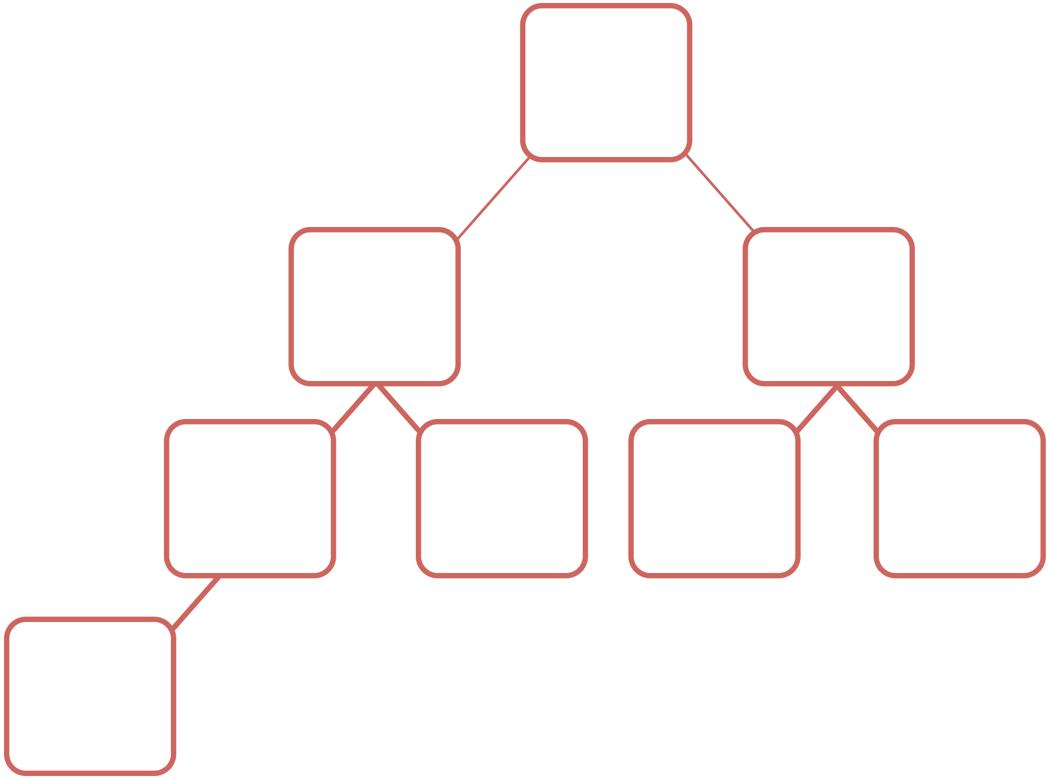
Heaps

Read Ch10.1

Heaps – Complete Binary Tree

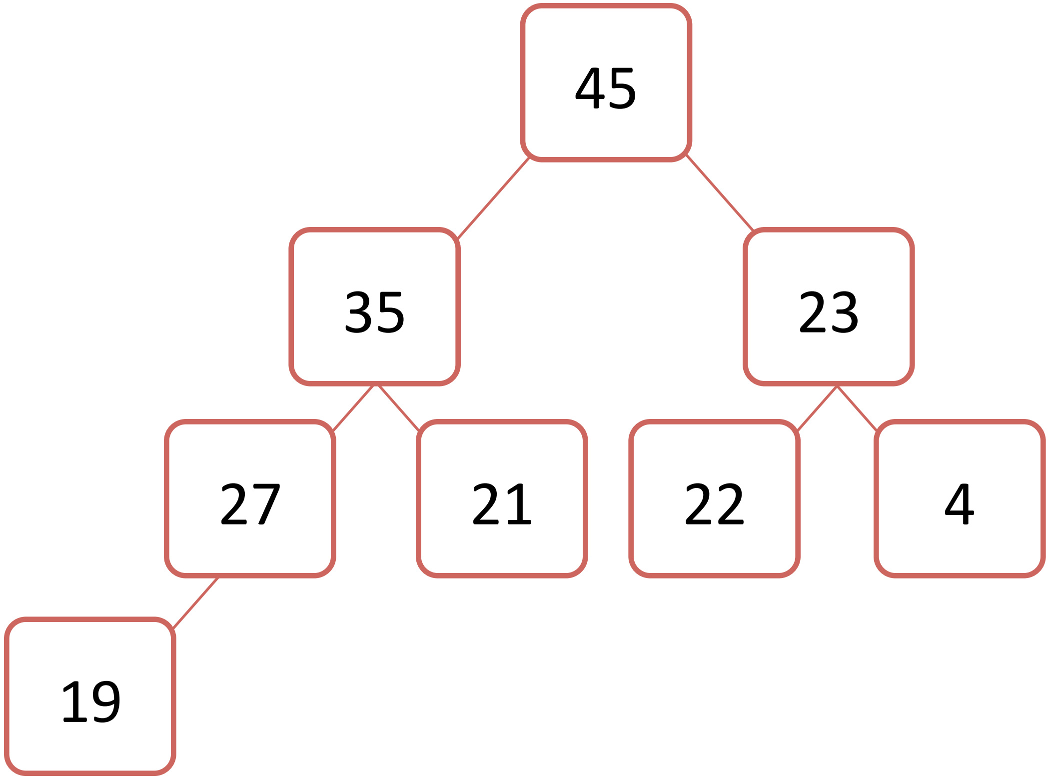
Complete binary tree.

- Level *i* has 2^i nodes $(0 \leq i \leq h-1)$
- Level *h* fill this level from left to right.



