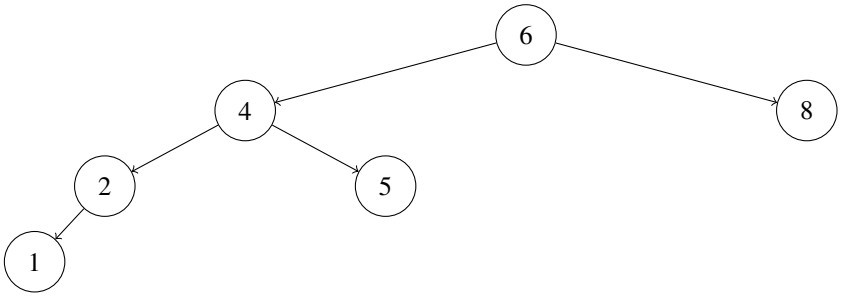
CPSC 035: Data Structures and Algorithms

Lab 7: Binary Search Trees

Roger Wang and Nina Zhuo

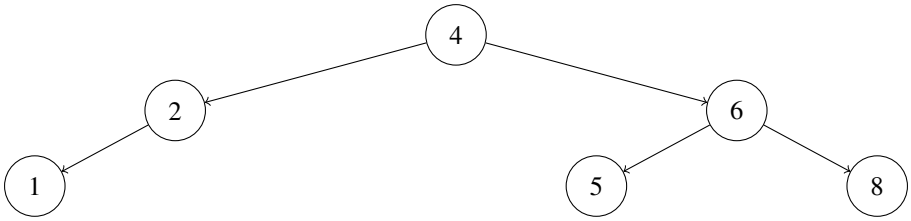
1 AVL Trees

Problem 1. Perform a right rotation on the root of the following tree. Be sure to specify the X, Y, and Z subtrees used in the rotation.



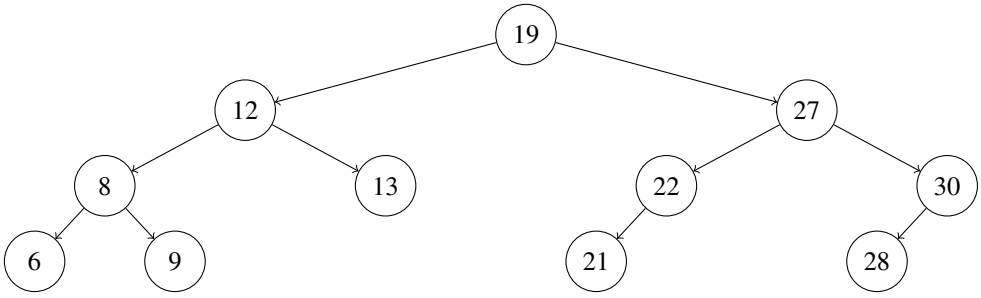
We see that the left child of root 6 has a height of 2 while the right child has a height of 0, and since the height of the left subtree is greater than the height of the right subtree by 2 and $2 > 1$, we need to perform an right rotation to balance the tree.

The binary search property dictates that $A < x < B < y < C$ where x is the node we want to rotate around, y is the node rotating around it. For this tree, $(A = 2) < (x = 4) < (B = 5) < (y = 6) < (C = 8)$. So, when we call `rightRotate(6)`, we get the resulting tree:



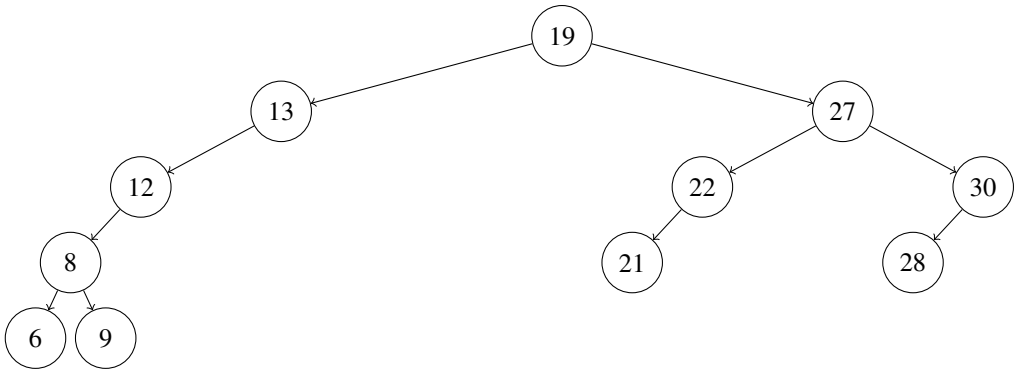
Since the heights of root 4's children differ by at most one, we know that this is an AVL tree.

