Binary Heaps & Priority Queues

Computer Science Department
University of Central Florida

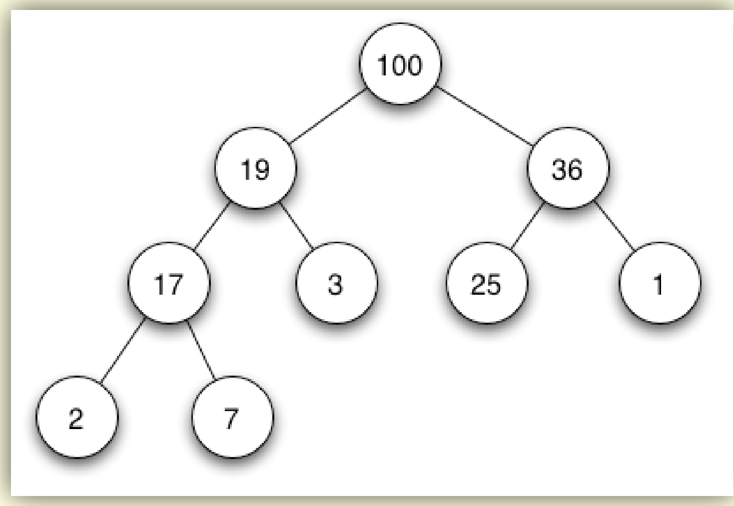
COP 3502 – Computer Science I

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

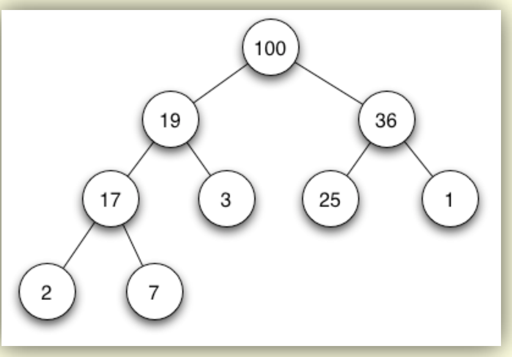
- Heap:
  - In a heap, all values stored in the subtree of a given node <u>must be</u> less than or equal to the value stored in that node
    - This is known as the **<u>heap property</u>**



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

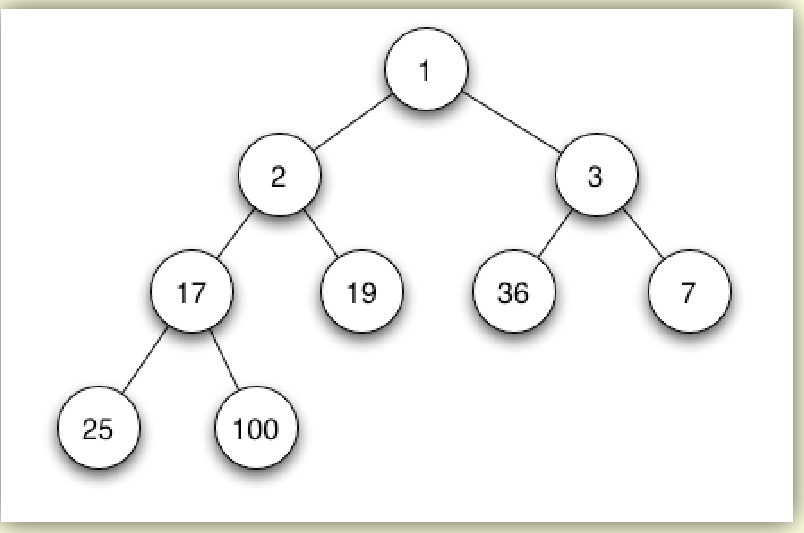
Binary Heaps

- 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)

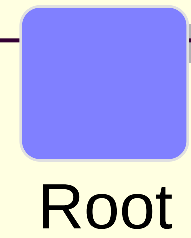


Binary Heaps

- 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

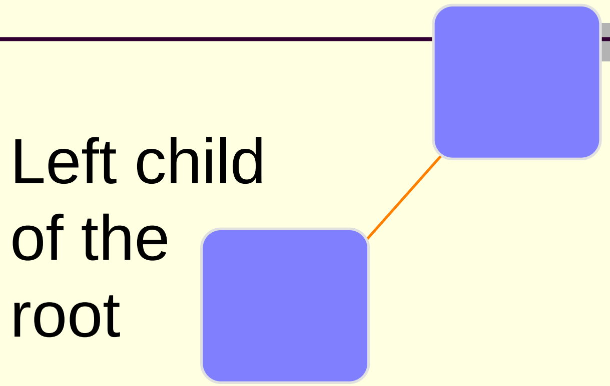
- Building a Complete Binary Tree:



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

Binary Heaps

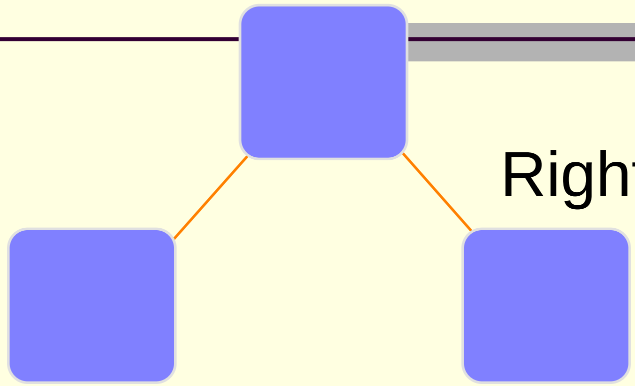
- Building a Complete Binary Tree:



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

Binary Heaps

- Building a Complete Binary Tree:

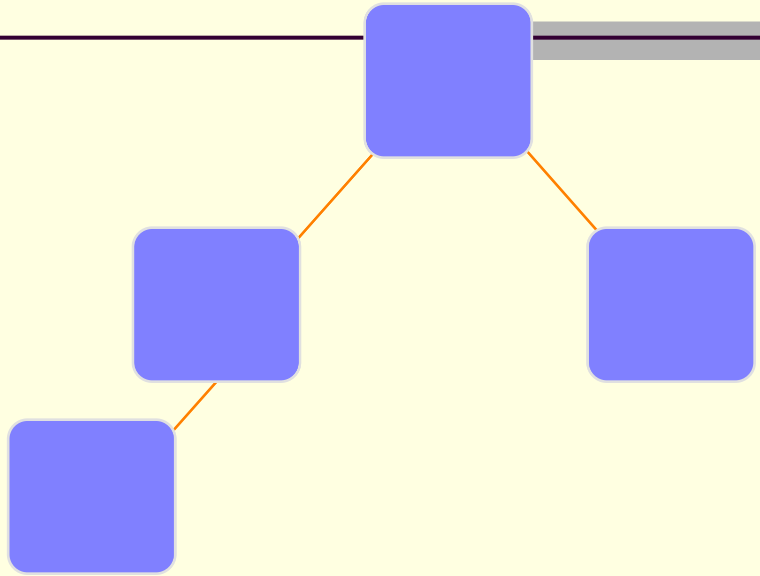


Right child of the root

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

Binary Heaps

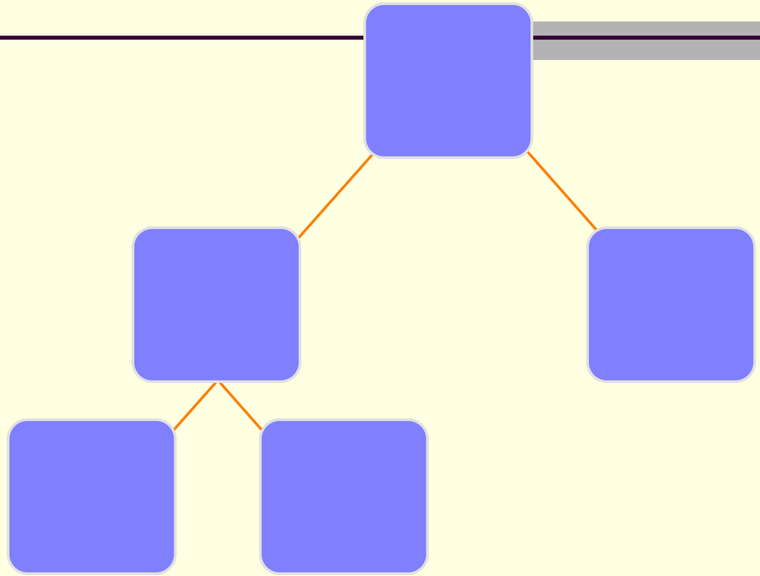
- Building a Complete Binary Tree:



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

Binary Heaps

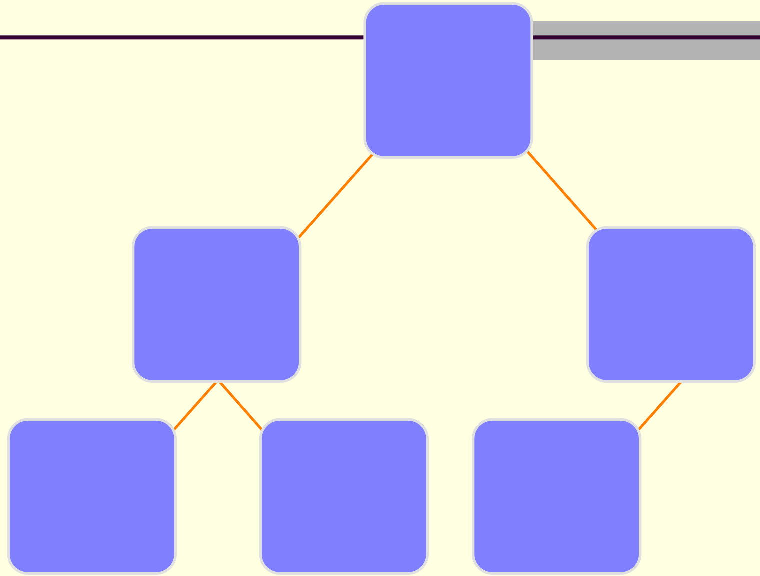
- Building a Complete Binary Tree:



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

Binary Heaps

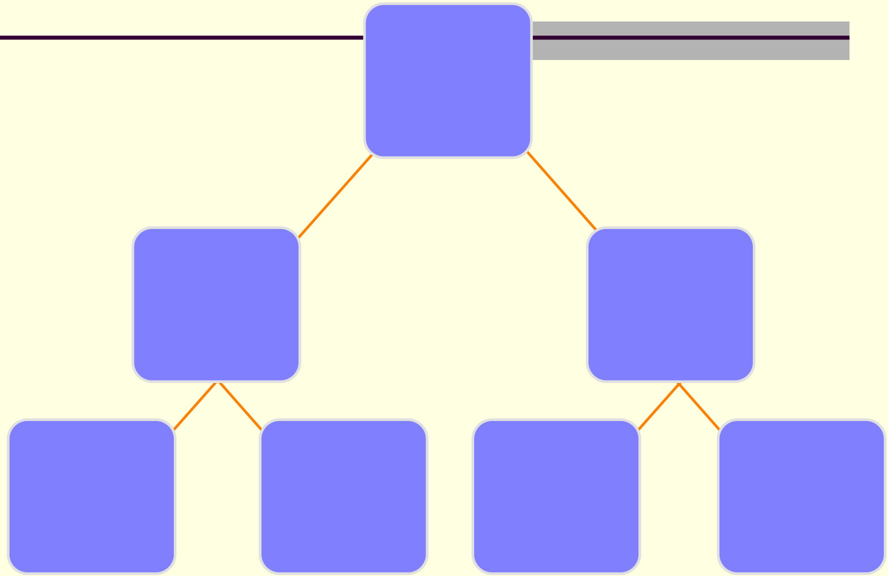
- Building a Complete Binary Tree:



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

Binary Heaps

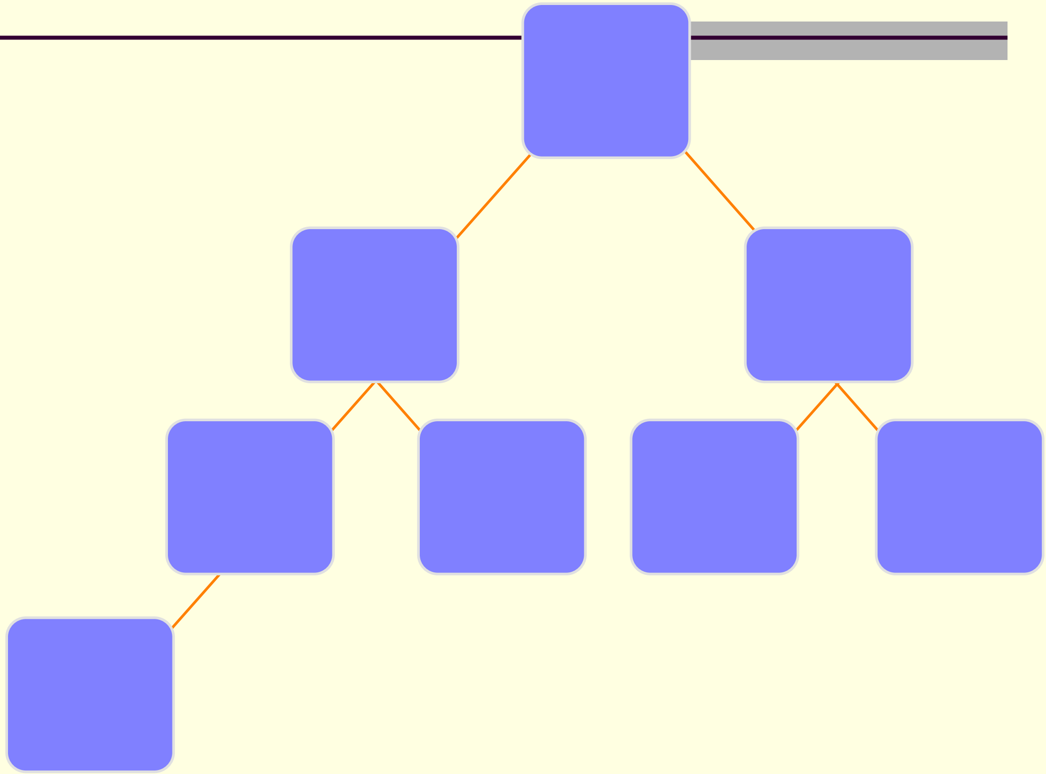
- Building a Complete Binary Tree:



