COMP90038
Algorithms and Complexity

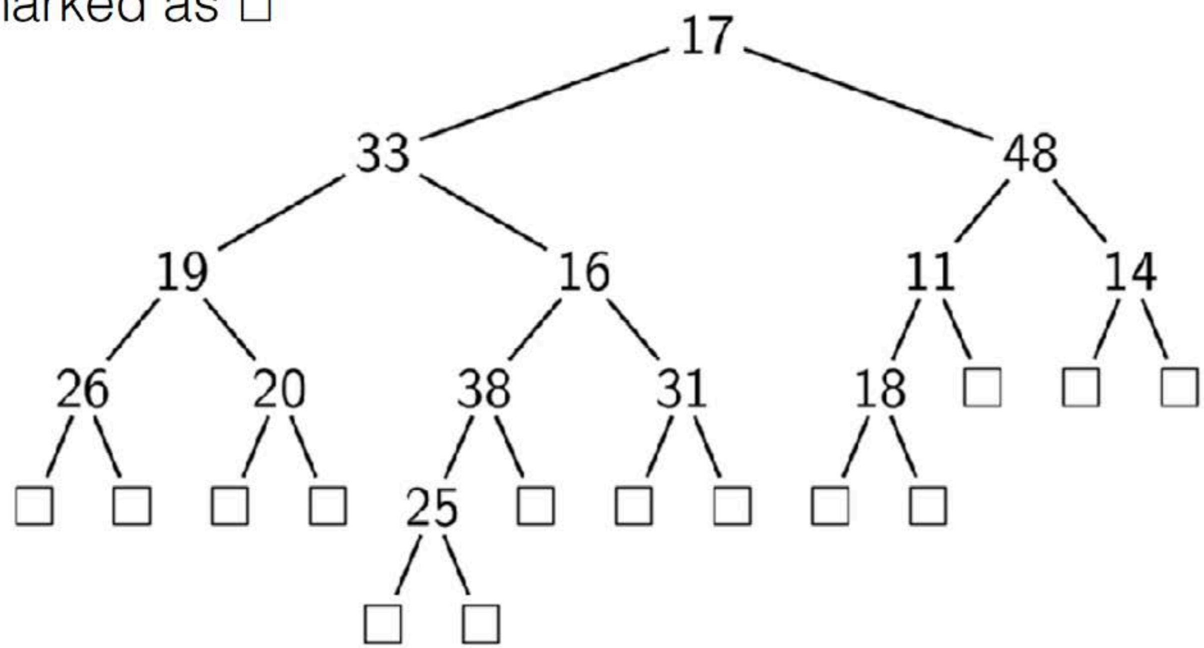
Lecture 13: Priority Queues, Heaps and Heapsort
(with thanks to Harald Søndergaard & Michael Kirley)

Casey Myers
Casey.Myers@unimelb.edu.au
David Caro Building (Physics) 274

Review from Lecture 12

Binary Trees

- An example of a **binary tree**, with empty subtrees marked as □



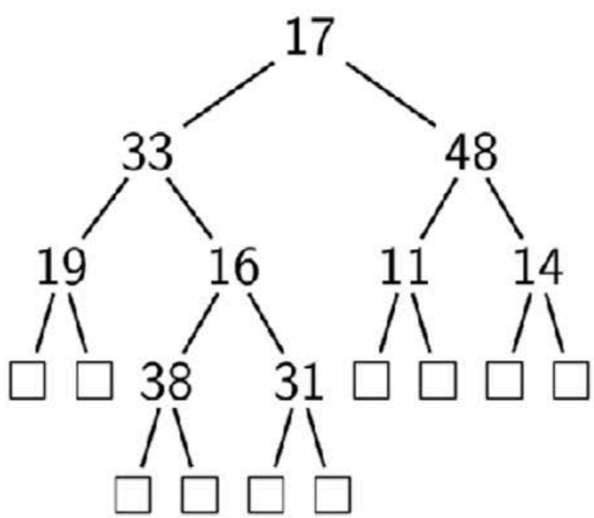
- This tree has **height** 4, the empty tree having height -1

Review from Lecture 12

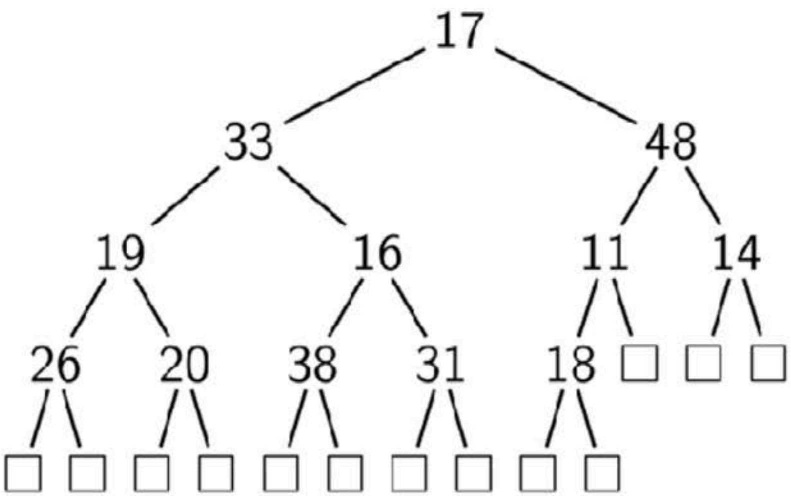
Binary Tree Concepts

THE UNIVERSITY OF MELBOURNE

- Special trees have their **external nodes** □ only at level *h* and *h*+1 for some *h*:



A **full** binary tree: Each node has 0 or 2 (non-empty) children.

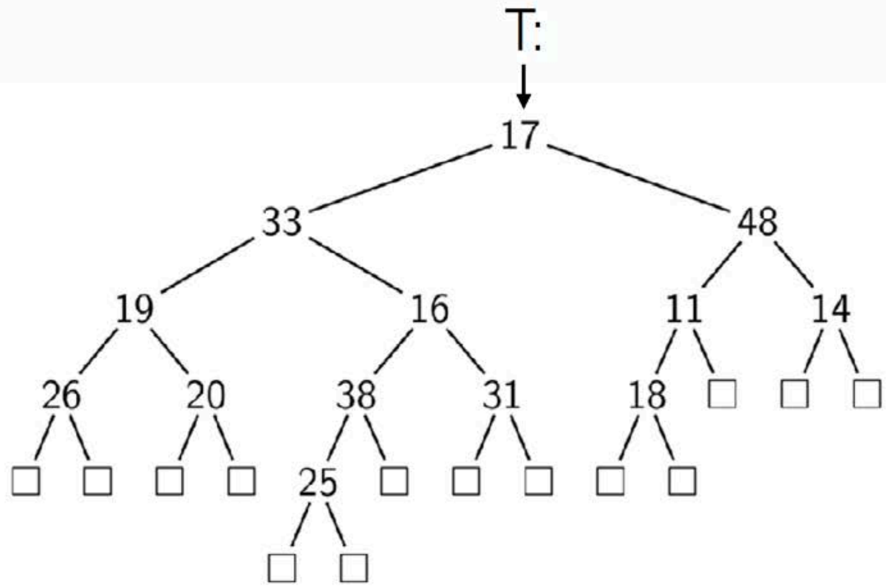


A **complete** tree: Each level filled left to right. (Every level except perhaps the last is completely filled.)

Review from Lecture 12

Calculating the Height

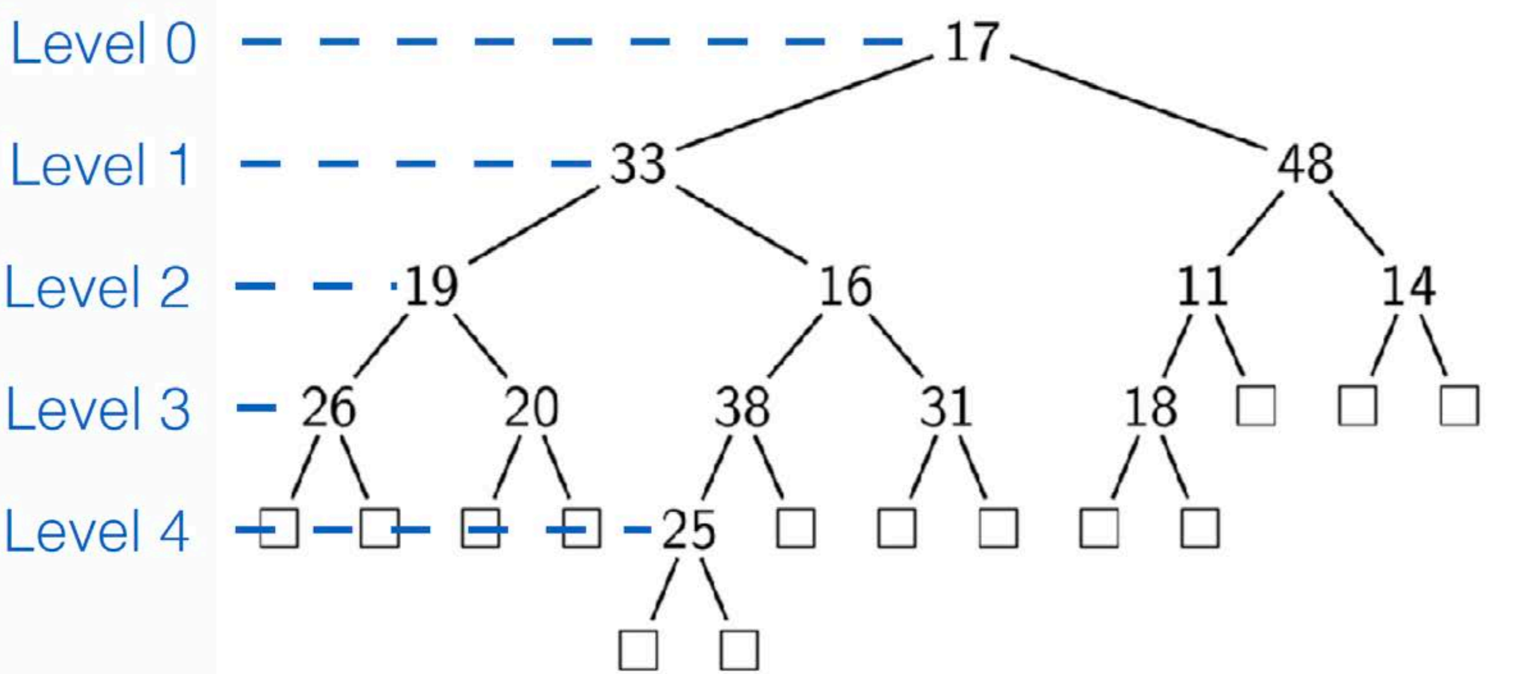
- Recursion is the natural way of calculating the **height**:




```
function HEIGHT(T)
    if T = null then
        return −1
    else
        return max(HEIGHT(T.left), HEIGHT(T.right)) + 1
```


Review from Lecture 12

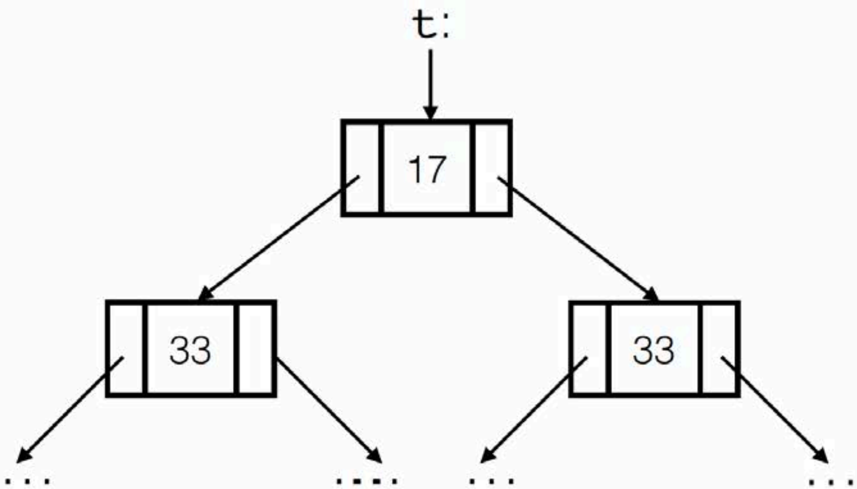
Levels and Height



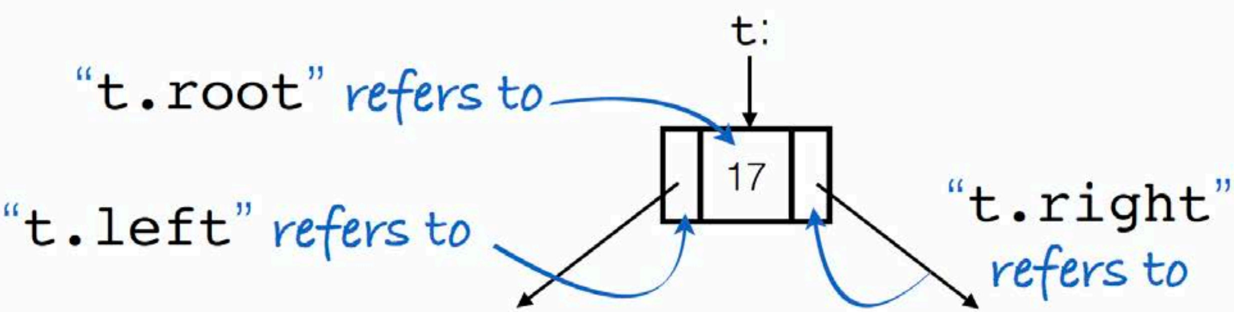
So the tree has **height** 4 (its **maximum level**)

Review from Lecture 12

Tree Terminology



t is (a pointer to) the **root node** of the tree



Review from Lecture 12

Binary Tree Traversal

THE UNIVERSITY OF MELBOURNE

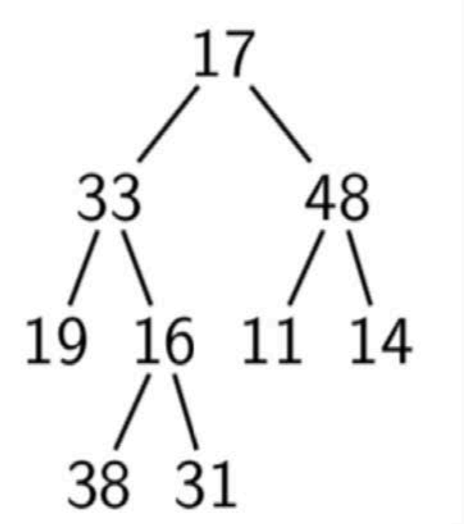
- **Preorder** traversal visits the root, then the left subtree, and finally the right subtree.
- **Inorder** traversal visits the left subtree, then the root, and finally the right subtree.
- **Postorder** traversal visits the left subtree, the right subtree, and finally the root.
- **Level-order** traversal visits the nodes, level by level, starting from the root.

Review from Lecture 12

Preorder Traversal

Visit order: 17 33 19 16 38 31 48 11 14

```
procedure PreorderTraverse(T)
    if T ≠ null then
        visit T.root
        PreorderTraverse(T.left)
        PreorderTraverse(T.right)
```



Call Stack

Heaps and Priority Queues

- The heap is a very useful data structure for priority queues, used in many algorithms.
- A priority queue is a set (or pool) of elements.
- An element is injected into the priority queue together with a priority (often the key value itself) and elements are ejected according to priority.
- We think of the heap as a partially ordered binary tree.
- Since it can be easily maintained as a complete tree, the standard implementation uses an array to represent the tree.

