COMP251: Heaps & Heapsort

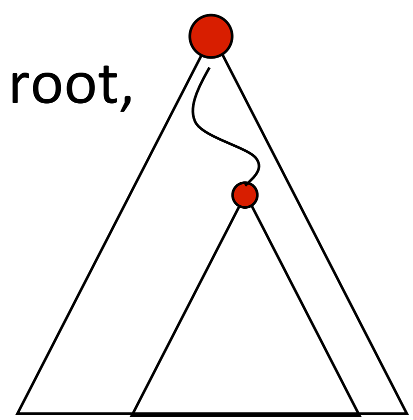
Jérôme Waldispühl
School of Computer Science
McGill University

From (Cormen et al., 2002)

Based on slides from D. Plaisted (UNC)

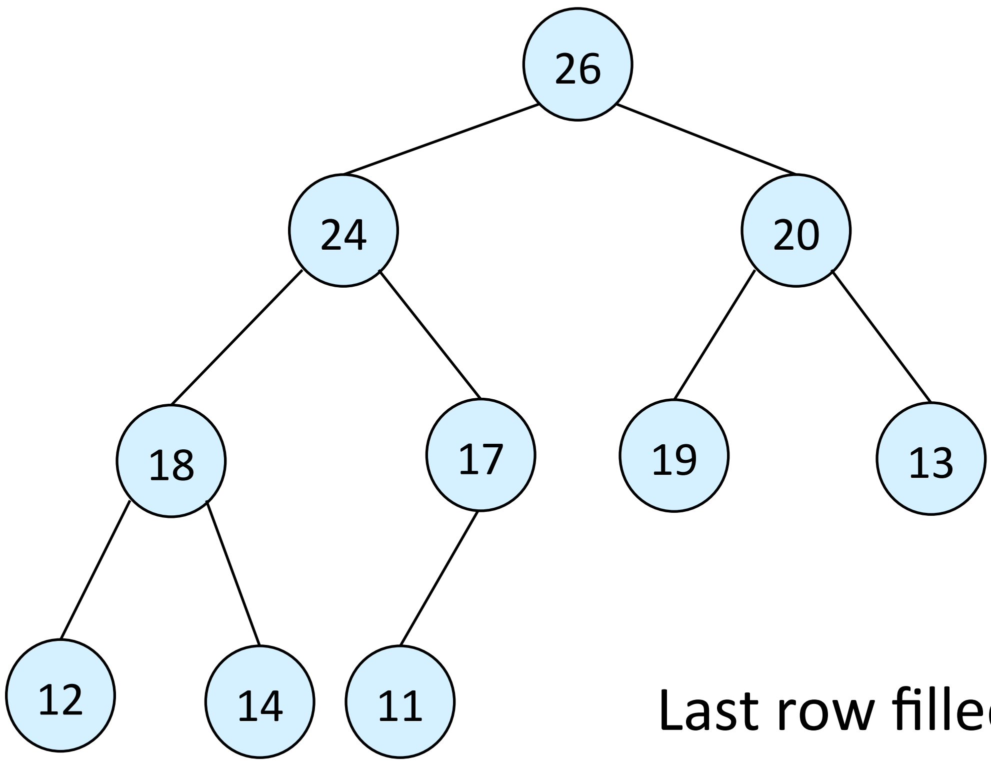
Heap data structure

- 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, $A[\text{PARENT}(i)] \geq A[i]$.
- Min-Heap
  - Smallest element is stored at the root.
  - for all nodes *i*, excluding the root, excluding the root, $A[\text{PARENT}(i)] \leq A[i]$.



