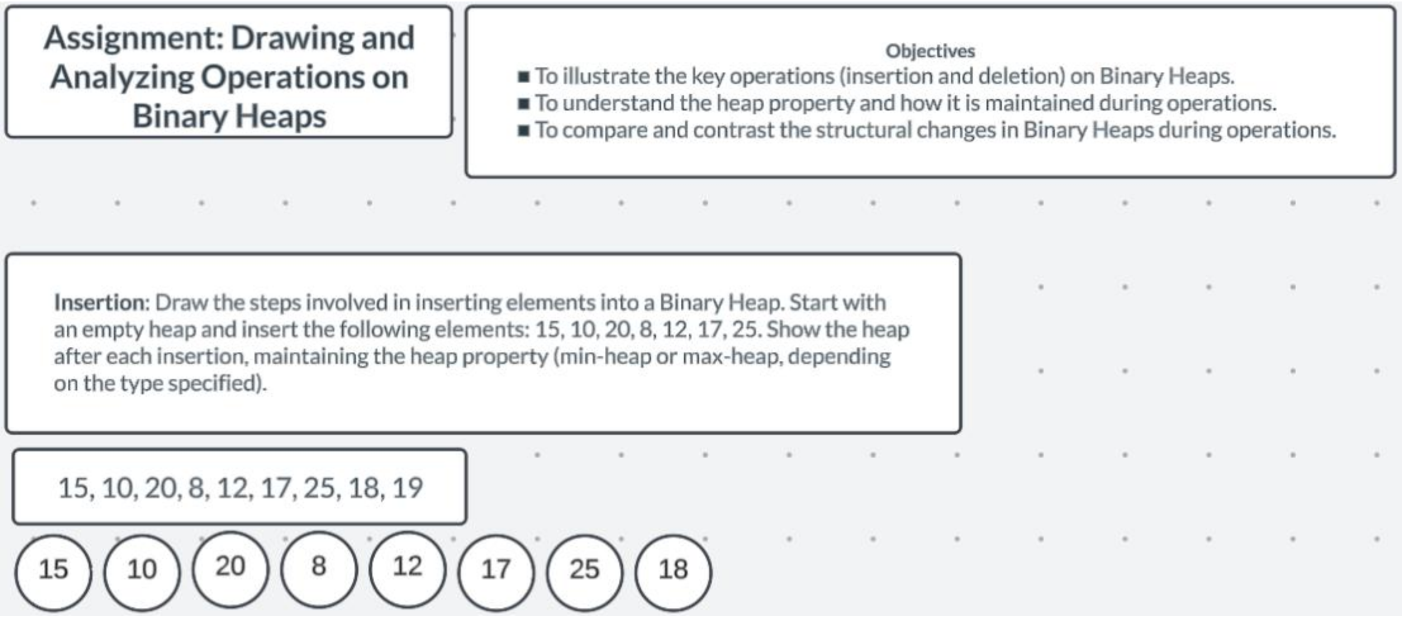
Keiffer Tan

Assignment: Drawing and Analyzing Operations on Binary Heaps



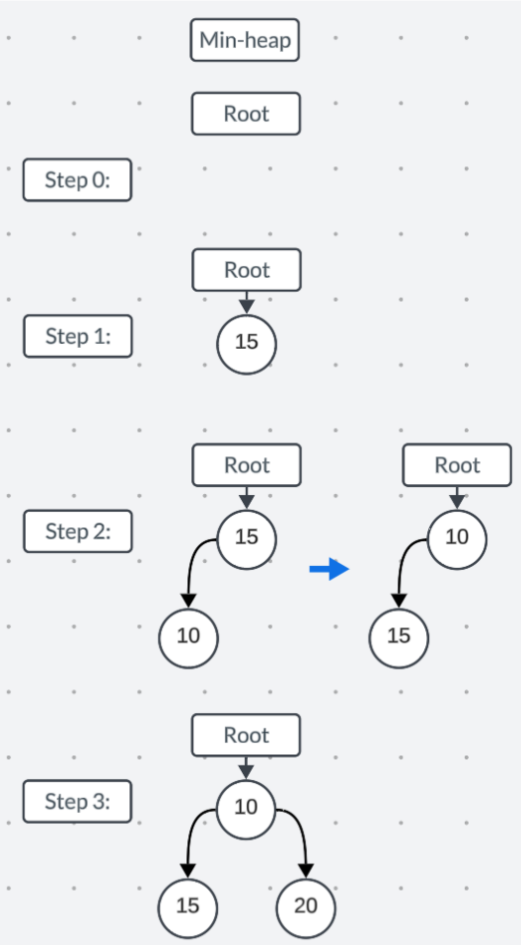
Assignment: Drawing and Analyzing Operations on Binary Heaps

