CS302

Topic: Priority Queues / Heaps

Tuesday, Sept. 26, 2006

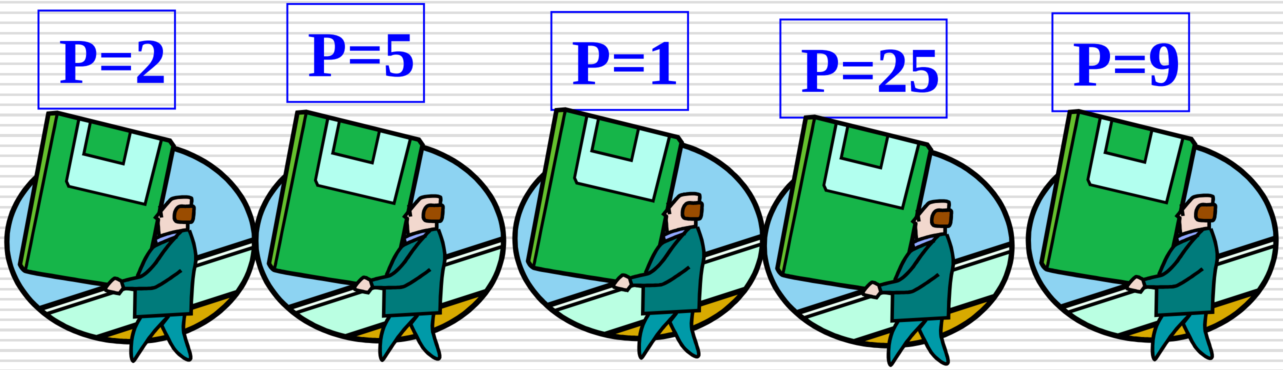
Announcements

- Lab 3 (Graphical Stock Charts); due Monday, Oct. 2
- Don’t procrastinate!!

I love deadlines. I like the whooshing sound as they make as they fly by.

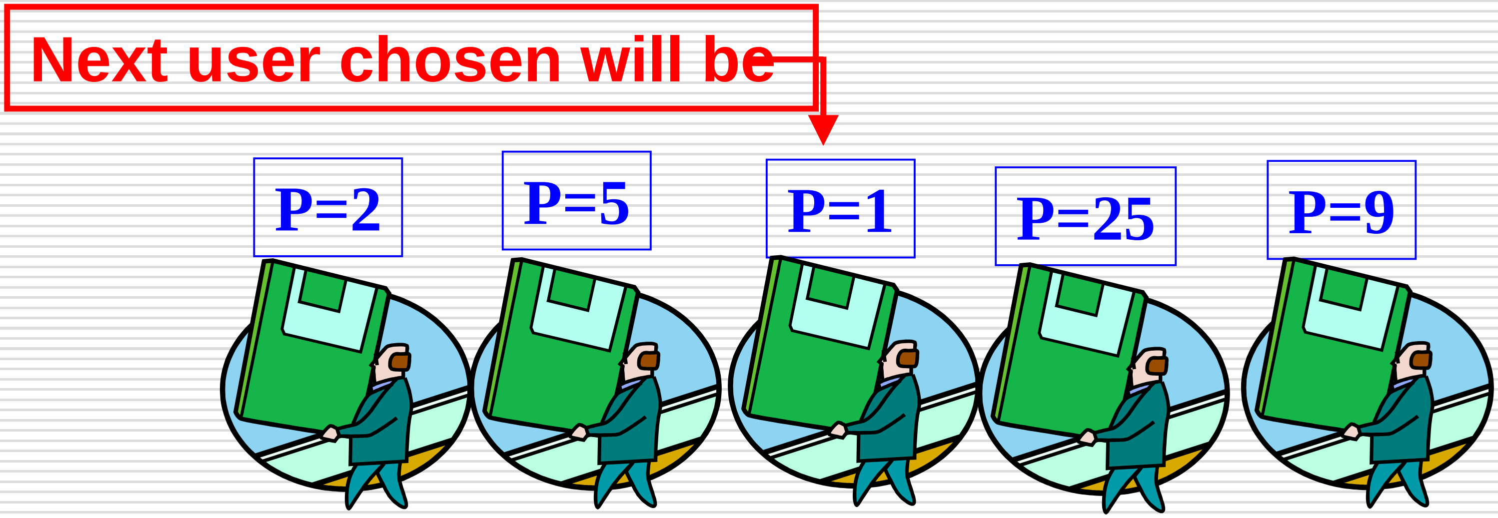
-- Douglas Adams

The Priority Queue



- We call it a priority *queue* - but its not FIFO
- Items in queue have PRIORITY
- Elements are removed from priority queue in either increasing or decreasing priority
  - Min Priority Queue
  - Max Priority Queue

The Priority Queue (Example #1)

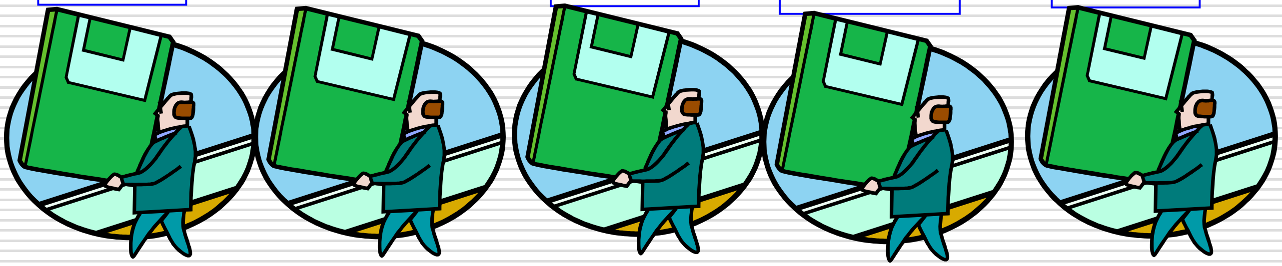


- Consider situation where we have a computer whose services we are selling
- Users need different amounts of time
- Maximize earnings by min priority queue of users
  - i.e. when machine becomes free, the user who needs least time gets the machine; get through more users quicker

The Priority Queue (Example #2)

Next user chosen will be

P=2 **P=5** **P=1** **P=25** **P=9**



- Consider situation where users are willing to pay more to secure access - they are in effect bidding against each other
- Maximize earnings by max priority queue of users
  - i.e. when machine becomes free, the user who is willing to pay most gets the machine

The Priority Queue

- Priority queue is a common data structure used in CS

- Example Applications:
  - Unix/Linux job scheduling
    - Processes given a priority
    - Time allocated to process is based on priority of job
  - Priority of jobs in printer queue
  - Sorting
  - Standby passengers at airport
  - Auctions
  - Stock market
  - Event-driven simulations
  - VLSI design (channel routing, pin layout)
  - Artificial intelligence search algorithms

Min Priority Queue ADT (i.e, Abstract Data Type)

- Instances:

  Finite collection of zero or more elements; each has a priority; represented as Key-Value pair

- Main Operations of a Min Priority Queue:

  insert(key, value) : Add element into the priority queue

  deleteMin() : Remove the element with the min priority

- Additional Operations that are often supplied:

  minKey() : Return the key with the minimum priority

  minVal(): Return the value of the node with min priority

  size() : Return number of elements in the queue

  empty() : Return true if the queue is empty; else false

(We'll look at STL C++ interface/implementation next time)

Total Order Relations

- Keys in a priority queue can be arbitrary objects on which an order is defined
- Mathematical concept of “total order relation $\leq$”:
  - Reflexive property:
    - $x \leq x$
  - Antisymmetric property:
    - $x \leq y \wedge y \leq x \Rightarrow x = y$
  - Transitive property:
    - $x \leq y \wedge y \leq z \Rightarrow x \leq z$
- Two distinct entries in a priority queue can have the same key

Priority Queue Implementation Options

- Unsorted linear list

  4 — 5 — 2 — 3 — 1

- Time complexities:
  - insert: $O(1)$, since we can insert item at the beginning or end of sequence
  - deleteMin: $O(n)$, since we have to traverse entire sequence to find smallest key

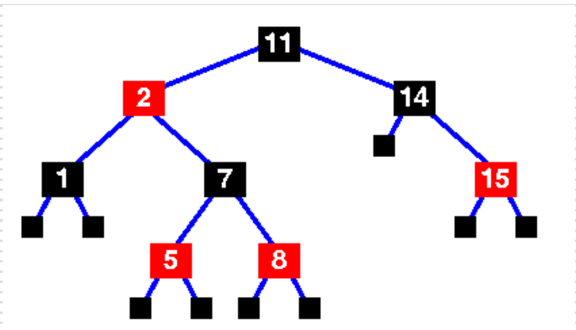
- Sorted linear list

  1 — 2 — 3 — 4 — 5

- Time complexities:
  - insert: $O(n)$, since we have to find the place to insert the item
  - deleteMin: $O(1)$, since the smallest key is at the beginning

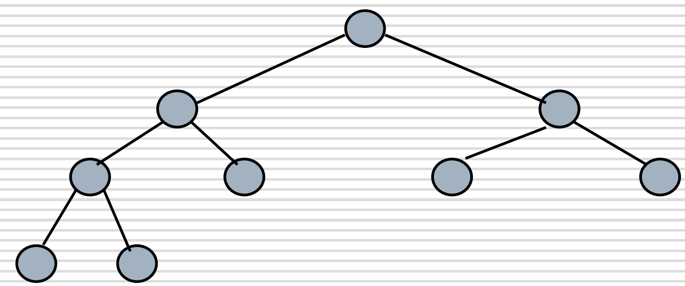
Priority Queue Implementation Options (con't.)

- Red-black tree



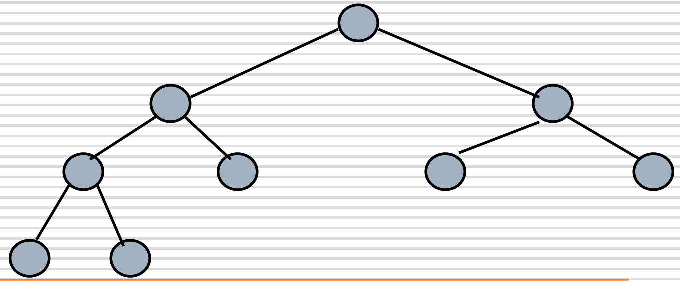
- Time complexities:
  - insert: $O(\log n)$
  - deleteMin: $O(\log n)$

- Heap



- Time complexities:
  - insert: $O(1)$, on average
  - deleteMin: $O(\log n)$

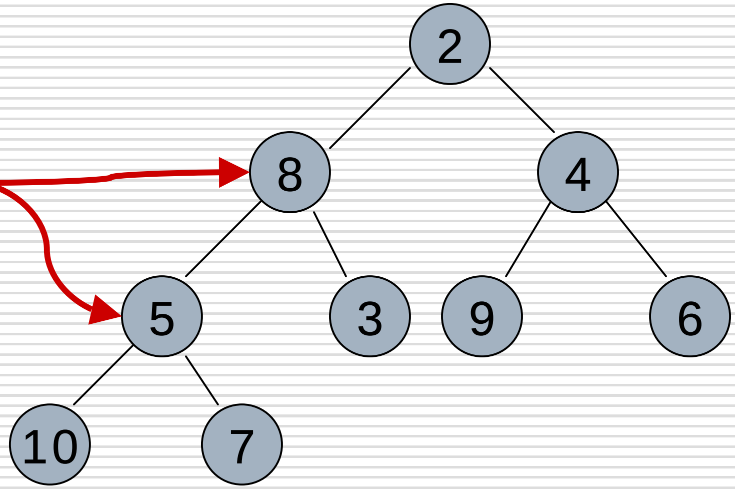
Heaps



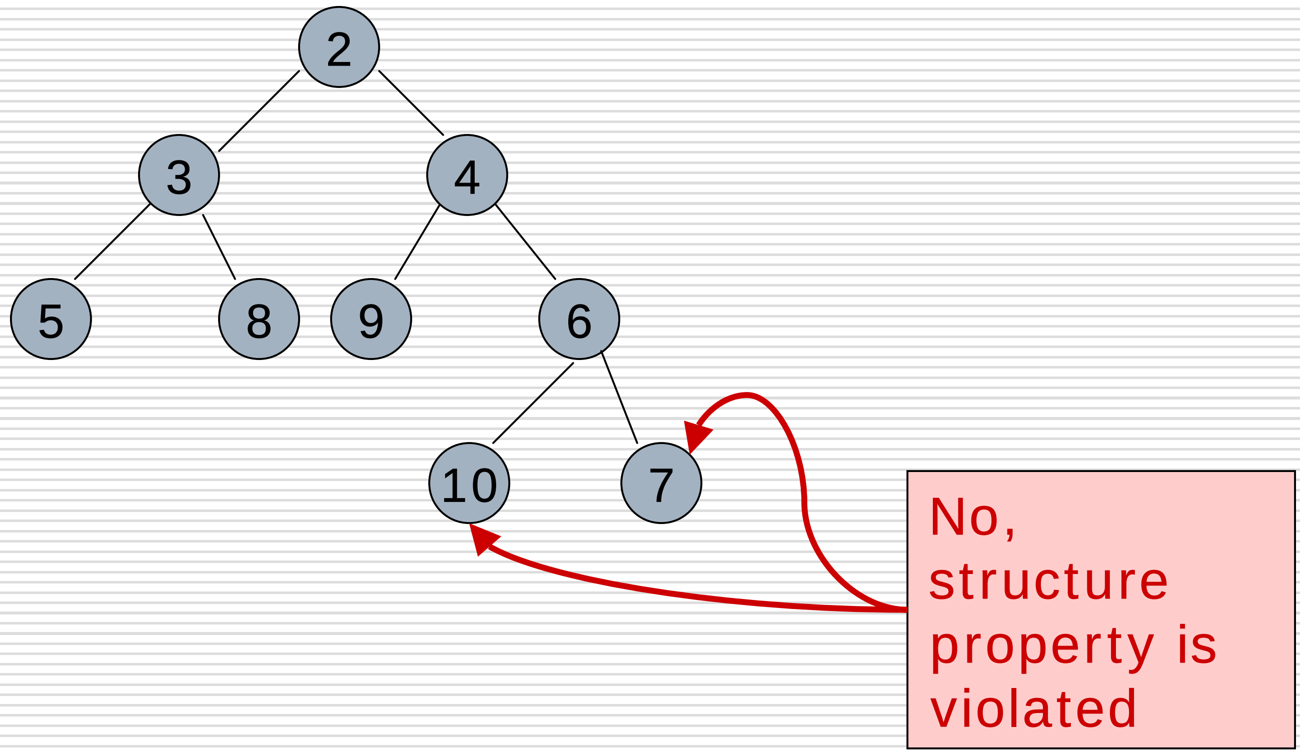
- A heap is a binary tree storing keys at its nodes and satisfying the following properties:
  - Structure property:
    - Complete binary tree. Let h be the height of the heap. Then:
      - For $i = 0, \ldots, h - 1$, there are 2^i nodes of depth i
      - At depth $h - 1$, the internal nodes are to the left of the external nodes
    - In other words, tree is completely filled, with possible exception of last level, which is filled from left to right
  - Heap-Order property (for a min heap):
    - For every internal node v other than the root,
      - $key(v) \geq key(parent(v))$

Examples: Is the following a min heap?

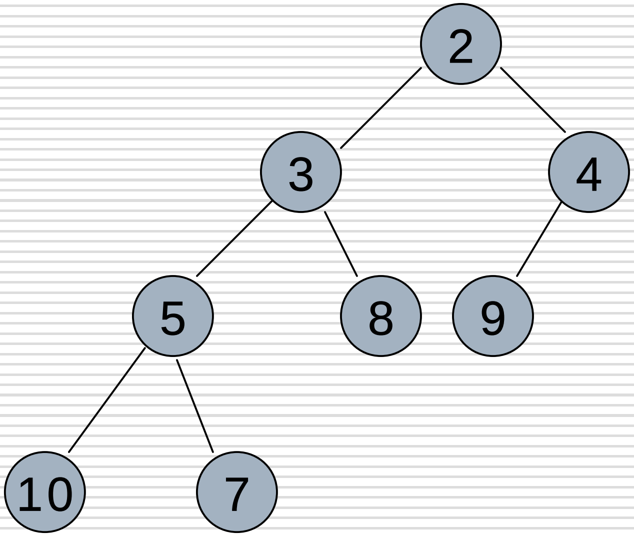
No, heap-order property is violated



Examples: Is the following a min heap?



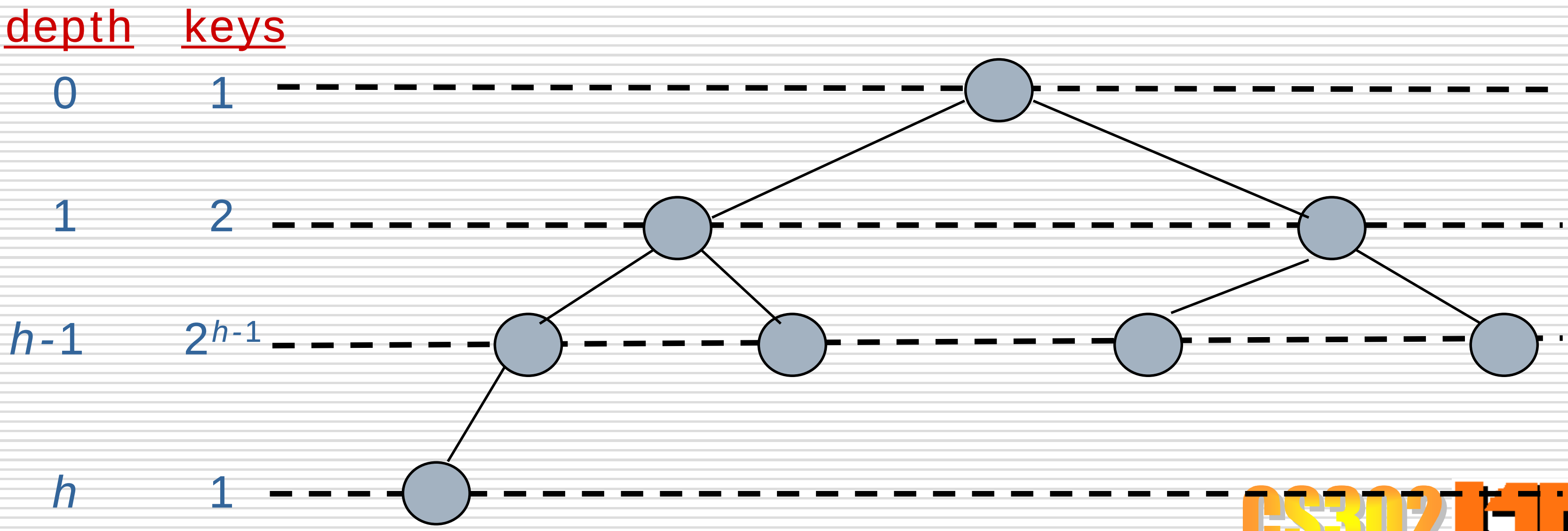
Examples: Is the following a min heap?



