Binary Heaps & Priority Queues

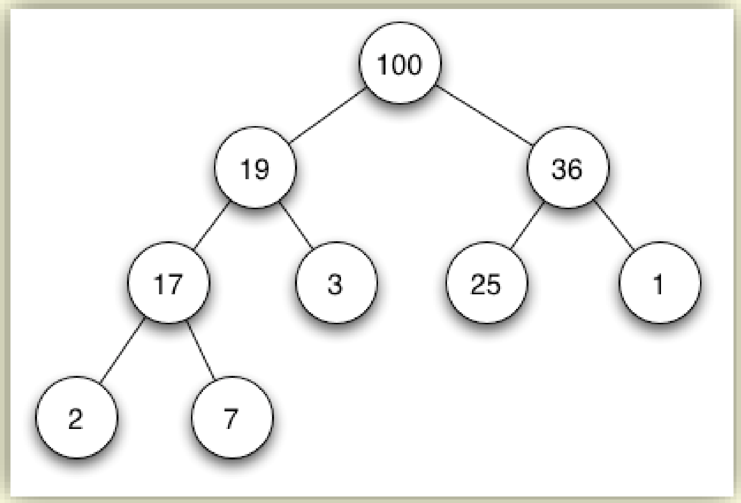
Kumkum Saxena

Binary Heaps

- Heap:
  - A heap is an Abstract Data Type
    - Just like stacks and queues are ADTs
    - Meaning, we will define certain behaviors that dictate whether or not a certain data structure is a heap
  - So what is a heap?
    - More specifically, what does it do or how do they work?
  - A heap looks similar to a tree
    - But a heap has a specific property/invariant that each node in the tree MUST follow

Binary Heaps

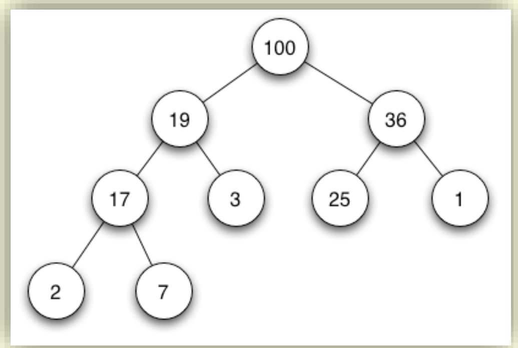
- Heap:
  - In a heap, all values stored in the subtree of a given node <u>must be</u> less than or equal to the value stored in that node
    - This is known as the **<u>heap property</u>**



And it is this property that makes a heap a heap!

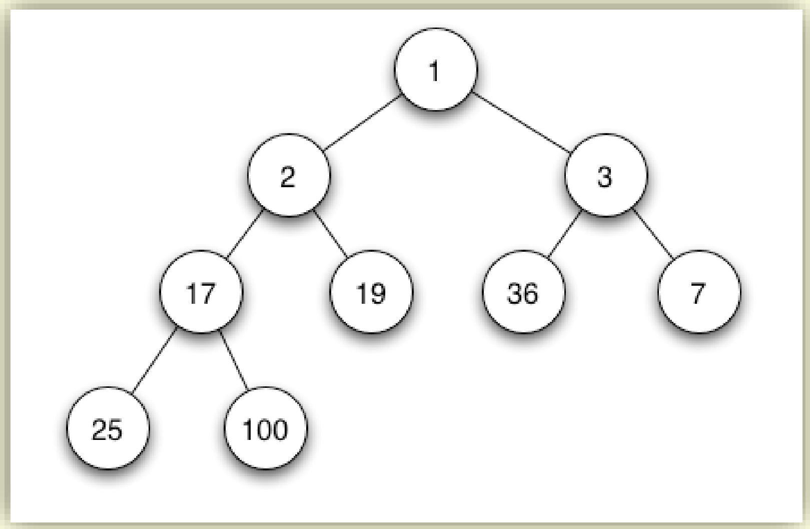
Binary Heaps

- Heap:
  - In a heap, all values stored in the subtree of a given node must be less than or equal to the value stored in that node
    - If B is a child of node A, then the value of node A must be greater than or equal to the value of node B
      - This is a called a **Max-Heap**
        - Where the root stores the highest value of any given subtree



Binary Heaps

- Heap:
  - Alternatively, if all values stored in the subtree of a given node are greater than or equal to the value stored in that node
    - This is called a **Min-Heap** (where root is smallest value)

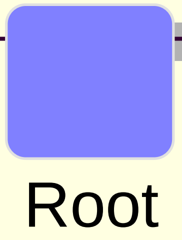


Binary Heaps

- Binary Heap:
  - What we just described was a basic Heap
  - Now for a heap to be <u>Binary Heap</u>, it must adhere to one other property:
  - The **<u>Shape Property</u>**:
    - The heap must be a <u>complete binary tree</u>
    - Meaning, all levels of the tree, except possibly the last one, must be fully filled
    - And if the last level is not complete, the nodes of the level are filled from left to right
      - ***And it just so happens that the previous pictures shown were all examples of binary heaps

Binary Heaps

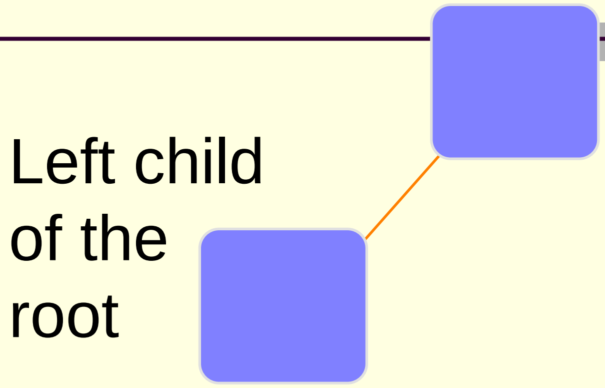
- Building a Complete Binary Tree:



When a complete binary tree is built, its first node must be the root.

Binary Heaps

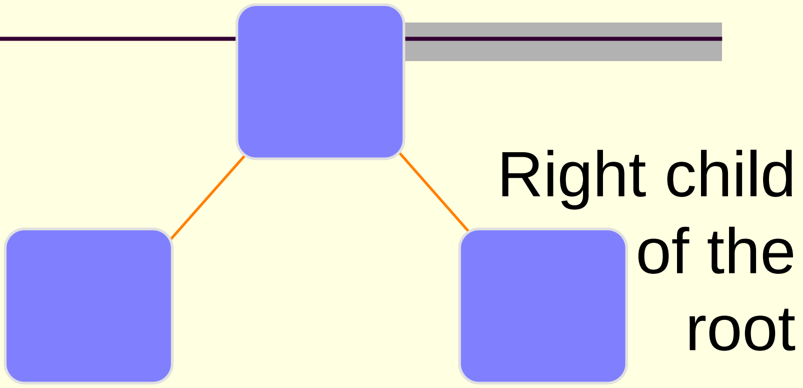
- Building a Complete Binary Tree:



The second node is always the left child of the root.

