CSC2001F: Data Structures II

Omowunmi Isafiade

Email: omowunmi.isafiade@uct.ac.za

Office: Room 306

Outline

- Building Binary Heaps (from unsorted to sorted)
  - Recursively
  - Iteratively
- Internal Sorting - Heapsort
- Double Ended PQ - Min-Max Heap

Binary Heap – Build Heap Operation

- **Goal**: Take a binary heap that violates the heap order and reinstate it
- **Advantage**:
  - Reduces the cost of insertions from O(N log N) to O(N)
- Recall: An insertion takes O(log N) time (particularly if new element to be inserted is new "min")
  - Implies N insertions take O(N log N)
- Insertion becomes costly since heap order must be maintained after every insertion

Binary Heap – Build Heap Implementation

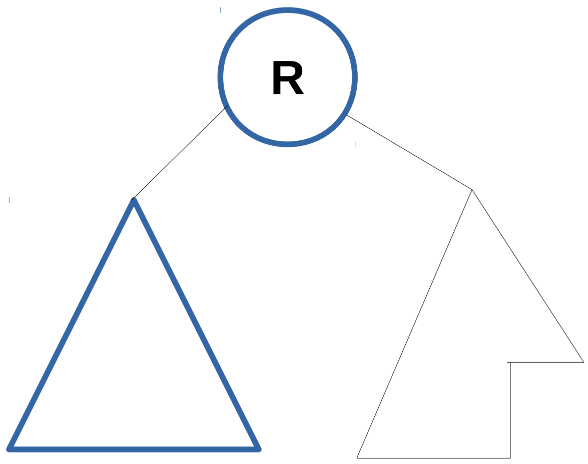
- **Goal**: Take a binary heap that **violates the heap order** and reinstate it

  - Recursive building: view heap as a recursively defined structure

  - Iterative building: view heap in terms of a hierarchy of elements that needs to be re-ordered from the bottom upwards

Build Heap Implementation (Recursively)

- Concept: view the heap as a recursively defined structure



- Recursively call "buildHeap" on the left and right sub-heaps. Then move the root element downwards in the tree until an appropriate position is found

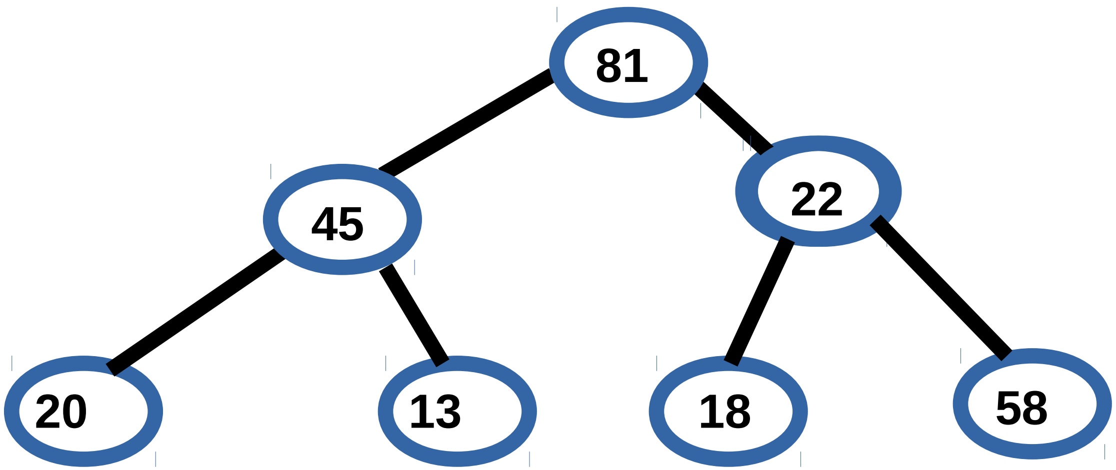
Build Heap Implementation (Iteratively)

- Concept: view heap in terms of a hierarchy of elements that needs to be re-ordered from the bottom upwards

- Principle: Re-order elements starting with leaf nodes moving towards the root node, in order to ensure ordering property is obeyed
  - Operates like the deletion method except no elements are removed

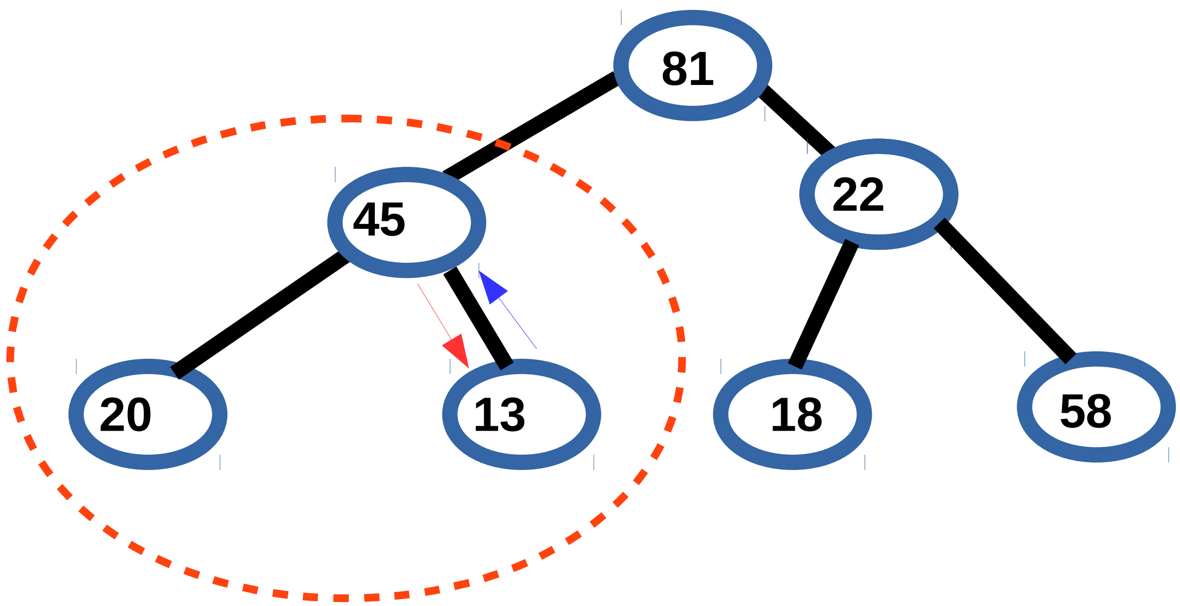
Build Heap Implementation (Recursively)

- Example: Building a heap recursively



Build Heap Implementation (Recursively)

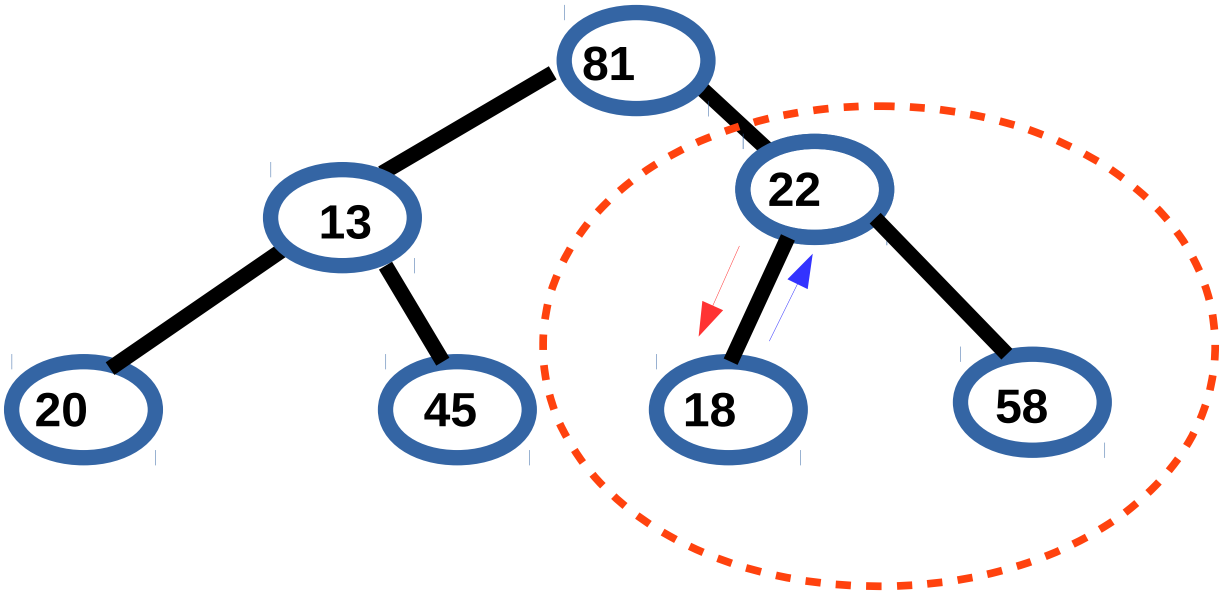
- Example:



- Compare 20 and 13, 13 < 20 and 13 < 45, so move 45 down

Build Heap Implementation (Recursively)

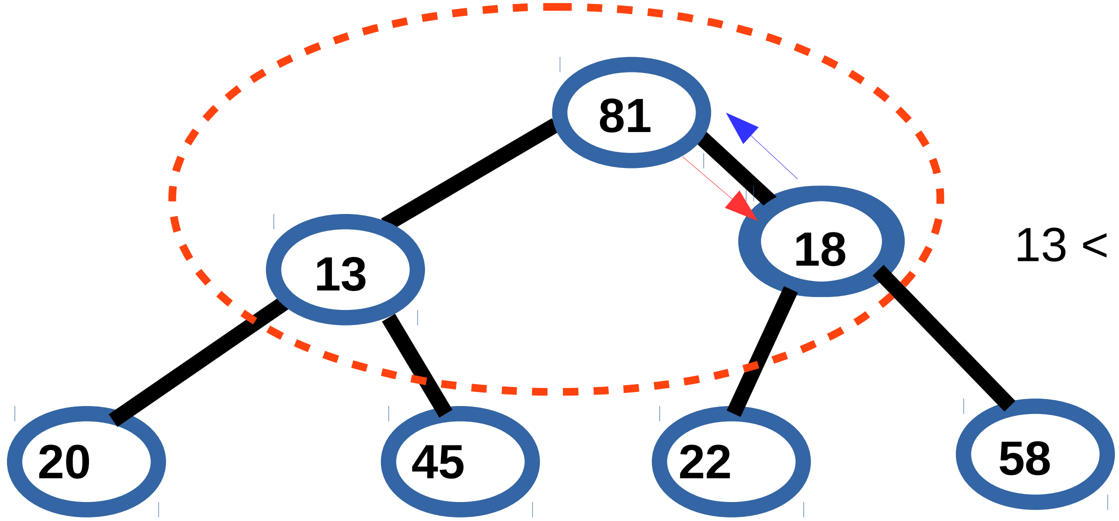
- Example:



- Likewise, simultaneously compare 18 and 58, 18 < 58 and 18 < 22, so move 22 down

Build Heap Implementation (Recursively)

- Example:

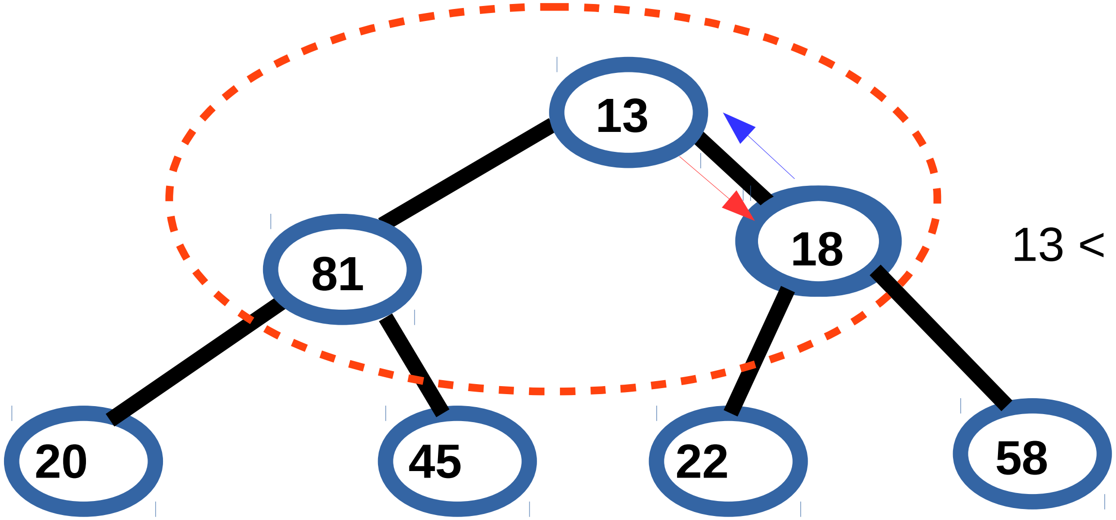


13 < 18, so moves up

- So left and right sub-heaps are sorted
- Final step: move 81 to its correct position

Build Heap Implementation (Recursively)

- Example:



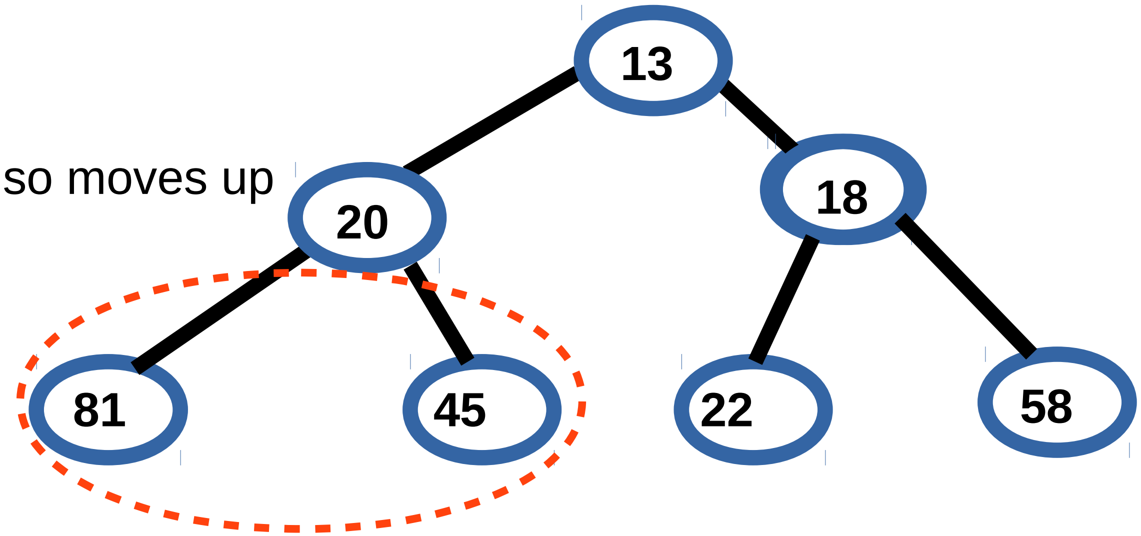
13 < 18, so moves up

- Final step: move 81 to its correct position

Build Heap Implementation (Recursively)

- Example:

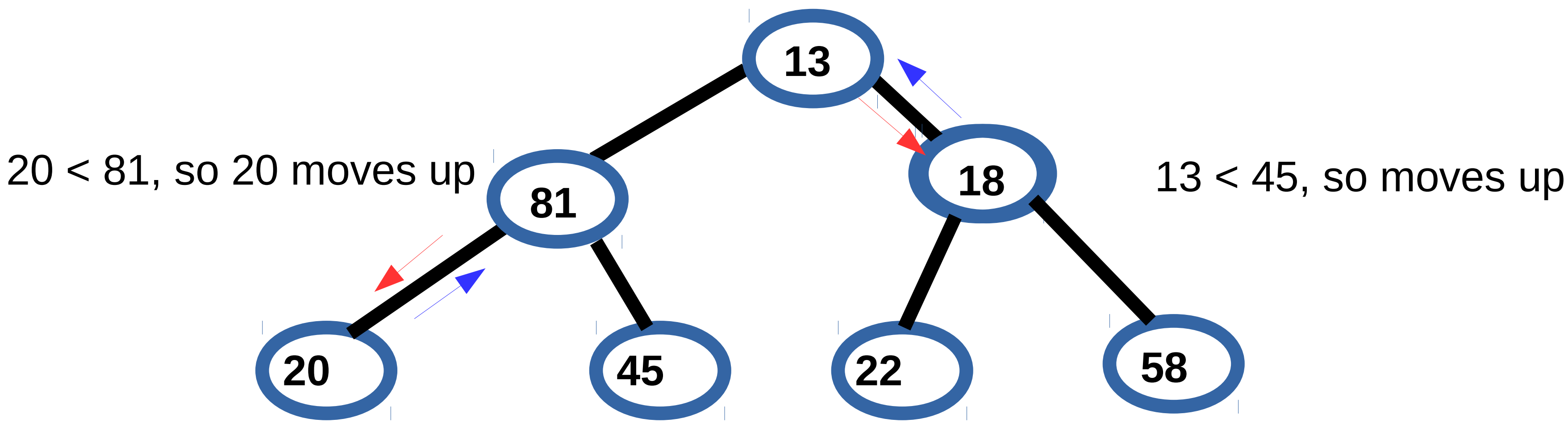
20 < 45 and 20 < 81, so moves up



- Final step: move 81 to its correct position

Build Heap Implementation (Recursively)

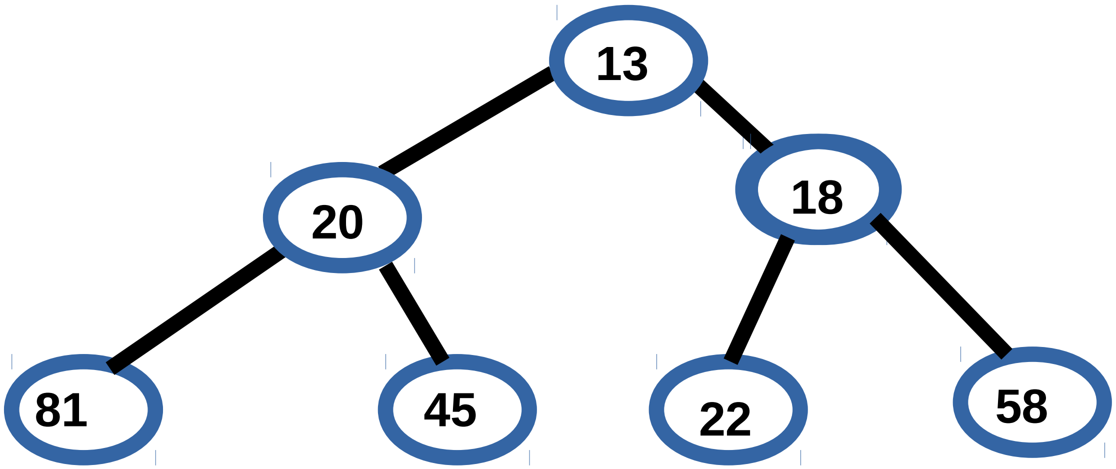
- Example:



- 81 finally in its correct position
- <u>Structure and order property in place</u>

Build Heap Implementation (Recursively)

- Example:



- Final step: move 81 to its correct position
- <u>Heap is finally sorted (recursively)</u>
- <u>Structure and order properties are in place</u>

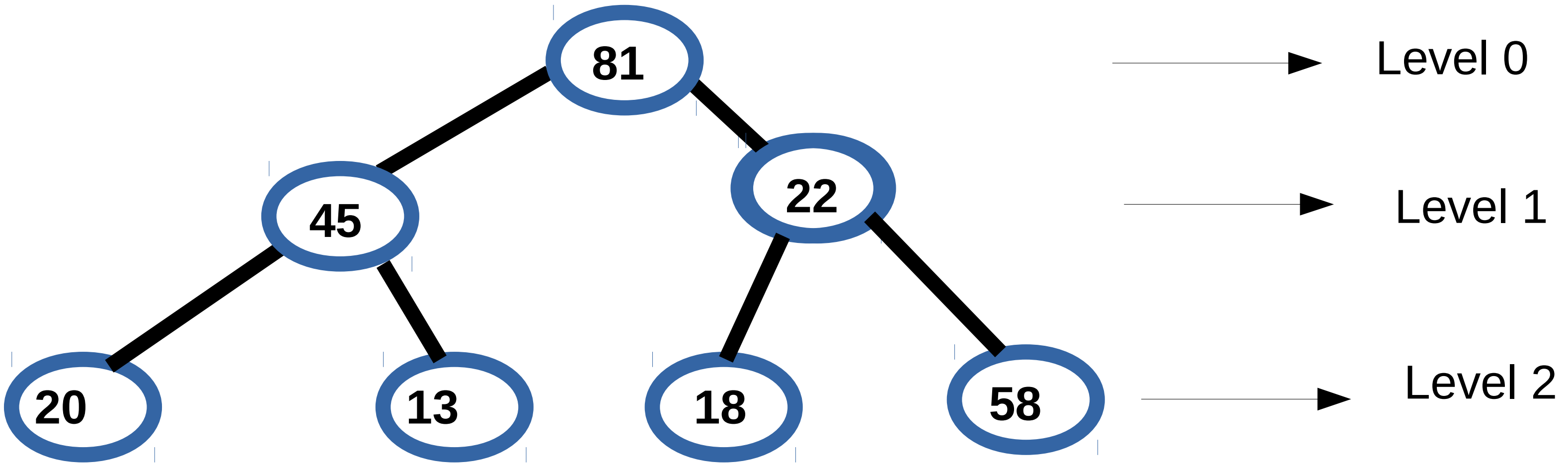
Build Heap Implementation (Iteratively)

- Concept: view heap in terms of a hierarchy of elements that needs to be re-ordered from the bottom upwards

- Principle: Re-order elements starting with leaf nodes moving towards the root node, in order to ensure ordering property is obeyed
  - Operates like the deletion method except no elements are removed