No, structure property is violated

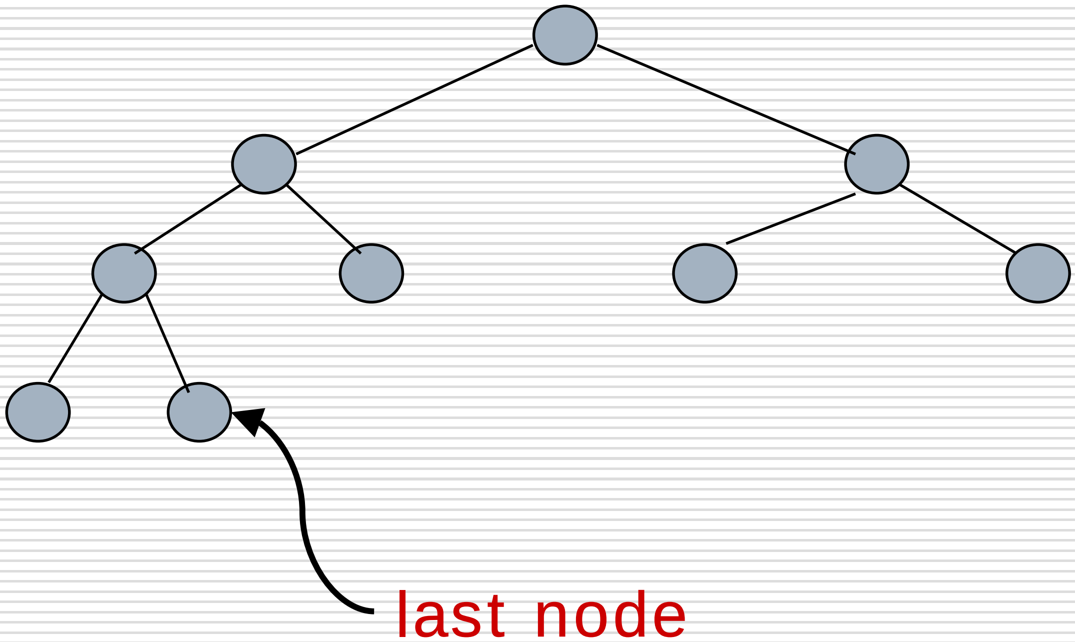
Height of a Heap

- **Theorem:** A heap storing n keys has height $O(\log n)$
- **Proof:** (we apply the complete binary tree property)
  - Let h be the height of a heap storing n keys
  - Since there are 2^i keys at depth $i = 0, \ldots, h-1$ and at least one key at depth h, we have $n \geq 2^h \Rightarrow h \leq \log n$



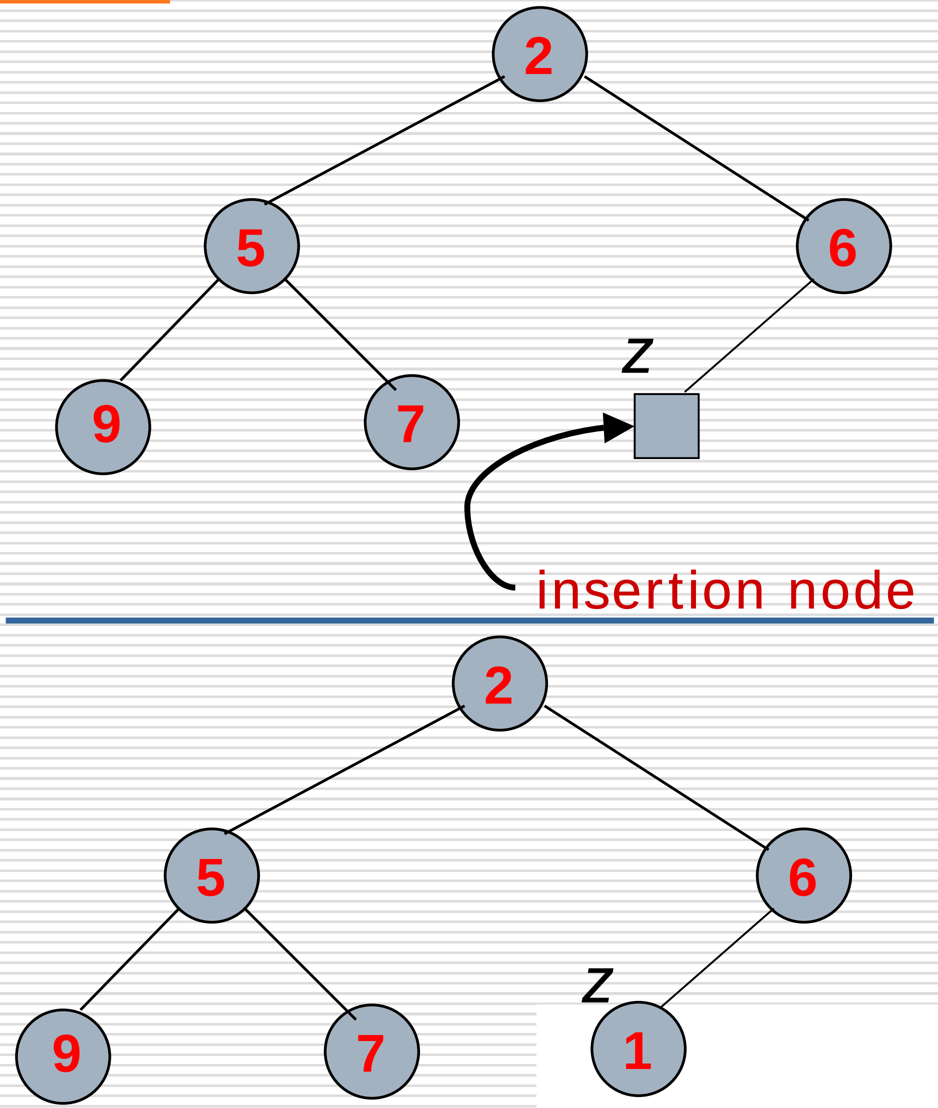
“Last node” of a Heap

- Define “last node” of a heap as the rightmost node of depth *h*



Insertion of key *k* into a Heap

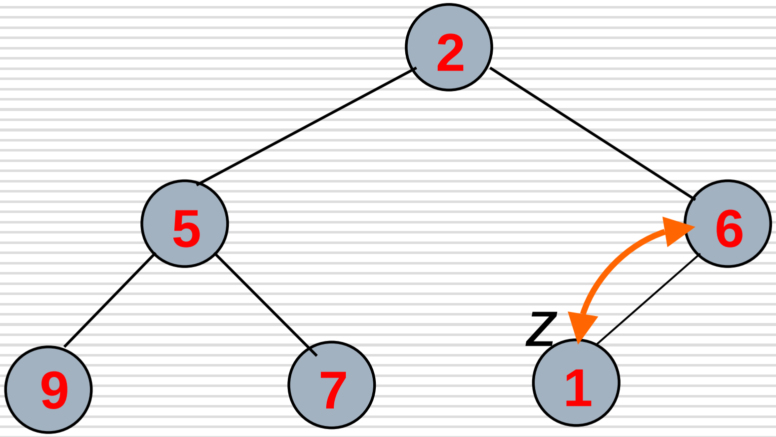
- Insert algorithm has 3 steps:
  - Find the insertion node *z* (i.e., the new last node)
  - Store *k* at *z*
  - Restore the heap-order property (we'll discuss this next)

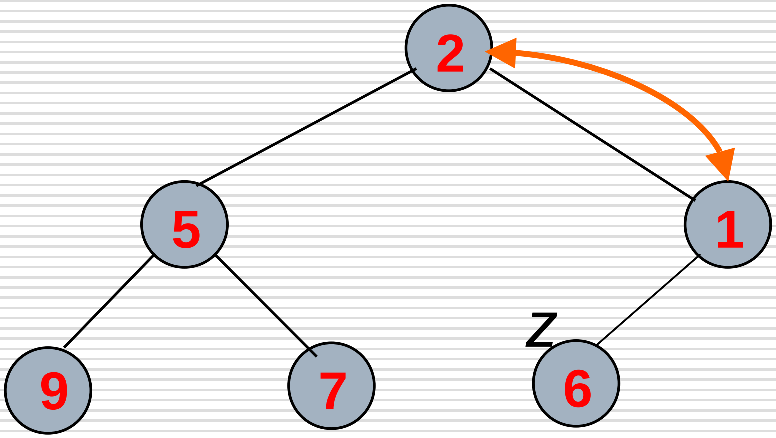


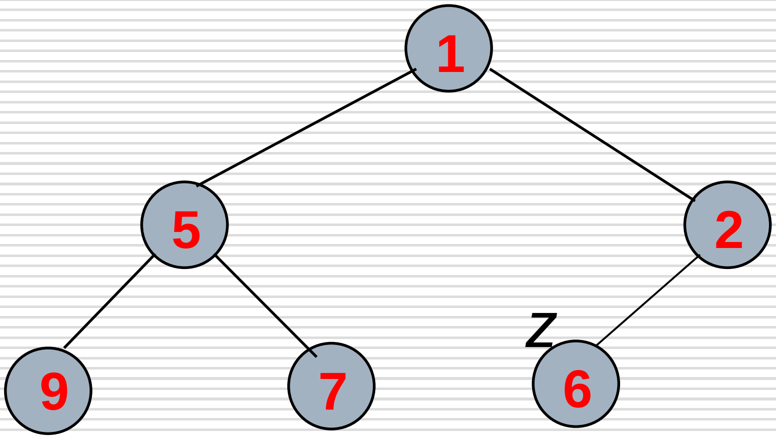
Restoring heap property after insert

- After the insertion of a new key *k*, the heap-order property may be violated
- "Percolate up": Restore heap-order property by swapping *k* along an upward path from the insertion node
- Terminate when key *k* reaches the node or a root whose parent has a key $\leq k$
- Since heap has height $O(\log n)$, restoring heap-order can be done in $O(\log n)$ time
  - But, average case requires 2.607 comparisons ➔ 1.607 element moves ➔ O(1) average time

Example:
Restoring Heap Property after insert

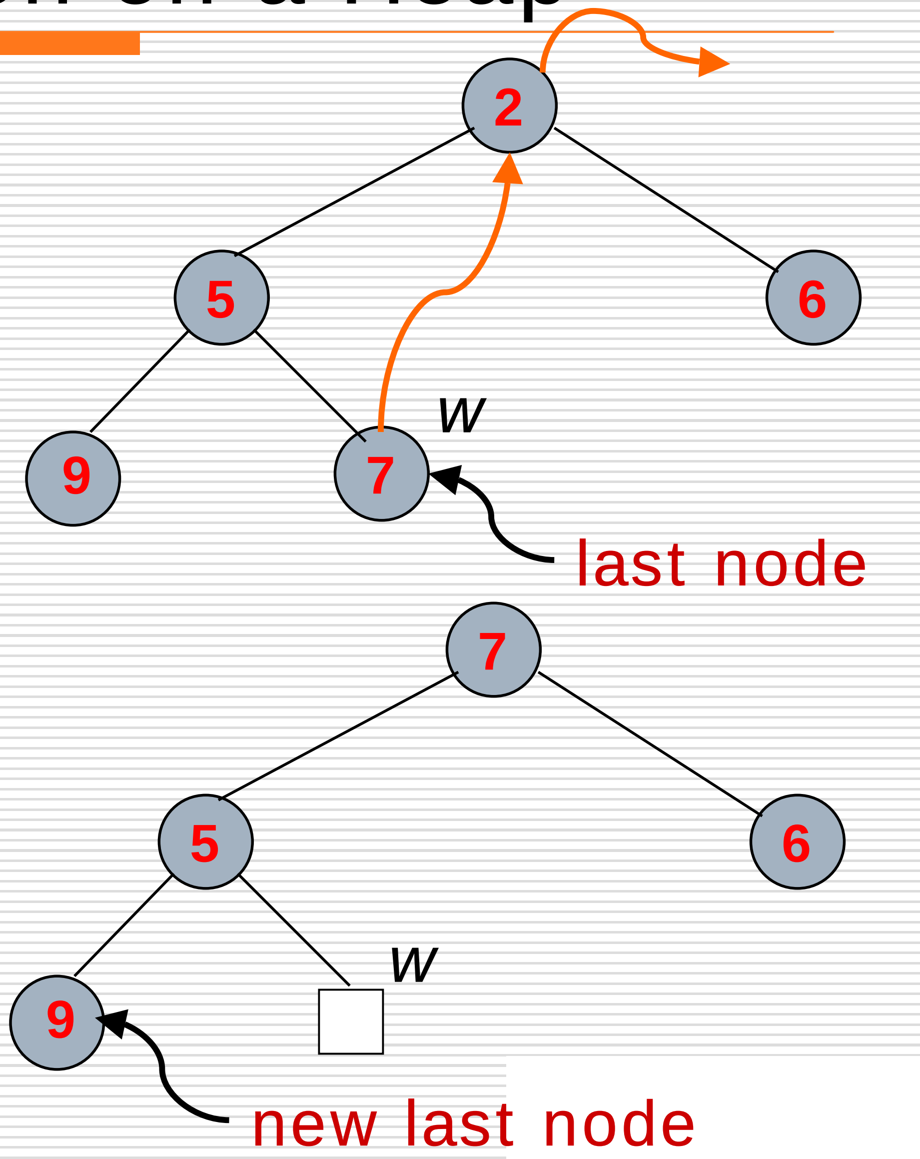






deleteMin operation on a Heap

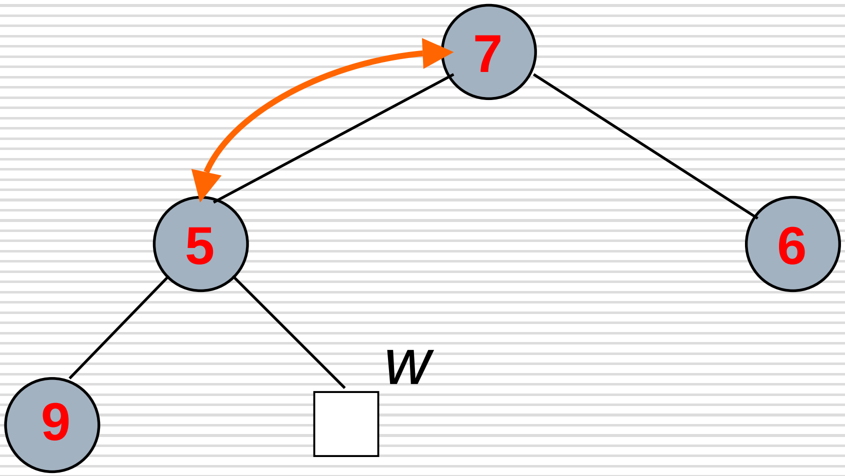
- deleteMin corresponds to removal of root key from the heap
- deleteMin algorithm has 3 steps:
  - Replace root key with the key of the last node *w*
  - Remove *w*
  - Restore the heap-order property (discussed next)

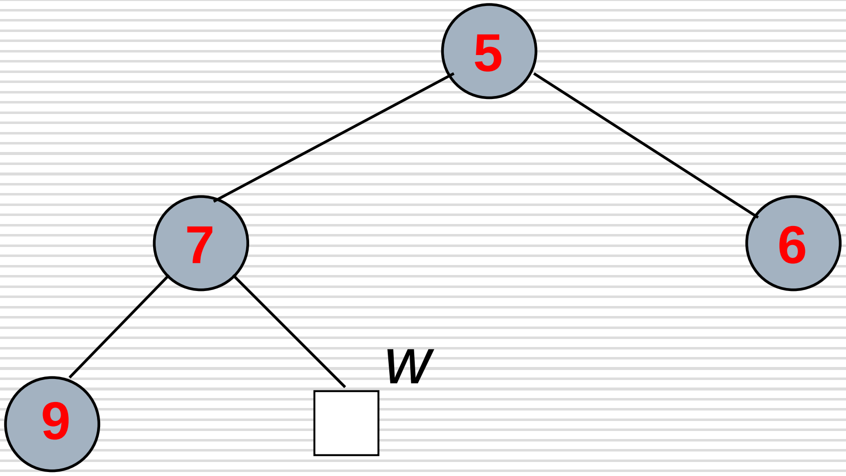


Restoring heap property after deleteMin

- After replacing the root key with the key *k* of the last node, the heap-order property may be violated
- "Percolate down": Restore heap-property by swapping key *k* along a downward path from the root, swapping it with smaller child
- Terminate when key *k* reaches a leaf or a node whose children have keys $\geq k$
- Since heap has height $O(\log n)$, restoring heap-order can be done in $O(\log n)$ time

Example:
Restoring Heap Property after deleteMin





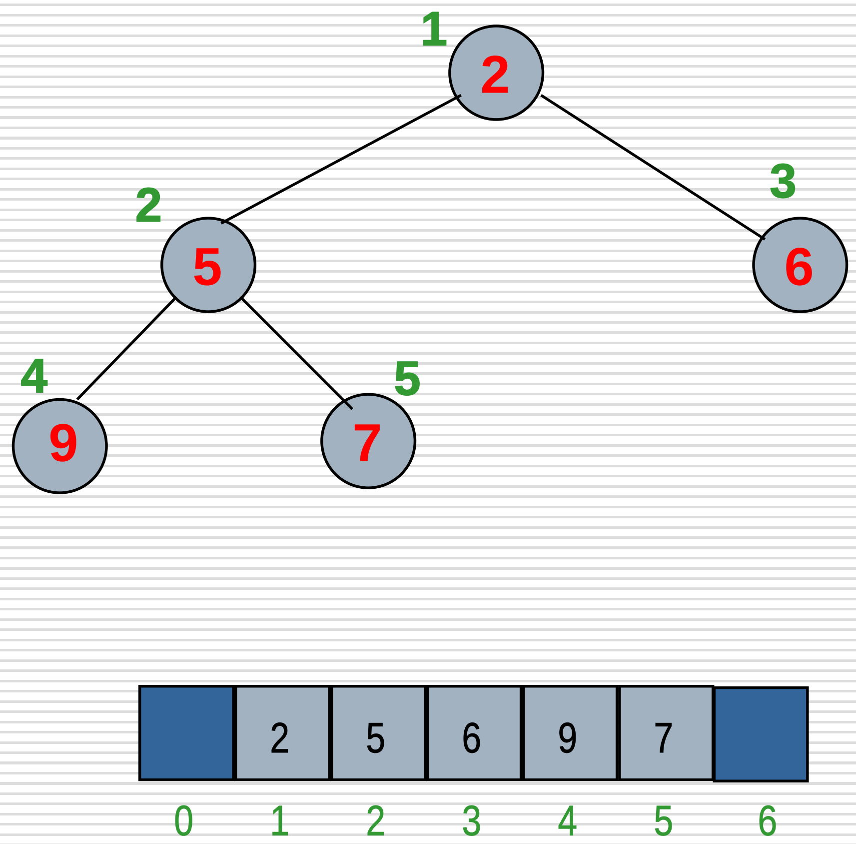
Pseudocode for percolateDown

```
/* Given a node i in the heap with children l and r.
   Each sub-tree rooted at l and r is assumed to be a heap. The
    sub-tree rooted at i may violate the heap property
    [ key(i) > key(l)    OR   key(i) > key(r) ]
    Thus Heapify lets the value of the parent node "percolate"
    down so the sub- tree at i satisfies the heap property.  */

    PercolateDown(A, i)
        l ← LEFT_CHILD (i);
        r ← RIGHT_CHILD (i);

        if (l ≤ heap_size[A]) and (A[l] < A[i])
           then smallest ← l;
           else smallest ← i;

        if (r ≤ heap_size[A]) and (A[r] < A[smallest])
           then smallest ← r;

        if smallest ≠ i
           then  exchange A[i] ⇔ A[smallest]
                 percolateDown (A,smallest)
```