The Priority Queue

- As an abstract data type, the priority queue supports the following operations on a “pool” of elements (ordered by some linear order):
  - find an item with maximal priority
  - insert a new item with associated priority
  - test whether a priority queue is empty
  - eject the largest element.
- Other operations may be relevant, for example:
  - replace the maximal item with some new item
  - construct a priority queue from a list of items
  - join two priority queue.

Some Uses of Priority Queues

- Job scheduling done by you operating system. The OS will usually have a notion of "importance" of different jobs.

- (Discrete event) simulation of complex systems (like traffic, or weather). Here priorities are typically event times.

- Numerical computations involving floating point numbers. Here priorities are measures of computational "error".

Many sophisticated algorithms make essential use of priority queues (Huffman encoding and many shortest-path algorithms, for example).

Possible Implementations of the Priority Queue

Assume priority = key.

	INJECT(e)	EJECT()
Unsorted array or list		
Sorted array or list		
Heap	$O(\log n)$	$O(\log n)$

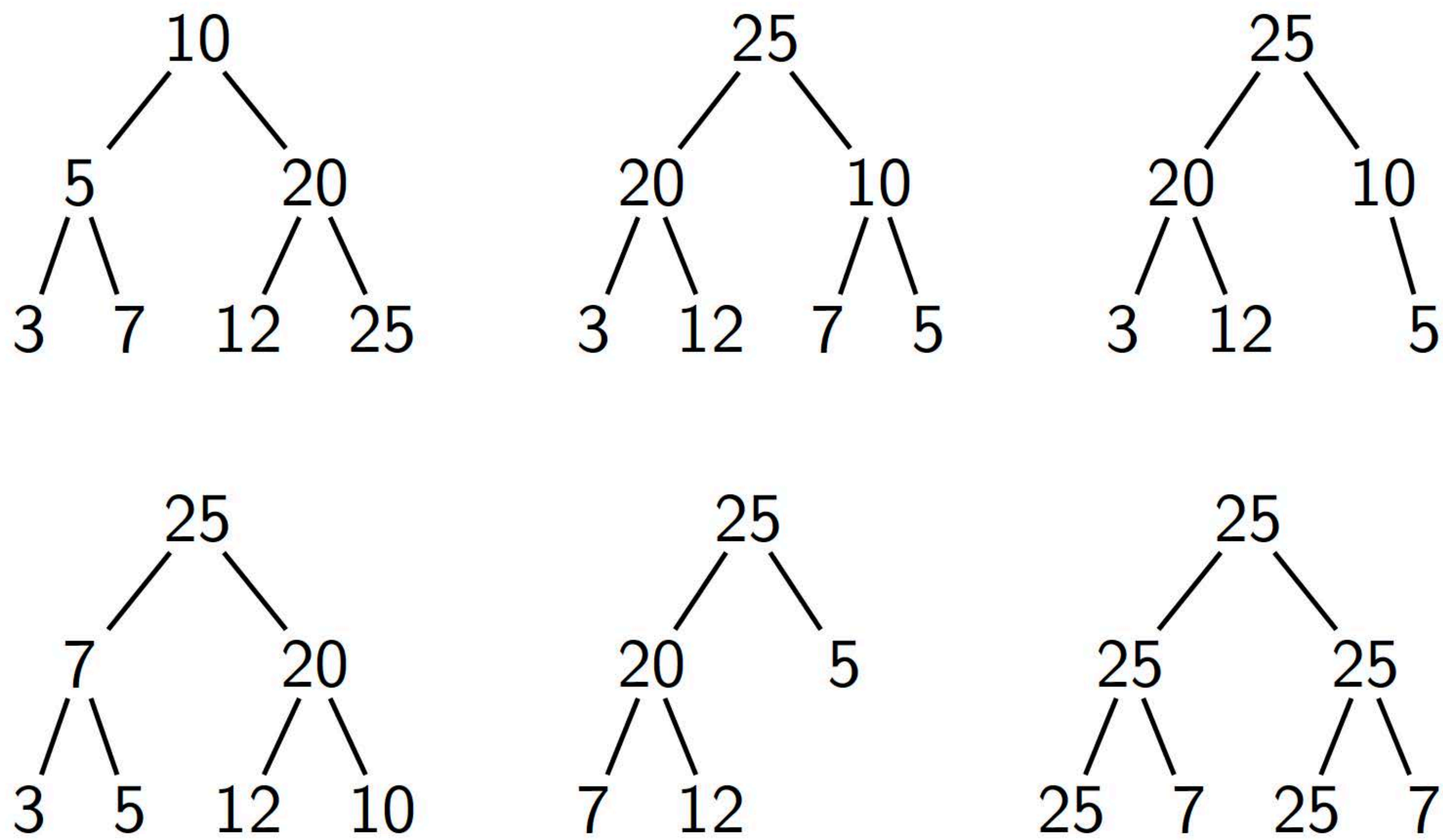
How is this accomplished?

The Heap

- A heap is a complete binary tree which satisfies the heap condition:
  - Each child has a priority (key) which is not greater than its parents.
- This guarantees the root of the tree is a maximal element.
- (Sometimes we talk about this as a max-heap—one can equally well have min-heaps, in which each child is no smaller that its parents.)

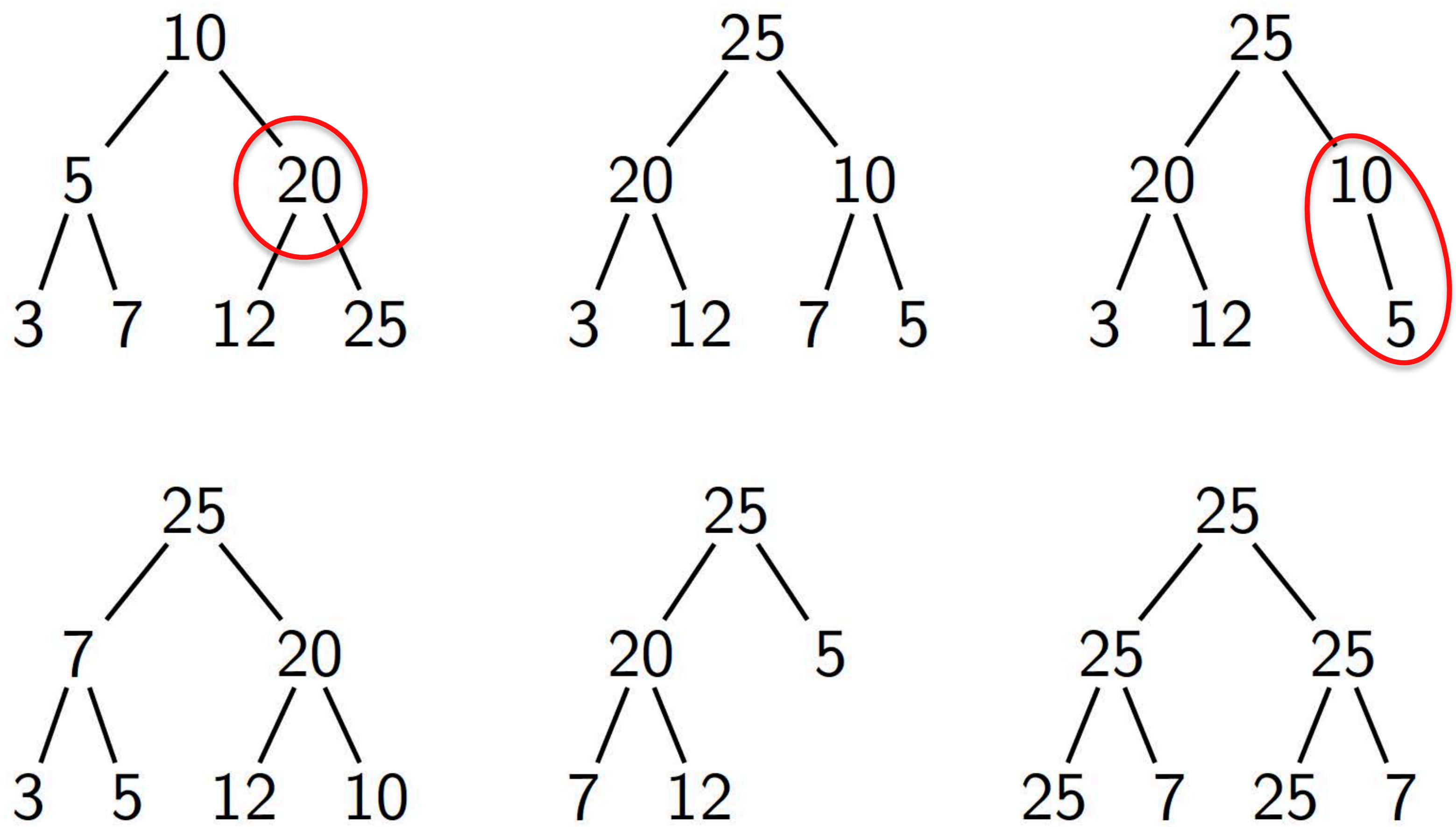
Heaps and Non-Heaps

Which of these are heaps?



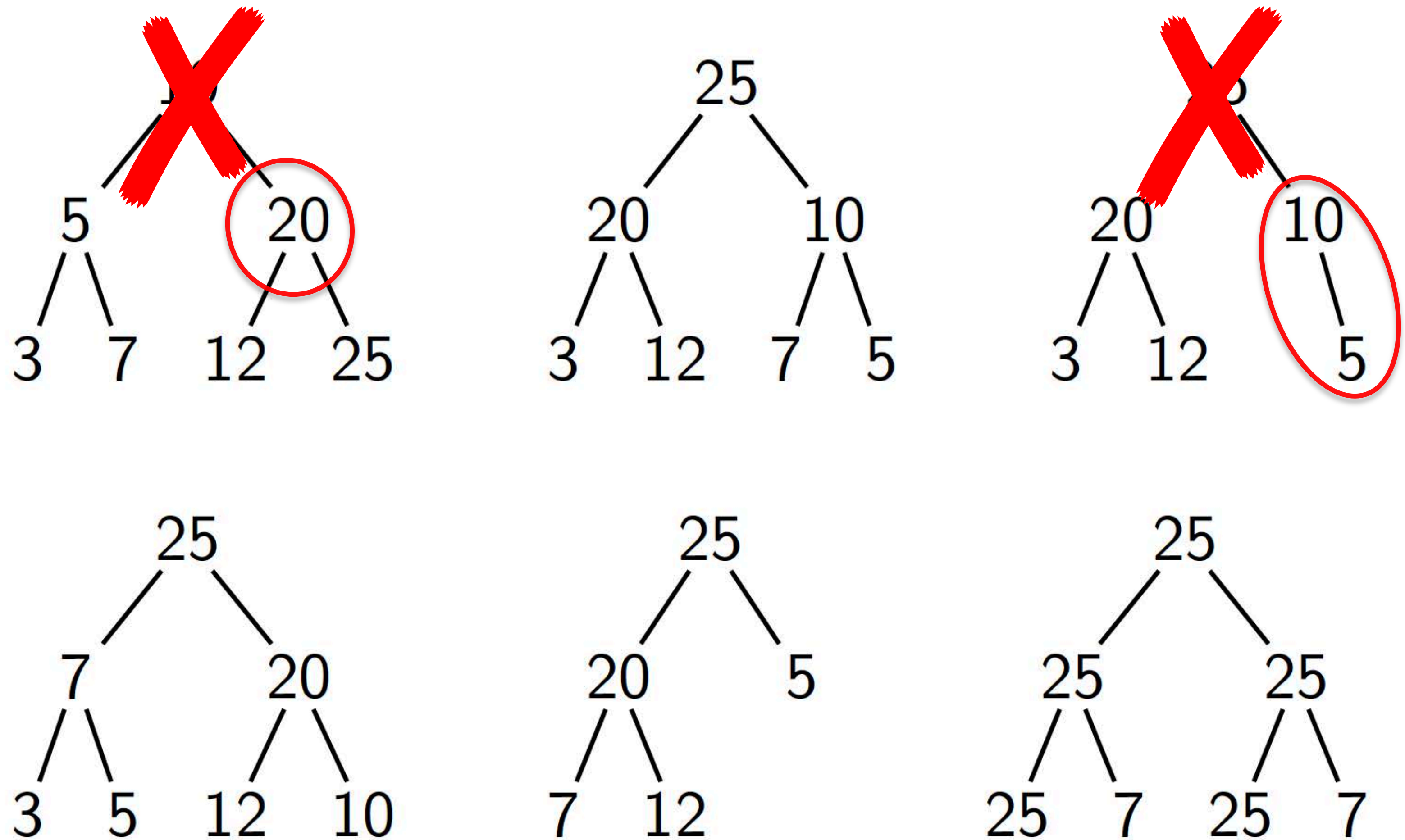
Heaps and Non-Heaps

Which of these are heaps?



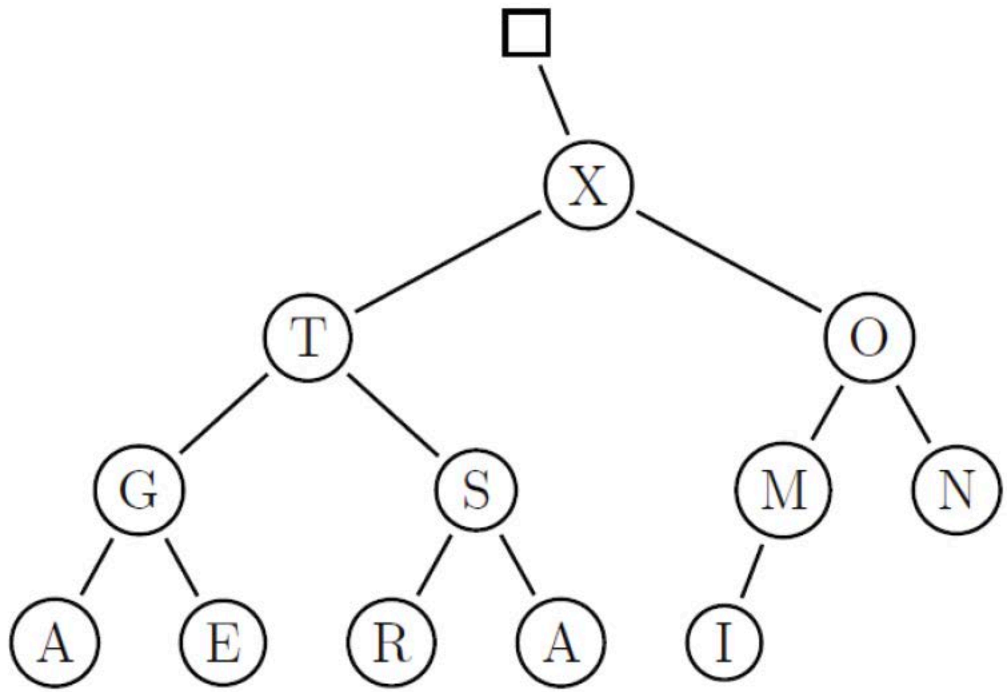
Heaps and Non-Heaps

Which of these are heaps?