Problem 2. Show the left rotation of the subtree rooted at 12. Be sure to specify the X, Y, and Z subtrees used in the rotation.

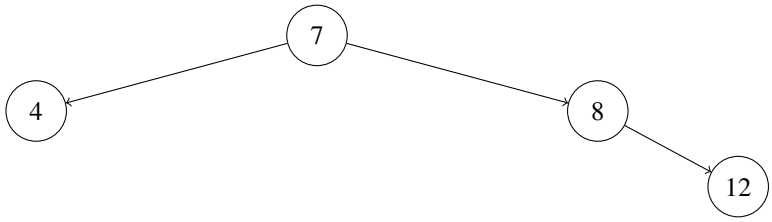


The binary search property dictates that $A < x < B < y < C$ where x is the node we want to rotate around, y is the node rotating around it. For this tree, $(A = 8) < (x = 12) < (y = 13)$. There is no B or C in this instance because the right child of the subtree rooted at $x = 12$ is y and the children of $y = 13$ are `null`.

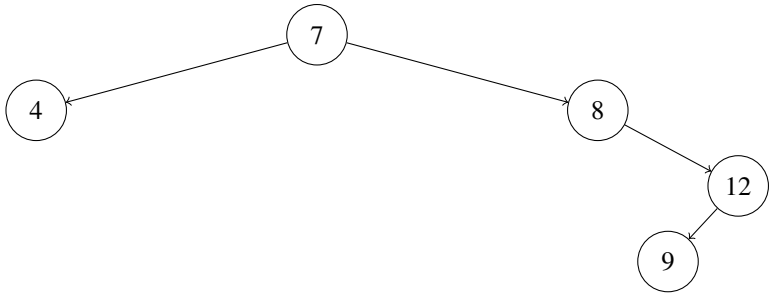
So, when we call `leftRotate(12)`, we get the resulting tree:



Problem 3. Using the appropriate AVL tree algorithm, insert the value 9 into the following tree. Show the tree before and after rebalancing.

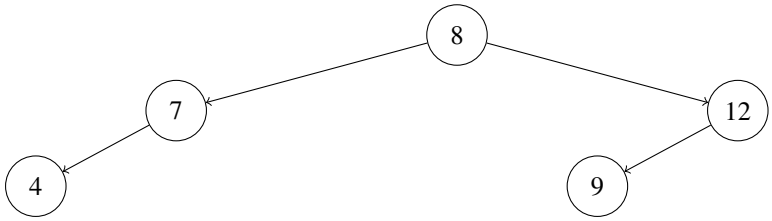


First, we perform `insert(9)` to get the following BST:



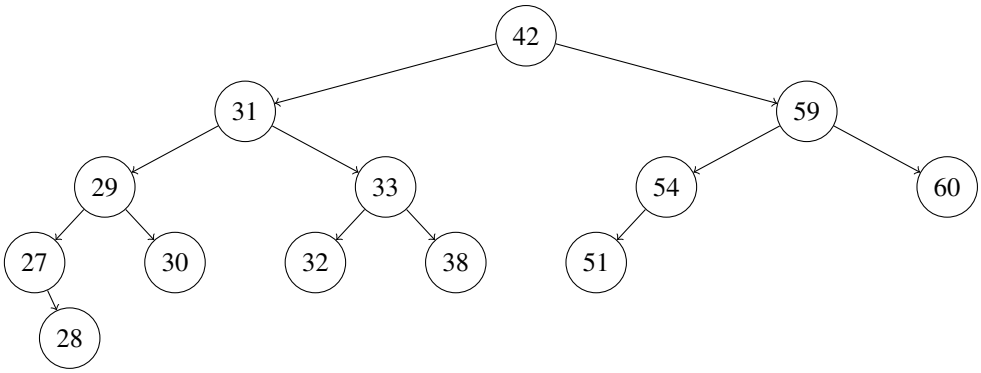
We see that the subtree to the right of 7 has a height of 0 while the subtree to the left of 7 has a height of 2. Since the difference of the two heights is greater than 1, we need to perform an left rotation to balance the tree.

