CIT-237

More about Binary Trees

December 11, 2019

NOTICE to Students: Portions of these lecture slides are
Copyright 2018 Pearson Education, Inc.
We have a license to use this material ***only*** in the context of this course.
There is NO PERMISSION to share these lecture slides with anyone not taking the course.
There is NO PERMISSION to post these lecture slides on the Internet.

Reminder / Announcement

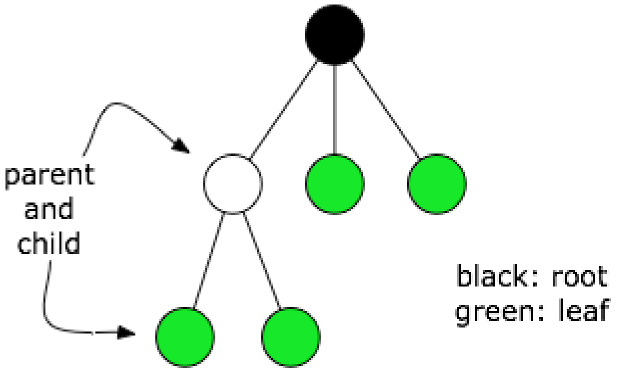
- The EXTENDED Due date for Project 3 was Monday, December 9.
- If you are still having difficulty completing Project 3, ask for help during the Lab portion of today's class.
- Our last day of class is Monday, December 16.
  - Obviously, that is the last day to demonstrate the solutions to Lab Exercises.
  - We will NOT have a Final Exam.

Tree Data Structures

- In a previous class, we introduced the concept of a Tree data structure.

tree (data structure): A data structure accessed beginning at the *root* node. Each *node* is either a *leaf* or an *internal node*. An internal node has one or more *child* nodes and is called the *parent* of its child nodes. All children of the same node are *siblings*. Contrary to a physical tree, the root is usually depicted at the top of the structure, and the leaves are depicted at the bottom.

(Source: https://xlinux.nist.gov/dads/HTML/tree.html)



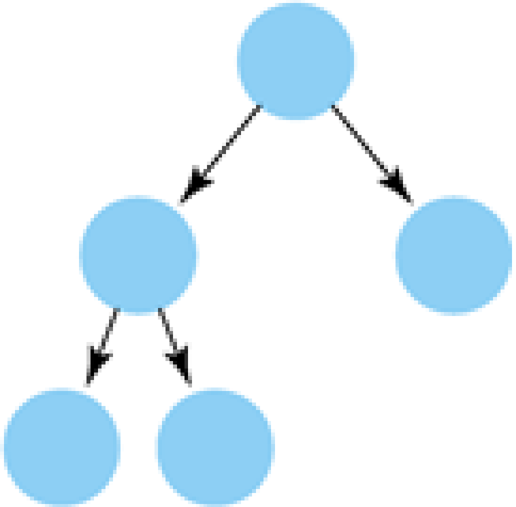
Binary Tree

- If each node in a tree is assumed to have no more that two child nodes, it is called a *binary tree*:

binary tree (data structure): A *tree* with at most two *children* for each *node*.

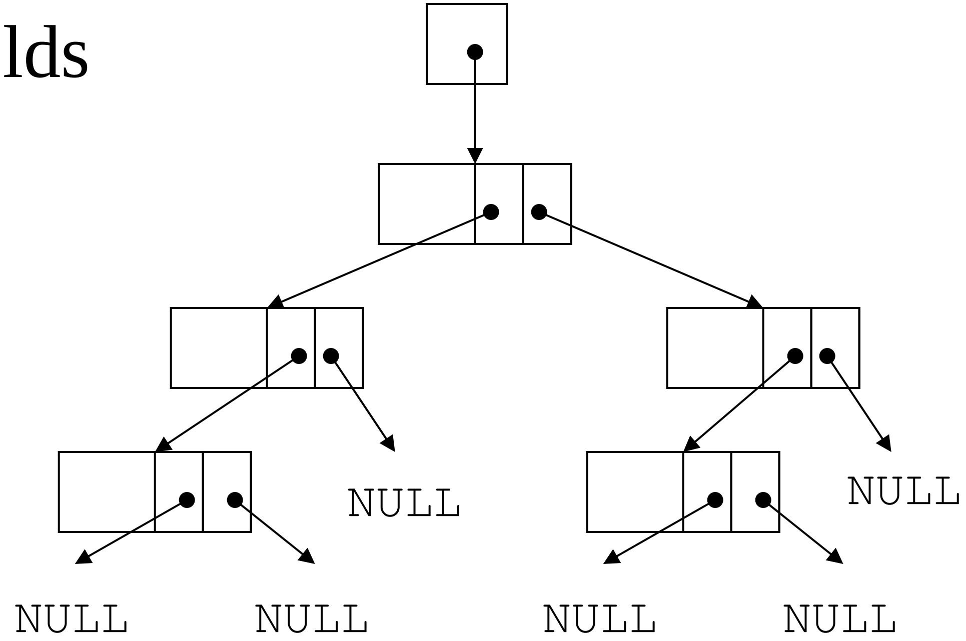
Formal Definition: A binary tree either is empty (no nodes), or has a *root* node, a left binary tree, and a right binary tree.