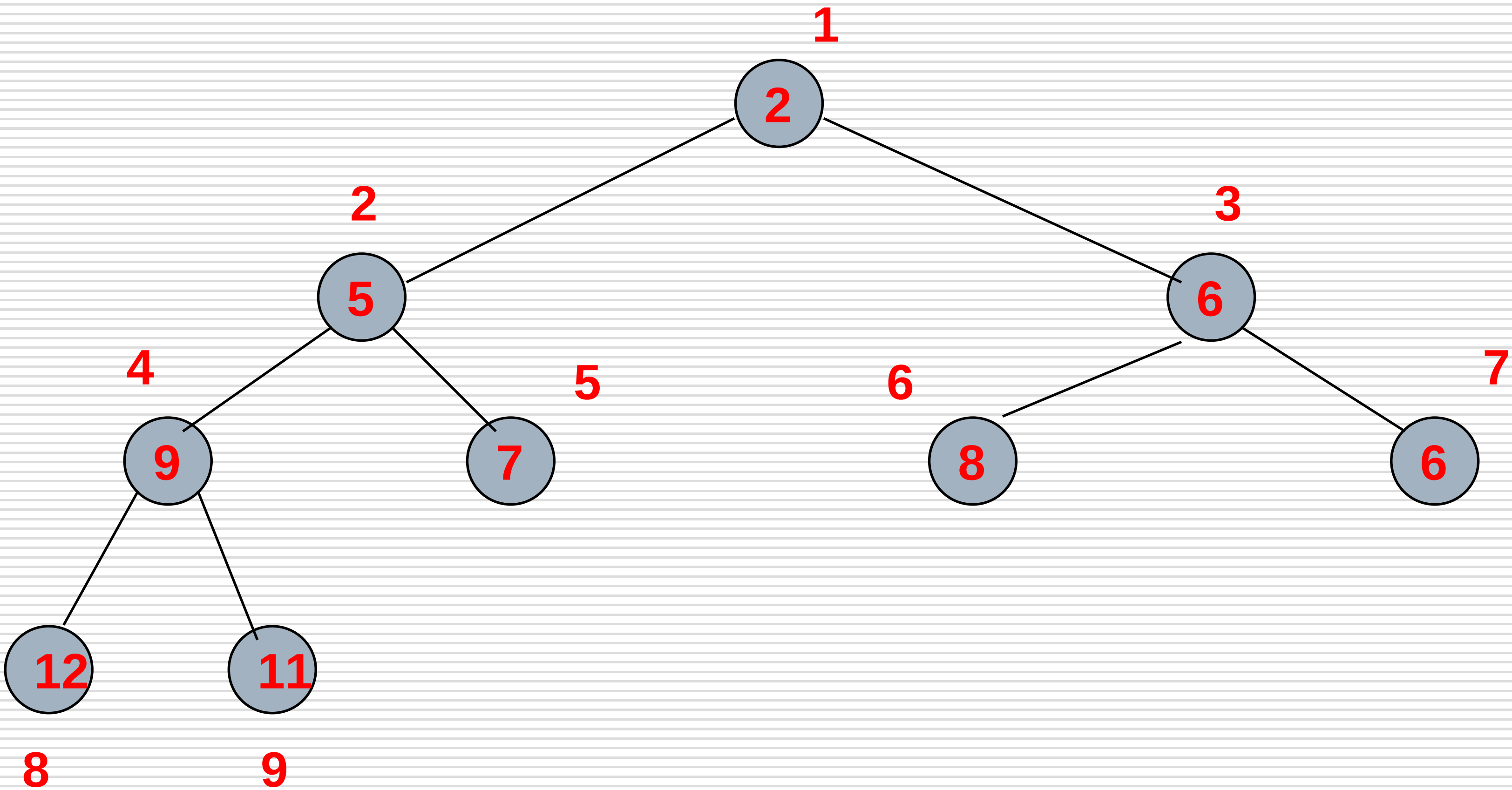
CS302

Array-Based Heap Implementation

- We can represent heap with n keys by means of a vector of length $n+1$
- For the node at rank i
  - The left child is at rank $2i$
  - The right child is at rank $2i+1$
- Links between nodes are not explicitly stored
- The cell at rank 0 is not used
- Operation insert corresponds to inserting at rank $n+1$
- Operation deleteMin corresponds to removing at rank n

