# COMP 251

## Algorithms & Data Structures (Winter 2021)

## Heaps

School of Computer Science
McGill University

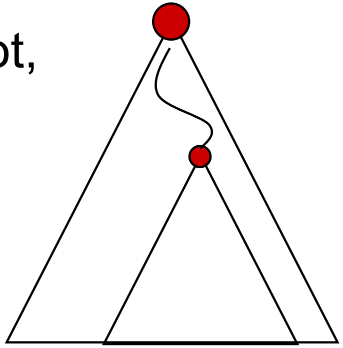
Based on (Cormen *et al.*, 2002) & slides of (Waldispuhl,2020), (Langer, 2004) and (D. Plaisted).

# Outline

- Introduction.
- Operations.
- Application.

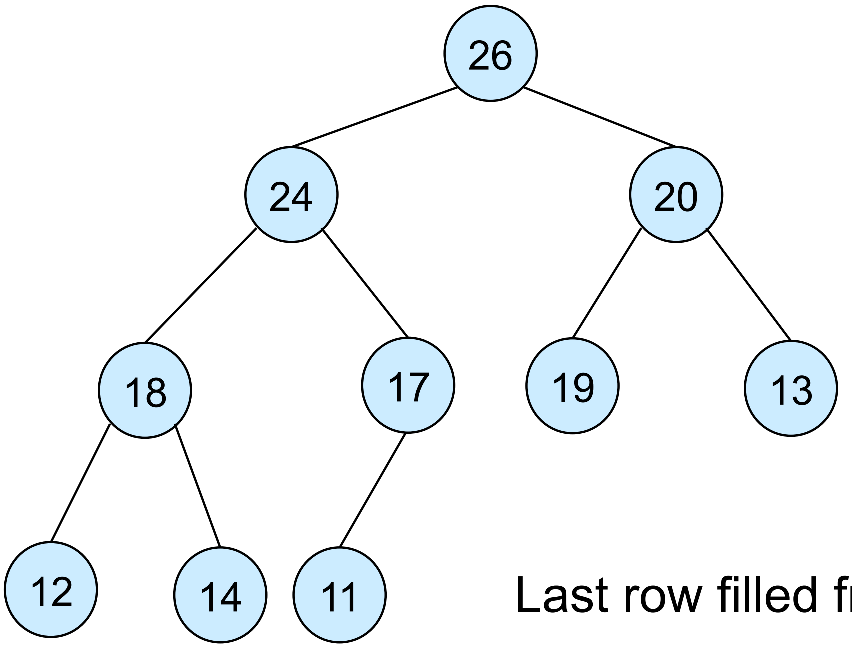
# Introduction – Heap data structure

- Tree-based data structure (here, binary tree, but we can also use k-ary trees)
- Max-Heap
  - Largest element is stored at the root.
  - for all nodes *i*, excluding the root, $A[\mathrm{PARENT}(i)] \geq A[i]$.
- Min-Heap
  - Smallest element is stored at the root.
  - for all nodes *i*, excluding the root, excluding the root, $A[\mathrm{PARENT}(i)] \leq A[i]$.
- Tree is filled top-down from left to right.



# Introduction – Heap - example

Max-heap as a binary tree.

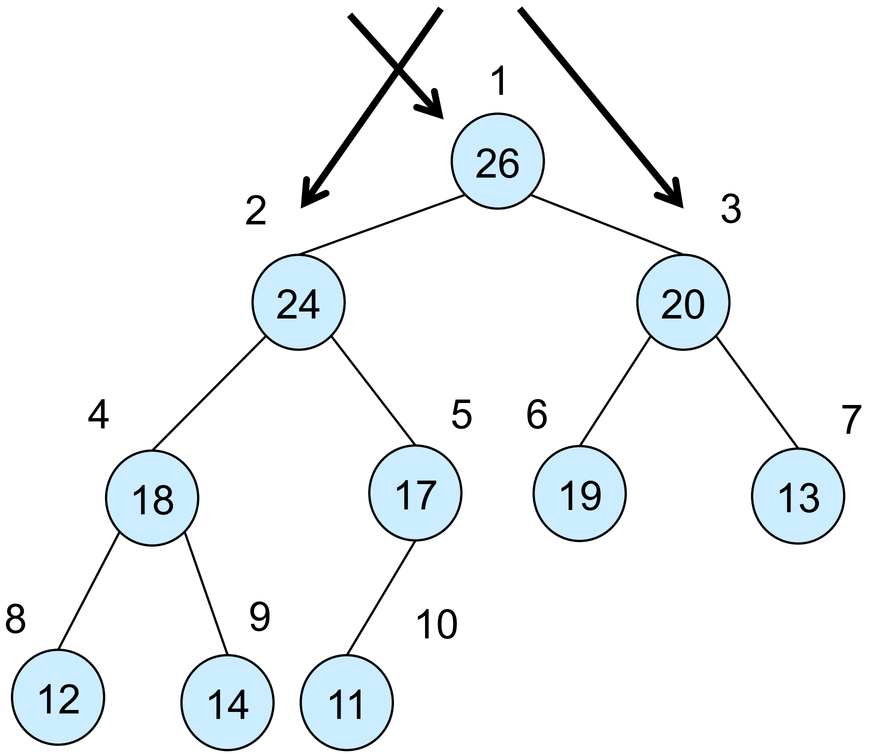



Last row filled from left to right.

# Introduction – Heap as arrays - example

| 26 | 24 | 20 | 18 | 17 | 19 | 13 | 12 | 14 | 11 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Max-heap as an array.




Map from array elements to tree nodes:

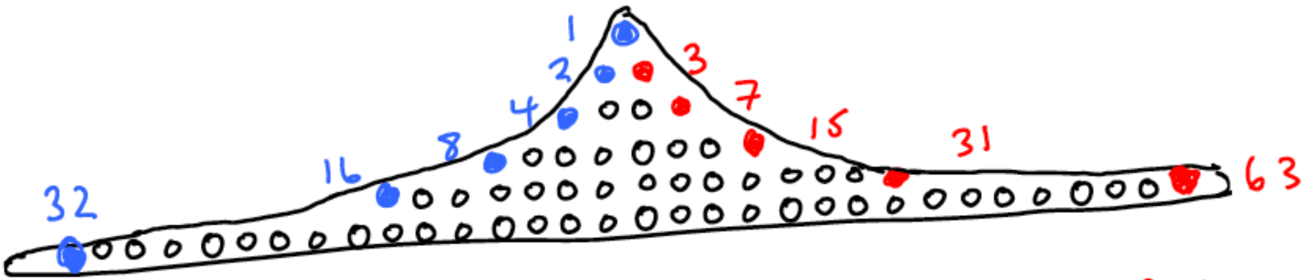
- Root : $A[1]$
- Left[$i$] : $A[2i]$
- Right[$i$] : $A[2i+1]$
- Parent[$i$] : $A[\lfloor i/2 \rfloor]$

# Introduction – Heap - Height

- *Height of a node in a tree*: the number of edges on the longest simple path down from the node to a leaf.
- *Height of a heap* = *height of the root* = O(lg n).
- Most Basic operations on a heap run in $O(\lg n)$ time
- Shape of a heap

# Introduction – Heap - Height

Consider a complete binary tree of height $h$, with all levels full.




level $\ell$ has $2^\ell$ nodes : $2^\ell, \ldots, 2^{\ell+1}-1$

number of nodes : $n = 2^{h+1} - 1$

height of tree : $h = \log(n+1) - 1$

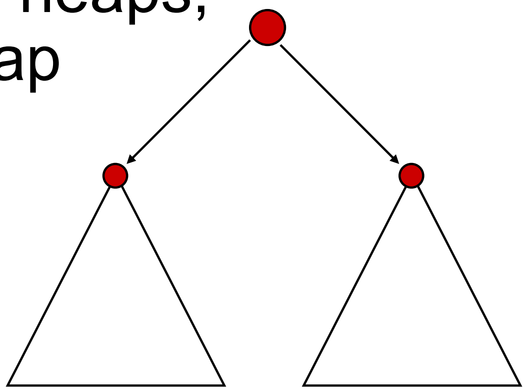
Taken from Langer2014

# Outline

- Introduction.
- Operations.
- Application.

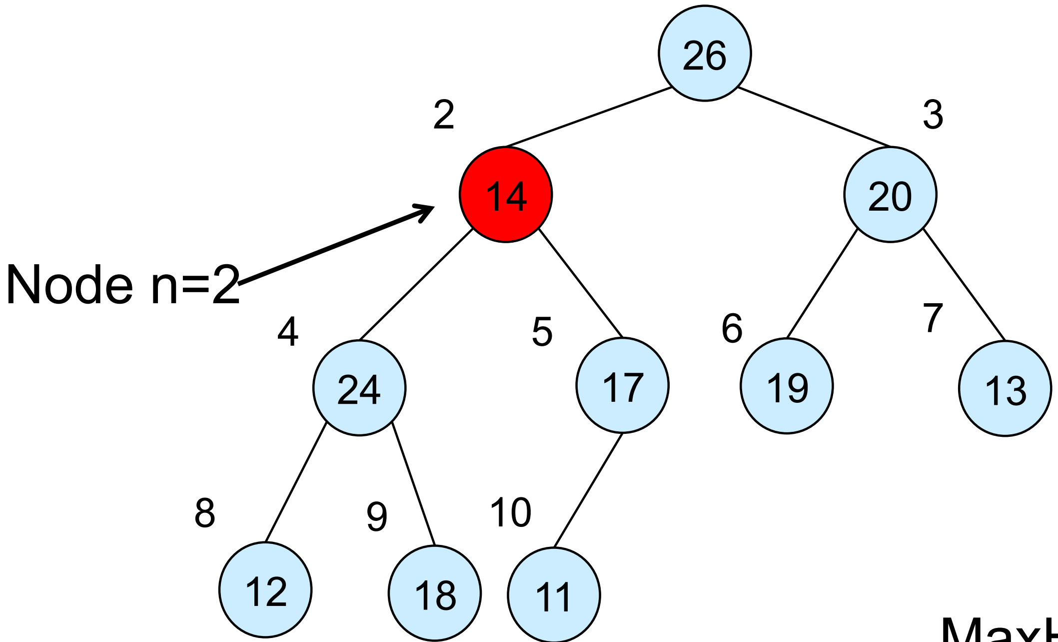
# Operations – Maintaining the heap property

- Suppose two sub-trees are max-heaps, but the root violates the max-heap property.