Binary Heaps

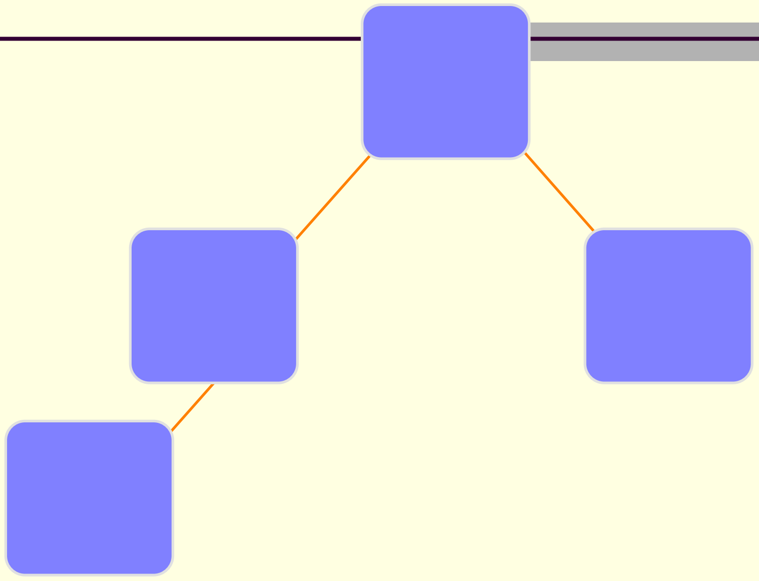
- Building a Complete Binary Tree:



The third node is always the right child of the root.

Binary Heaps

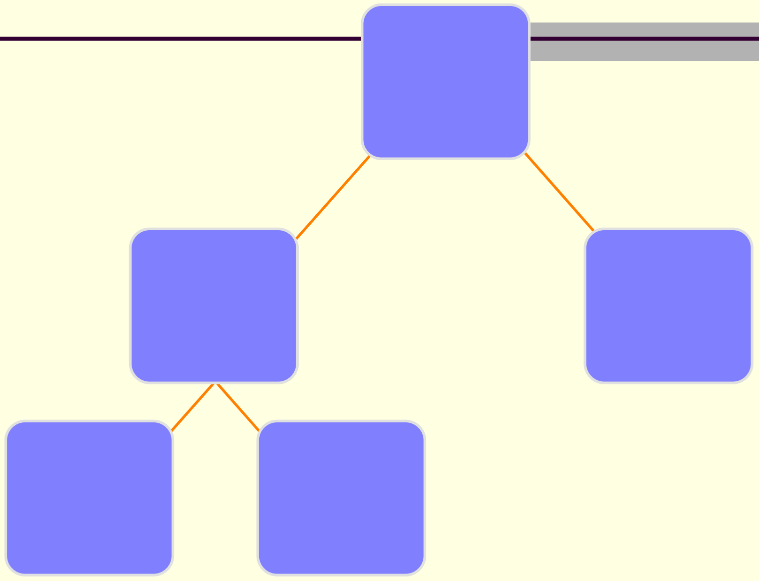
- Building a Complete Binary Tree:



The next nodes always fill the next level from left-to-right.

Binary Heaps

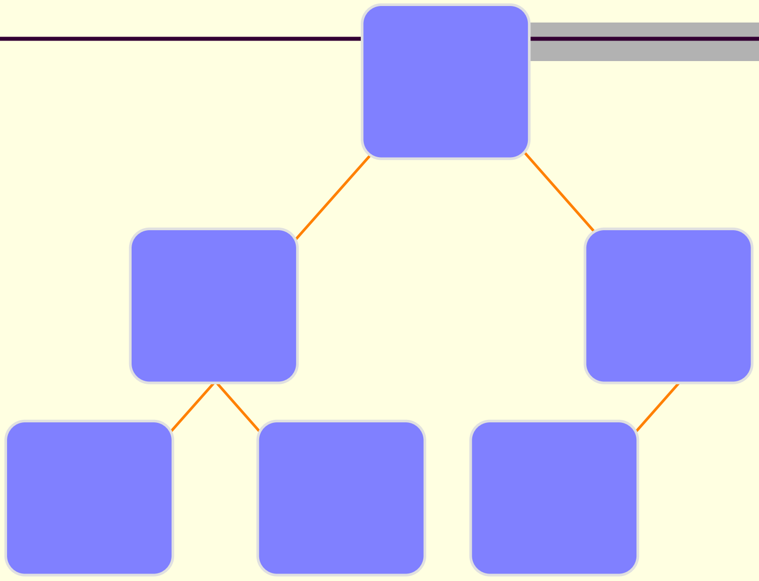
- Building a Complete Binary Tree:



The next nodes always fill the next level from left-to-right.

Binary Heaps

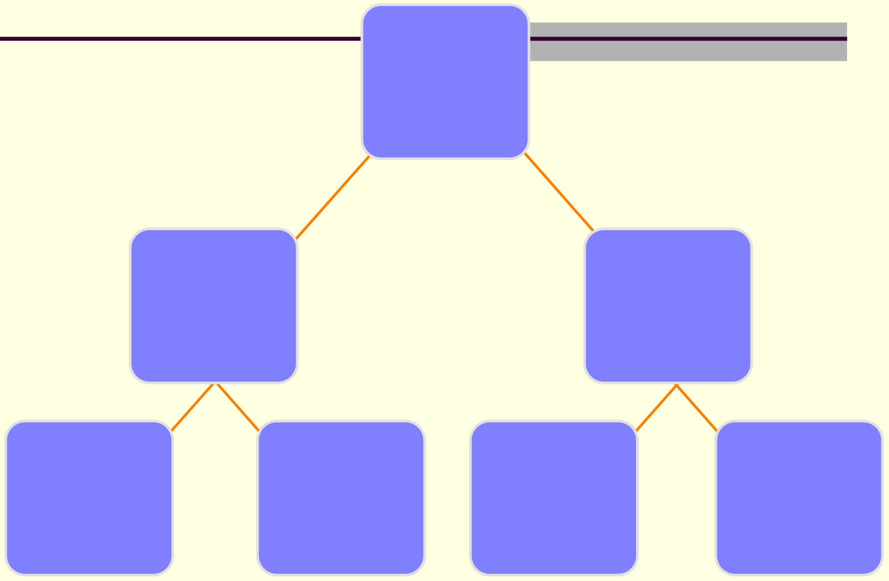
- Building a Complete Binary Tree:



The next nodes always fill the next level from left-to-right.

Binary Heaps

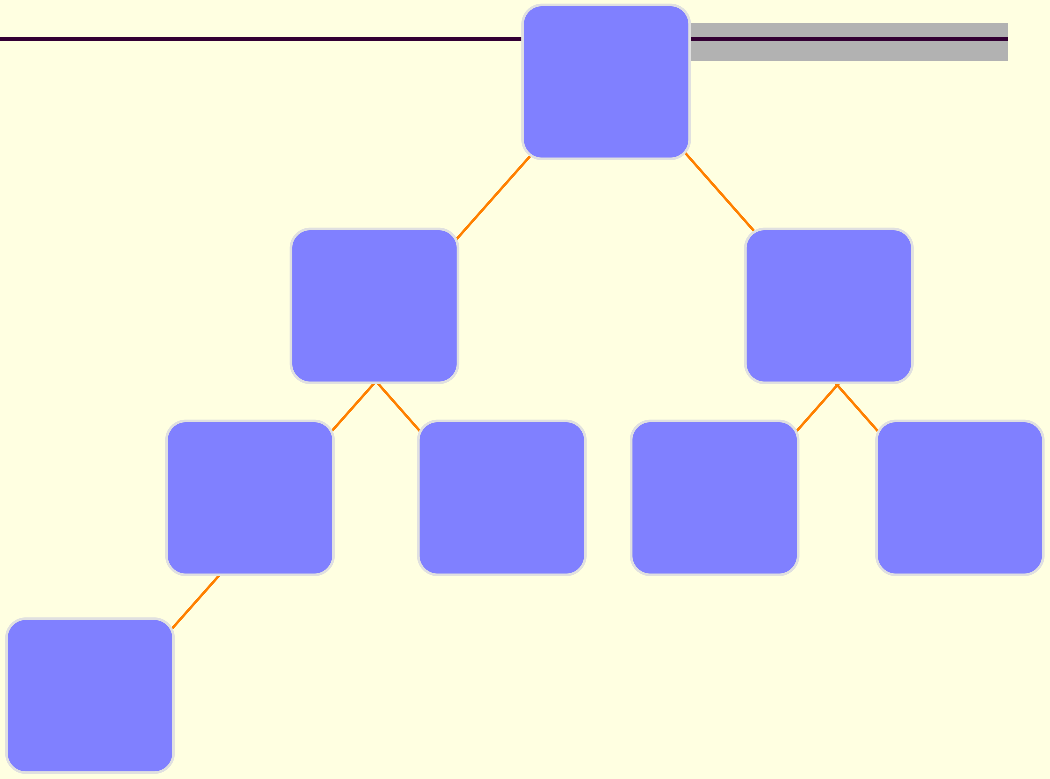
- Building a Complete Binary Tree:



The next nodes always fill the next level from left-to-right.