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

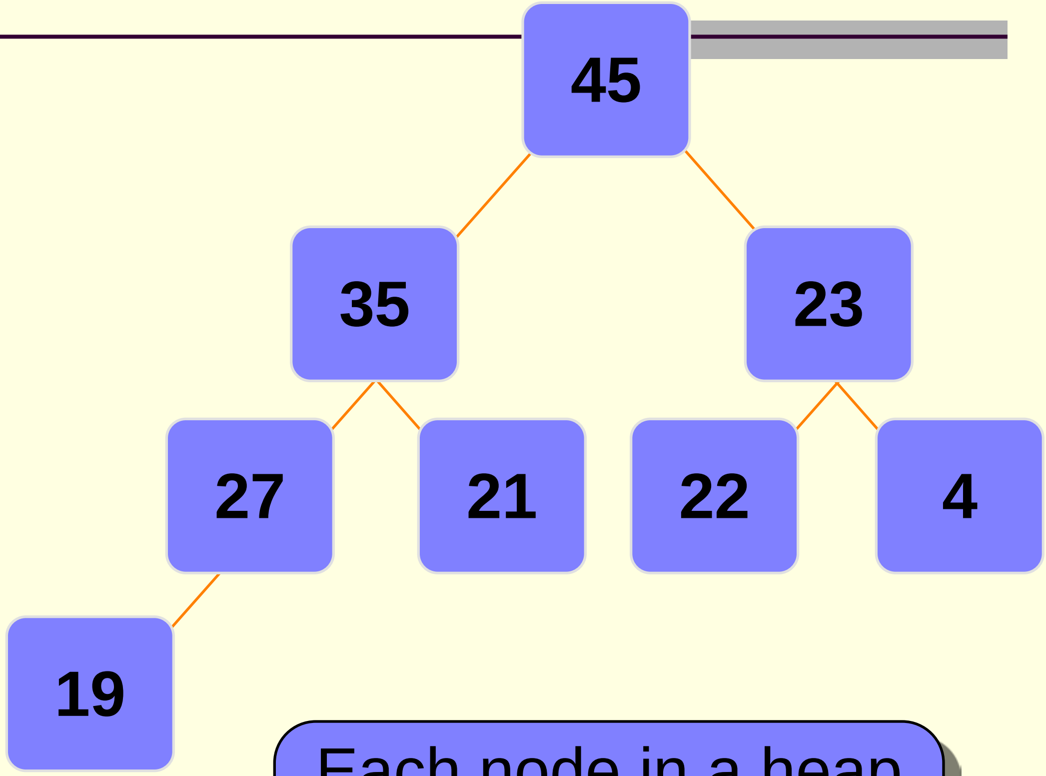
Binary Heaps

- Building a Complete Binary Tree:



Binary Heaps

- Building a Complete Binary Tree:



This is an example of a **MaxHeap**

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

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 request with the highest priority is processed first
      - 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?
    - **Correct Method: Binary Heap!**
    - 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 O(logn) for both insertion and deletion into a Heap
    - FindMin (finding the minimum) ends up being 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

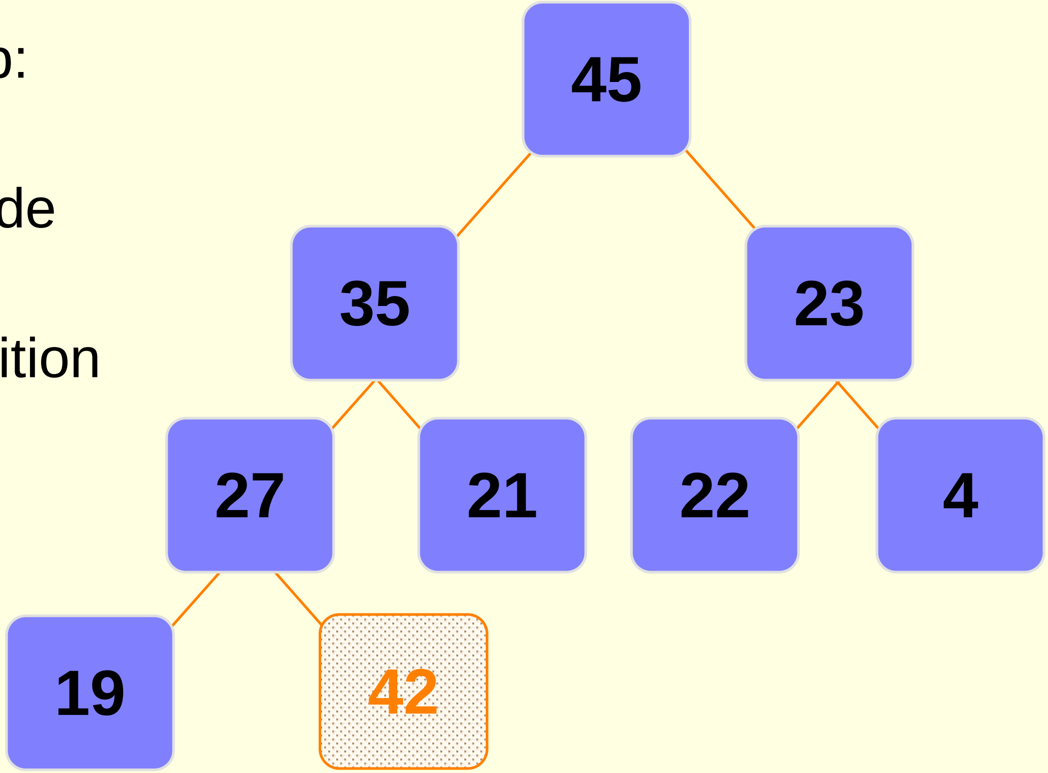
Binary Heaps

- 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!

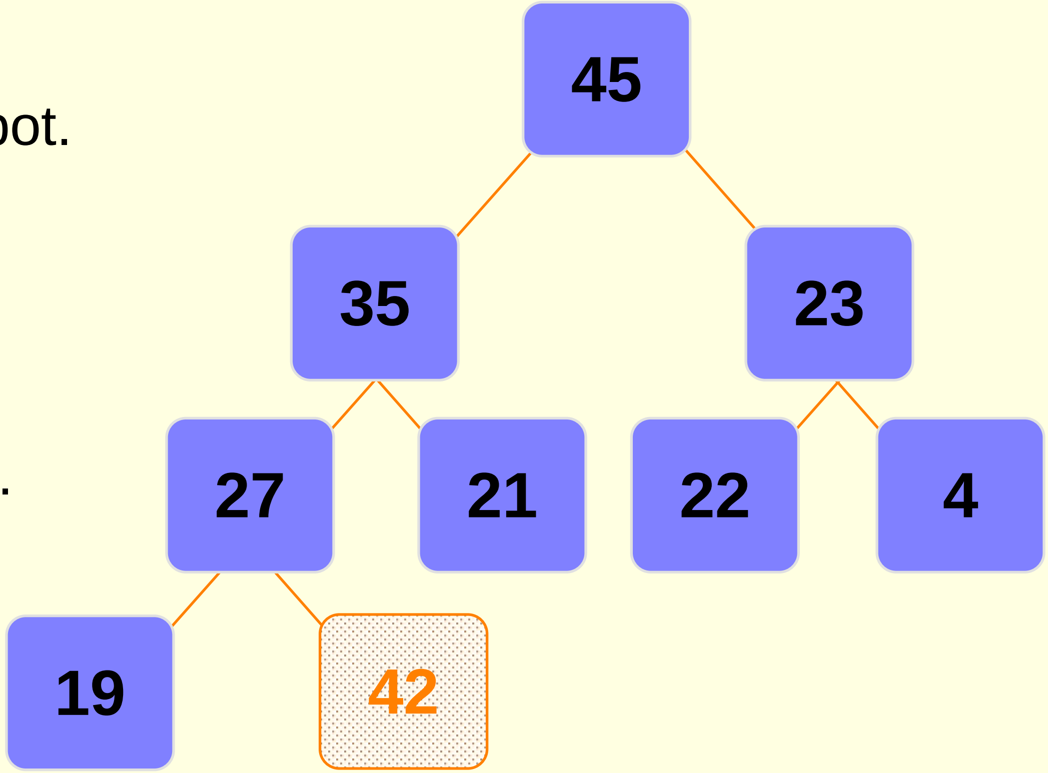


Binary Heaps

- Adding Nodes to a Binary Heap
  - Percolate Up 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
        - 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

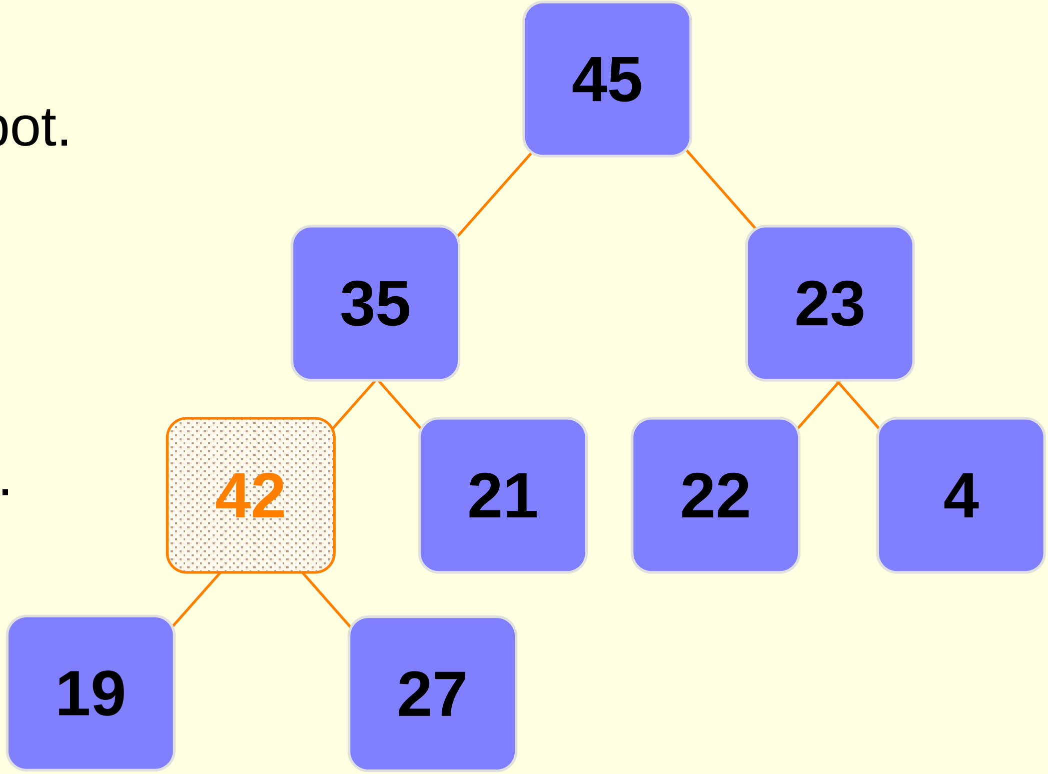
Binary Heaps

- 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.