- Fix the offending node by exchanging the value at the node with the larger of the values at its children.
  - The resulting tree may have a sub-tree that is not a heap.
- Recursively fix the children until all of them satisfy the max-heap property.

# MaxHeapify – Example

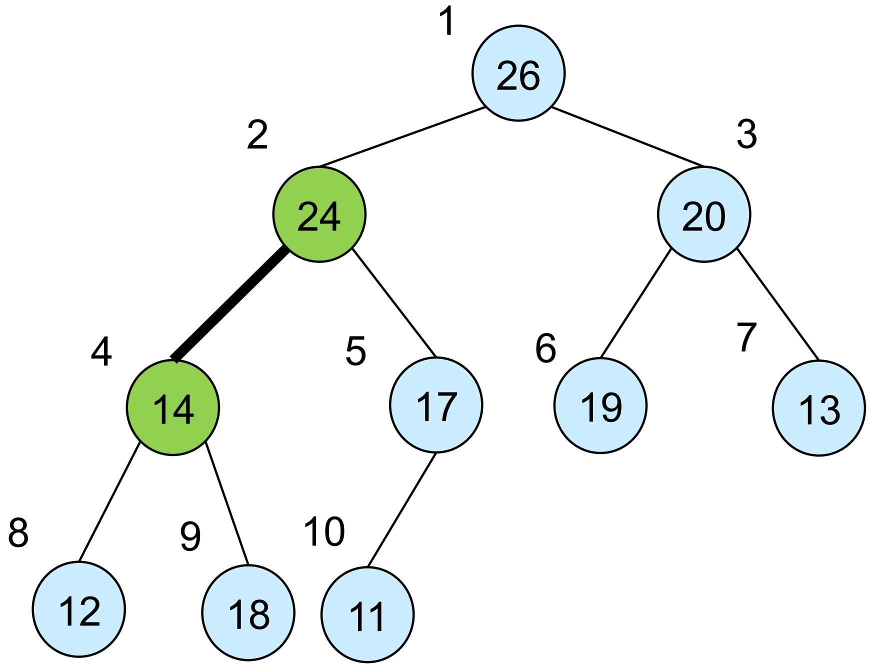



MaxHeapify($A$, 2)

array index

Note: The call assumes that the left and right sub-trees of the node i are max-heaps, but that the node i might be smaller than its children (violating the max-heap property)

# MaxHeapify – Example



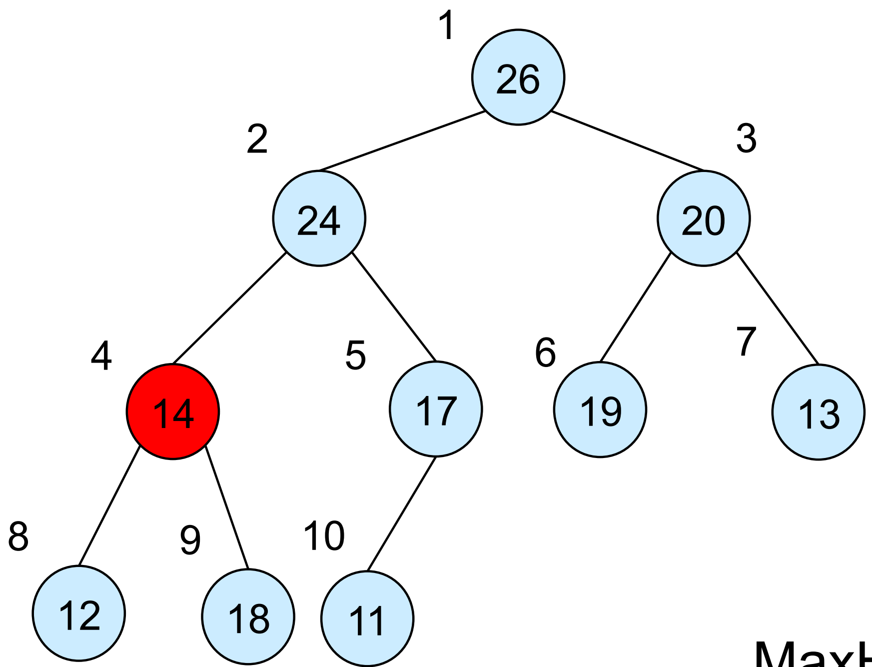

MaxHeapify($A$, 2)

array

index

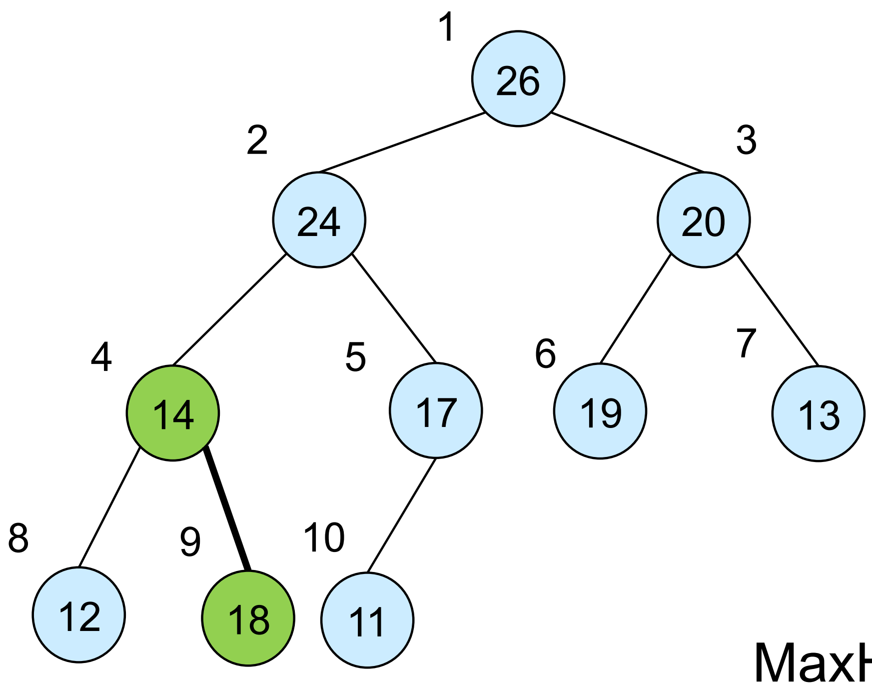
Note: The call assumes that the left and right sub-trees of the node i are max-heaps, but that the node i might be smaller than its children (violating the max-heap property)

# MaxHeapify – Example



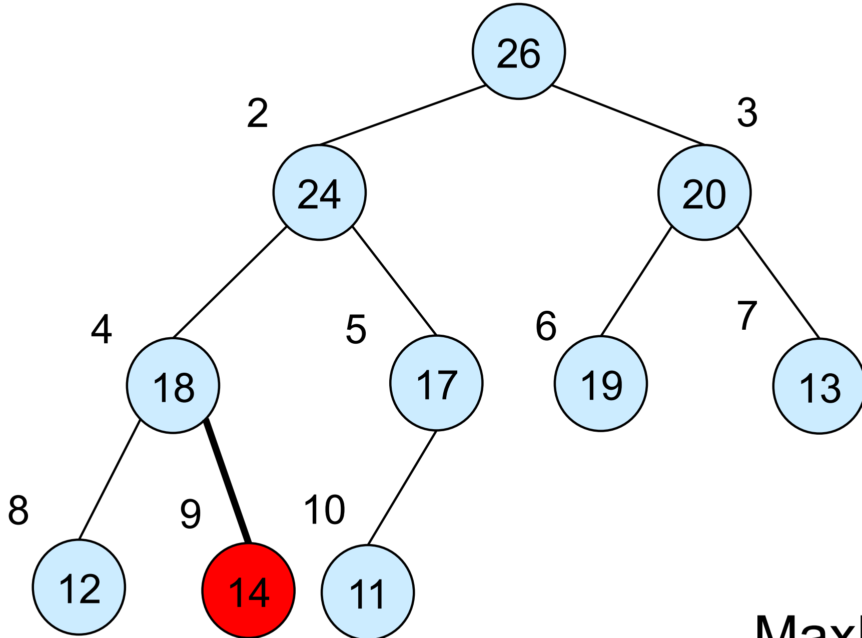

MaxHeapify($A$, 2)
MaxHeapify($A$, 4)