(Source: https://xlinux.nist.gov/dads/HTML/binarytree.html)



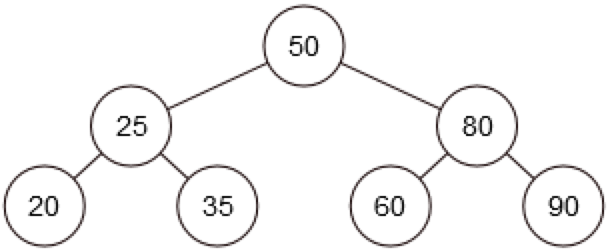
Linked-Node Binary Trees

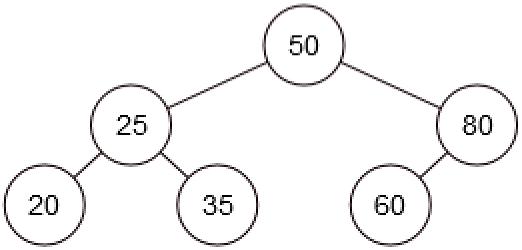
- Binary tree: a nonlinear linked structure in which each node may link to 0, 1, or 2 other nodes
- Each node contains:
  - one or more data fields
  - two pointers



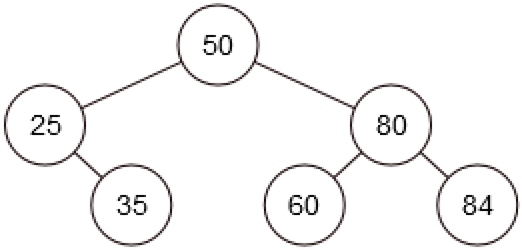
Complete Binary Tree

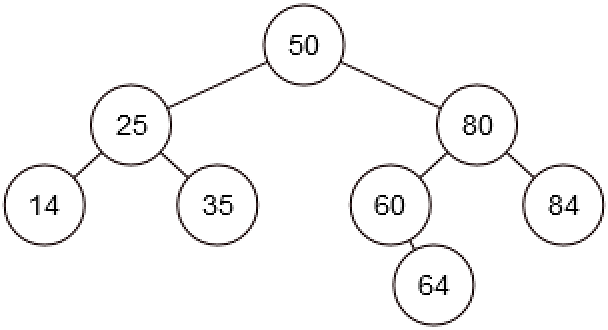
- A binary tree is *complete* if every level of the tree is full, or
- If the last level is not full, all the nodes on the last level are placed left-most.
- For example, these two example trees are *complete*:





- But these two are <u>not</u> *complete*:





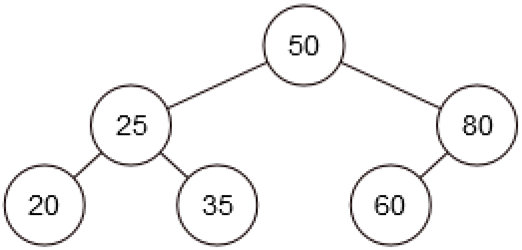
Complete Binary Tree Representation (1)

- Previously, we looked at binary trees represented by linked objects.
- An alternate representation of a <u>complete</u> binary tree, using an array, is sometimes preferable:
  - It saves space.
  - It provides an easier way for a program to find the <u>parent</u> node of any particular node.

