i206: Lecture 13:
Heaps

Tapan Parikh
Spring 2013

Some slides courtesy Marti Hearst, John Chuang and others

Heap

- A specialized binary tree that satisfies
  - Heap property
  - Complete binary tree property
- The heap property:
  - Each node's key is larger than its childrens' keys.
- The complete property:
  - All leaves at same level, and each node has two children, except possibly the last level.
- Useful for implementing priority queue, heap-sort algorithm

Heap Methods

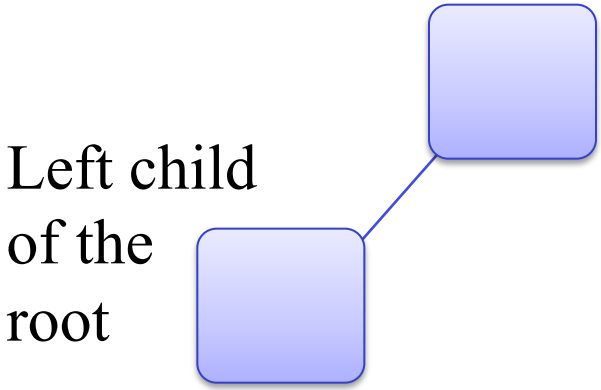
- Insert
  - Insert element as last node of the heap
  - May need to perform up-heap bubbling to restore heap-order property
- Remove
  - Remove and return element at root node
  - Move last node to root node
  - May need to perform down-heap bubbling to restore heap-order property

Heaps

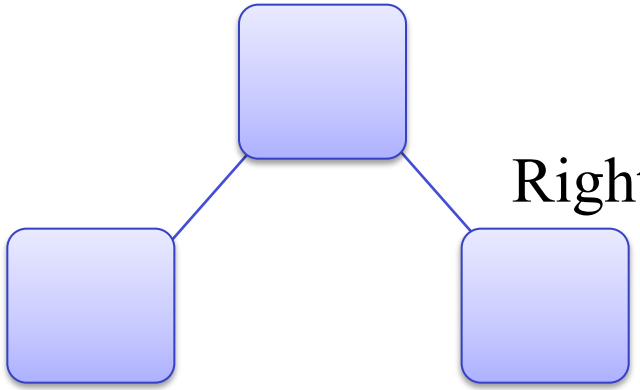
Root

A **heap** is a certain kind of complete binary tree.

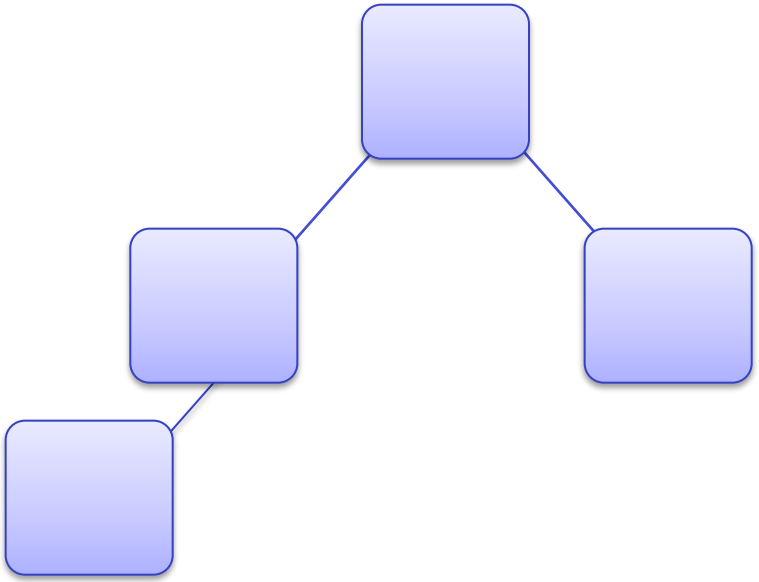
When a complete binary tree is built, its first node must be the root.



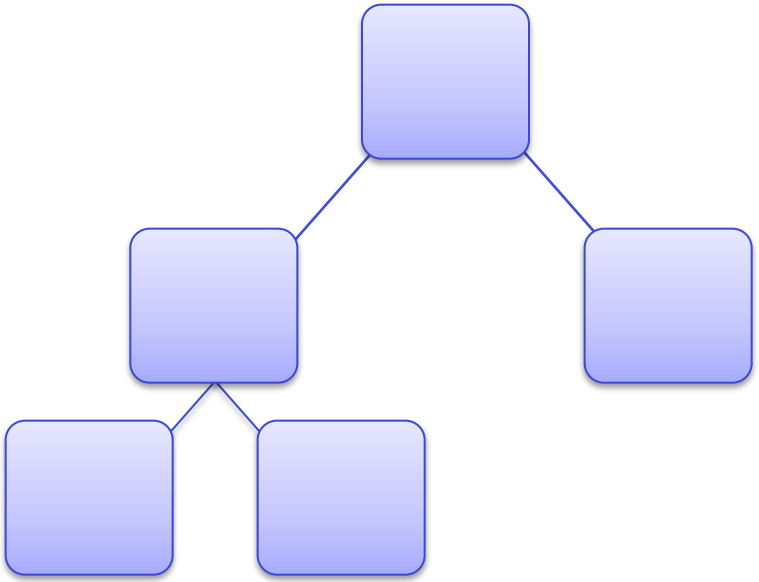
The second node is always the left child of the root.



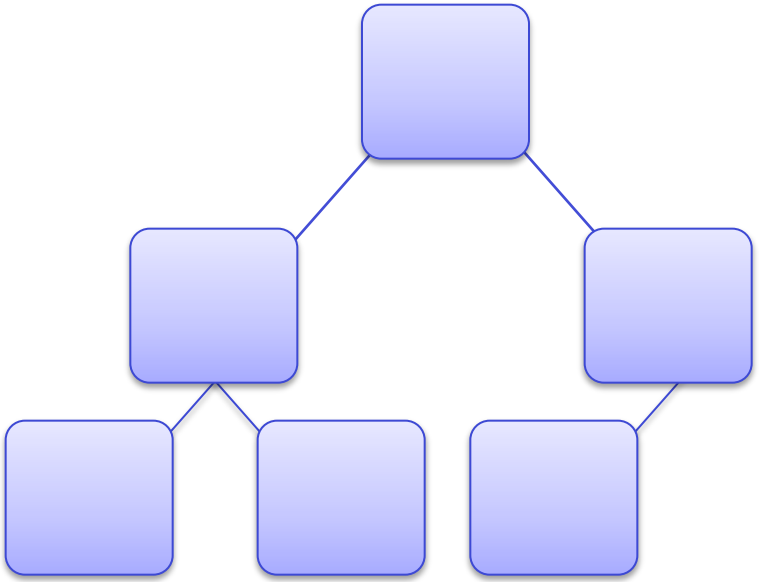
The third node is always the right child of the root.