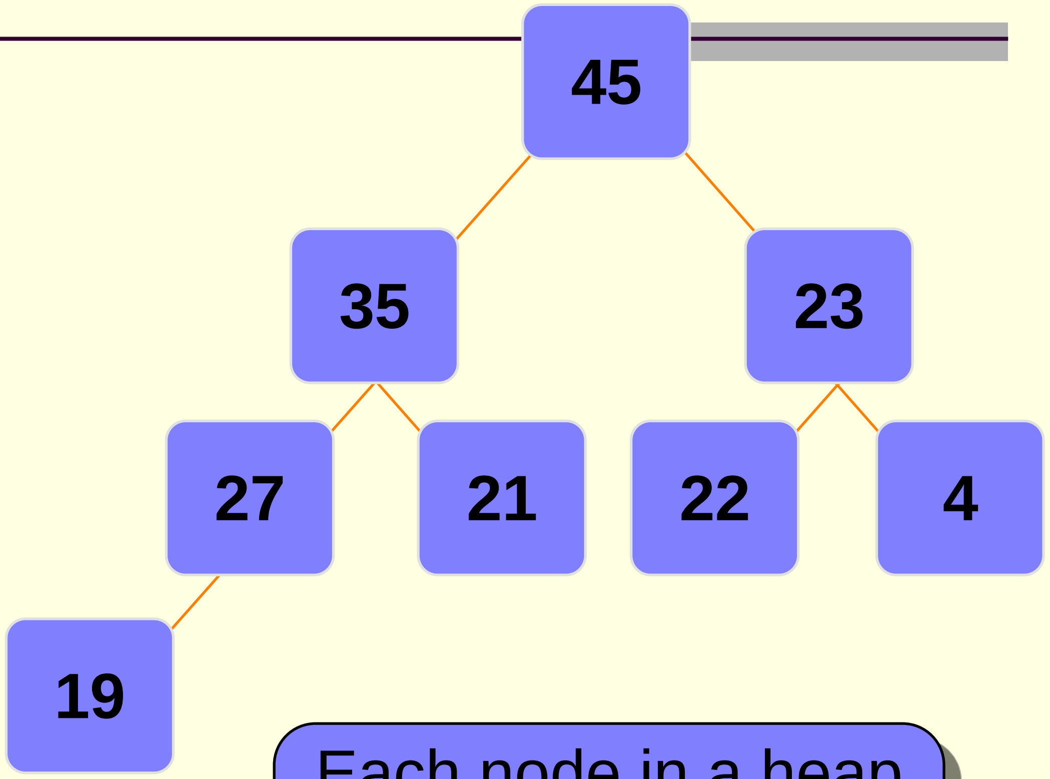
Binary Heaps

- Building a Complete Binary Tree:



Binary Heaps

- Building a Complete Binary Tree:



Each node in a heap contains a key that can be compared to other nodes' keys.

This is an example of a **<u>MaxHeap</u>**

Binary Heaps

- Binary Heap:
  - New nodes are always added at the lowest level
    - And are inserted from left to right
  - There is no particular relationship among the data items in nodes on any given level
    - Even if the nodes have the same parent
    - Example: the right node does not necessarily have to be larger than the left node (as in BSTs)
  - The only ordering property for heaps is the one already defined
    - Root of any given subtree is either largest or smallest element in that tree…either a max-heap or a min-heap

Binary Heaps

- Binary Heap:
  - The tree never becomes unbalanced
  - A heap is not a sorted structure
    - But it can be regarded as partially ordered
      - Since the minimum value is always at the root
  - A given set of data can be formed into many different heaps
    - Depending on the order in which the data arrives

Binary Heaps

- Binary Heap:
  - “Okay, great…whupdedoo”
  - Yeah, we now know what a binary heap is
  - But how does it help us?
  - What is its purpose?

  - Binary heaps are usually used to implement another abstract data type:
    - A **priority queue**

Binary Heaps

- Priority Queues:
  - A priority queue is basically what it sounds like
    - it is a queue
    - Which means that we will have a line
    - <u>But the first person in line is not necessarily the first person out of line</u>
    - Rather, the **<u>queuing order is based on a priority</u>**
    - Meaning, if one person has a higher priority, that person goes right to the front
  - Examples:
    - Emergency room:
      - Higher priority injuries are taken first

Binary Heaps

- Priority Queues:
  - The model:
    - Requests are inserted in the order of arrival
    - The <u>request with the highest priority is processed first</u>
      - Meaning, it is removed from the queue
    - Priority can be indicated by a number
      - But you have to determine what has most priority
      - Maybe your application results in smallest number having the highest priority
      - Maybe the largest number has the highest priority
        - This really isn’t important and is an implementation detail

Binary Heaps

- Priority Queues:
  - So how could we implement a priority queue?
    - Sorted Linked List
      - Higher priority items are ALWAYS at the front of the list
      - Example: a check out line in a supermarket
        - But people who are more important can cut in line
      - Running Time:
        - O(n) insertion time: you have to search through, potentially, n nodes to find the correct spot (based on priority)
        - O(1) deletion time (finding the node with the highest priority) since the highest priority node is first node of the list

Binary Heaps

- Priority Queues:
  - So how could we implement a priority queue?
    - Unsorted Linked List
      - Keep a list of elements as a queue
      - To add an element, append it to the end
      - To remove an element, search through all the elements for the one with the highest priority
      - Running Time:
        - O(1) insertion time: you simple add to the end of the list
        - O(n) deletion time: you have to, potentially, search through all n nodes to find the correct node to delete

Binary Heaps

- Priority Queues:
  - So how could we implement a priority queue?
    - **<u>Correct Method: Binary Heap!</u>**
    - We use a binary heap to implement a priority queue
      - So we are using one abstract data type to implement another abstract data type
    - Running time ends up being <u>O(logn) for both insertion and deletion into a Heap</u>
    - FindMin (finding the minimum) ends up being O(1)
      - cuz we just find (look at) the root, which is O(1)
    - So now we look at how to maintain a heap/priority queue
      - How to insert into and delete from a heap
      - And how to build a heap