# MaxHeapify – Example



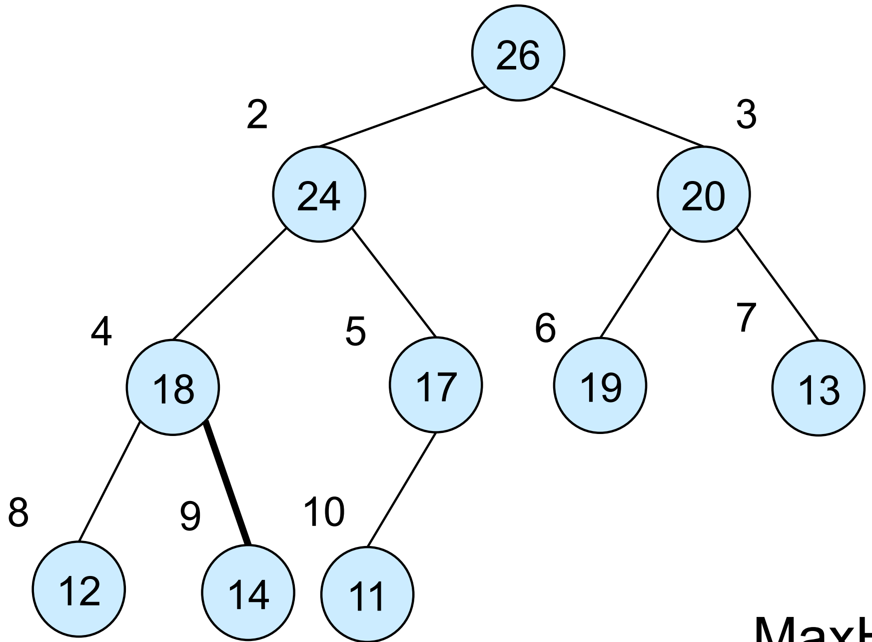

MaxHeapify($A$, 2)
MaxHeapify($A$, 4)

# MaxHeapify – Example




MaxHeapify($A$, 2)
MaxHeapify($A$, 4)
MaxHeapify($A$, 9)

# MaxHeapify – Example




MaxHeapify($A$, 2)
MaxHeapify($A$, 4)
MaxHeapify($A$, 9)

# MaxHeapify – Procedure

**Assumption:** Left(*i*) and Right(*i*) are max-heaps.
n is the size of the heap.

```
MaxHeapify(A, i)
  l ← leftNode(i)
  r ← rightNode(i)
  n ← HeapSize(i)
  if l ≤ n and A[l]>A[i]
      then largest ← l
      else largest ← i
  if r ≤ n and A[r]>A[largest]
      then largest ← r
  if largest ≠ i
      then exchange A[i] ↔ A[largest]
           MaxHeapify(A, largest)
```

largest $= \max\{l, r, i\}$

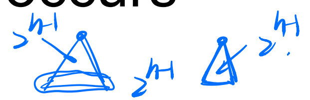
Time to determine if there is a conflict and find the largest children is O(1)

Time to fix the subtree rooted at one of *i*'s children is O(size of subtree)

# MaxHeapify – worst case

<u>`MaxHeapify(A, i)`</u>

- *Size of a tree = number of nodes in this tree*
- *T(n): time used for an input of size n*
- $T(n) = T(\textit{size of the largest subtree}) + O(1)$
- *Size of the largest subtree* $\leq 2n/3$ (worst case occurs when the last row of tree is exactly half full)

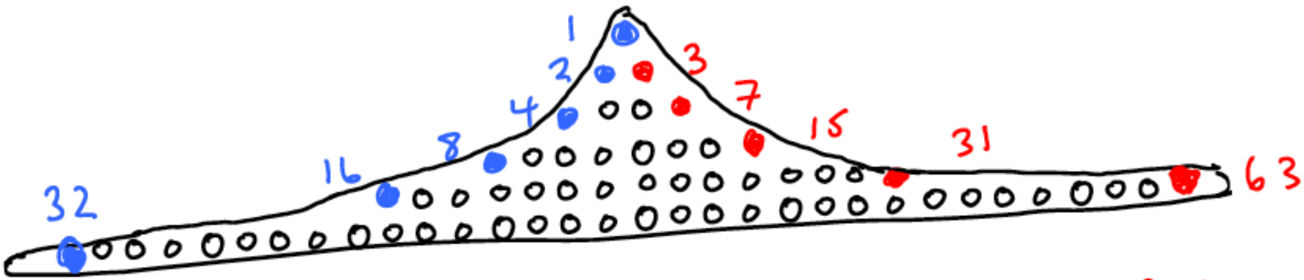


$\Rightarrow T(n) \leq T(2n/3) + \Theta(1) \Rightarrow$ $\boldsymbol{T(n) = O(\lg n)}$ Master's Theorem.

Alternately, MaxHeapify takes $O(h)$ where $h$ is the height of the node where MaxHeapify is applied

# Introduction – Heap - Height

Consider a complete binary tree of height $h$, with <u>all</u> levels full.




