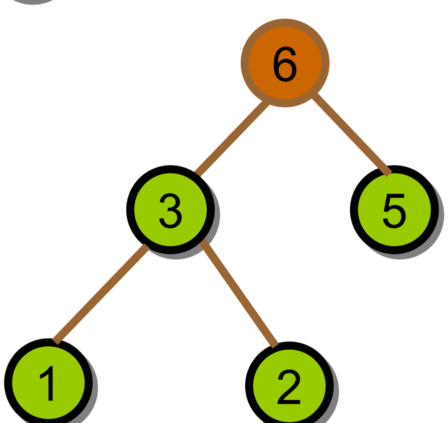
- Convert an array into a heap using insertion



6	1	5	3



6	3	5	1



6	3	5	1	2

Build a Heap

- Convert an array into a heap using insertion
- Time ~ O(n logn) where n denotes the number nodes
- Heap construction using insertion is not the fastest heap construction algorithm. A faster algorithm builds the heap using “heapify” operation in a bottom-up fashion. That algorithm can reach O(n).

Heap construction with O(n)

```
Def Build-Heap(H)
        for i = ⌊N/2⌋ down to 1
                do Heapify(H,i)
```

$\left\lfloor \frac{N}{2} \right\rfloor$: the last non-leaf node

```
Example

Input array: 1 5 6 3 2     N=5, N/2 rounds to 2

Heapify(H,2)  1 5 6 3 2 →1 5 6 3 2 //no change
Heapify(H,1)  1 5 6 3 2 →6 5 1 3 2
```

Rough running time analysis: O($\left\lfloor \frac{N}{2} \right\rfloor$ logN) = O(NlogN)

In-class exercise: Build a heap for input array: 3 4 1 9 2 8 0 using the above pseudocode. Show the output of each Heapify operation

Tighter build-heap running time

- We build the heap (tree) from the bottom
- Tree starts small and grows bigger
- The worst case for any "heapify" is moving i to bottom. This is the height of i
- Total time is thus $\sum_{i=1}^{N/2} height(i)$

```
Def Build-Heap(H)
        for i = ⌊N/2⌋  down to 1
                do Heapify(H,i)
```

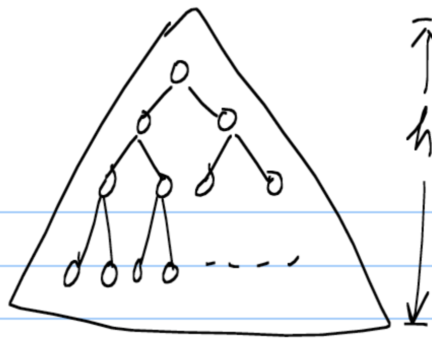
Question about heap construction's running time

- Q1: What is the tight running time?
  Is it the sum of the **depths** or **heights** of all nodes?

```
Tips:
Depth of the root = 0
Height of a leaf = 0
```

- Q2:
- Is the sum of **depths** equal to the sum of **heights** of all nodes?
- Exercise: compute the sum of depths and heights of all nodes, respectively.