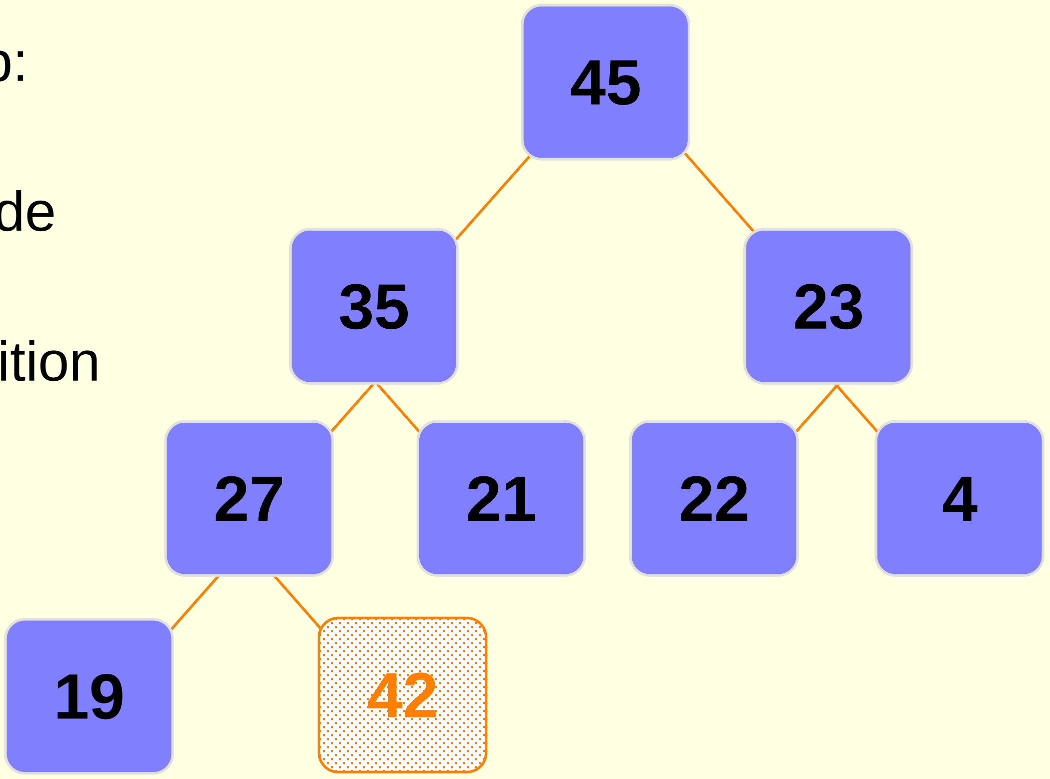
Binary Heaps

- Adding Nodes to a Binary Heap
  - Assume the existence of a current heap
  - Remember:
    - The binary heap MUST follow the Shape property
      - The tree must be balanced
  - Insertions will be made in the **next available spot**
    - Meaning, at the last level
    - and at the next spot, going from left to right
  - But what will most likely happen when you do this?
    - **The Heap property will NOT be maintained**

Binary Heaps

- Adding Nodes to a Binary Heap

- Given this Binary Heap:
  - And it is a Max-heap
- We now add a new node
  - With data value 42
- We add at the last position
- But this voids the Heap Property
  - 42 is greater than both 27 and 35
- So we must fix this!

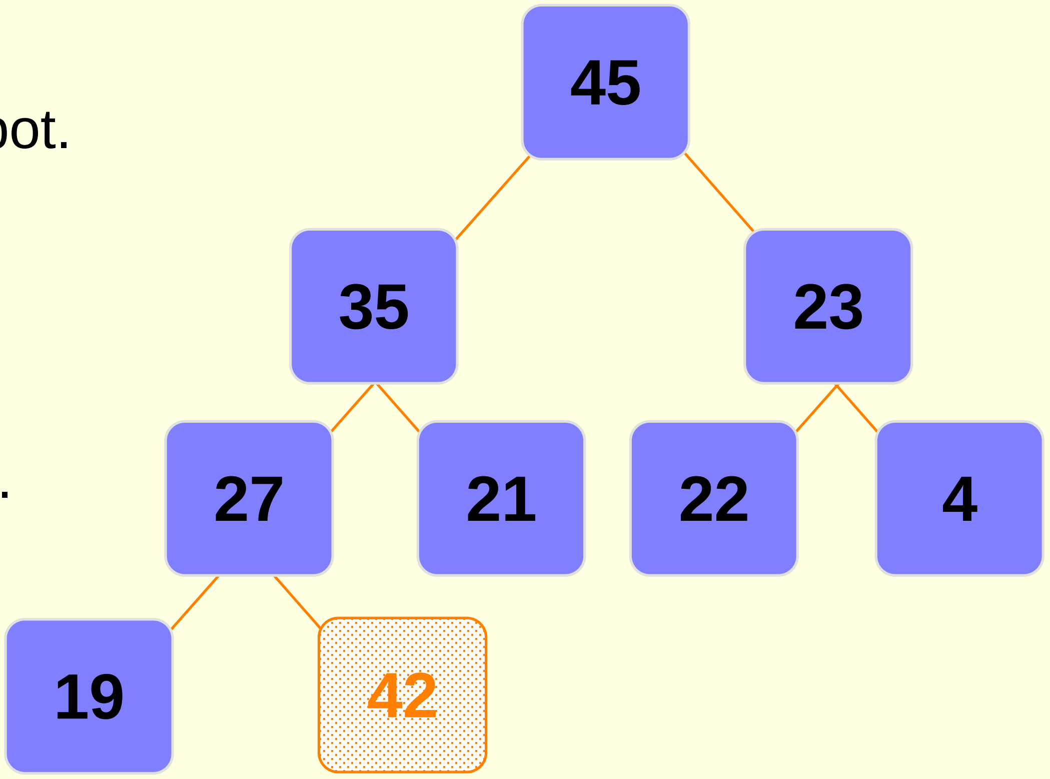


Binary Heaps

- Adding Nodes to a Binary Heap
  - **<u>Percolate Up</u>** procedure
    - In order to fix the out of place node, we must follow the following “Percolate Up” procedure
      - If the parent of the newly inserted node is less than the newly inserted node (this is clearly for a “max heap”)
        - Then SWAP them
      - This counts as one “Percolate Up” step
      - Continue this process until the new node finds the correct spot
        - Continue SWAPPING until the parent of the new node has a value that is greater than the new node
        - Or if the new node reaches all the way to the root
        - This is now the new “home” for this node

Binary Heaps

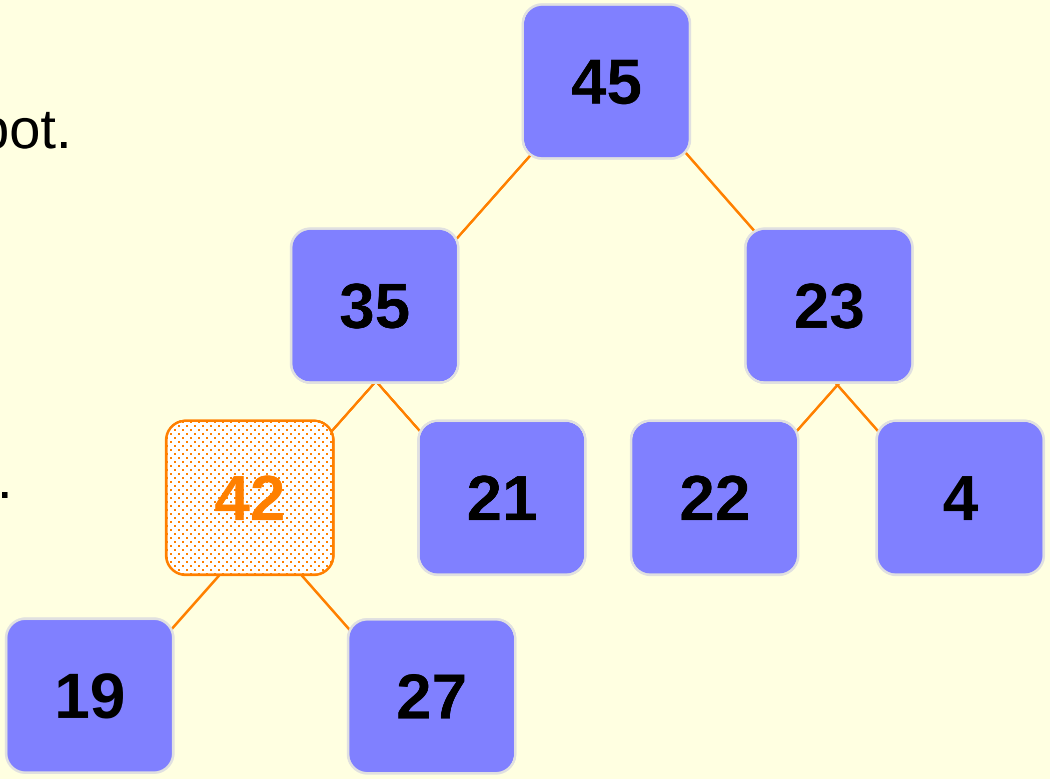
- Adding Nodes to a Binary Heap
  - Put the new node in the next available spot.
  - Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