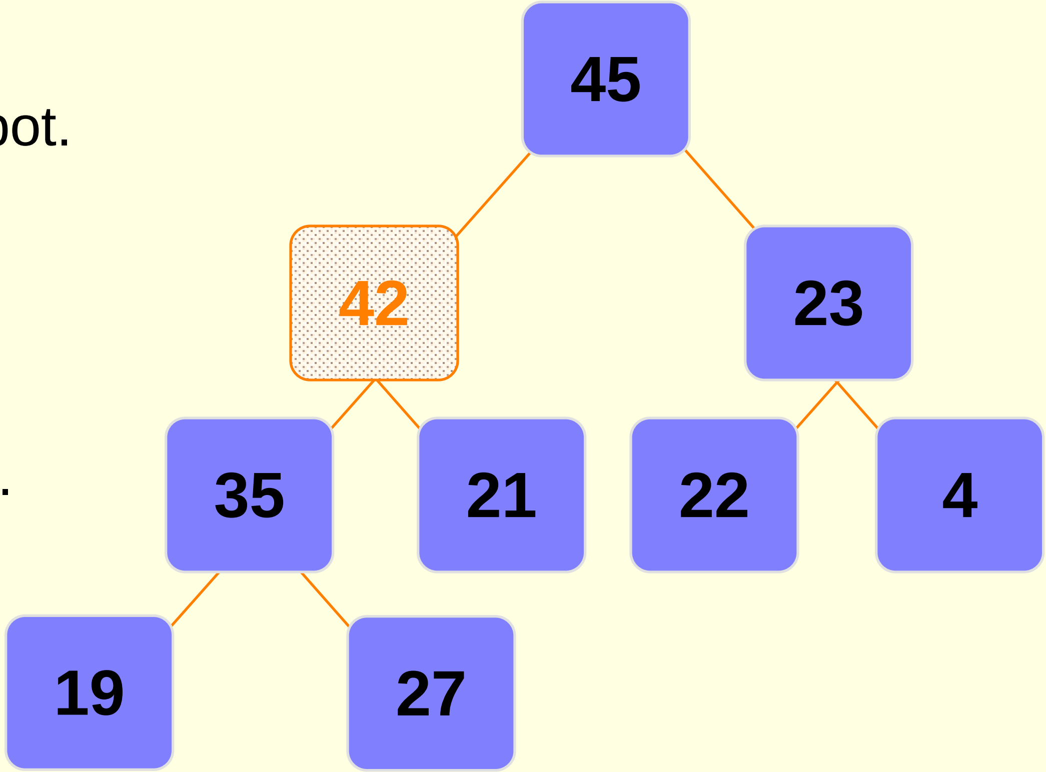
Binary Heaps

- 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

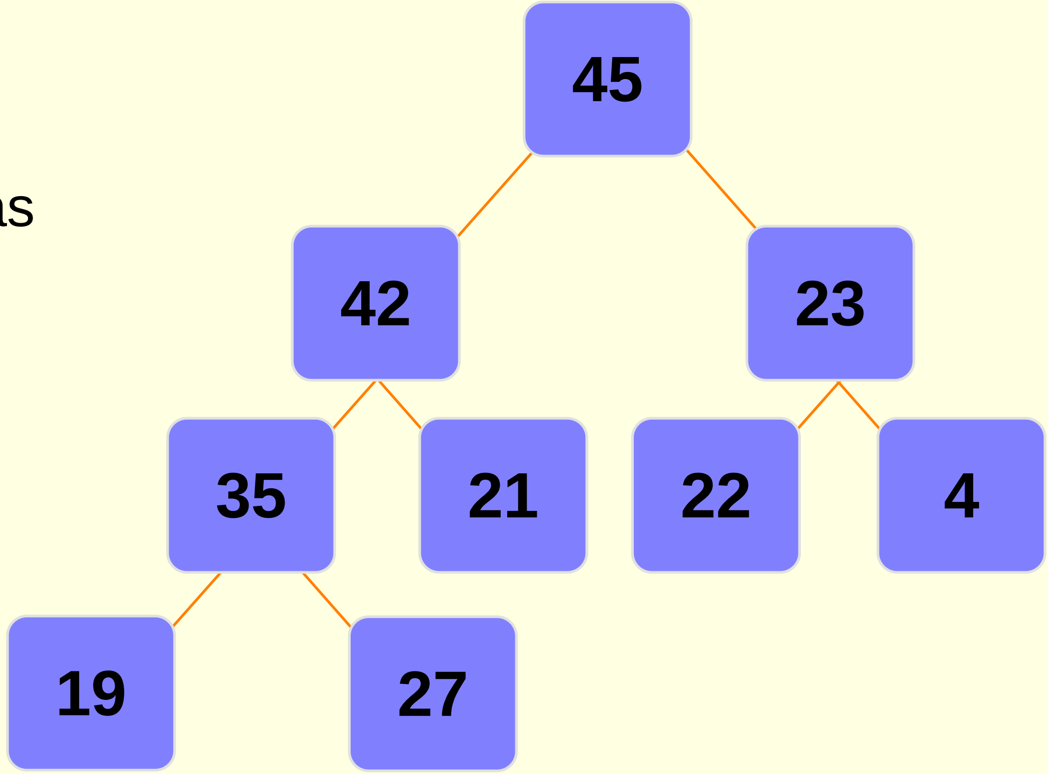
- Adding Nodes to a Binary Heap
  - Put the new node in the next available spot.
  - Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



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 Percolate Up
  - Other books call it Heapification Upward
  - 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?
    - Inserting the element is simply O(1)
      - We simply insert at the last position
      - And you will see (in a bit) how we quick access to this position
    - 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
    - So the overall running time of an insertion is O(logn)

Binary Heaps

- Deleting Nodes from a Binary Heap
  - We will write a function called deleteMin
  - Which node will we ALWAYS be deleting?
  - Remember:
    - We are using a Heap to implement a priority queue!
      - And in a priority queue, we always delete the first element
      - The one with the highest priority
  - So we will ALWAYS be deleting the ROOT of the tree
    - So this is quite easy!
    - deleteMin 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:

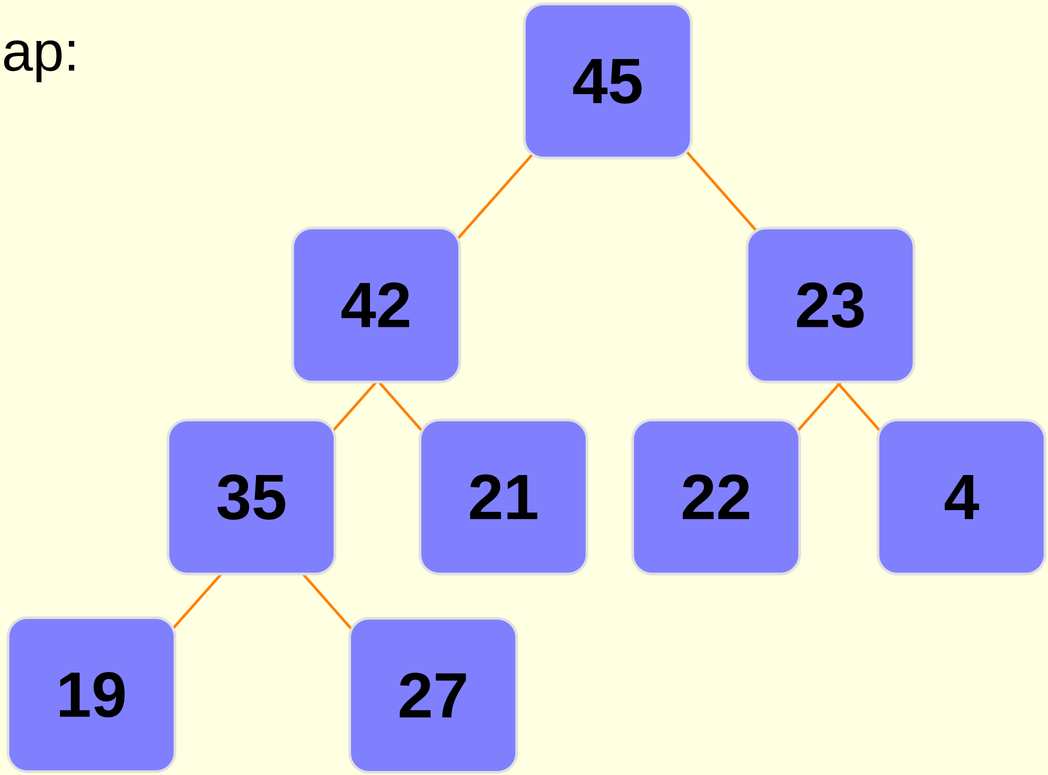
1) Copy the last node of the tree into the position of the root
2) Then remove that last node (to avoid duplicates)
   - Note: **<u>The new root is almost assuredly out of place</u>**
   - Most likely, one, or both, of its children will have a greater value than it
   - If so:
3) Swap the new root node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

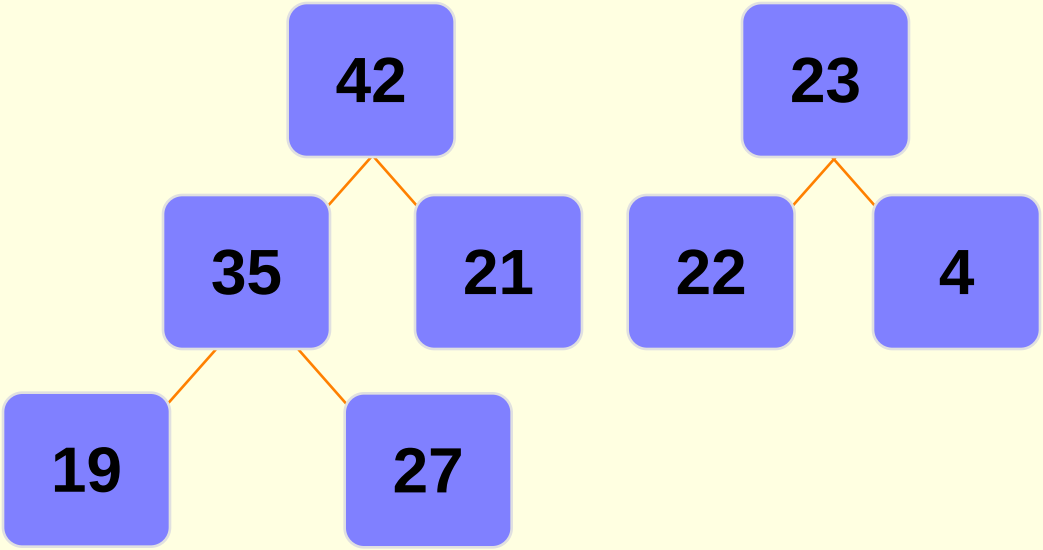
- Given the following Heap:
- We perform a delete
- Which means 45 will get deleted



Binary Heaps

- Deleting Nodes from a Binary Heap

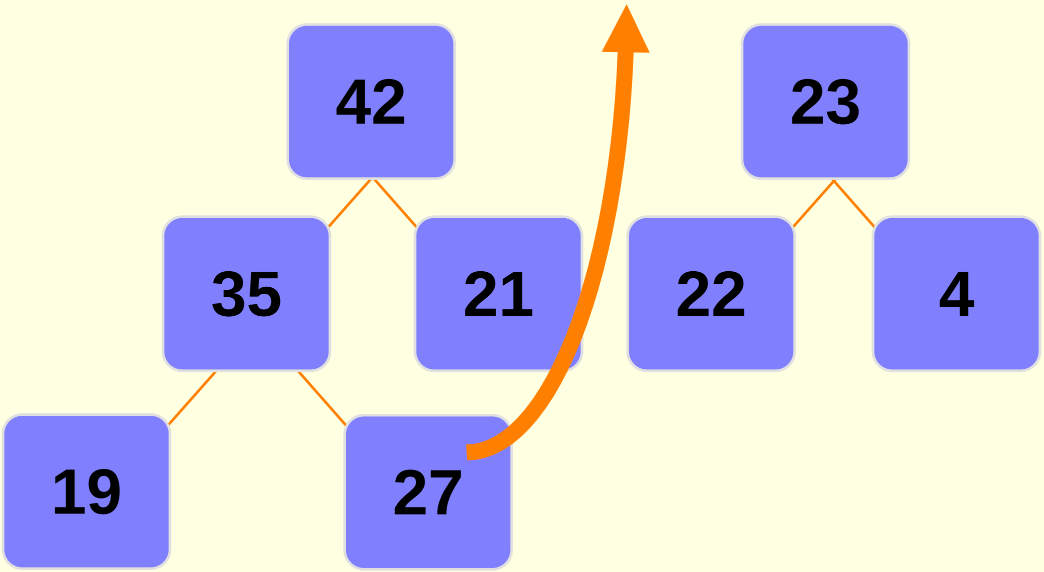
- Given the following Heap:
- We perform a delete
- Which means 45 will get deleted



Binary Heaps

- Deleting Nodes from a Binary Heap

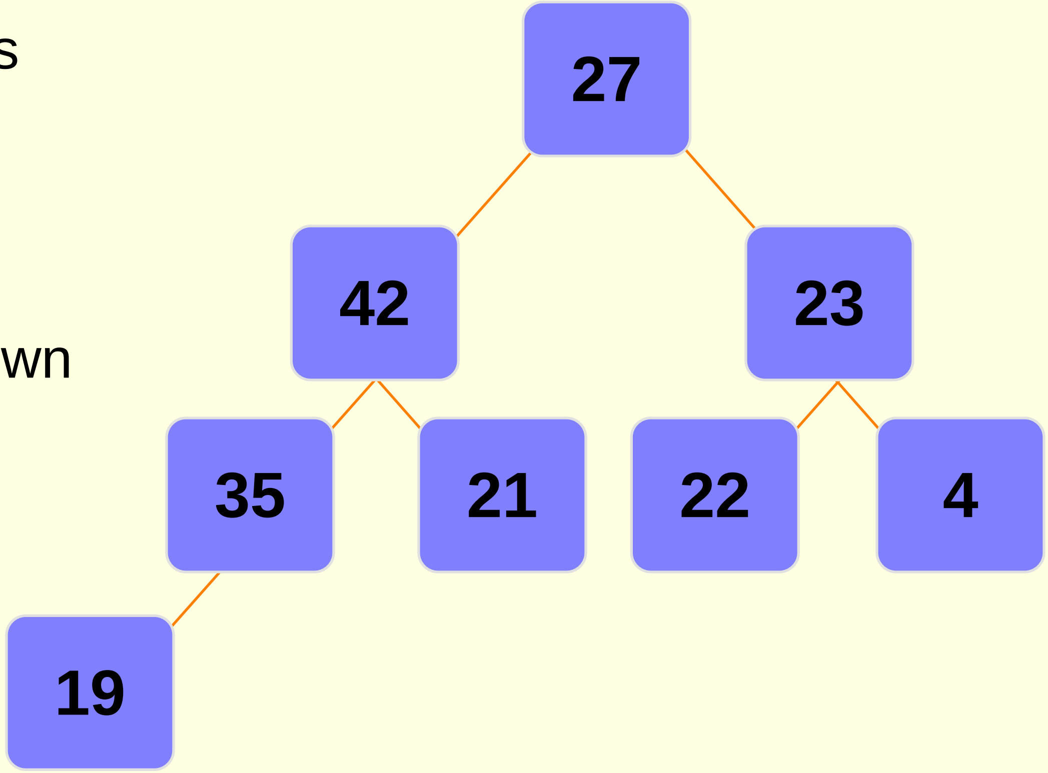
- The last node now gets moved to the root
- So 27 goes to the root