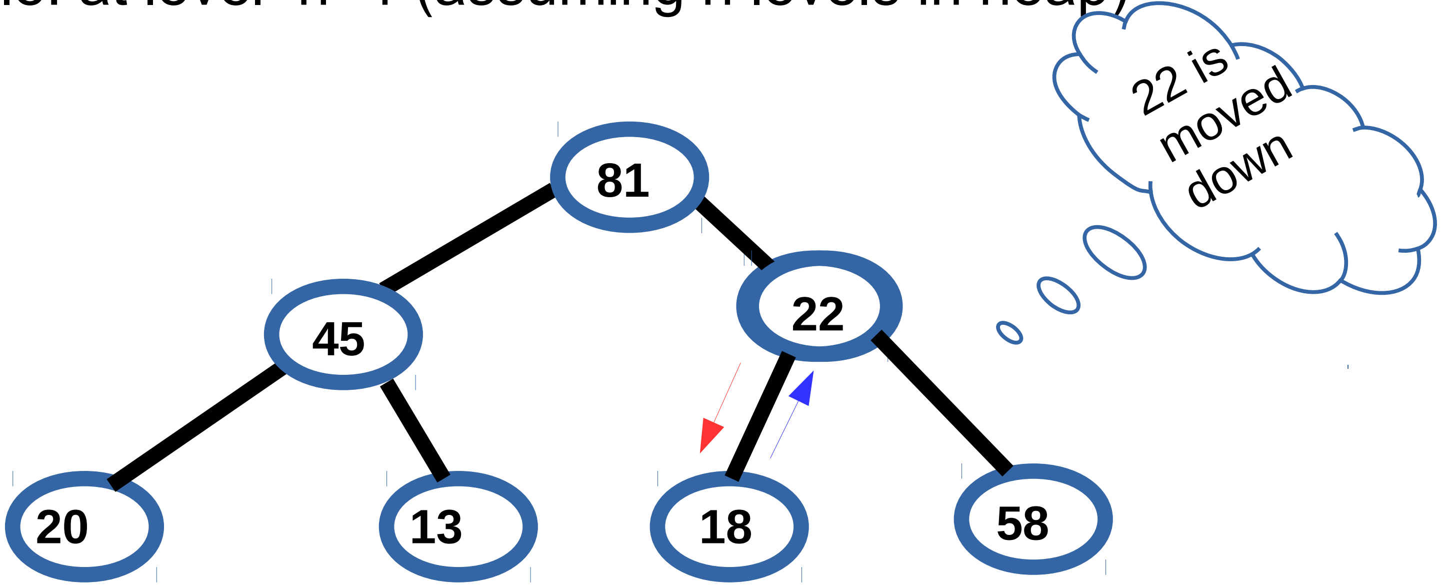
Build Heap Implementation (Iteratively)

- Example: No need to worry about ordering at (bottommost) leaf nodes, so move one level up



Build Heap Implementation (Iteratively)

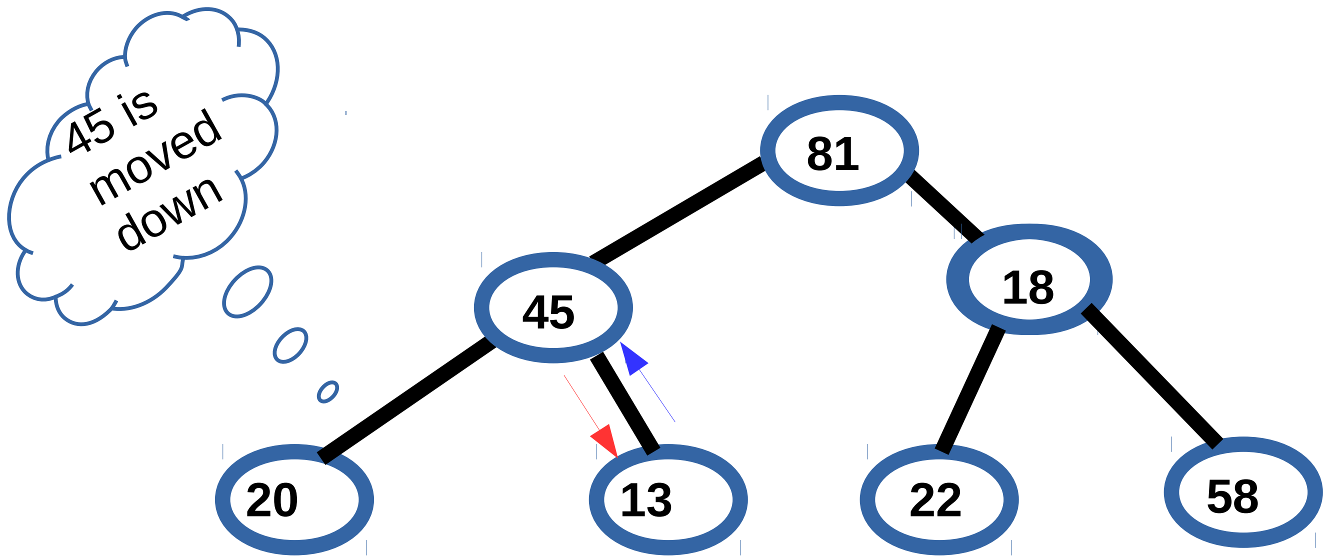
- Example: at level n -1 (assuming n levels in heap)



- Heap after ordering rightmost sub tree

Build Heap Implementation (Iteratively)

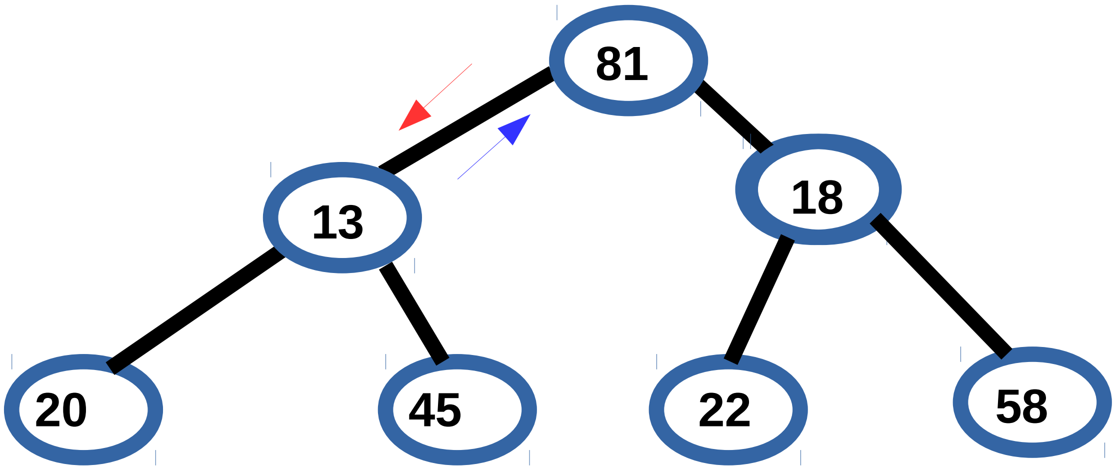
- Example: Still at level n-1, next sub-heap is examined



- Heap after ordering leftmost sub tree

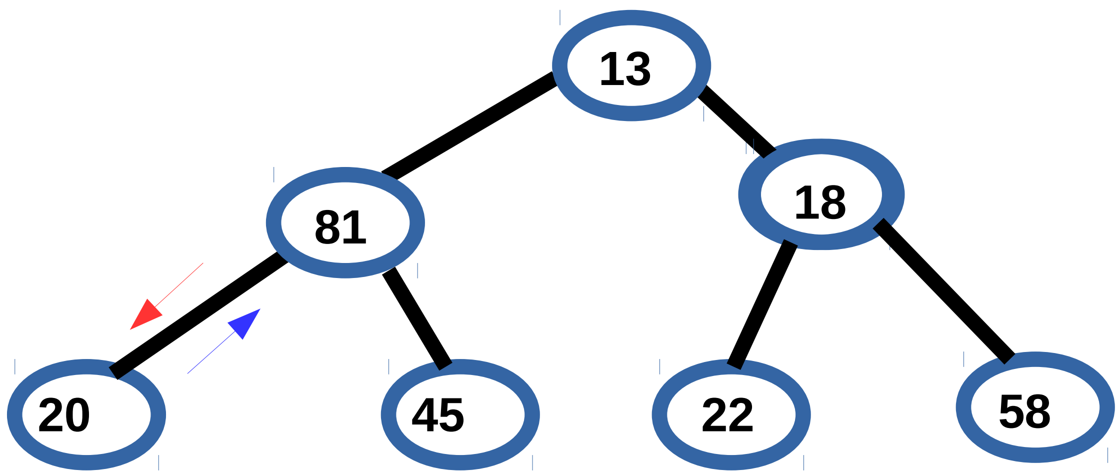
Build Heap Implementation (Iteratively)

- Example: at level n -1, check completed so move on to level n-2



Build Heap Implementation (Iteratively)

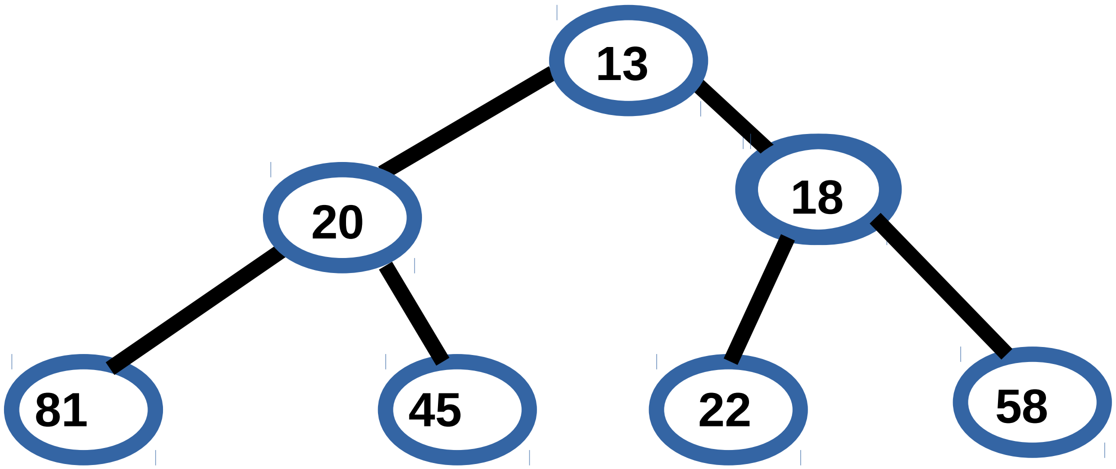
- Example: at level n-1, check completed so moving on to level n-2



- 81 is still not in the correct position with respect to its children

Build Heap Implementation (Iteratively)

- Example: at level n-1, check completed so moving on to level n-2



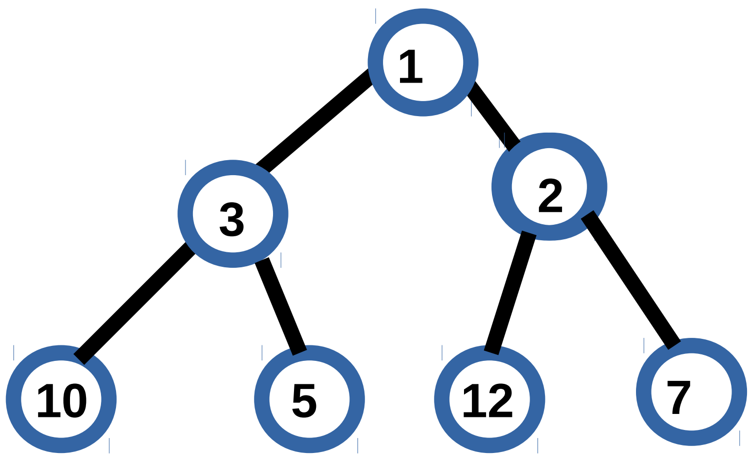
- 81 now in final position
- <u>Heap order is finally maintained</u>

BuildHeap vs Sequential Insertion

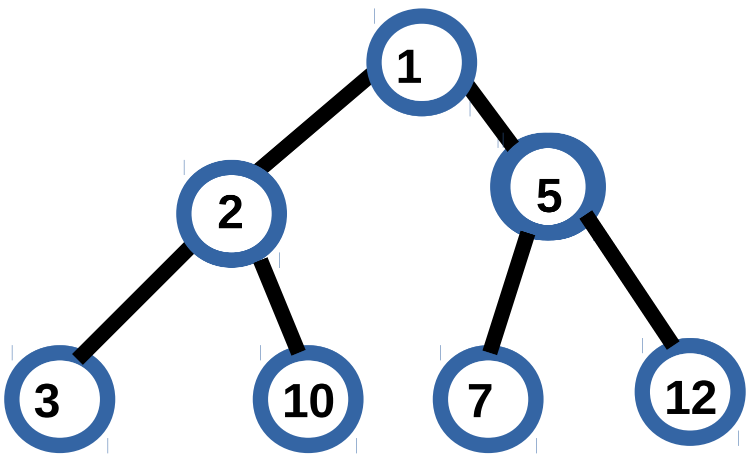
- **Exercise:** Show the result of inserting the following sequence **{5, 10, 12, 3, 2, 7, 1}** one at a time, in an initially empty heap.
- Then show the result of using the linear time buildHeap algorithm instead.