level $\ell$ has $2^\ell$ nodes : $2^\ell, \ldots, 2^{\ell+1}-1$

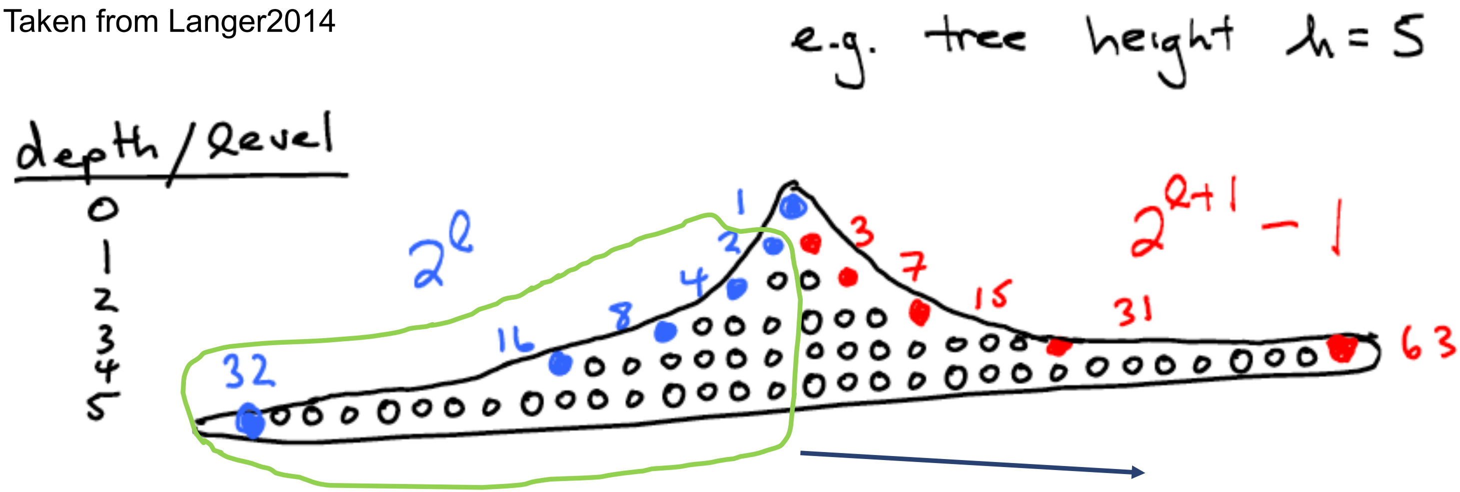
number of nodes : $n = 2^{h+1} - 1$

height of tree : $h = \log(n+1) - 1$

Taken from Langer2014

# MaxHeapify – supplemental material

Taken from Langer2014



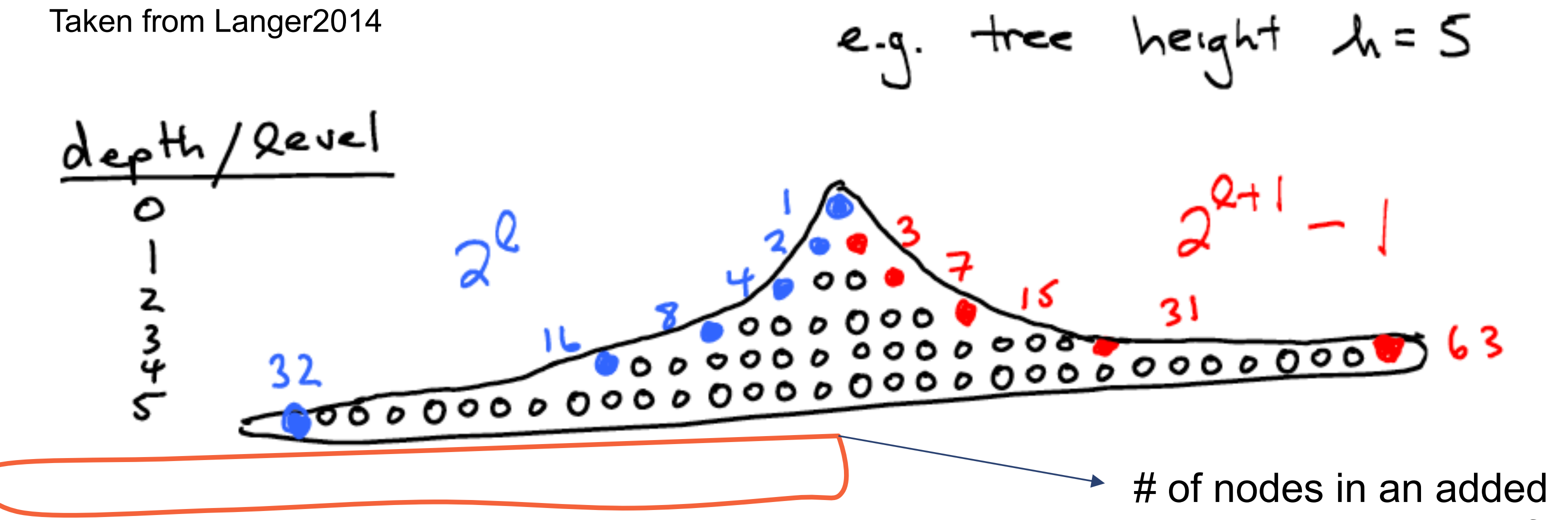

# of nodes in the left subtree

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

root

# MaxHeapify – supplemental material

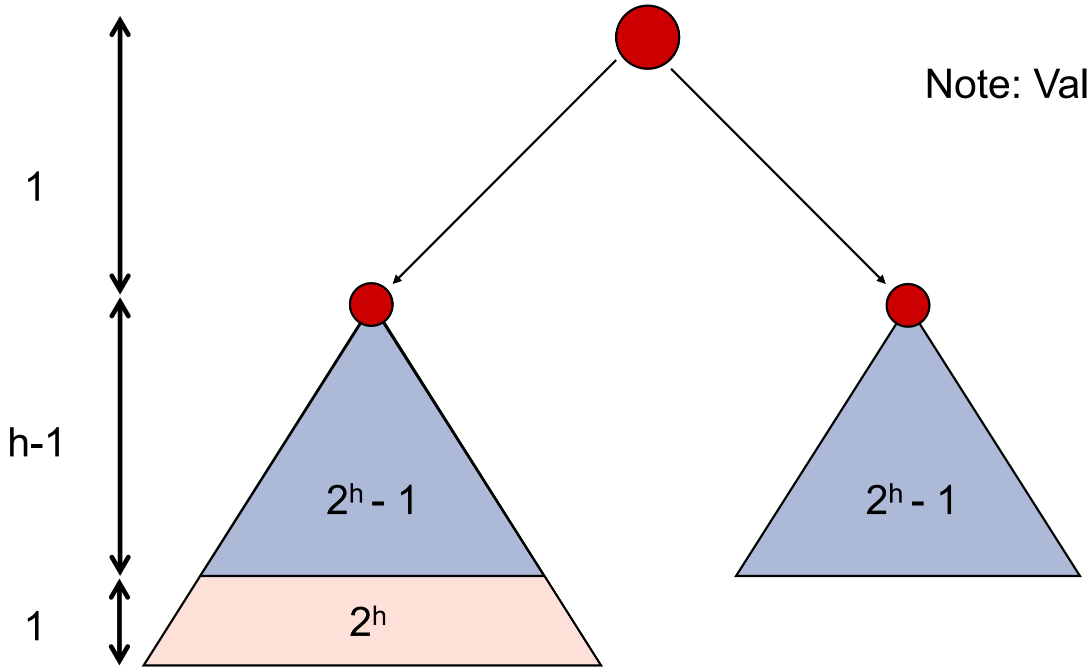
Taken from Langer2014




# of nodes in an added last row that is exactly half full.

$$\frac{2^{h+1}}{2} = 2^h$$

# MaxHeapify – worst case

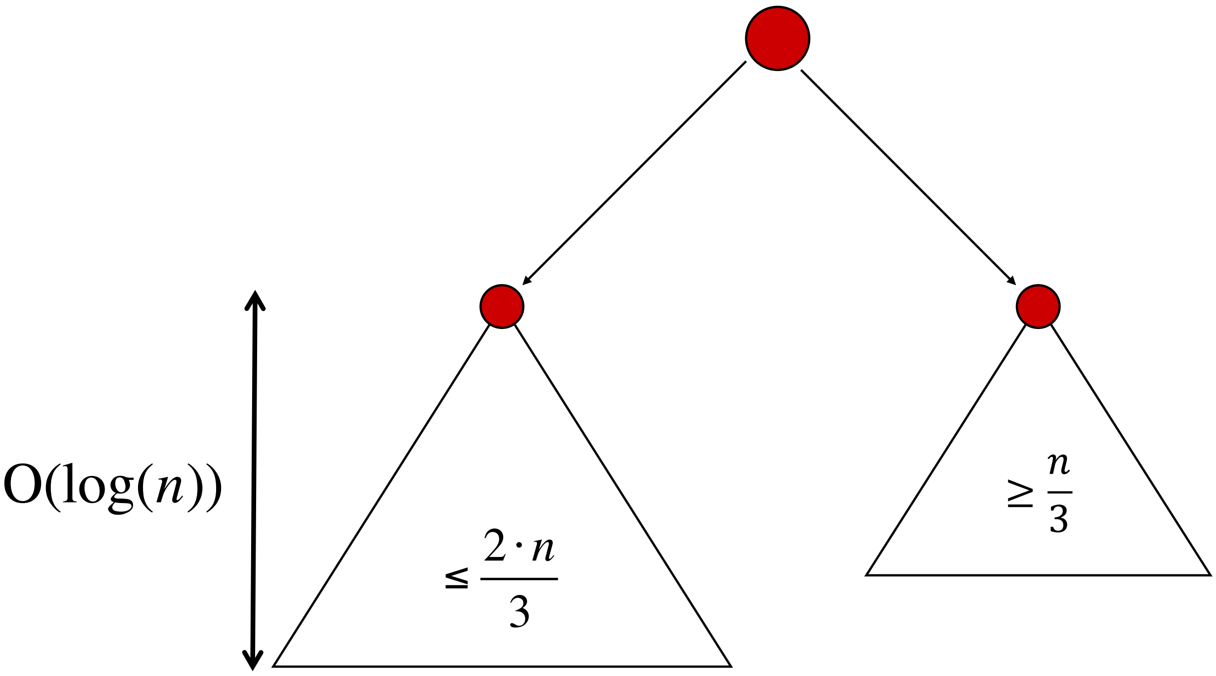



Note: Valid iff $h \geq 1$

Total in heap (n): $n = 2\left(2^h - 1\right) + \ 2^h + 1 = 3 \cdot 2^h - 1$

Total left subtree $n_{left} \leq 2^{h+1} - 1 = \frac{3}{3} \cdot 2 \cdot (2^h - \frac{1}{2}) = \frac{2}{3} \cdot (3 \cdot 2^h - \frac{3}{2}) \leq \frac{2}{3} \cdot n$

# MaxHeapify – worst case



# Operations – Building a heap

- Use *MaxHeapify* to convert an array *A* into a max-heap.
- Call MaxHeapify on each element in a bottom-up manner.

<u>*BuildMaxHeap(A)*</u>

$n \leftarrow length[A]$ leaf are max heapified

**for** $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1

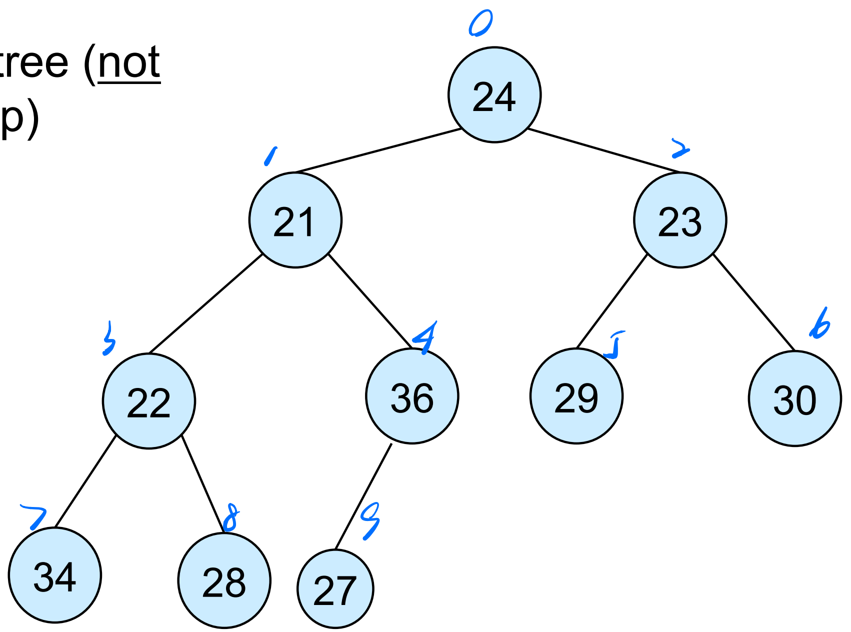
  **do** *MaxHeapify*(*A*, *i*)