Binary Heaps

- Deleting Nodes from a Binary Heap

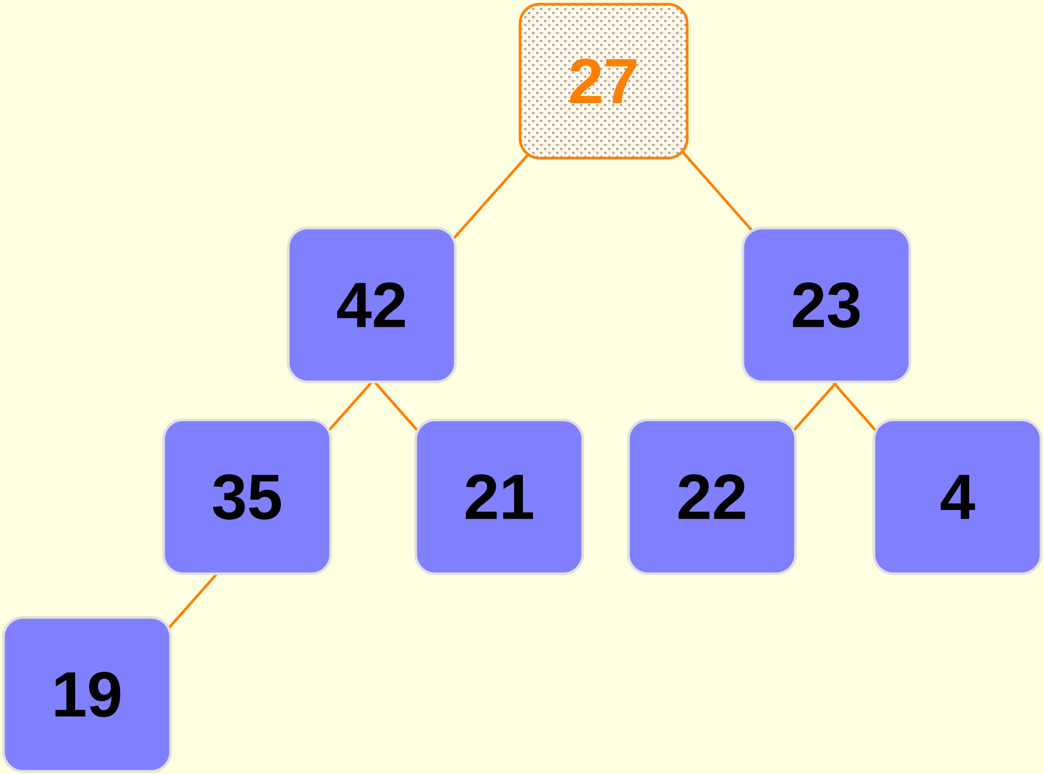
  - The last node now gets moved to the root
  - So 27 goes to the root
  - 27 is now out of place
  - We must Percolate Down



Binary Heaps

- Deleting Nodes from a Binary Heap

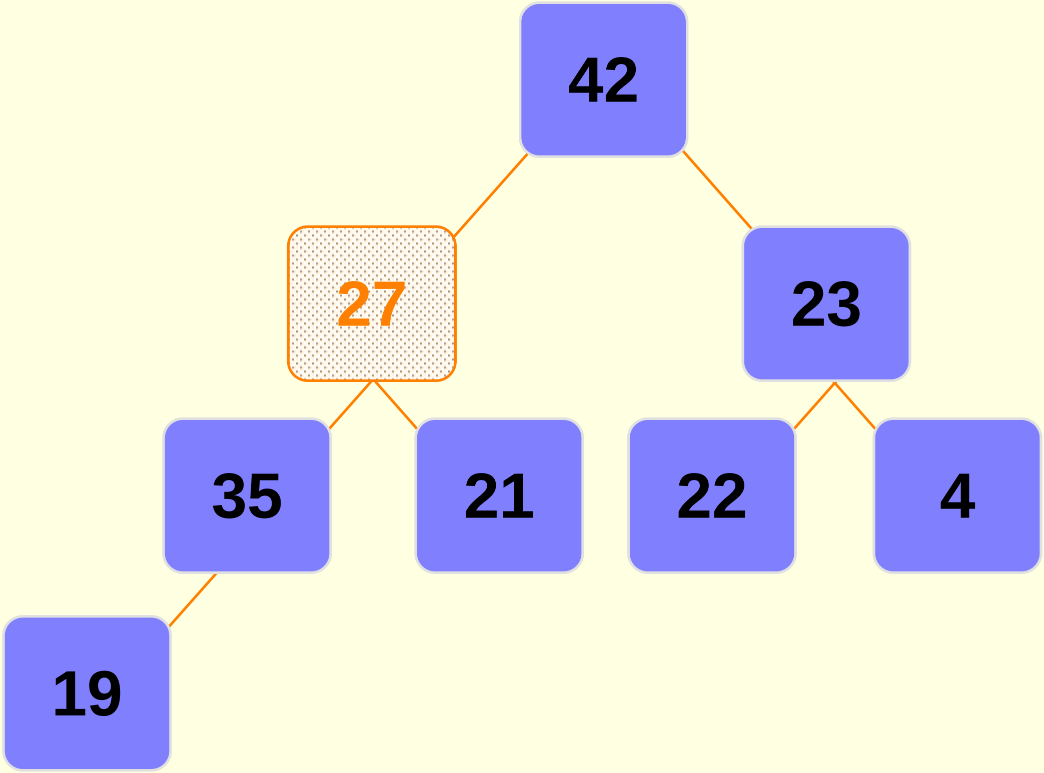
- **Percolate Down:**
- Push the out-of-place node downward,
  - swapping with its **larger** child
- until the out-of-place node reaches an acceptable location



Binary Heaps

- Deleting Nodes from a Binary Heap

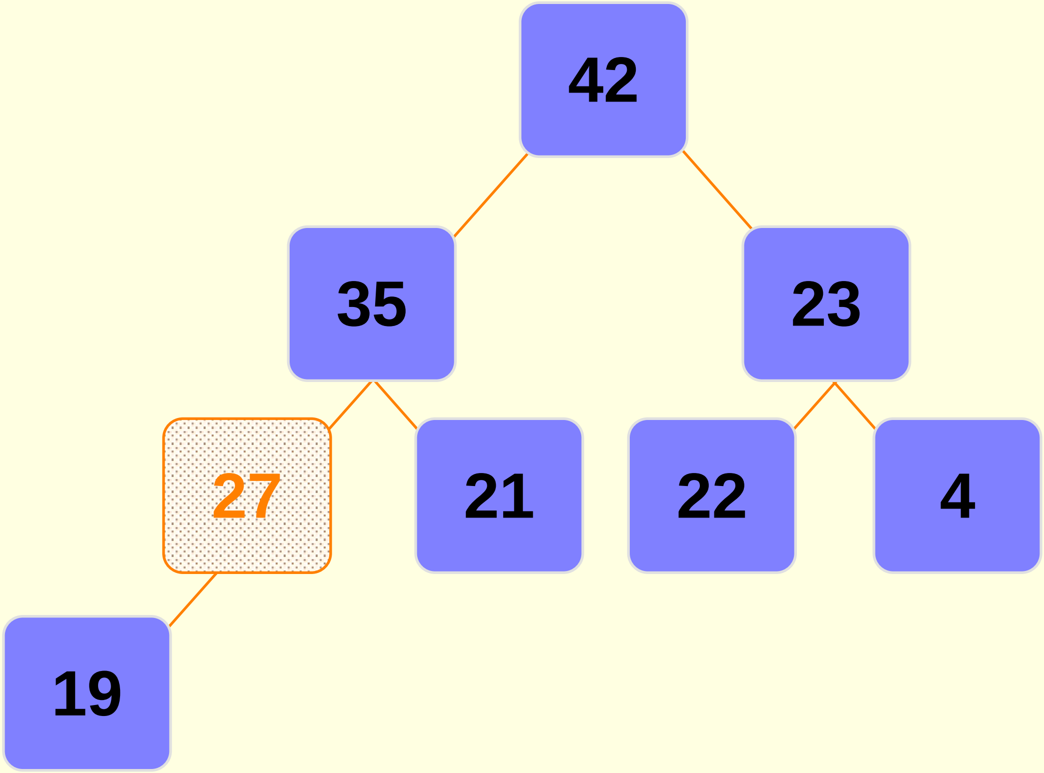
- **Percolate Down:**
- Push the out-of-place node downward,
  - swapping with its **larger** child
- until the out-of-place node reaches an acceptable location



Binary Heaps

- Deleting Nodes from a Binary Heap

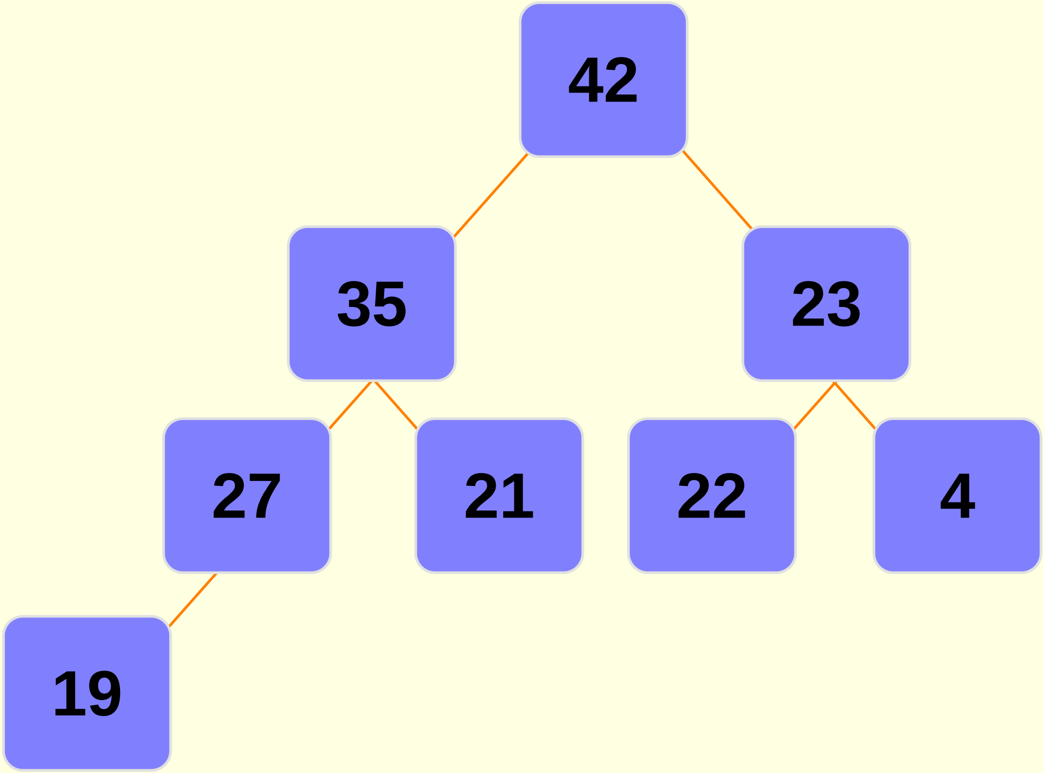
- **<u>Percolate Down:</u>**
- Push the out-of-place node downward,
  - swapping with its **<u>larger</u>** child
- until the out-of-place node reaches an acceptable location



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?
  - Deleting the minimum value 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)

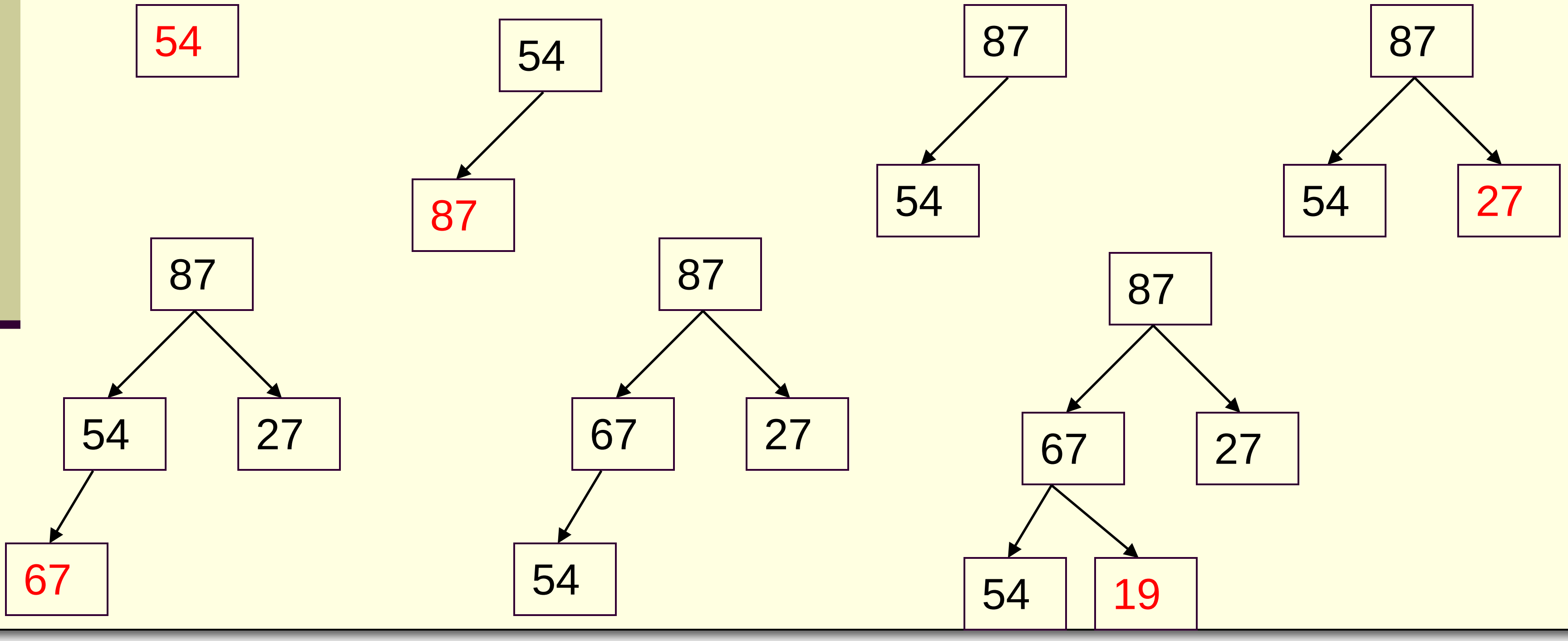
Brief Interlude: FAIL Picture

Binary Heaps

- Building a Heap from scratch
  - Given: an unsorted list of n values
    - **54, 87, 27, 67, 19, 31, 29, 18, 32, 56, 7, 12, 31**
  - How can we build a heap from these values?
    - It is really just a series of "insertions"
    - Simply insert the n elements into the heap in the order that they arrive (in our case, from left to right)
    - WHILE there are more elements:
      1) Insert the next element
      2) Percolate Up to a suitable position
  - Once all elements are inserted, we have our heap