Binary Heaps

- Adding Nodes to a Binary Heap

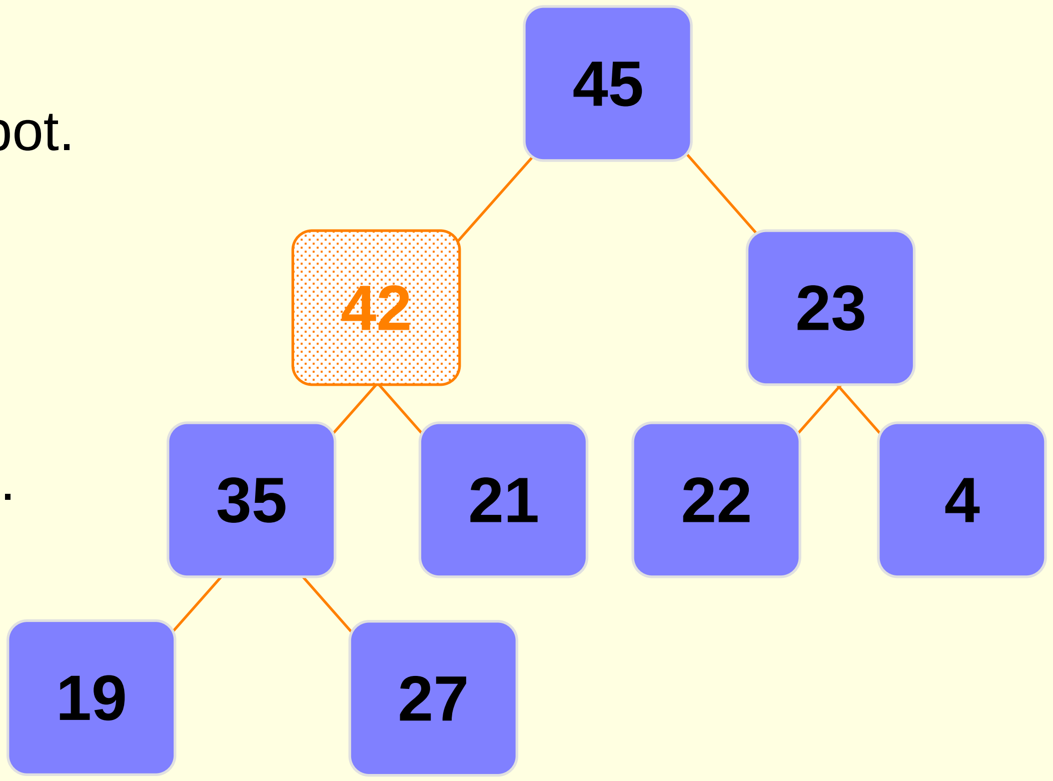
- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



Binary Heaps

- Adding Nodes to a Binary Heap

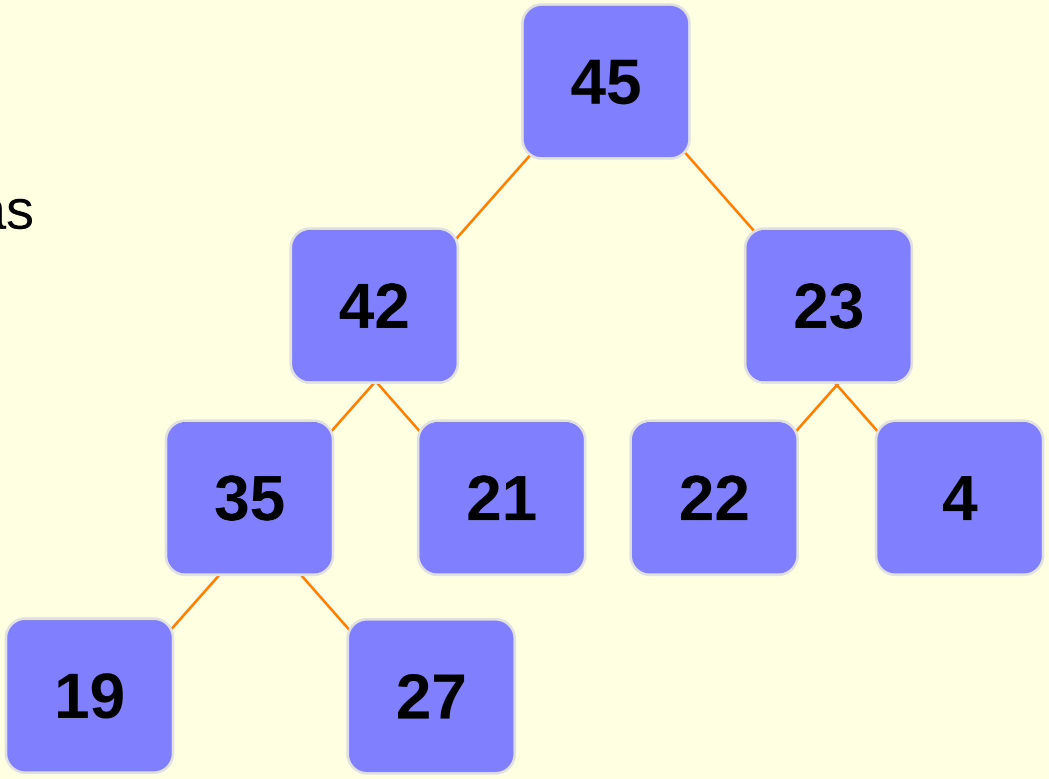
- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



Binary Heaps

- Adding Nodes to a Binary Heap

- 42 has now reached an acceptable location
- Its parent (node 45) has a value that is greater than 42
- This process is called <u>Percolate Up</u>
- Other books call it <u>Heapification Upward</u>
- What is important is how it works



Binary Heaps

- Adding Nodes to a Binary Heap
  - Percolate Up procedure
    - What is the Big-O running time of insertion into a heap?
    - The actual insertion is simply O(1)
      - We simply insert at the last position
      - <u>And you will see (in a bit) how we quick access to this position</u>
    - But when we do this,
      - We need to fix the tree to maintain the Heap Property
    - Percolate Up takes O(logn) time
      - Why?
      - Because the height of the tree is log n
      - Worst case scenario is having to SWAP all the way to the root
    - **<u>So the overall running time of an insertion is O(logn)</u>**

Binary Heaps

- Deleting Nodes from a Binary Heap
  - We will write a function called deleteMin (or deleteMax)
  - Which node will we ALWAYS be deleting?
  - Remember:
    - We are using a Heap to implement a priority queue!
      - And in a priority queue, <u>we always delete the first element</u>
      - The one with the highest priority
  - So we will ALWAYS be deleting the ROOT of the tree
    - So this is quite easy!
    - deleteMin (or deleteMax for a Max Heap) simply deletes the root and returns its value to main