BuildHeap vs Sequential Insertion

- **Exercise:** Show the result of inserting the following sequence {5, 10, 12, 3, 2, 7, 1} one at a time, in an initially empty heap. Then show the result of using the linear time buildHeap algorithm instead.



Result: sequential insertion



Result: buildHeap

Internal Sorting: heapsort

- A priority queue can be used to sort N items as follows:
  - Insert every item into a binary heap
  - Extract every item by calling <u>deleteMin</u> or <u>deleteMax</u> N times

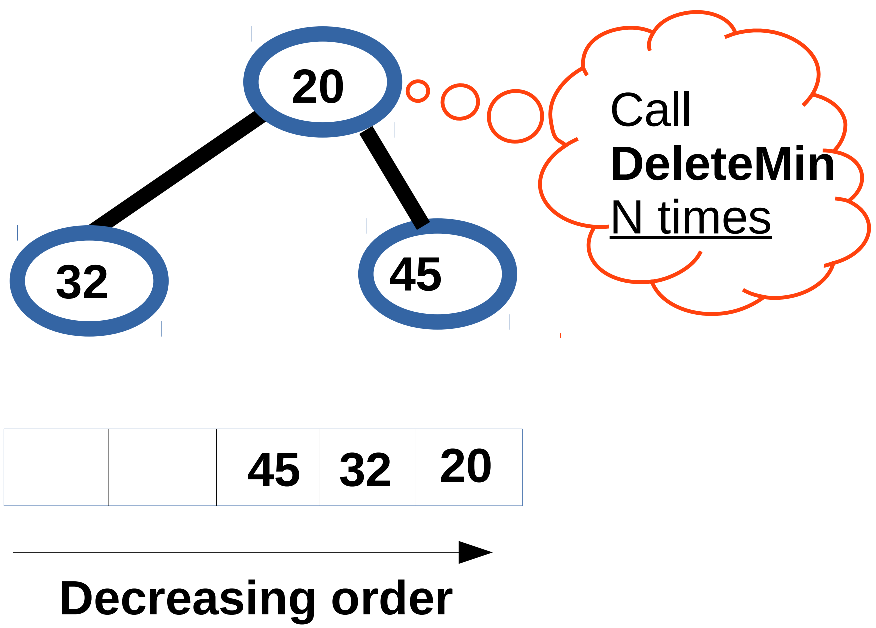
- By observation, we can implement this procedure more efficiently by:
  - Tossing the elements into a binary heap
  - Applying **<u>buildHeap</u>** (to order the heap)
  - Calling deleteMin or deleteMax N times to extract the items in sorted order

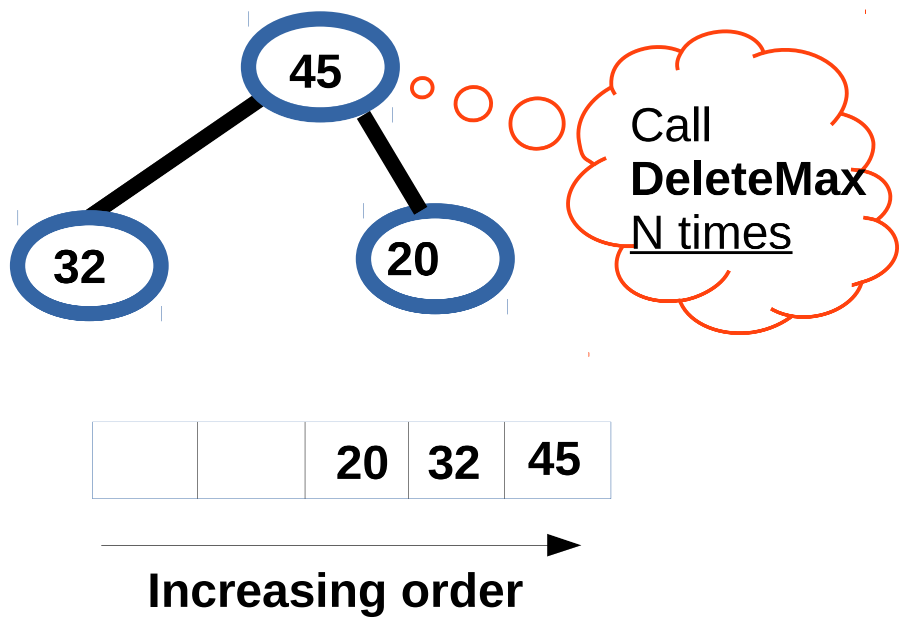
Internal Sorting: heapsort (Observations)

- Sorting like this with a binary heap is termed "heapsort"
- By using empty slots of the array, we can perform the sort in place
- If we use a max heap – obtain items in increasing order

- Heapsort:
  - Duplicates do not retain their initial ordering amongst themselves
  - Not a stable sorting algorithm
  - Internal sorting: assumes all data will fit in memory

Internal Sorting Example : Heapsort





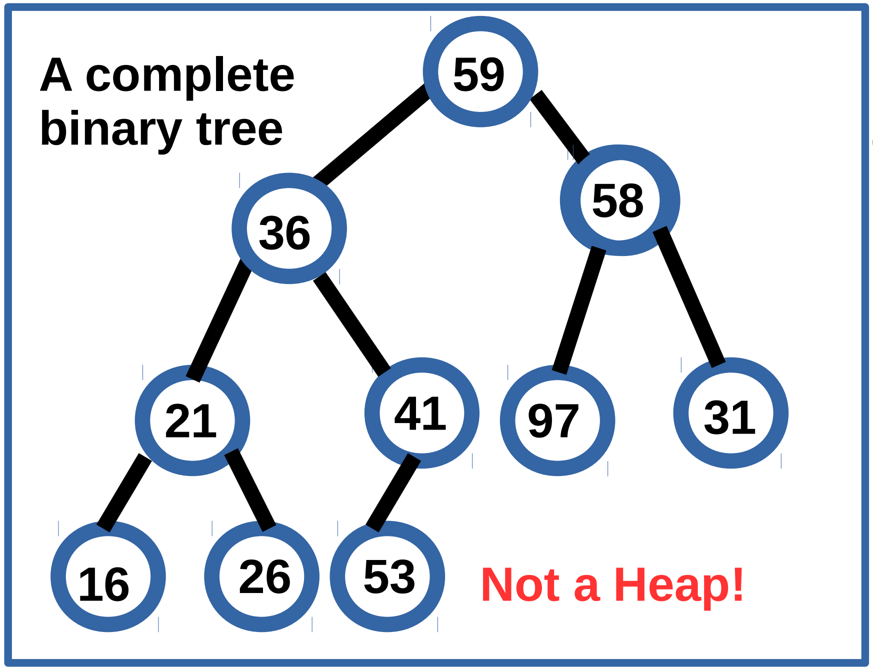
- **Note:**
  - If we use a max heap – obtain items in increasing order
  - If we use a min heap – obtain items in decreasing order

Heapsort – Internal Sorting: Exercise

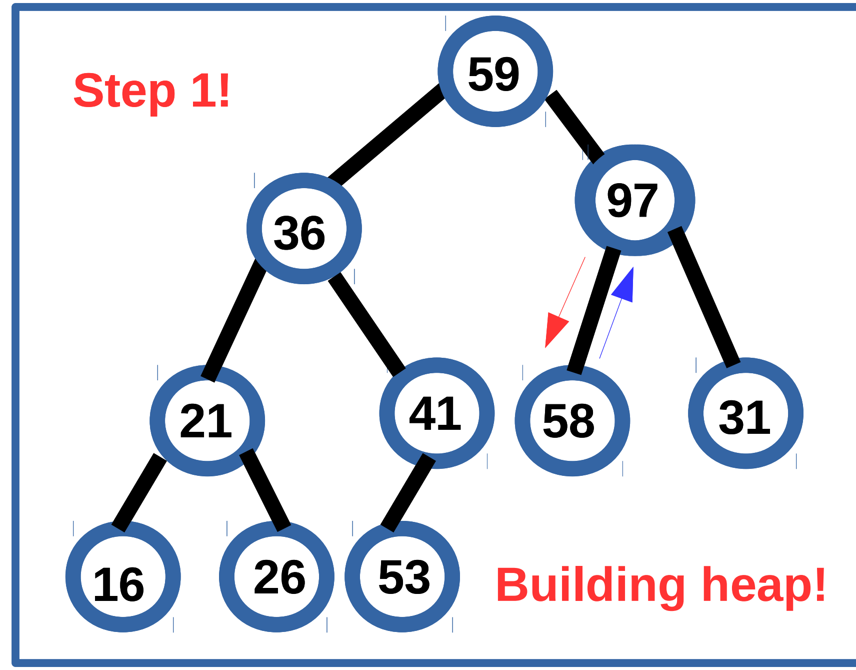
- Sort the following input sequence using heap sort
- {59, 36, 58, 21, 41, 97, 31, 16, 26, 53}
- Note: show the resulting heap and array representation at every step (**assume max heap**)

- Solution Strategy
  - First represent the sequence of items in a (complete) binary tree
  - Apply "buildHeap" to the resulting tree (order property)
  - Give the implicit representation of the resulting heap (**note:** root node should start at index 0 – no sentinel)
  - Then call deleteMax N times (insert max value at empty slot in the array each time deleteMax is called)

Internal Sorting: Exercise - Solution



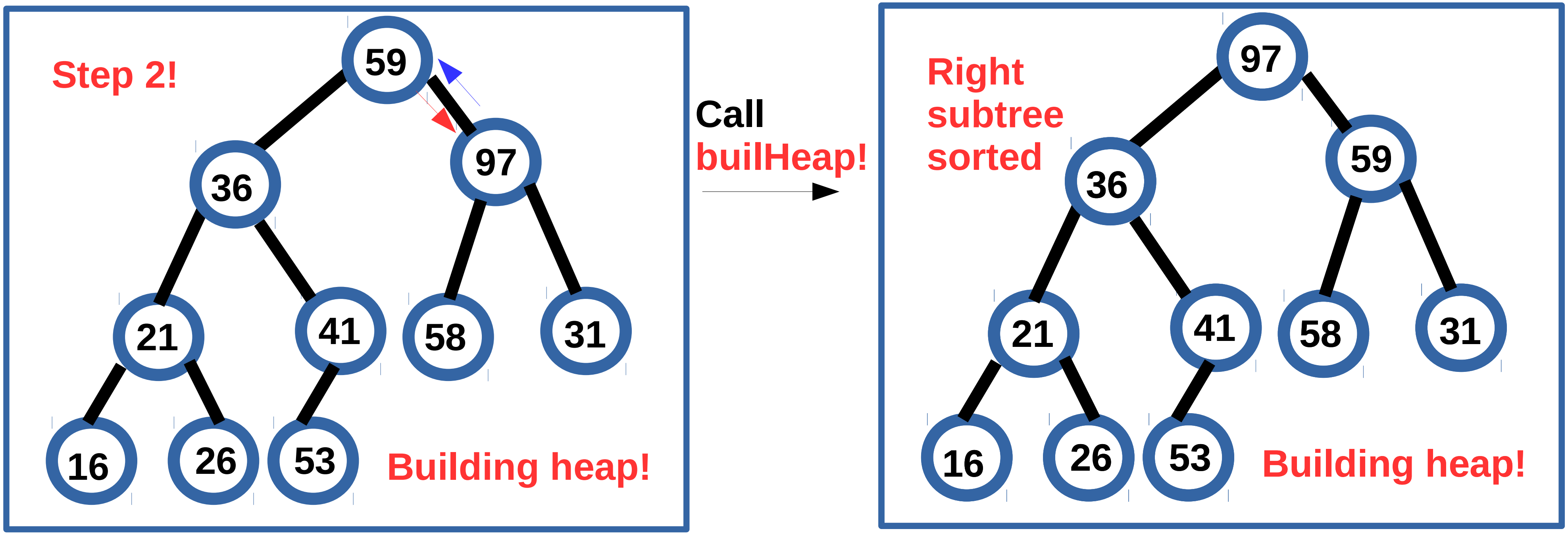
Call
builHeap!