Complete Binary Tree Representation (2)

- While our conceptual picture of any binary tree is as a tree, we can choose to represent a *complete* binary tree using an array.

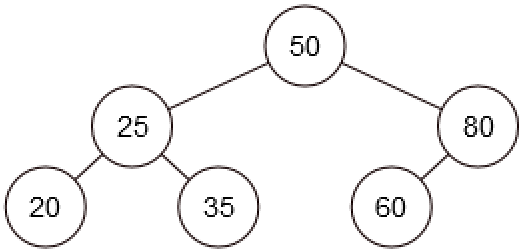
For example, the tree shown below:

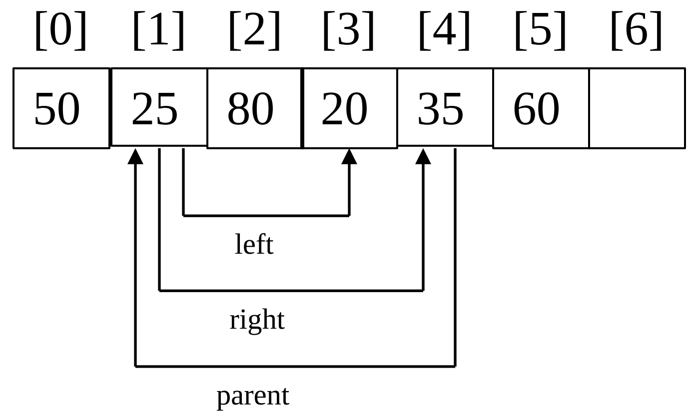


can be represented by the following array:

[0]	[1]	[2]	[3]	[4]	[5]	[6]
50	25	80	20	35	60	

Complete Binary Tree Representation (3)