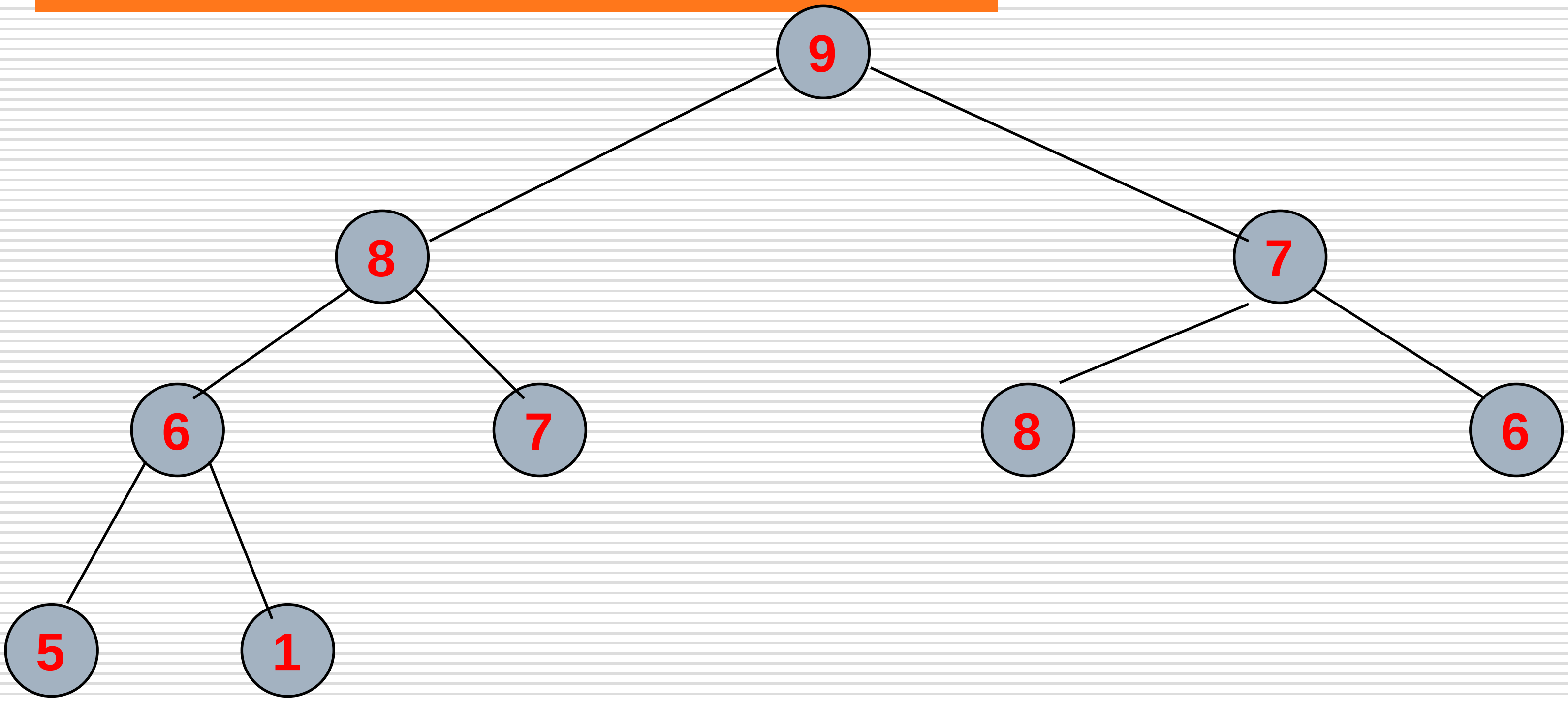


Another Example of Heap Implementation

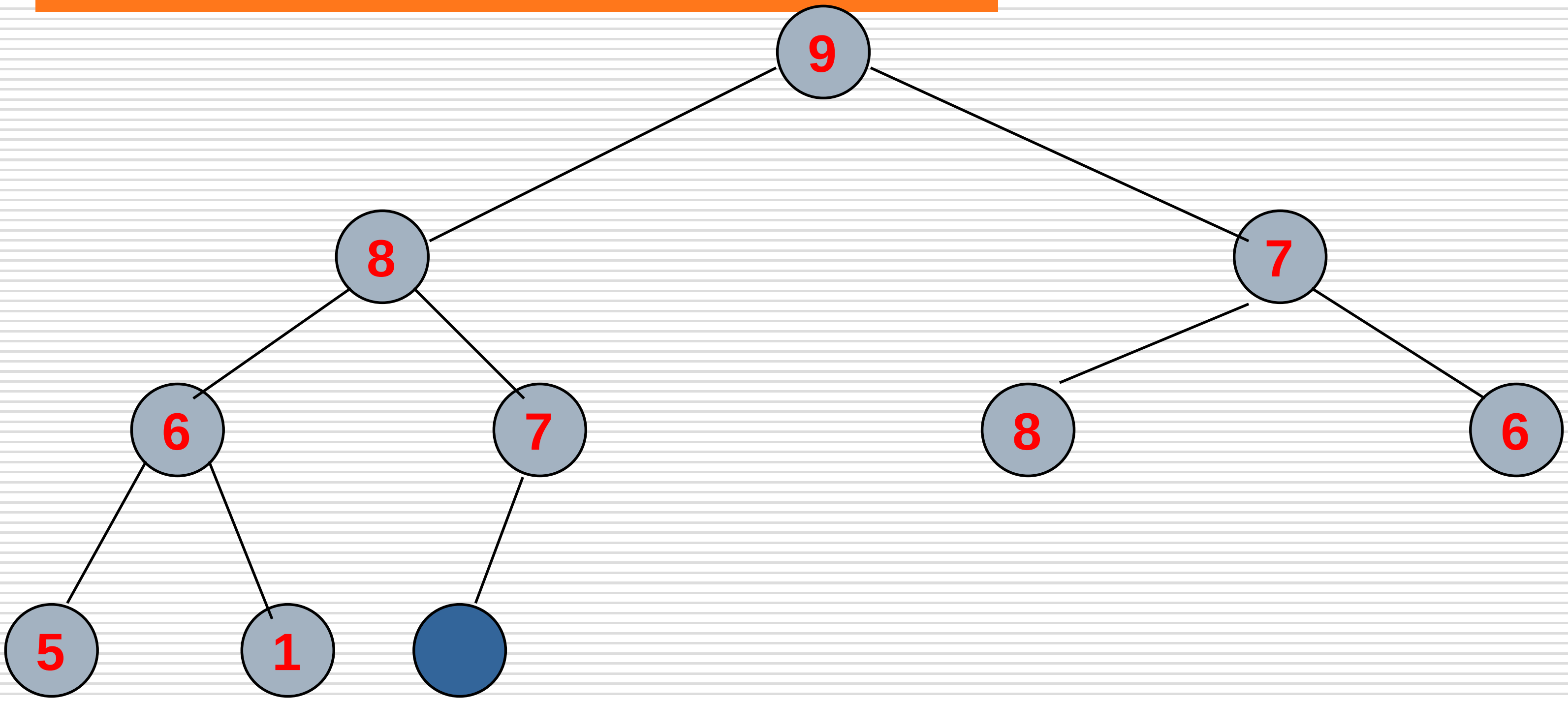


	2	5	6	9	7	2	6	12	11							
0	1	2	3	4	5	6	7	8	9	10						

Example of Max Heap

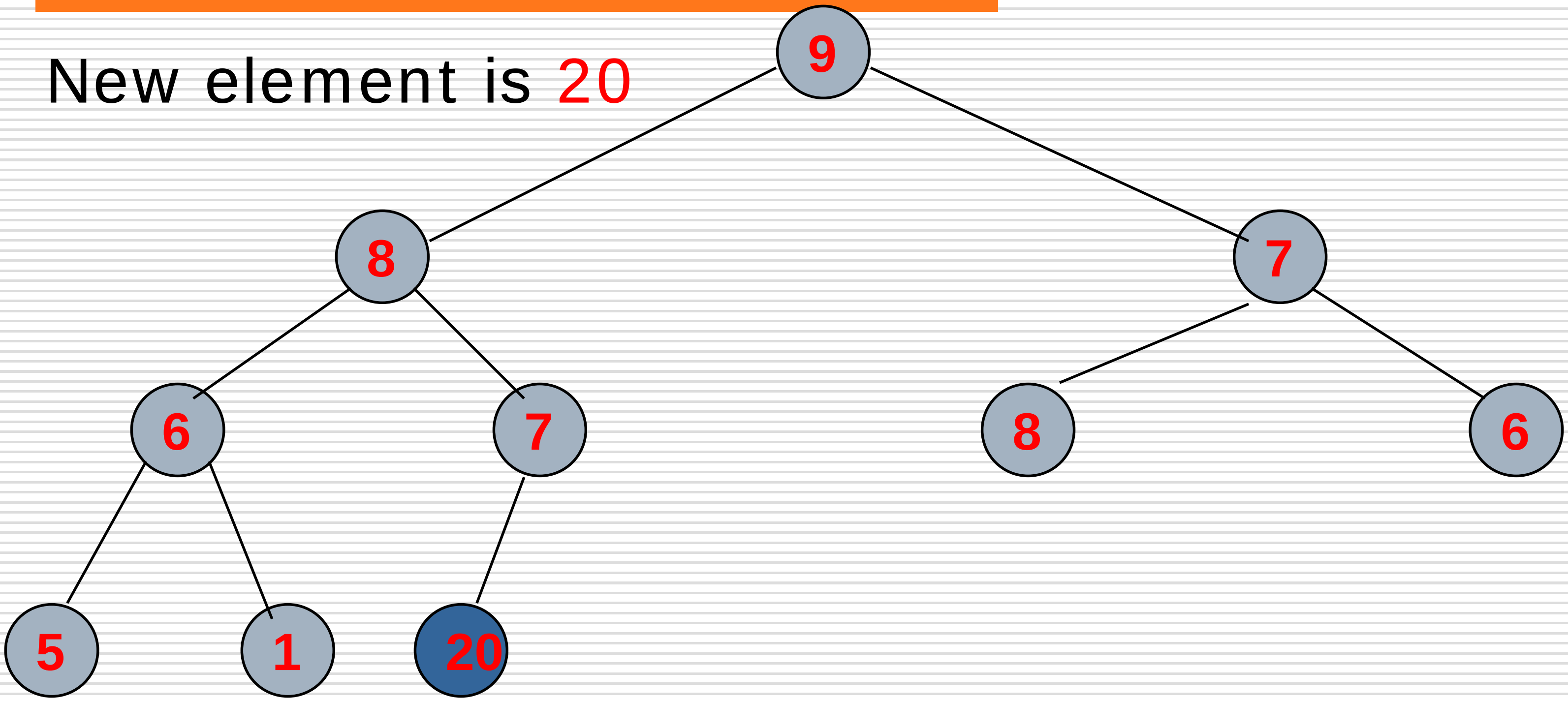


Inserting An Element Into A Max Heap



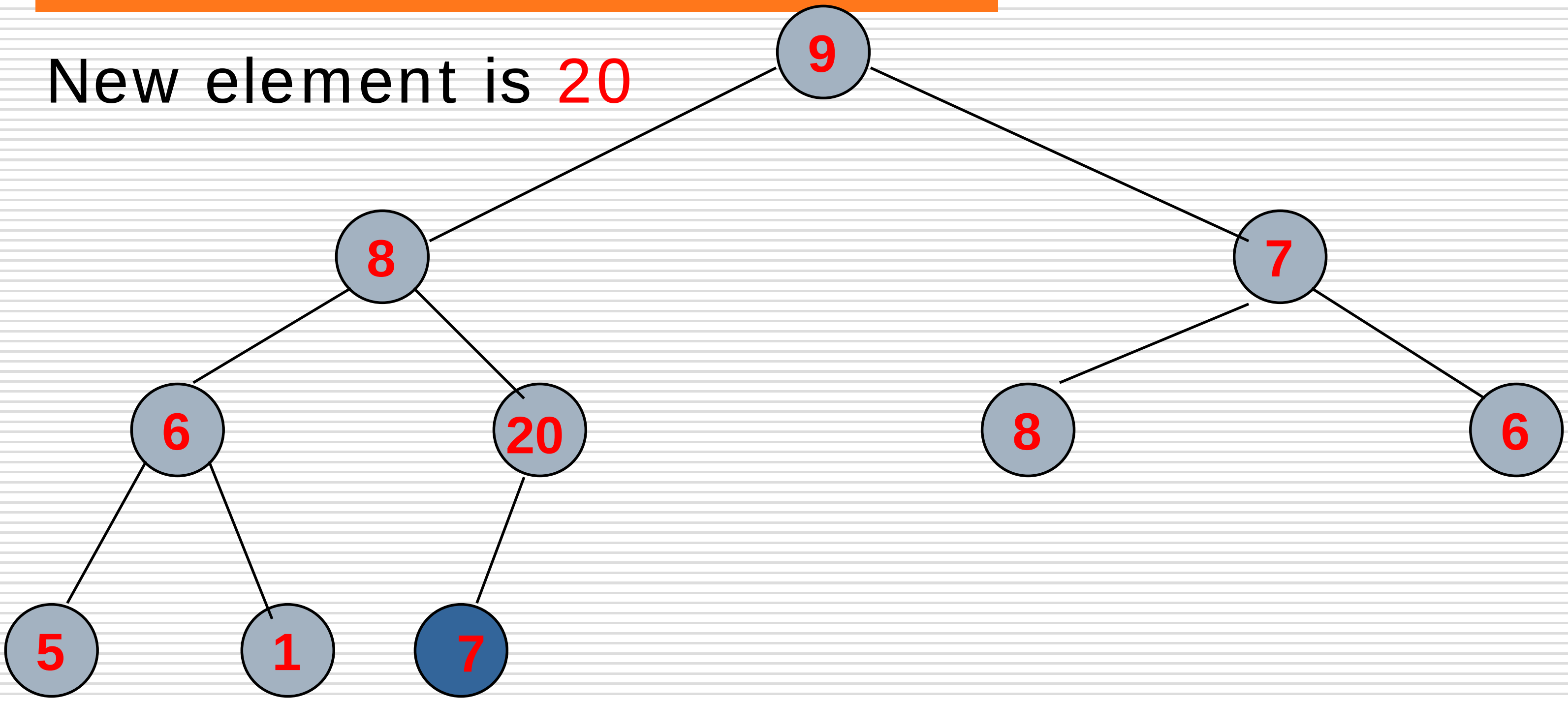
Inserting An Element Into A Max Heap

- New element is 20



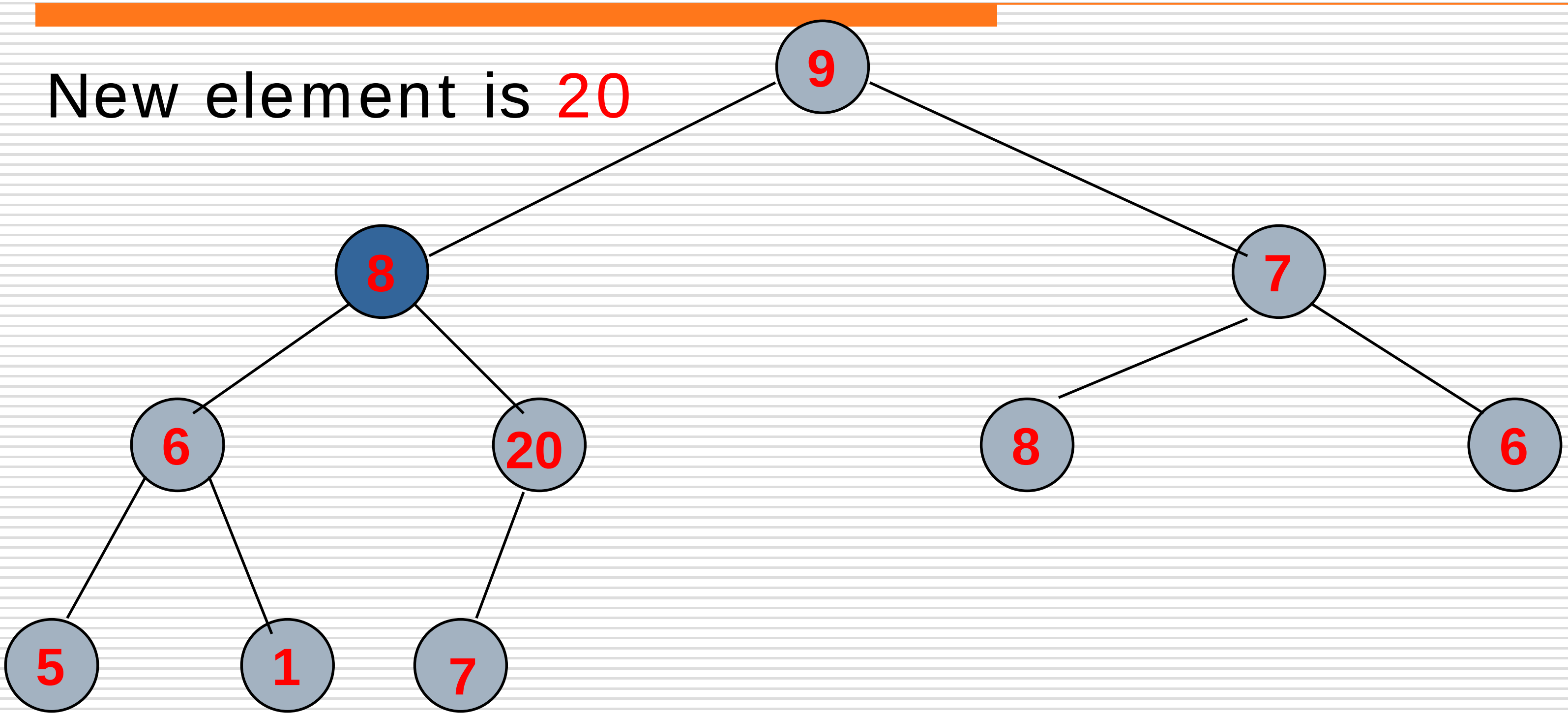
Inserting An Element Into A Max Heap

- New element is 20



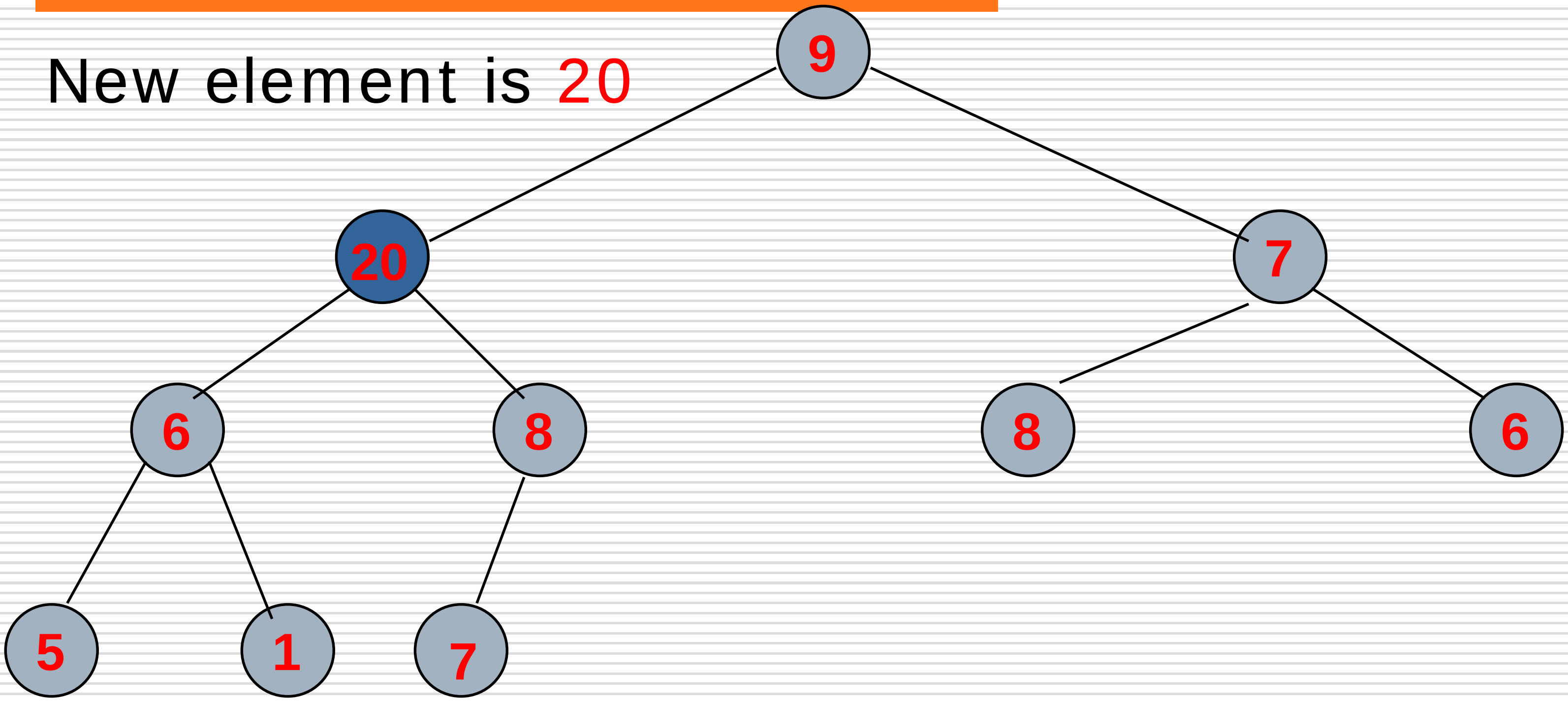
Inserting An Element Into A Max Heap

- New element is 20



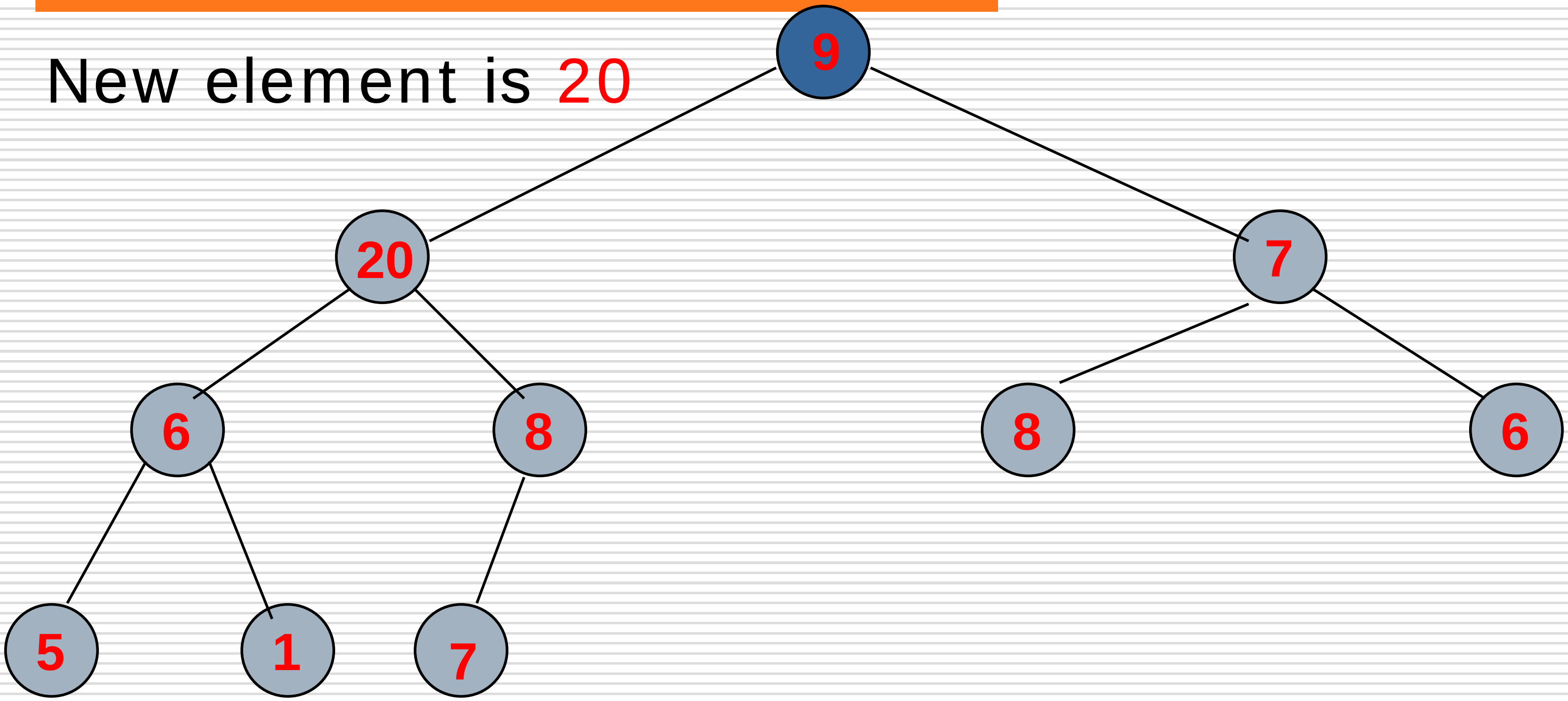
Inserting An Element Into A Max Heap

- New element is 20



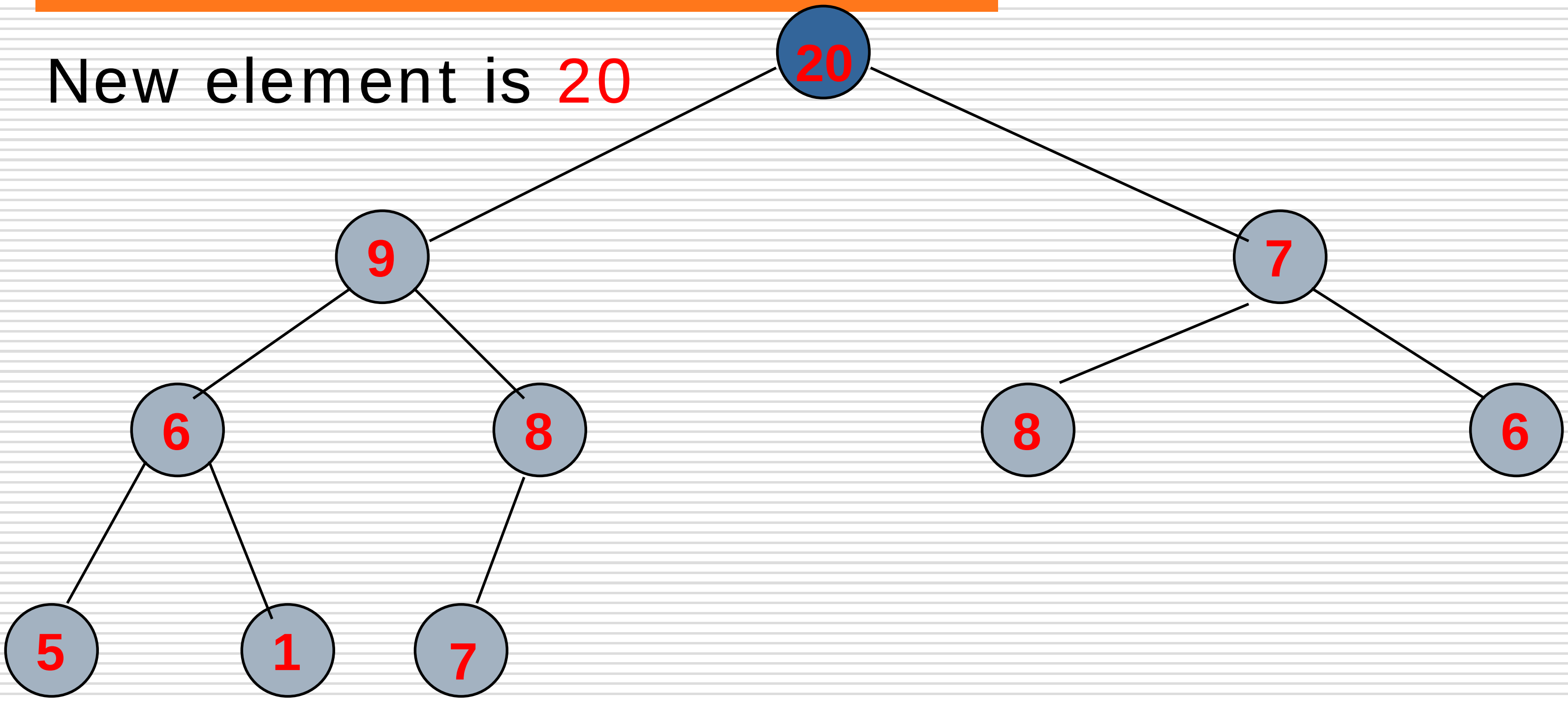
Inserting An Element Into A Max Heap

- New element is 20



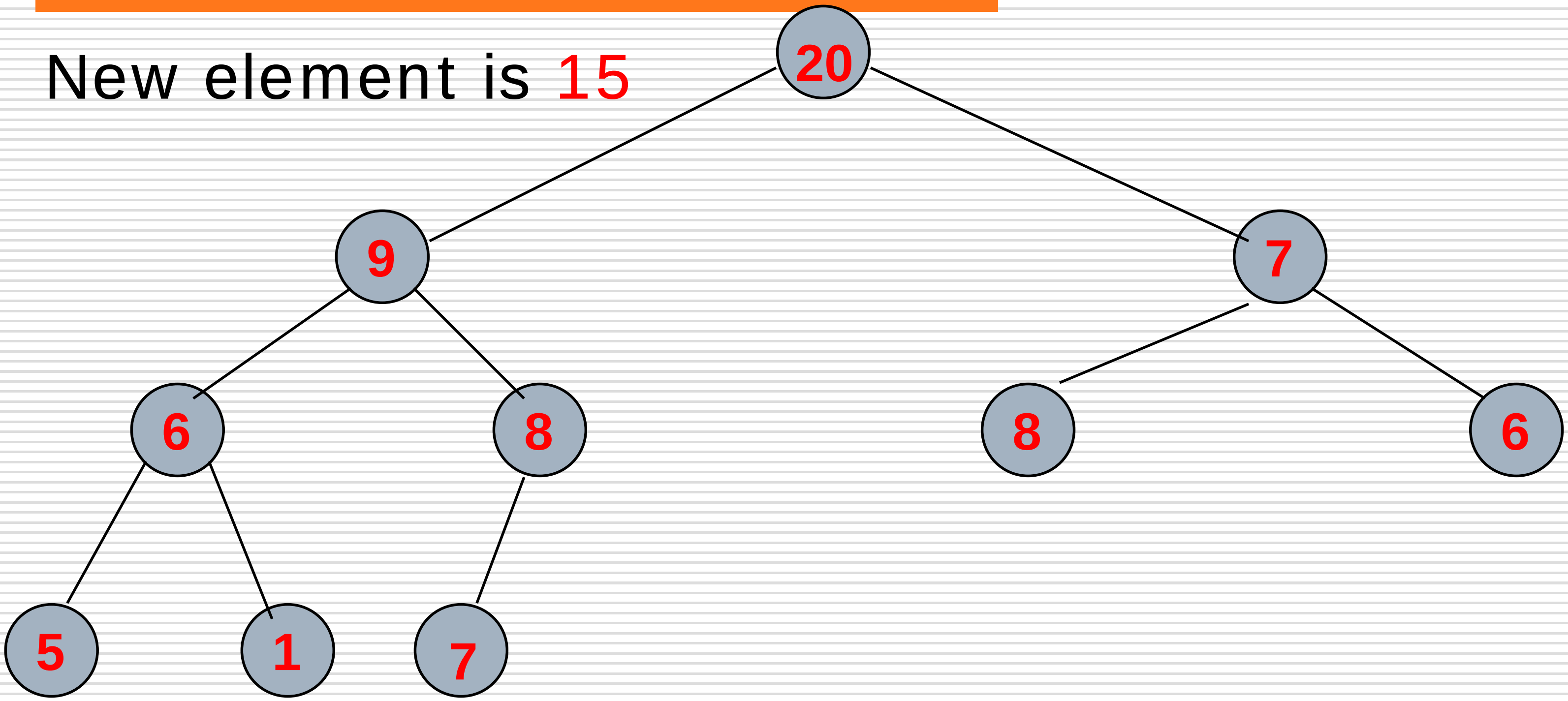