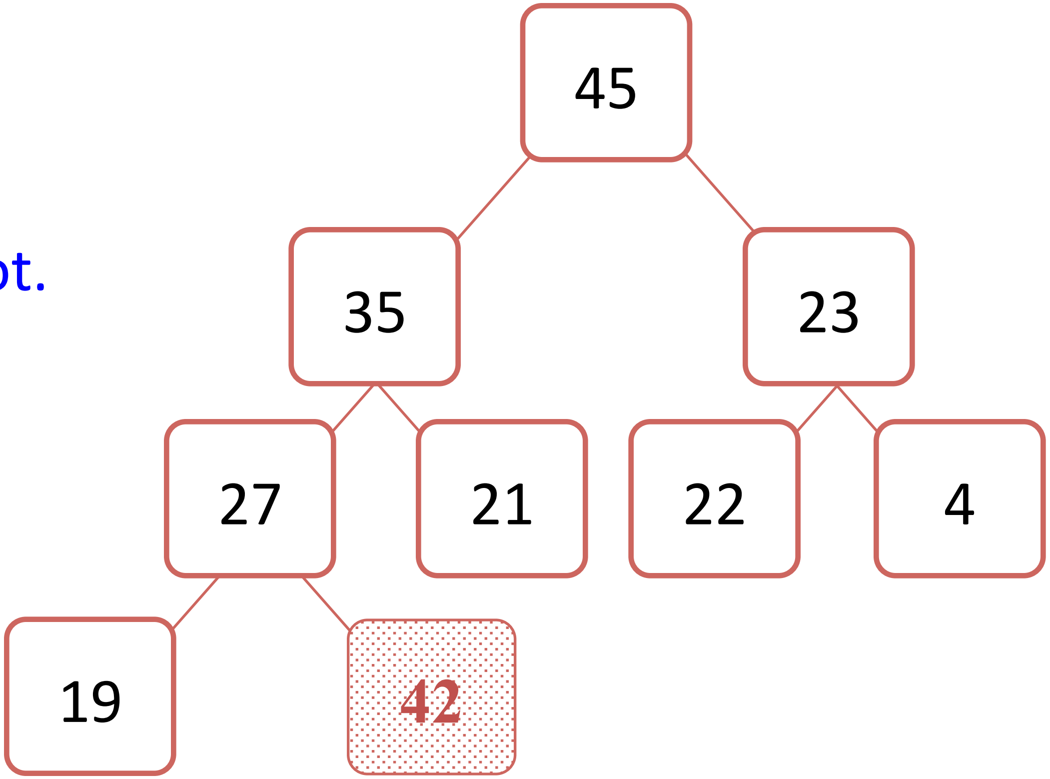
Heaps -- Heap-order property

- For every node v other than the root, the key associated with v is smaller than or equal to the key associated with v's parent.