The binary search property dictates that $A < x < B < y < C$ where x is the node we want to rotate around, y is the node rotating around it. For this tree, $(A = 4) < (x = 7) < (y = 8) < (C = 12)$. So, when we call, `leftRotate(7)` we get the resulting tree:

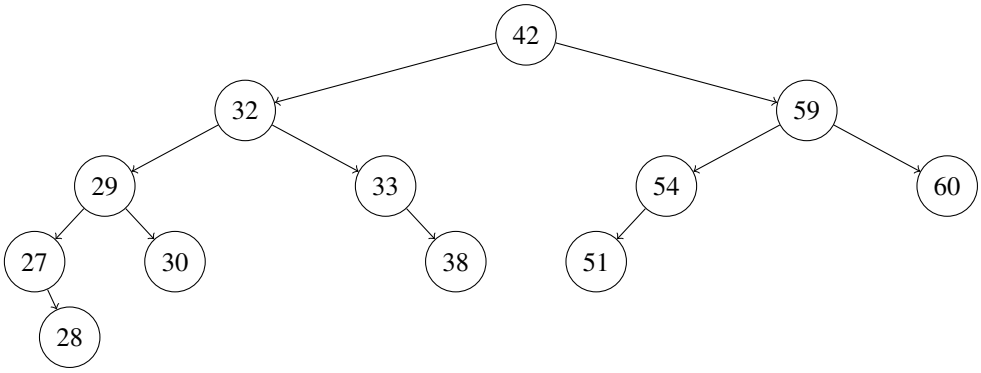


Since the heights of root 8's children differ by at most one, we know that this is an AVL tree.

Problem 4. Using the appropriate AVL tree algorithm, remove the value 31 from the following tree. Show the tree before and after rebalancing.



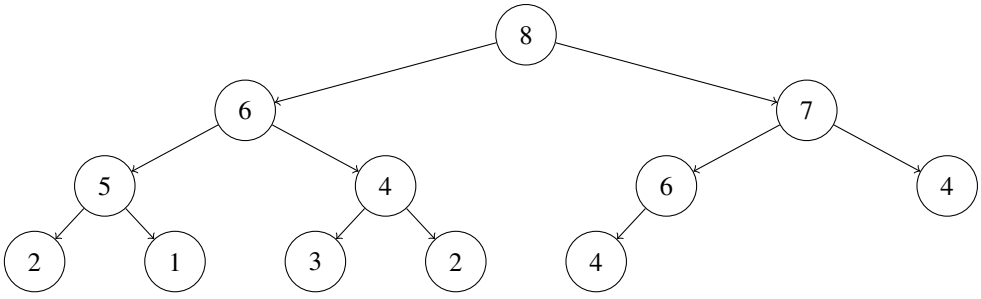
First, we perform `remove(31)` by replacing 31 with its successor 32 and deleting the node to the left of 33 to get the resulting tree:



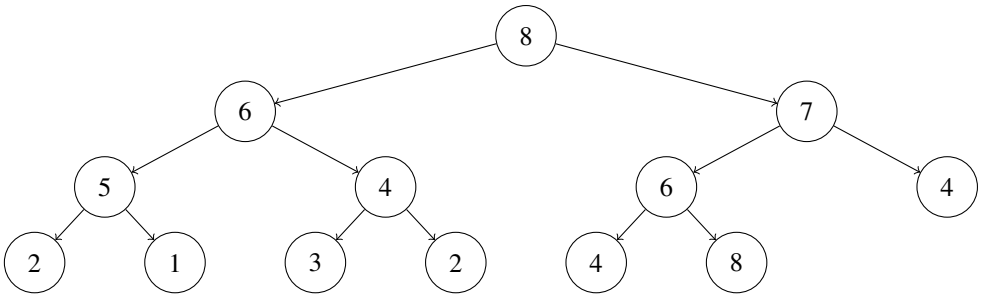
We see that the left subtree of 32 has a height of 2 and the right subtree of 32 has a height of 1. Since the difference between the two hieghts is 1, the subtree is balanced at this node. Similarly, the left subtree of 33 has a height of -1 and the right subtree of 38 has a height of 0. Since the difference between the two heights is 1, the subtree is balanced at this node. We can apply to same logic to the left subtree of 32 and find that, since the difference between the heights of each node's subtree is at most one, the tree is already an AVL tree, so we do not need to perform any rotations.