Inserting An Element Into A Max Heap

- New element is 20



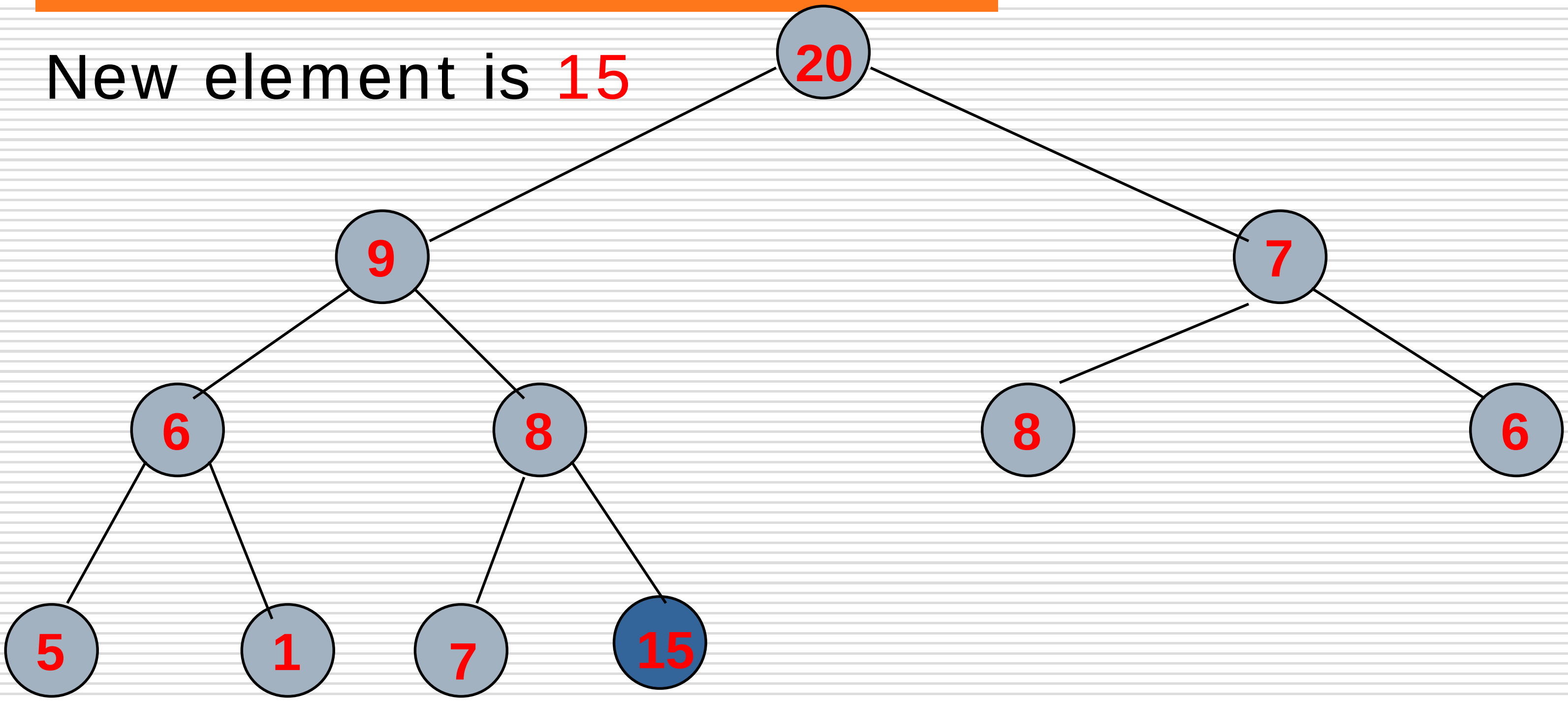
Inserting An Element Into A Max Heap

- New element is 15



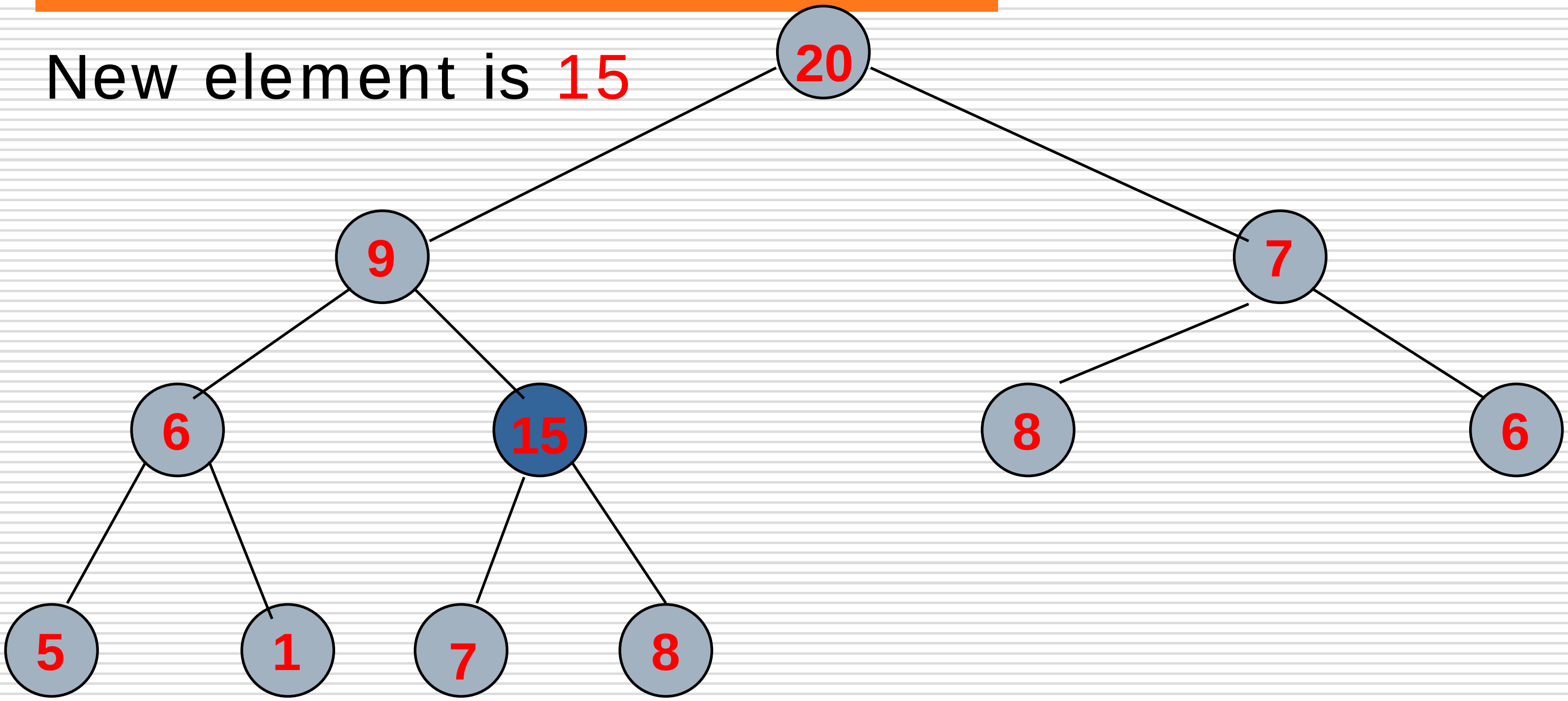
Inserting An Element Into A Max Heap

- New element is 15



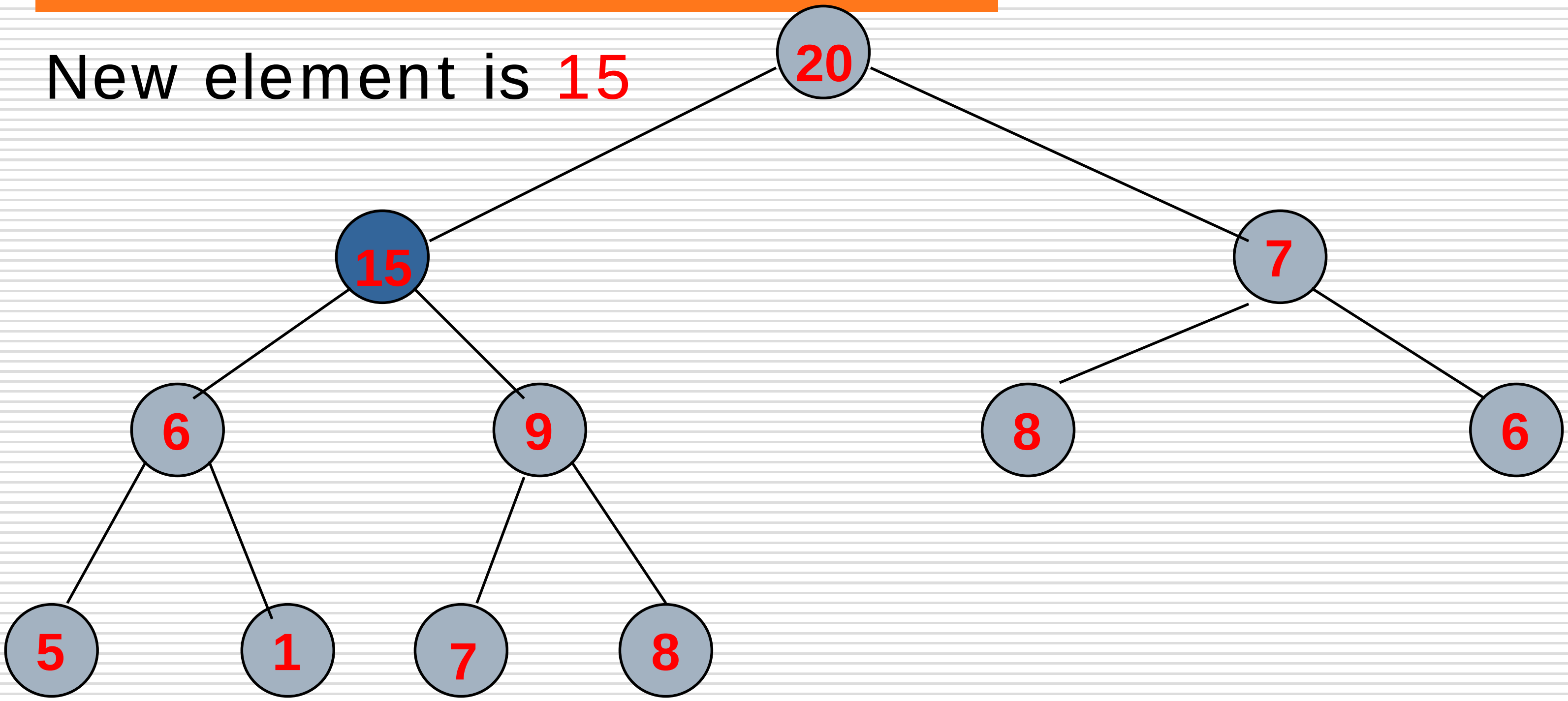
Inserting An Element Into A Max Heap

- New element is 15



Inserting An Element Into A Max Heap

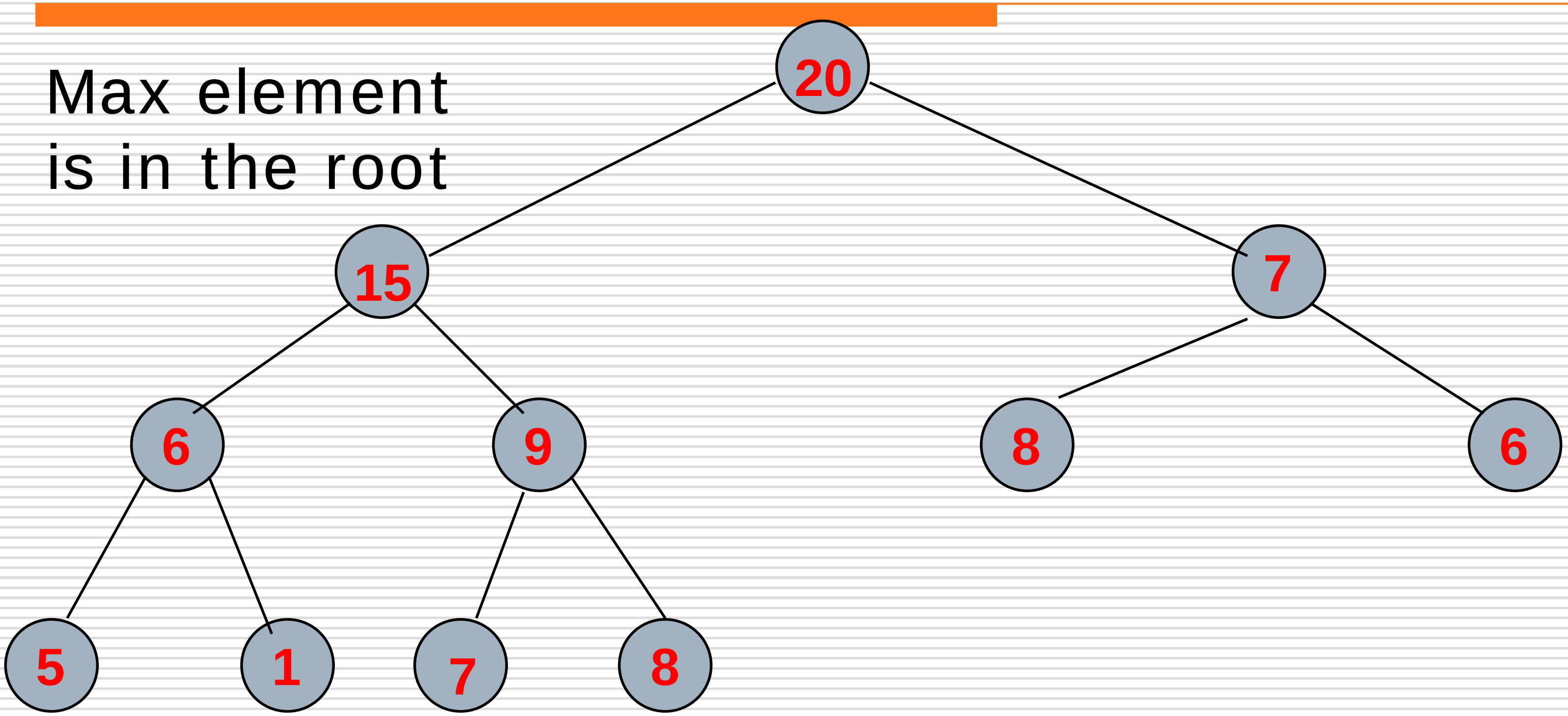
- New element is 15



Complexity is O(log n), where n is heap size

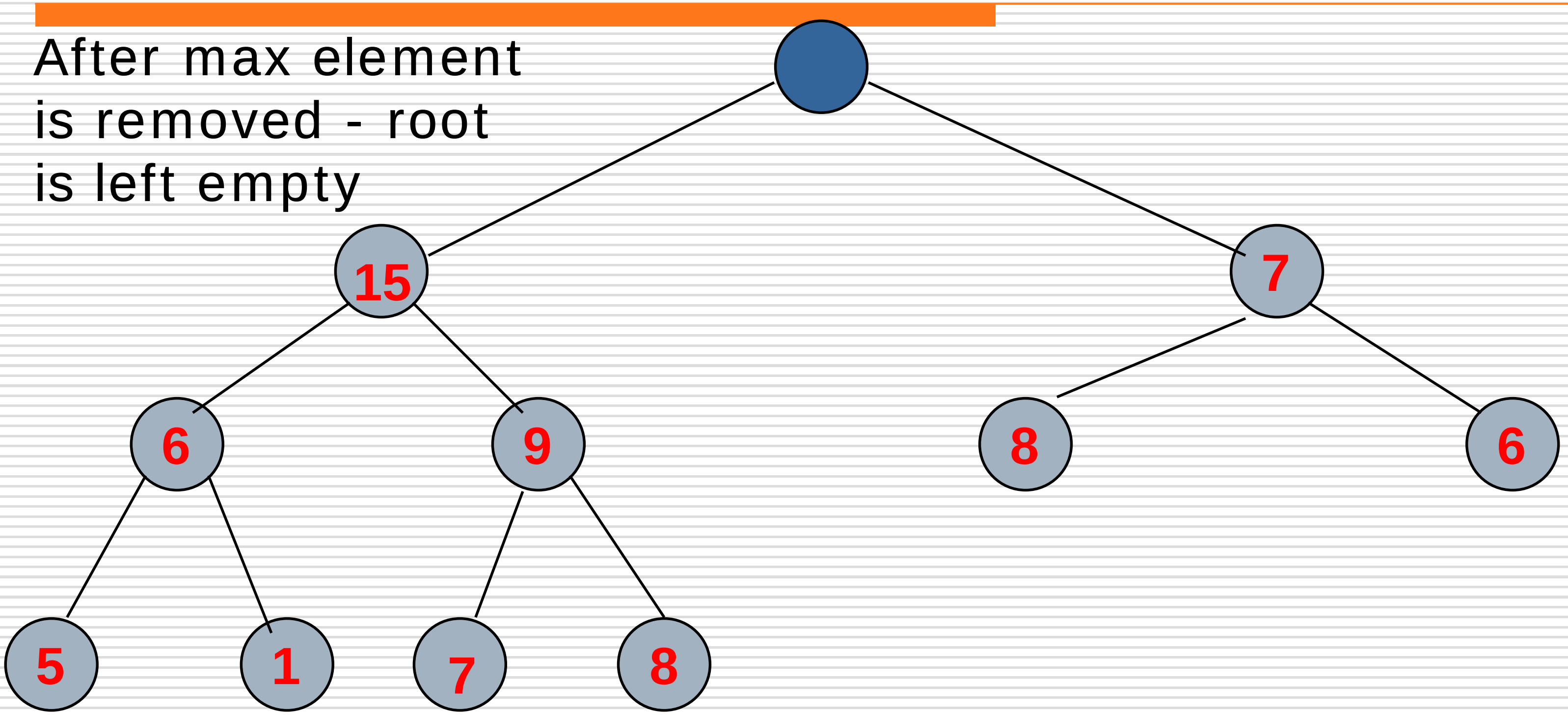
Removing The Max Element

- Max element is in the root



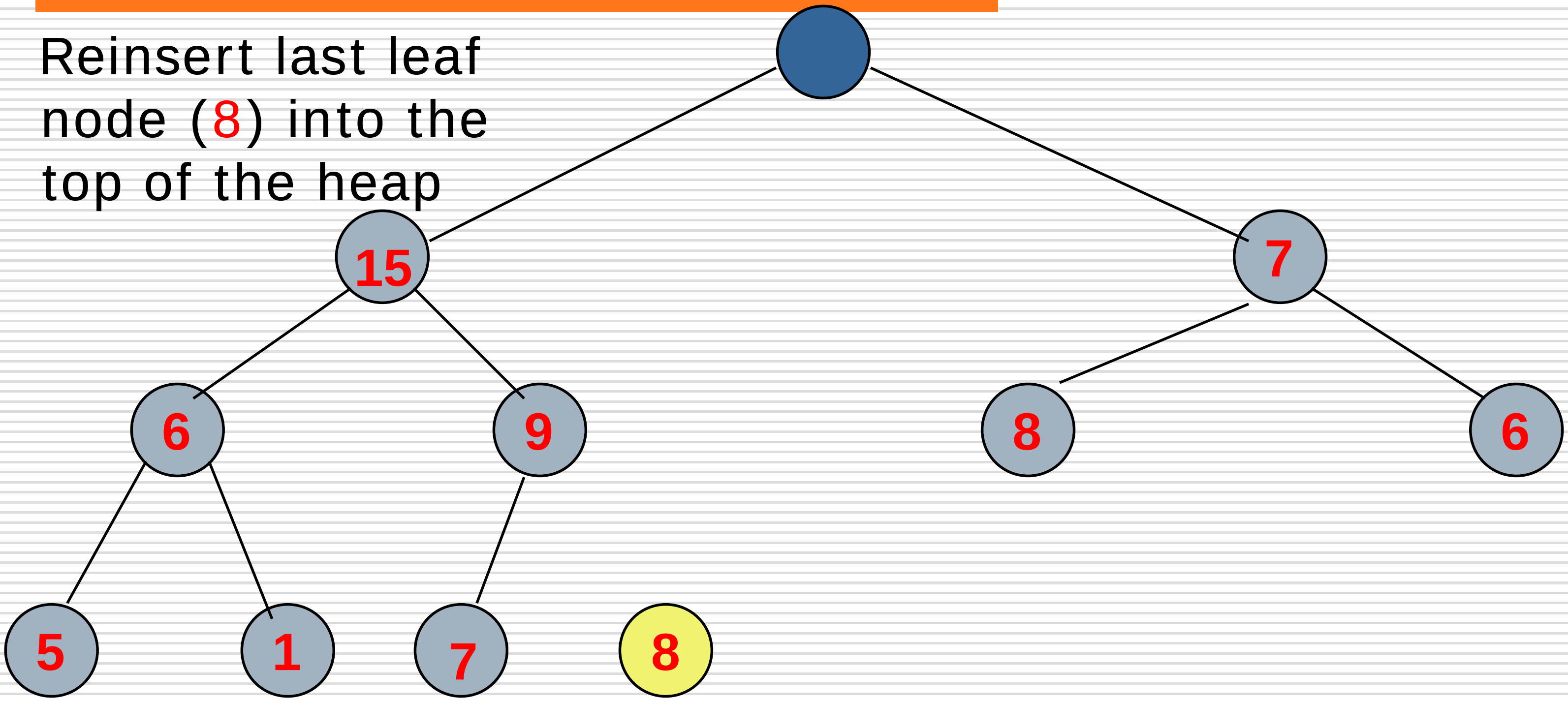
Removing The Max Element

- After max element is removed - root is left empty

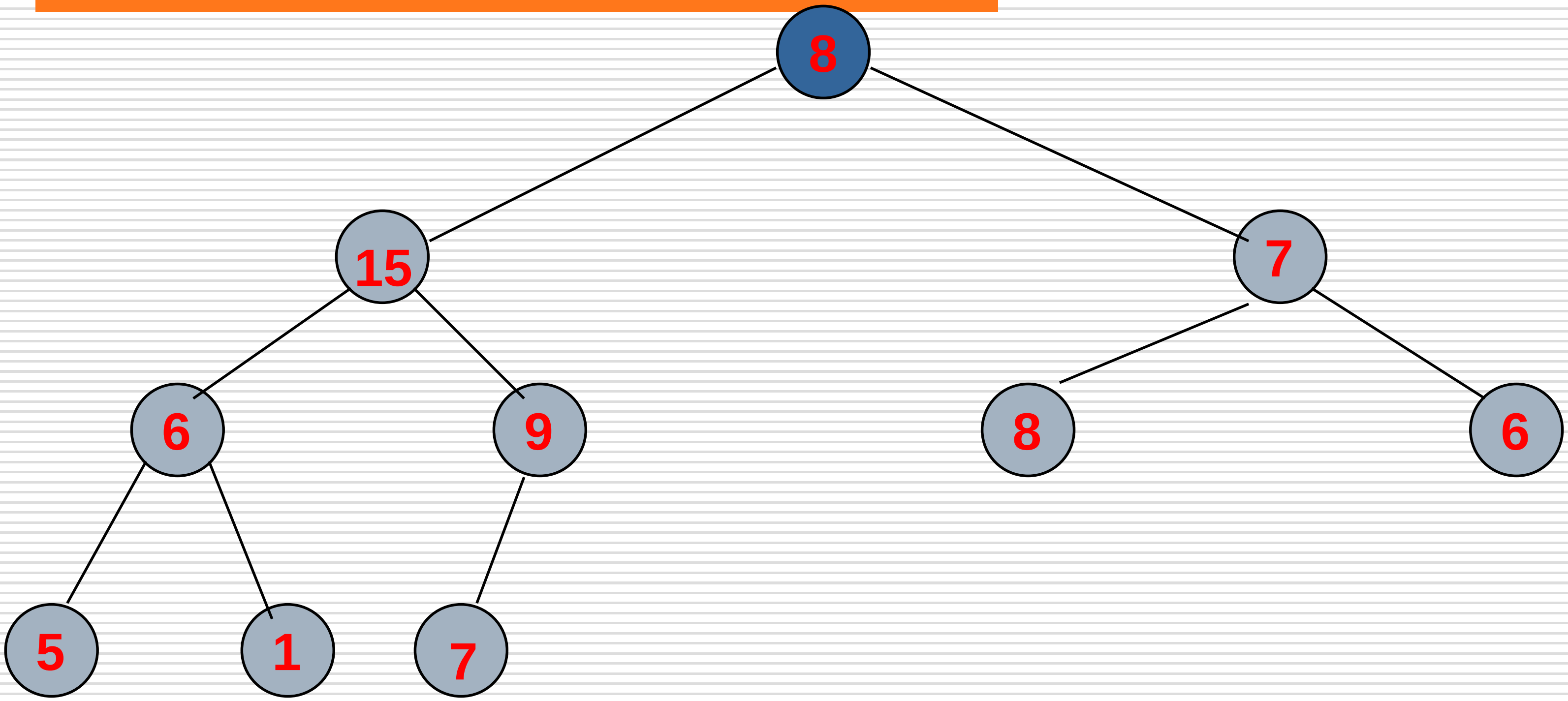


Removing The Max Element

- Reinsert last leaf node (8) into the top of the heap