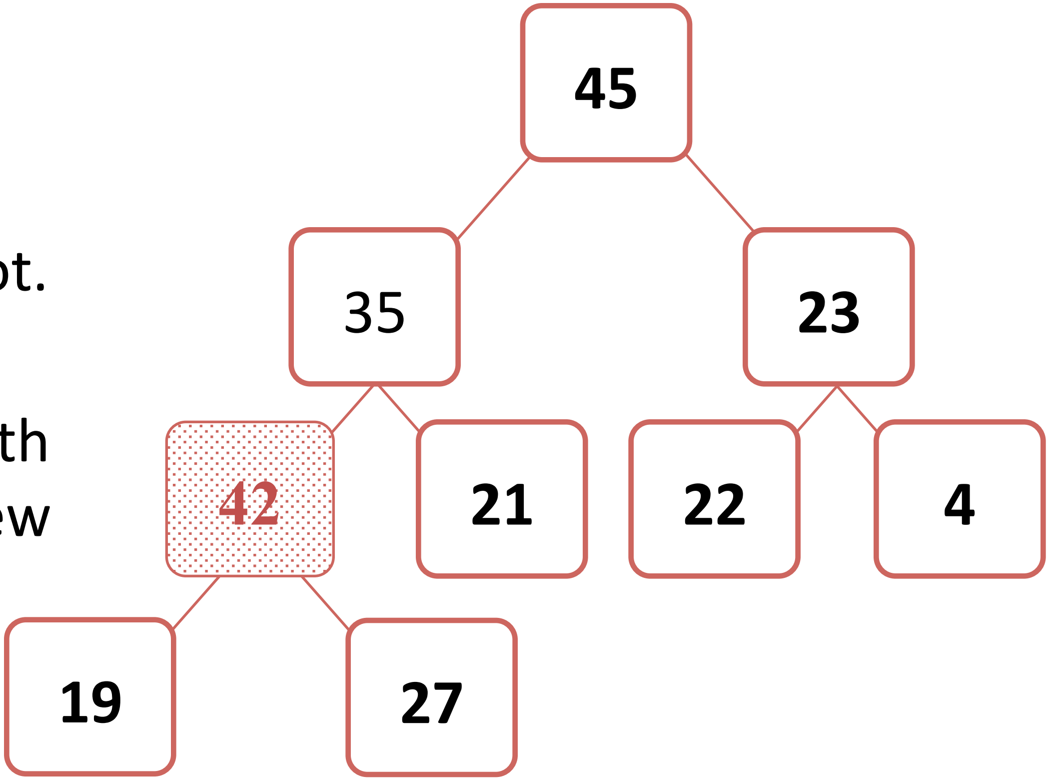
Adding a Node to a Heap

1. Put the new node in the next available spot.



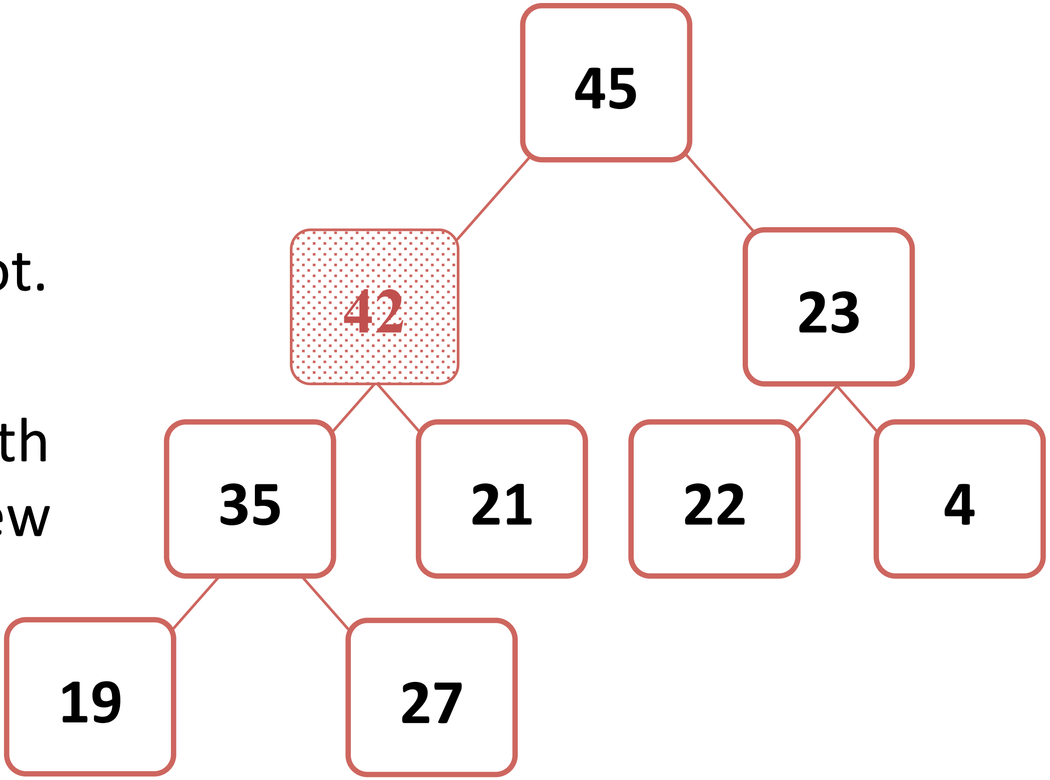
Adding a Node to a Heap

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



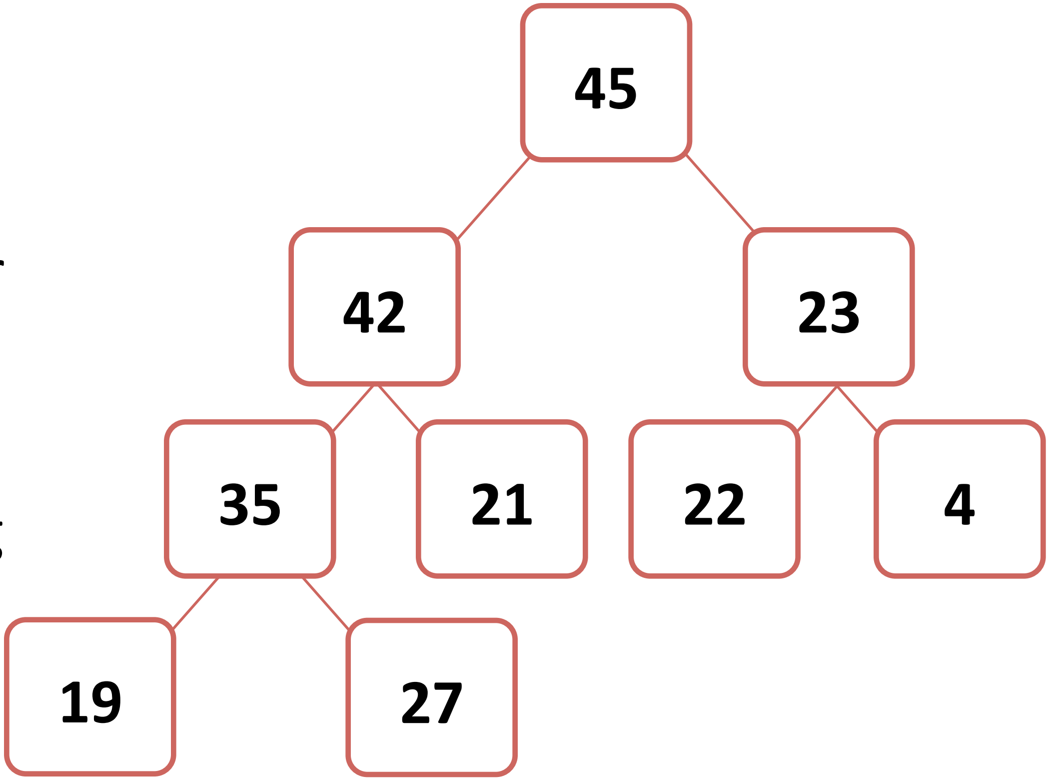
Adding a Node to a Heap

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