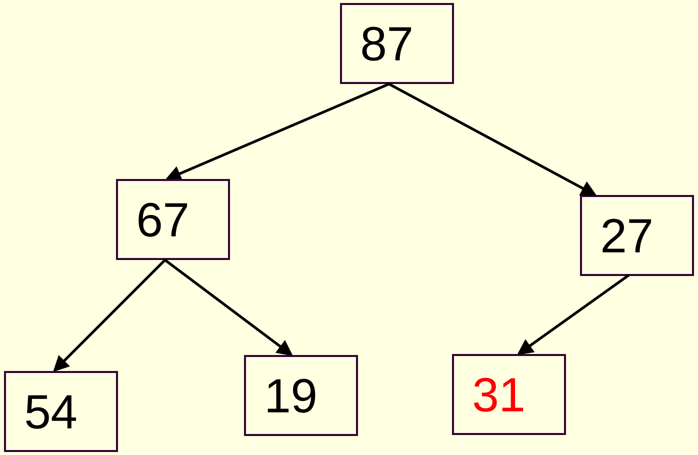
Binary Heaps

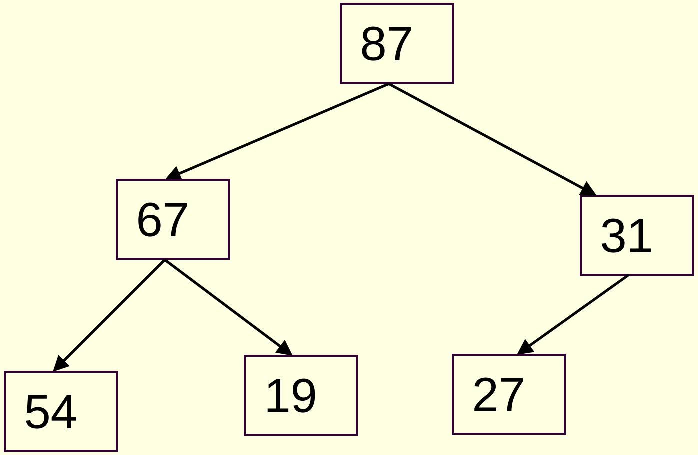
- Building a Heap from scratch
  - Given: an unsorted list of n values
    - **54, 87, 27, 67, 19, 31, 29, 18, 32, 56, 7, 12, 31**



Binary Heaps

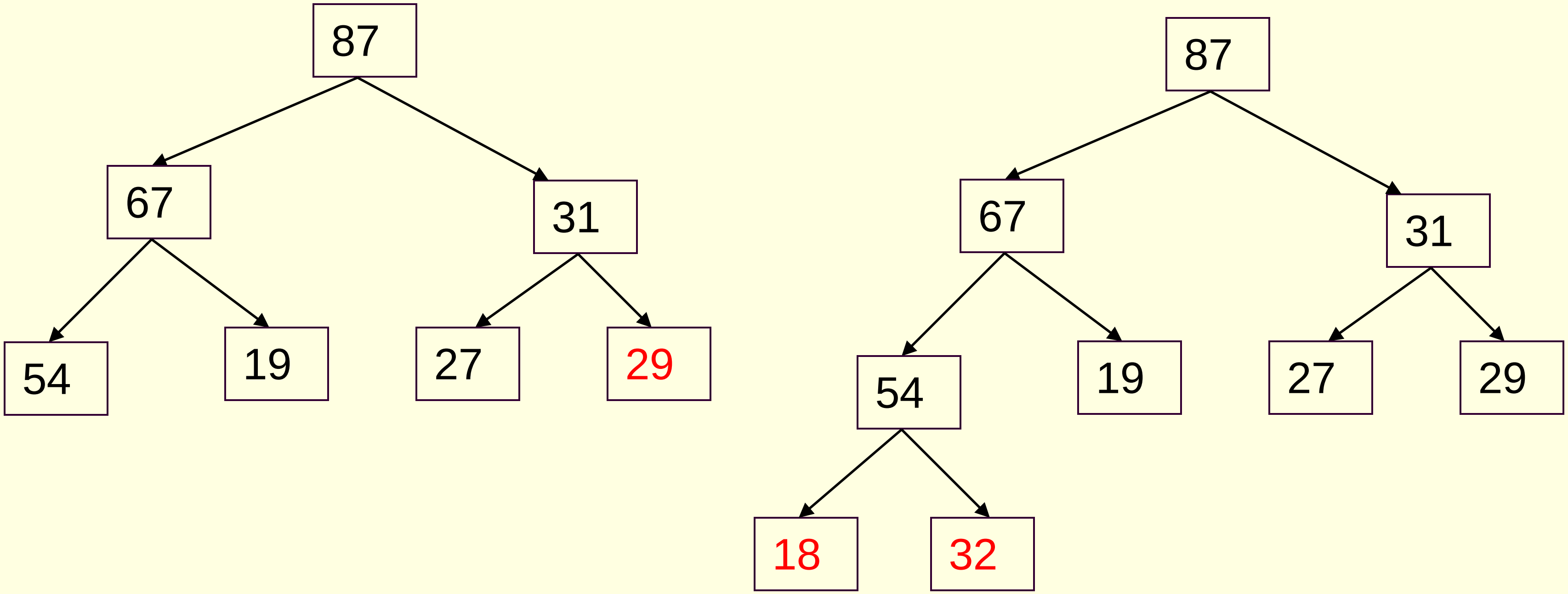
- Building a Heap from scratch
  - Given: an unsorted list of n values
    - **54, 87, 27, 67, 19, 31, 29, 18, 32, 56, 7, 12, 31**





Binary Heaps

- Building a Heap from scratch
  - Given: an unsorted list of n values
    - **54, 87, 27, 67, 19, 31, 29, 18, 32, 56, 7, 12, 31**



Binary Heaps

- Building a Heap from scratch
  - Running time:
    - How long does it take to do one insertion?
      - We just covered this!
      - An insertion takes O(logn)
        - As in the worst case, it has to Percolate all the way Up to root
    - And we have **n** elements to insert
    - Running time to make a heap from n elements is O(nlogn)

Binary Heaps

- Building a Heap from scratch
  - Can we do better than O(nlogn) time?
    - Turns out that we can
  - Start by arbitrarily placing your elements into a complete binary tree
  - Then, starting at the lowest level,
  - Perform a Percolate Down (if necessary
  - So we work from the bottom and go up to the root
  - Performing a Percolate Down at each node
    - Only if necessary
  - This function is known as **Heapify**

Binary Heaps

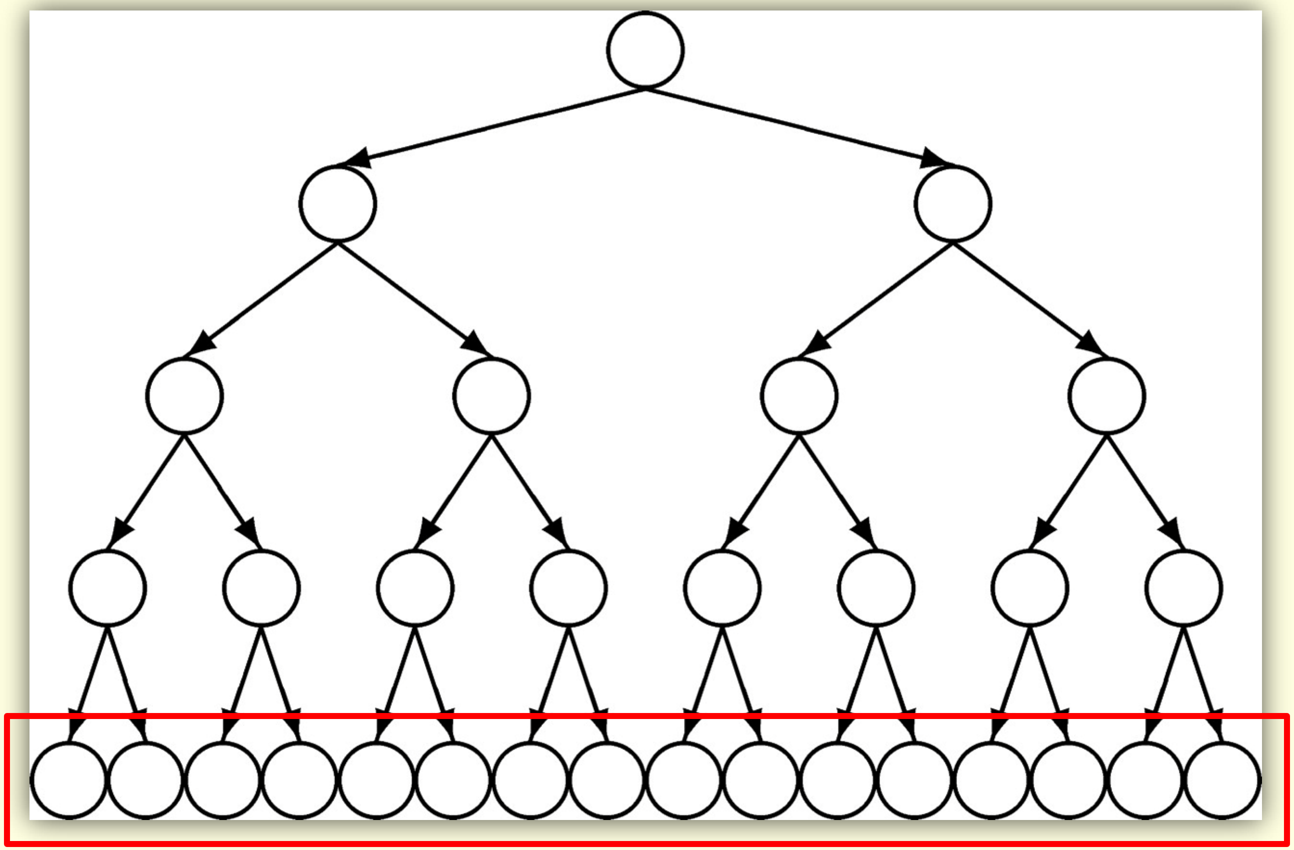
- Building a Heap from scratch
  - Running time:
    - Note:
      - Realize that for any given complete tree, that is completely filled, the lowest level has ½ of the total nodes in a tree
      - In a complete tree of 31 nodes, the lowest level has 16 nodes
        - And since they are already at the lowest level,
        - Those 16 nodes will NOT need to Percolate Down

Binary Heaps

- Building a Heap from scratch

These nodes do NOT have to Percolate Down!

They are already at the bottom most level.