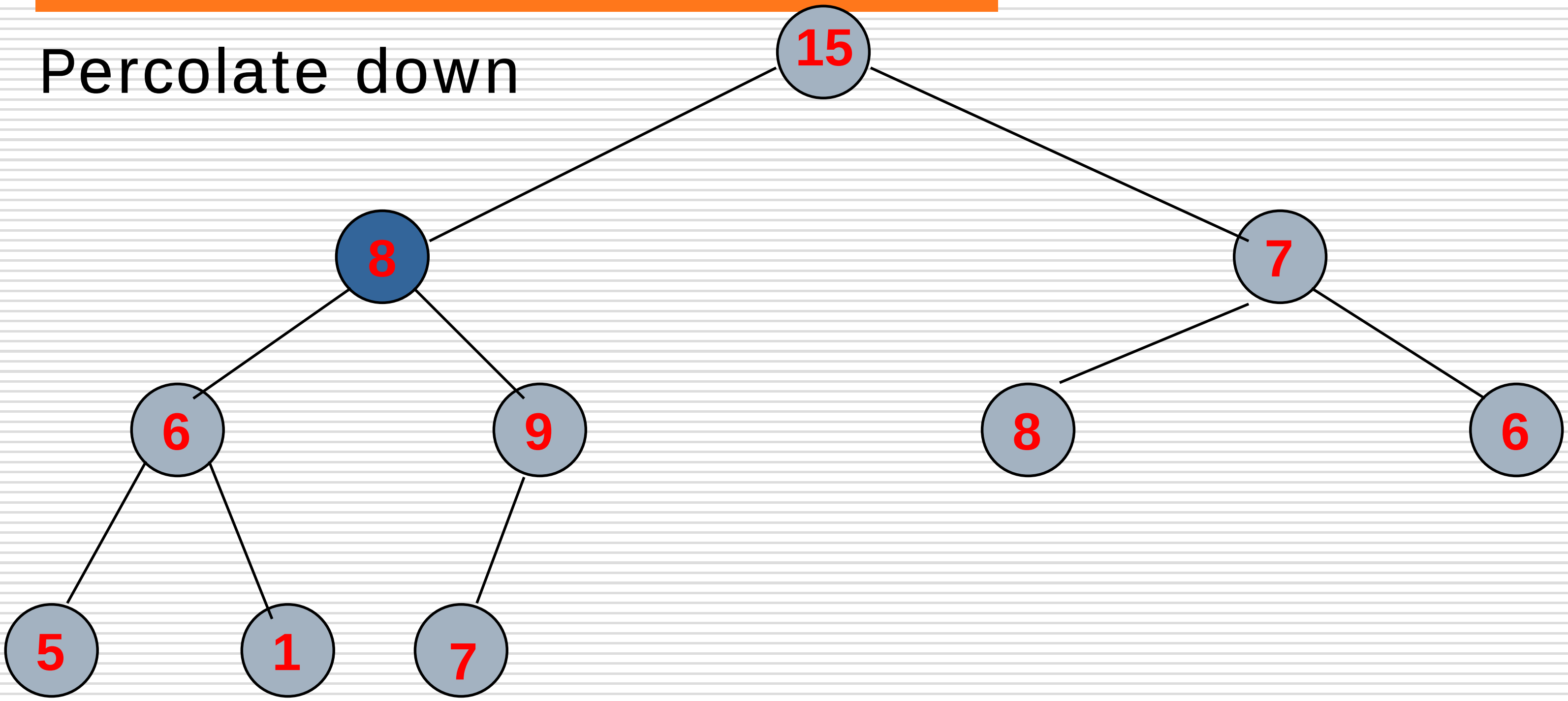


Removing The Max Element



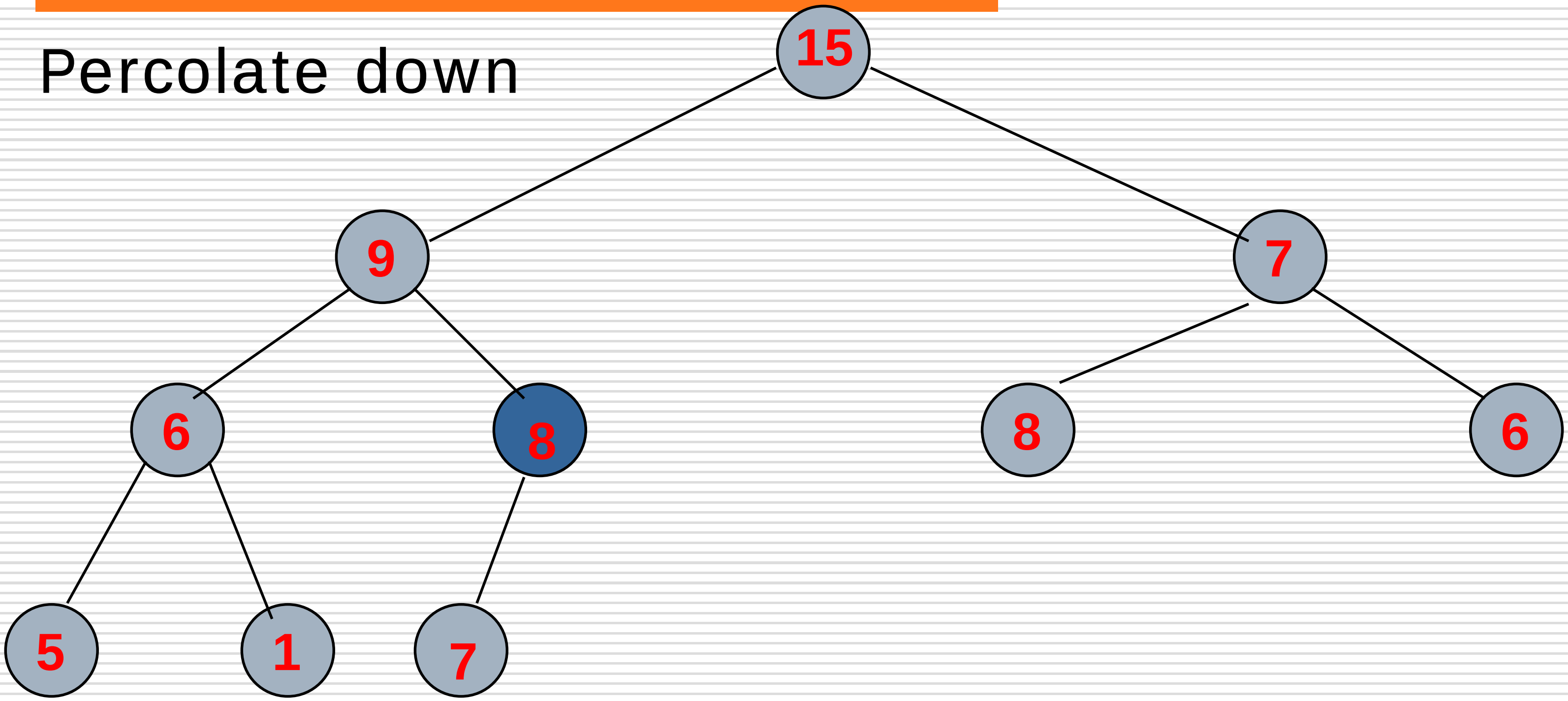
Removing The Max Element

- Percolate down



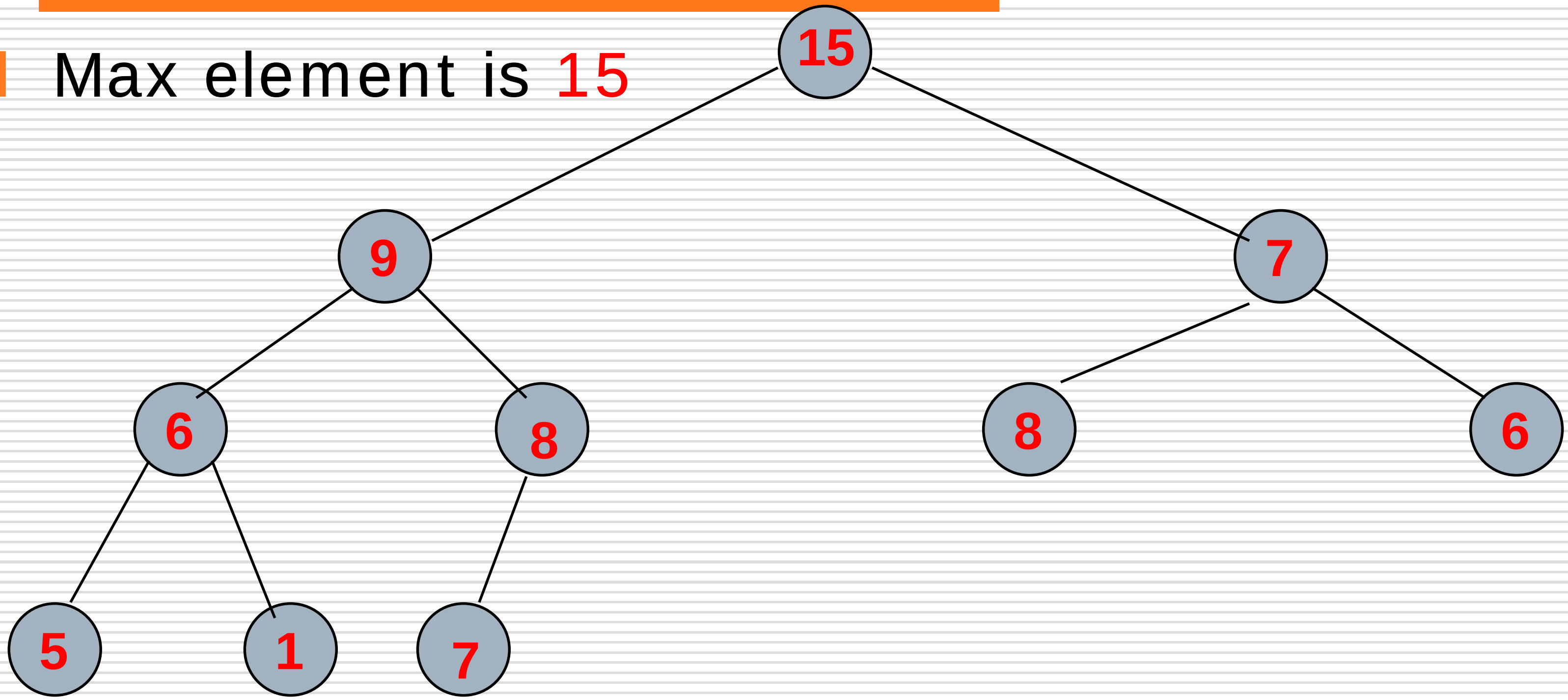
Removing The Max Element

- Percolate down



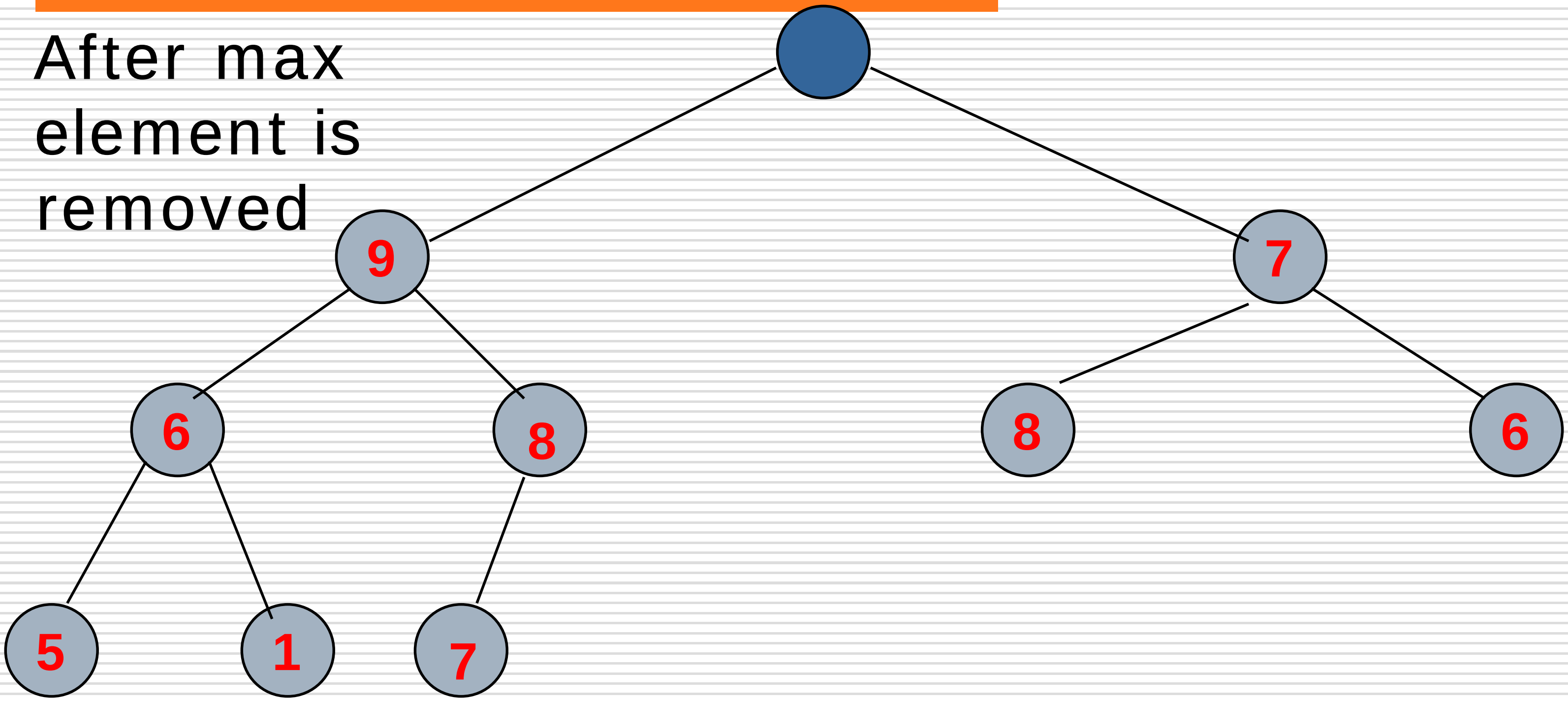
Removing The Max Element

- Max element is 15



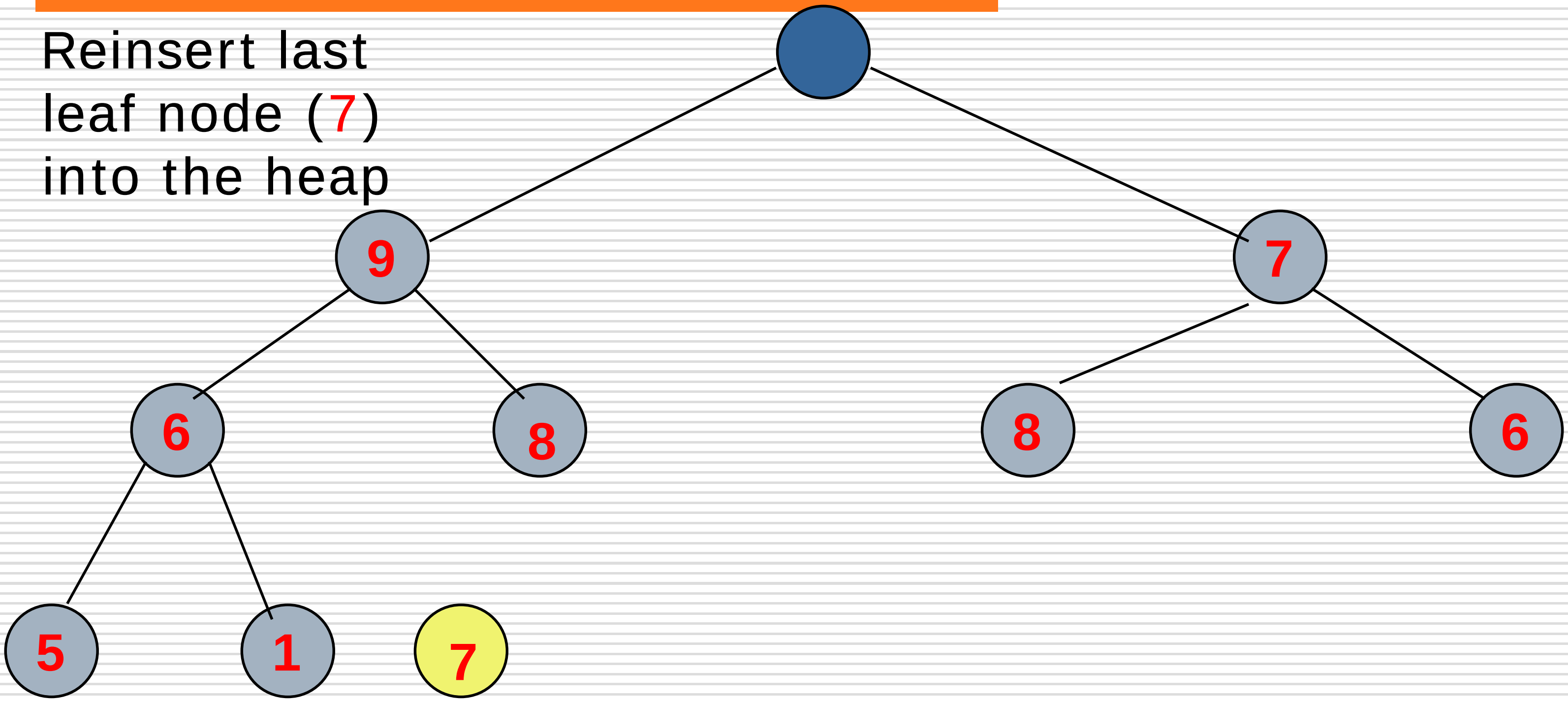
Removing The Max Element

- After max element is removed

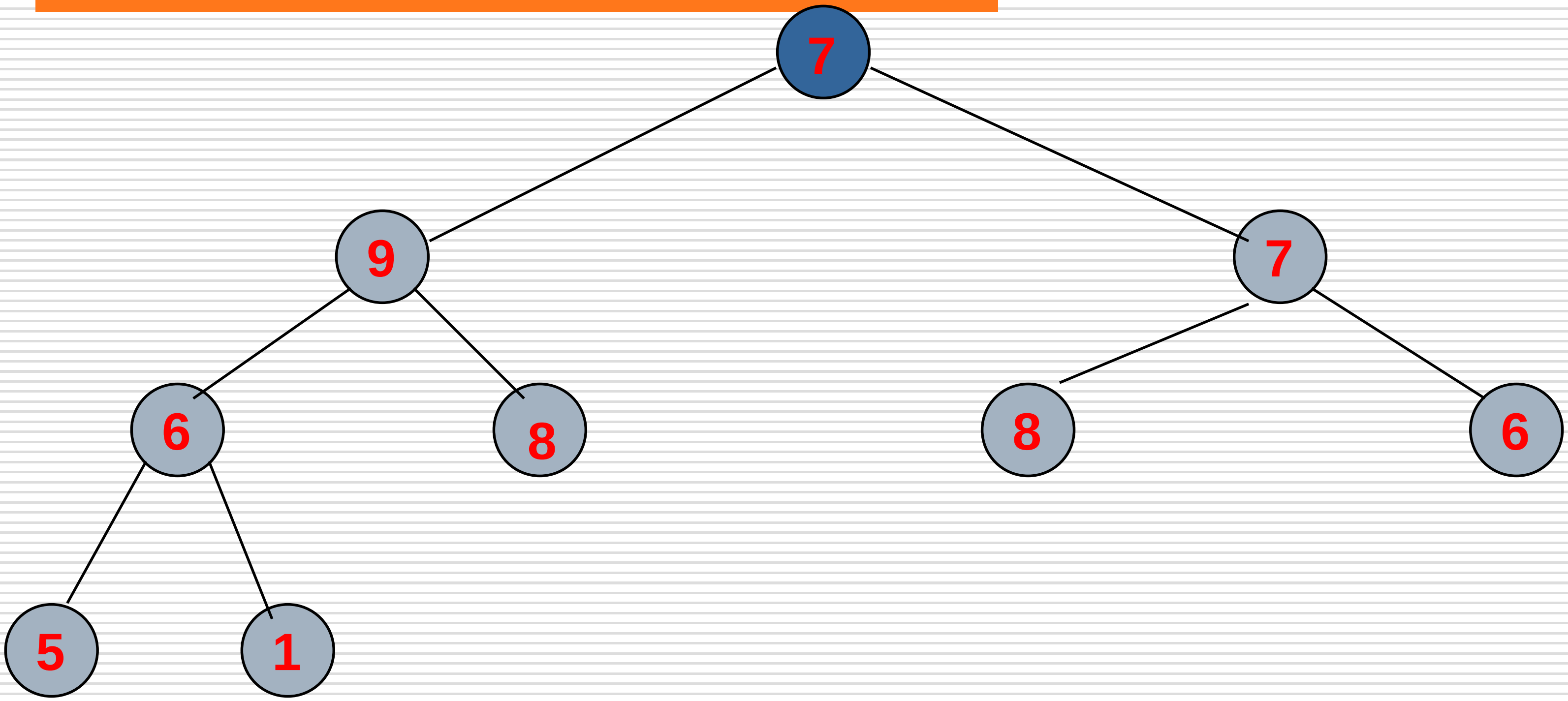


Removing The Max Element

- Reinsert last leaf node (7) into the heap

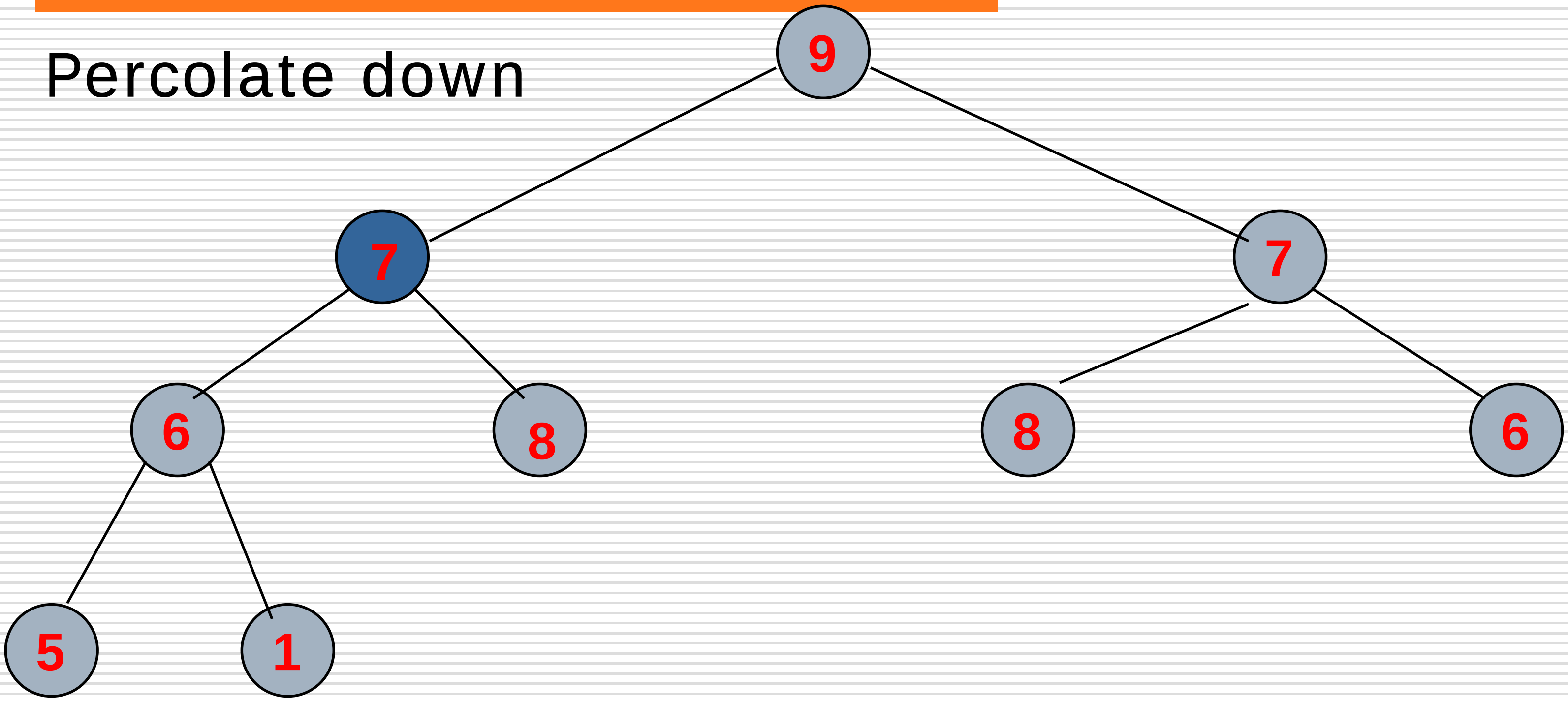


Removing The Max Element

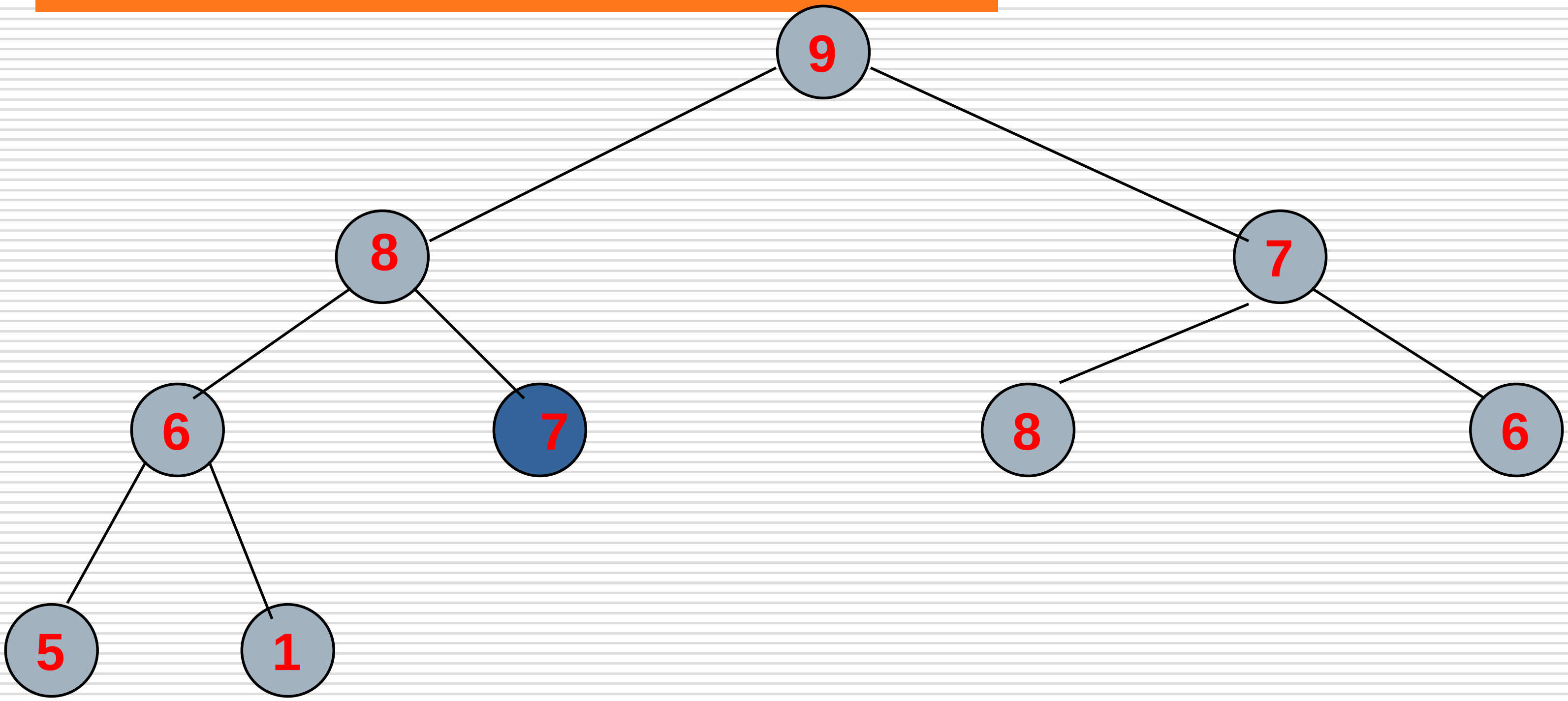


Removing The Max Element

- Percolate down



Removing The Max Element



Complexity is again O(log n)

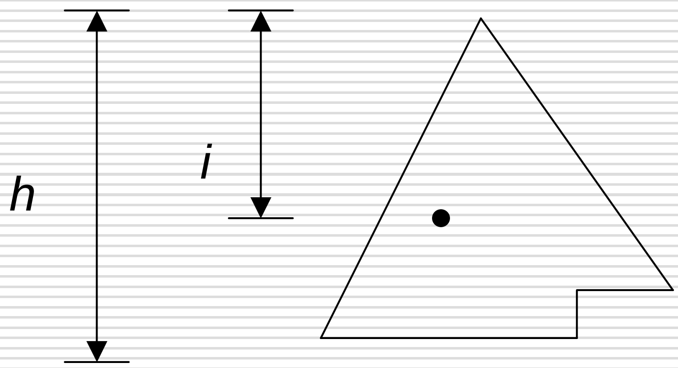
Build Heap

- To build heap originally, two alternatives:
  - One approach (not the best):
    - Repeatedly insert into initially empty heap
    - Runtime: O(n log n)
  - Better approach ("Build Heap"):
    - Start with elements in any order
    - Apply "percolate down" for nodes $n/2$ down to 1
- **buildHeap(A)**

  1. **for** $i \leftarrow \lfloor \text{length}[A]/2 \rfloor$ **downto** 1 **do**
  2. percolateDown(A, i)

Running Time of buildHeap

- We represent a heap in the following manner:



For nodes at level i , there are 2^i nodes. Work is done for h-i levels.

Total work done to build the heap is the sum of the work for these levels

Total work to Build Heap:

$$\sum_{i=1}^{h=\log n} 2^i (h-i)$$

Taking $h = \log\ n$:

$$= \sum_{i=1}^{h=\log n} 2^i (\log n - i)$$

Substituting $j = \log n - i$, we get:

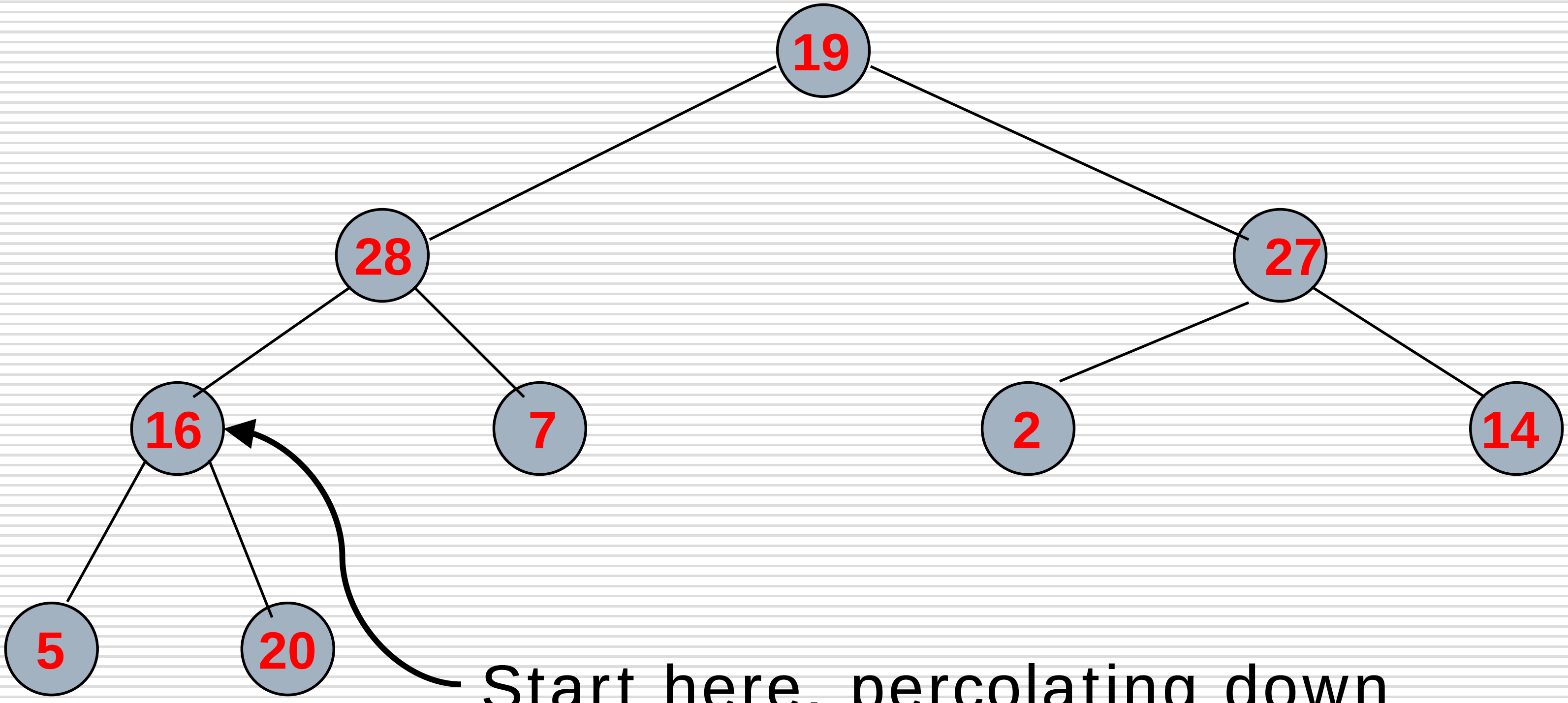
$$= \sum_{j=\log n}^{1} 2^{\log n - j} j$$