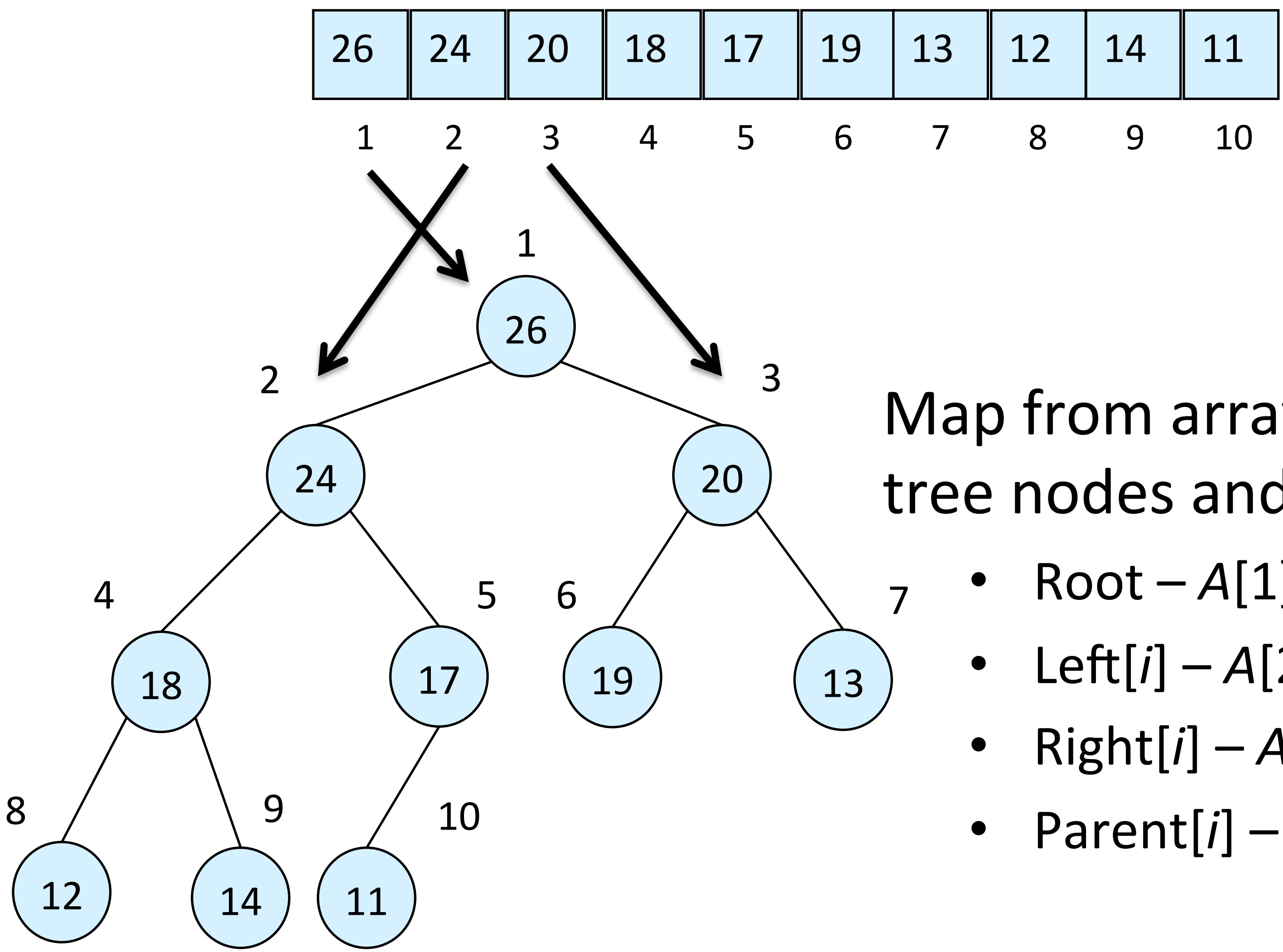
Heaps – Example

Max-heap as a binary tree.



Last row filled from left to right.

Heaps as arrays



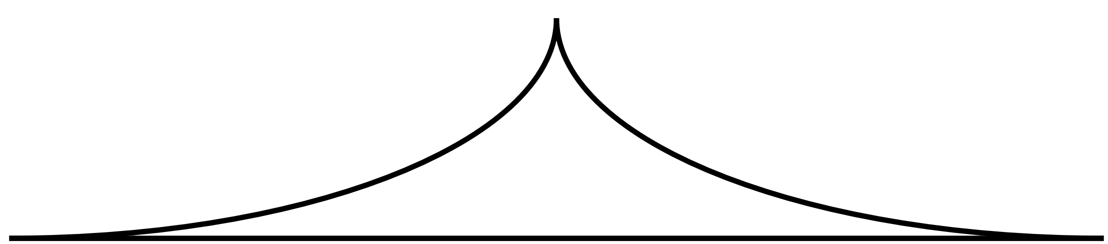
Max-heap as an array.

Map from array elements to tree nodes and vice versa

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

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 =* Θ(lg n).
- Most Basic operations on a heap run in $O(\lg n)$ time
- Shape of a heap



Sorting with Heaps

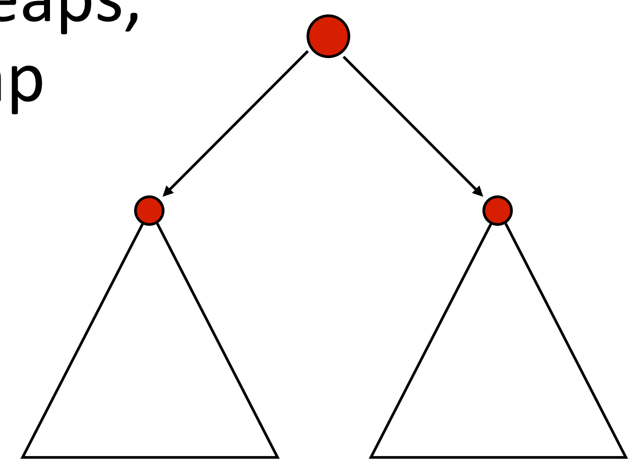
- Use max-heaps for sorting.
- The array representation of max-heap is not sorted.
- Steps in sorting
  - Convert the given array of size n to a max-heap (*BuildMaxHeap*)
  - Swap the first and last elements of the array.
    - Now, the largest element is in the last position – where it belongs.
    - That leaves $n - 1$ elements to be placed in their appropriate locations.
    - However, the array of first $n - 1$ elements is no longer a max-heap.
    - Float the element at the root down one of its subtrees so that the array remains a max-heap (MaxHeapify)
    - Repeat step 2 until the array is sorted.