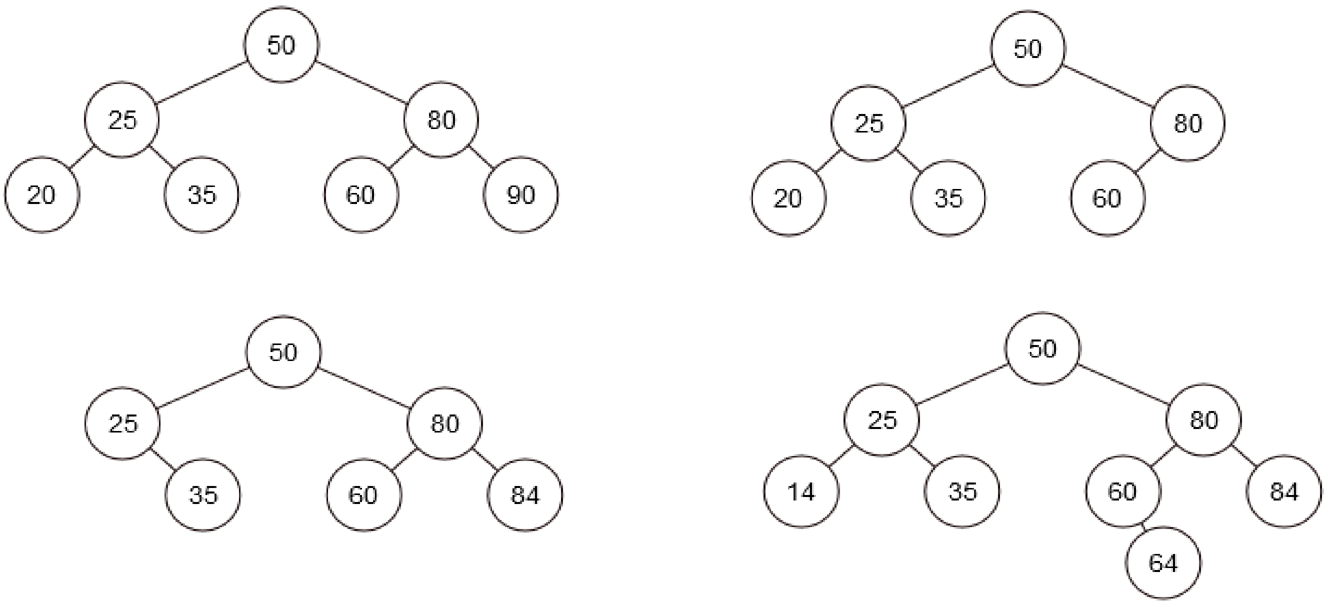




- For a node at position *i*:
  - its left child is at position *2i+1*
  - its right child is at position *2i+2*
  - its parent is at index (*i-1)/2* (Integer division – remainder is discarded.)
- For example: The node for element 25 is at position 1.
  - its left child (element 20) is at position 3 (*2*1+1*)
  - its right child (element 35) is at position 4 (*2*1+2*)
  - Element 25 is the parent of element 35: position of element 35 is 4. The parent of the element at position 4 is at position (4-1)/2 = 1.

Binary Search Trees

- We have seen a common use of a binary tree: a *binary search tree*.
  - Any node in the *left* subtree of a node contains data that is "less than" the data in the current node.
  - Any node in the *right* subtree of a node contains data that is "greater than" the data in the current node.
- All of the trees shown below are examples of binary search trees:



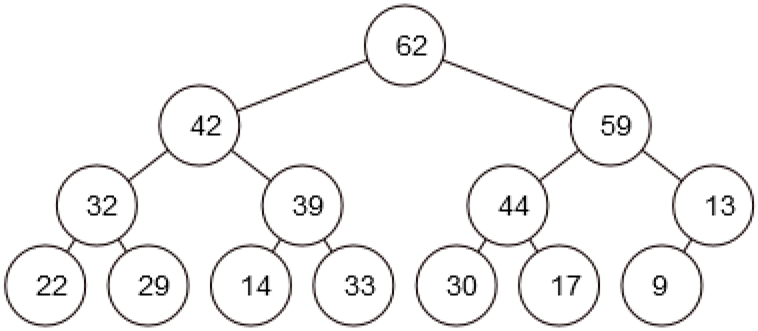
- But *not all* binary trees are binary search trees.

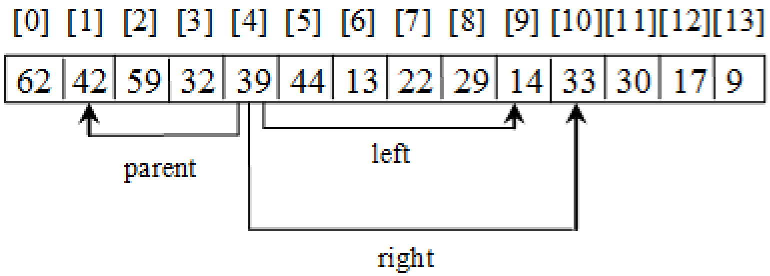
Binary Heap

- The term ***heap*** has multiple meanings:
  - In everyday life, a heap might be a disorganized jumble of items, such as a "heap" of clean, unsorted laundry.
  - We refer to the region of memory where dynamically allocated objects are created as a "heap".
  - In Data Structures, a heap is a special kind of binary tree.
- A *binary heap* is a binary tree with the following properties:
  - It is a complete binary tree.
  - If it is a "max-heap", then the data in each node is greater than or equal to the data in any of its children. If it is a "min-heap", then the data in each node is less than or equal to the data in any of its children. (The examples in this lecture are all "max-heap" examples.)

Representing a Heap

- For a node at position *i*:
  - its left child is at position *2i+1*
  - its right child is at position *2i+2*
  - its parent is at index (*i-1)/2*.
- In the example below: the node for element 39 is at position 4.
  - its left child (element 14) is at position 9 (*2*4+1*)
  - its right child (element 33) is at position 10 (*2*4+2*)
  - its parent (element 42) is at position 1 (*(4-1)/2*).
- This particular example is a ***max-heap***: the data in any particular node is greater than or equal to the data in each of the child nodes.





Modifying a Heap

- Whenever a node is added to a binary heap, or removed from a binary heap, the structure may need to be adjusted to restore the *max-heap* or *min-heap* rules:
  - Scan the heap to find any nodes that break the *max-heap* (or *min-heap*) rule.
  - Swap the two offending values so that they obey the rule.
  - Continue scanning and swapping until the entire tree obeys the heap rules.