Binary Heaps

- Deleting Nodes from a Binary Heap
  - We will write a function called deleteMin
    - deleteMin simply deletes the root and returns its value to main
  - But what will happen when we delete the root?
    - We will have a tree with no root!
    - The root will be missing
  - So clearly this needs to be fixed

Binary Heaps

This process is for a Max-heap

- Deleting Nodes from a Binary Heap
  - Fixing the tree after deleting the root:

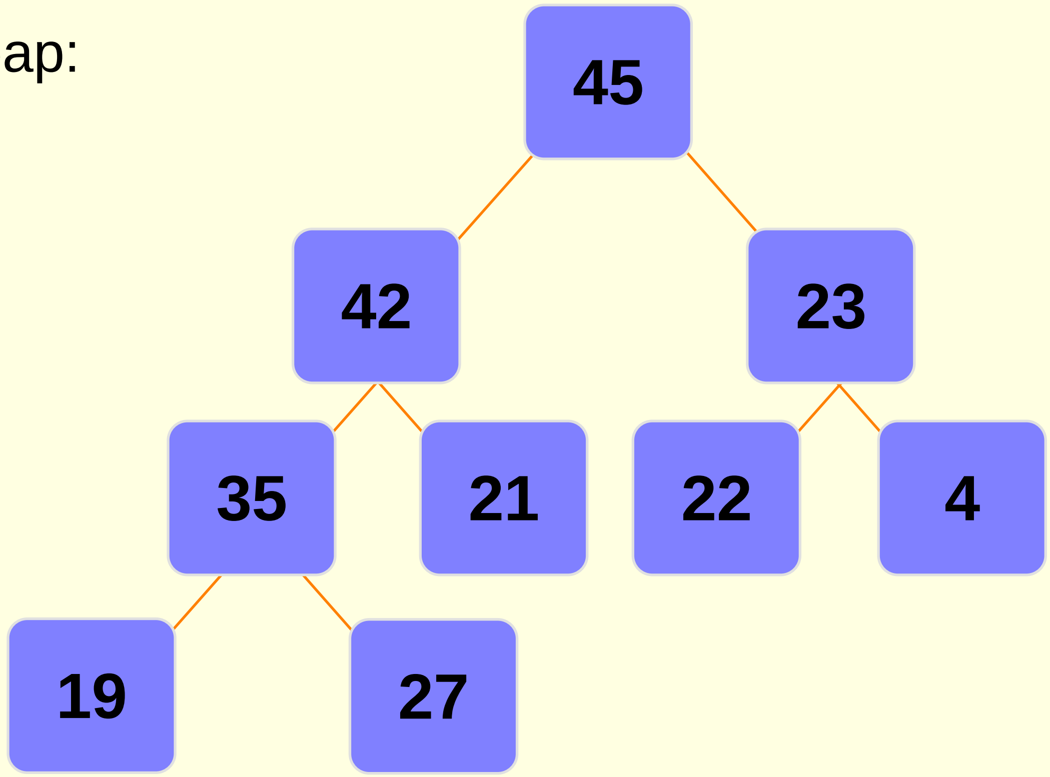
1) Copy the last node of the tree into the position of the root
2) Then remove that last node (to avoid duplicates)
   - Note: **<u>The new root is almost assuredly out of place</u>**
   - Most likely, one, or both, of its children will have a greater value than it
   - If so:
3) Swap the new root node with the **<u>greater</u>** of its child nodes
   - This is considered one "**<u>Percolate Down</u>**" step

- Continue this process until the "last node" ends up in a spot where its children have values smaller than it
  - Neither child can have a value greater than it

Binary Heaps

- Deleting Nodes from a Binary Heap

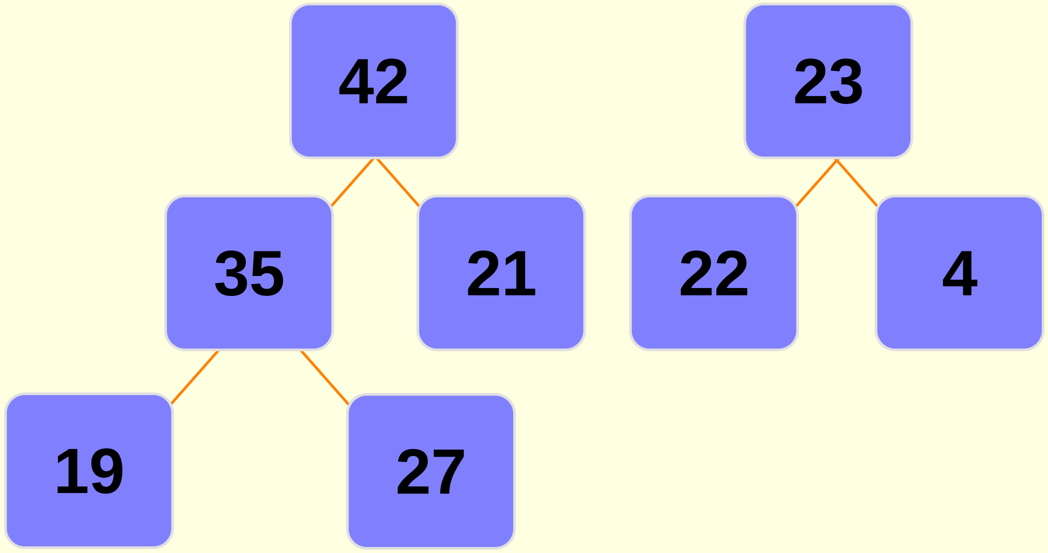
- Given the following Heap:
- We perform a delete
- Which means 45 will get deleted



Binary Heaps

- Deleting Nodes from a Binary Heap

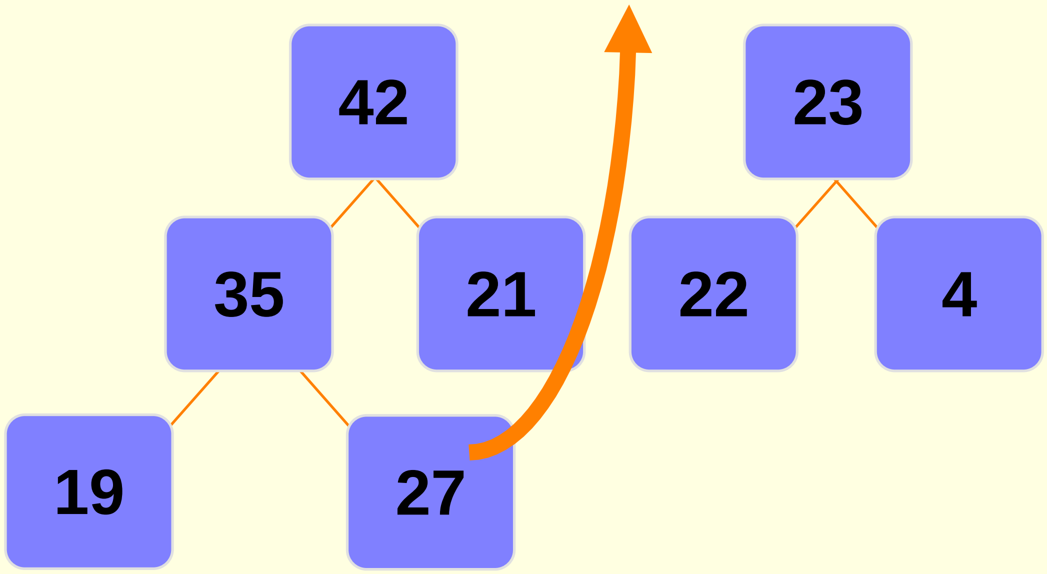
- Given the following Heap:
- We perform a delete
- Which means 45 will get deleted