Heapsort

- Combines the better attributes of merge sort and insertion sort.
  - Like merge sort, but unlike insertion sort, running time is $O(n \lg n)$.
  - Like insertion sort, but unlike merge 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 (See COMP250)

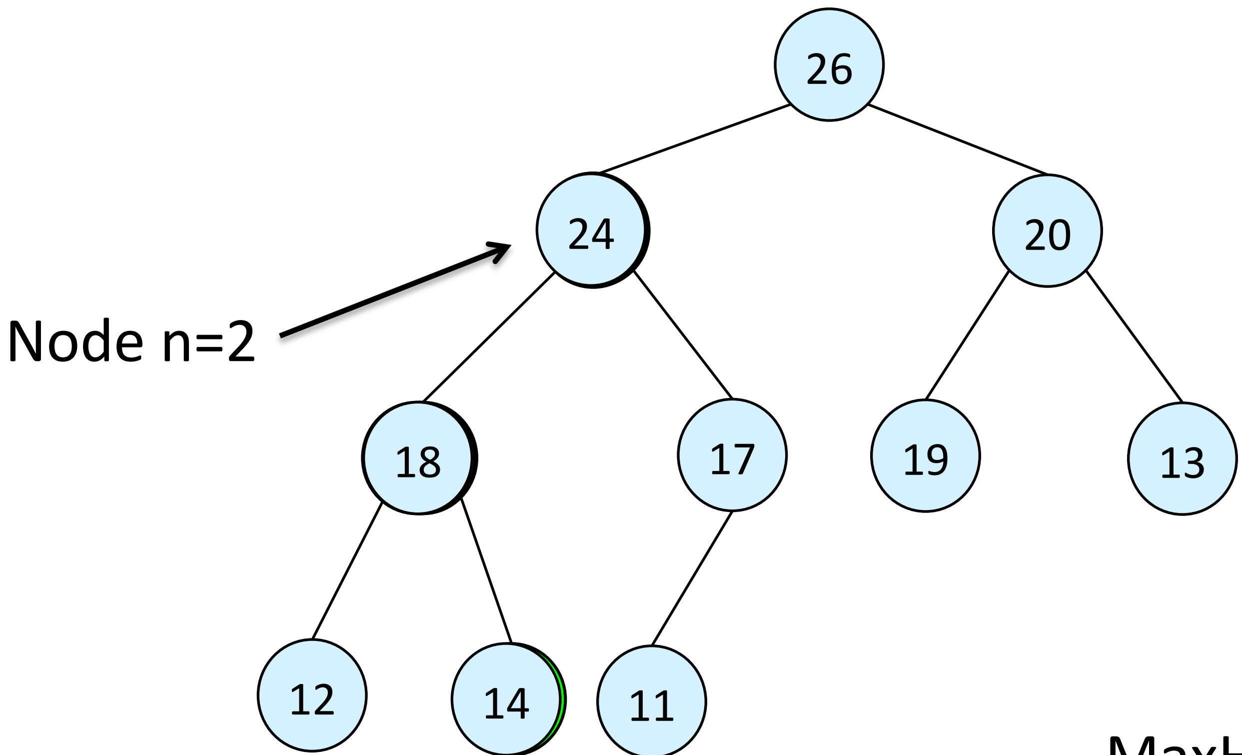
Maintaining the heap property

- Suppose two subtrees 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 subtree that is not a heap.
- Recursively fix the children until all of them satisfy the max-heap property.

MaxHeapify – Example



MaxHeapify(A, 2)

Procedure MaxHeapify

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

```
MaxHeapify(A, i, n)
   l ← leftNode(i)
   r ← rightNode(i)
   if l ≤ heap-size[A] 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)
```