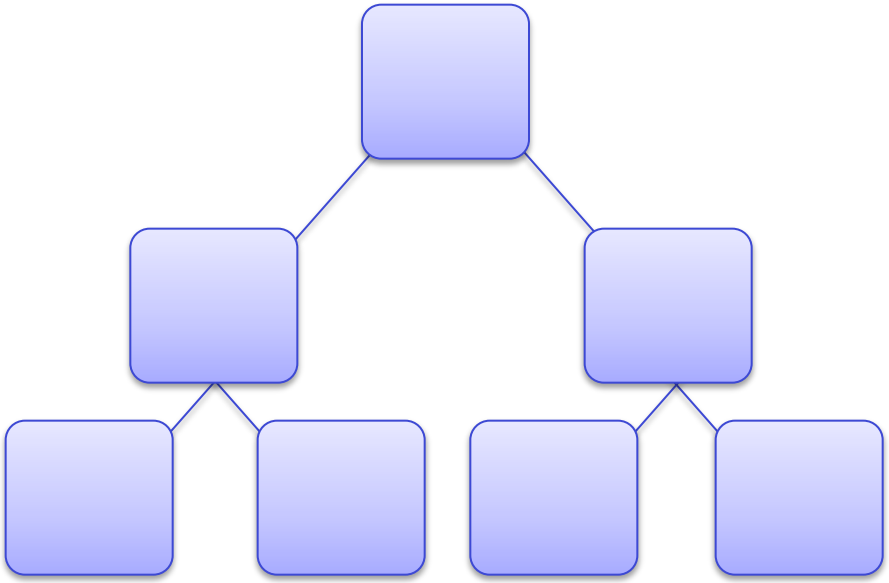


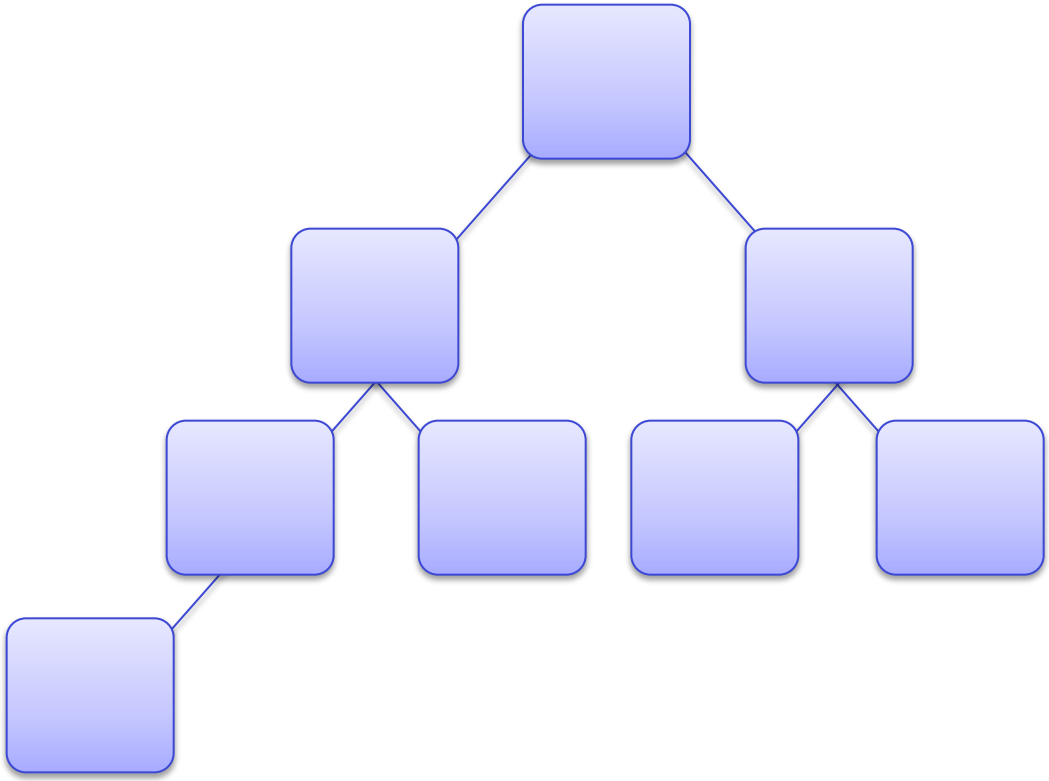
The next nodes always fill the next level from left-to-right.

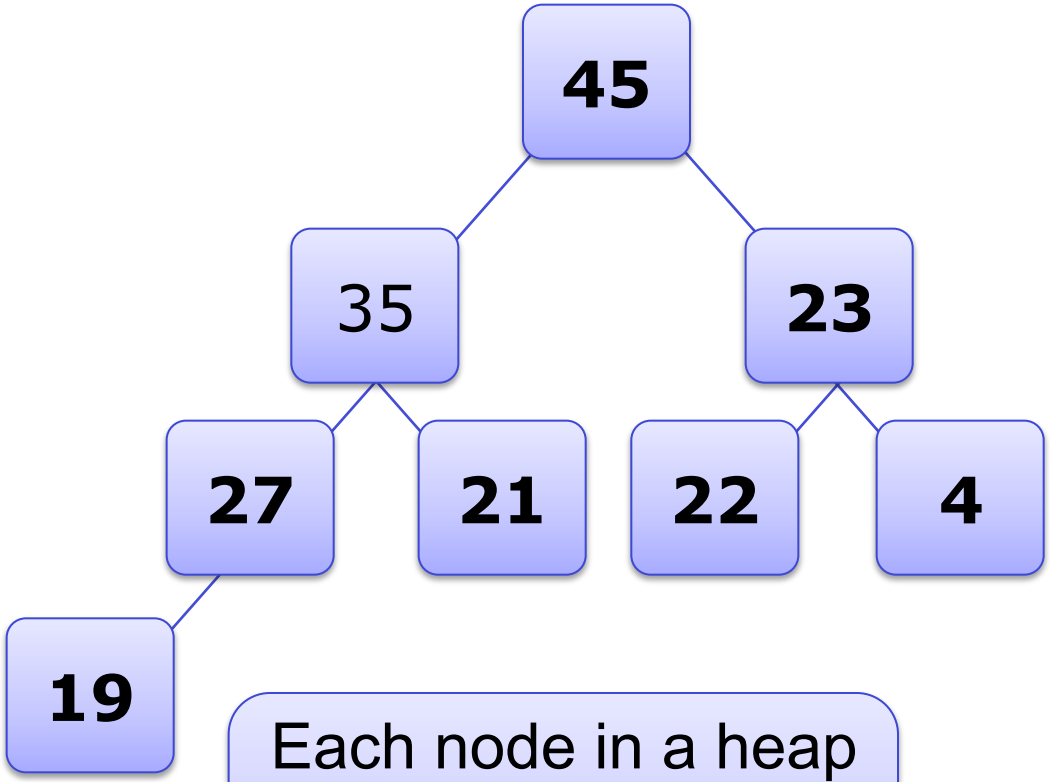


The next nodes always fill the next level from left-to-right.



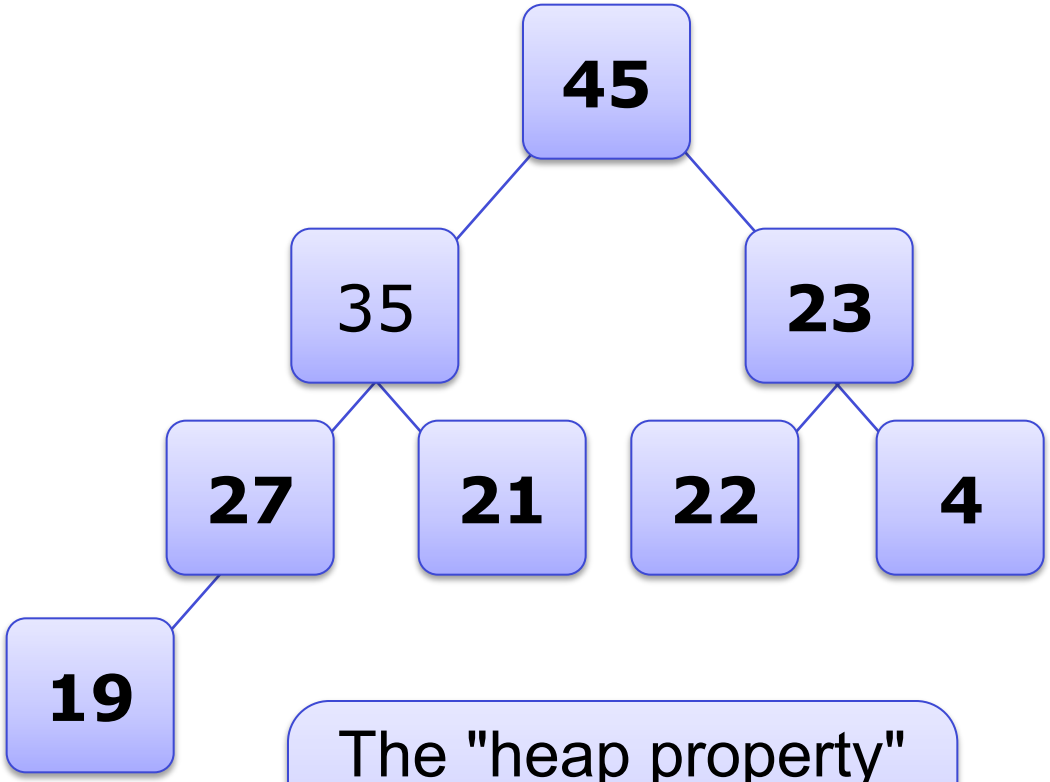






Each node in a heap contains a key that can be compared to other nodes' keys.

Largest node is always on the top.