# Building a heap - Example

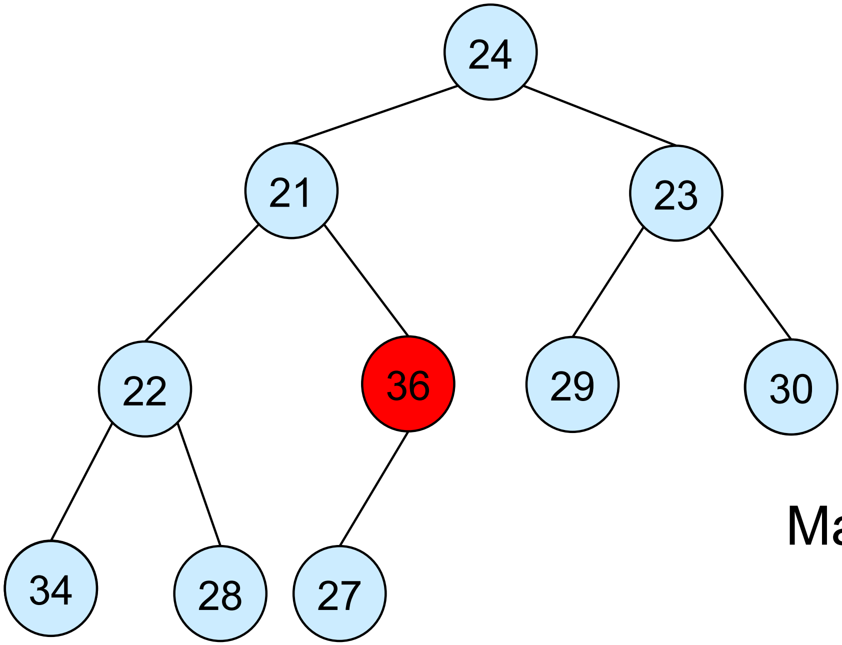
Input Array:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 24 | 21 | 23 | 22 | 36 | 29 | 30 | 34 | 28 | 27 |

Starting tree (<u>not</u> max-heap)

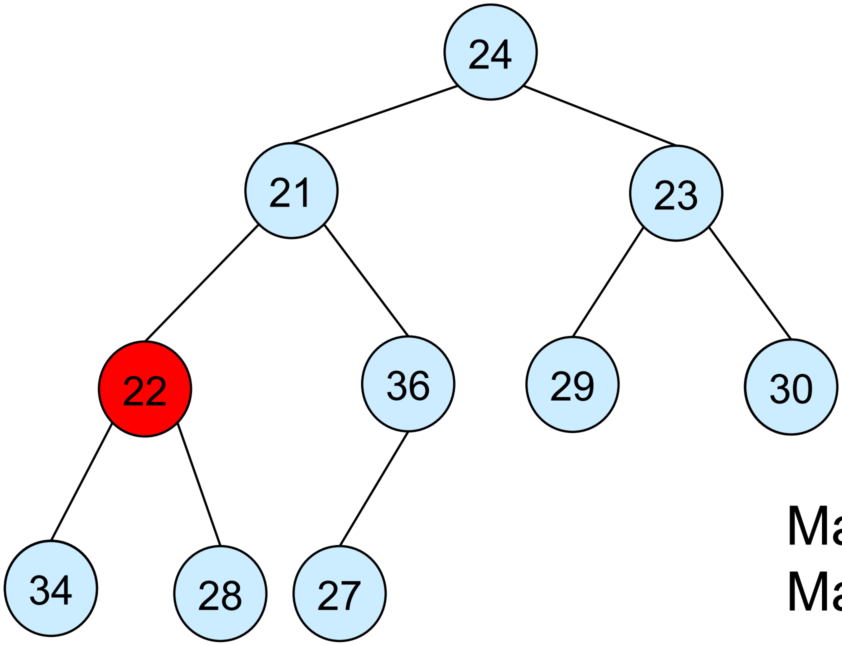


# Building a heap - Example



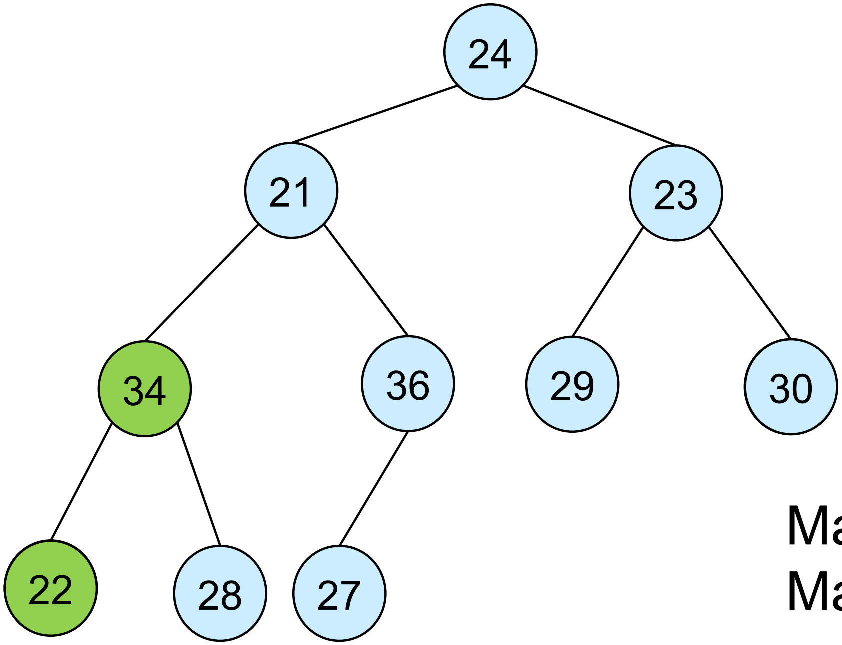

MaxHeapify($\lfloor 10/2 \rfloor = 5$)

# Building a heap - Example



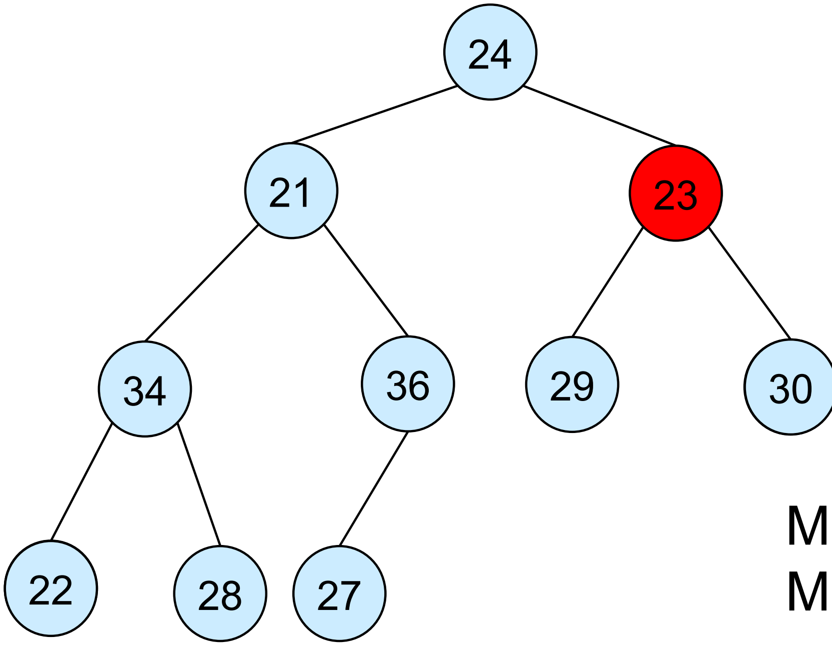

MaxHeapify($\lfloor 10/2 \rfloor = 5$)
MaxHeapify(4)

# Building a heap - Example



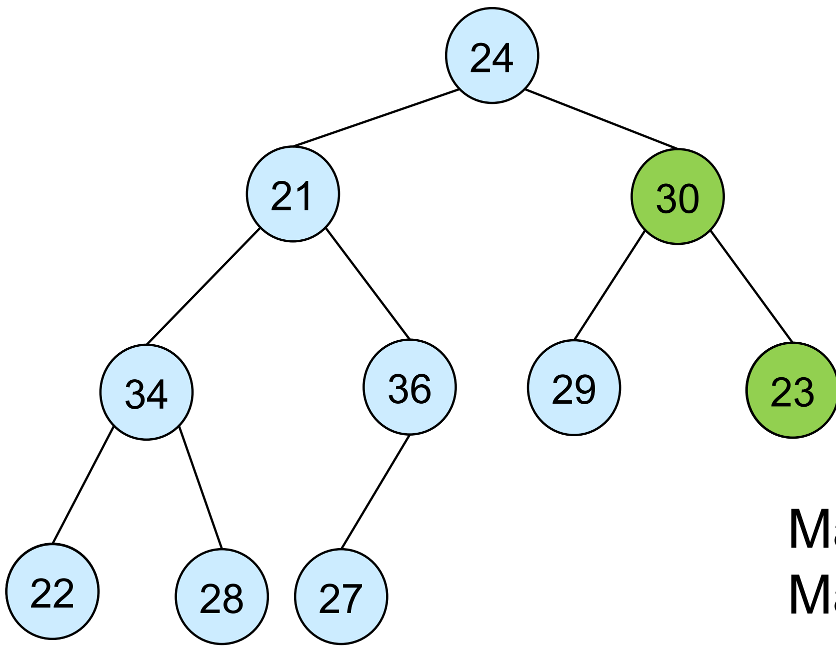

MaxHeapify($\lfloor 10/2 \rfloor = 5$)
MaxHeapify(4)