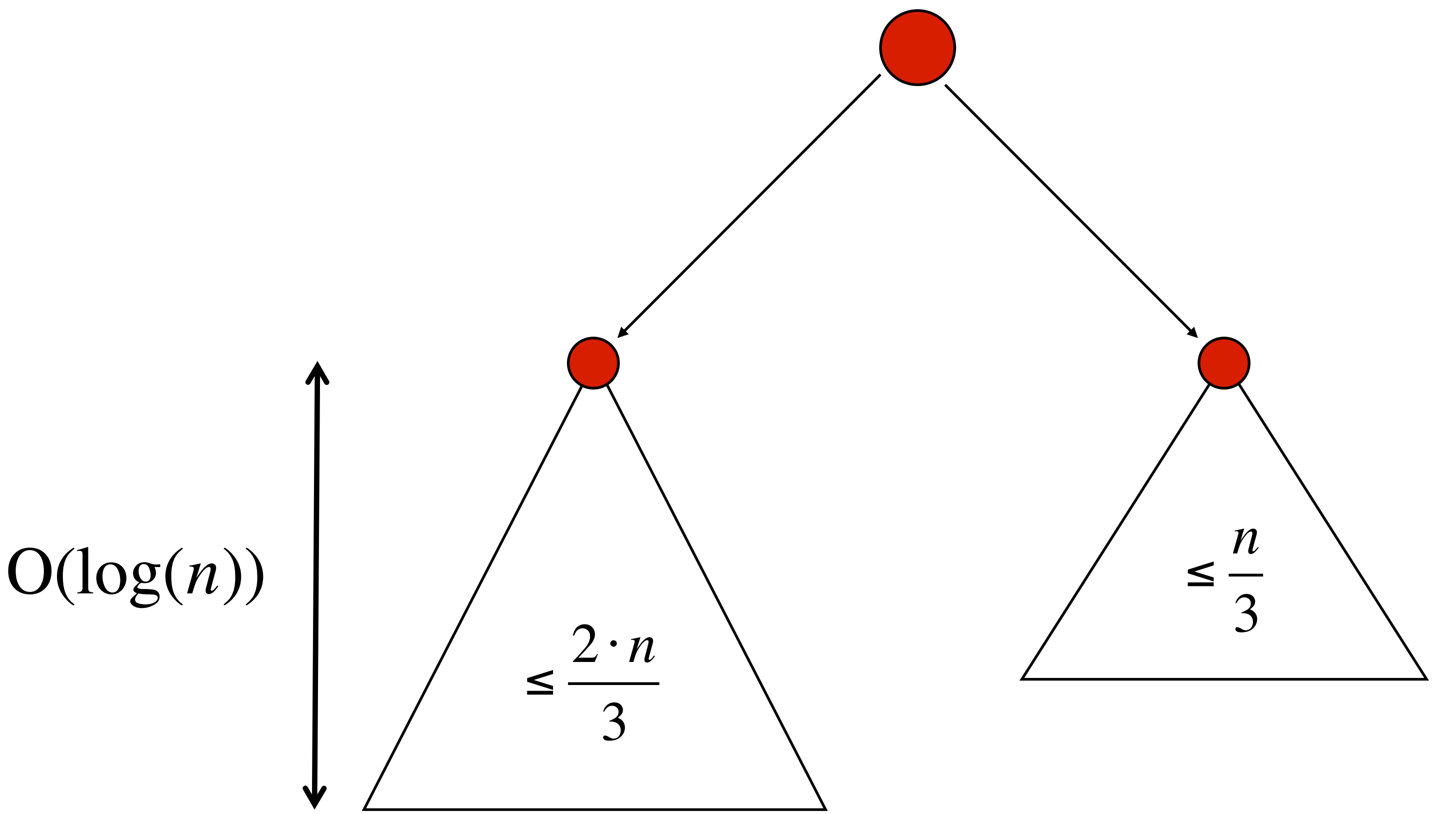
Time to fix node *i* and its children = **Θ(1)**

Time to fix the subtree rooted at one of *i*'s children = *T*(size of subtree)

Worst case running time of MaxHeapify(A, n)

- $T(n) = T(largest) + \Theta(1)$
- *largest* $\leq 2n/3$ (worst case occurs when the last row of tree is exactly half full)
- $T(n) \leq T(2n/3) + \Theta(1) \Rightarrow T(n) = O(\lg n)$
- Alternately, MaxHeapify takes $O(h)$ where h is the height of the node where MaxHeapify is applied

Worst case running time of MaxHeapify(*A*, *n*)