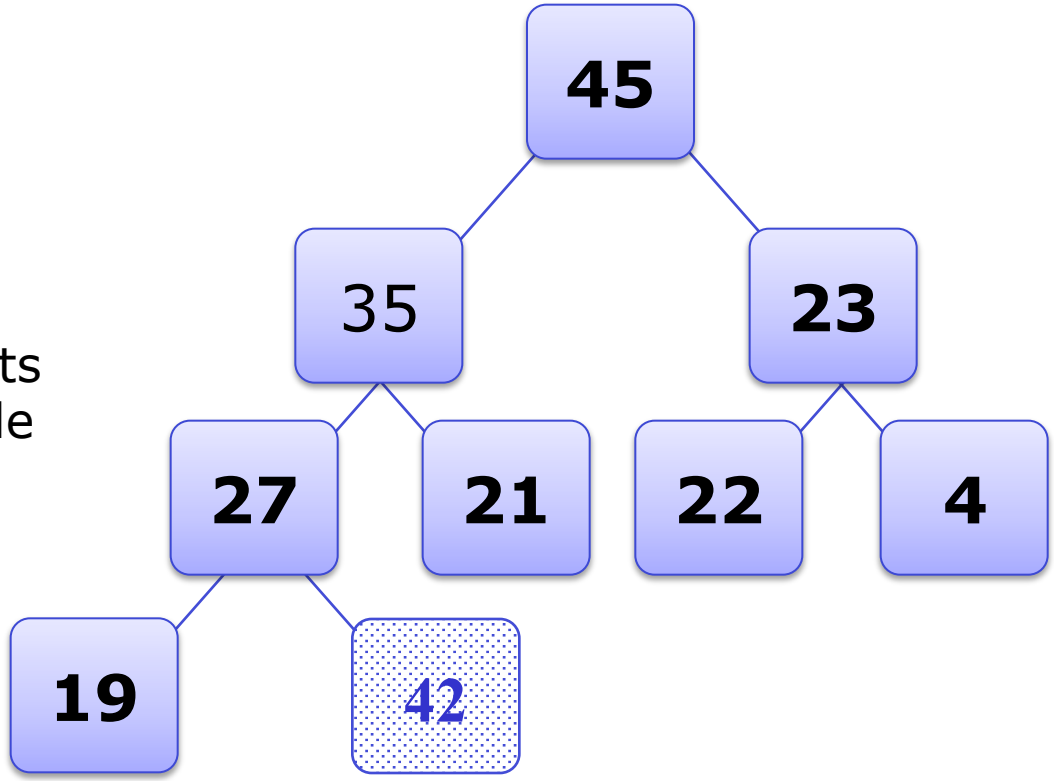


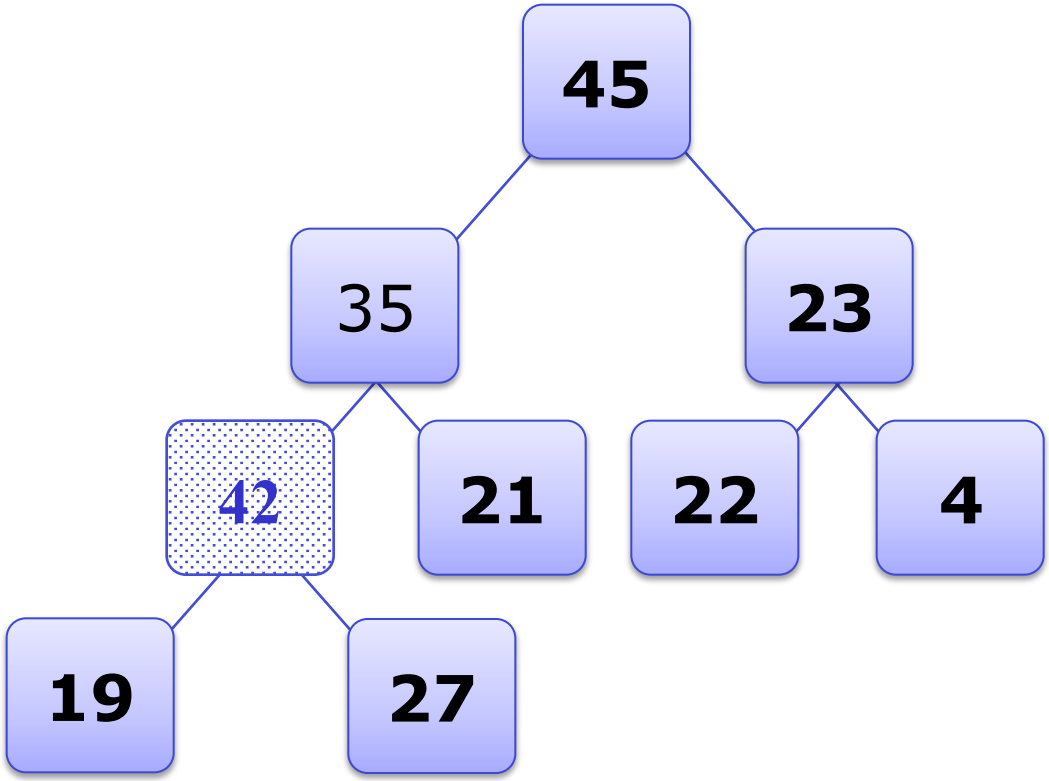
The "heap property" requires that each node's key is >= the keys of its children

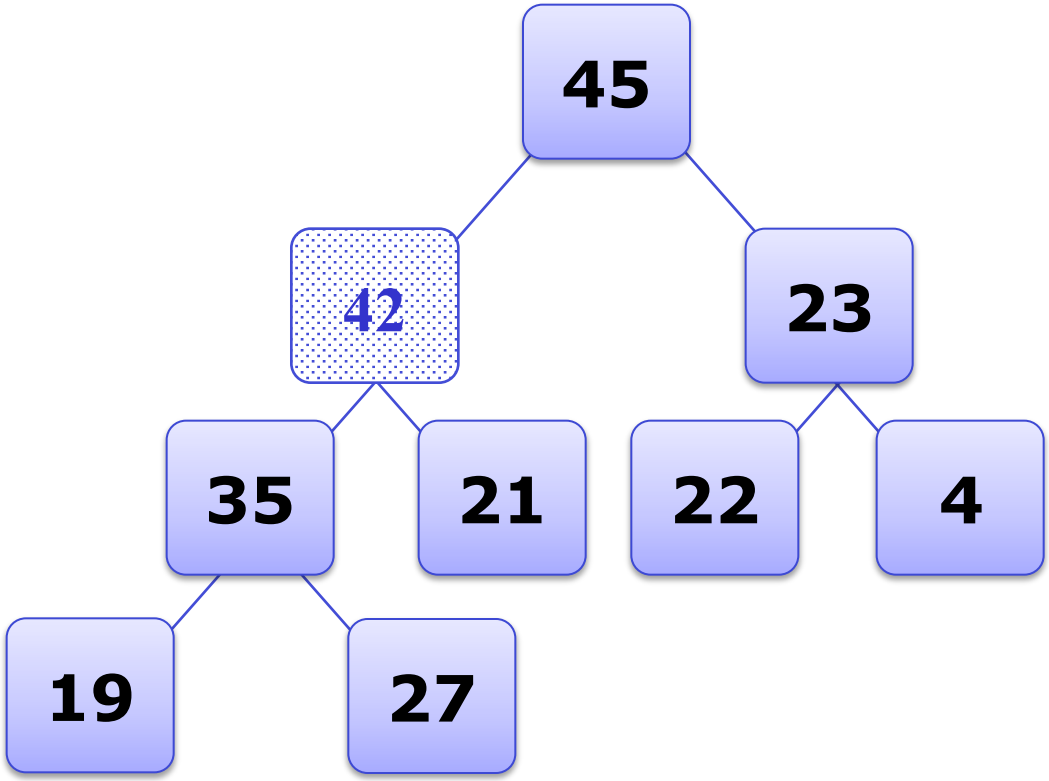
Adding a Node to a Heap

❶ Put the new node in the next available spot.

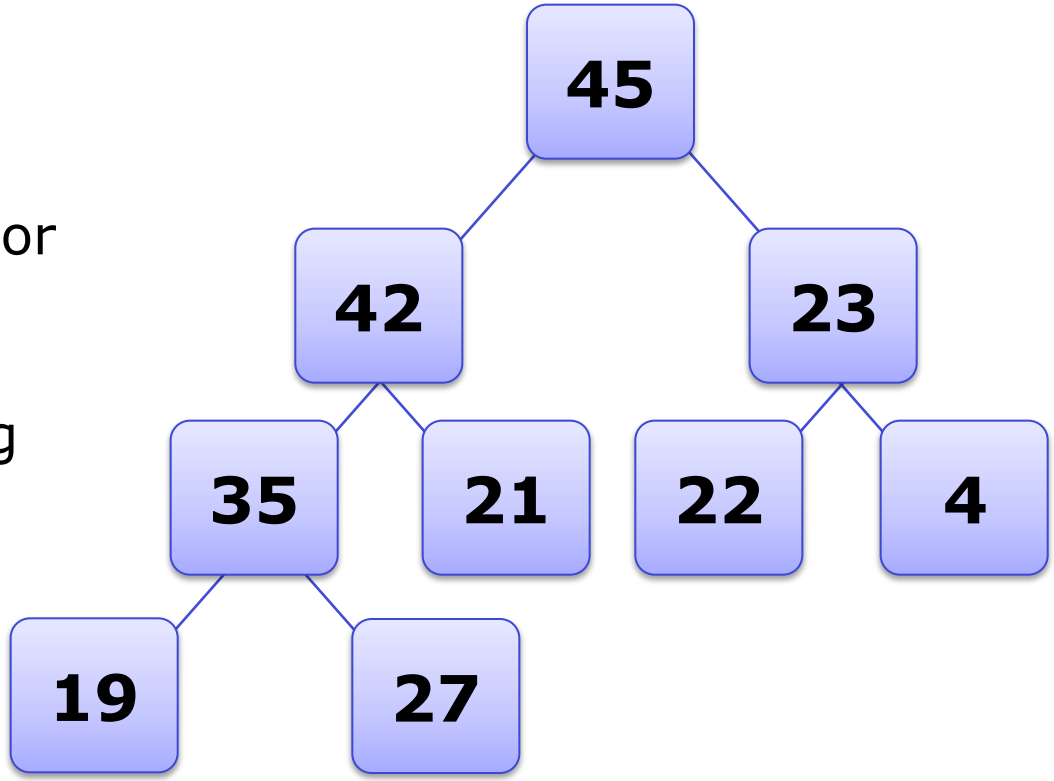
❷ Move the new node upward, swapping with its parent until the new node reaches an acceptable location.