Heaps and Arrays

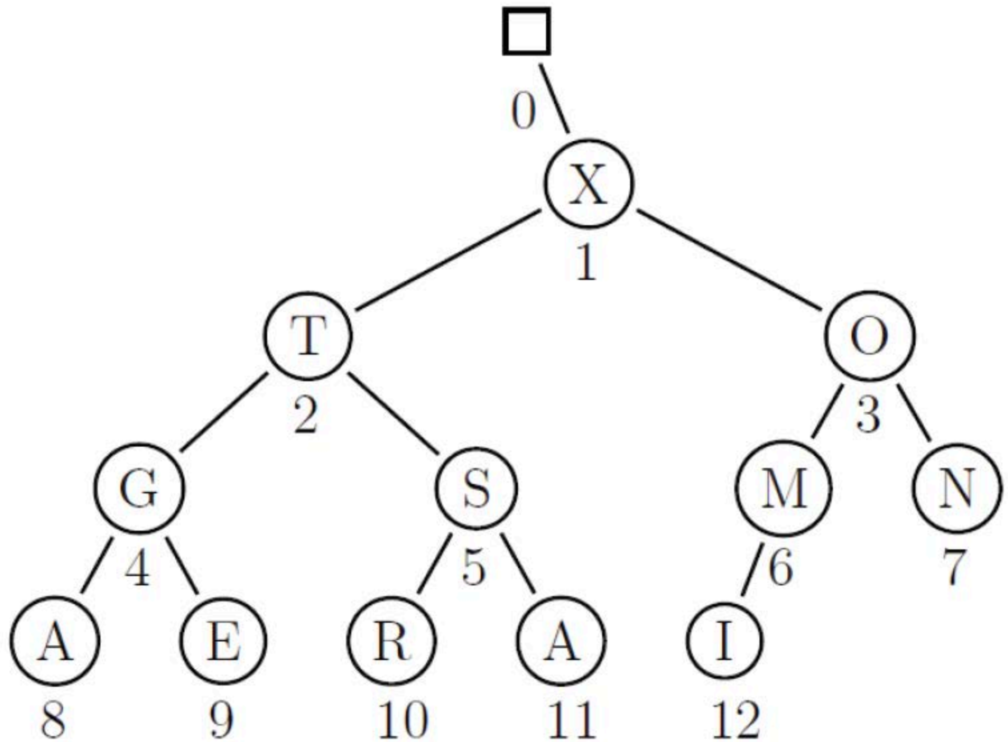
We can utilise the completeness of the tree and place its elements in level-order in an array H.



Heaps and Arrays

We can utilise the completeness of the tree and place its elements in level-order in an array H.

Note that the children of node i will be nodes 2i and 2i+1.

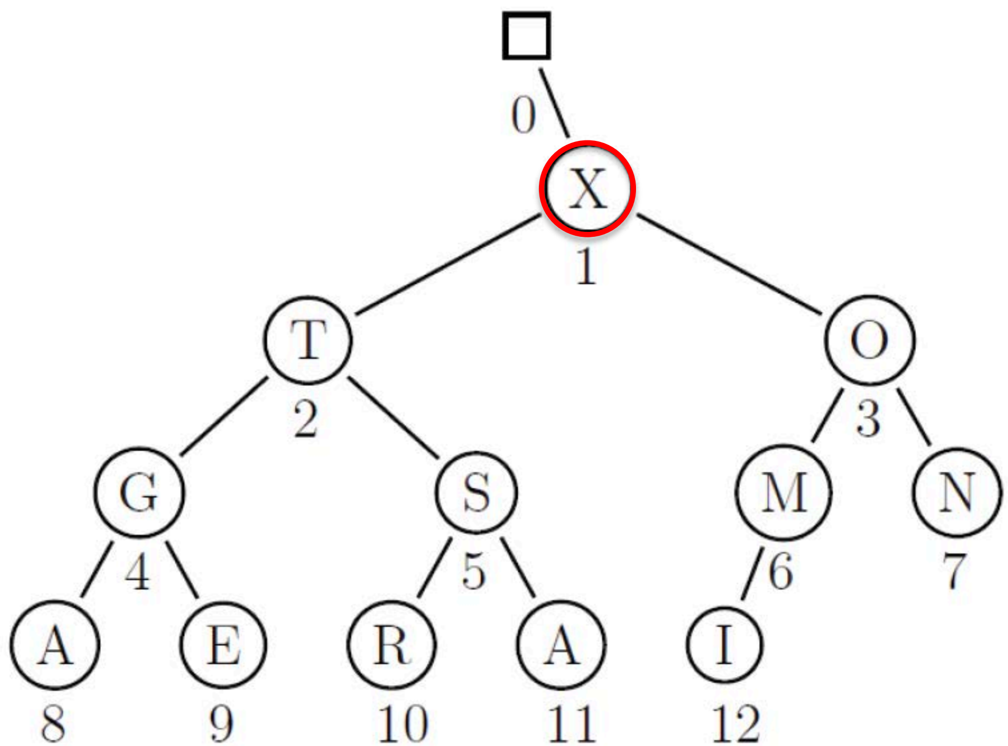


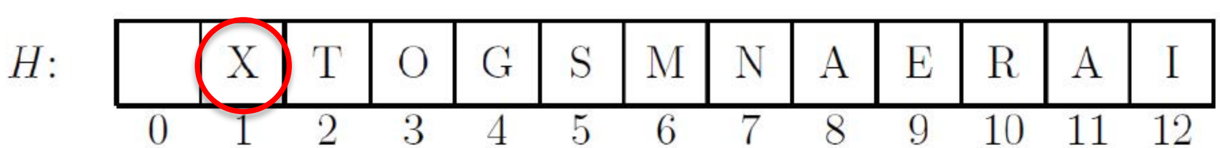
H:	0	1	2	3	4	5	6	7	8	9	10	11	12
		X	T	O	G	S	M	N	A	E	R	A	I

Heaps and Arrays

We can utilise the completeness of the tree and place its elements in level-order in an array H.

Note that the children of node i will be nodes 2i and 2i+1.

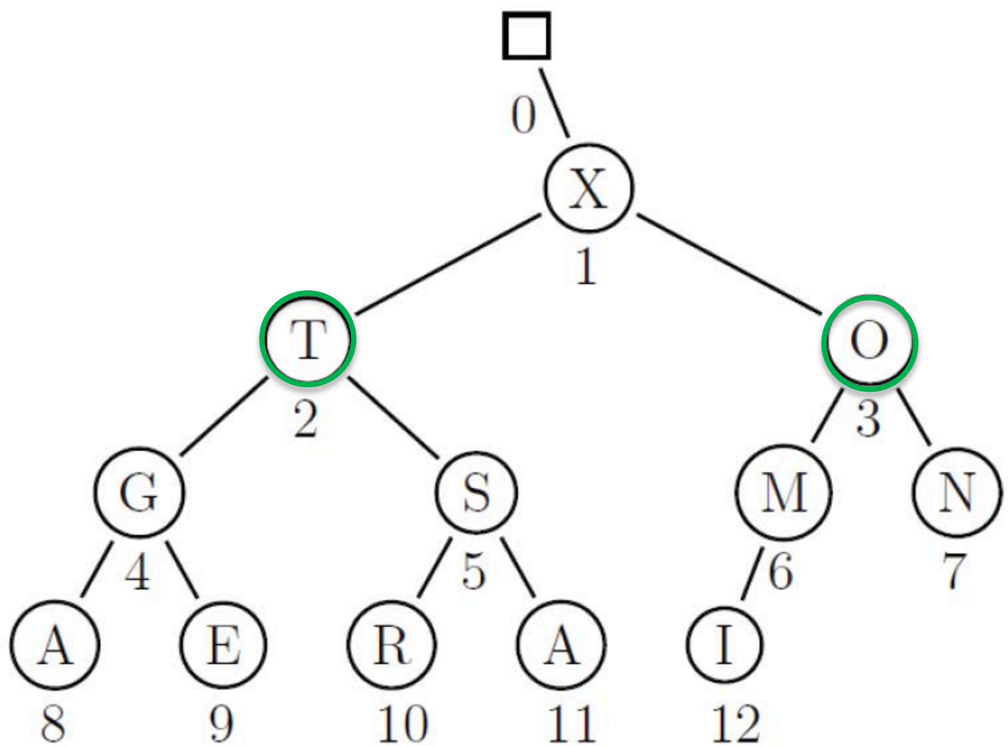




Heaps and Arrays

We can utilise the completeness of the tree and place its elements in level-order in an array H.

Note that the children of node i will be nodes 2i and 2i+1.

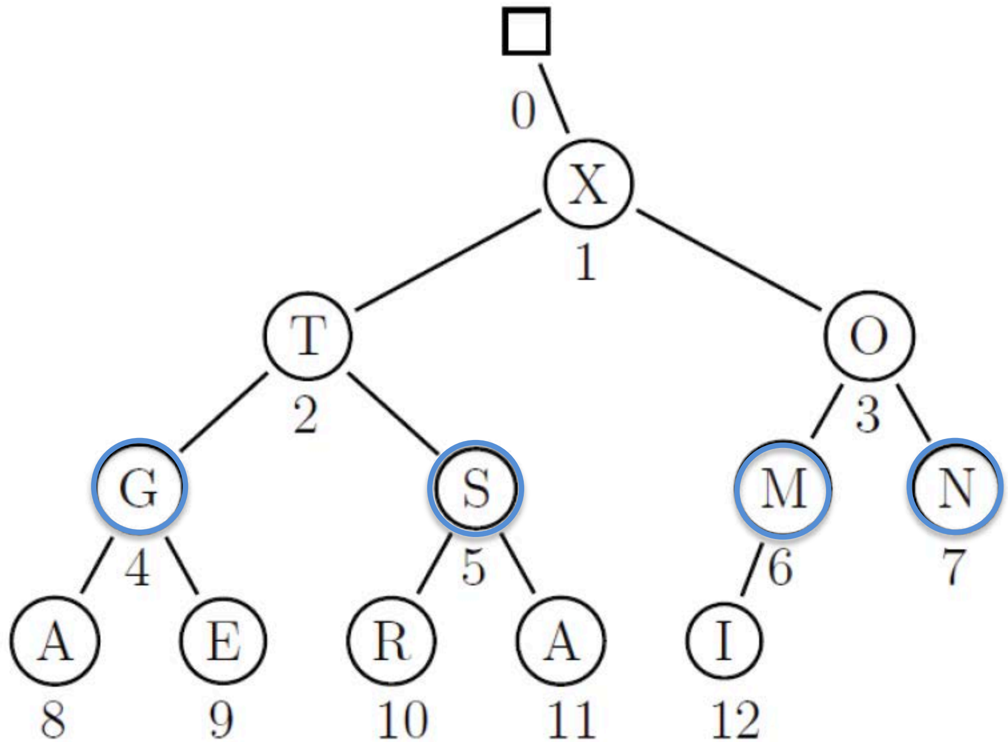


H:		X	T	O	G	S	M	N	A	E	R	A	I
	0	1	2	3	4	5	6	7	8	9	10	11	12

Heaps and Arrays

We can utilise the completeness of the tree and place its elements in level-order in an array H.

Note that the children of node i will be nodes 2i and 2i+1.

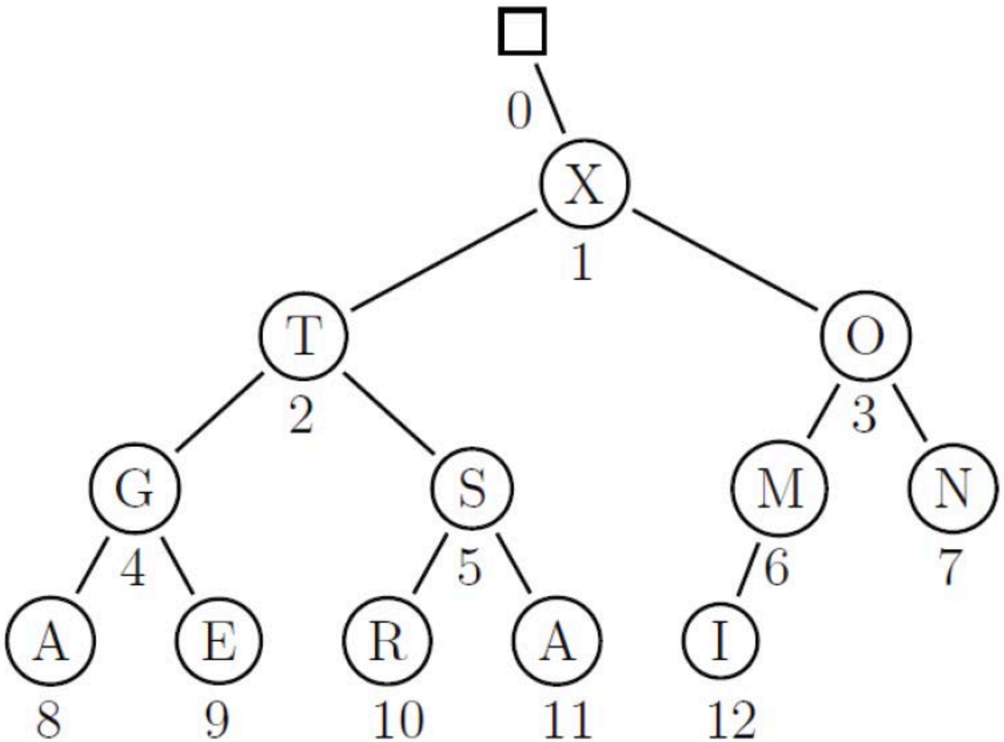


	0	1	2	3	4	5	6	7	8	9	10	11	12
H:		X	T	O	G	S	M	N	A	E	R	A	I

Heaps and Arrays

This way, the heap condition is very simple:

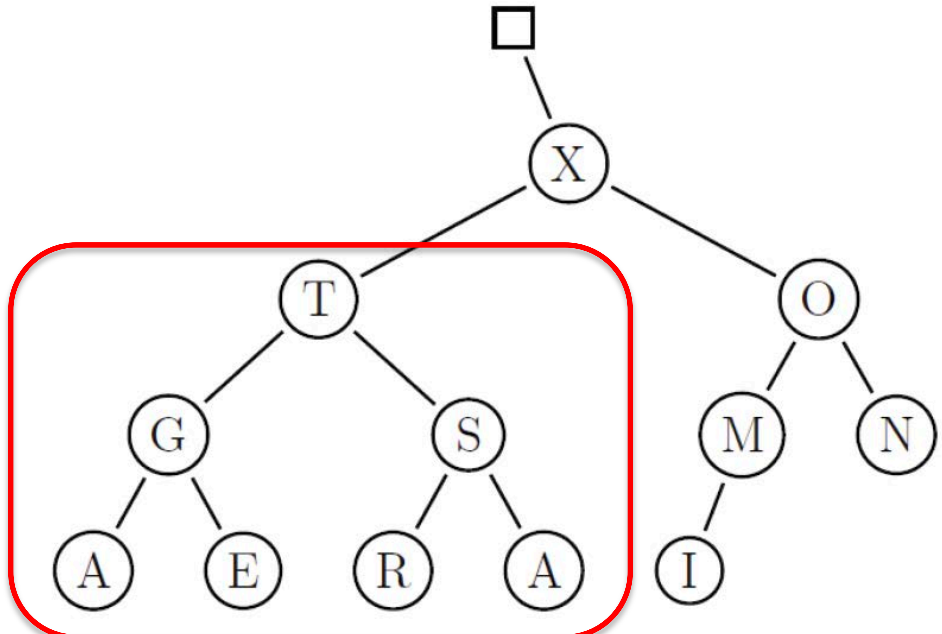
For all $i \in \{0, 1, \cdots, n\}$, we must have $H[i] \leq H[i/2]$



H:		X	T	O	G	S	M	N	A	E	R	A	I
	0	1	2	3	4	5	6	7	8	9	10	11	12

Properties of the Heap

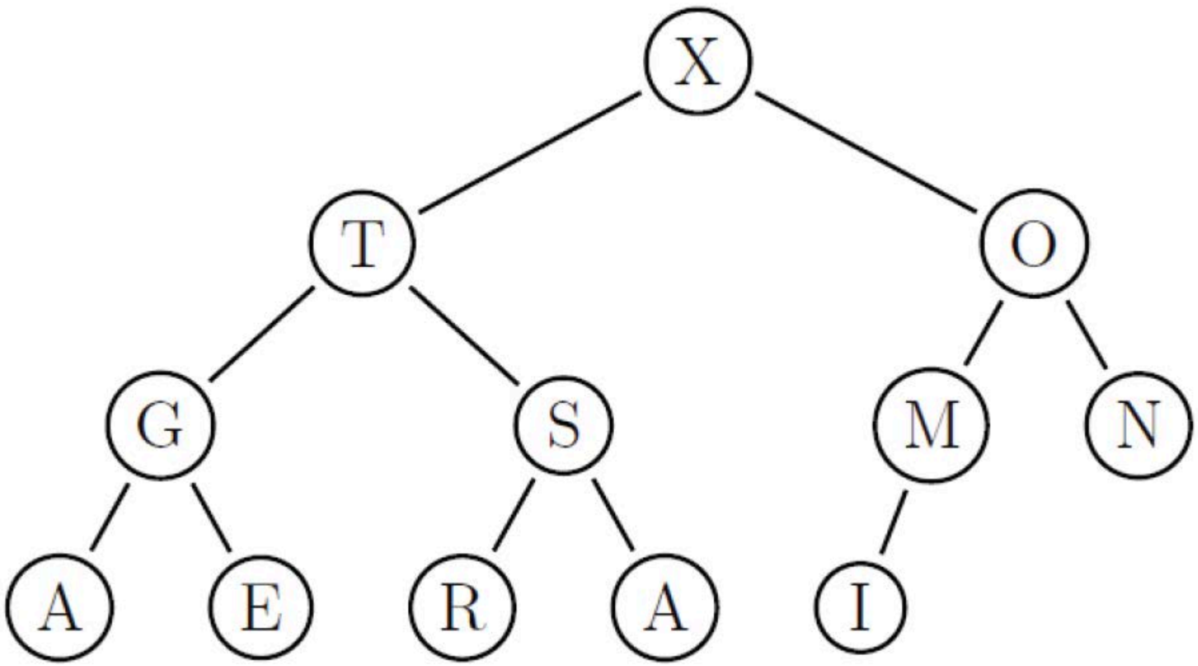
- The root of the tree H[1] holds a maximal item; the cost of EJECT is $O(1)$ plus time to restore the heap.