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>

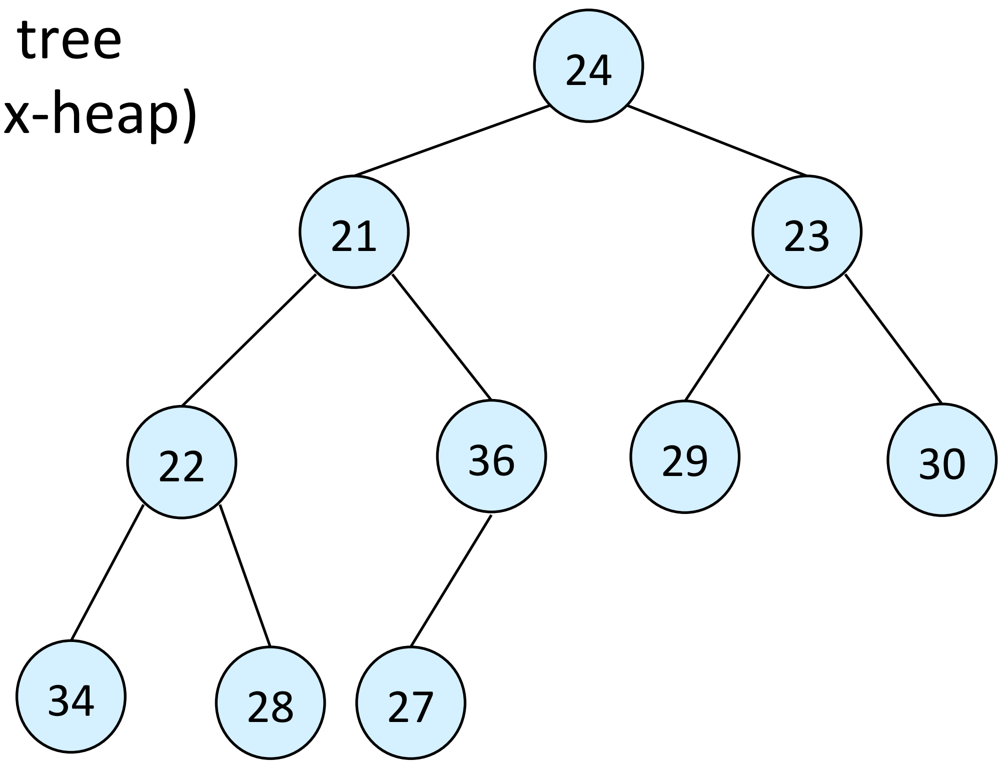
```
n ← length[A]
for i ← ⌊length[A]/2⌋ downto 1
        do MaxHeapify(A, i, n)
```

BuildMaxHeap – Example

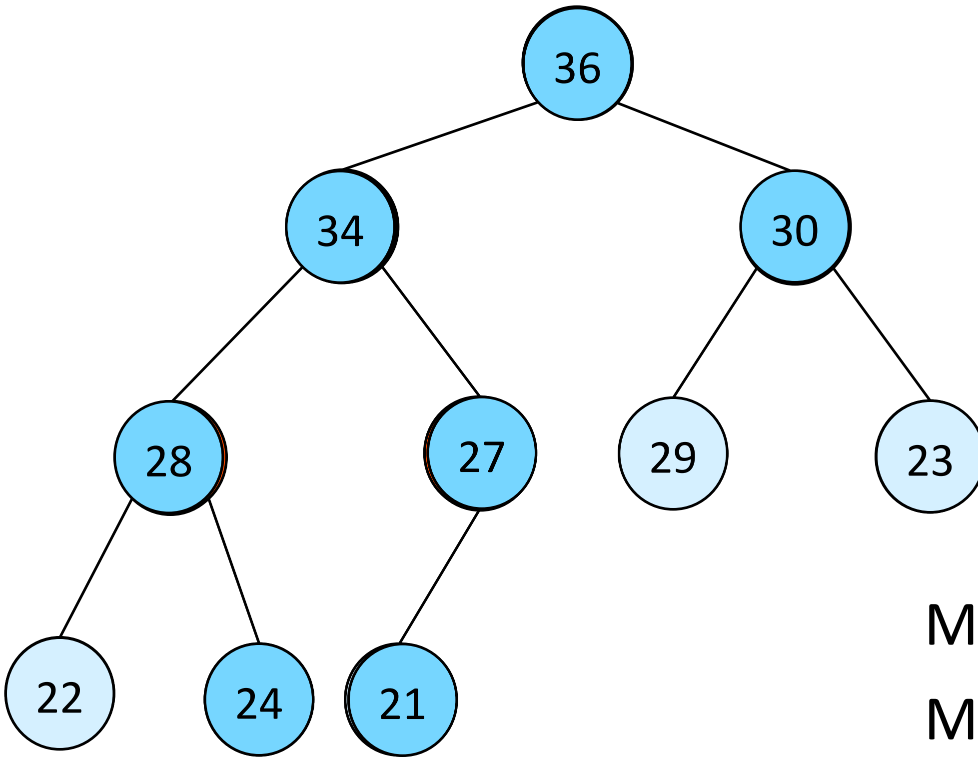
Input Array:

24	21	23	22	36	29	30	34	28	27

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



BuildMaxHeap – Example



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

MaxHeapify(4)

MaxHeapify(3)

MaxHeapify(2)

MaxHeapify(1)

Correctness of *BuildMaxHeap*

- **Loop Invariant:** 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, hence roots of trivial 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.

Running Time of *BuildMaxHeap*

- **Loose upper bound:**
  - Cost of a *MaxHeapify* call × No. of calls to *MaxHeapify*
  - $O(\lg n) \times O(n) = O(n \lg n)$
- **Tighter bound:**
  - Cost of *MaxHeapify* is $O(h)$.
  - $\leq \lceil n/2^{h+1} \rceil$ nodes of height h.
  - Height of heap is $\lfloor \lg n \rfloor$

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

Running time of BuildMaxHeap is O(n)