Follow the procedure for buildHeap until every node is correctly placed

Note:
*** All operations are aimed at finding a correct slot for the items**
*** Order and structure properties must be strictly obeyed**

Internal Sorting: Exercise - Solution



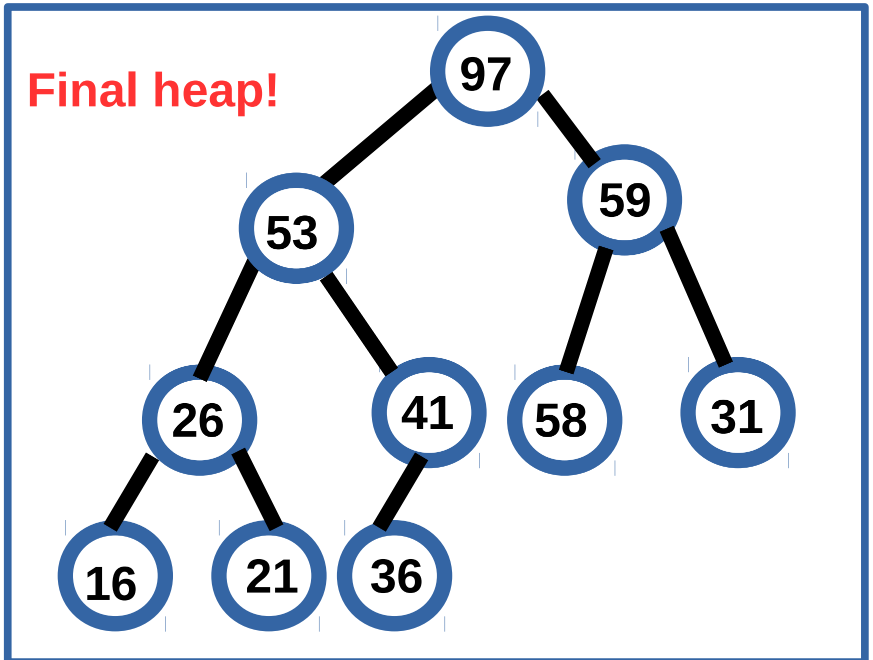
Follow the procedure for buildHeap until every node is correctly placed

Note:

*** All operations are aimed at finding a correct slot for the items**

*** <u>Order and structure properties</u> must be strictly obeyed**

Internal Sorting: Exercise - Solution



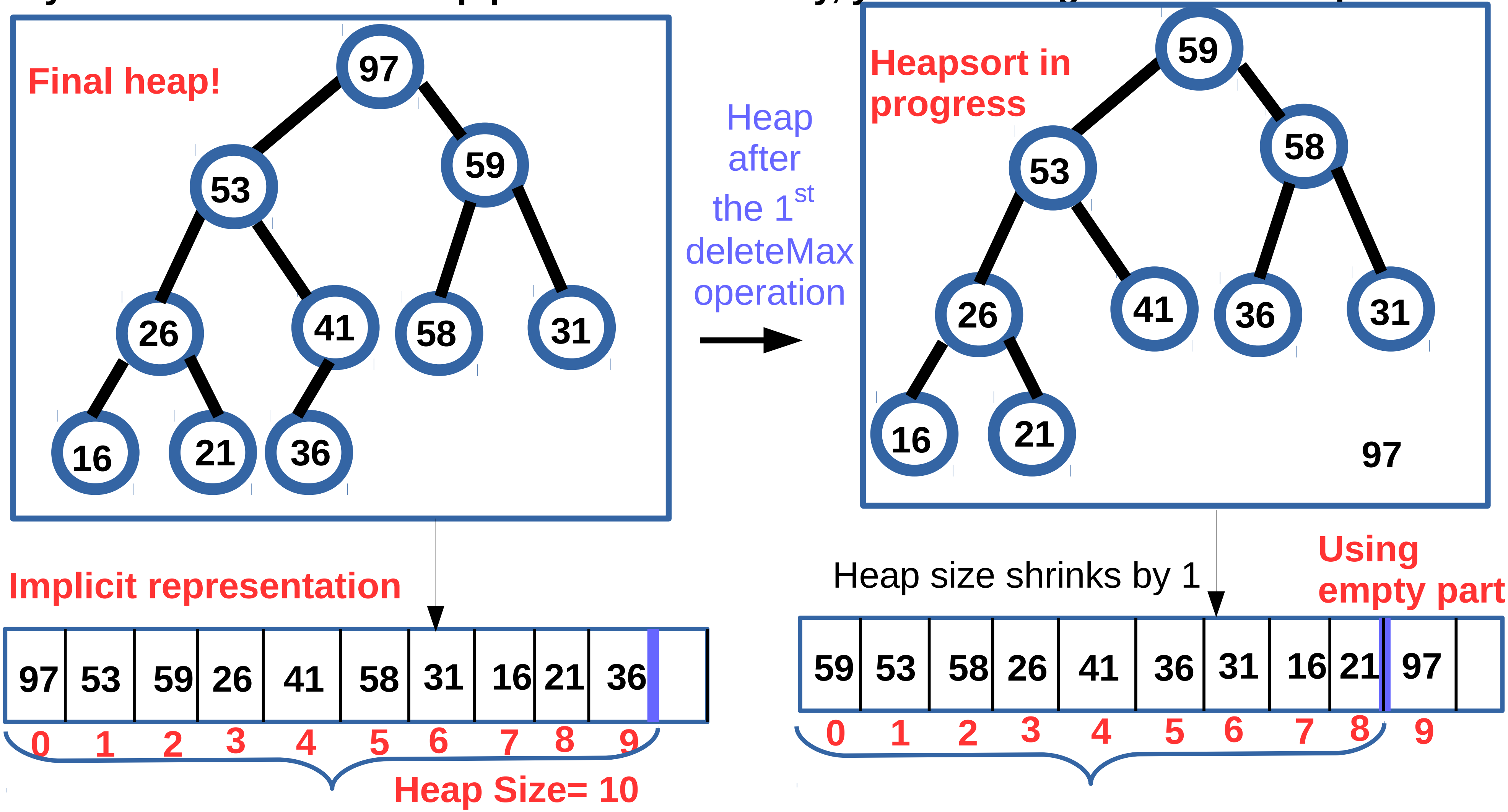
Implicit representation

0	1	2	3	4	5	6	7	8	9				
97	53	59	26	41	58	31	16	21	36				

Now call deleteMax N times to obtain heapsort result

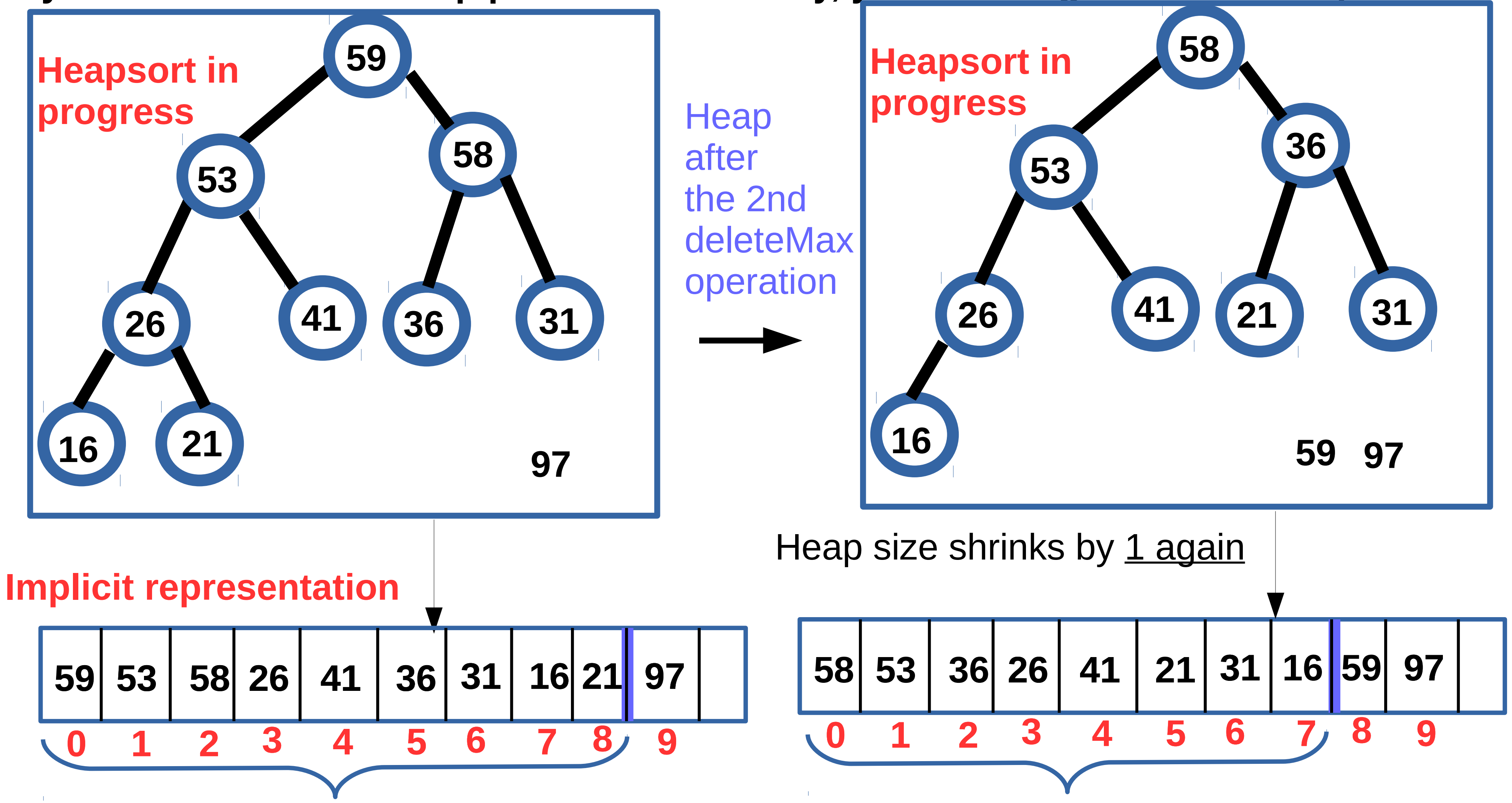
Internal Sorting: Exercise - Solution

If you follow the buildHeap procedure correctly, you should get the max heap below



Internal Sorting: Exercise - Solution

If you follow the buildHeap procedure correctly, you should get the max heap below