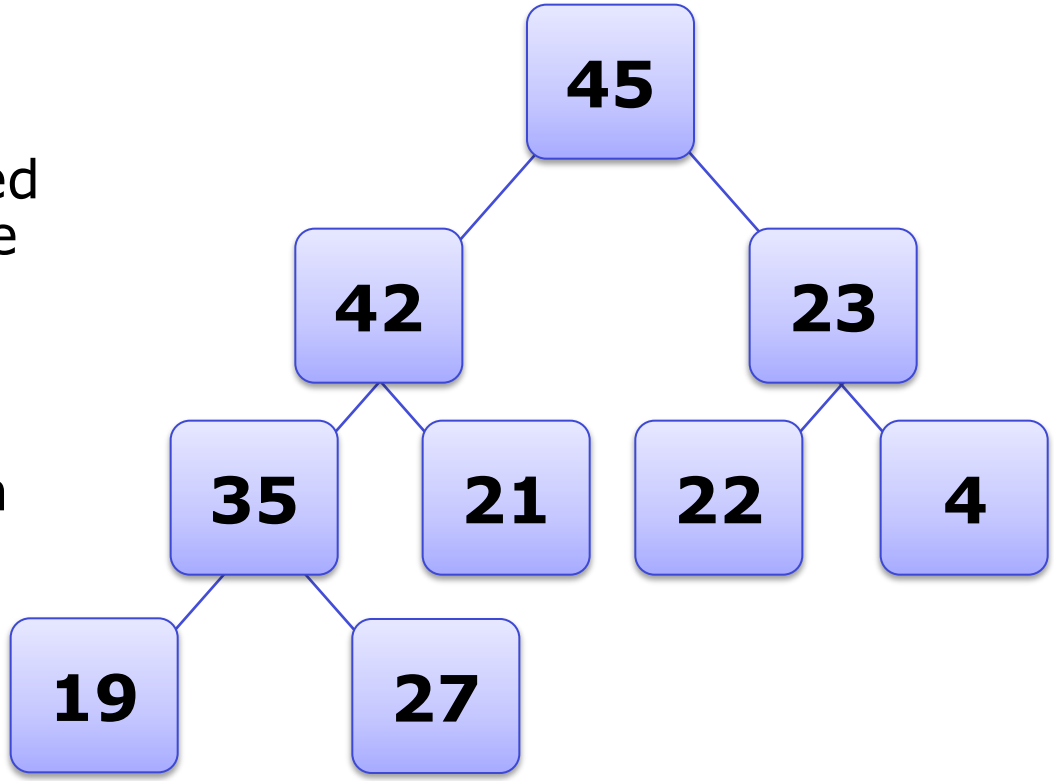




- ✓ The parent has a key that is >= new node, or
- ✓ The node reaches the root.
- ➚ The process of moving the new node upward is called **reheapification upward**.

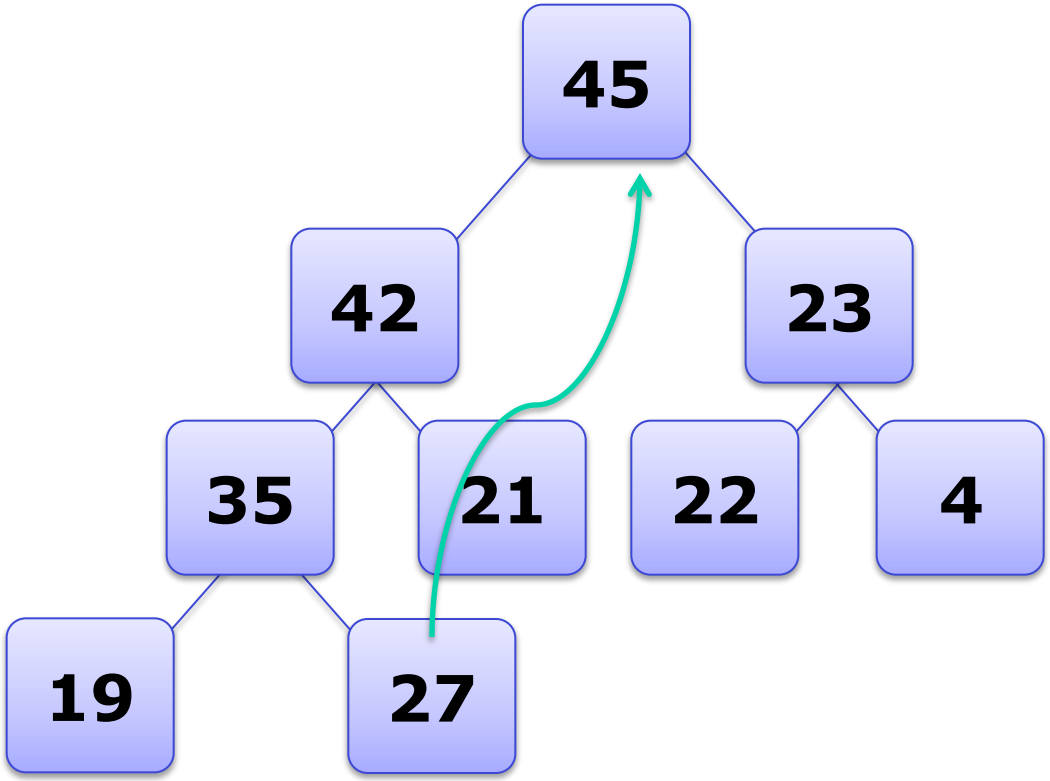


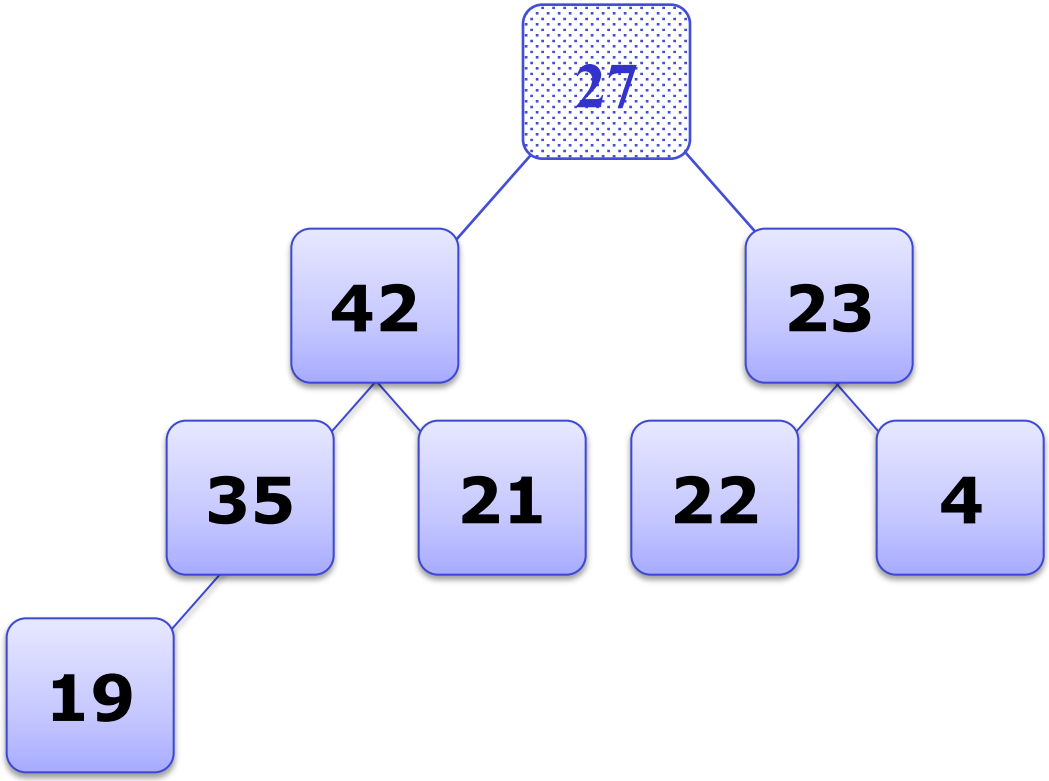
- ✓ What if we now wanted to add a node with the value 33?
- ✓ 44?
- ✓ How would the heap look after adding each of these?