# Building a heap - Example



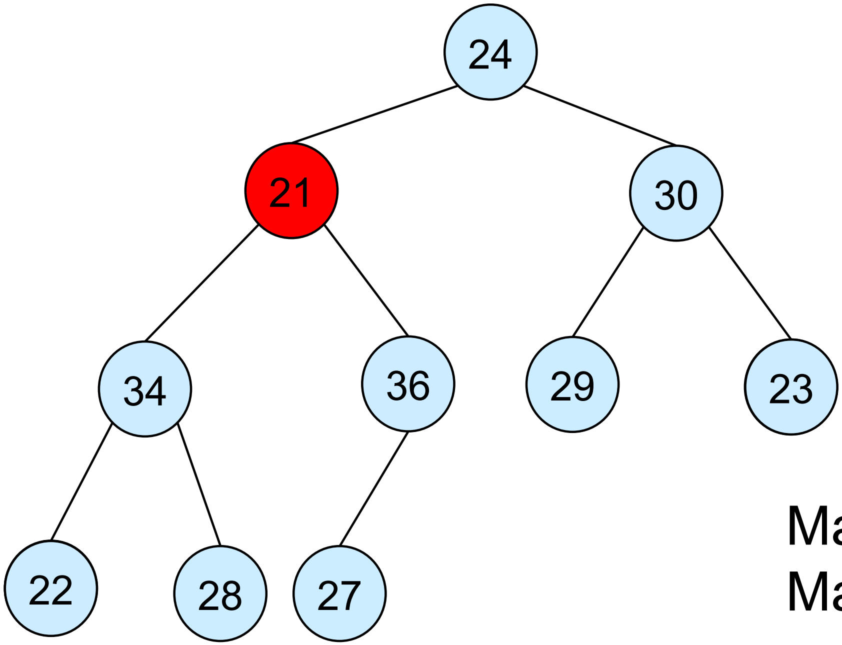

MaxHeapify($\lfloor 10/2 \rfloor = 5$)
MaxHeapify(4)
MaxHeapify(3)

# Building a heap - Example



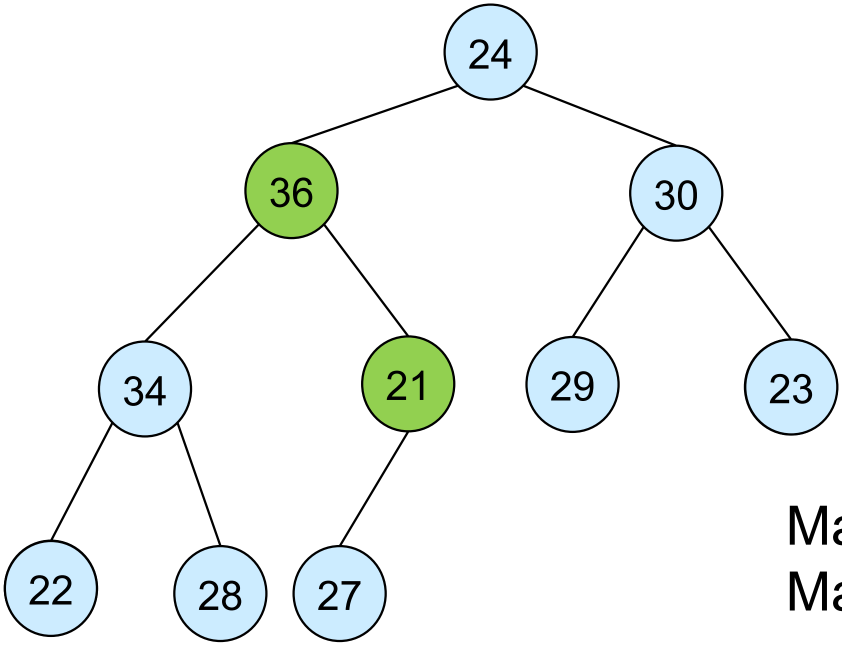

MaxHeapify($\lfloor 10/2 \rfloor = 5$)
MaxHeapify(4)
MaxHeapify(3)

# Building a heap - Example



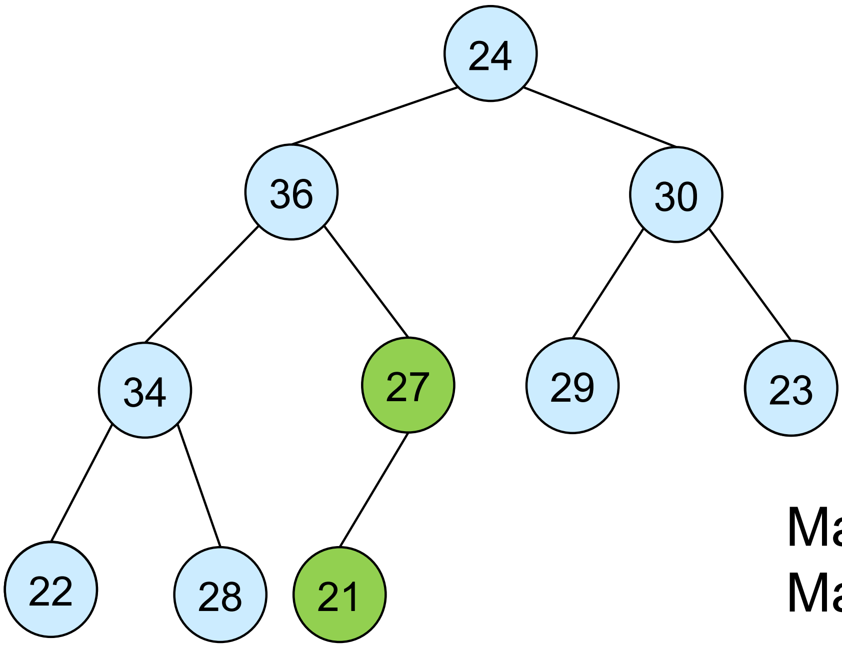

MaxHeapify($\lfloor 10/2 \rfloor = 5$)
MaxHeapify(4)
MaxHeapify(3)
MaxHeapify(2)

# Building a heap - Example



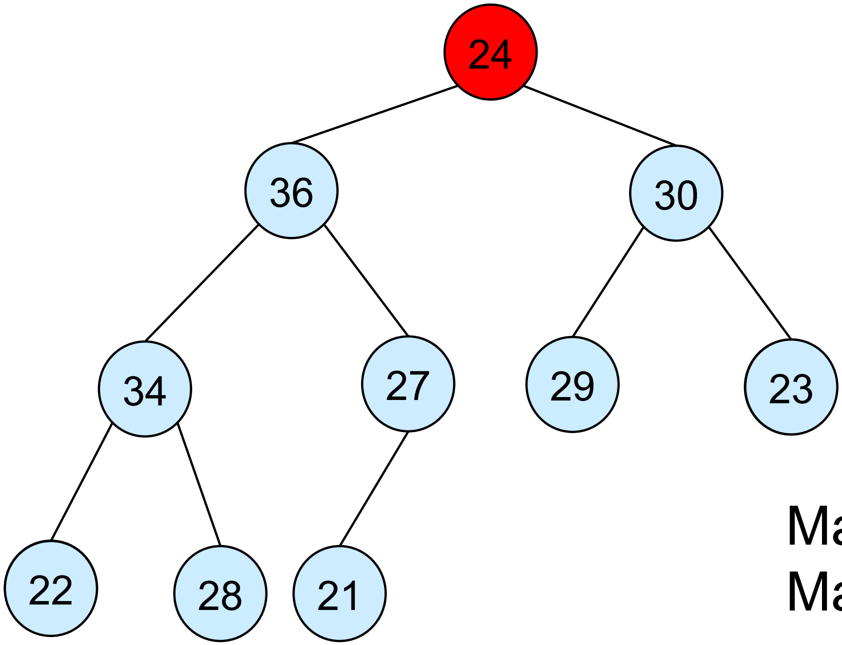

MaxHeapify($\lfloor 10/2 \rfloor = 5$)
MaxHeapify(4)
MaxHeapify(3)
MaxHeapify(2)

# Building a heap - Example



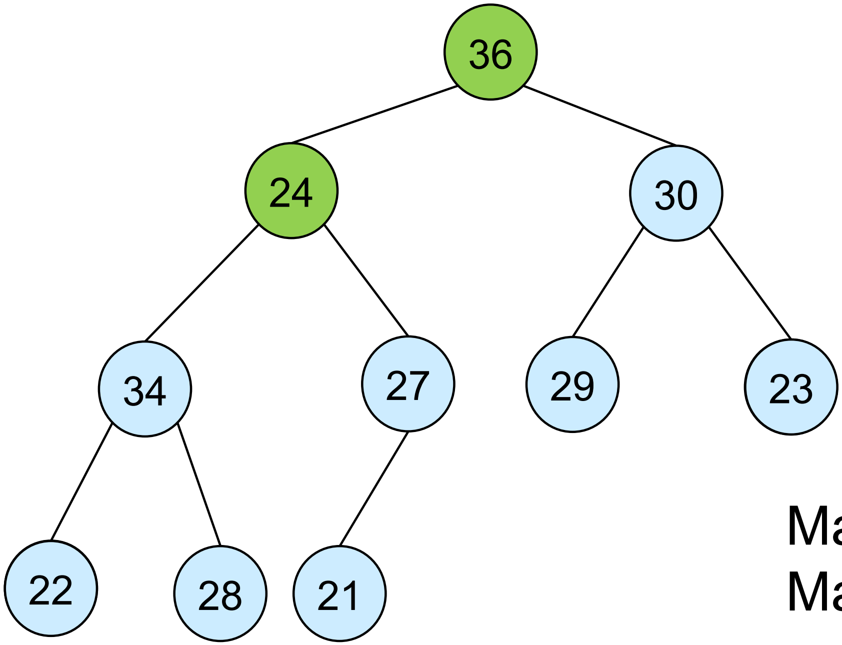

MaxHeapify($\lfloor 10/2 \rfloor = 5$)
MaxHeapify(4)
MaxHeapify(3)
MaxHeapify(2)