$$= \sum_{j=1}^{\log n} \frac{2^{\log n}}{2^j} j$$

$$= n \sum_{j=1}^{\log n} \frac{j}{2^j}$$

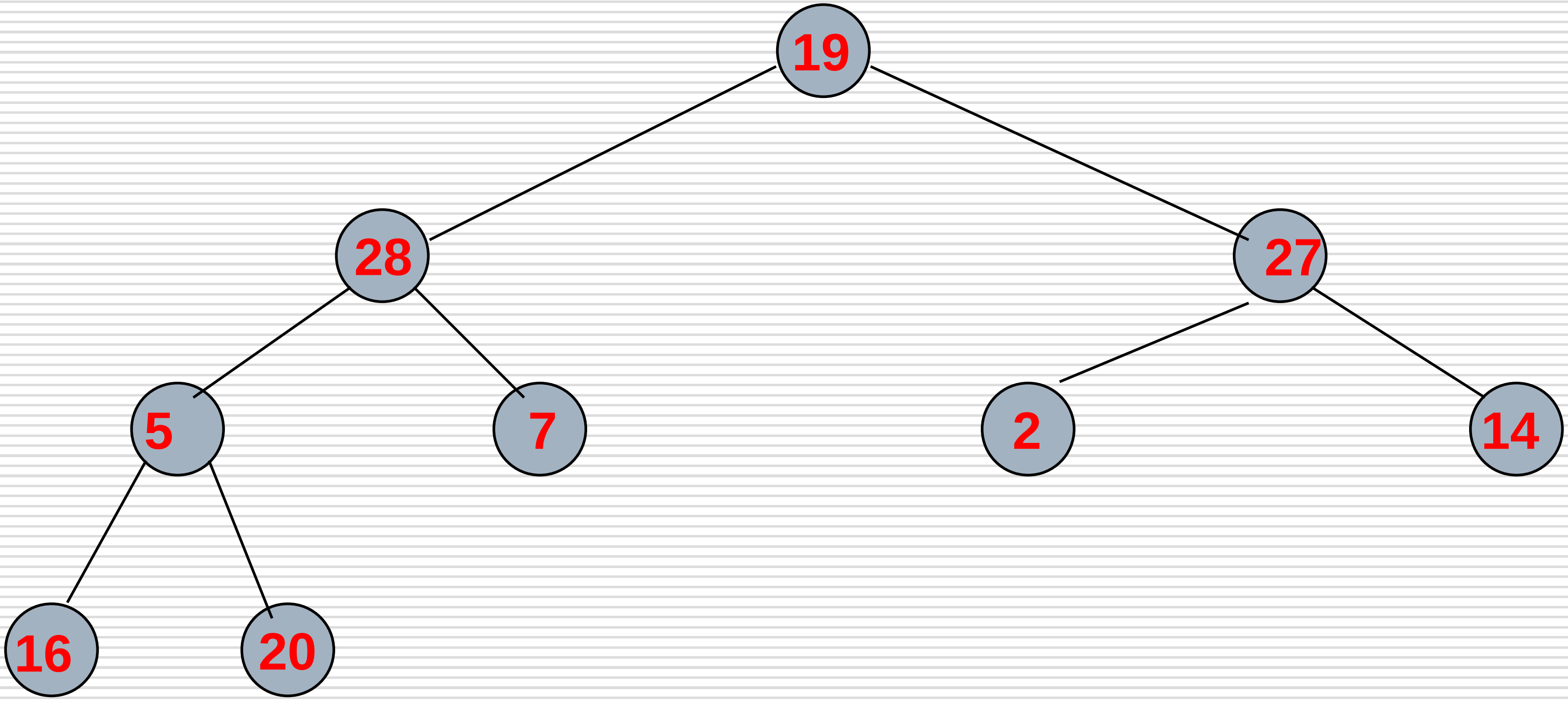
$$= O(n)$$

Build min Heap example

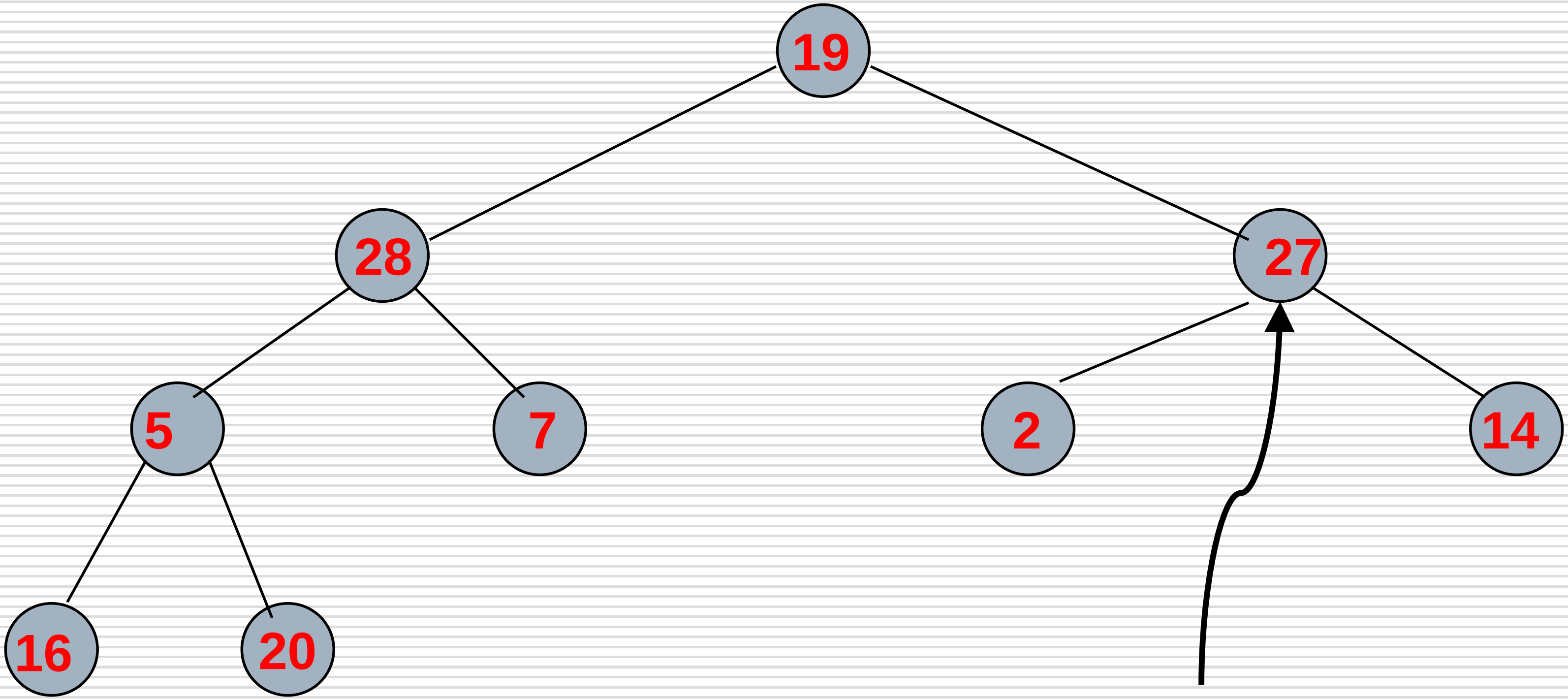


Start here, percolating down

Build min Heap example

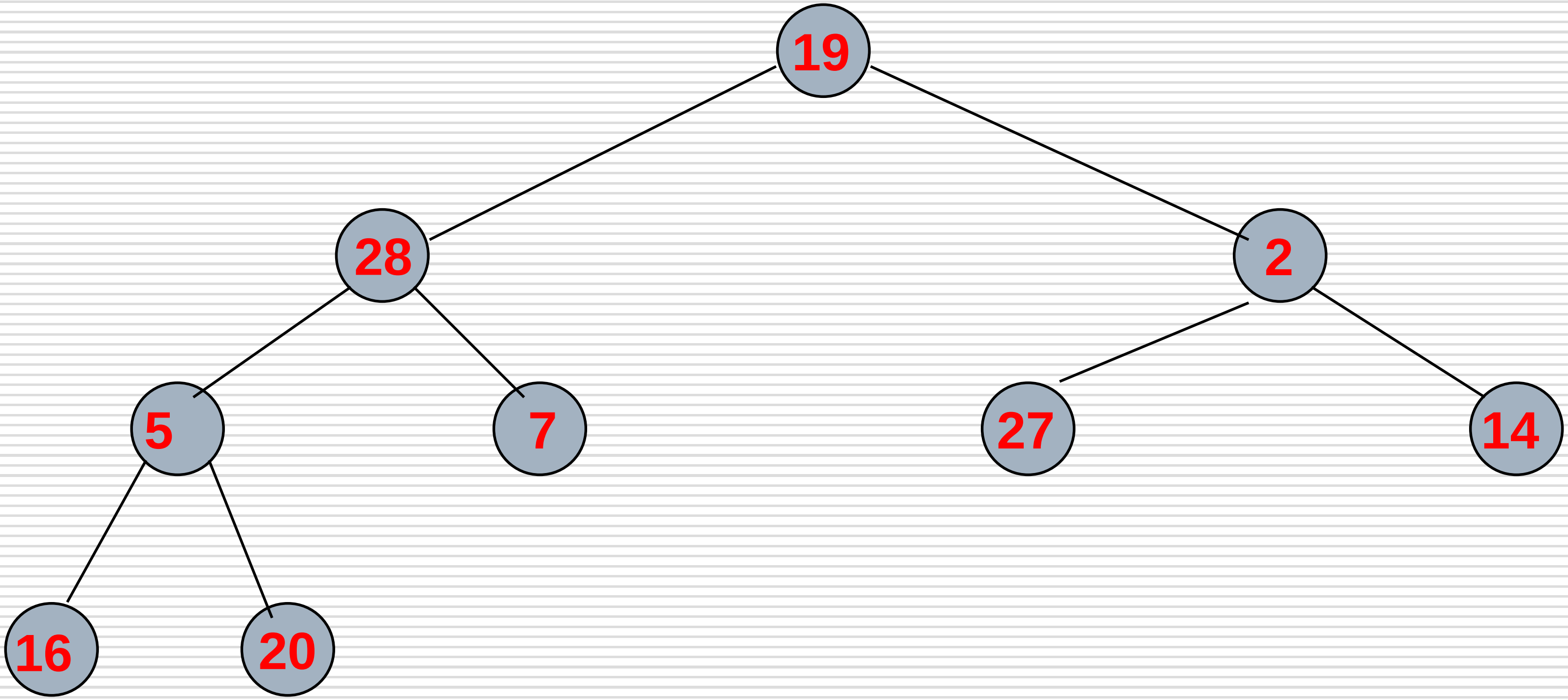


Build min Heap example

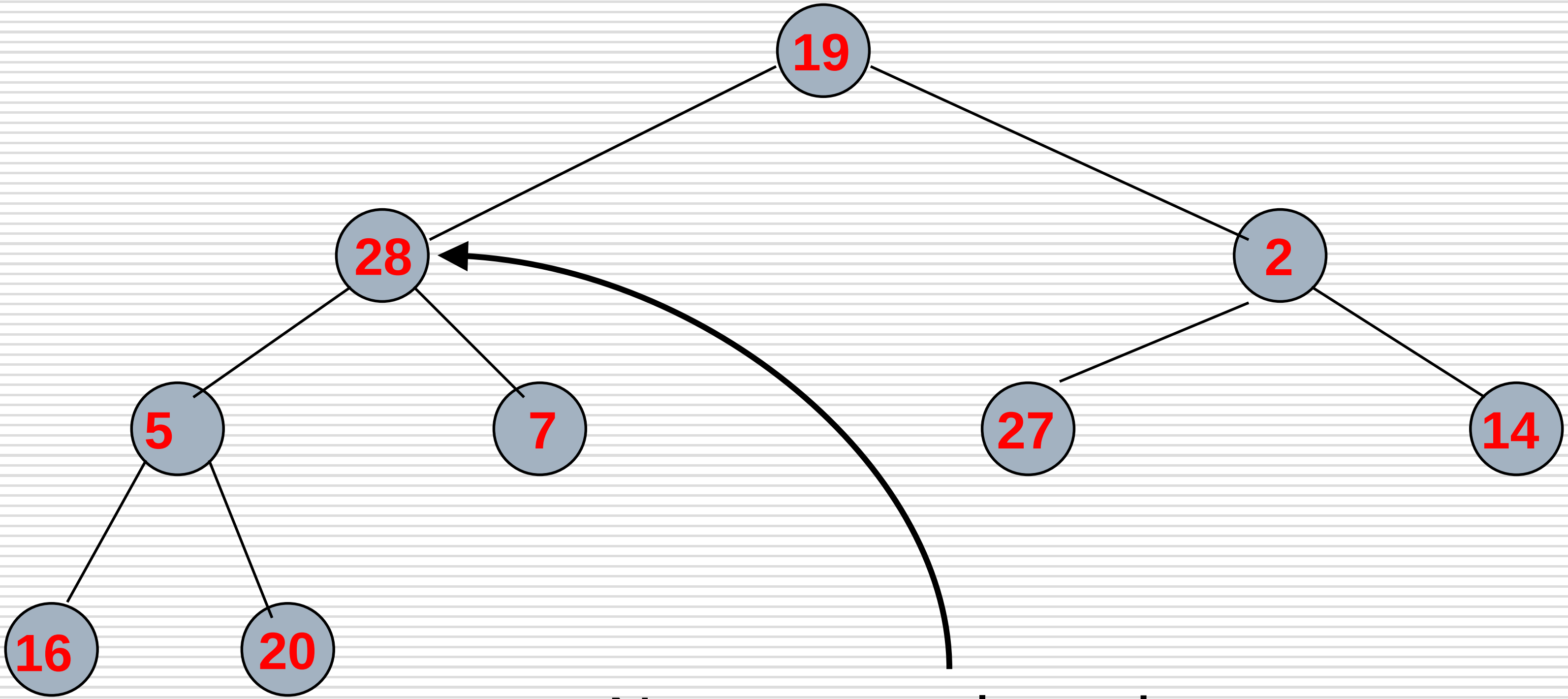


Next, percolate down

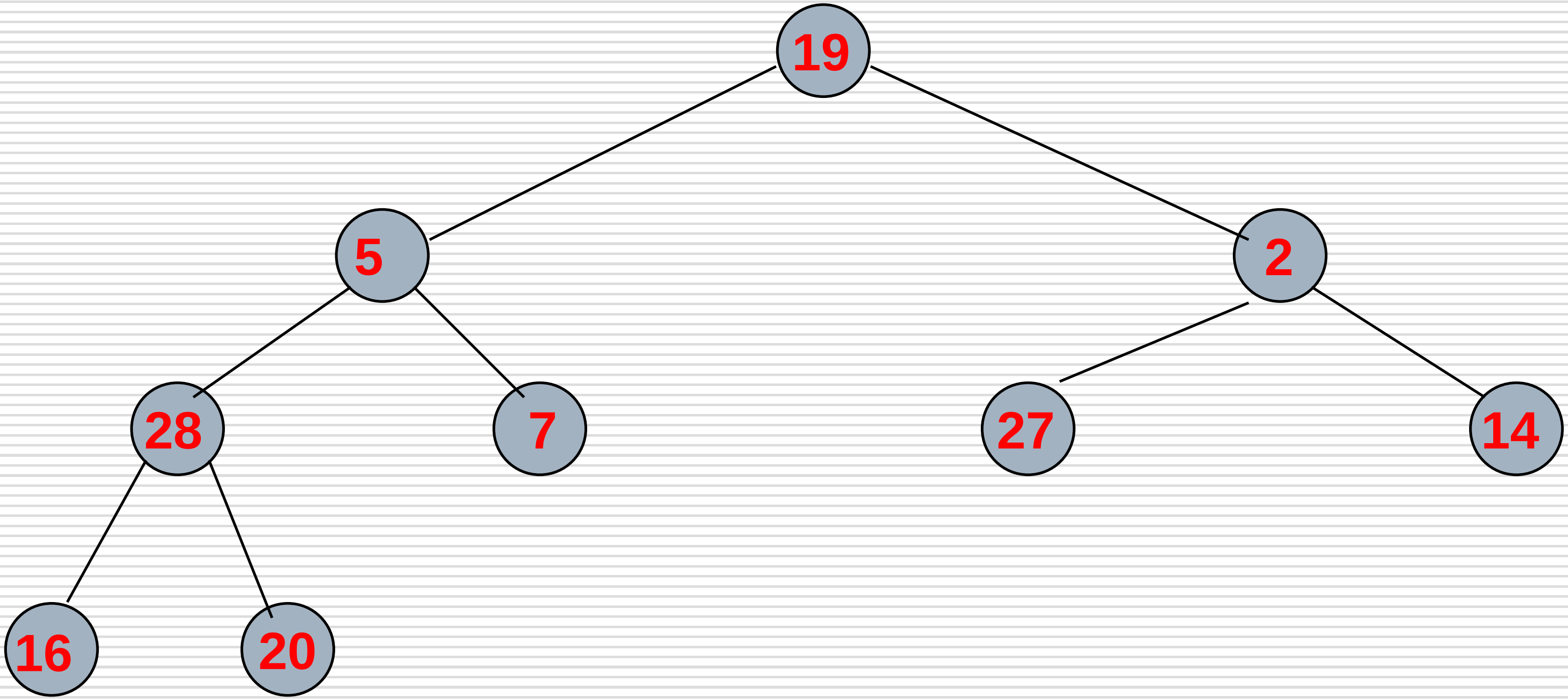
Build min Heap example



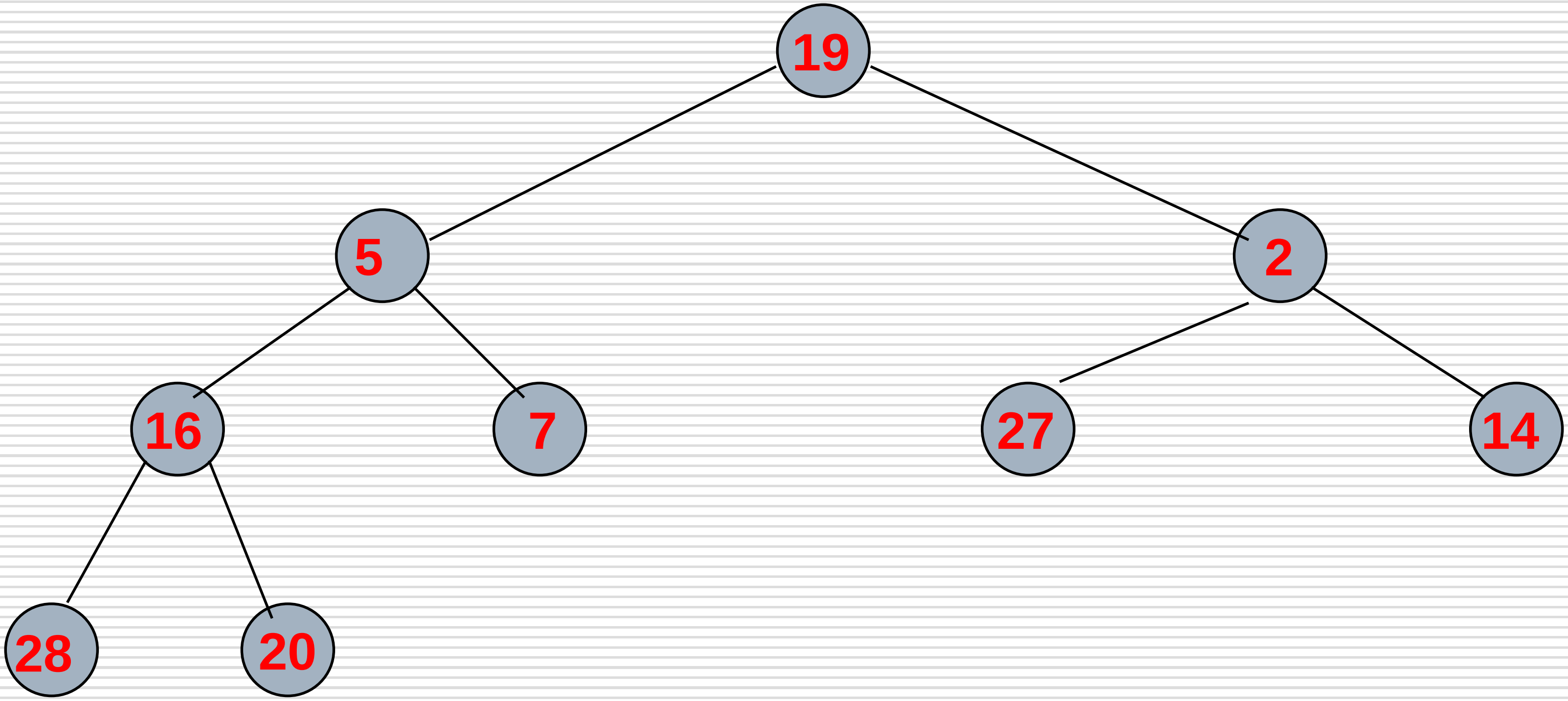
Build min Heap example



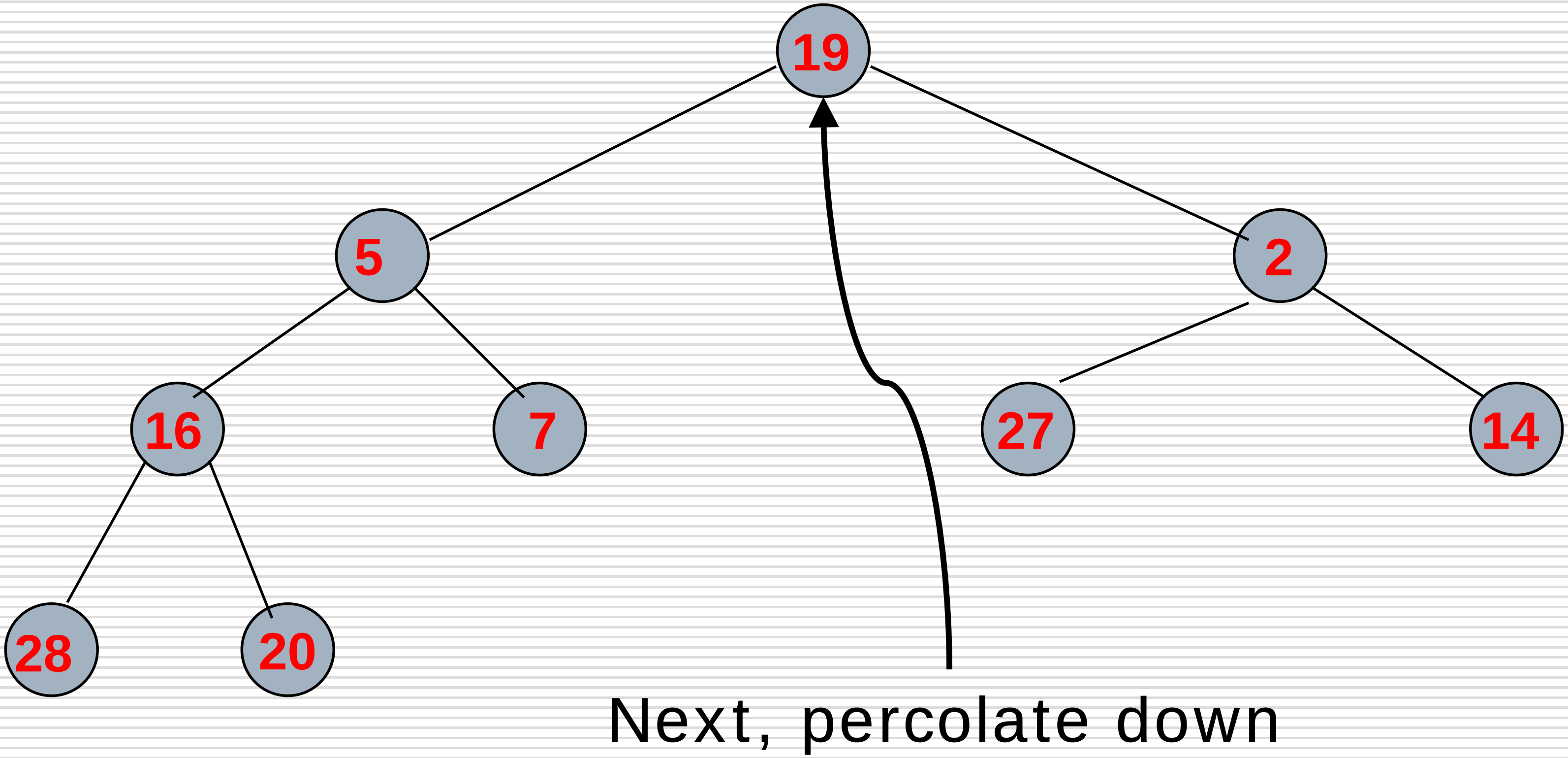
Build min Heap example



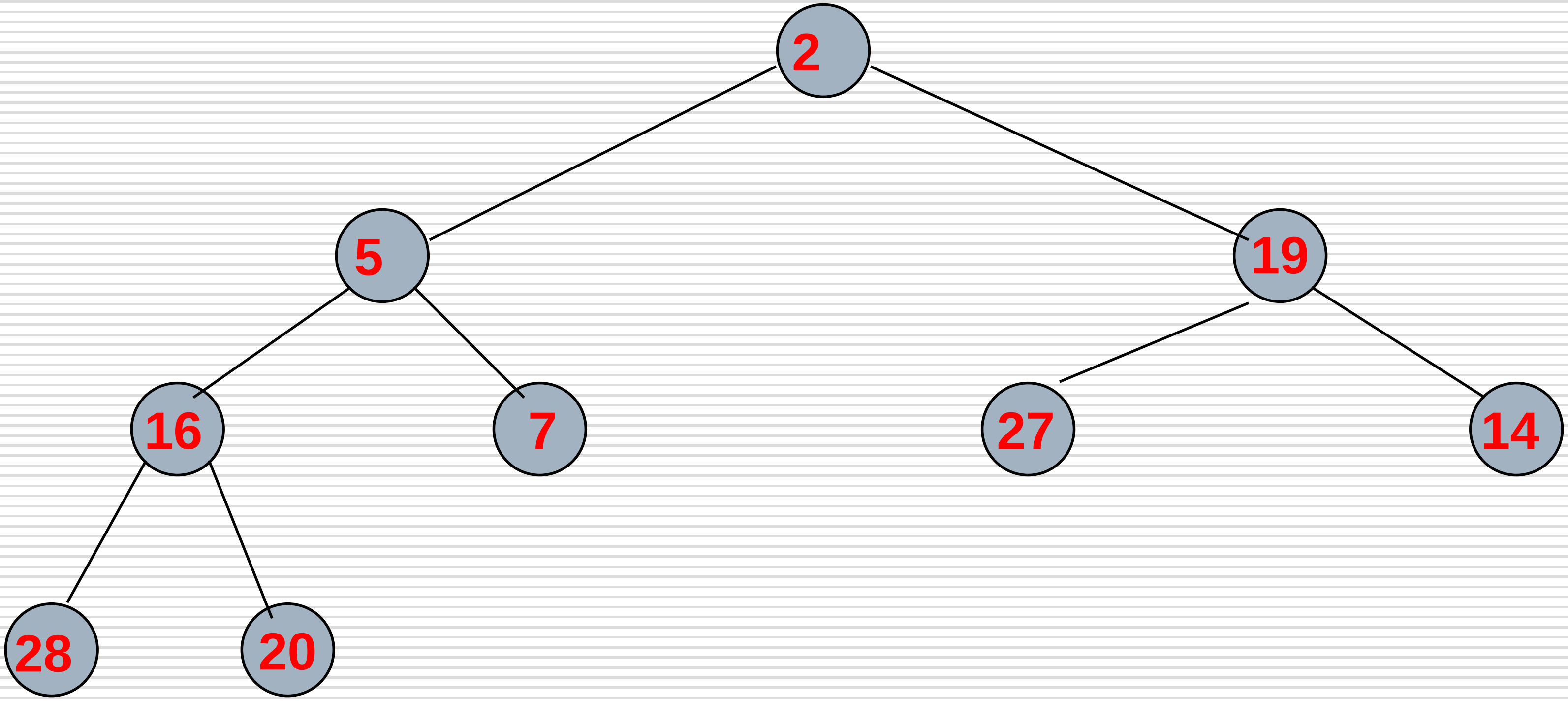
Build min Heap example



Build min Heap example



Build min Heap example



Build min Heap example