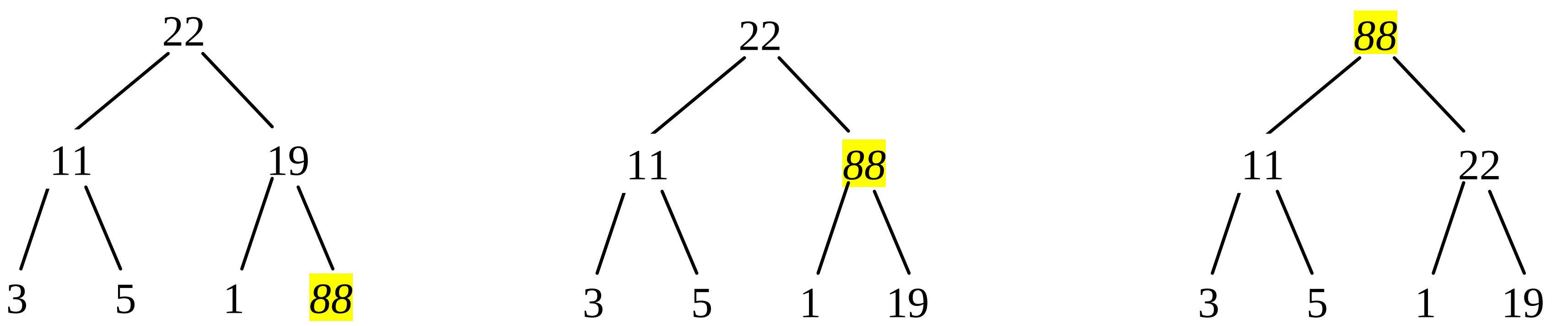
- Adding a node:
  - The new node is added at the bottom of the tree, so that the tree is still a *complete* binary tree.
  - The tree is scanned / adjusted starting at the bottom.

- Removing a node:
  - The node removed is always the **root** node.
  - The right-most node of the bottom row is temporarily moved to the root position.
  - The tree is scanned / adjusted starting at the root.

Details of the Scan / Adjust Process

Each time a node is added, the scan / adjust process moves the new value to the correct position.

In this example, the value 88 is added to an existing max-heap.

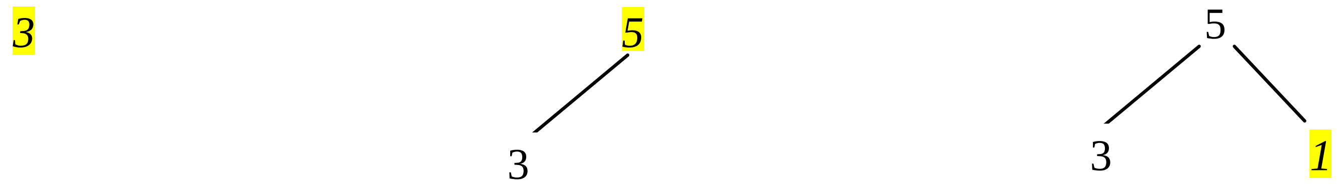


(a) Add 88 to a heap (b) After swapping 88 with 19 (b) After swapping 88 with 22

Adding elements to a Heap

This example begins with an empty *max-heap* and then the values 3, 5, 1, 19, 11, and 22 are added.