- The height of the heap is $\lfloor log_2 n \rfloor$.
- Each subtree is also a heap
- The children of node i are 2i and 2i+1.
- The nodes which happen to be parents are in array position 1 to $\lfloor n/2 \rfloor$.
- It is easier to understand heap operations if we think of the heap as a tree.

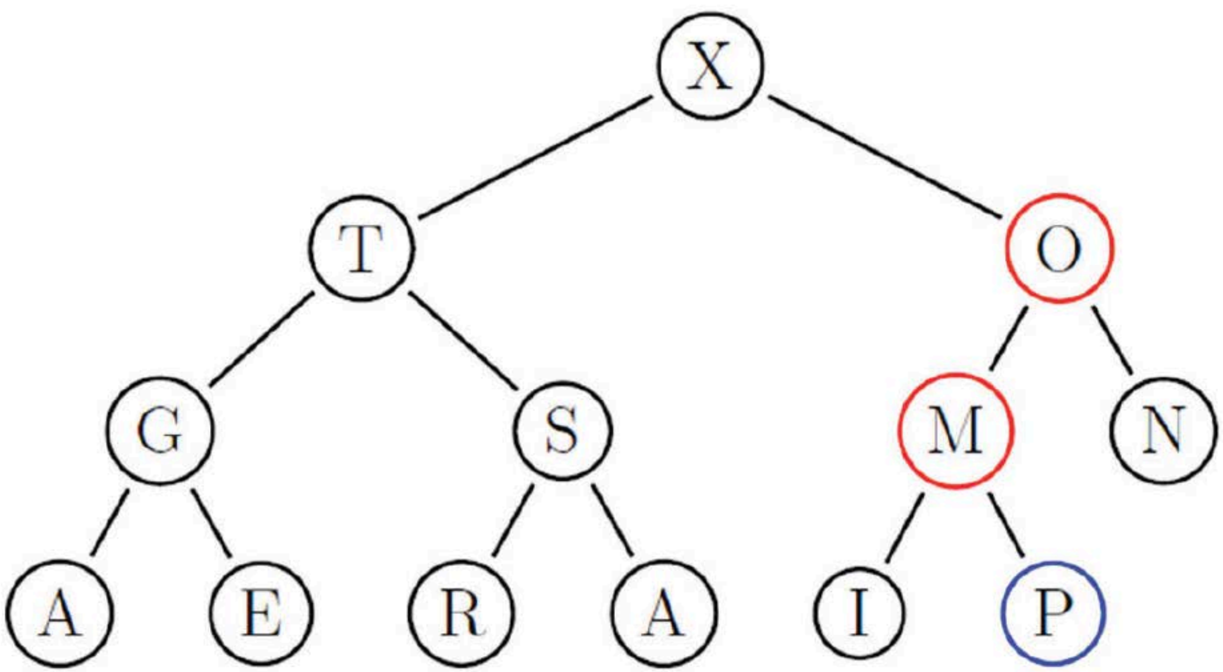
Injecting a New Item

- Place the new item at the end; then let it "climb up", repeatedly swapping with parents that are smaller.



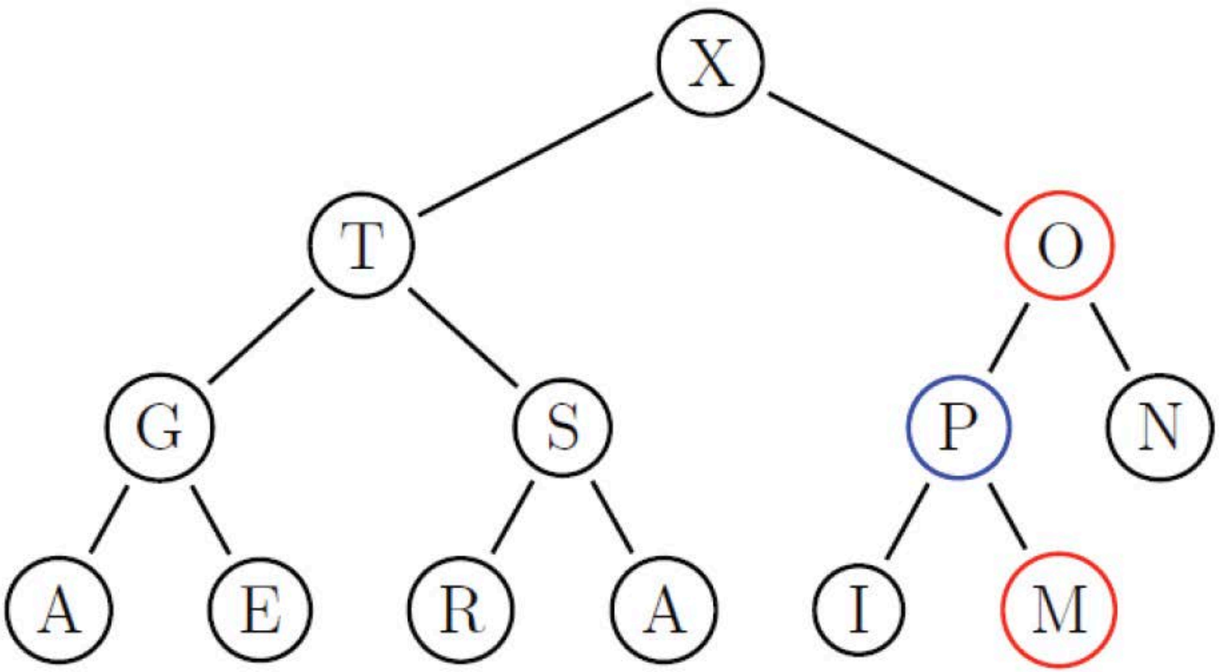
Injecting a New Item

- Place the new item at the end; then let it "climb up", repeatedly swapping with parents that are smaller.
- We want to inject "P".
- We place "P" at the end.



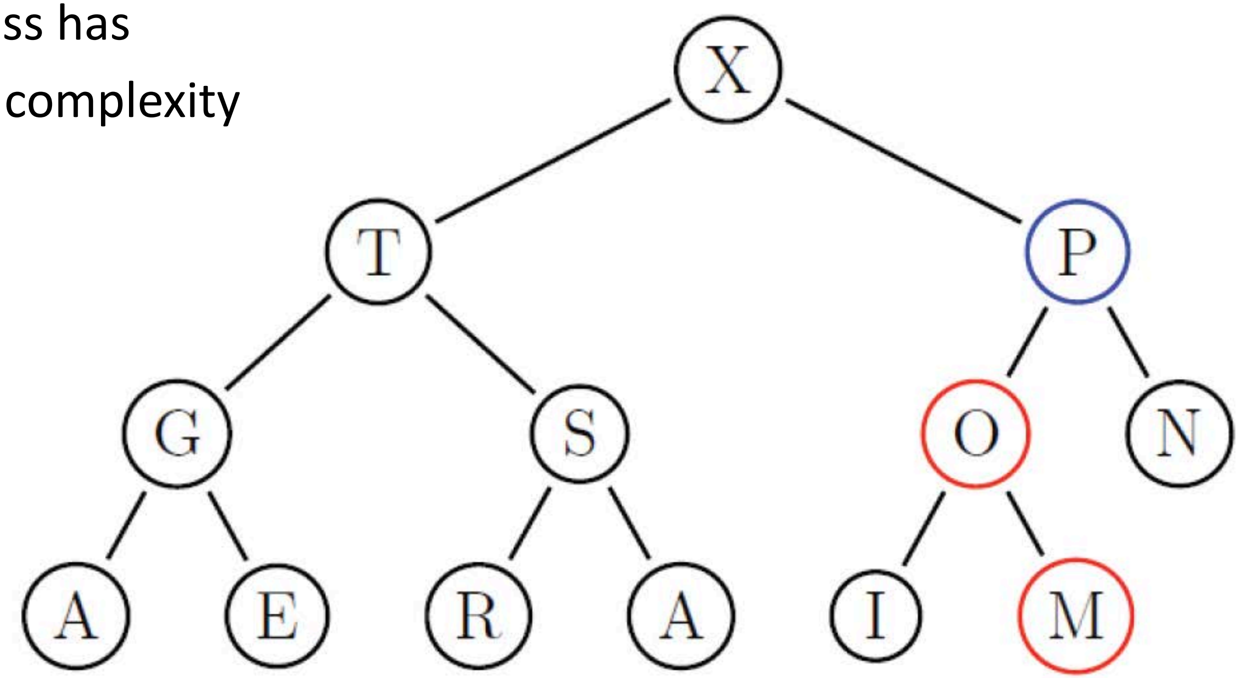
Injecting a New Item

- Place the new item at the end; then let it "climb up", repeatedly swapping with parents that are smaller.
- We want to inject "P".
- We place "P" at the end.
- We let it "climb up", swapping with smaller parents ("M" and "O")



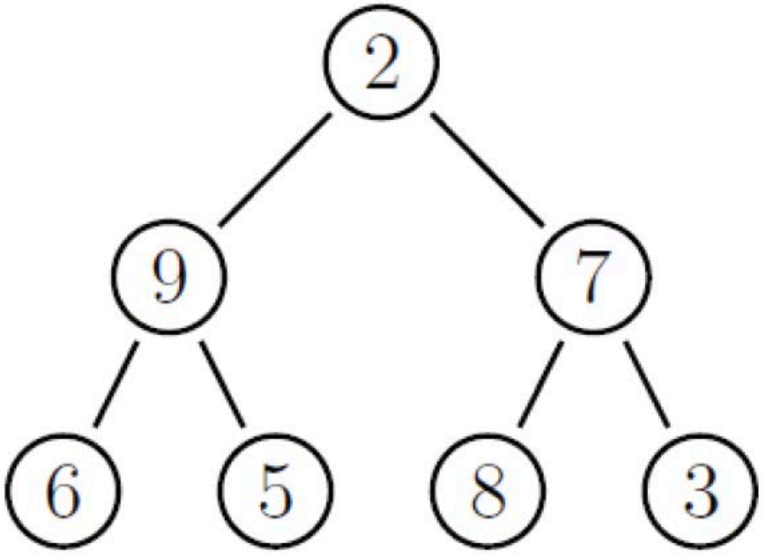
Injecting a New Item

- Place the new item at the end; then let it "climb up", repeatedly swapping with parents that are smaller.
- We want to inject "P".
- We place "P" at the end.
- We let it "climb up", swapping with smaller parents ("M" and "O")
- This process has $O(\log n)$ complexity



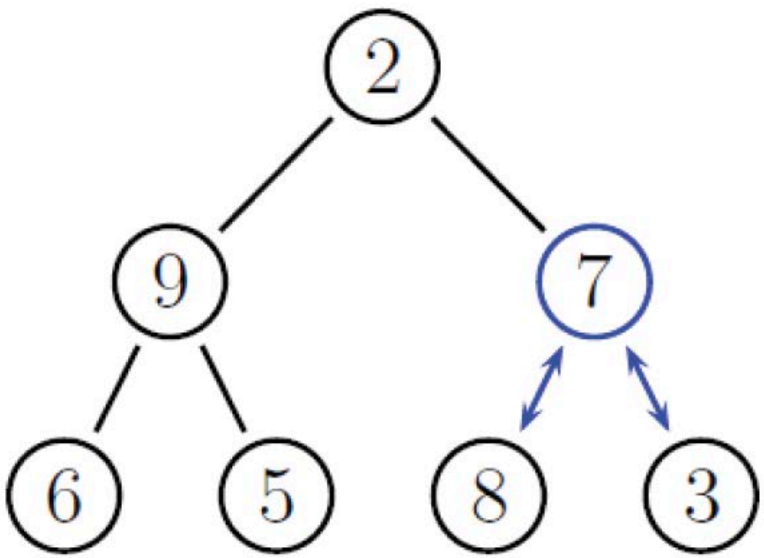
Building a Heap Bottom-Up

- To construct a heap from an arbitrary set of elements, we can just use the inject operation repeatedly. But the construction cost will be $n \log n$.