Heap Sort - Observation

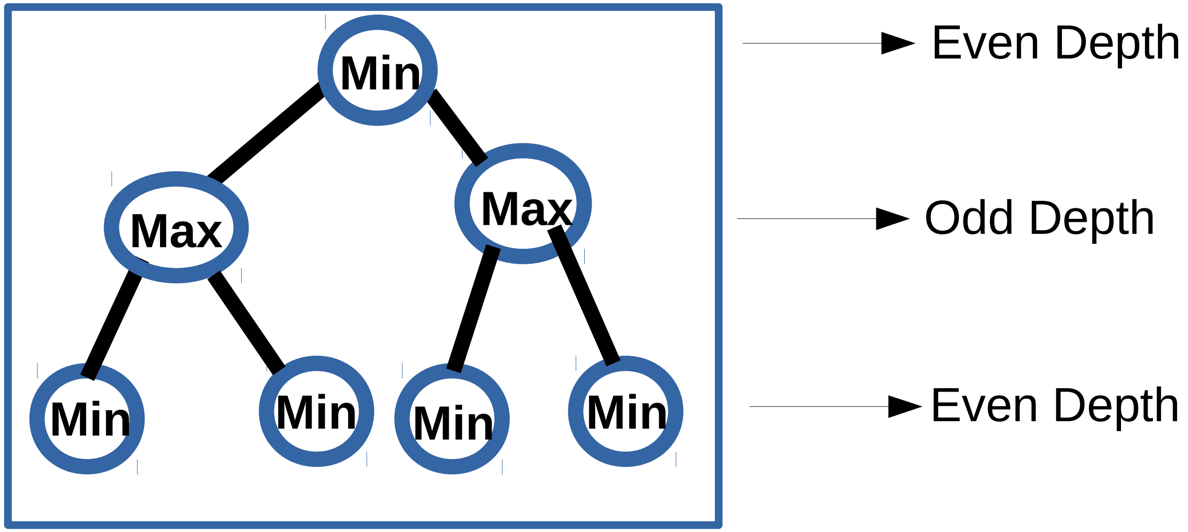
- If you continue the process correctly in the exercise, all items would have been sorted in <u>increasing sequence order</u>

- Note: Heapsort is not as fast as quicksort, it can still be useful
  - But! It is certainly easier to implement

- Heapsort:
  - Duplicates do not retain their initial ordering amongst themselves
  - Not a stable sorting algorithm
  - Internal sorting: assumes all data will fit in memory

Double Ended PQ: Min – Max Heap

- A double ended priority queue (PQ) is a data structure that supports the following operations:
  - Inserting an element with an arbitrary key
  - Deleting an element with the smallest key
  - Deleting an element with the largest key
- A <u>Min-Max Heap</u> supports all of the above operations.



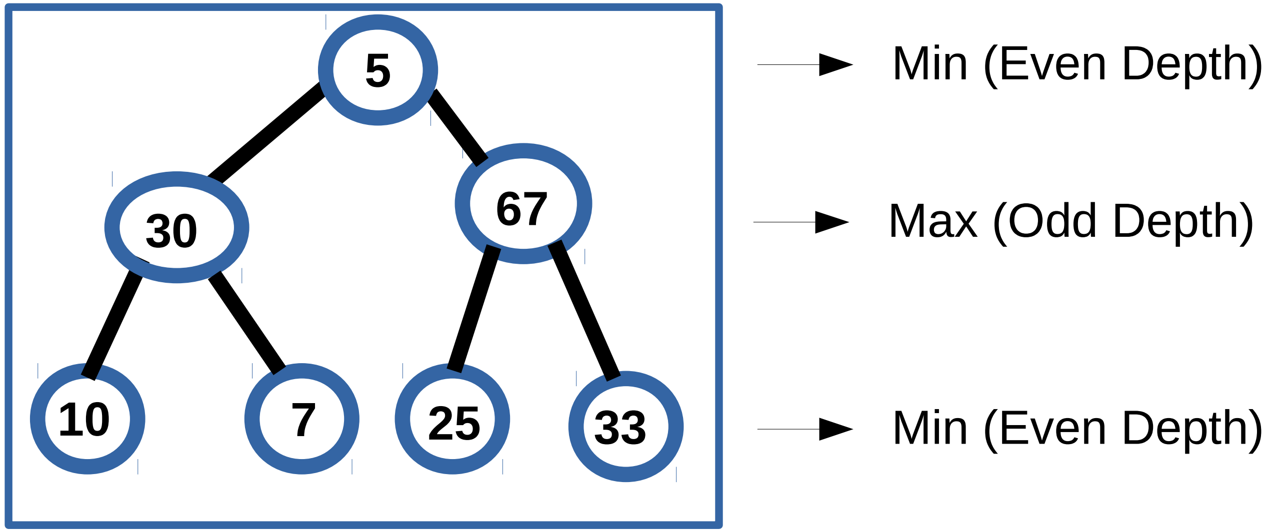
Min–Max Heap: What it is

- Min-max heap:
  - A data structure that supports both deleteMin and deleteMax operations at logarithmic cost.
  - The structure is identical to the binary heap (complete B-tree)
  - Min-max ordered

- Min-max heap order property:
  - For every node X at even depth, the key stored at X is the smallest in its subtree
  - For every node X at odd depth, the key stored at X is the largest in its subtree
  - The root is at even depth

Min–Max Heap: Example

- Example: A 7- element min-max heap



- Note:
  - For every node X at even depth, the key stored at X is the smallest in its subtree
  - For every node X at odd depth, the key stored at X is the largest in its subtree (root is at even depth)

Min-Max Heap Operations - Insertions

- **Note:** <u>Structure and order properties </u>must always be obeyed
- Methodology:
  - Create a new node in the tree in next available position (to avoid violating structure property – complete binary tree)
  - Check to ensure that <u>min-max</u> order property is satisfied

- General Strategy ("***Percolate up"***)
  - Create a hole at the next available location
    - If heap order is not violated, place item in the hole
    - else "bubble-up" the hole toward the root

Min-Max Heap Operations - Insertions

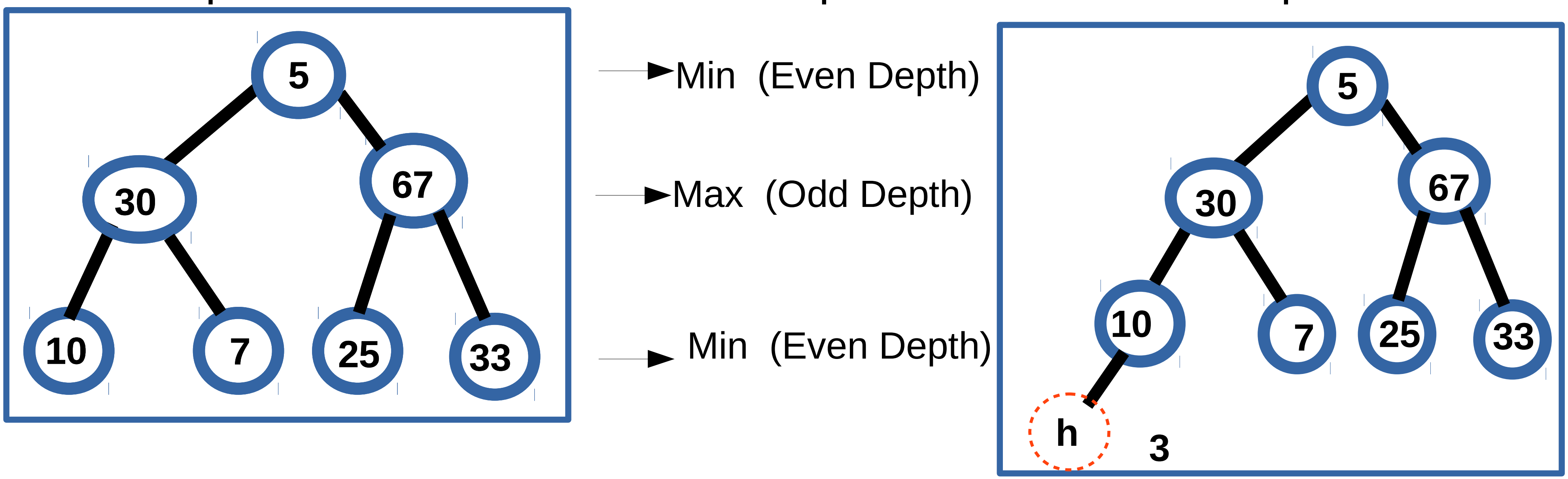
- **Note:** When inserting an element X into a min-max heap
  - **First:** create a hole (h) at the next available location (**structure property**). So X is to be inserted in "h"
  - If heap order is not violated, place item in the hole else "bubble-up" the hole toward the root as follows:
    - Compare X with its parent node (P), if X < P and P is at odd depth (max-level). Then X is guaranteed to be smaller than all keys in nodes that are both on max levels and on the path from "h" to root.
    - So, <u>only need to check nodes on **min levels**</u>

Min-Max Heap Operations - Insertions

- **Note:** When inserting an element X into a min-max heap
  - Conversely, if X > P and P is at even depth (min-level), then X is guaranteed to be larger than all keys in the nodes that are both on min levels and on the path from "j" to the root.
    - So, only need to check nodes on **max levels**

Min–Max Heap: Exercise in Class

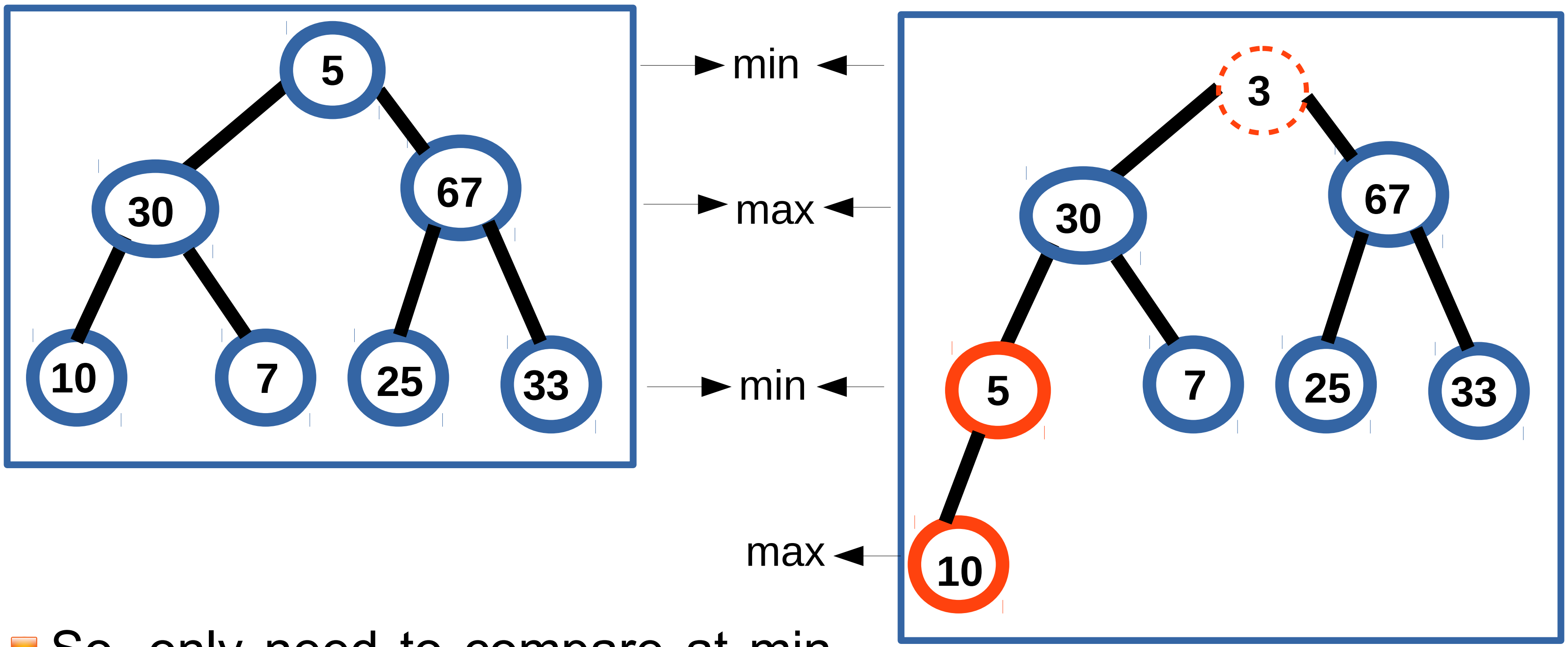
- Example: A 7-element min-max heap. Insert 3 into the heap.