Objectives

- To illustrate the key operations (insertion and deletion) on Binary Heaps.
- To understand the heap property and how it is maintained during operations.
- To compare and contrast the structural changes in Binary Heaps during operations.

Insertion: Draw the steps involved in inserting elements into a Binary Heap. Start with an empty heap and insert the following elements: 15, 10, 20, 8, 12, 17, 25. Show the heap after each insertion, maintaining the heap property (min-heap or max-heap, depending on the type specified).

15, 10, 20, 8, 12, 17, 25, 18, 19



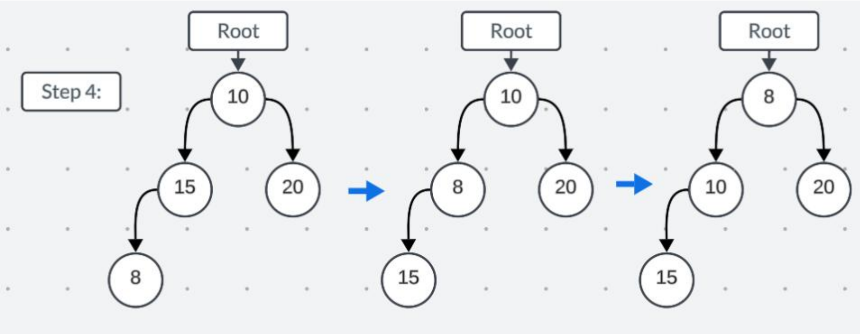
Starting off with an empty tree, we don't have any insertions yet. I am going to be working with a Min-heap tree for this assignment. Though Max-heap is similar just with the parent nodes being greater than the child nodes.

At the first insert, (15) is set as the root of the tree since there are no heap nodes.

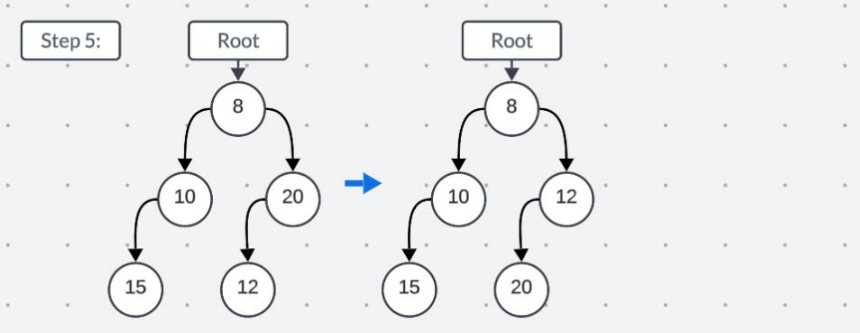
With the insertion of (10), we first place it as the left child of the root node. Then we check to see if the tree is sorted correctly, so we heapify up. We notice that the child's key is less than the parent's key, so we swap them.

Next inserting (20), we insert it as the right child of the root (10), and since the child's key is less than the parent, there is no need to make any further actions.