Removing the Top of the Heap

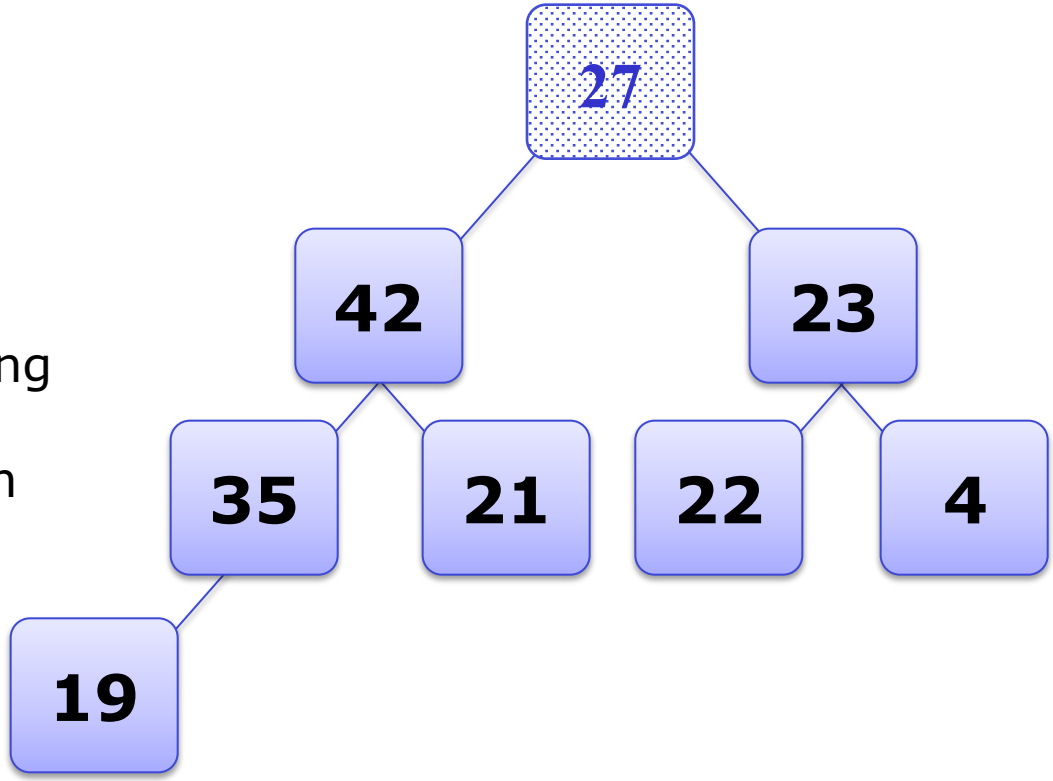
❶ Move the last node onto the root.





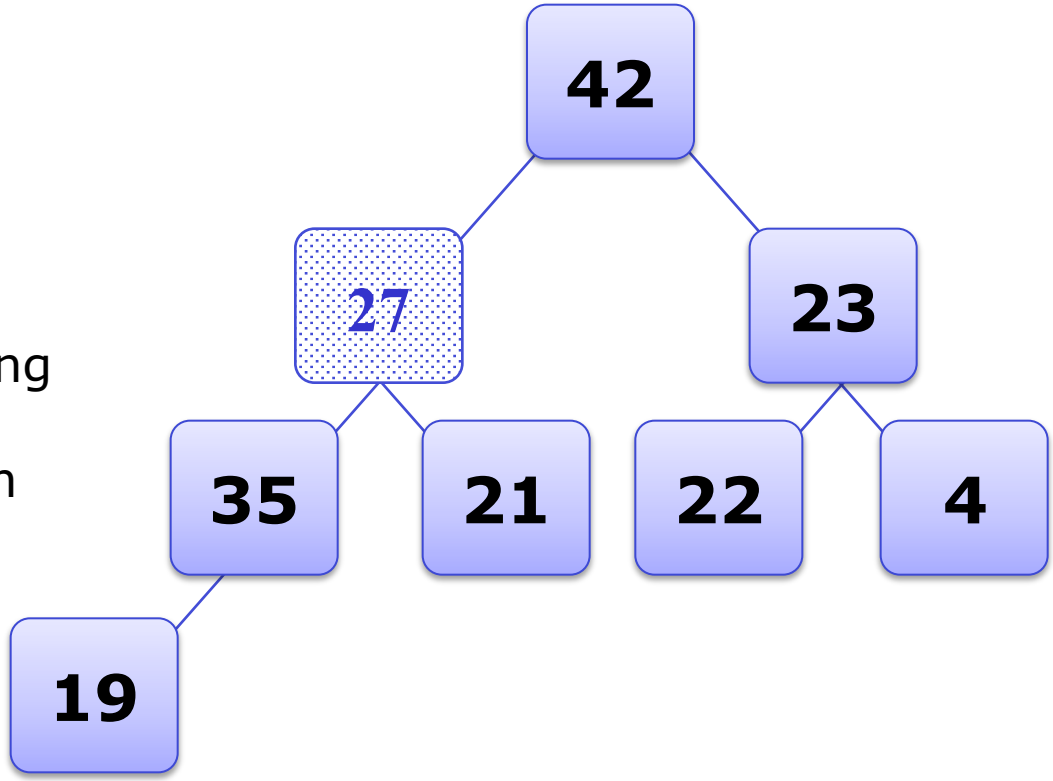
❶ Move the last node onto the root.

❷ Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



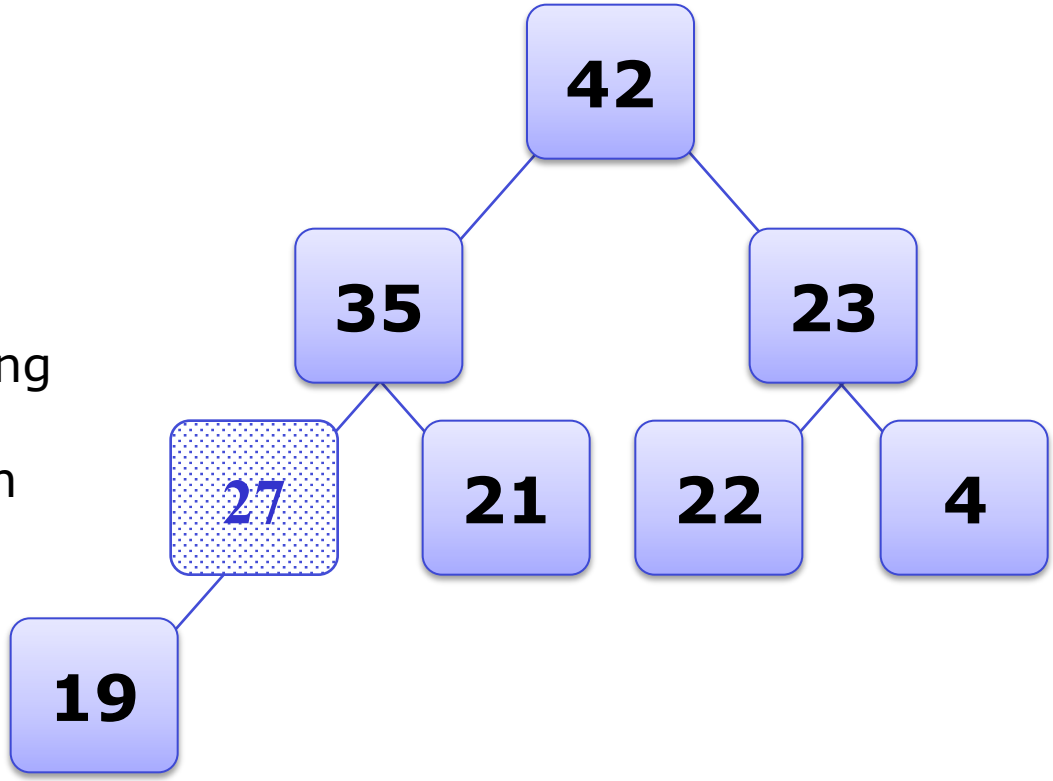
❶ Move the last node onto the root.

❷ Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.