# Building a heap - Example



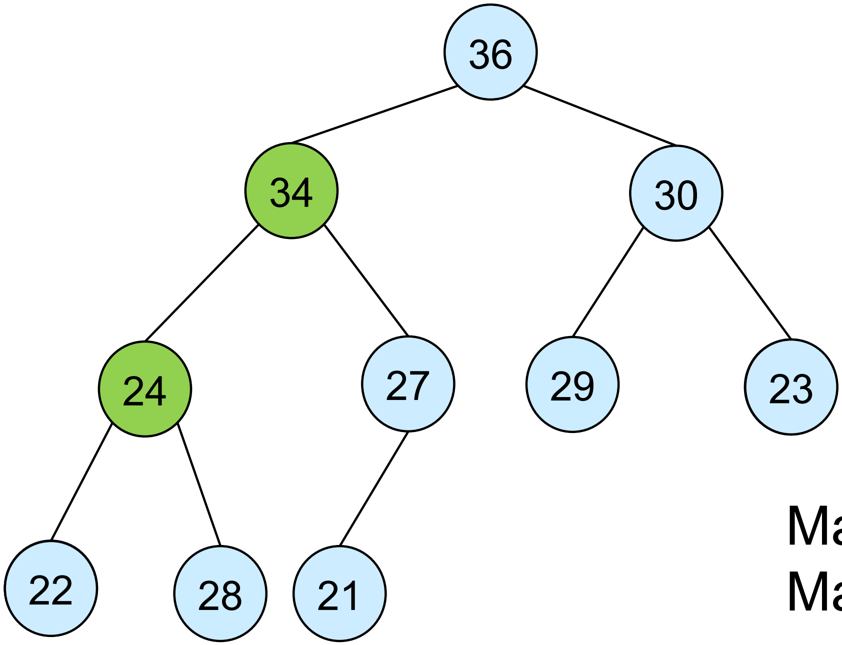

MaxHeapify($\lfloor 10/2 \rfloor = 5$)
MaxHeapify(4)
MaxHeapify(3)
MaxHeapify(2)
MaxHeapify(1)

# Building a heap - Example



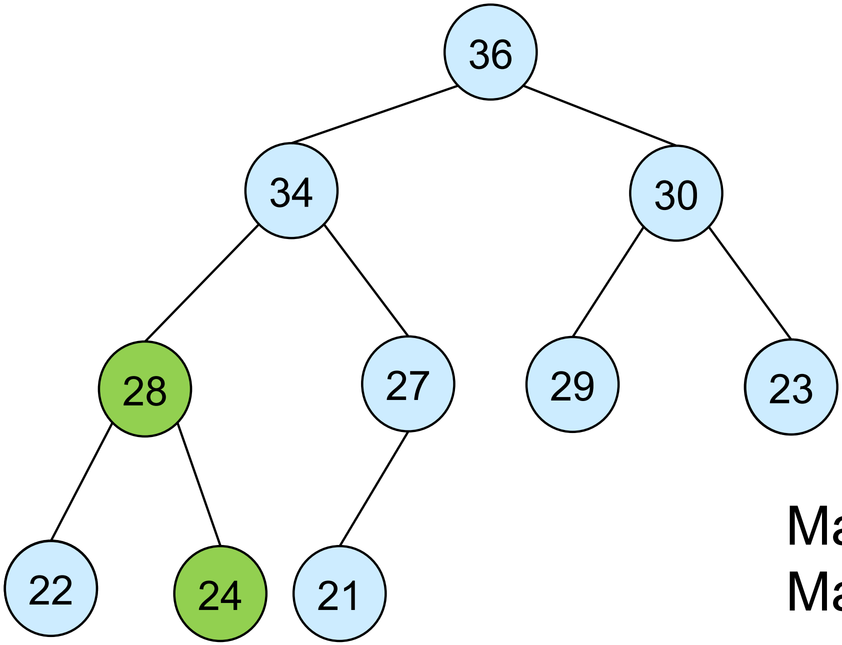

MaxHeapify($\lfloor 10/2 \rfloor = 5$)
MaxHeapify(4)
MaxHeapify(3)
MaxHeapify(2)
MaxHeapify(1)

# Building a heap - Example




MaxHeapify($\lfloor 10/2 \rfloor = 5$)
MaxHeapify(4)
MaxHeapify(3)
MaxHeapify(2)
MaxHeapify(1)

# Building a heap - Example




MaxHeapify($\lfloor 10/2 \rfloor = 5$)
MaxHeapify(4)
MaxHeapify(3)
MaxHeapify(2)
MaxHeapify(1)

# Building a heap - Correctness

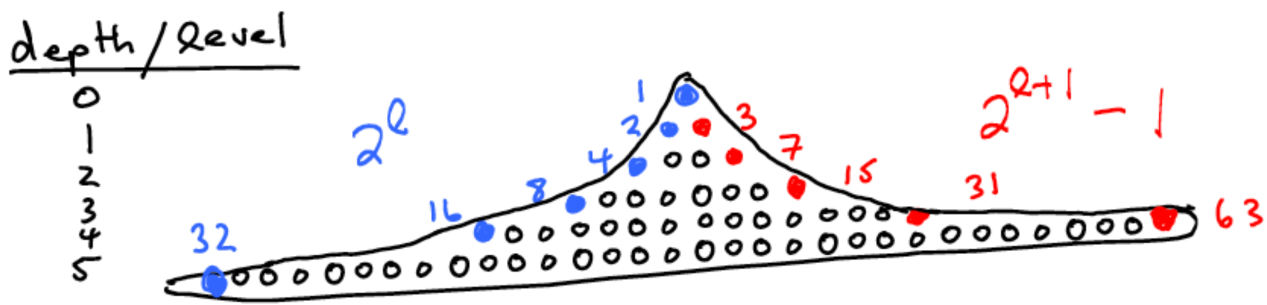
- **Loop Invariant property (LI):** At the start of each iteration of the **for** loop, each node $i+1$, $i+2$, …, $n$ is the root of a max-heap.
- **Initialization:**
  - Before first iteration $i = \lfloor n/2 \rfloor$
  - Nodes $\lfloor n/2 \rfloor+1$, $\lfloor n/2 \rfloor+2$, …, $n$ are leaves, thus max-heaps.
- **Maintenance:**
  - By LI, subtrees at children of node $i$ are max heaps.
  - Hence, MaxHeapify($i$) renders node $i$ a max heap root (while preserving the max heap root property of higher-numbered nodes).
  - Decrementing $i$ reestablishes the loop invariant for the next iteration.
- **Termination:**
  - bounded number of calls to MaxHeapify
  - At termination, $i$ =0. Then, each node 1,2,…,n is the root of a max-heap. In particular, node 1 is.

# Building a heap – Running Time

- **Loose upper bound:**
  - Cost of a *MaxHeapify* call × # calls to *MaxHeapify*
  - $O(\lg n) \times O(n) = O(n \lg n)$
- **Tighter bound:**
  - Cost of *MaxHeapify* is $O(h)$.
  - # of nodes with height $h \leq \lceil n/2^{h+1} \rceil$
  - Height of heap is $\lfloor \lg n \rfloor$

Taken from Langer2014

e.g. tree height $h = 5$



# Building a heap – Running Time

- **Tighter bound:**
  - Cost of *MaxHeapify* is $O(h)$.
  - #nodes with height $h \leq \lceil n/2^{h+1} \rceil$
  - Height of heap is $\lfloor \lg n \rfloor$

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

$$\frac{1}{1-x} = \sum_{n=0}^{\infty} x^n$$

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

$$= x^{-1} \sum_{n=0}^{\infty} n x^n$$

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left( n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = O(n)$$

Running time of BuildMaxHeap is O(n)

# Outline

- Introduction.
- Operations.
- Application.