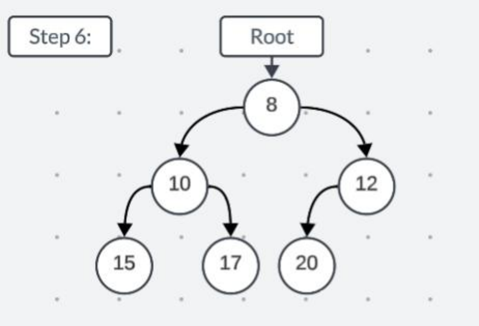
Keiffer Tan



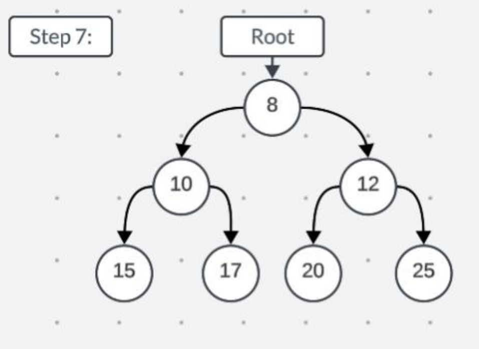
As we insert (8) to the bottom left most of the tree, we heapify up to ensure that priority (key) balance is preserved. (8) and (15) are swapped, and then (8) and the root (10) are swapped again to ensure that the node with the smallest key value is always at the root and every child’s key value is greater than the parent.



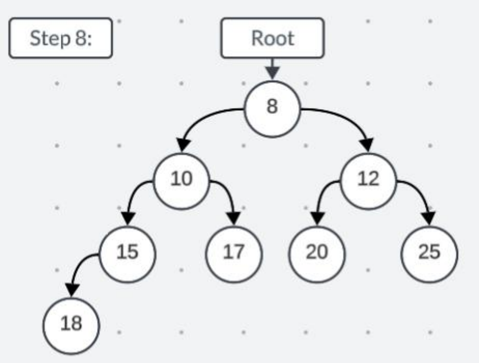
Then inserting (12) at the left child of the next open position (to preserve balance we need to add to the left children of the level first and then add to the right children.). It is noticed from the heapify up that the (12) is less than the parent (20) so a swap is made there.



Inserting (17) to the first right child of the lowest level, there are no additional swaps or actions needed as the child key is greater than the parent key.

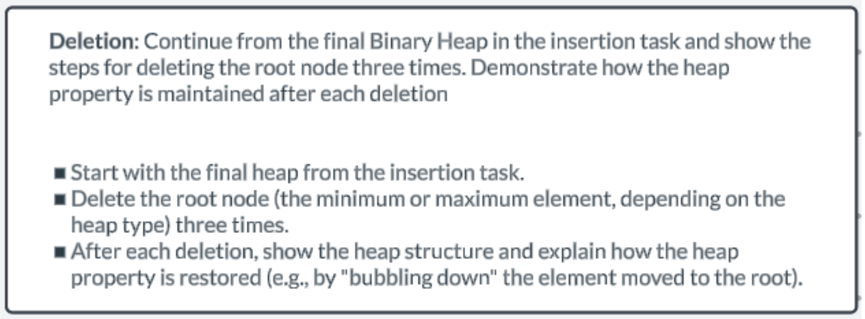


Inserting (25) to the open right child of the lowest level, there are no additional swaps or actions needed as the child key is greater than the parent key.



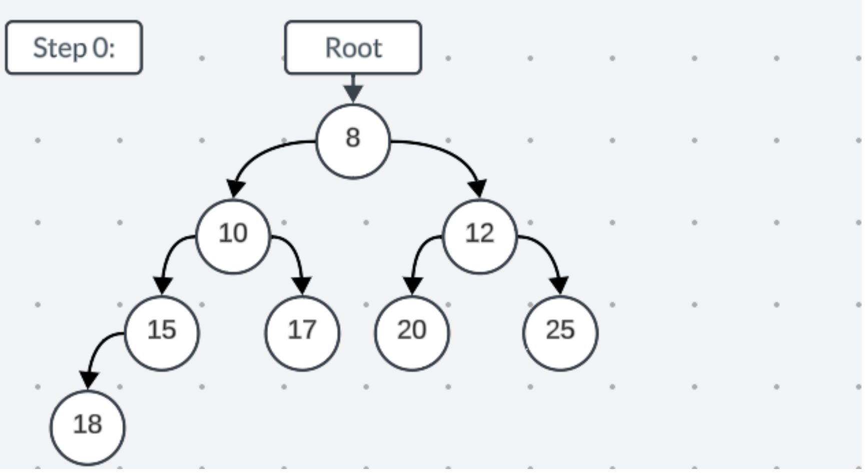
Inserting (18) to the first left child of the lowest level, there are no additional swaps or actions needed as the child key is greater than the parent key.