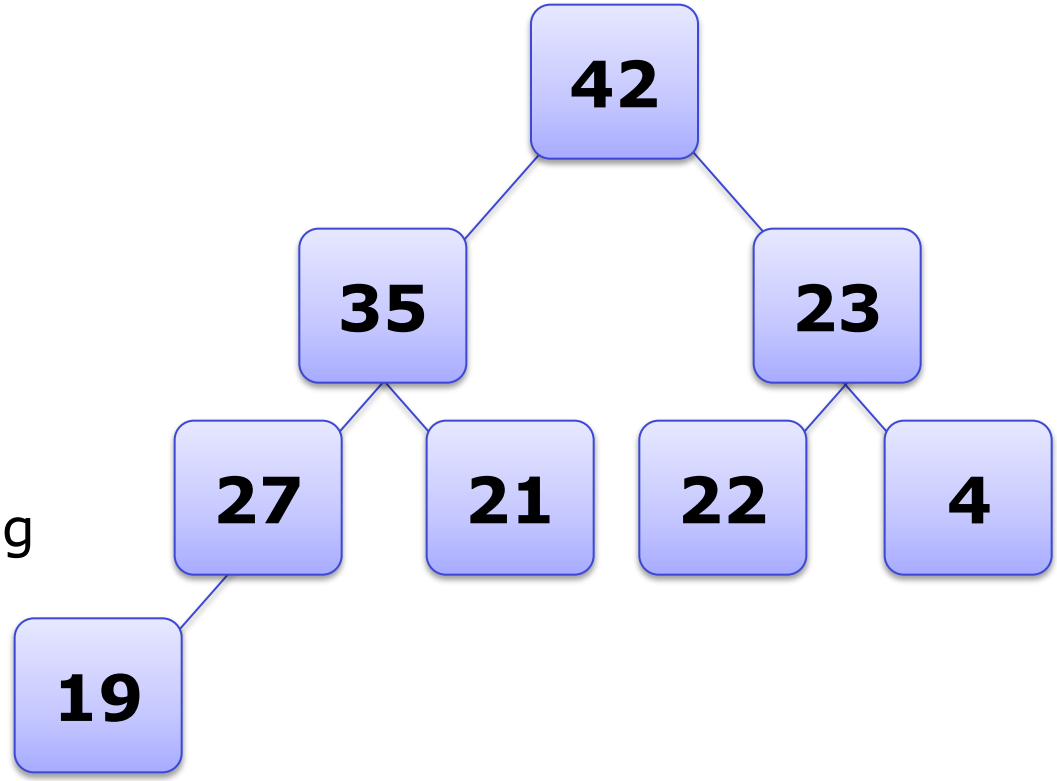


❶ Move the last node onto the root.

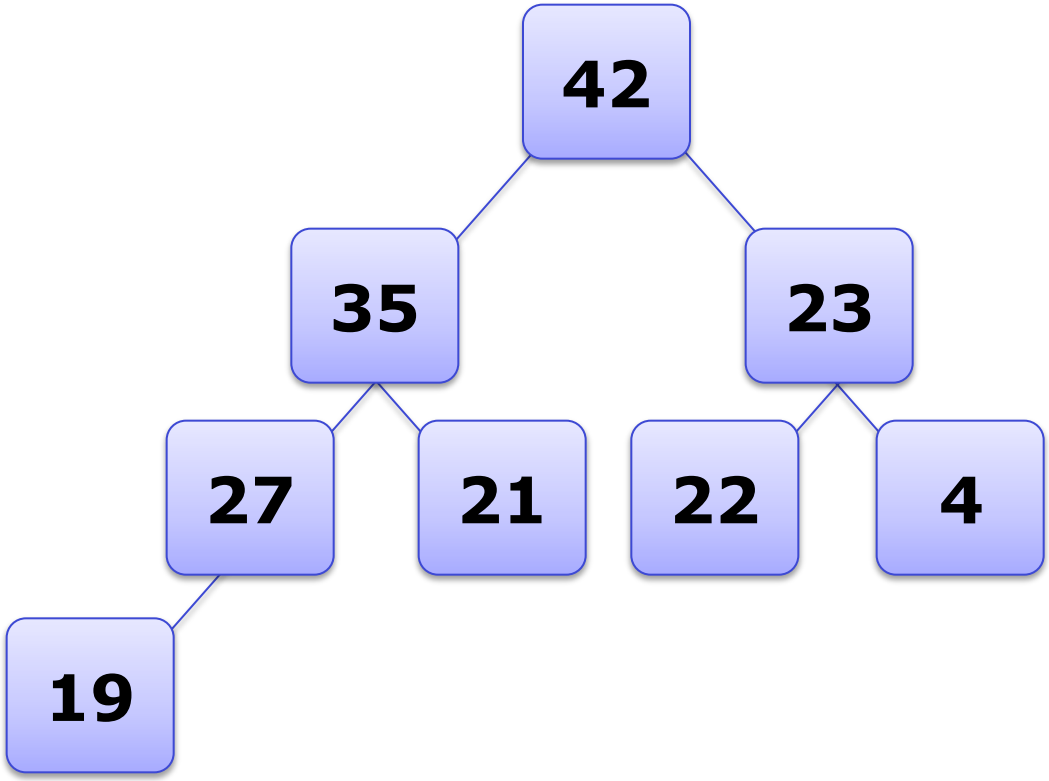
❷ Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



- ✓ The children all have keys <= the out-of-place node, or
- ✓ The node reaches the leaf.
- ↘ The process of pushing the new node downward is called **reheapification downward**.

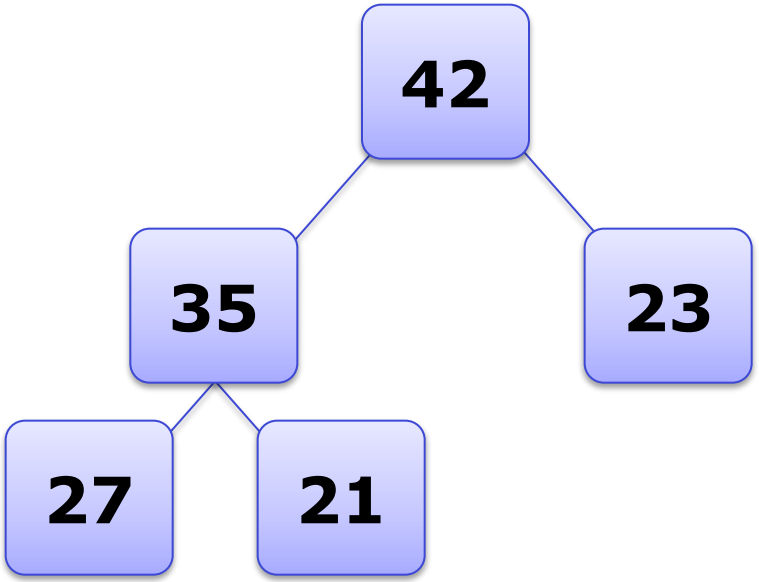


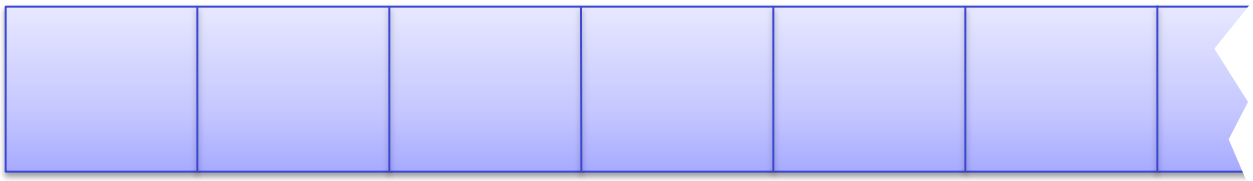
- ✓ What if we wanted to remove a node other then the top?
- ✓ Say, for example, 23?



Implementing a Heap

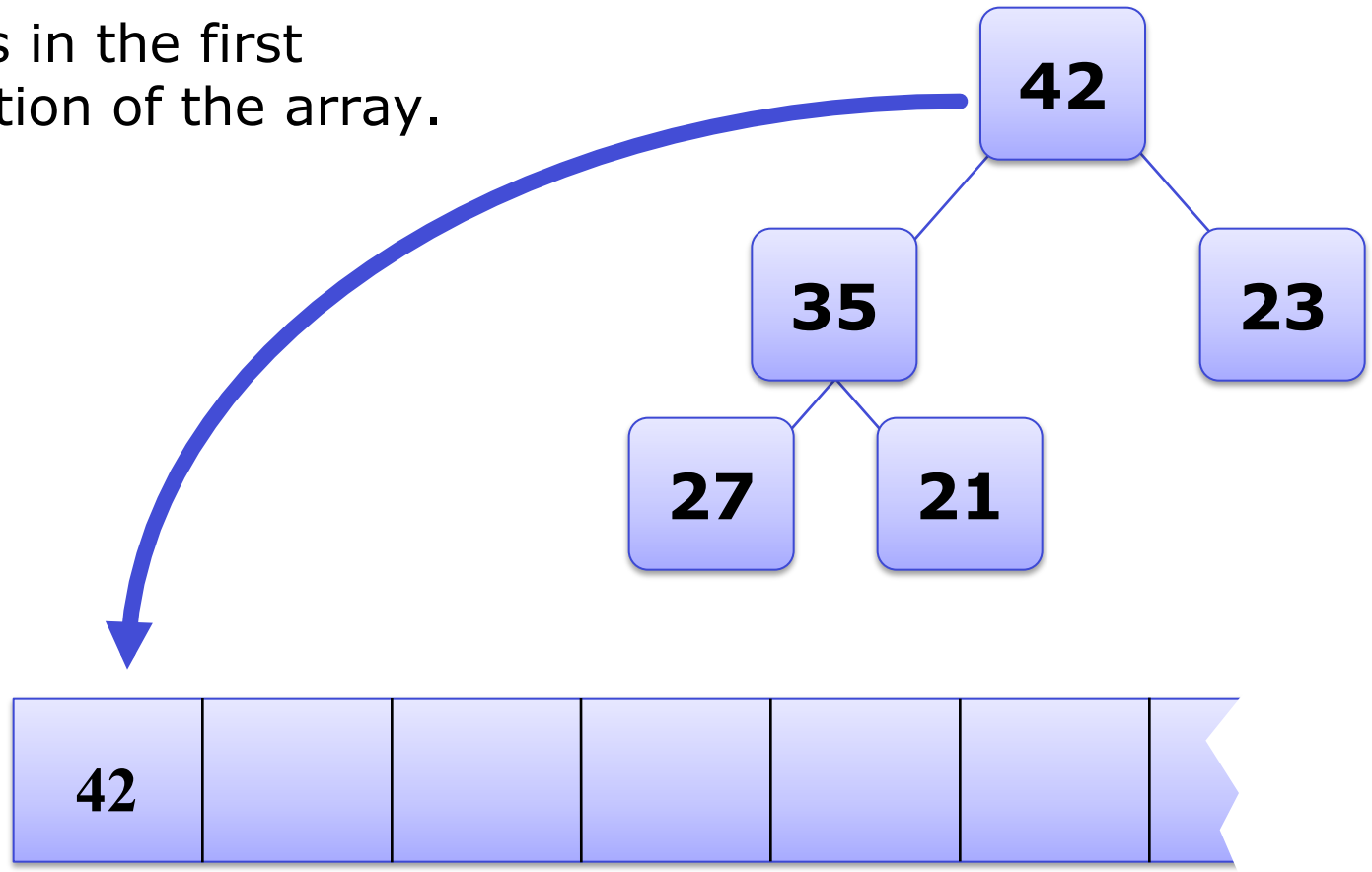
- Store the data from the nodes in a partially-filled array.