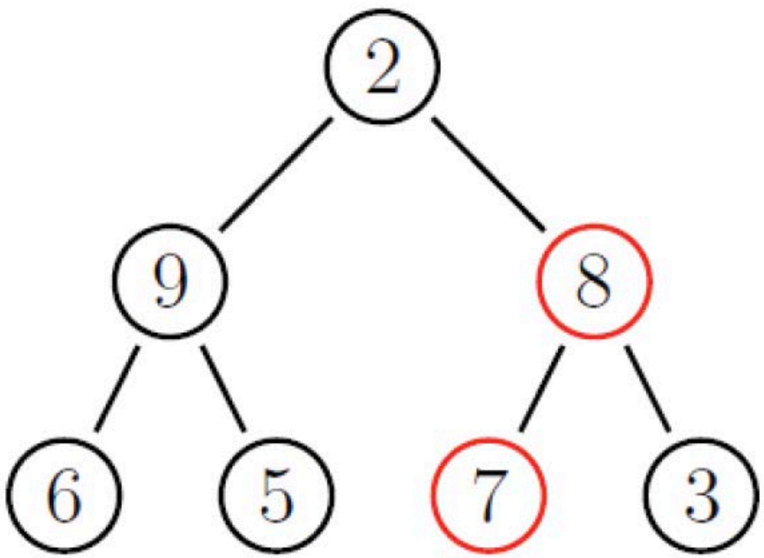
Building a Heap Bottom-Up: Sifting Down

- To construct a heap from an arbitrary set of elements, we can just use the inject operation repeatedly. But the construction cost will be $n \log n$.
- But there is a better way.
- Start with the last parent and move backwards, in level-order.
- Whenever a parent is found to be out of order, let it “sift down” until both children are smaller:



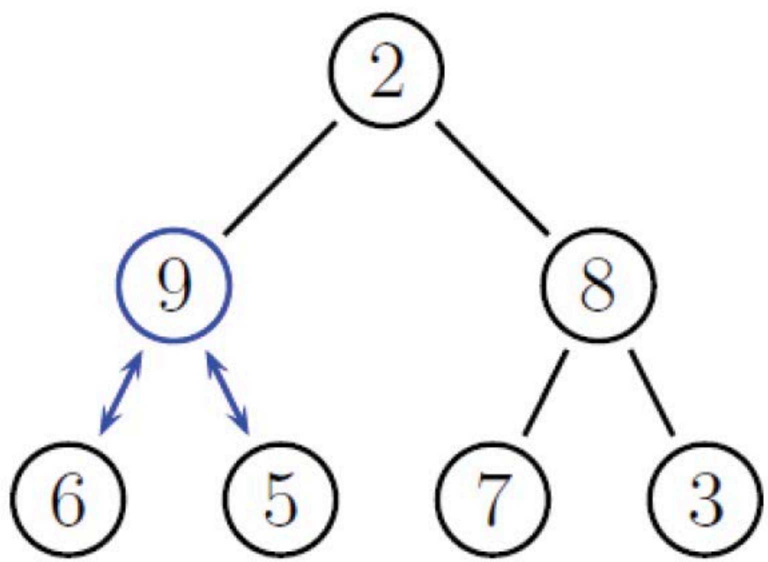
Building a Heap Bottom-Up: Sifting Down

- To construct a heap from an arbitrary set of elements, we can just use the inject operation repeatedly. But the construction cost will be $n \log n$.
- But there is a better way.
- Start with the last parent and move backwards, in level-order.
- Whenever a parent is found to be out of order, let it "sift down" until both children are smaller:



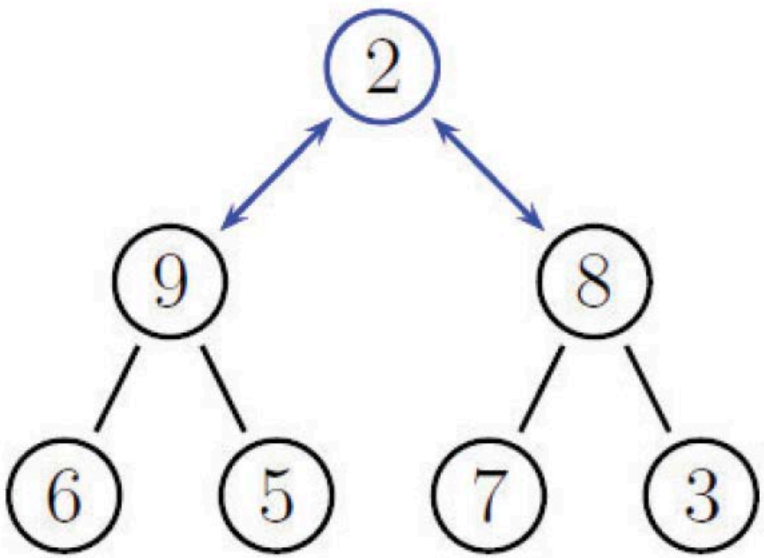
Building a Heap Bottom-Up: Sifting Down

- To construct a heap from an arbitrary set of elements, we can just use the inject operation repeatedly. But the construction cost will be $n \log n$.
- But there is a better way.
- Start with the last parent and move backwards, in level-order.
- Whenever a parent is found to be out of order, let it “sift down” until both children are smaller:



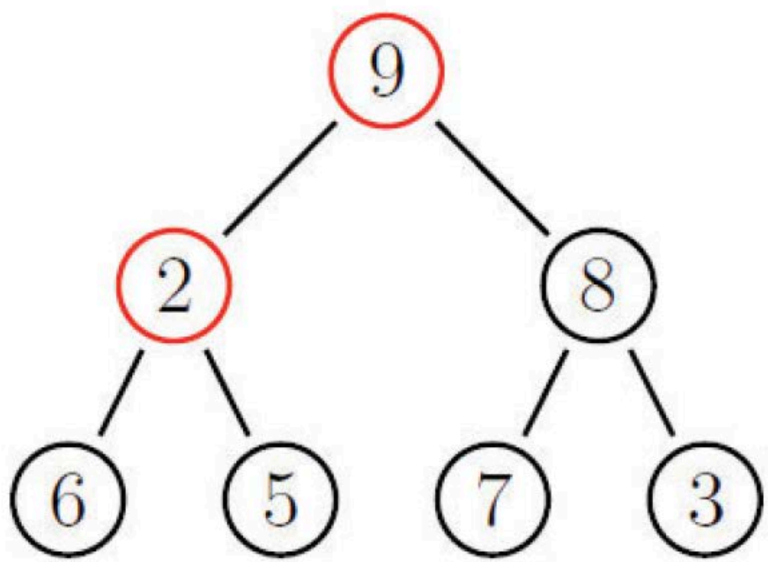
Building a Heap Bottom-Up: Sifting Down

- To construct a heap from an arbitrary set of elements, we can just use the inject operation repeatedly. But the construction cost will be $n \log n$.
- But there is a better way.
- Start with the last parent and move backwards, in level-order.
- Whenever a parent is found to be out of order, let it "sift down" until both children are smaller:



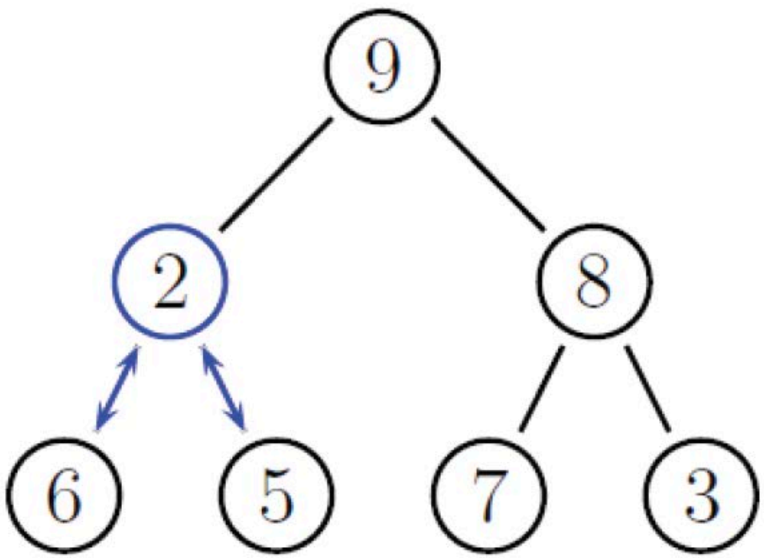
Building a Heap Bottom-Up: Sifting Down

- To construct a heap from an arbitrary set of elements, we can just use the inject operation repeatedly. But the construction cost will be $n \log n$.
- But there is a better way.
- Start with the last parent and move backwards, in level-order.
- Whenever a parent is found to be out of order, let it “sift down” until both children are smaller:



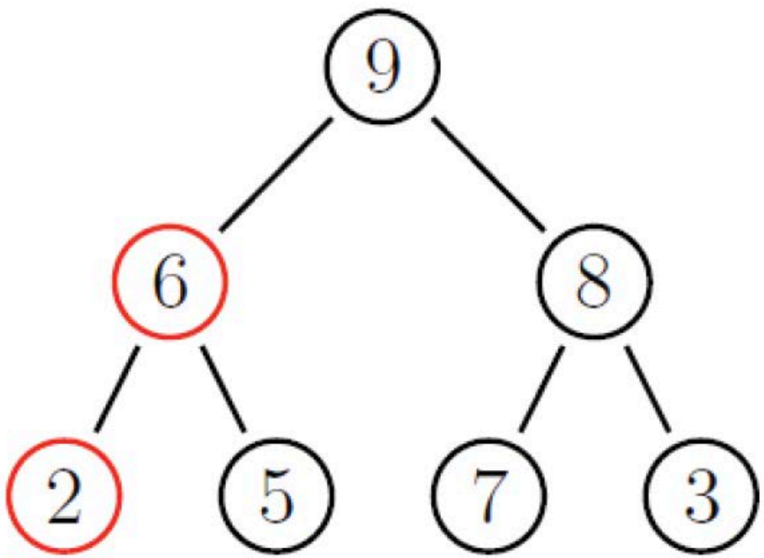
Building a Heap Bottom-Up: Sifting Down

- To construct a heap from an arbitrary set of elements, we can just use the inject operation repeatedly. But the construction cost will be $n \log n$.
- But there is a better way.
- Start with the last parent and move backwards, in level-order.
- Whenever a parent is found to be out of order, let it "sift down" until both children are smaller:



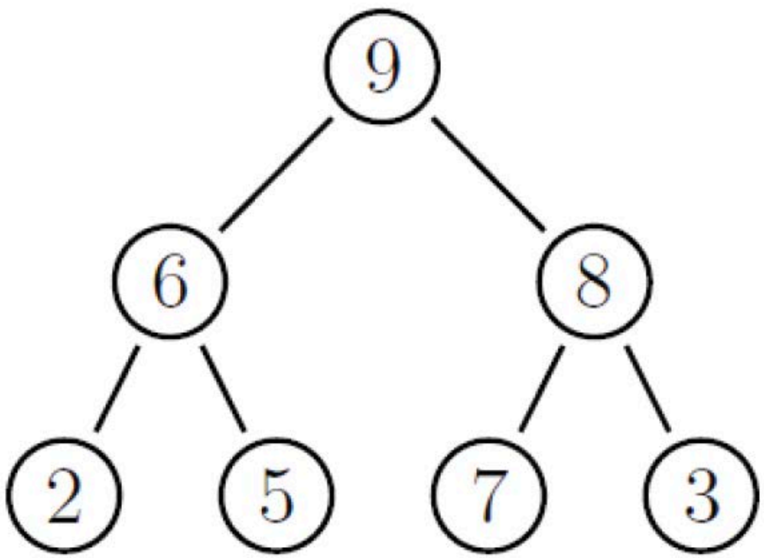
Building a Heap Bottom-Up: Sifting Down

- To construct a heap from an arbitrary set of elements, we can just use the inject operation repeatedly. But the construction cost will be $n \log n$.
- But there is a better way.
- Start with the last parent and move backwards, in level-order.
- Whenever a parent is found to be out of order, let it “sift down” until both children are smaller:



Building a Heap Bottom-Up: Sifting Down

- To construct a heap from an arbitrary set of elements, we can just use the inject operation repeatedly. But the construction cost will be $n \log n$.
- But there is a better way.
- Start with the last parent and move backwards, in level-order.
- Whenever a parent is found to be out of order, let it "sift down" until both children are smaller:



Algorithm to Turn $H[1, \cdots, n]$ into a Heap, Bottom-Up

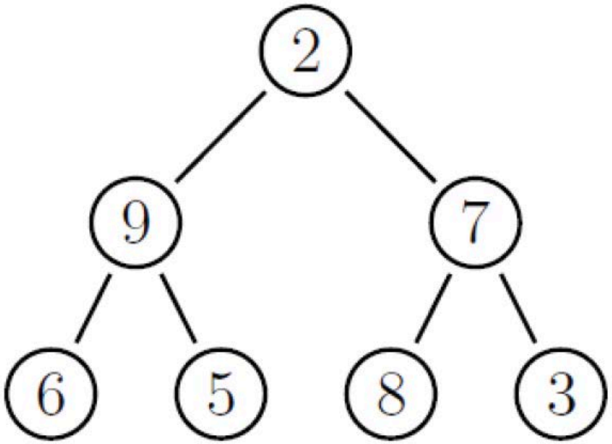
```
for i ← ⌊n/2⌋ downto 1 do
    k ← i
    v ← H[k]
    heap ← False
    while not heap and 2 × k ≤ n do
        j ← 2 × k
        if j < n then
            if H[j] < H[j + 1] then
                j ← j + 1
        if v ≥ H[j] then
            heap ← True
        else
            H[k] ← H[j]
            k ← j
    H[k] ← v
```

Algorithm to Turn $H[1, \cdots, n]$ into a Heap, Bottom-Up

for $i \leftarrow \lfloor n/2 \rfloor$ downto 1 **do**
    $k \leftarrow i$
    $v \leftarrow H[k]$
    $heap \leftarrow False$
    while not $heap$ **and** $2 \times k \leq n$ **do**
        $j \leftarrow 2 \times k$ ← j is k's left child
        if $j < n$ **then**
            if $H[j] < H[j+1]$ **then**
                $j \leftarrow j + 1$ ← j is k's largest child
        if $v \geq H[j]$ **then**
            $heap \leftarrow True$
        else
            $H[k] \leftarrow H[j]$ ← Promote H[j]
            $k \leftarrow j$
    $H[k] \leftarrow v$

Algorithm to Turn $H[1, \cdots, n]$ into a Heap, Bottom-Up

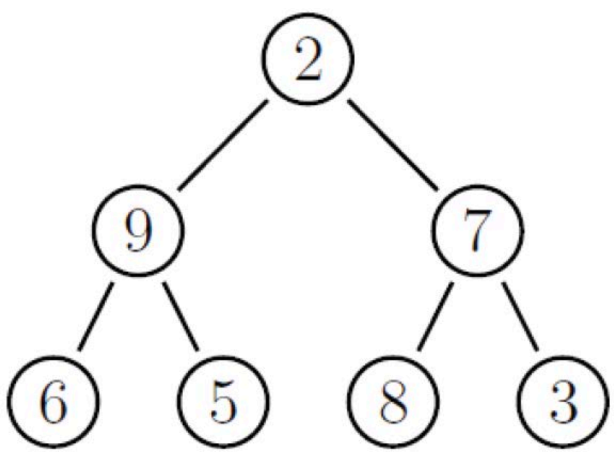
```
for i ← ⌊n/2⌋ downto 1 do
    k ← i
    v ← H[k]
    heap ← False
    while not heap and 2 × k ≤ n do
        j ← 2 × k
        if j < n then
            if H[j] < H[j + 1] then
                j ← j + 1
        if v ≥ H[j] then
            heap ← True
        else
            H[k] ← H[j]
            k ← j
    H[k] ← v
```



$H[1, \cdots, 7]$	[2 9 7 6 5 8 3]
n	7
i	
k	
v	
j	
$H[k]$	
$H[j]$	
$H[j+1]$	
not $HEAP$	
$2 \times k \leq n$	
$j < n$	
$H[j] < H[j+1]$	
$v \geq H[j]$	

Algorithm to Turn $H[1, \cdots, n]$ into a Heap, Bottom-Up

```
for i ← ⌊n/2⌋ downto 1 do
    k ← i
    v ← H[k]
    heap ← False
    while not heap and 2 × k ≤ n do
        j ← 2 × k
        if j < n then
            if H[j] < H[j + 1] then
                j ← j + 1
        if v ≥ H[j] then
            heap ← True
        else
            H[k] ← H[j]
            k ← j
    H[k] ← v
```