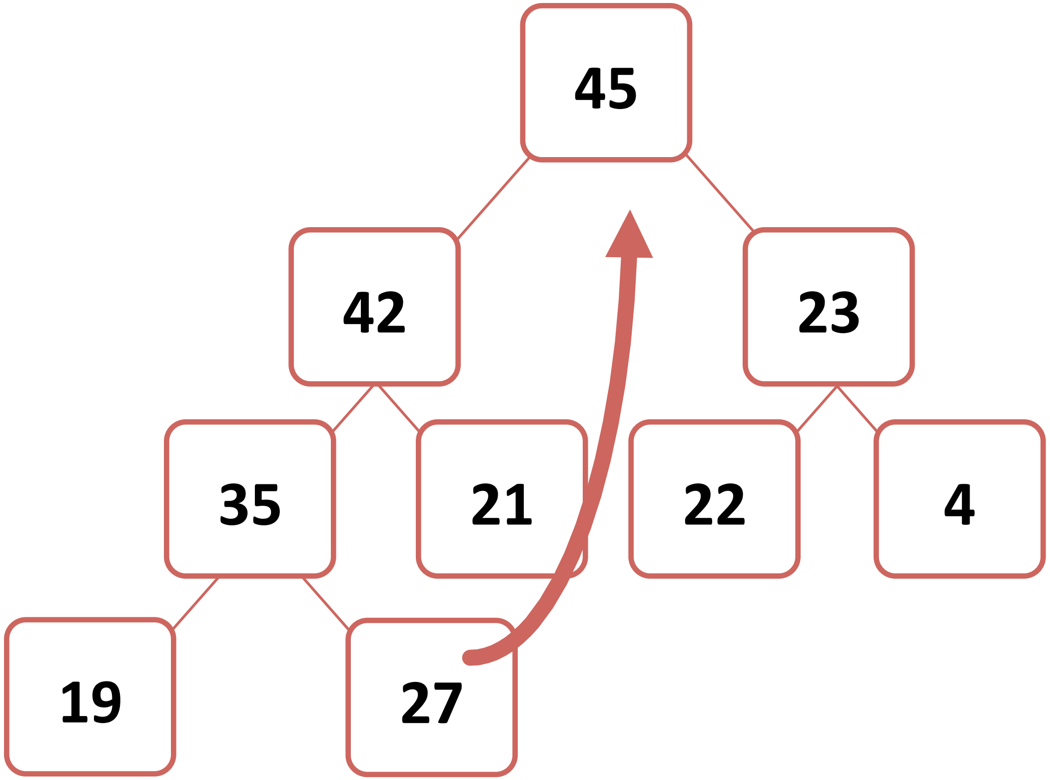
Adding a Node to a Heap

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



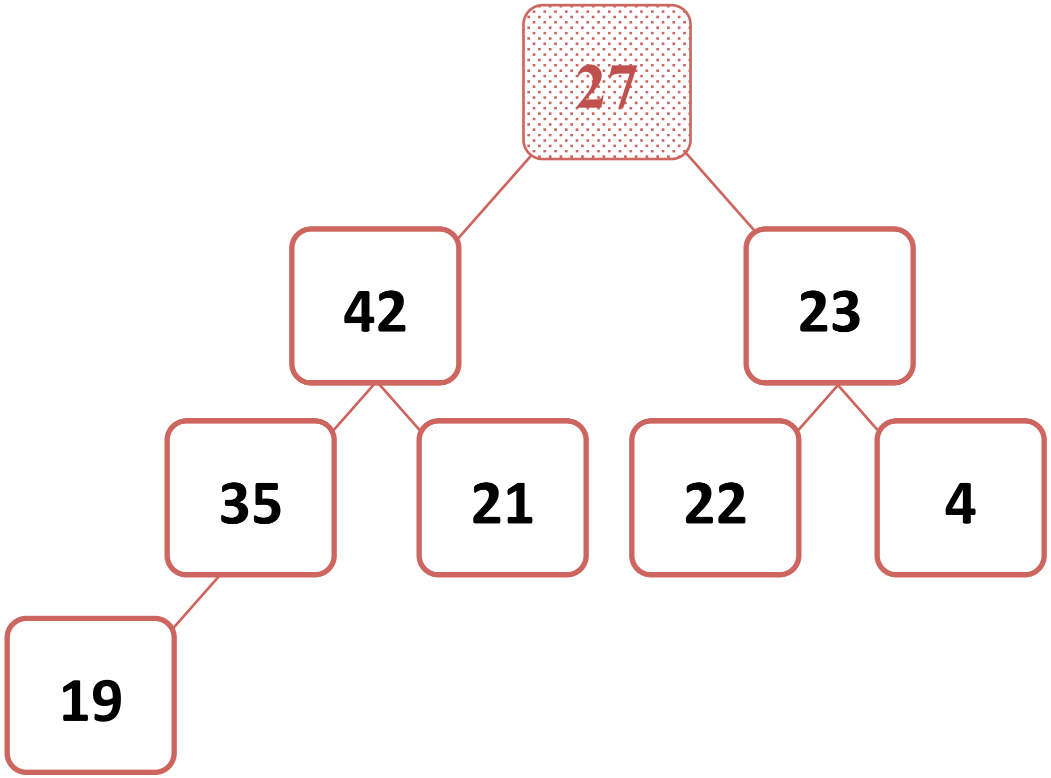
Removing the Top of a Heap

- Move the last node onto the root.



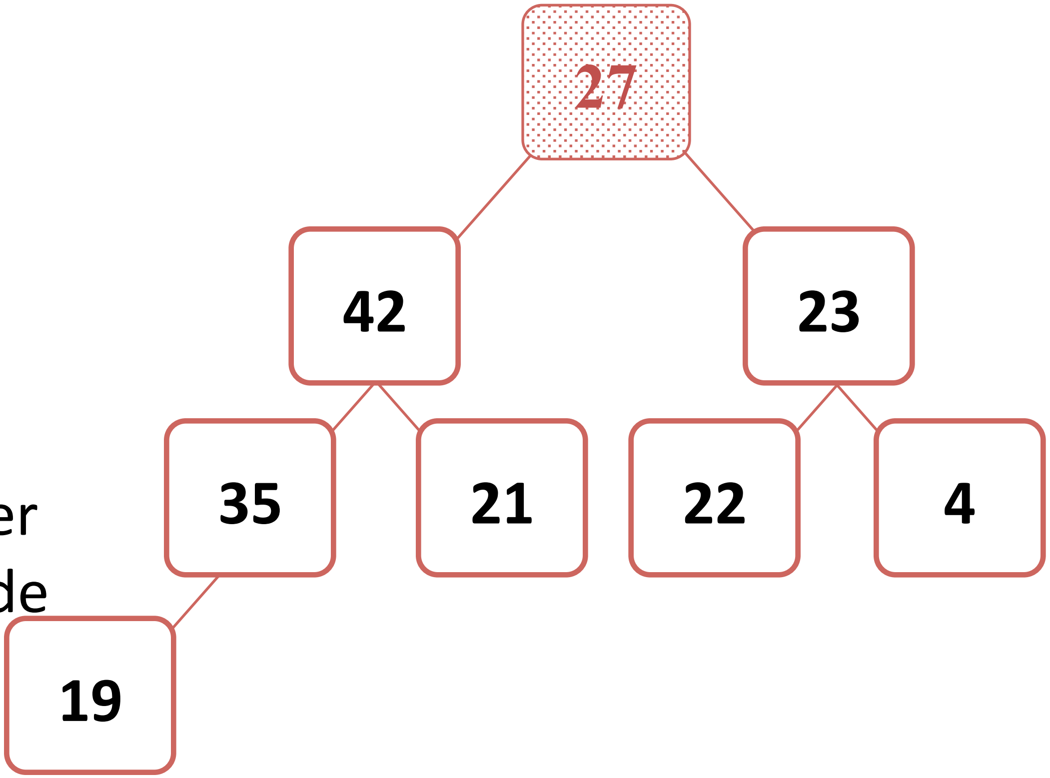
Removing the Top of a Heap

- Move the last node onto the root.