Heapsort

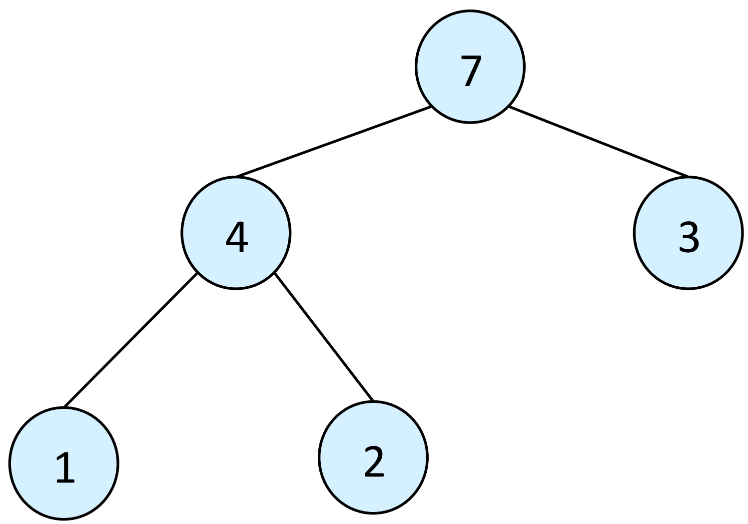
1. Builds a max-heap from the array.
2. Put the maximum element (i.e. the root) at the correct place in the array by swapping it with the element in the last position in the array.
3. “Discard” this last node (knowing that it is in its correct place) by decreasing the heap size, and call MAX-HEAPIFY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