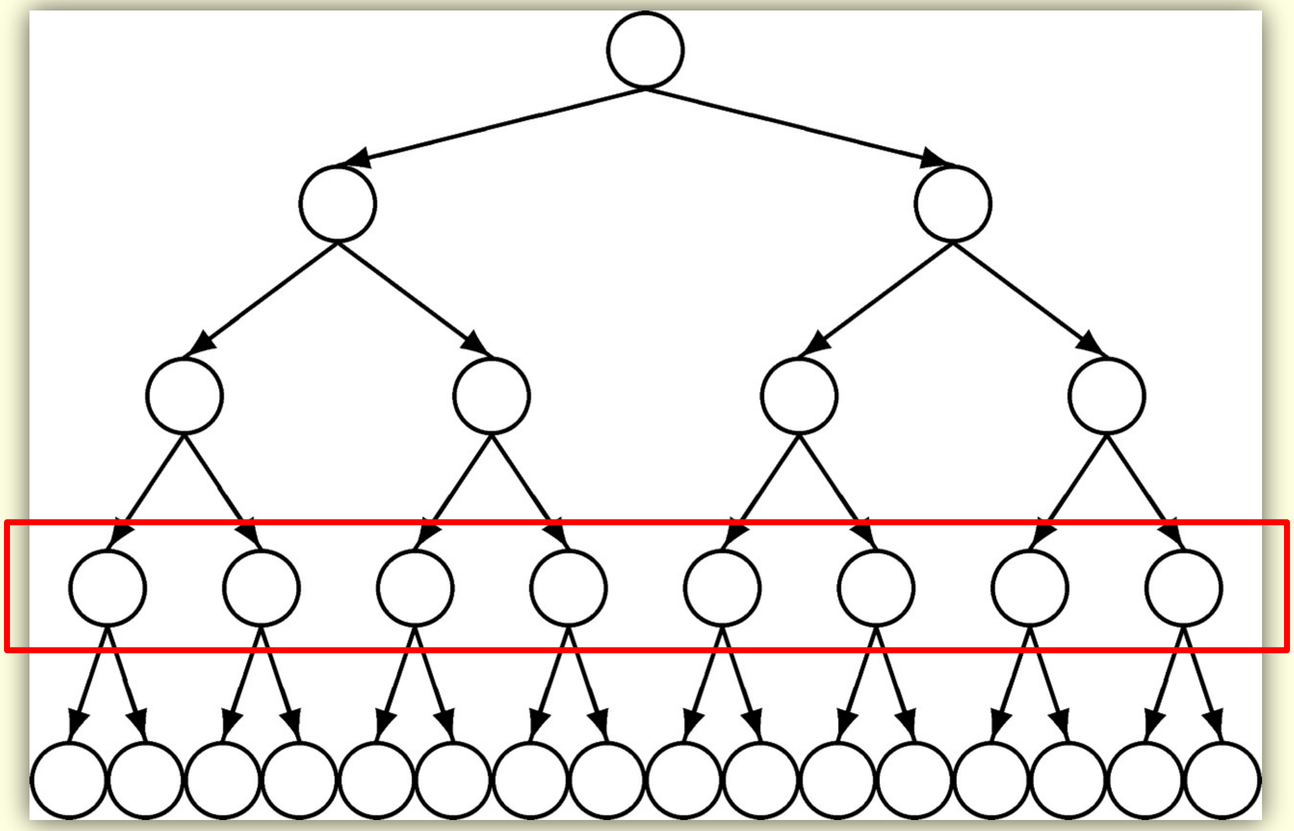
Binary Heaps

- Building a Heap from scratch
  - Running time:
    - Note:
      - Realize that for any given complete tree, that is completely filled, the lowest level has ½ of the total nodes in a tree
      - In a complete tree of 31 nodes, the lowest level has 16 nodes
        - And since they are already at the lowest level,
        - Those 16 nodes will NOT need to Percolate Down
      - <u>The level above the 16 nodes has 8 nodes</u>
      - What can we say about those 8 nodes?
      - Notice that, at MOST, those 8 nodes will have to Percolate Down only one level

Binary Heaps

- Building a Heap from scratch

These nodes only have to Percolate Down one level.



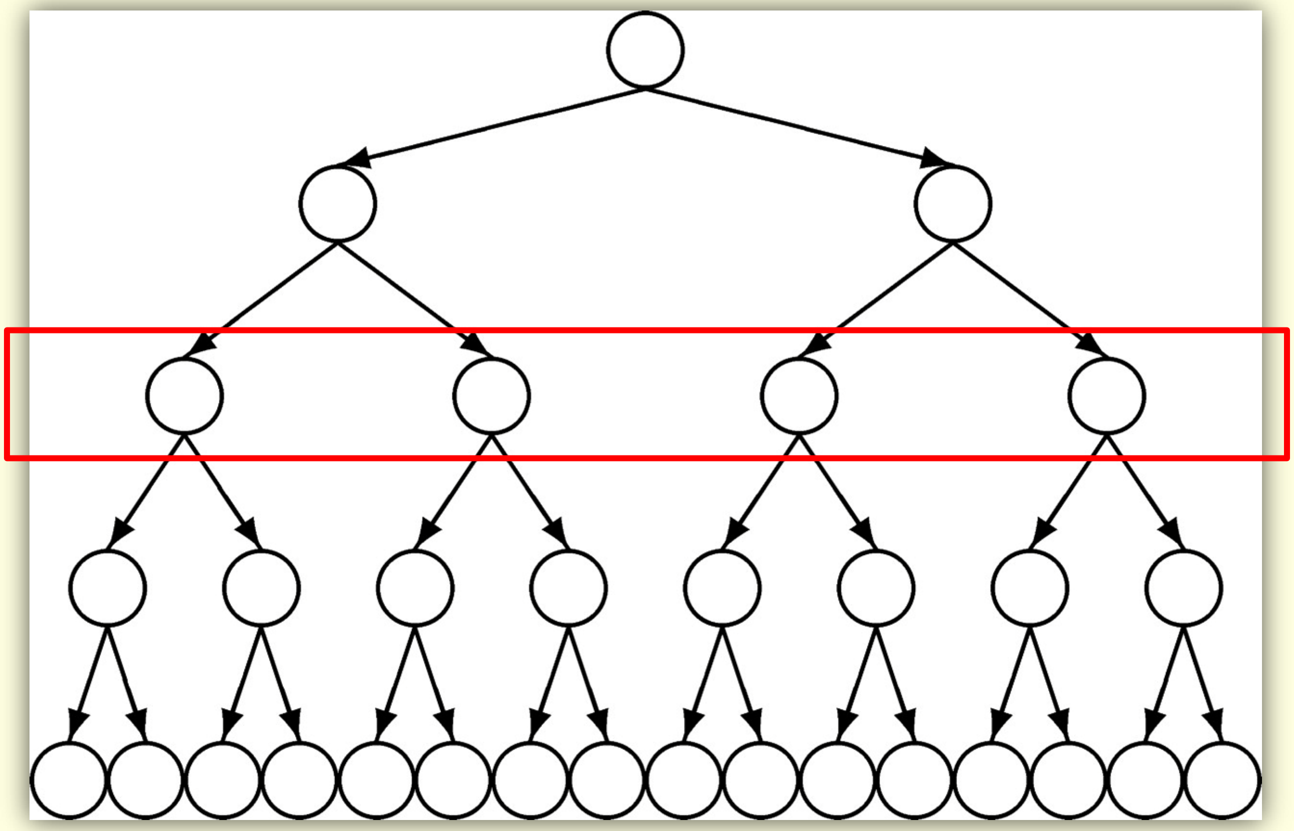
Binary Heaps

- Building a Heap from scratch
  - Running time:
    - Note:
      - Realize that for any given complete tree, that is completely filled, the lowest level has ½ of the total nodes in a tree
      - In a complete tree of 31 nodes, the lowest level has 16 nodes
        - And since they are already at the lowest level,
        - Those 16 nodes will NOT need to Percolate Down
      - <u>The level above the 16 nodes has 8 nodes</u>
      - What can we say about those 8 nodes?
      - Notice that, at MOST, those 8 nodes will have to Percolate Down only one level
      - And the level above the 8 nodes has 4 nodes
      - Those 4 nodes, at most, percolate down 2 levels, etc, etc.

Binary Heaps

- Building a Heap from scratch

These nodes only have to Percolate Down two levels.



Binary Heaps

- Building a Heap from scratch
  - Running time:
    - So only ½ of the nodes in a tree may need to be percolated down one level or more
    - Only ½ of those (1/4 of the total) may have to be percolated down two or more levels
    - Only ½ of those (1/8 of the total) may have to be percolated down three or more levels, etc., etc.
    - So if we add up the total number of swaps, we get:
    - (1/2)*n + (1/4)*n + (1/8)*n + … ≈ n
    - **<u>So this Heapify function runs in O(n) time</u>**

Binary Heaps

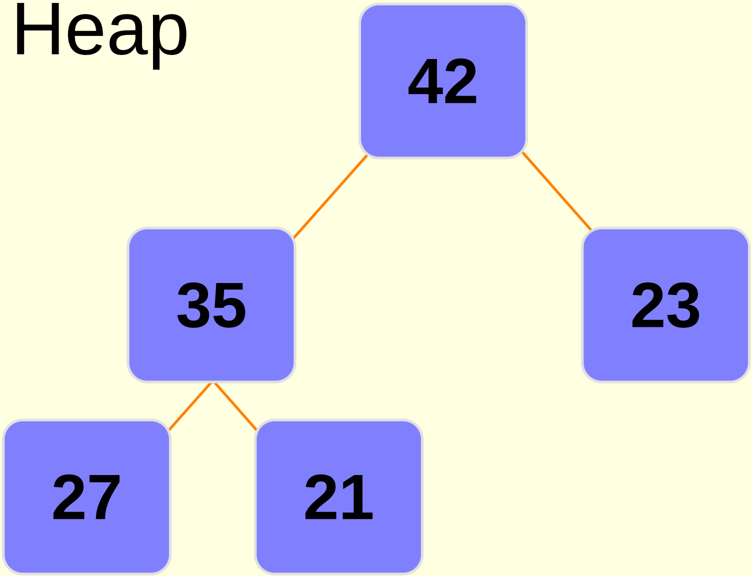
- Implementing a Binary Heap
  - Remember:
    - a binary heap is a complete binary tree
  - So we can implement this binary tree as an array!
  - How?
    - If a tree is "complete",
      - The root would be the 1st position of the array (index 1)
      - The two children of the node would be in index 2 and 3
      - The 4 nodes on the next level would be in index 4 – 7
      - The 8 nodes on the next level would be in index 8 - 15
      - and so on

Binary Heaps

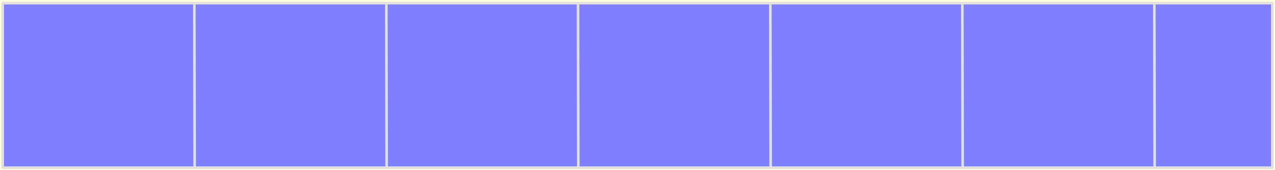
- Implementing a Binary Heap
  - Notes:
    - So we are wanting to implement one ADT
      - A Priority Queue
    - To do so, we utilize another ADT
      - A Heap
    - And to implement the actual Heap, which, in turn, implements the Priority Queue
      - We use an array!
    - So after all of this, we simply use an array
    - And the way we dereference the array and manipulate the data is what makes “the array a tree”

Binary Heaps

- Implementing a Binary Heap



- We store the data from the nodes in a partially-filled array.

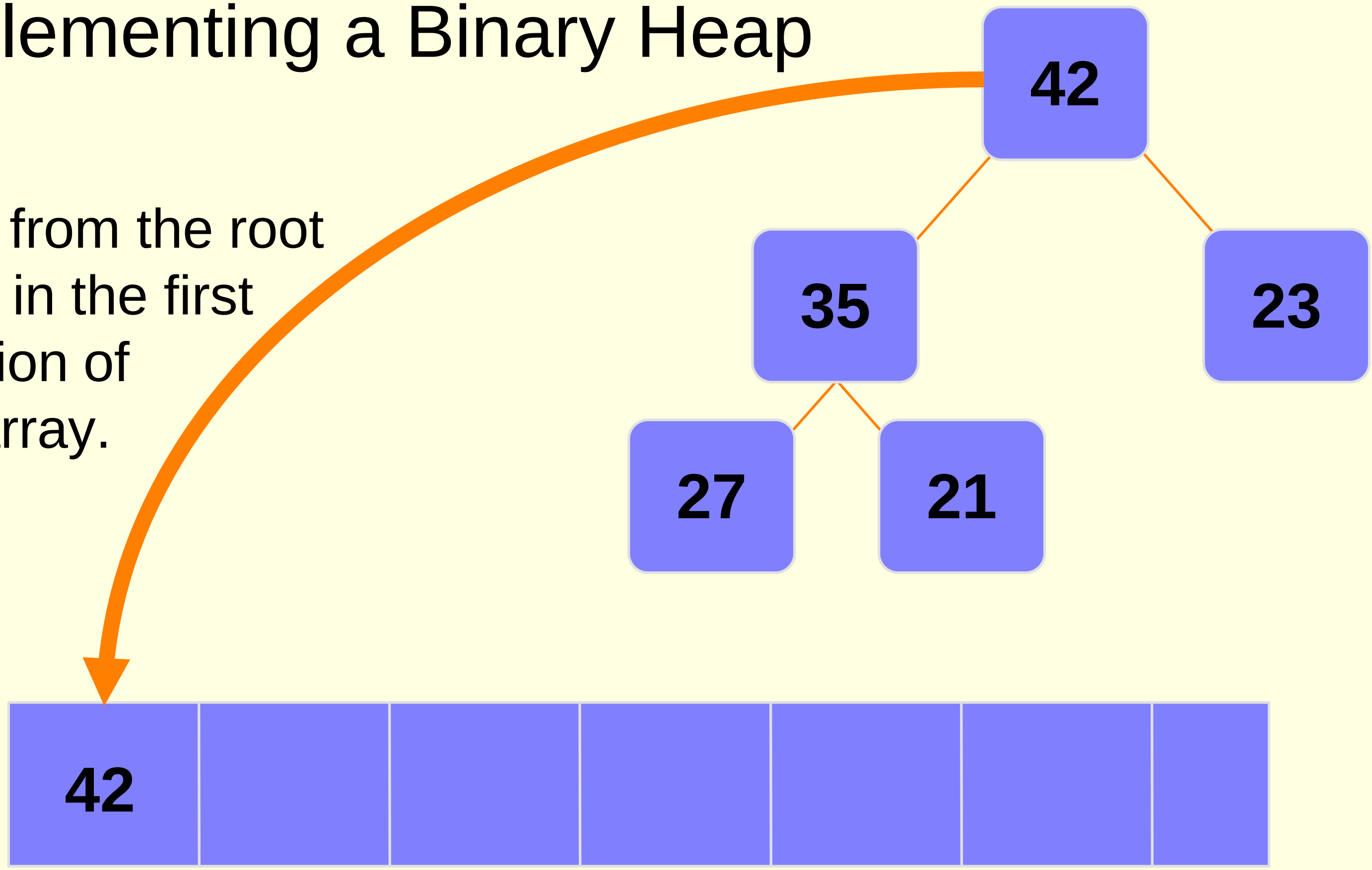


An array of data

Binary Heaps

- Implementing a Binary Heap

- Data from the root goes in the first location of the array.