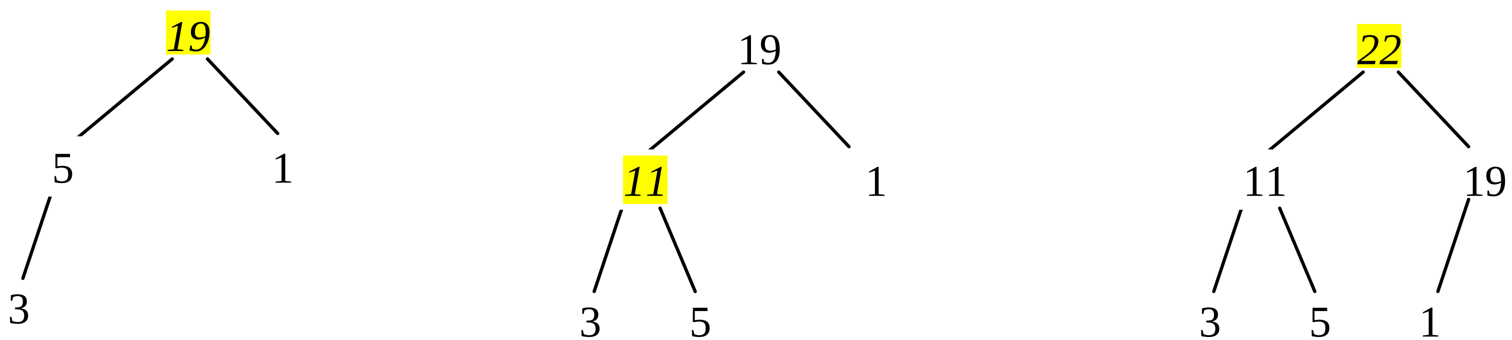
The scan / adjust process must be performed when inserting each new value.



(a) After adding 3

(b) After adding 5

(c) After adding 1



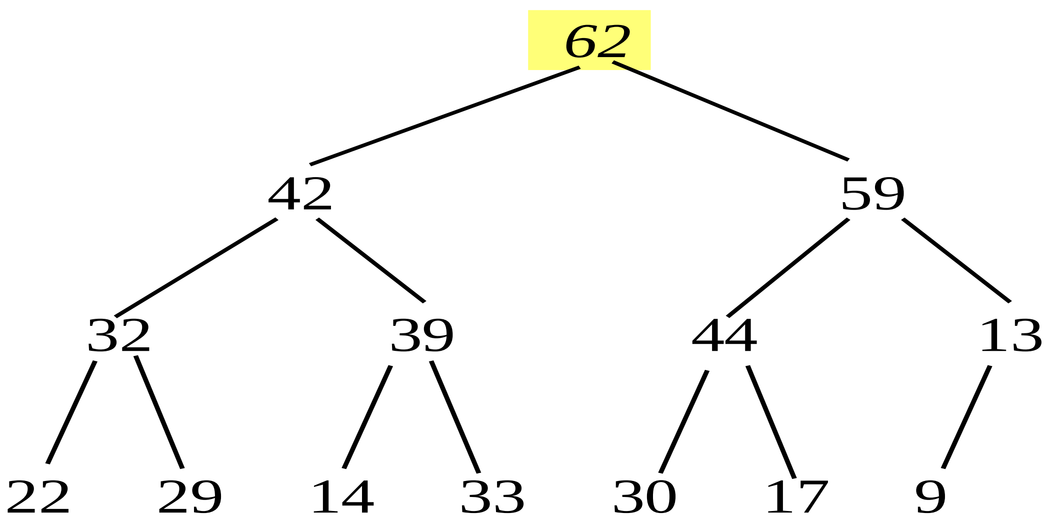
(d) After adding 19

(e) After adding 11

(f) After adding 22

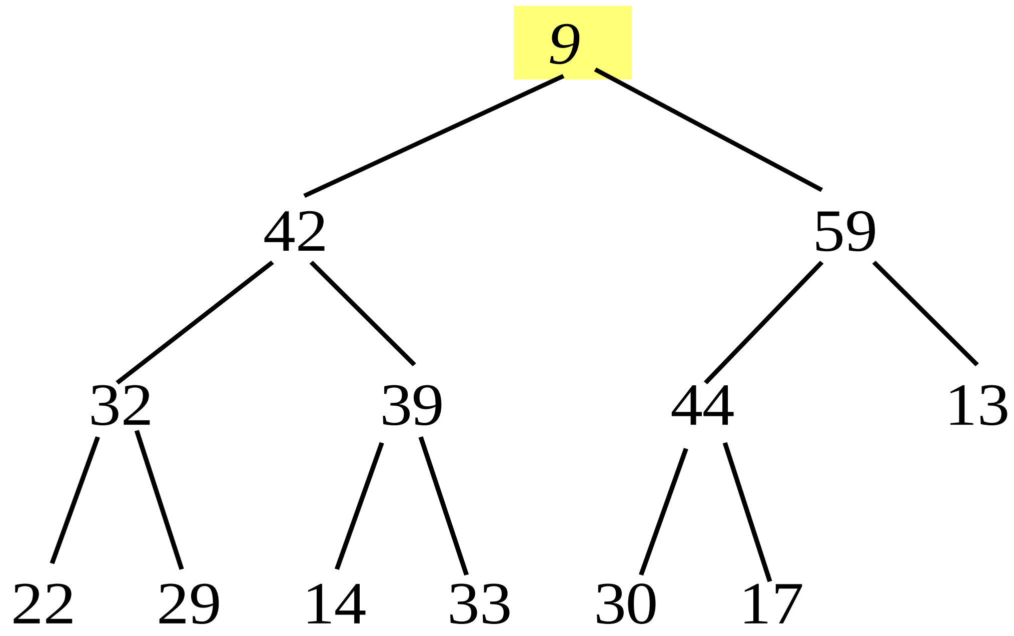
Removing the root node (1)