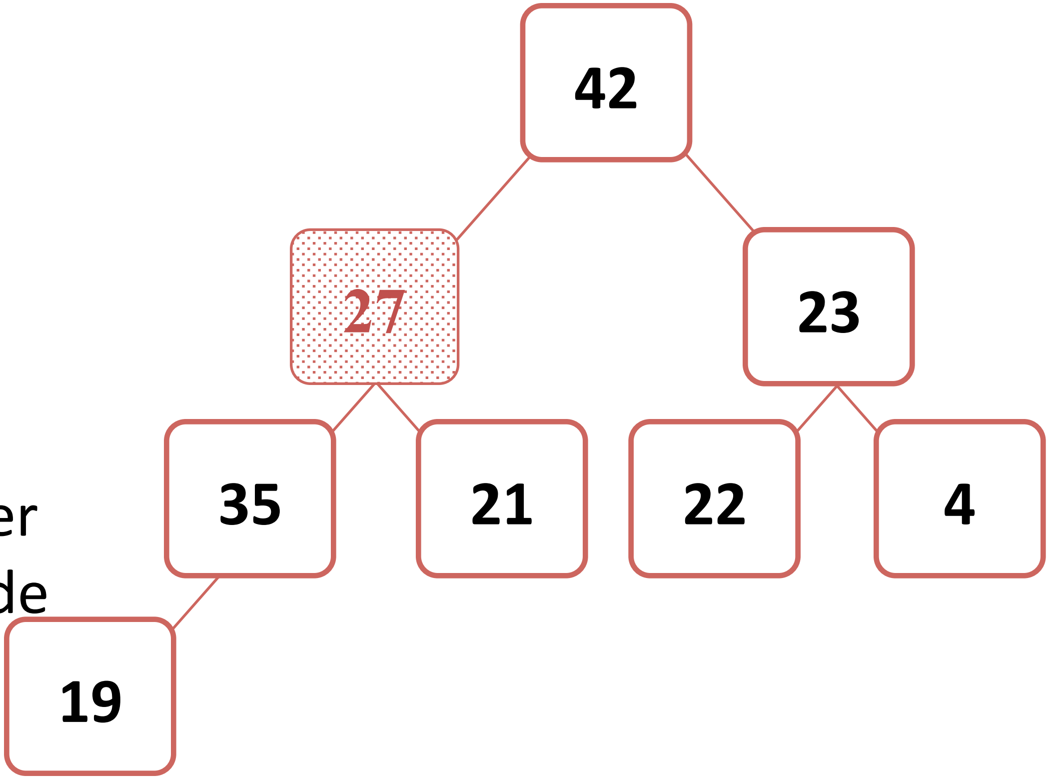
Removing the Top of a Heap

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



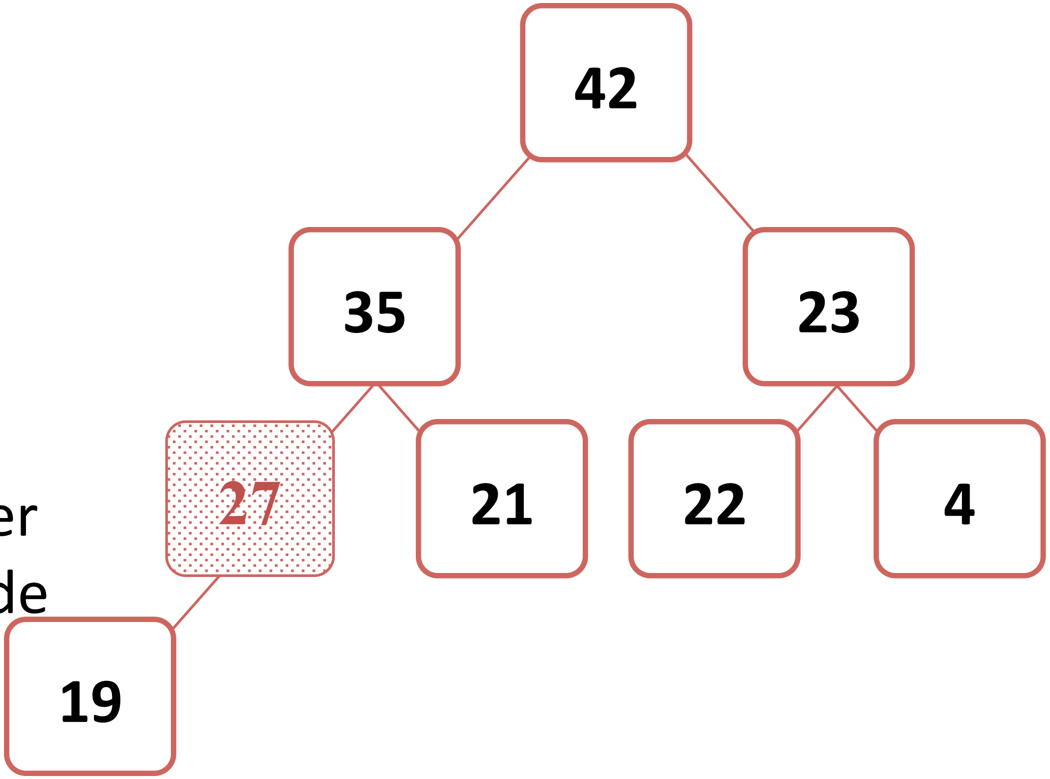
Removing the Top of a Heap

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



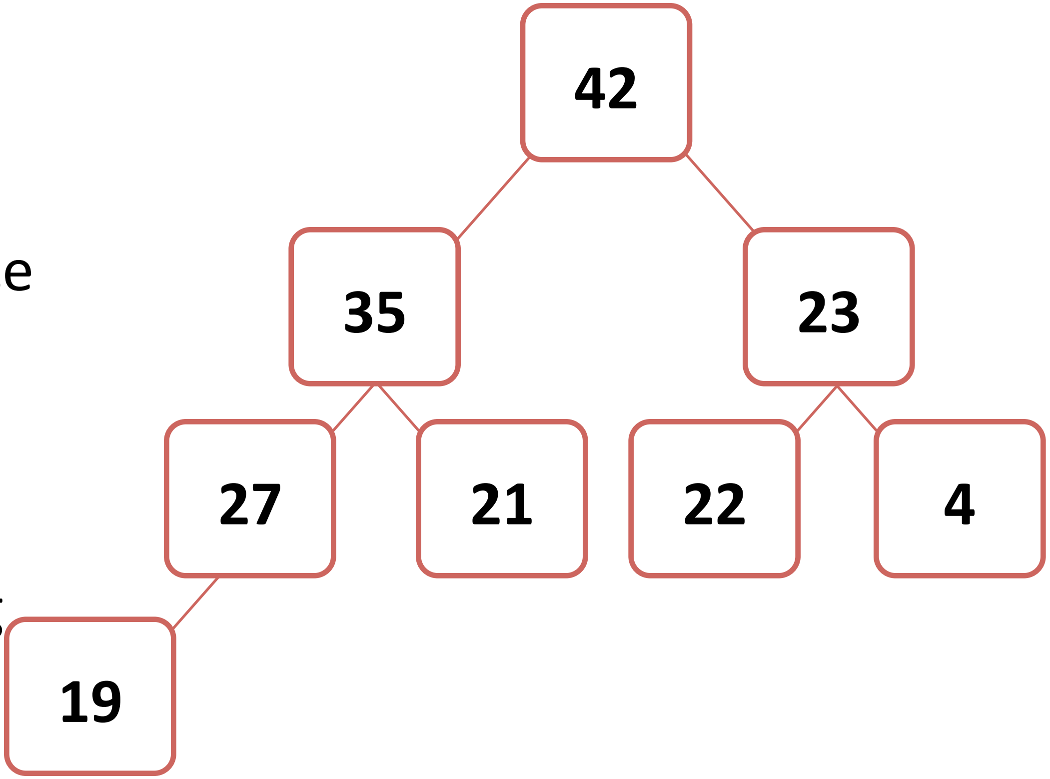
Removing the Top of a Heap

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



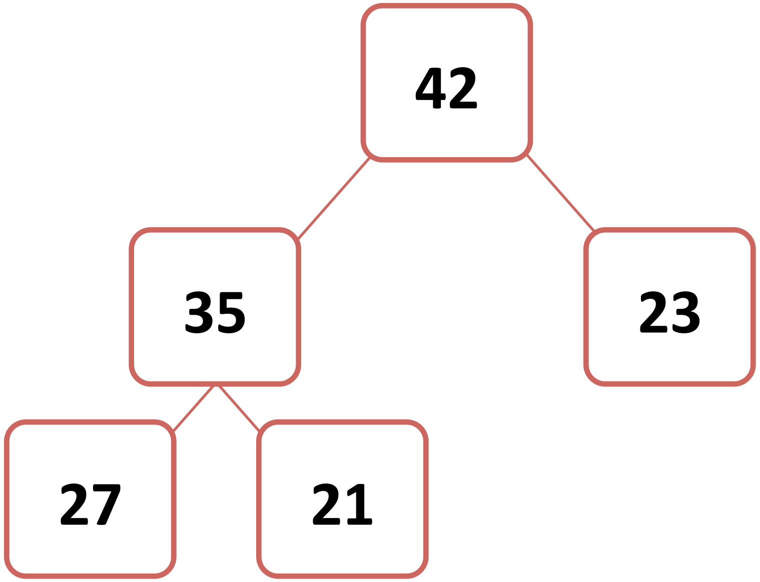
Removing the Top of a Heap

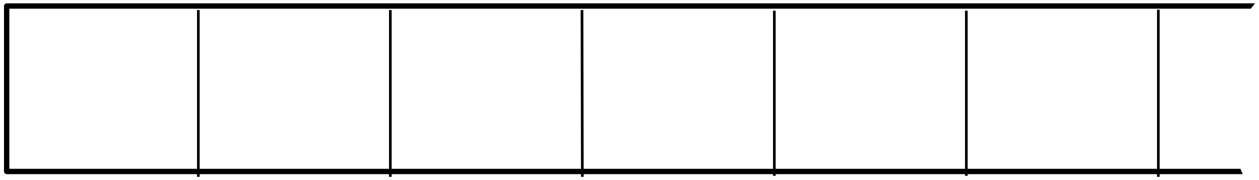
- ✓ The children all have keys <= the out-of-place node, or
- ✓ The node reaches the leaf.
- ➘ The process of pushing the new node downward is called **reheapification downward**.