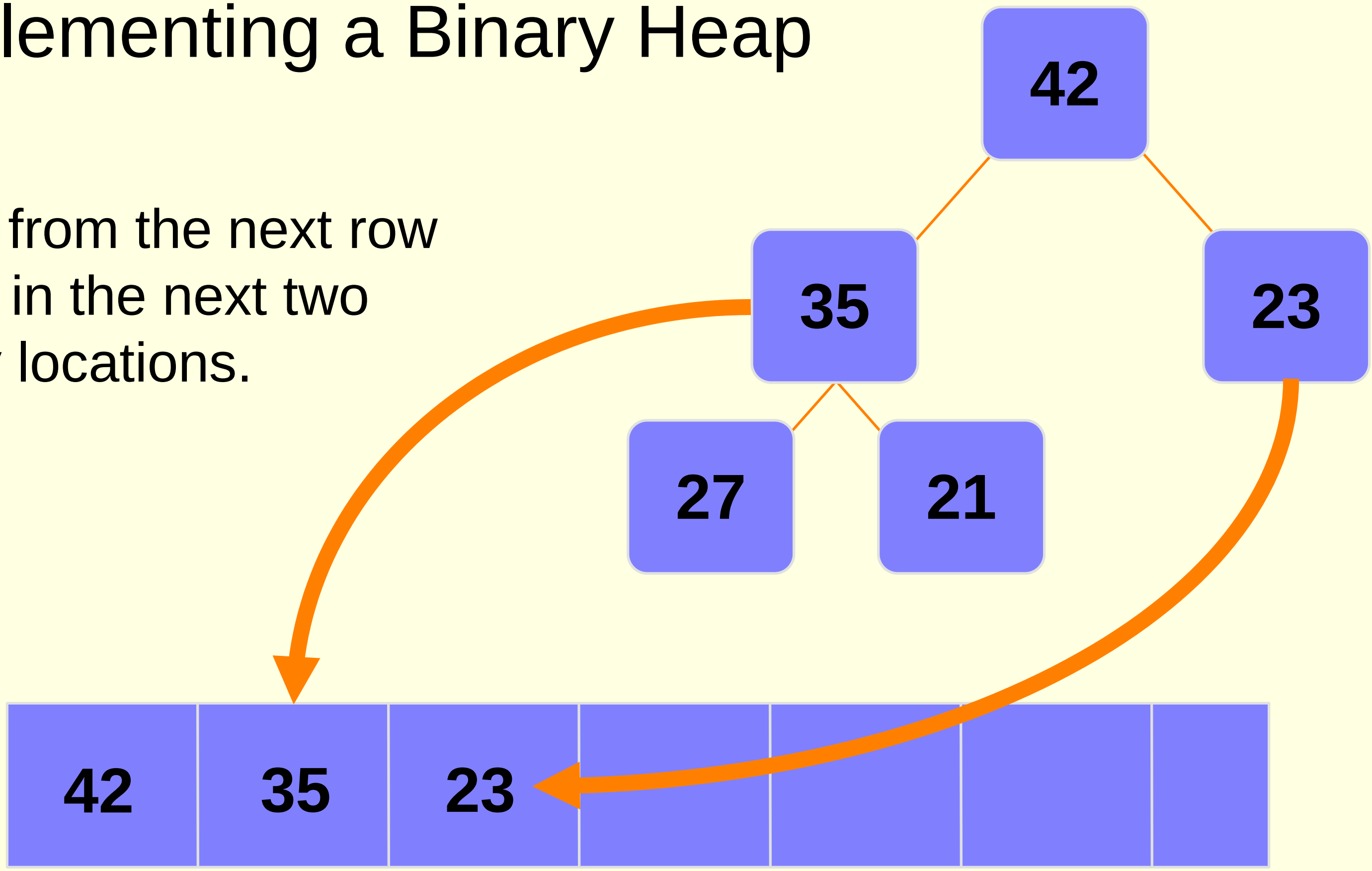


An array of data

Binary Heaps

- Implementing a Binary Heap

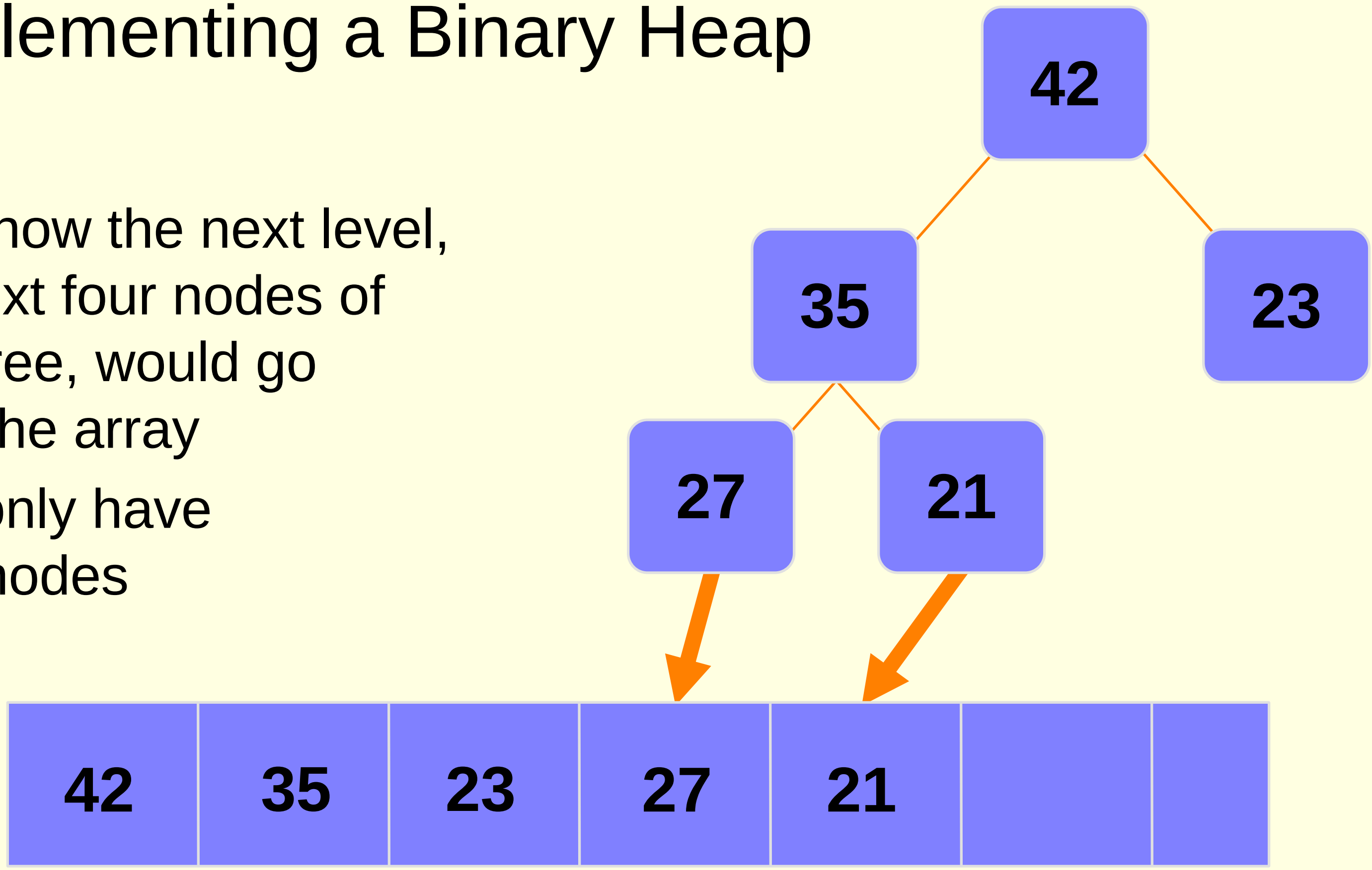
- Data from the next row goes in the next two array locations.



An array of data

Binary Heaps

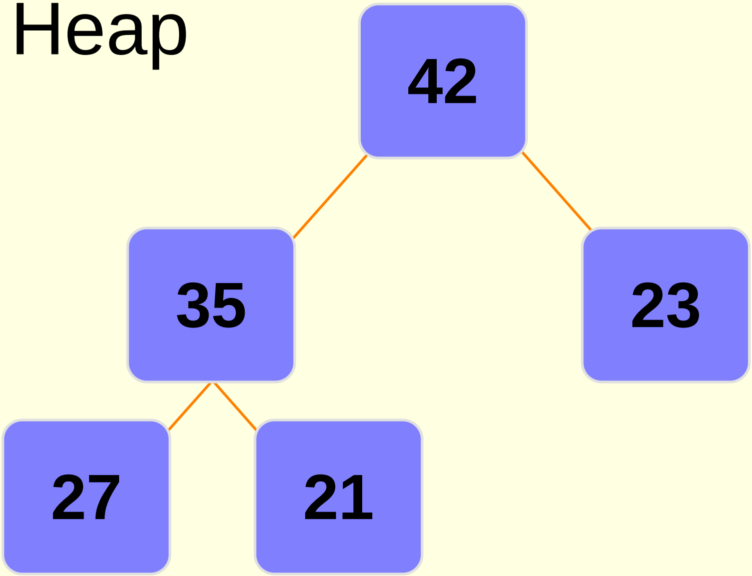
- Implementing a Binary Heap
- And now the next level, or next four nodes of the tree, would go into the array
- We only have two nodes



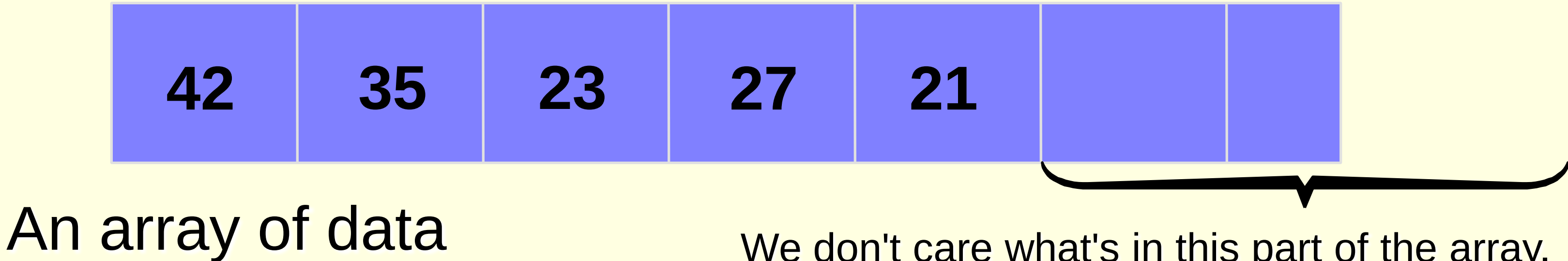
An array of data

Binary Heaps

- Implementing a Binary Heap

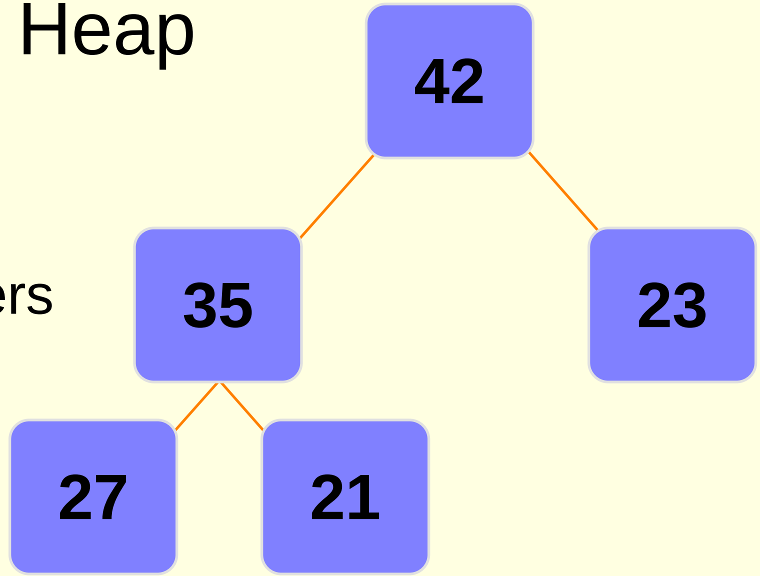


- We are only concerned with the front part of the array
- If the tree has 5 nodes, then we only care about the first five spots of the array