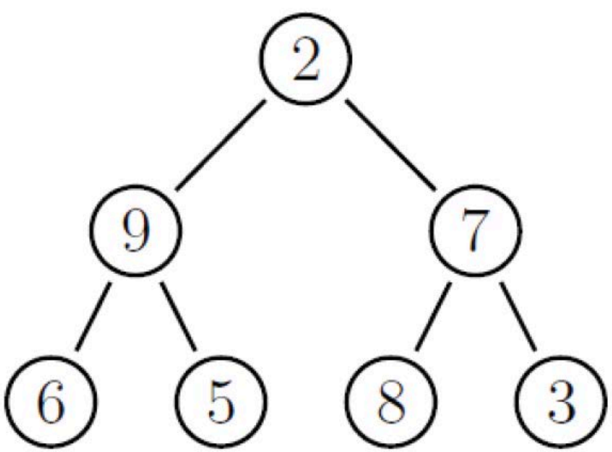


$H[1, \cdots, 7]$	[2 9 7 6 5 8 3]
n	7
i	3
k	3
v	7
j	
$H[k]$	7
$H[j]$	
$H[j+1]$	
not $HEAP$	$TRUE$
$2 \times k \leq n$	$TRUE$
$j < n$	
$H[j] < H[j+1]$	
$v \geq H[j]$	

Algorithm to Turn $H[1, \cdots, n]$ into a Heap, Bottom-Up

```
for i ← ⌊n/2⌋ downto 1 do
    k ← i
    v ← H[k]
    heap ← False
    while not heap and 2 × k ≤ n do
        j ← 2 × k
        if j < n then
            if H[j] < H[j + 1] then
                j ← j + 1
        if v ≥ H[j] then
            heap ← True
        else
            H[k] ← H[j]
            k ← j
    H[k] ← v
```

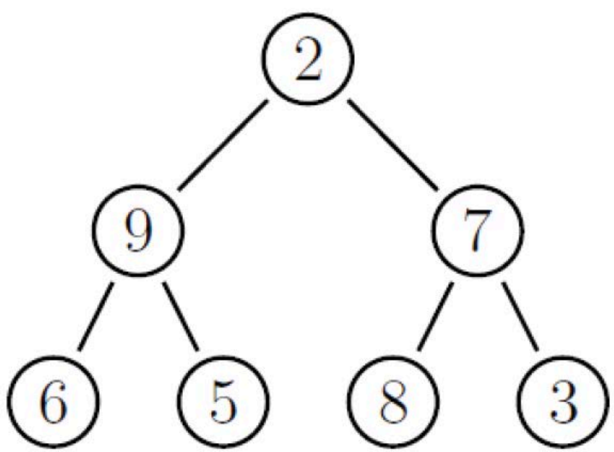
$H[1, \cdots, 7]$	[2 9 7 6 5 8 3]
n	7
i	3
k	3
v	7
j	6
$H[k]$	7
$H[j]$	8
$H[j+1]$	3
not $HEAP$	$TRUE$
$2 \times k \le n$	$TRUE$
$j < n$	$TRUE$
$H[j] < H[j+1]$	
$v \geq H[j]$	



Algorithm to Turn $H[1, \cdots, n]$ into a Heap, Bottom-Up

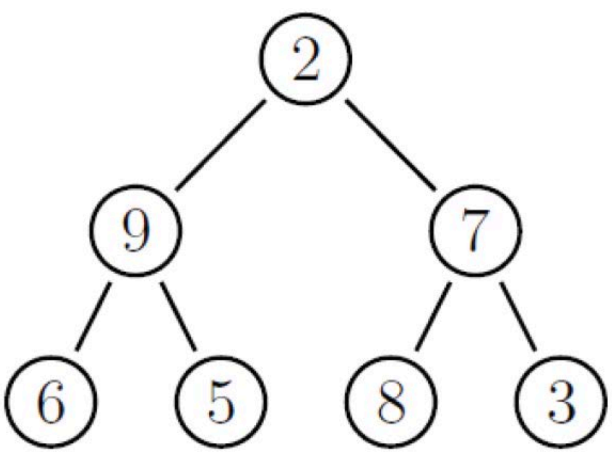
for $i \leftarrow \lfloor n/2 \rfloor$ downto 1 **do**
    $k \leftarrow i$
    $v \leftarrow H[k]$
    $heap \leftarrow False$
    while not $heap$ **and** $2 \times k \leq n$ **do**
        $j \leftarrow 2 \times k$
        if $j < n$ **then**
            if $H[j] < H[j+1]$ **then**
                $j \leftarrow j + 1$
        if $v \geq H[j]$ **then**
            $heap \leftarrow True$
        else
            $H[k] \leftarrow H[j]$
            $k \leftarrow j$
    $H[k] \leftarrow v$

$H[1, \cdots, 7]$	[2 9 7 6 5 8 3]
n	7
i	3
k	3
v	7
j	6
$H[k]$	7
$H[j]$	8
$H[j+1]$	3
not $HEAP$	$TRUE$
$2 \times k \leq n$	$TRUE$
$j < n$	$TRUE$
$H[j] < H[j+1]$	$FALSE$
$v \geq H[j]$	



Algorithm to Turn $H[1, \cdots, n]$ into a Heap, Bottom-Up

for $i \leftarrow \lfloor n/2 \rfloor$ downto 1 **do**
  $k \leftarrow i$
  $v \leftarrow H[k]$
  $heap \leftarrow False$
  while not $heap$ **and** $2 \times k \leq n$ **do**
    $j \leftarrow 2 \times k$
    if $j < n$ **then**
      if $H[j] < H[j+1]$ **then**
        $j \leftarrow j + 1$
    if $v \geq H[j]$ **then**
      $heap \leftarrow True$
    else
      $H[k] \leftarrow H[j]$
      $k \leftarrow j$
  $H[k] \leftarrow v$

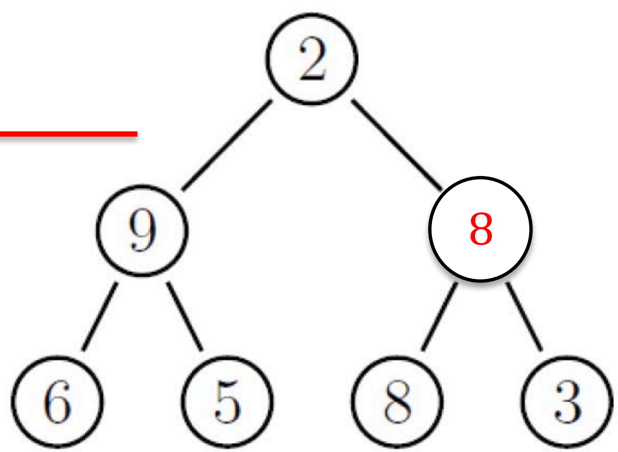


$H[1, \cdots, 7]$	[2 9 7 6 5 8 3]
n	7
i	3
k	3
v	7
j	6
$H[k]$	7
$H[j]$	8
$H[j+1]$	3
not $HEAP$	$TRUE$
$2 \times k \leq n$	$TRUE$
$j < n$	$TRUE$
$H[j] < H[j+1]$	$FALSE$
$v \geq H[j]$	$FALSE$

Algorithm to Turn $H[1,\cdots,n]$ into a Heap, Bottom-Up

```
for i ← ⌊n/2⌋ downto 1 do
    k ← i
    v ← H[k]
    heap ← False
    while not heap and 2 × k ≤ n do
        j ← 2 × k
        if j < n then
            if H[j] < H[j + 1] then
                j ← j + 1
        if v ≥ H[j] then
            heap ← True
        else
            H[k] ← H[j]
            k ← j
    H[k] ← v
```

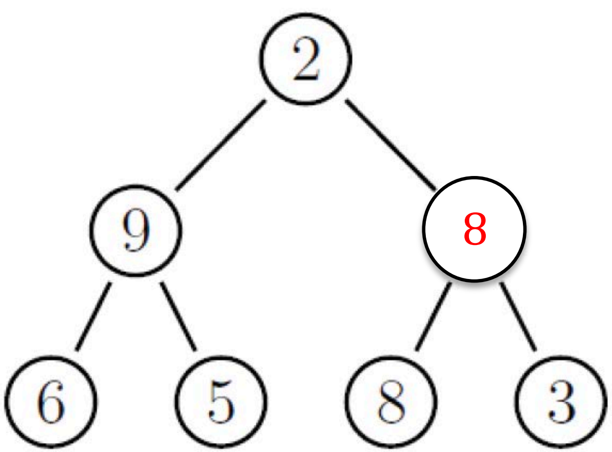
$H[1,\cdots,7]$	[2 9 8 6 5 8 3]
n	7
i	3
k	6
v	7
j	6
$H[k]$	8
$H[j]$	8
$H[j+1]$	3
not $HEAP$	$TRUE$
$2\times k \le n$	$TRUE$
$j < n$	$TRUE$
$H[j] < H[j+1]$	$FALSE$
$v \ge H[j]$	$FALSE$



Algorithm to Turn $H[1,\cdots,n]$ into a Heap, Bottom-Up

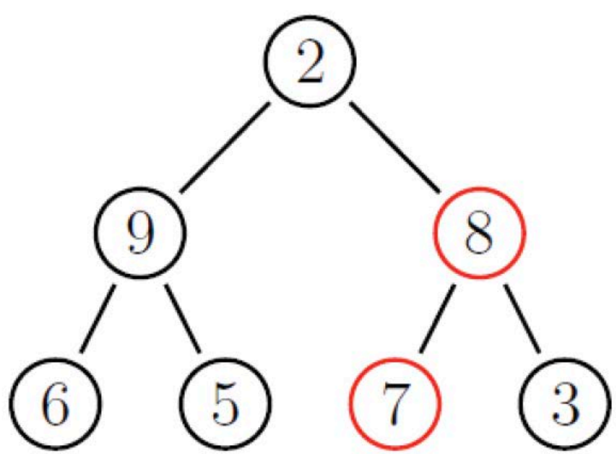
```
for i ← ⌊n/2⌋ downto 1 do
    k ← i
    v ← H[k]
    heap ← False
    while not heap and 2 × k ≤ n do
        j ← 2 × k
        if j < n then
            if H[j] < H[j + 1] then
                j ← j + 1
        if v ≥ H[j] then
            heap ← True
        else
            H[k] ← H[j]
            k ← j
    H[k] ← v
```

$H[1,\cdots,7]$	[2 9 8 6 5 8 3]
n	7
i	3
k	6
v	7
j	6
$H[k]$	8
$H[j]$	8
$H[j+1]$	3
not $HEAP$	$TRUE$
$2\times k \leq n$	$FALSE$
$j < n$	$TRUE$
$H[j] < H[j+1]$	$FALSE$
$v \geq H[j]$	$FALSE$



Algorithm to Turn $H[1, \cdots, n]$ into a Heap, Bottom-Up

```
for i ← ⌊n/2⌋ downto 1 do
    k ← i
    v ← H[k]
    heap ← False
    while not heap and 2 × k ≤ n do
        j ← 2 × k
        if j < n then
            if H[j] < H[j + 1] then
                j ← j + 1
        if v ≥ H[j] then
            heap ← True
        else
            H[k] ← H[j]
            k ← j
    H[k] ← v
```



$H[1, \cdots, 7]$	[2 9 8 6 5 8 3]
n	7
i	3
k	6
v	7
j	6
$H[k]$	7
$H[j]$	8
$H[j+1]$	3
not $HEAP$	$TRUE$
$2 \times k \leq n$	$FALSE$
$j < n$	$TRUE$
$H[j] < H[j+1]$	$FALSE$
$v \geq H[j]$	$FALSE$

Analysis of Bottom-Up Heap Creation

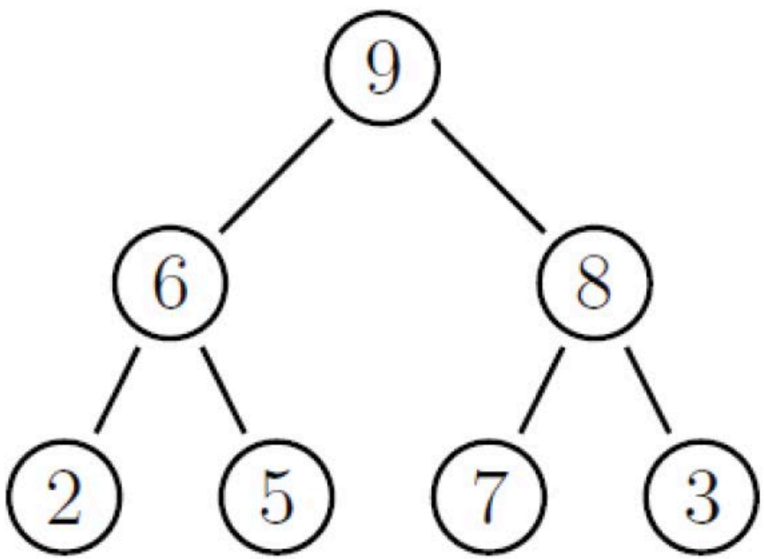
- For simplicity, assume the heap is a full binary tree: $n = 2^{h+1} - 1$.
- Here is an upper bound on the number of "down-sifts" needed (consider the root to be at level h, so leaves are at level 0):

$$\sum_{i=1}^{h} \sum_{nodes\ at\ level\ i} i = \sum_{i=1}^{h} i \cdot 2^{h-i} = 2^{h+1} - h - 2$$

- The last equation is easily proved by mathematical induction (or see Levitin Appendix A).
- Note that $2^{h+1} - h - 2 < $ n, so we perform at most a linear number of down-sift operations. Each down-sift is preceded by two key comparison, so the number of comparison is also linear.
- Hence we have a linear-time algorithm for heap creation.

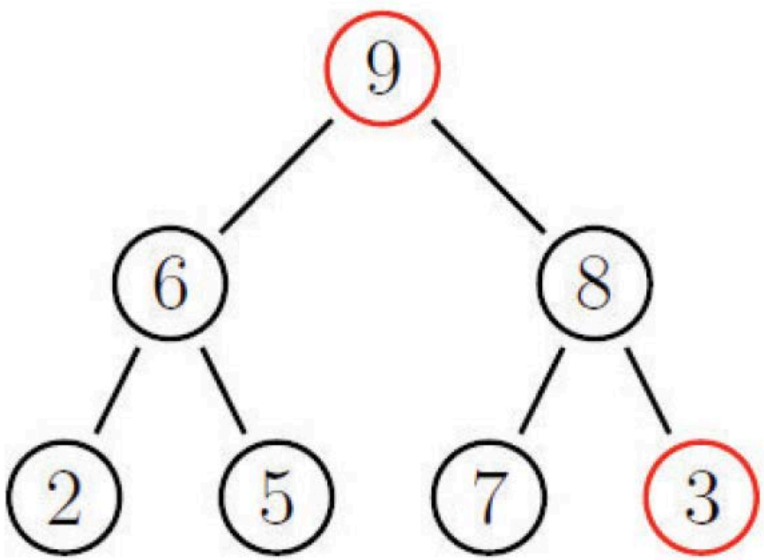
Ejecting a Maximal Element from a Heap

- Here the idea is to swap the root with the last item z in the heap, and then let the z "sift-down" to its proper place.



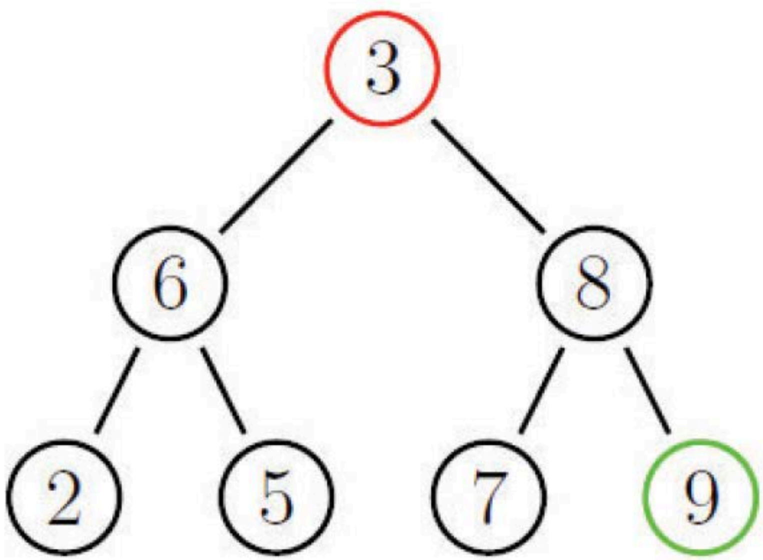
Ejecting a Maximal Element from a Heap

- Here the idea is to swap the root with the last item z in the heap, and then let the z "sift-down" to its proper place.