Implementing a Heap

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

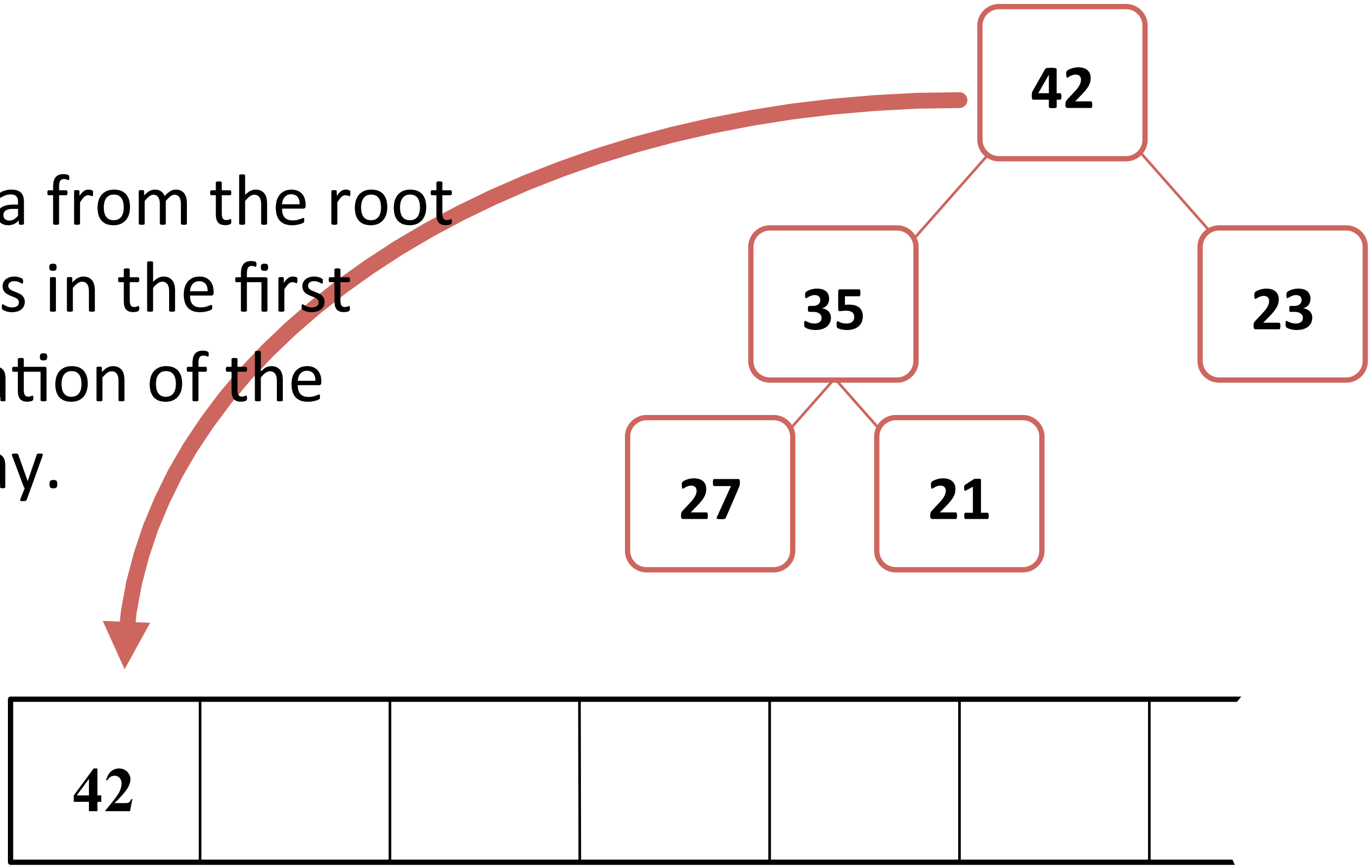




An array of data

Implementing a Heap

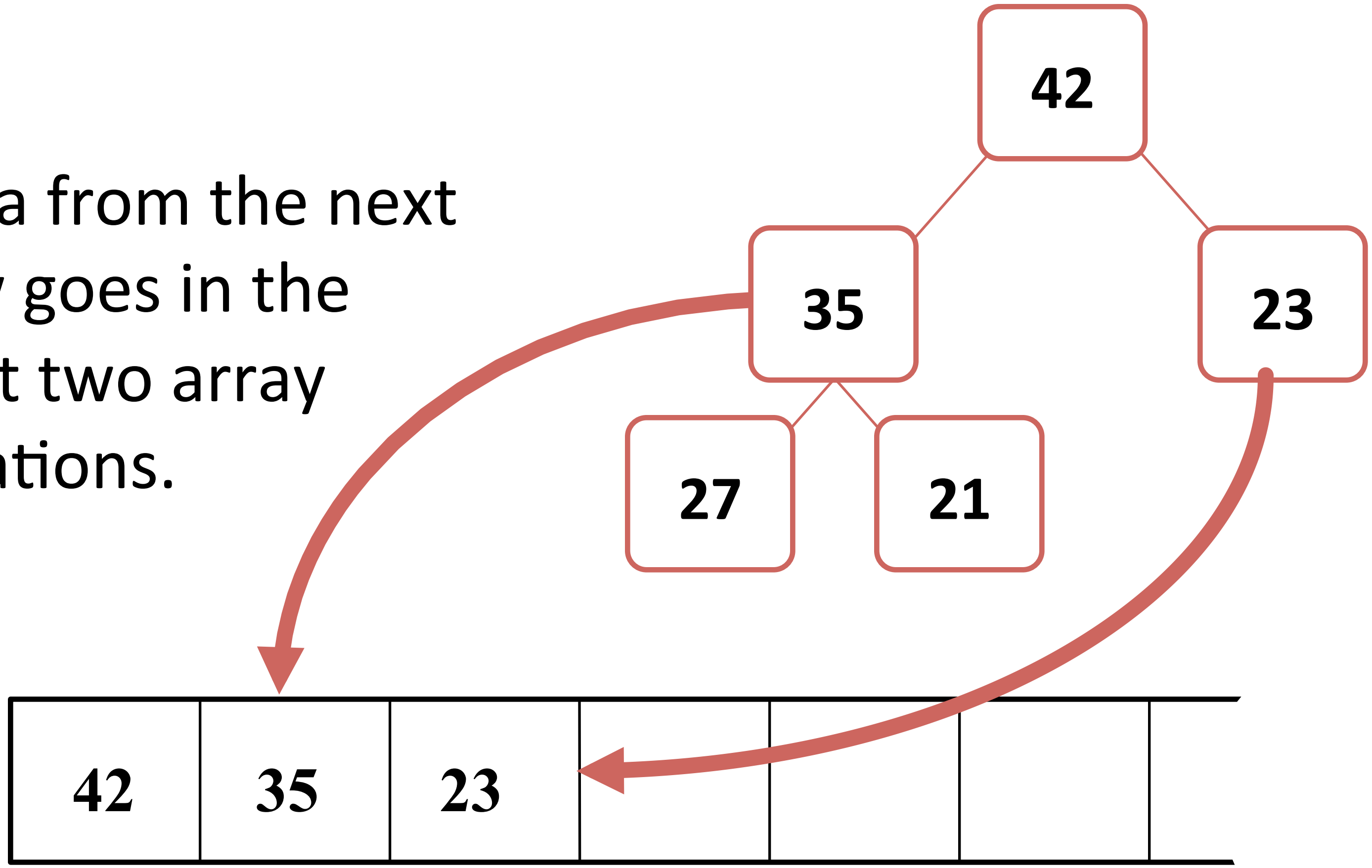