- create a hole (h) at the next available location (**structure property**). So 3 is to be inserted in “h”.
- Now since 3 < 10 and 10 is at even depth (min-level). Move 10 down.
- Compare 3 with (new) P (i.e 30). Since 3 < 30 and 30 is on max level. We are guaranteed that 3 is < all keys in nodes that are both on max levels and on the path from (new) “h” position to root.

Min–Max Heap: Insertion Exercise in Class

- Example: A 7-element min-max heap. Insert 3 into the heap.



- So, only need to compare at min level(s) afterwards.
- Since 3 < 5 => 5 moves down

Min-Max Heap Operations - Deletion

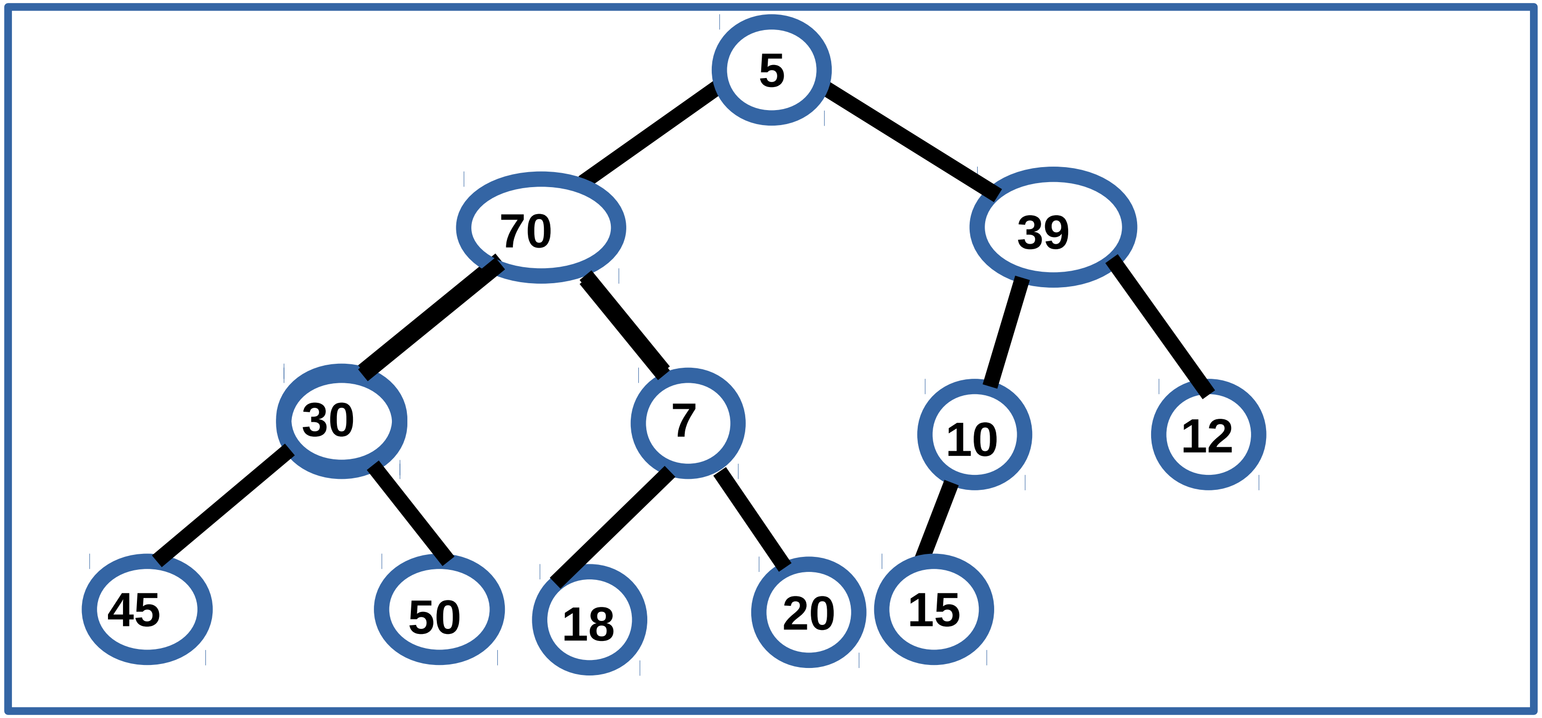
- **Note:** When deleteMin is called in a min-max heap
  - **DeleteMin - obvious @ root node**
  - **Recall:** last item (X) at the bottom level has to be placed in an appropriate slot (to maintain tree structure). Now, **2 steps to follow:**
    - If root has no children, then X is inserted into the root node
    - If root has at least 1 child => smallest key in the min-max heap is in one of the children/grandchildren of the root. Assume node k has the smallest key, then consider the following:
      - X <= h[k].key, then X goes to root
      - If X > h[k].key and k is a child of the root, since k is a max node, we are sure that there is no descendants of k with a larger key than X. So h[k] moves to root and X inserted into node k

Min-Max Heap Operations - Deletion

- **Note:** When deleteMin is called in a min-max heap
  - If root has at least 1 child => smallest key in the min-max heap is in one of the children/grandchildren of the root. Assume node k has the smallest key, then consider the following:
    - X <= h[k].key, then X goes to root
    - If X > h[k].key and k is a child of the root, since k is a max node, we are sure that there is no descendants of k with a larger key than X. So h[k] moves to root and X inserted into node k
    - Else If X > h[k].key and k is a grandchild of the root, h[k] moves to the root. Suppose P is the parent of K, if X > P then h[p] and X are swapped

Min–Max Heap: Exercise in Class

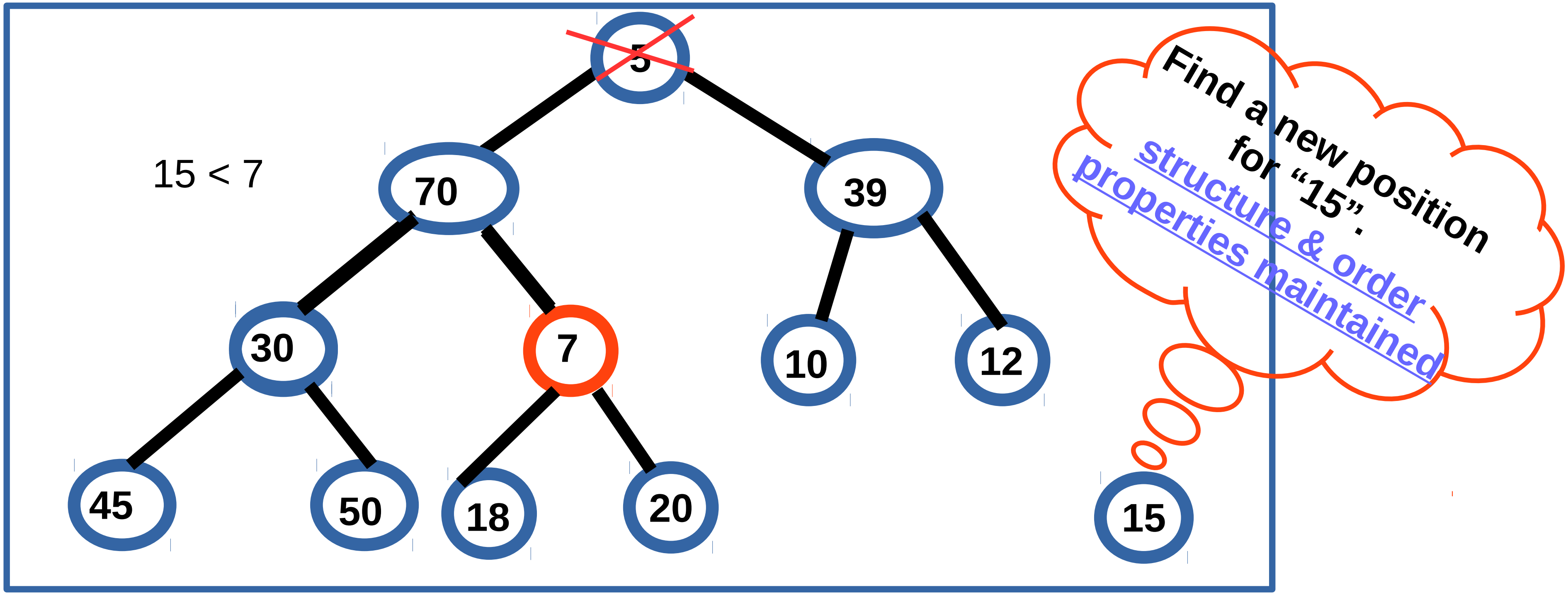
- Example: A 12-element min-max heap. **<u>deleteMin</u>** from the heap and show the resulting solution



<u>Note</u>: You need to find a new slot for 15, since order and structure properties Must be satisfied

Min–Max Heap: Exercise in Class

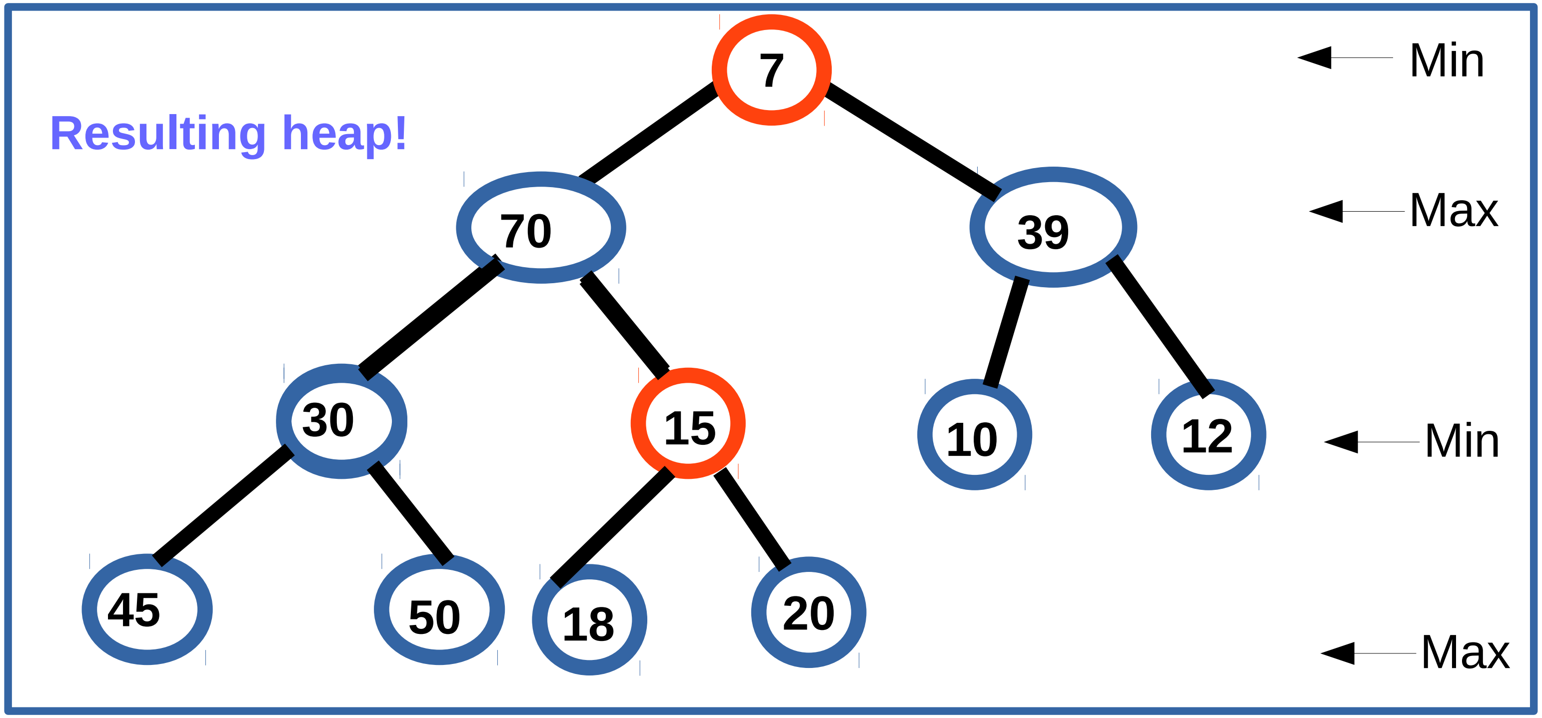
- Example: A 12-element min-max heap. **deleteMin** from the heap and show the resulting solution