- If we remove the root node (node with hightest value), we have to put *something* in its place, so we initially choose the last node, and then reposition that node to its correct position in the tree.
- In this example, we choose to remove the **62** value:



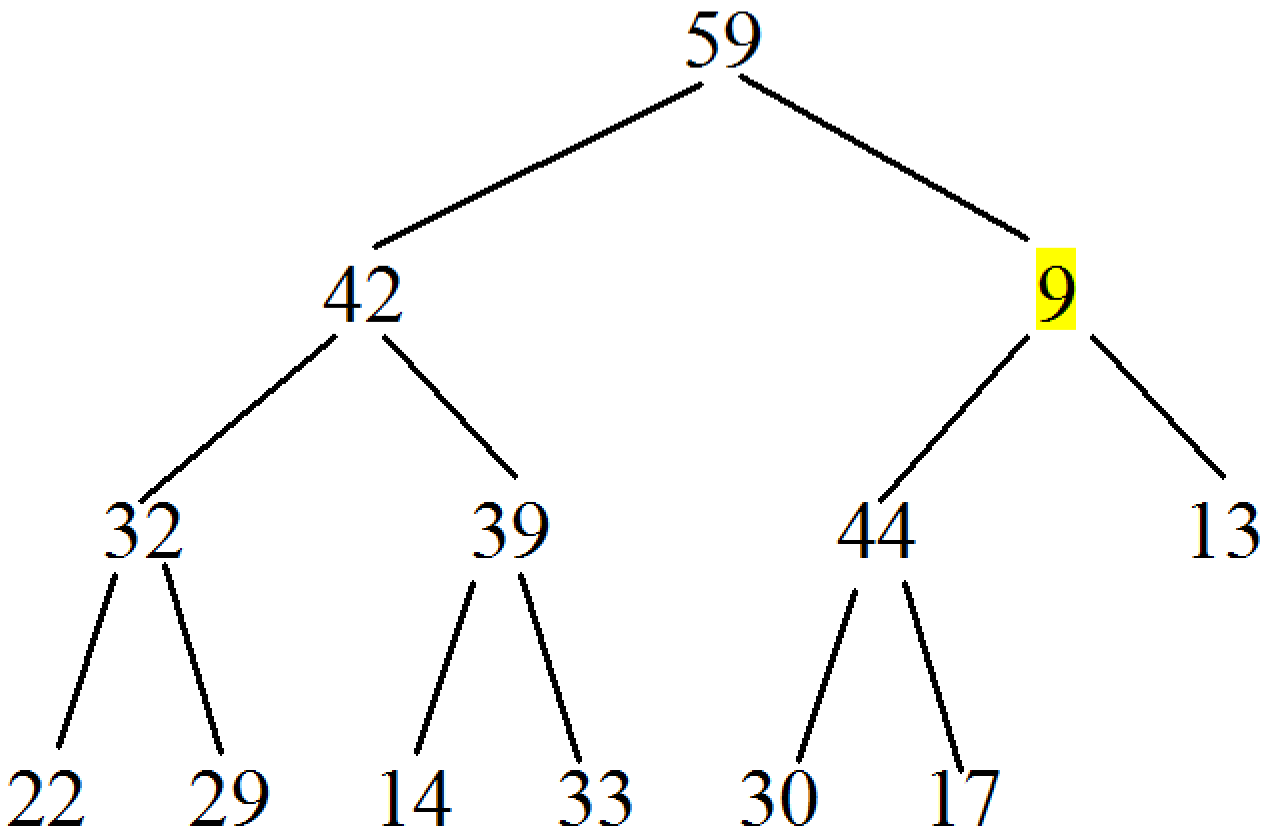
Removing the root node (2)

- So we fill in the root position with the last node of the last row: in this case, the **9** value.