# Application - Heapsort

- Combines the better attributes of merge sort and insertion sort.
  - Like merge sort, worst-case running time is $O(n \lg n)$.
  - Like insertion sort, sorts in place.
- Introduces an algorithm design technique
  - Create data structure (*heap*) to manage information during the execution of an algorithm.
- The *heap* has other applications beside sorting.
  - Priority Queues (recall COMP250)

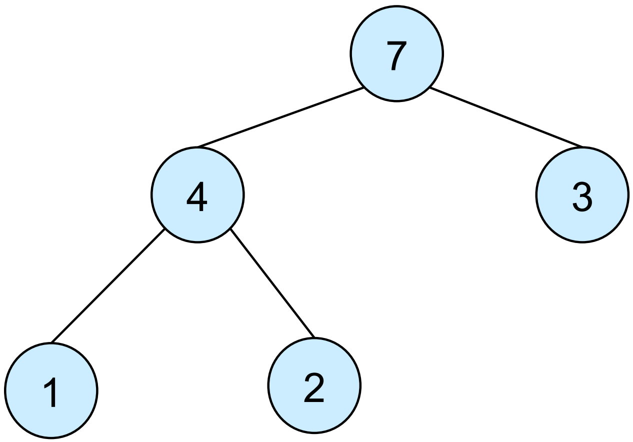
# Application - Heapsort

1. Builds a max-heap from the array.
2. Put the maximum element (i.e. the root) at the correct place in the array by swapping it with the element in the last position in the array.
3. “Discard” this last node (knowing that it is in its correct place) by decreasing the heap size, and call MAX-HEAPIFY on the new root.
4. Repeat this process (goto 2) until only one node remains.

```
HeapSort(A)
1.   Build-Max-Heap(A)
2.   for i ← length[A] downto 2
3.         do exchange A[1] ↔ A[i]
4.            MaxHeapify(A, 1, i-1)
```

# Heapsort - Example

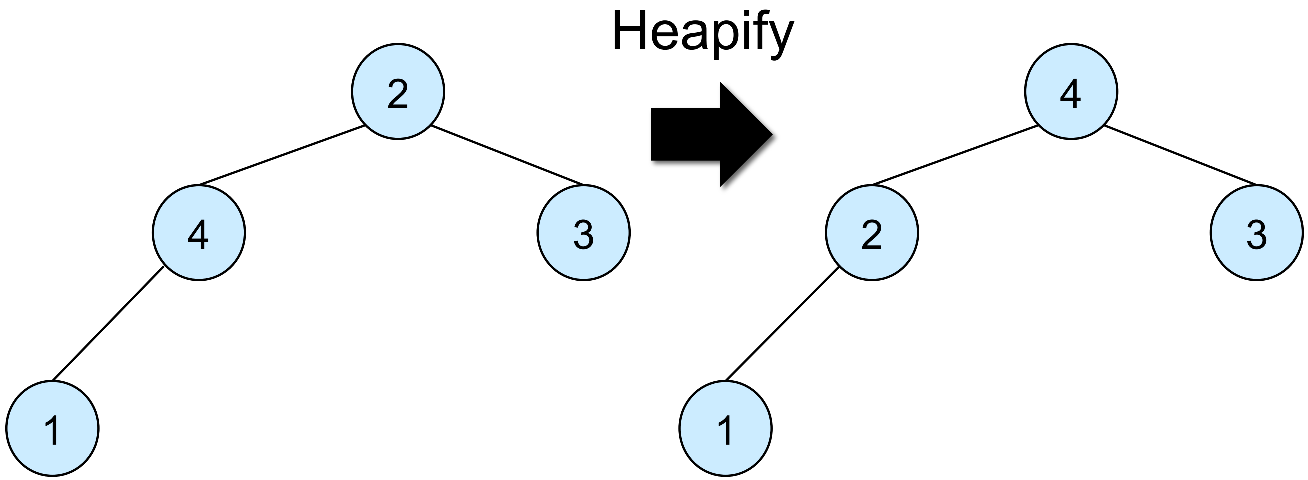
| | | | | |
|---|---|---|---|---|
| 7 | 4 | 3 | 1 | 2 |



# Heapsort - Example

| | | | | |
|---|---|---|---|---|
| 2 | 4 | 3 | 1 | 7 |

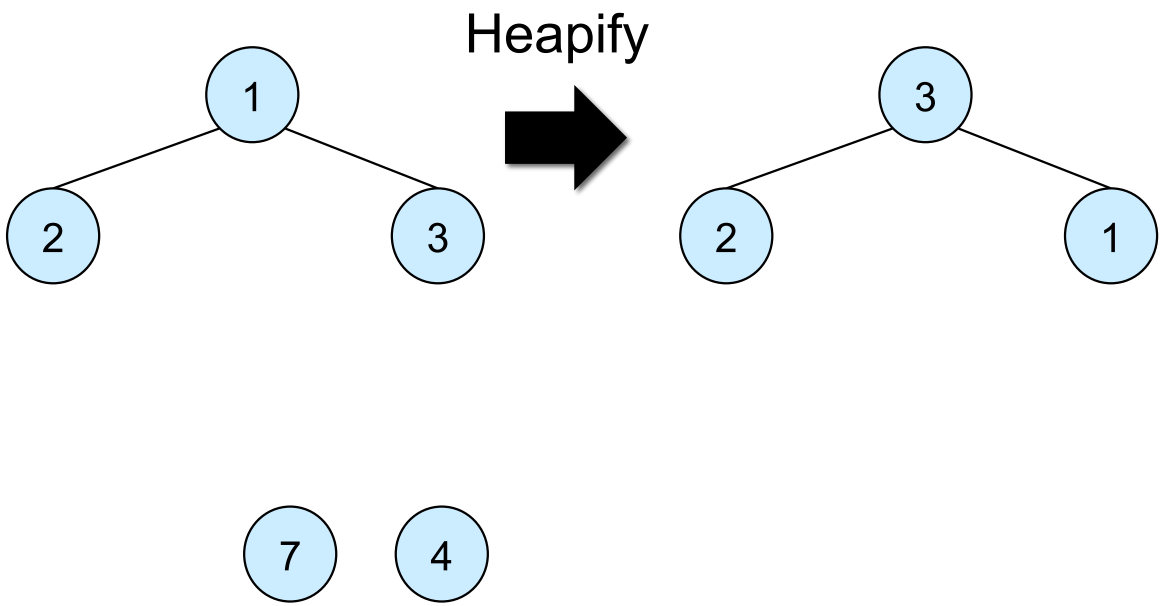
| | | | | |
|---|---|---|---|---|
| 4 | 2 | 3 | 1 | 7 |



# Heapsort - Example

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 7 |

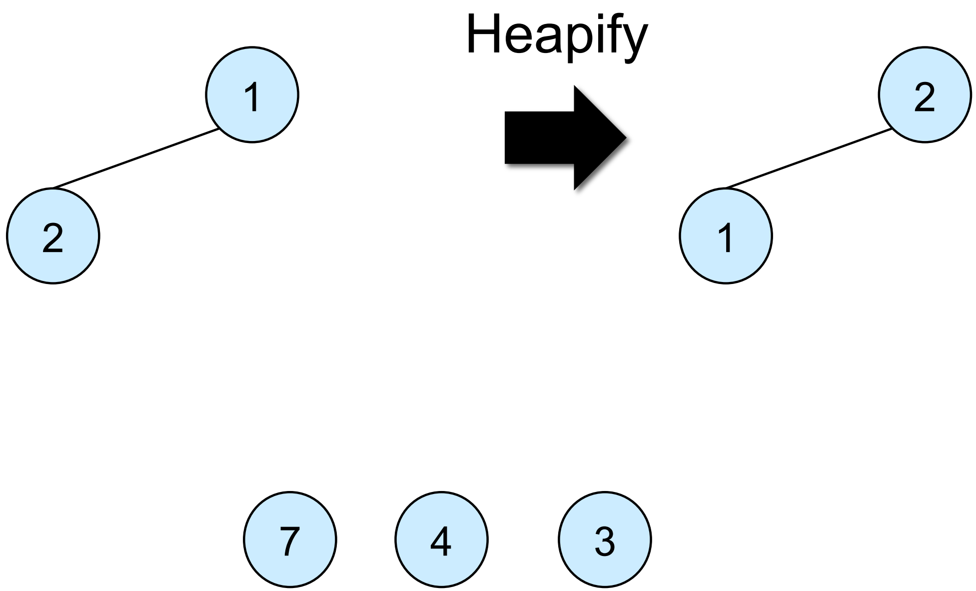
| | | | | |
|---|---|---|---|---|
| 3 | 2 | 1 | 4 | 7 |



# Heapsort - Example

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 7 |

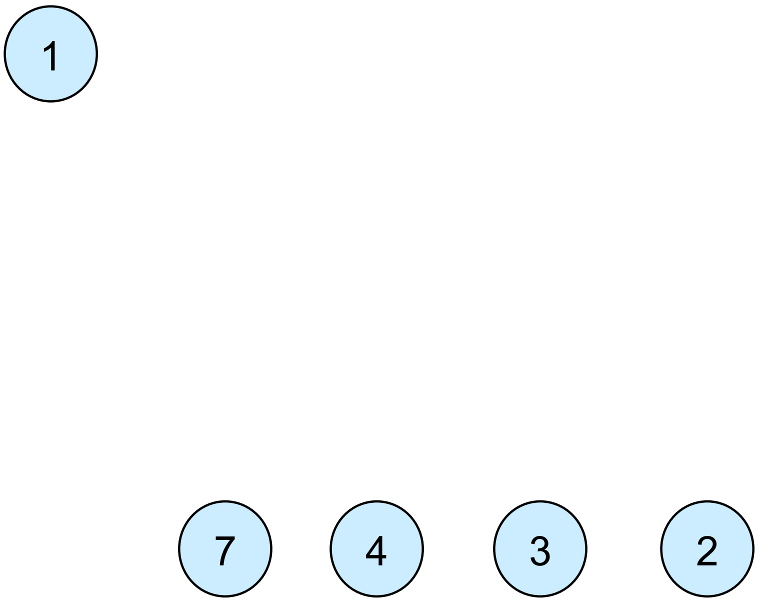
| | | | | |
|---|---|---|---|---|
| 2 | 1 | 3 | 4 | 7 |



# Heapsort - Example

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 7 |

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 7 |



# Heapsort - Example

- BuildMaxHeap $O(n)$
- for loop n-1 times (i.e. O(n))
  - exchange elements O(1)
  - MaxHeapify O(lg n)

=> HeapSort $O(n \lg n)$

# Outline

- Introduction.
- Operations.
- Application.


time for questions