Note: You need to find a new slot for 15, since order and structure properties Must be satisfied

Min–Max Heap: Exercise in Class

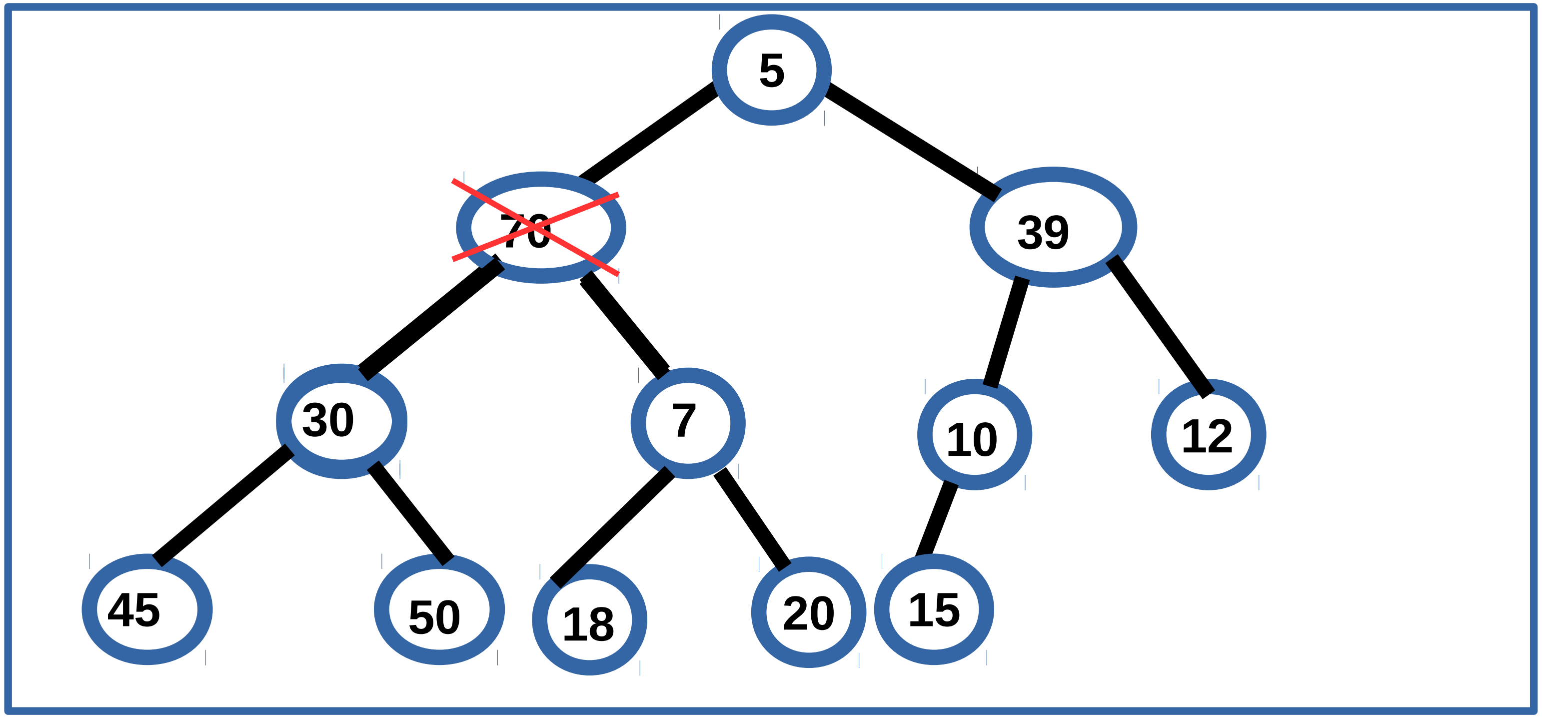
- Example: A 12-element min-max heap. **<u>deleteMin</u>** from the heap and show the resulting solution



<u>Note</u>: In this case, minimum item is found in root node's grandchildren after comparison. Therefore the nodes are swapped accordingly

Min–Max Heap: Exercise in Class

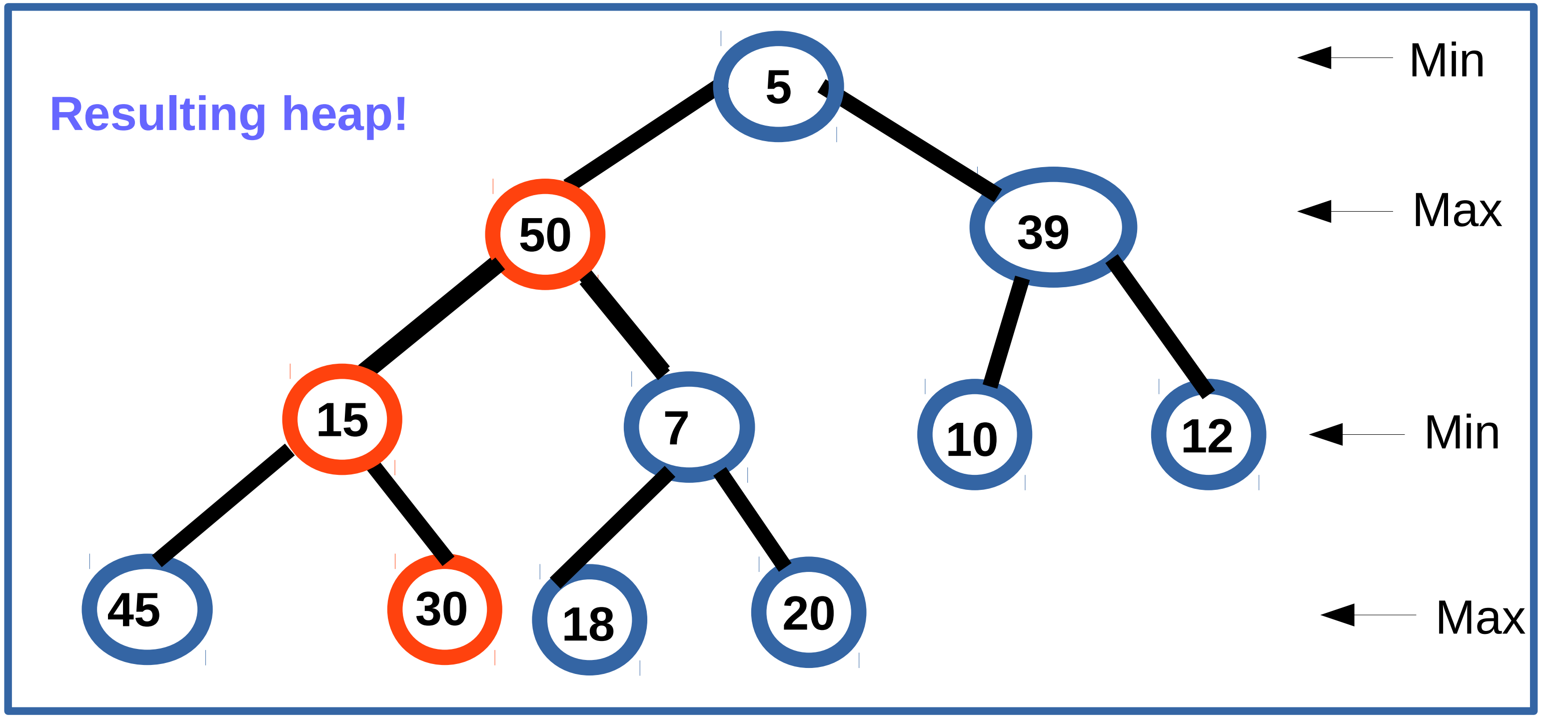
- Example: A 12-element **min-max** heap. **deleteMax** from the heap and show the resulting solution



Note: You need to find a new slot for 15, since order and structure properties Must be satisfied. 70 is the max and deleted in this case

Min–Max Heap: Exercise in Class

- Example: A 12-element min-max heap. **deleteMax** from the heap and show the resulting solution



Other Nice Things To Know About PQs...

- **Theorem 1**: An almost complete binary tree with N internal nodes has height $\lfloor \log(N) \rfloor + 1$
- Proof: (by induction)
- Recall that a complete binary tree (heap) of height h has $(2^h - 1)$ internal nodes. This can be proved by simple induction on *h*.
- Base Case: A 1 node heap has height 1. $(\lfloor \log(1) \rfloor + 1 = 1)$
- Inductive Step: Since the number of nodes in a heap of height h is > the number of nodes in a heap of height h-1, and at most the number of nodes in a tree of height h; for the n nodes in a heap of height h, we have: $2^{(h-1)} - 1 \leq N \leq 2^h - 1$
- For all N with $2^{(h-1)} - 1 \leq N \leq 2^h - 1$ we have $\lfloor \log(N) \rfloor = h - 1$ ■

Other Nice Things To Know About PQs...

- Insertion time is obtained from the observation that by theorem 1, $h \in \theta(\log N) \rightarrow$ the running time for insertions is $\theta(\log N)$

- Returning the max/min element can be done in O(1) time with a reasonable heap implementation

- But! Removing the maximum requires O(log N) time because it is in fact quite similar to insertion. Root element is replaced with an element at the furthest left node

Priority Queues - Exercise

- For the following key sequence determine the binary heap obtained when the keys are inserted sequentially (one at a time) into an initially empty heap

- Assume maximum value at root node (max-heap)

- 0,1,2,3,4,5,6,7,8,9

Priority Queues - Exercise

- For the following key sequence determine the binary heap obtained after 3 consecutive DequeueMax (DeleteMax) operations

- 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Min–Max Heap: Exercise

- **Example**: Insert the following sequence into a **min-max heap** {10, 11, 5, 13, 19, 22, 9, 8, 25, 7, 2} and show the final heap
- Perform a DeleteMin operation on the heap and show the resulting **min-max heap**

IMPORTANT NOTE

- You are **strongly advised** to practice **all** the exercises and examples in the lecture notes.
- Remember to also work on your hash tables assignment.

- Good luck and enjoy the vac!

Next Class...

- Graphs & Paths…

Reference Textbook:

“Data Structures & Problem Solving using Java”, 4th Ed., Mark A. Weiss.