Heapsort(A)

```
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

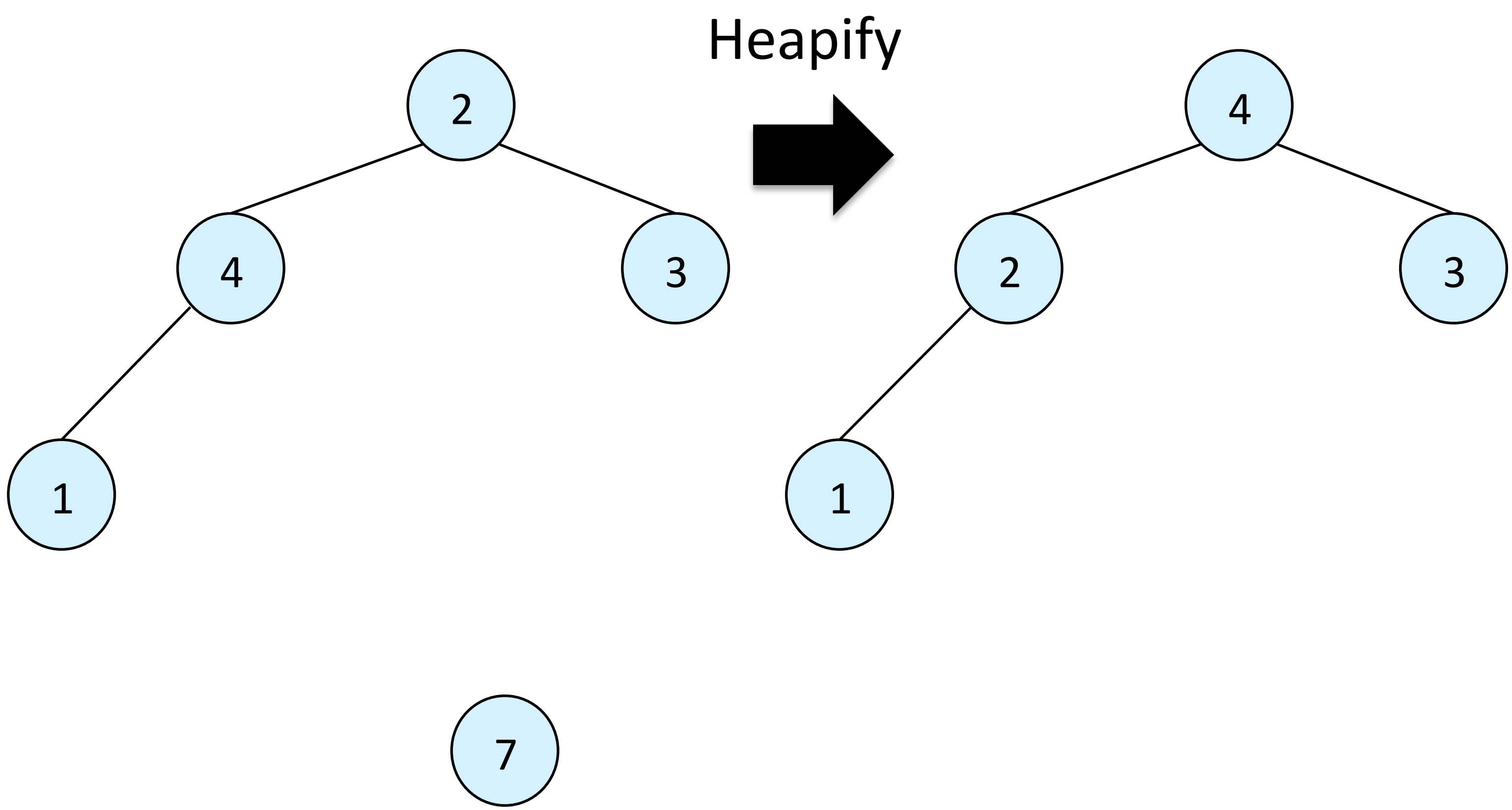
7	4	3	1	2



Heapsort – Example

2	4	3	1	7

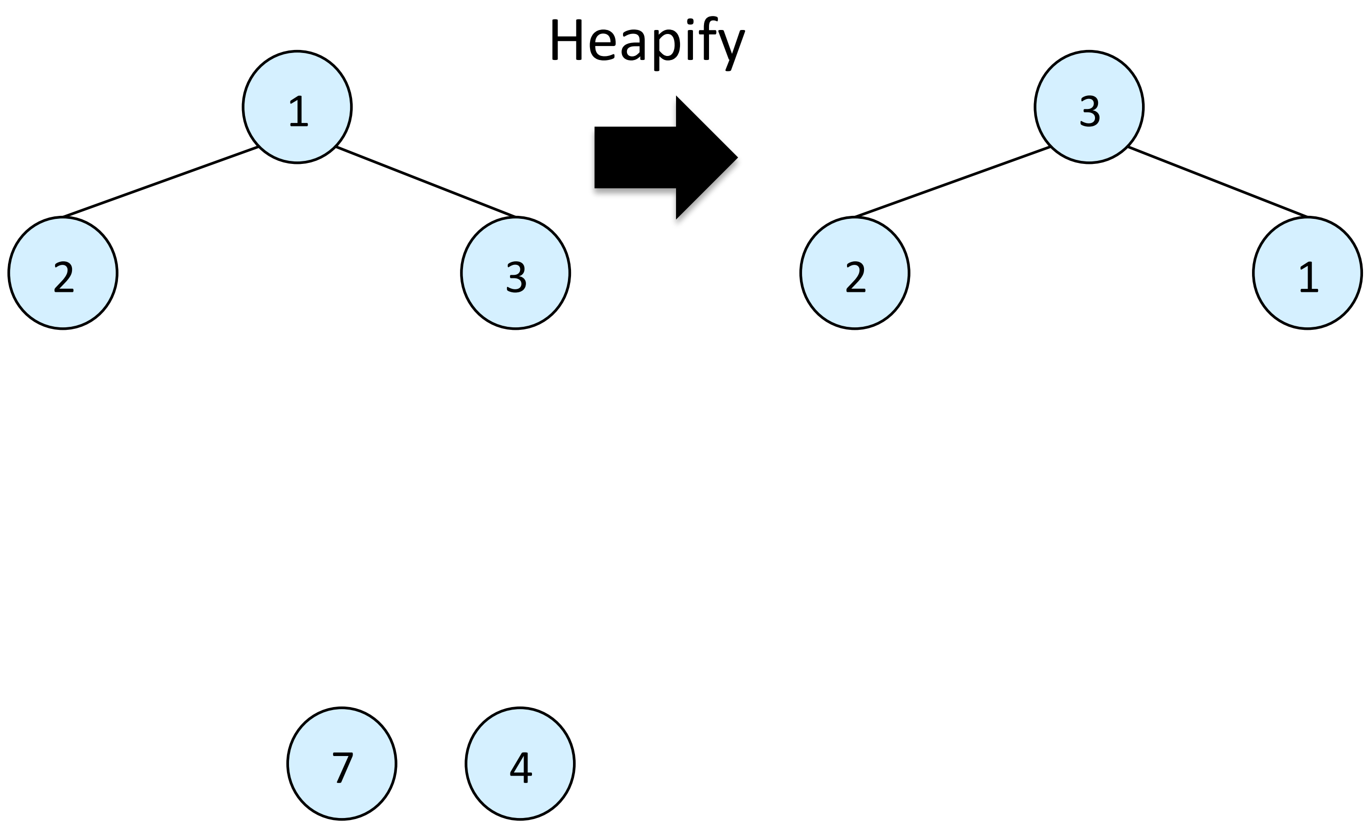
4	2	3	1	7



Heapsort – Example

1	2	3	4	7

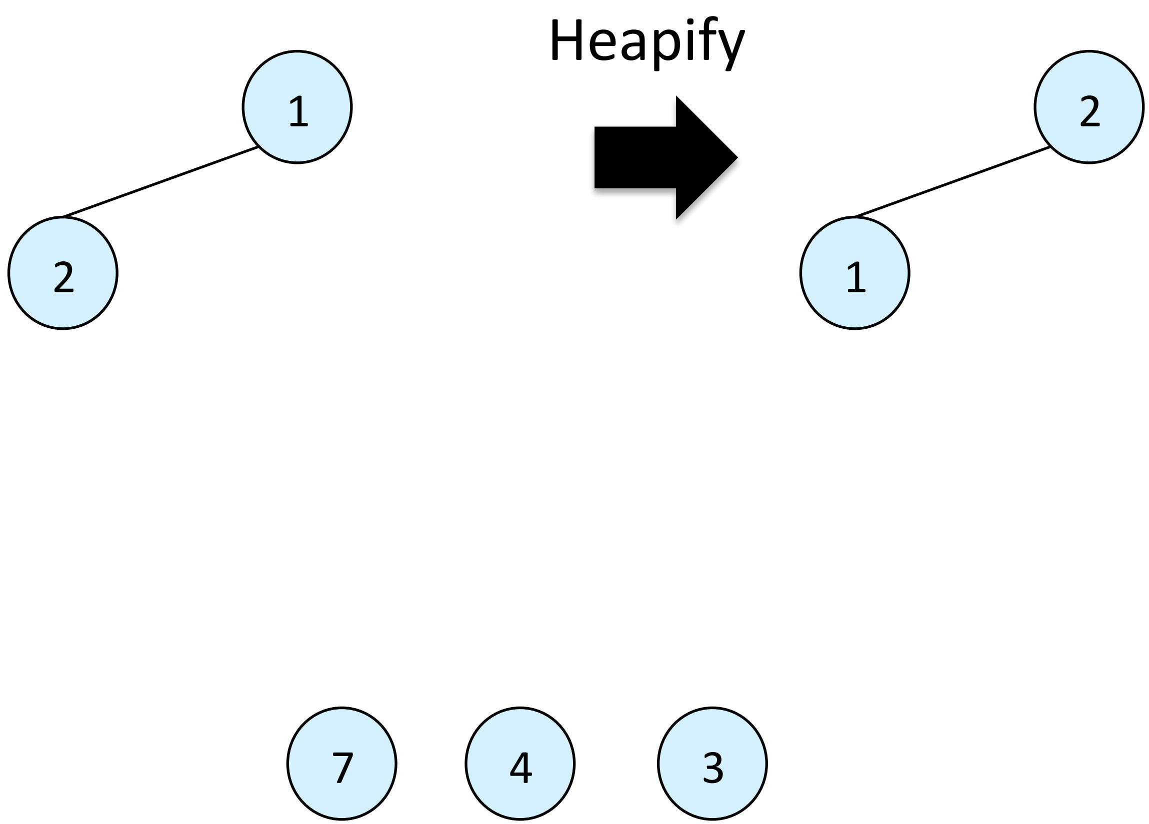