Binary Heaps

- Deleting Nodes from a Binary Heap

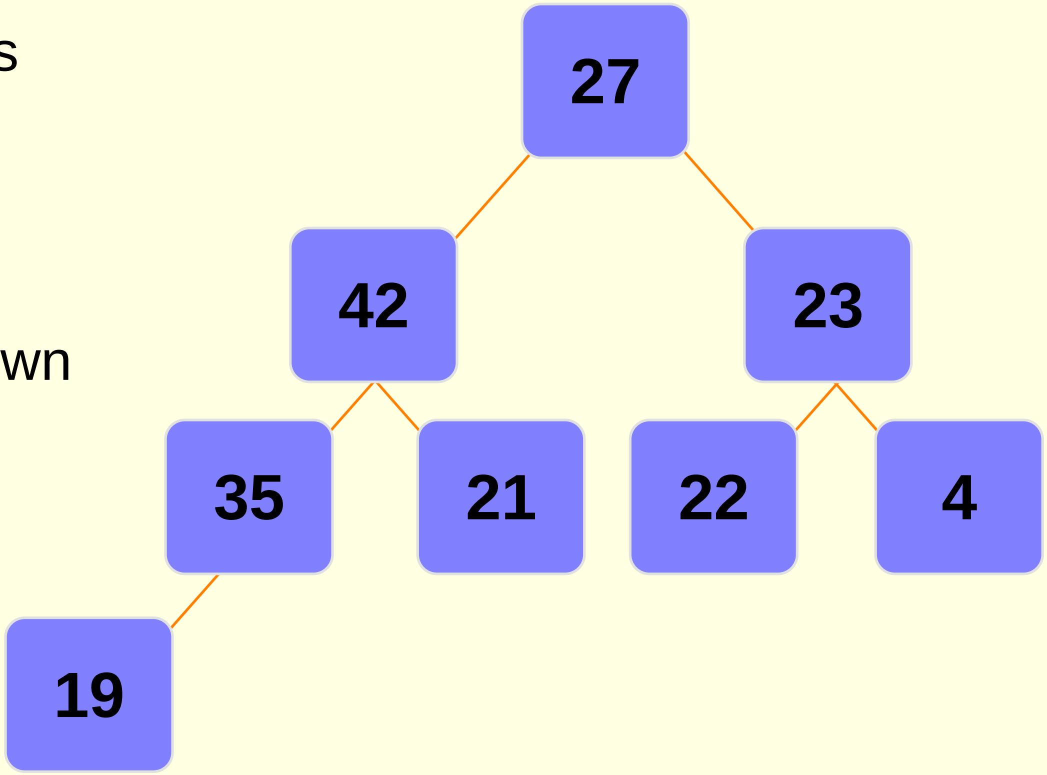
- The last node now gets moved to the root
- So 27 goes to the root



Binary Heaps

- Deleting Nodes from a Binary Heap

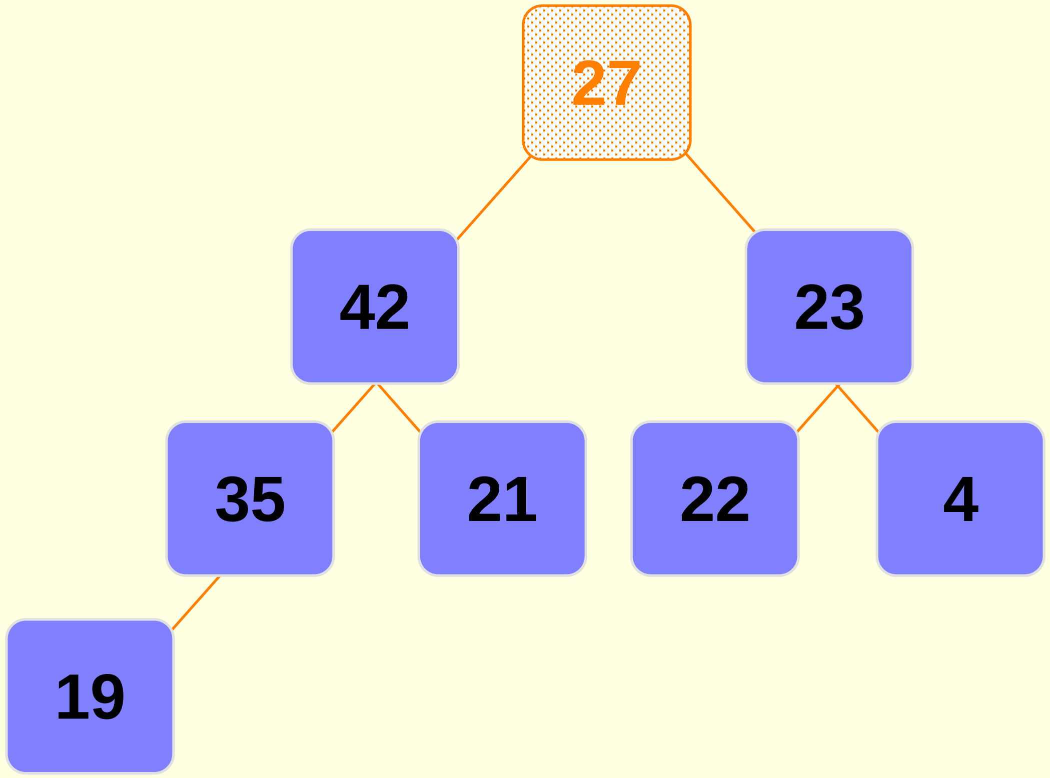
- The last node now gets moved to the root
- So 27 goes to the root
- 27 is now out of place
- We must Percolate Down



Binary Heaps

- Deleting Nodes from a Binary Heap

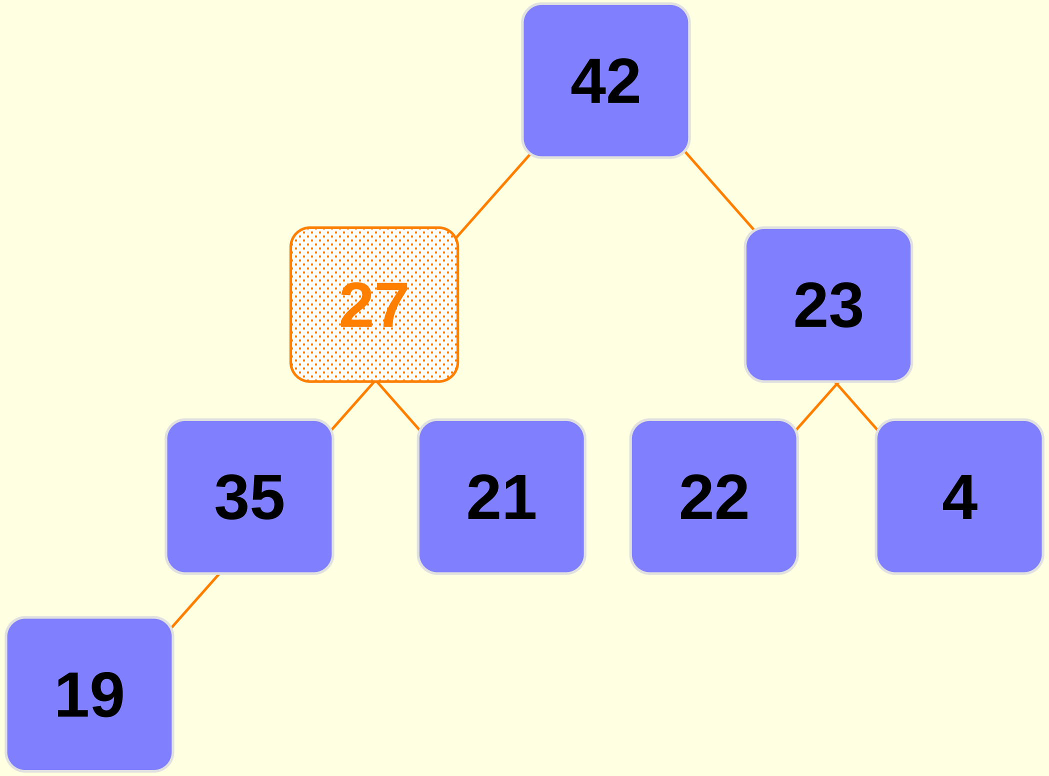
- **Percolate Down:**
- Push the out-of-place node downward,
  - swapping with its **larger** child
- until the out-of-place node reaches an acceptable location