2 Heaps

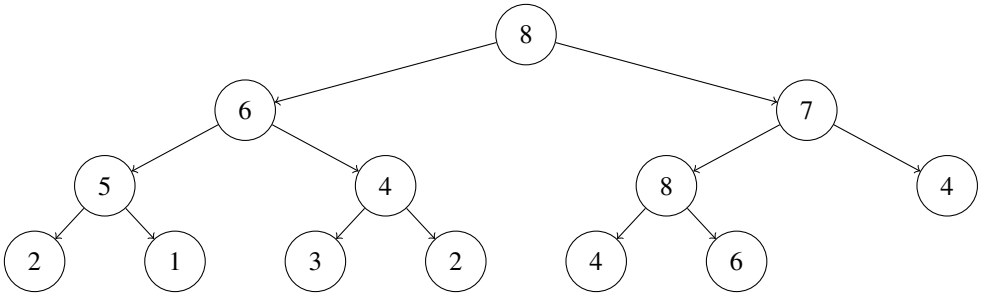
Problem 1. Show the addition of the element 8 to the max-heap below. First, show the addition of 8 to the tree; then, show each bubbling step.



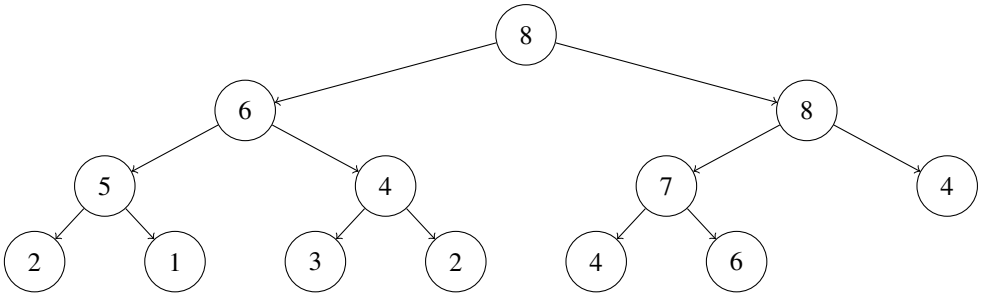
A heap is an invariant of a complete tree, and since elements in a complete tree are read in from left to right from the root down, when we perform `insert(8)`, we get the resulting tree:



By definition, a heap must be complete tree where each node is greater than or equal to all of its subsequent children. However, 8 is greater than its parent 6, so we call `bubbleUp(8)` to swap 8 with its parent to get the resulting tree:

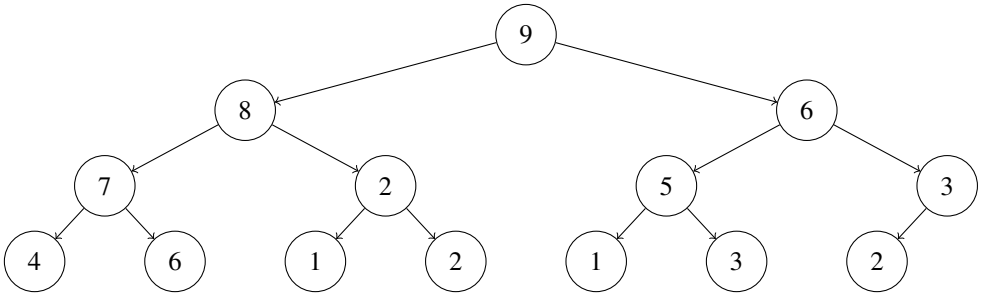


However, 8 is still greater than its parent 7, so we call `bubbleUp(8)` once more to swap 8 and 7 to get the resulting tree:

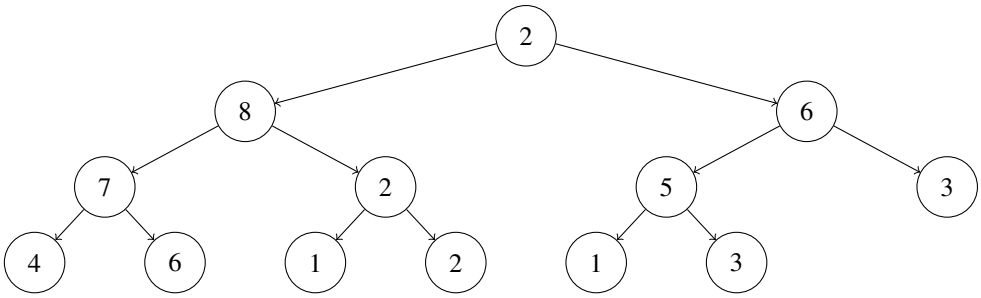


Now, the priority at each heap node is greater than or equal to all its children.