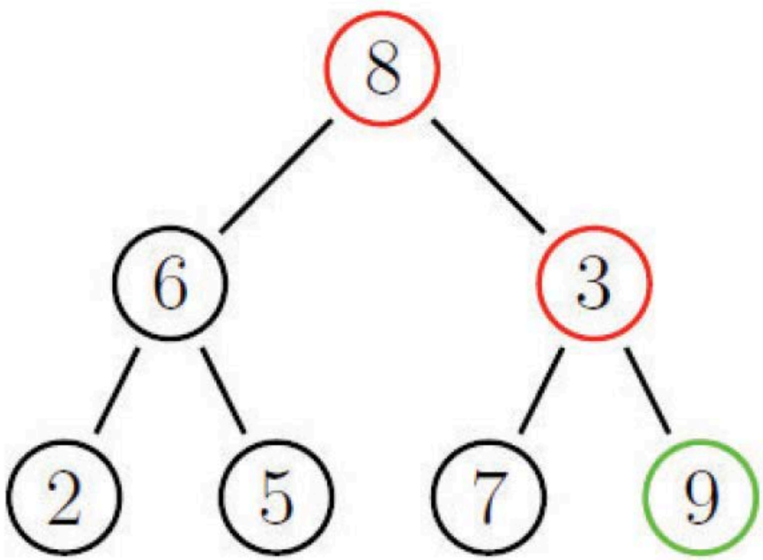
Ejecting a Maximal Element from a Heap

- Here the idea is to swap the root with the last item z in the heap, and then let the z "sift-down" to its proper place.
- After this, the last element (shown here in green) is no longer considered part of the heap, that is, n is decremented.
- Clearly ejection is $O(\log n)$.



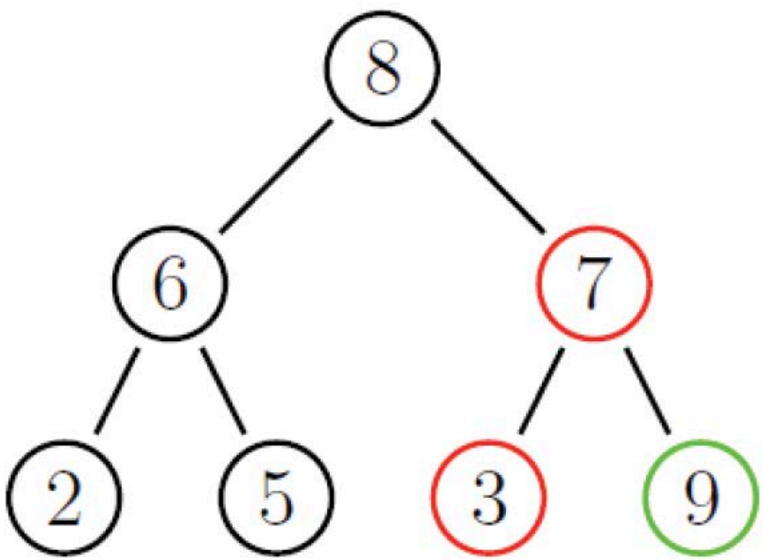
Ejecting a Maximal Element from a Heap

- Here the idea is to swap the root with the last item z in the heap, and then let the z "sift-down" to its proper place.
- After this, the last element (shown here in green) is no longer considered part of the heap, that is, n is decremented.
- Clearly ejection is $O(\log n)$.



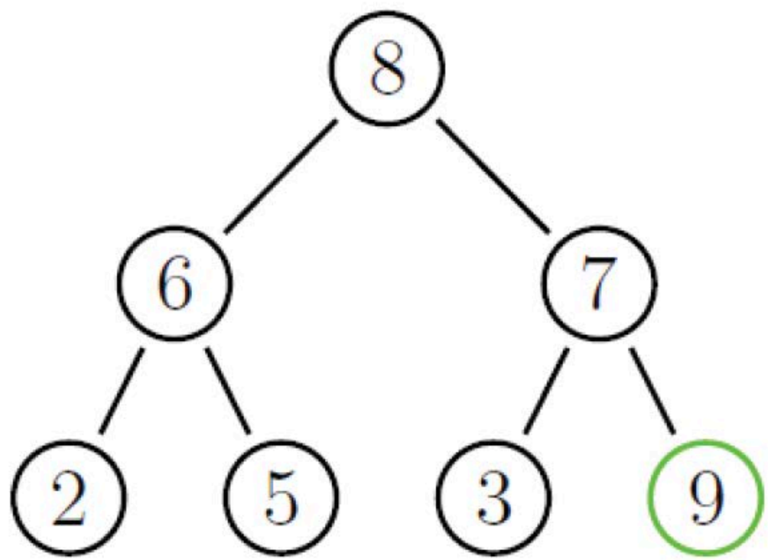
Ejecting a Maximal Element from a Heap

- Here the idea is to swap the root with the last item z in the heap, and then let the z "sift-down" to its proper place.
- After this, the last element (shown here in green) is no longer considered part of the heap, that is, n is decremented.
- Clearly ejection is $O(\log n)$.



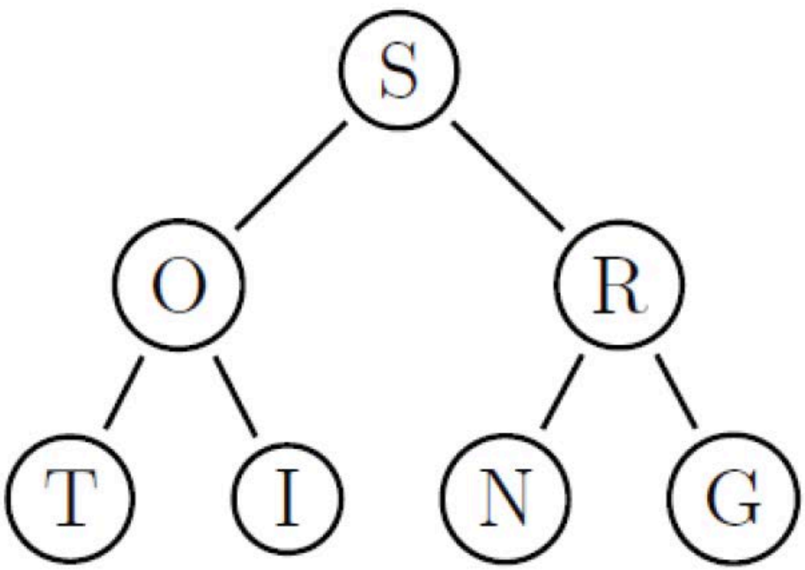
Ejecting a Maximal Element from a Heap

- Here the idea is to swap the root with the last item z in the heap, and then let the z "sift-down" to its proper place.
- After this, the last element (shown here in green) is no longer considered part of the heap, that is, n is decremented.
- Clearly ejection is $O(\log n)$.



Build and Then Deplete a Heap

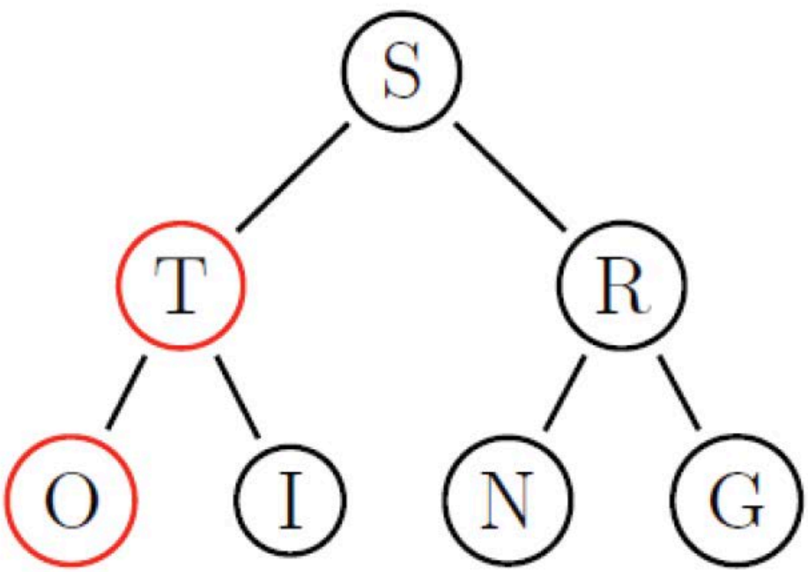
- First build a heap from the items: S, O, R, T, I, N, G
- Then repeatedly eject the largest, placing it at the end of the heap.



Build and Then Deplete a Heap

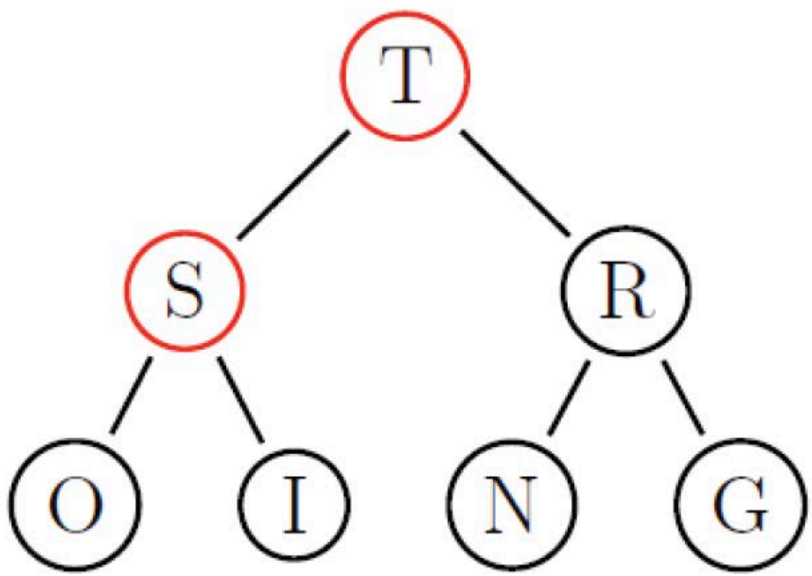
- First build a heap from the items: S, O, R, T, I, N, G

- Then repeatedly eject the largest, placing it at the end of the heap.



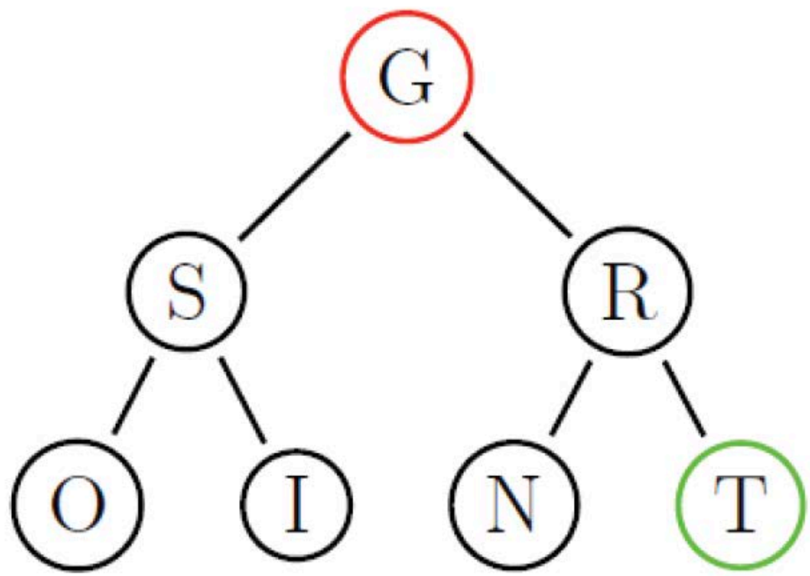
Build and Then Deplete a Heap

- First build a heap from the items: S, O, R, T, I, N, G
- Then repeatedly eject the largest, placing it at the end of the heap.



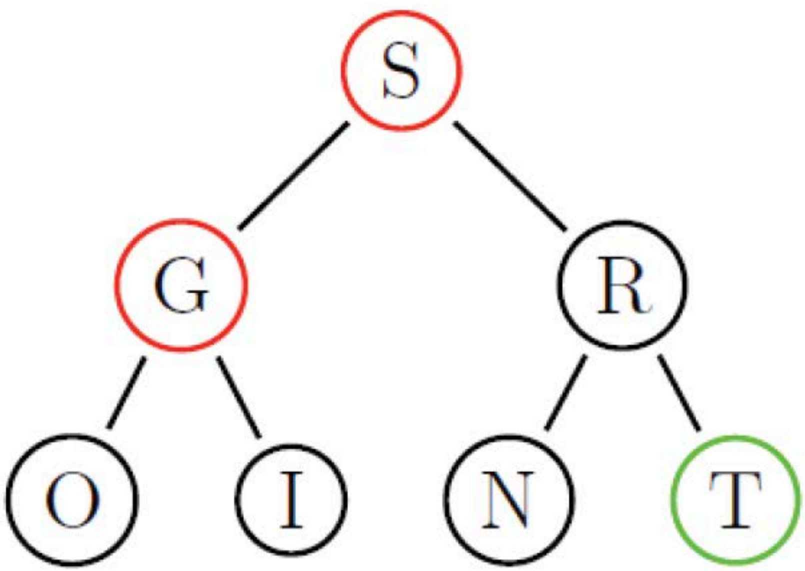
Build and Then Deplete a Heap

- First build a heap from the items: S, O, R, T, I, N, G
- Then repeatedly eject the largest, placing it at the end of the heap.
- Ejected: T



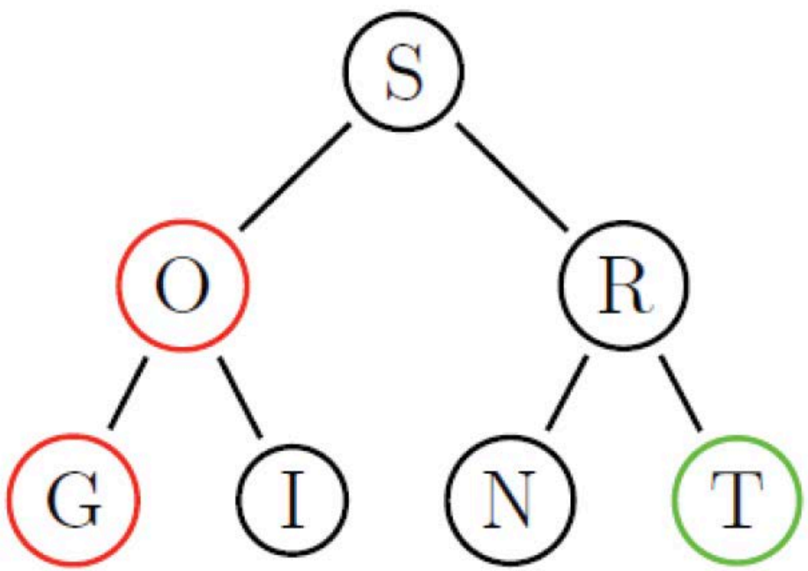
Build and Then Deplete a Heap

- First build a heap from the items: S, O, R, T, I, N, G
- Then repeatedly eject the largest, placing it at the end of the heap.
- Ejected: T