Tighter build-heap running time - continued



Assume a maximum tree of height h

1 node at height $h-0$

2 nodes at .... $h-1$

4 nodes ...... $h-2$

$\vdots$

2^i nodes .... $h-i$

$$S = \sum_{i=0 \text{ to } h} (h-i)\,2^i$$

$$S = h + 2(h-1) + 4(h-2) + 8(h-3) + \cdots + 2^{h-1}(1)$$

$$2S = 2h + 4(h-1) + 8(h-2) + \cdots + 2^{h}(1)$$

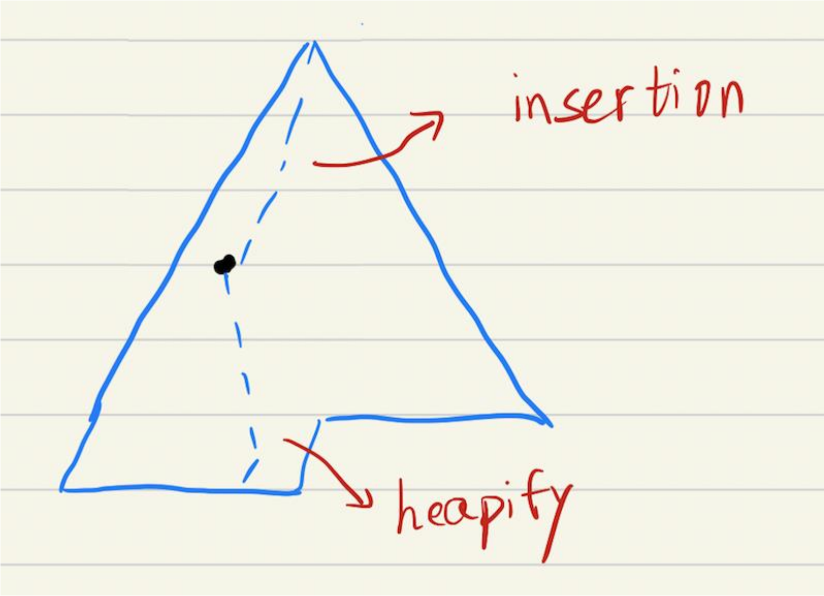
$$S = -h + \underbrace{2 + 4 + 8 + \cdots + 2^{h-1} + 2^{h}}$$

Makeheap $O(N)$

$$= -h + \frac{2(1-2^h)}{1-2} = (2^h-1)2 - h \le 2^{h+1} \sim N$$

Other operation(maxHeap)

- Increasekey(H,i,key): insertion
- Decreasekey(H,i,key): Heapify(H,i)
- Delete(H,i): Heapify(H,i)



In-class exercises

- 1. What is the heap after we perform extractMax on this input: {92, 80, 50, 70, 30, 8, 40, 6, 15, 12}
- 2. Is this following array a heap: {100, 80, 40, 50, 70, 2, 3}
- 3. What is the heap after we insert job with priority 100 to the current job system: {92, 80, 50, 70, 30, 8, 40, 6, 15, 12}?
- 4. How do you convert max-heap into min-heap?

Applications

1st Applications: Heap Sort

- Easy to find that, with the deletion operation: we can easily sort the whole array
- Strategy:
- Step1: Swap the top element with last element in the array
- Step2: minus the size of Heap(H) by 1
- Step3: Heapify(H,1) ,where the size = k-1
- recursively do step 1~3 until Heap_size=1

1st Applications: Heap Sort

```
Def Heap_sort(H)
        Build_Heap(H)                         //O(N)
        for i ~ length(H) downto 2           //O(N)
                swap(H[1],H[i])              //O(N)
                Heap_size(H)-=1              //O(N)
                Heapify(H,1)                 //O(N*log(N))
```

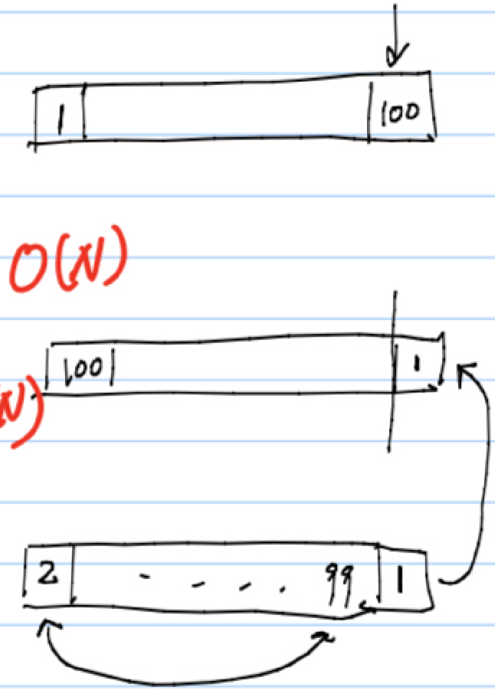
- Time ~ O(N*log(N))

In-class exercise 4.
Let H={100, 90, 95, 20, 30}. Apply Heap_sort(H). Show the output of Heapify(H,1) for i=length(H) to 2.

1st Applications: Heap Sort

* Heap Sort


```
HeapSort(H)
{ Build-Heap(H);                          O(N)
  for (i ← length[H] downto 2)            O(N)
      do swap H[1] ↔ H[i]                 O(N)
         heapsize(H)--                    O(N)
      ☆ heapify(H, 1)
         O(N logN)
}
```



1st Applications' extension

- Selection algorithm: find the K-th largest (smallest) elements
- Strategy: ExtractMin/Max for k times O(k*log(N))

- Heap is **NOT** designed for search