- Data from the root goes in the first location of the array.



An array of data

Implementing a Heap

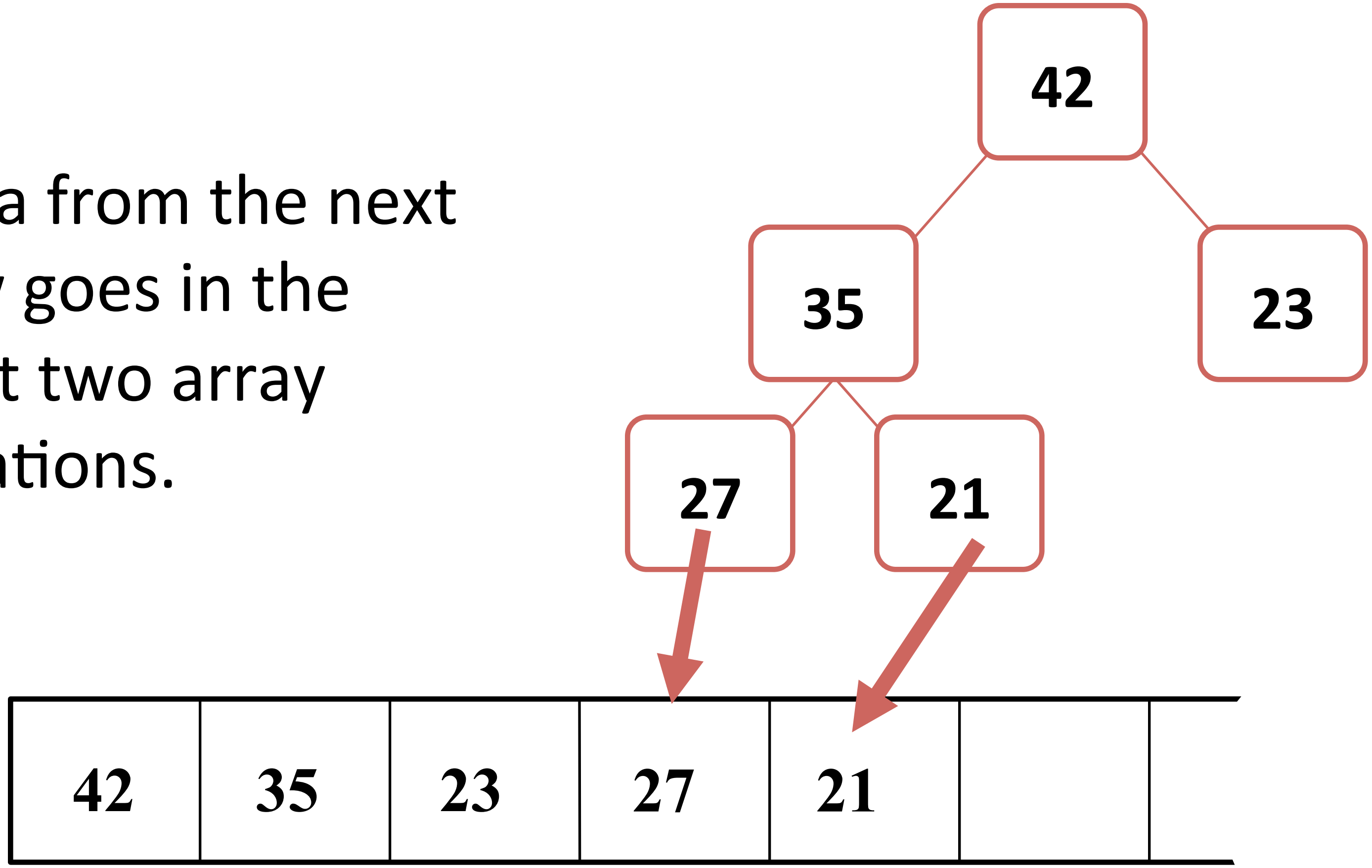
- Data from the next row goes in the next two array locations.