3	2	1	4	7



Heapsort – Example

1	2	3	4	7

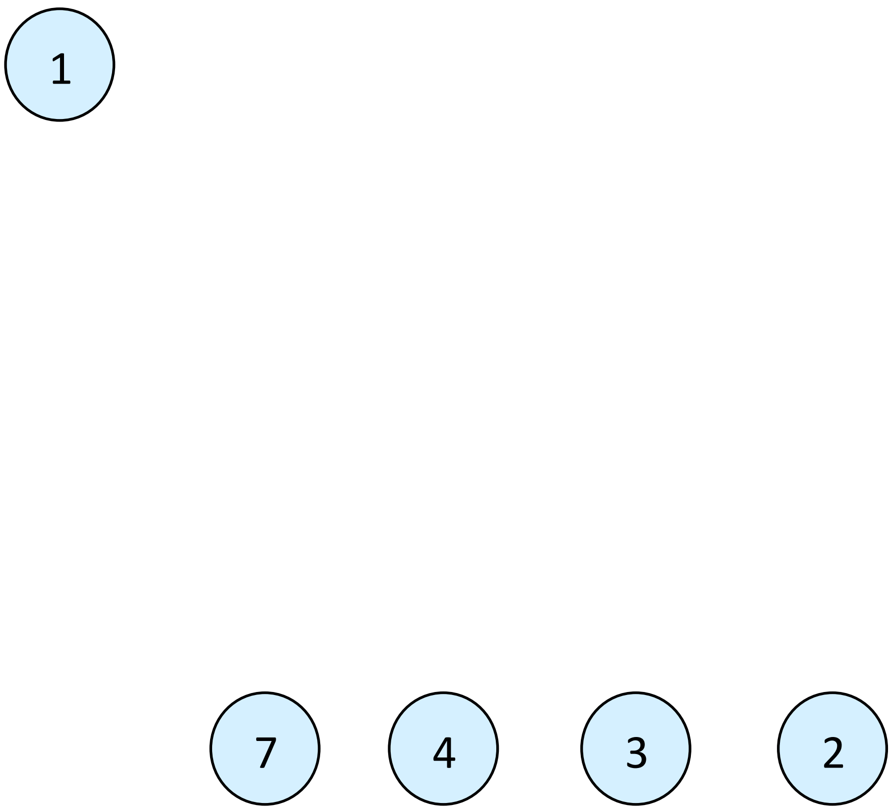
2	1	3	4	7



Heapsort – Example

1	2	3	4	7

1	2	3	4	7



Heap Procedures for Sorting

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

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