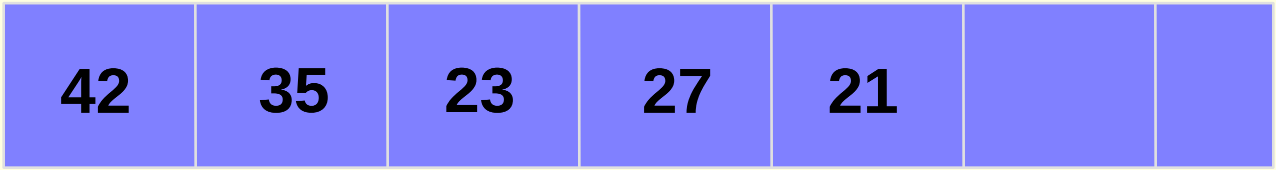


Binary Heaps

- Implementing a Binary Heap



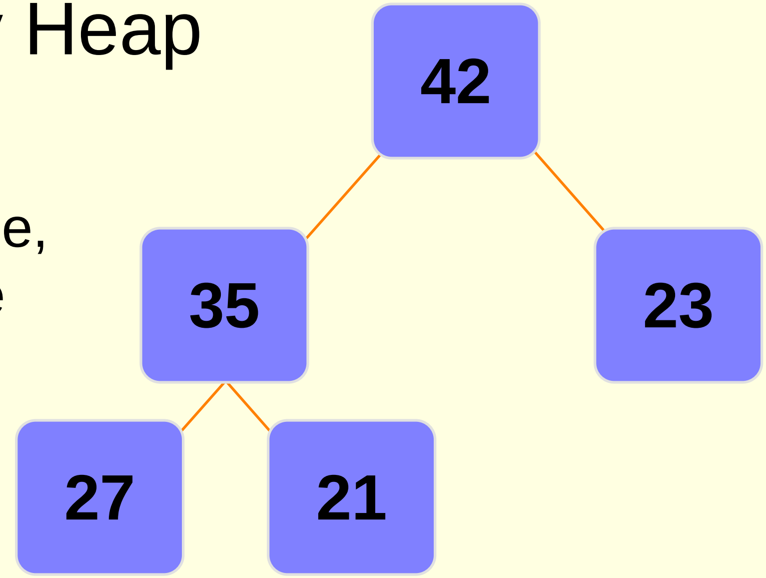
- The links between the tree's nodes are not stored as pointers
- The only way we "know" that the "array is a tree" is based on how we choose to manipulate the array



An array of data

Binary Heaps

- Implementing a Binary Heap

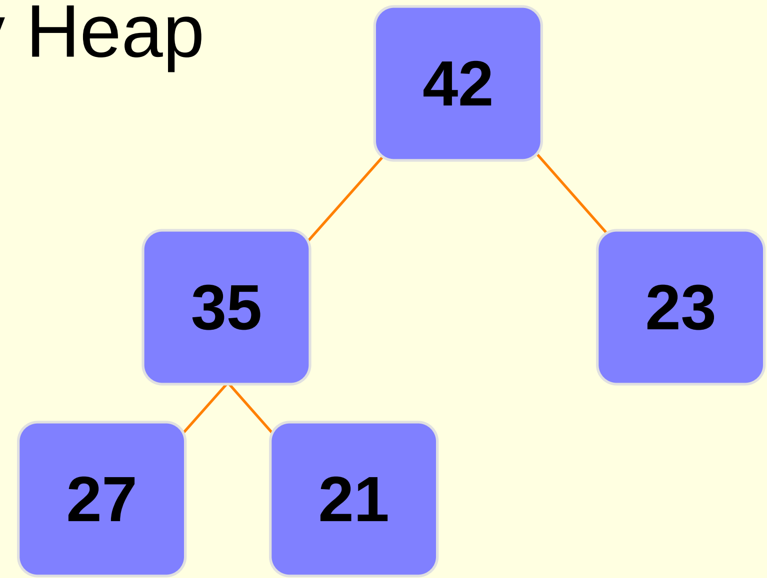


- If you know the index of a node, then it is easy to figure out the index of that node's parent or children

42	**35**	**23**	**27**	**21**		
[1]	[2]	[3]	[4]	[5]	[6]	

Binary Heaps

- Implementing a Binary Heap

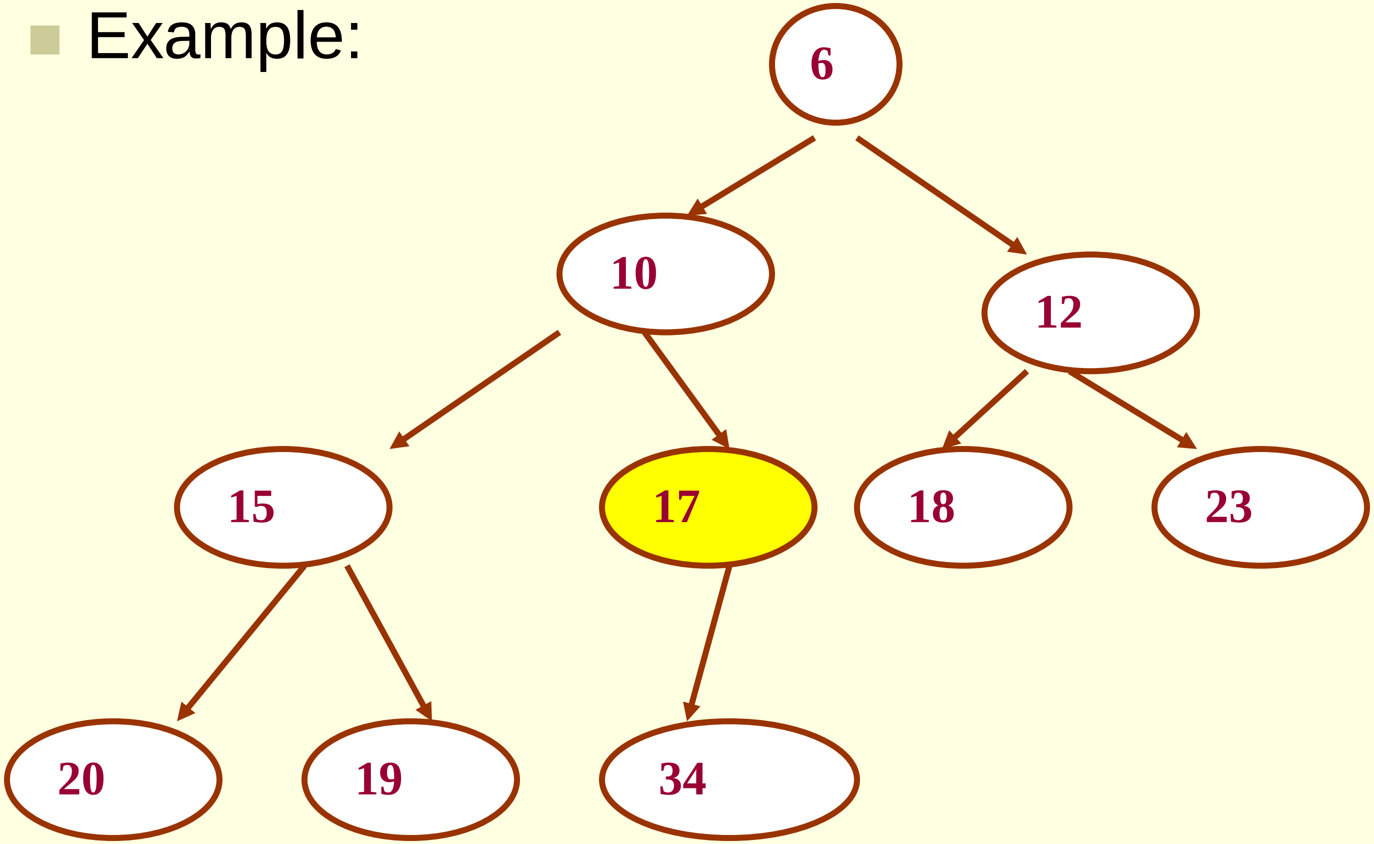


- The name of our array is A[]
- Root is at position A[1]
- Left child of A[i] = A[2i]
- Right child of A[i] = A[2i+1]
- Parent of A[i] = A[i/2]

42	35	23	27	21		
[1]	[2]	[3]	[4]	[5]	[6]	

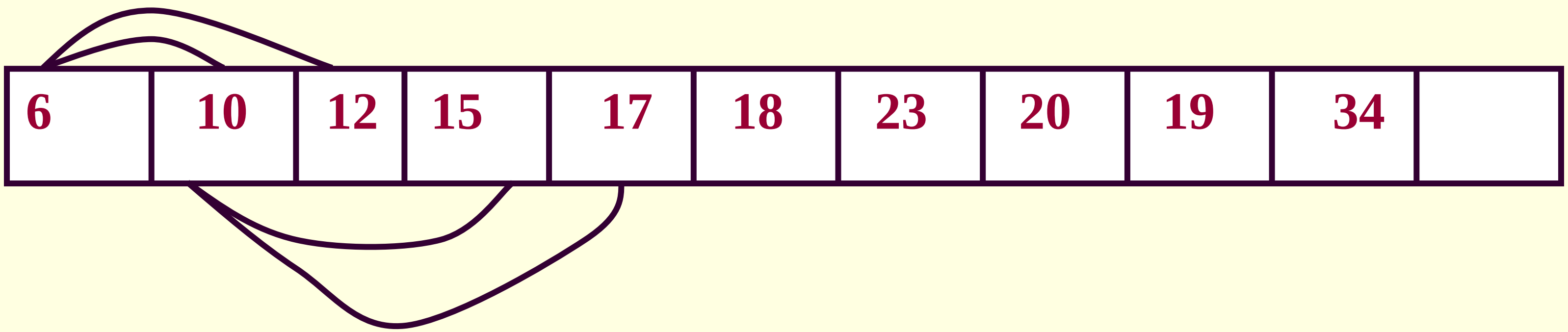
Binary Heaps

- Implementing a Binary Heap
  - Example:



Binary Heaps

- Implementing a Binary Heap
  - Example:



  - <u>Consider node 17:</u>
    - Position in the array: 5
    - It’s parent is 10, and is located at position [5/2] = 2
    - 17’s left child is node 34, and located at position 5*2 = 10
    - 17 has no right child. Position (2*5 + 1) = 11 (empty)

Binary Heaps

- Heapsort
  - We can use heaps to sort our data
  - Here's the algorithm:
    - Build a heap with all the n items
      - Takes O(n) time
    - Extract the minimum item (if a Min-heap)
      - O(1)
    - Fix the heap after extraction
      - O(logn)
    - Perform this extraction n times for all the elements
    - Store these removed items, sequentially, in an array
    - Running time: O(nlogn)

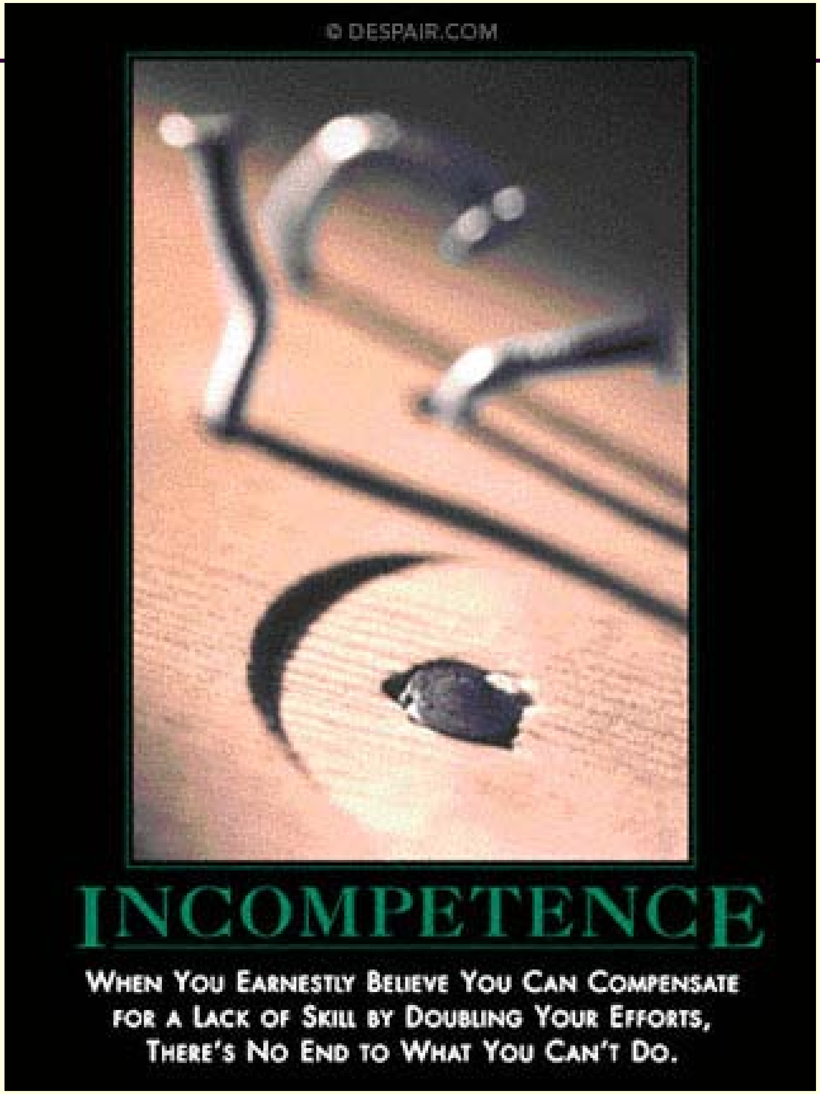
Binary Heaps

- Summary:
  - A binary heap is a tree that satisfies 2 properties:
    - The Heap Property
      - Max-heap or Min-heap
    - The Shape Property
      - Must be a complete binary tree
  - To add elements to a heap
    - Place element at next available spot and Percolate Up
  - To remove elements from a heap,
    - Delete root, as it is always the one you want to remove
    - Then copy last element to root's position
    - Finally, Percolate Down

Binary Heaps & Priority Queues

WASN'T THAT PRODIGIOUS!

Daily Demotivator



Heaps & Priority Queues

Computer Science Department
University of Central Florida

COP 3502 – Computer Science I