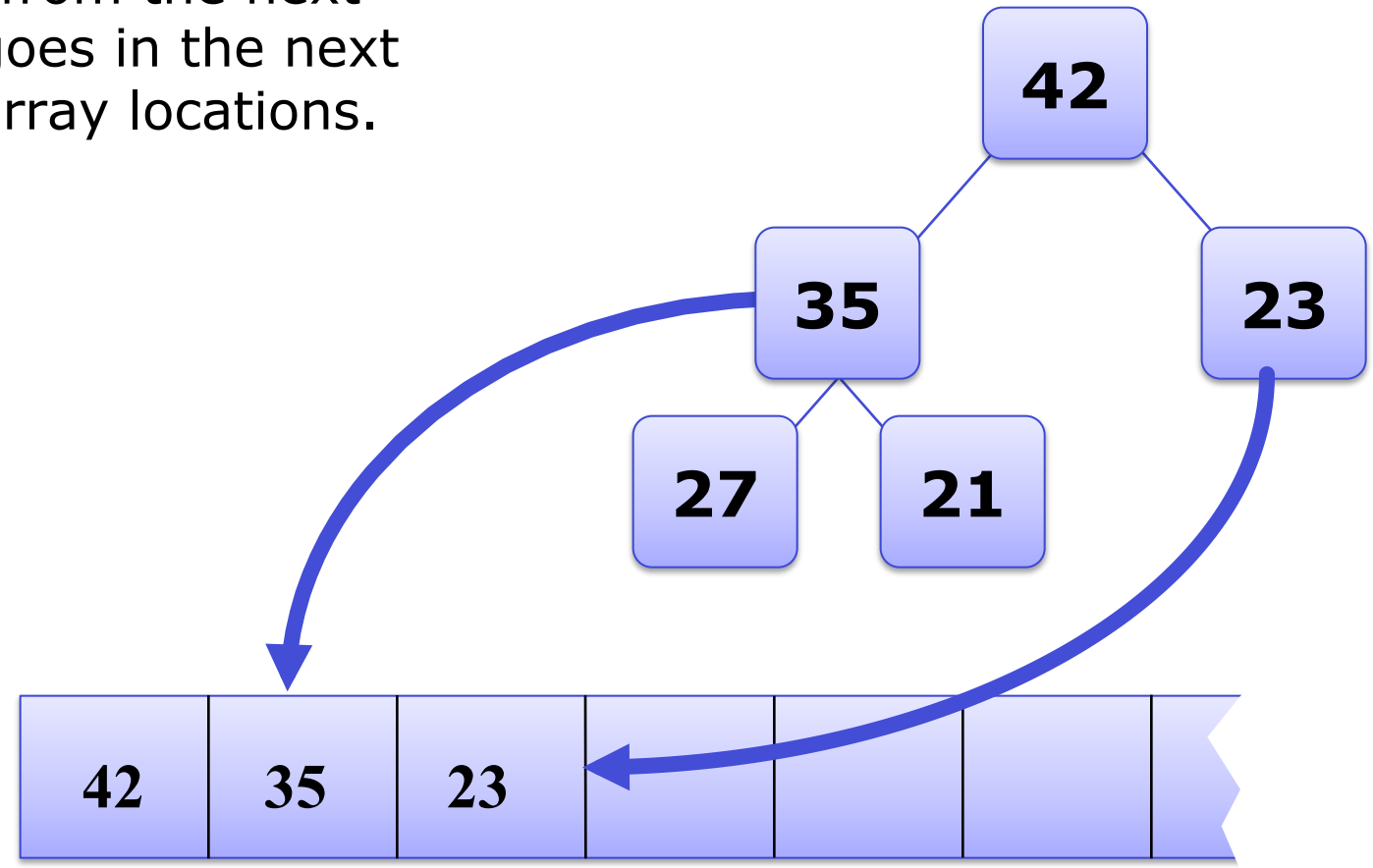
An array of data

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



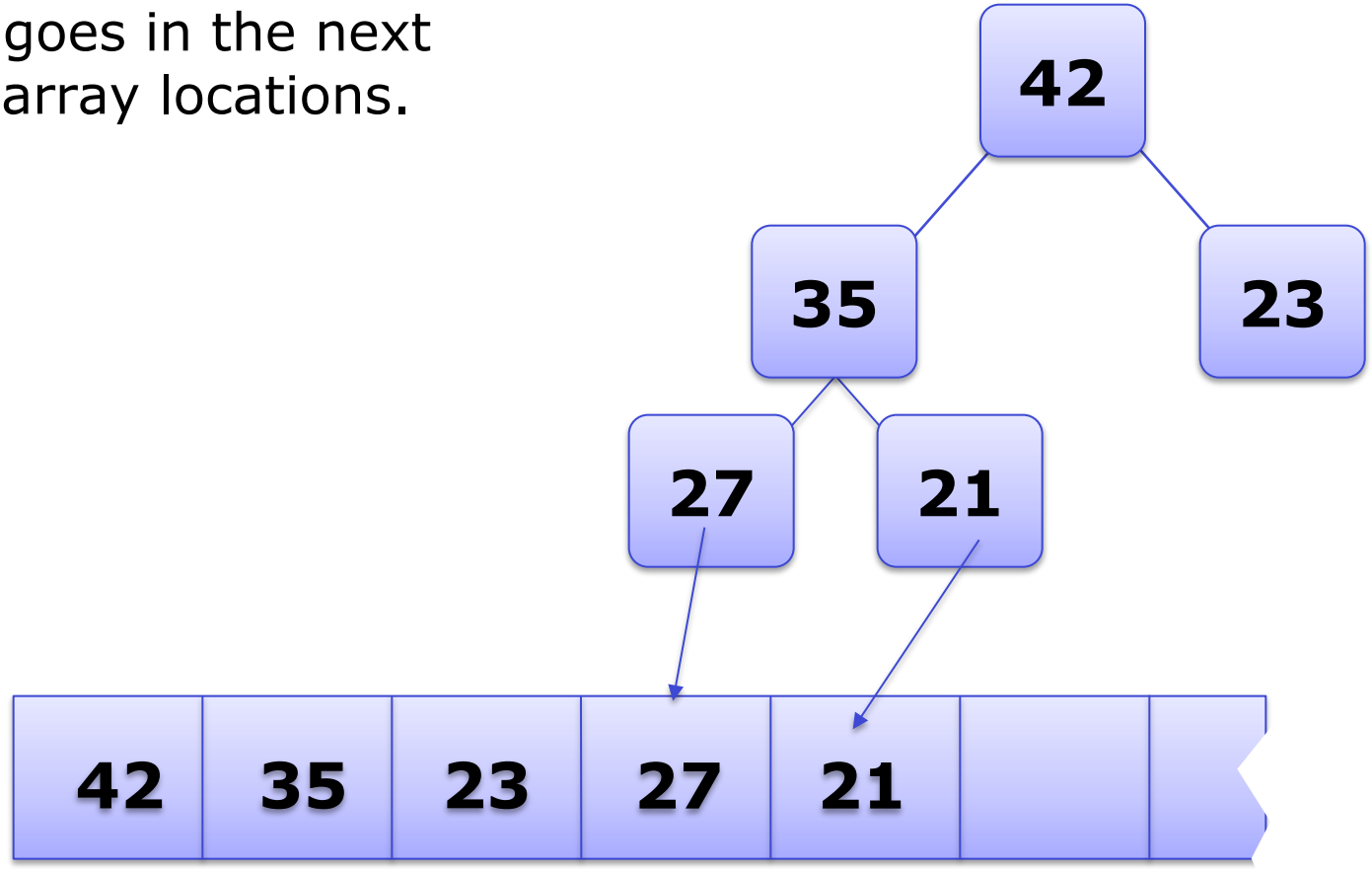
An array of data

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



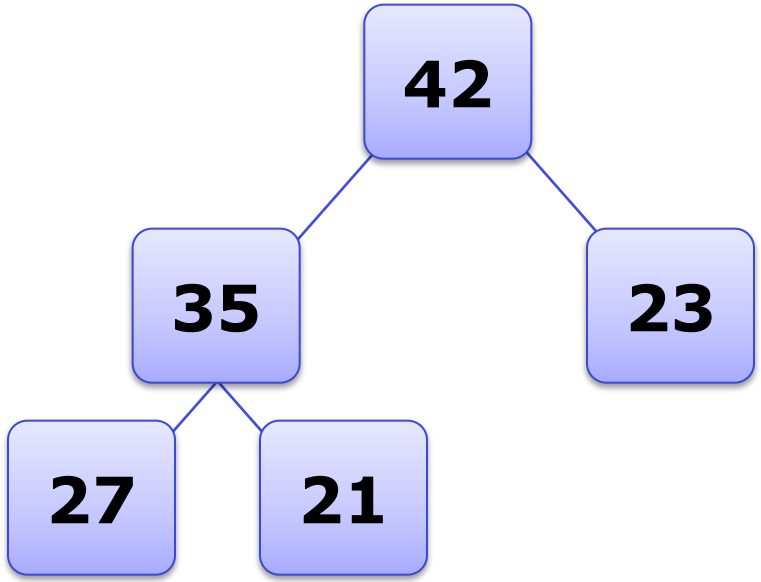
An array of data

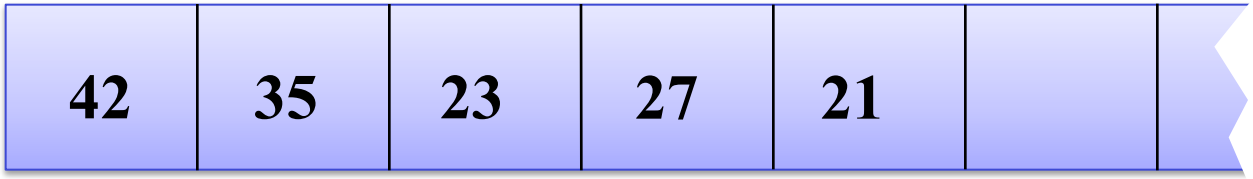
Data from the next row goes in the next two array locations.



An array of data

- The links between the tree's nodes are not actually stored as pointers, or in any other way.
- The only way we "know" that "the array is a tree" is from the way we manipulate the data.





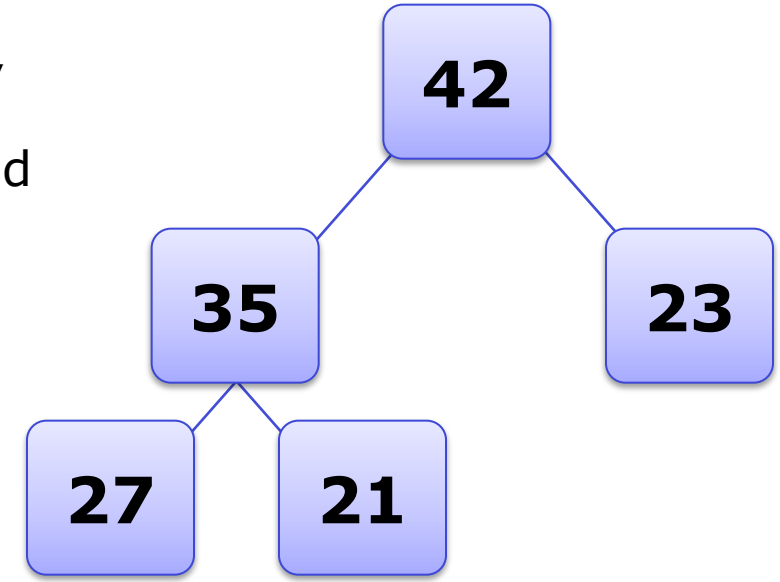
An array of data

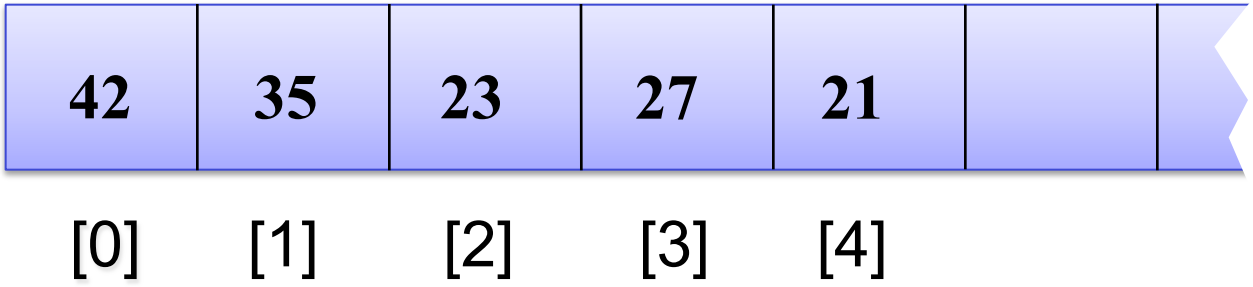
If you know the index of a node, then it is easy to figure out the indexes of that node's parent and children.

if i = location of current node:

location of left(i) = 2i + 1

location of right(i) = 2i + 2





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

if i = location of current node:

location of left(i) = 2i + 1

location of right(i) = 2i + 2