Binary Heaps

- Deleting Nodes from a Binary Heap

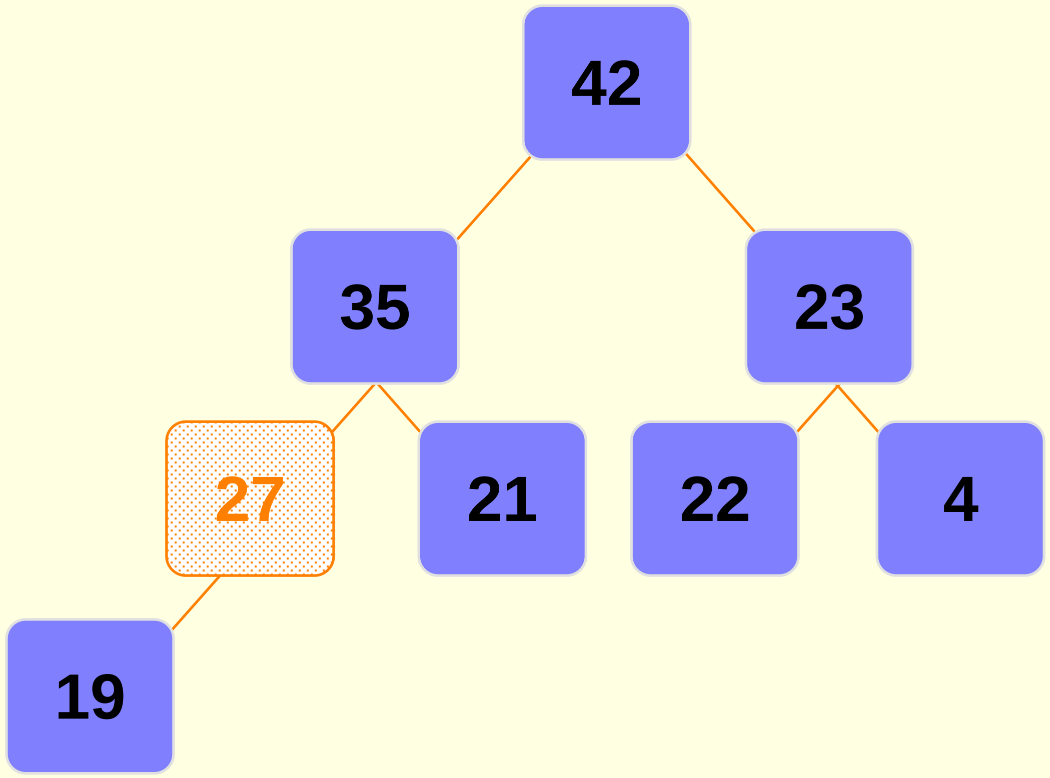
- **Percolate Down:**
- Push the out-of-place node downward,
  - swapping with its **larger** child
- until the out-of-place node reaches an acceptable location



Binary Heaps

- Deleting Nodes from a Binary Heap

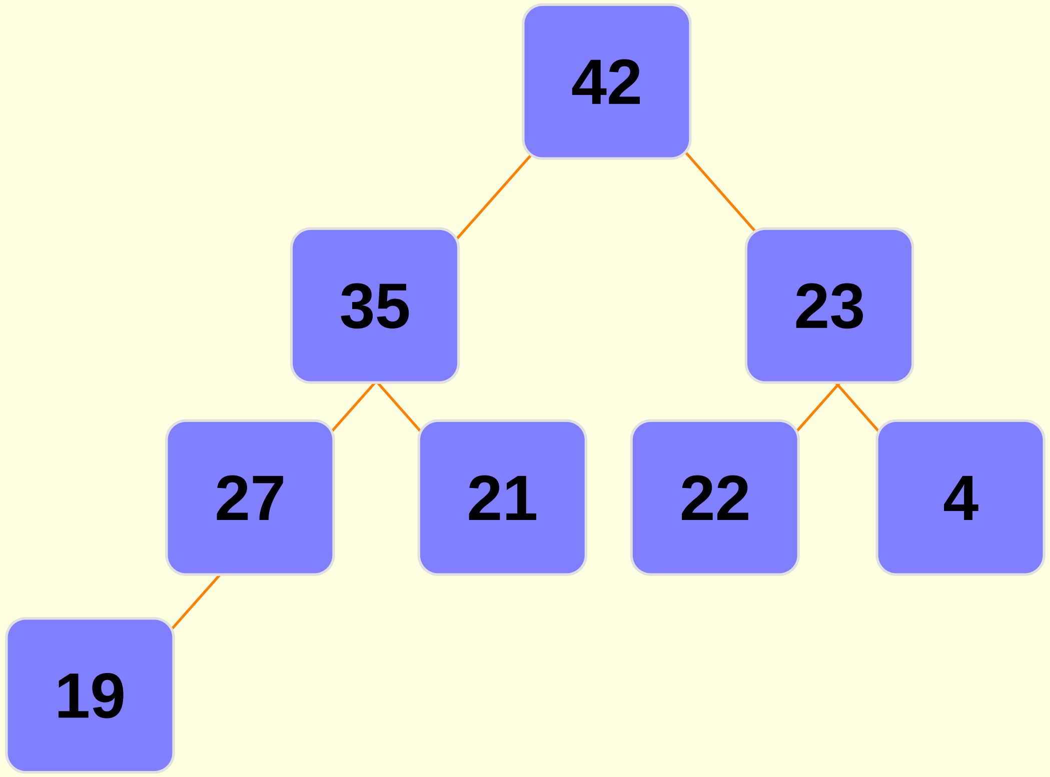
- **Percolate Down:**
- Push the out-of-place node downward,
  - swapping with its **larger** child
- until the out-of-place node reaches an acceptable location



Binary Heaps

- Deleting Nodes from a Binary Heap

- **<u>Percolate Down:</u>**
- 27 has reached an acceptable location
- Its lone child (19) has a value that is less than 27
- So we stop the Percolate Down procedure at this point



Binary Heaps

- Deleting Nodes from a Binary Heap
  - What is the Big-O running time of deletion from a heap?
  - The actual deletion itself is O(1)
    - cause the minimum value is at the root
      - and we can delete the root of a tree in O(1) time
  - But now we need to fix the tree
    - Moving the last node to the root is an O(1) step
    - But then we need to Percolate Down
  - Percolate Down takes O(logn)
    - Why?
      - Because the height of the tree is log n
      - And the worst case scenario is having to SWAP all the way to the farthest leaf
  - **So the overall running time of a deletion is O(logn)**