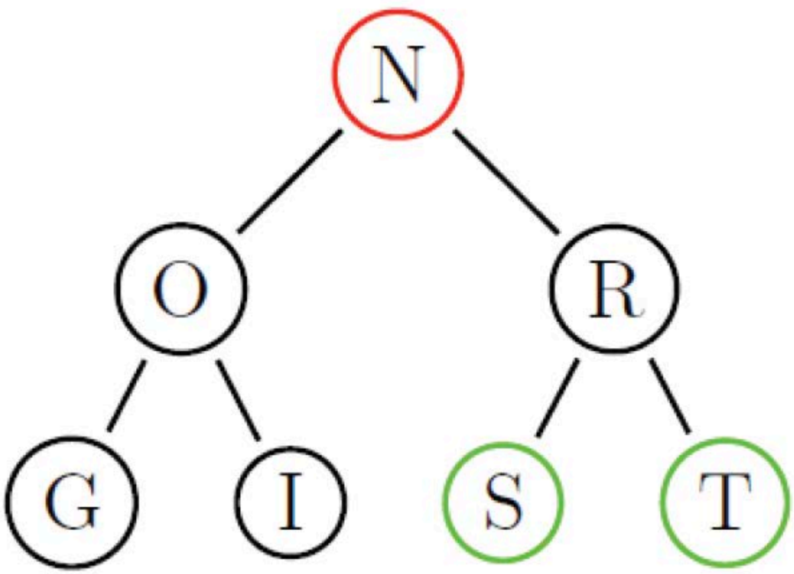
Build and Then Deplete a Heap

- First build a heap from the items: S, O, R, T, I, N, G
- Then repeatedly eject the largest, placing it at the end of the heap.
- Ejected: T



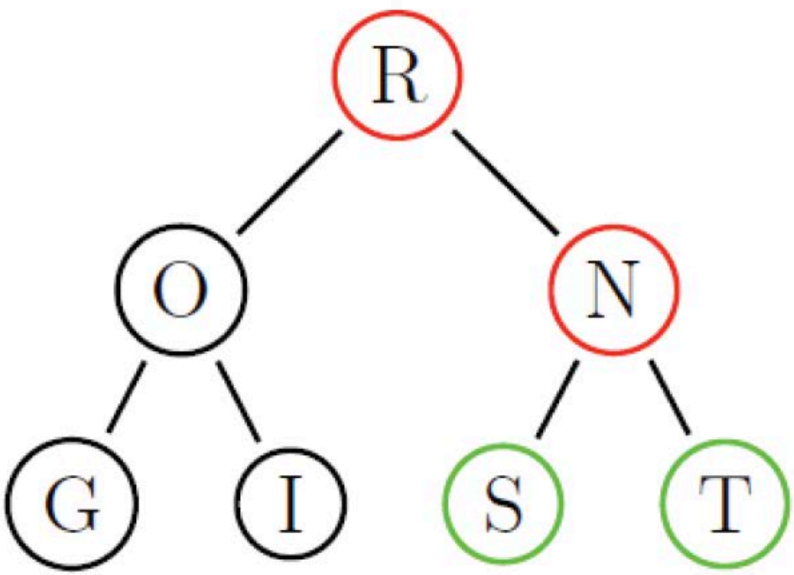
Build and Then Deplete a Heap

- First build a heap from the items: S, O, R, T, I, N, G
- Then repeatedly eject the largest, placing it at the end of the heap.
- Ejected: T, S



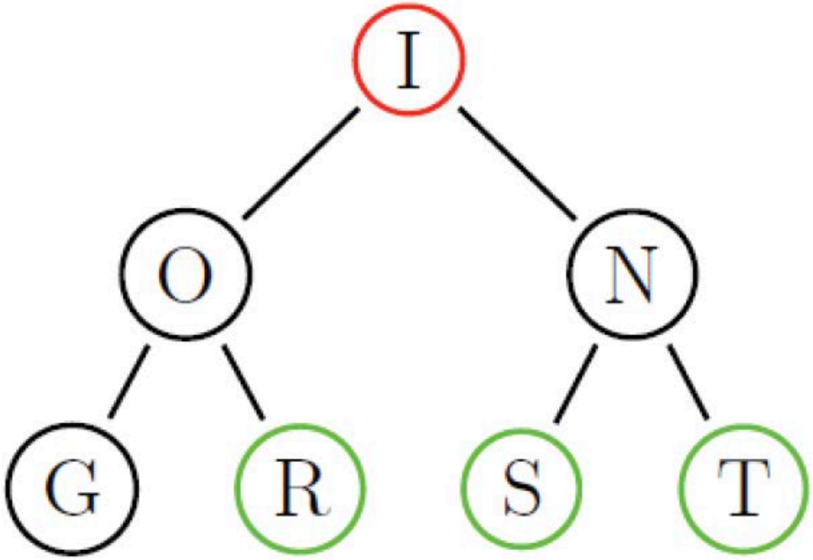
Build and Then Deplete a Heap

- First build a heap from the items: S, O, R, T, I, N, G
- Then repeatedly eject the largest, placing it at the end of the heap.
- Ejected: T, S
-



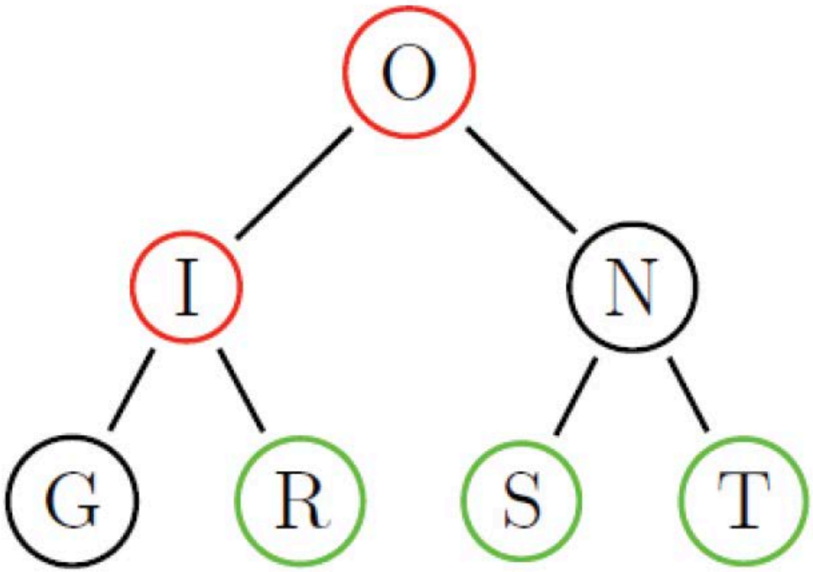
Build and Then Deplete a Heap

- First build a heap from the items: S, O, R, T, I, N, G
- Then repeatedly eject the largest, placing it at the end of the heap.
- Ejected: T, S, R



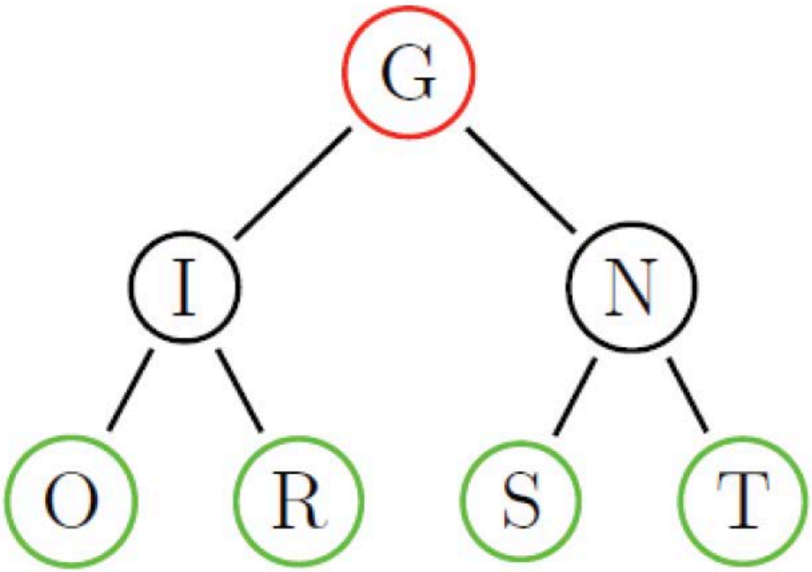
Build and Then Deplete a Heap

- First build a heap from the items: S, O, R, T, I, N, G
- Then repeatedly eject the largest, placing it at the end of the heap.
- Ejected: T, S, R



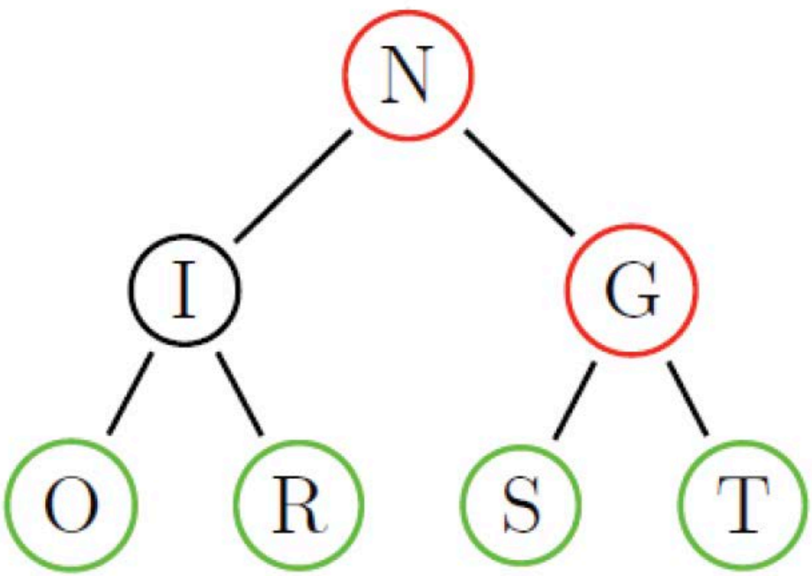
Build and Then Deplete a Heap

- First build a heap from the items: S, O, R, T, I, N, G
- Then repeatedly eject the largest, placing it at the end of the heap.
- Ejected: T, S, R, O



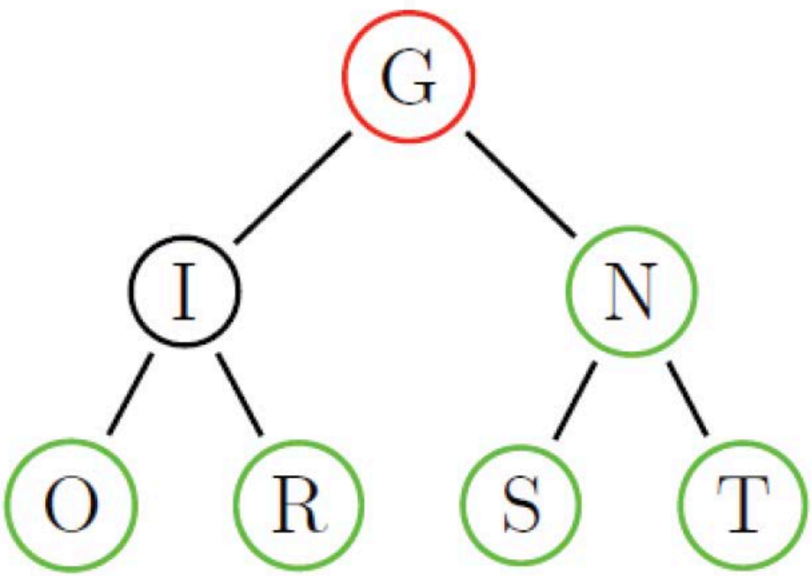
Build and Then Deplete a Heap

- First build a heap from the items: S, O, R, T, I, N, G
- Then repeatedly eject the largest, placing it at the end of the heap.
- Ejected: T, S, R, O



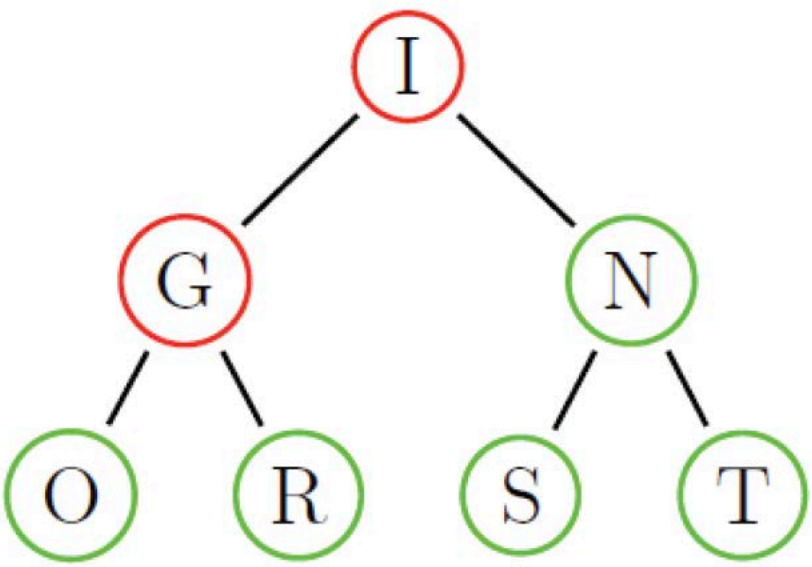
Build and Then Deplete a Heap

- First build a heap from the items: S, O, R, T, I, N, G
- Then repeatedly eject the largest, placing it at the end of the heap.
- Ejected: T, S, R, O, N



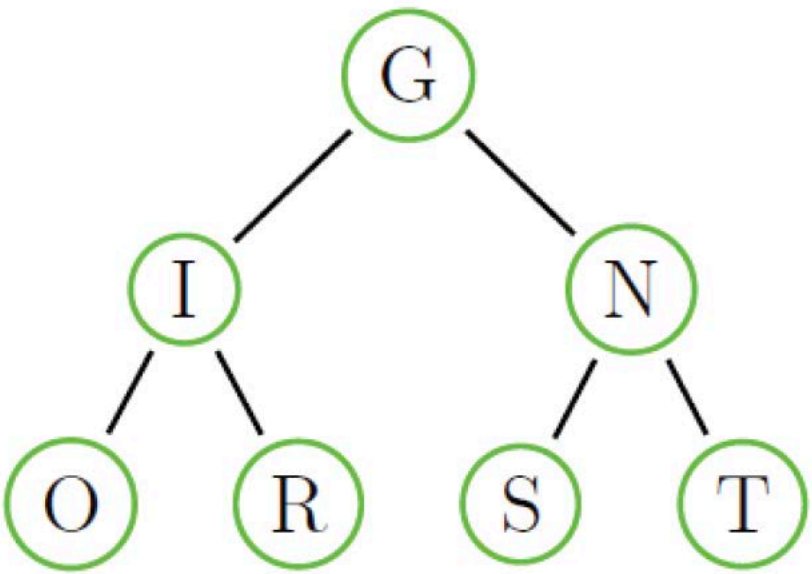
Build and Then Deplete a Heap

- First build a heap from the items: S, O, R, T, I, N, G
- Then repeatedly eject the largest, placing it at the end of the heap.
- Ejected: T, S, R, O, N



Build and Then Deplete a Heap

- First build a heap from the items: S, O, R, T, I, N, G
- Then repeatedly eject the largest, placing it at the end of the heap.
- Ejected: T, S, R, O, N, I, G



Heapsort

- Heapsort is a $\Theta(n \log n)$ sorting algorithm, based on the idea from this exercise.
- Given unsorted array $H[1, \cdots, n]$:
  - Step 1 Turn H into a heap.
  - Step 2 Apply the eject operation $n - 1$ times.

Properties of Heapsort

- On average slower that quicksort, but stronger performance guarantee.
- Truly in place.
- Not stable.

Coming Up Next

- We will look at the “Transform and Conquer” paradigm (Levitin Section 6.1).