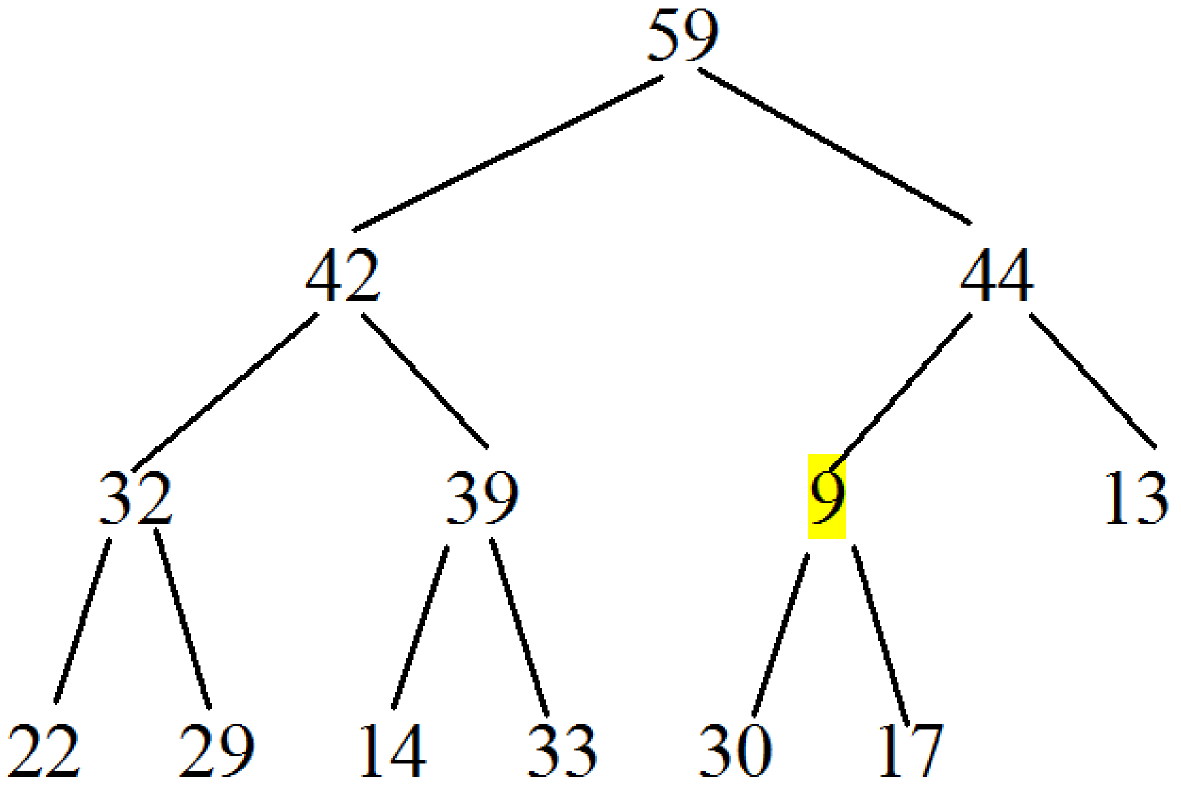
Removing the root node (3)

- Next we need to find the correct position for the **9** value:
  - swap it with the larger of its two children (in this case, **59**).



Removing the root node (4)

- The **9** value is still not in the correct position:
  - swap it with the larger of its two children (in this case, **44**).



Removing the root node (5)

- The **9** value is still not in the correct position:
  - swap it with the larger of its two children (in this case, **30**).
- Finally, the tree is again obeying the rules for a heap.