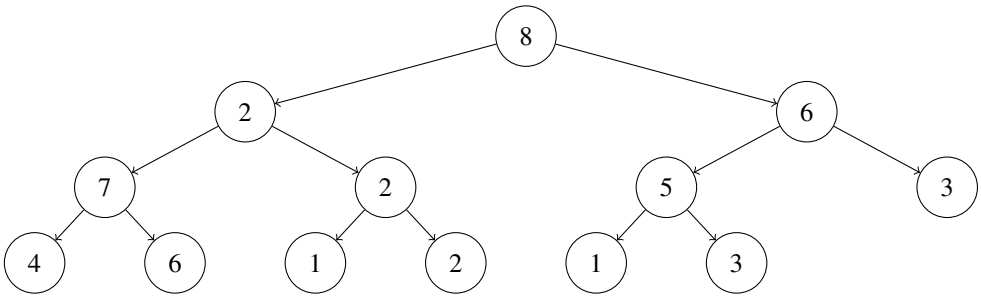
Problem 2. Show the removal of the top element of this max-heap. First, show the swap of the root node; then, show each bubbling step.



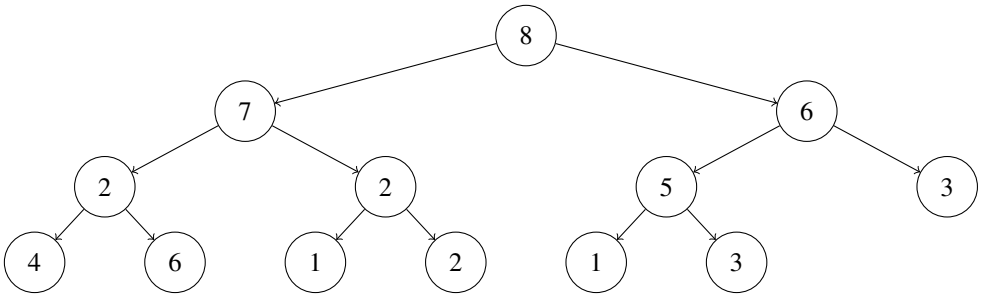
First, we perform `remove(9)` and swap the rightmost node at depth 0, which is the most recently added node, to get the following tree:



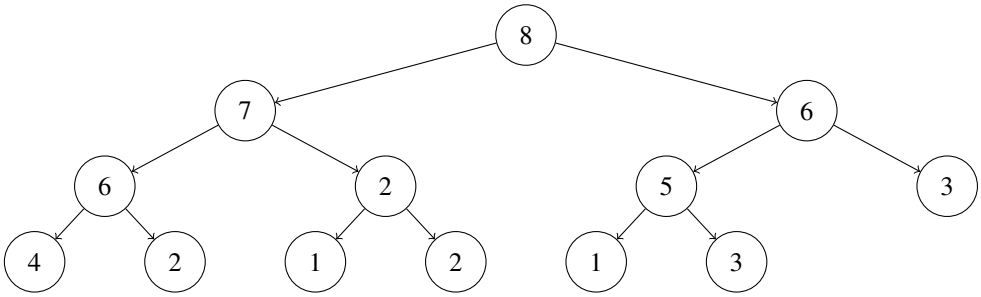
By definition, a heap must be complete tree where each node is greater than or equal to all of its subsequent children. Because $2 < 8$ and $2 < 6$, the above tree violates the second property. To rectify this, we call `bubbleDown(2)` and swap 2 with its max child 8 to get the resulting tree:



Here, 2 is still less than both of its children, so we recursively call `bubbleDown(2)` and swap 2 with its max child 7 to get the resulting tree:

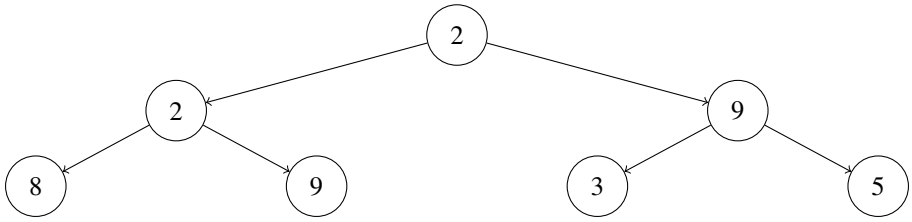


Since 2 is still less than both of its children, we call `bubbleDown(2)` once again and swap 2 with its max child 6 to get the resulting tree:



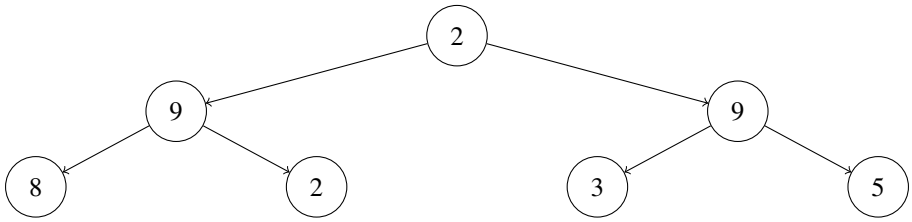
Problem 3. Consider the sequence of elements `[2,2,9,8,9,3,5]`. Using the representation discussed in class, show the tree to which this sequence corresponds. Then, show the *heapification* of this tree; that is, show how this tree is transformed into a heap. Demonstrate each bubbling step.

In a complete binary tree, each element would be read in from left to right to get the resulting tree:



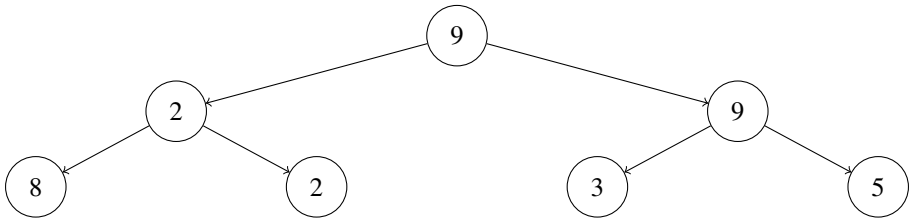
By definition, a heap must be complete tree where each node is greater than or equal to all of its subsequent children. Because the above tree violates the second property, it is not a heap, so we must call `heapify()` on it. `heapify` calls `bubbleDown()` on each node, starting from the leftmost node at depth 0 and moving up the tree from left to right. So, to heapify the complete tree, we start at 8 and call `bubbleDown(8)`. Since 8 has no children, the tree remains unchanged. The same is true when we call `bubbleDown()` on 9, 3, and 5.

As we move into the nodes at depth 1 and call `bubbleDown(2)`, we see that 2 is less than both its children 8 and 9, so we swap 2 with it's max child, which is 9, and get the resulting tree:

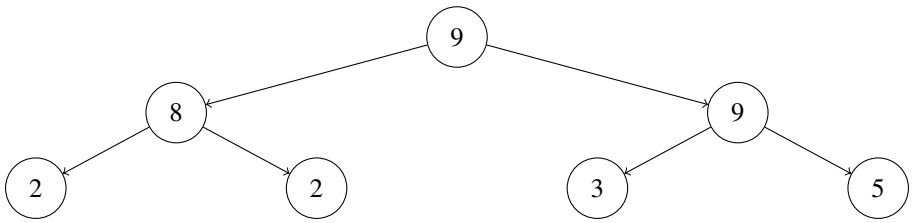


Since $9 > 8$ and $9 > 2$, the subtree at that node 9 is heapified. We then call `bubbleDown(9)` on the node to the right, but since 9 is greater than both of its children 3 and 5, the subtree remains unchanged.

Finally, we call `bubbleDown(2)` on the root node. Because $2 < 9$ and $leftChild = rightChild$, we swap 2 with the left child and get the resulting tree:



Because 2 is not greater than or equal to both of its subchildren, we call `bubbleDown(2)` again and swap 2 with its max child 8 to get the resulting tree:



Now the priority at each node is greater than the priority of its subchildren, so we know that complete tree has been heapified.