An array of data

Implementing a Heap

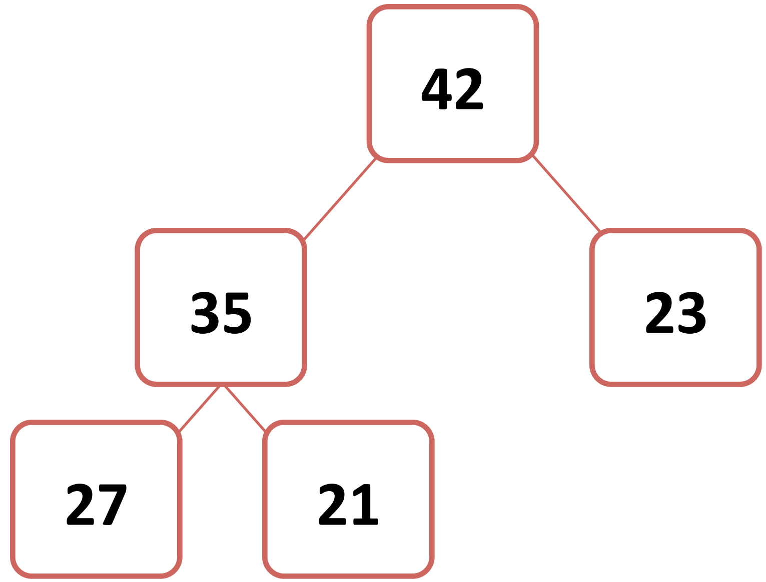
- Data from the next row goes in the next two array locations.

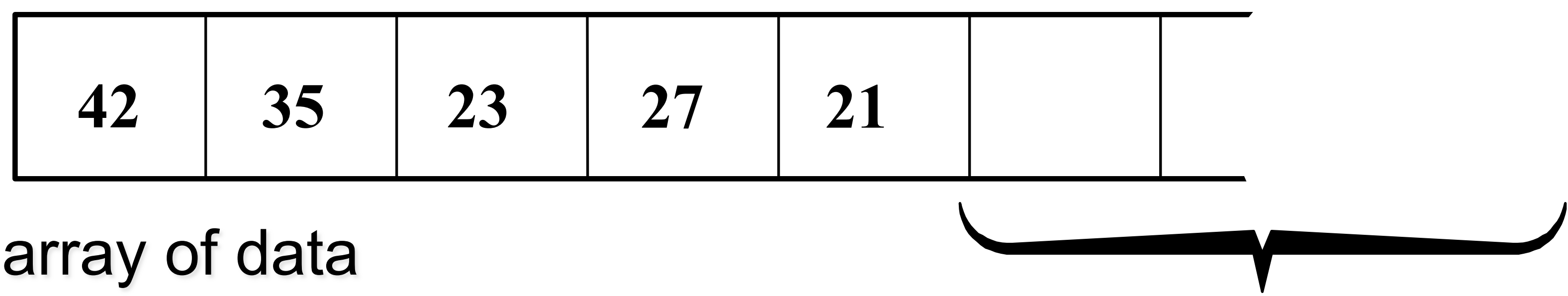


An array of data

Implementing a Heap

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

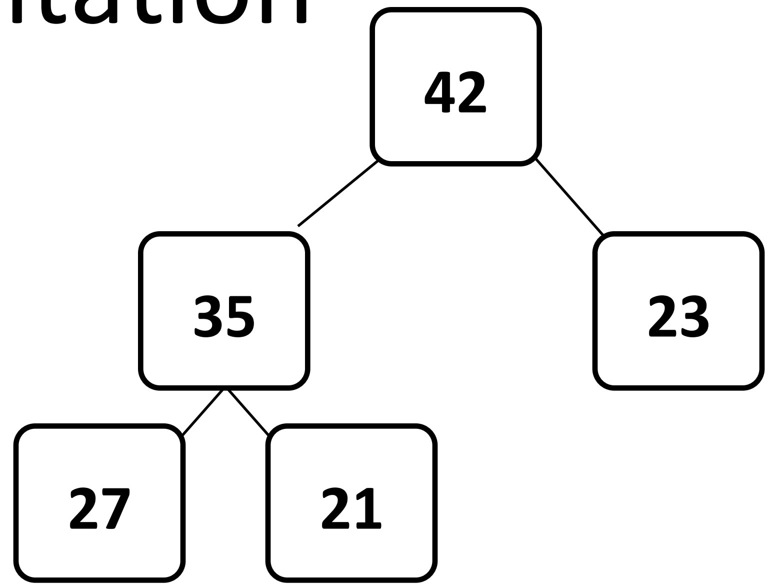




An array of data

What is the next available spot?

Important Points about the Implementation



- Left child of [i] = [2i+1]
- Right child of [i] = [2i+2]

- Parent of [i]= [floor((i-1)/2)]

42	**35**	**23**	**27**	**21**		
[0]	[1]	[2]	[3]	[4]		

void add(int e)

- 1. add e as the last element to the array/ arrayList
- 2. Push up the element until the heap property is satisfied
  - reheapUpward(int pos)

reheapUpward(pos)

- //pos: the position of element that violates the heap
- Base case: pos<=0
- Recursive
- 1. get the position of the parent, parentPos
- 2. if array[pos]>array[parentPos]
  - 2.1 Swap array[pos] and array[parentPos]
  - 2.2 Recursively reheapUp(parentPos)

int remove()

- 1. the answer is the first element, array[0]
- 2. if array contains more than one element
  - Put the last element to array[0]
  - Remove the last element
  - reheapDownward(int pos=0)

Time complexity

- Top
- Add
- Remove

The height of a heap

- A heap storing n entries has height h
  - Where h= floor (logn)
- Justification
  - Let h denote the height (# of levels) of the heap
  - For each each level i ($0 \le i \le h-1$),
    - The minimum number of nodes in the last level is 1
    - The maximum number of nodes in the last level is 2^{h-1},
    - The number of nodes in other levels 2^i
  - The total number of nodes
    - Minimum: $(1+2+\ldots+2^{h-2})+1 = (2^{h-1}-1)+1 = 2^{h-1}$
    - Maximum: $(1+2+\ldots+2^{h-2})++2^{h-1} = 2^h-1$
  - Thus, $2^{h-1} \le n \le 2^h-1$
  - Solving the above equation, we get $\log_2 (n+1) \le h \le 1+\log_2 n$
  - In Big-O: h is $O(\log_2 n)$

Heap

- Priority Queue
  - http://docs.oracle.com/javase/6/docs/api/
- What about implementing a heap using a binary tree?

Summary

- A heap is a complete binary tree, where the entry at each node is greater than or equal to the entries in its children.
- To add an entry to a heap, place the new entry at the next available spot, and perform a reheapification upward.
- To remove the biggest entry, move the last node onto the root, and perform a reheapification downward.