0	1	2
1	3	4
2	5	6
3	7	8
4	9	10
5	11	12
6	13	14
7	15	16
8	17	18
9	19	20
10	21	22
11	23	24
12	25	26
13	27	28
14	29	30
15	31	32
16	33	34
17	35	36
18	37	38
19	39	40
20	41	42
21	43	44
22	45	46
23	47	48
24	49	50
25	51	52
26	53	54
27	55	56
28	57	58
29	59	60
30	61	62
31	63	64

Heap Running Times

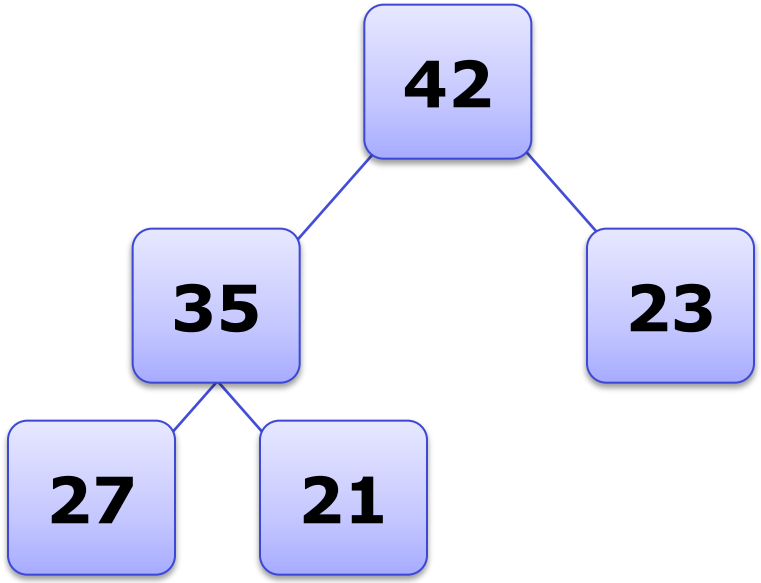
- What is the running time of each operation?
- Insert

  O(log n)
- Remove

  O(log n)

Note: Heap is NOT sorted

- It satisfies the heap property, but does not fully sort the keys.
- Heap property?



Heapsort

Given input items:

Build a heap from the input items

sortedlist = []

While the heap is not empty:

- Remove the largest item (the root), and place it at the end of sortedlist
- Re-heapify the heap

Return sortedlist.

Heapsort

- Running time is O(n log n)
- Almost as fast as quicksort, but doesn't have as bad a worst case running time
  - Quicksort worst case is O(n^2)
  - Heapsort can be done in place in one array of length n.
- Also competes with merge sort
  - Heapsort faster, but merge sort easier to parallelize