Keiffer Tan



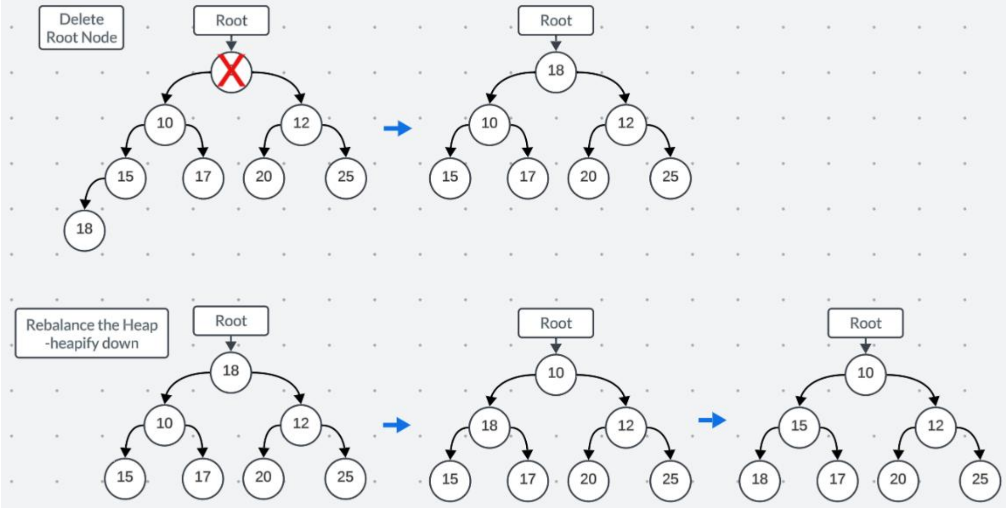
Deletion: Continue from the final Binary Heap in the insertion task and show the steps for deleting the root node three times. Demonstrate how the heap property is maintained after each deletion

- Start with the final heap from the insertion task.
- Delete the root node (the minimum or maximum element, depending on the heap type) three times.
- After each deletion, show the heap structure and explain how the heap property is restored (e.g., by "bubbling down" the element moved to the root).



In these deletion operations, we start off with the heap from the insertions operations.

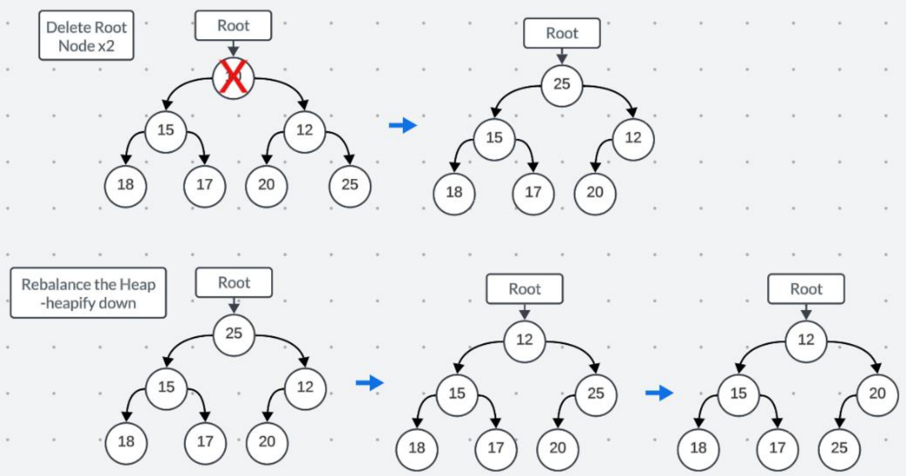
We need to delete the root node x3 for this exercise. (I am still using the min-heap as well)



When the first delete root operation is made, we need to find the last node of the tree to replace the root node of the heap. To find the last node, we look at the lowest level of the tree and retrieve the node based on the last insertion. (We need to insert the left side of the nodes from left to right first, and then insert the right side of the nodes from left to right as well.) We just need to follow the opposite direction when retrieving. Since there is only one leaf node currently, we retrieve (18) and place it at the root. Then to maintain the heap property, we need to heapify down making comparisons with the child nodes until the heap property is met.

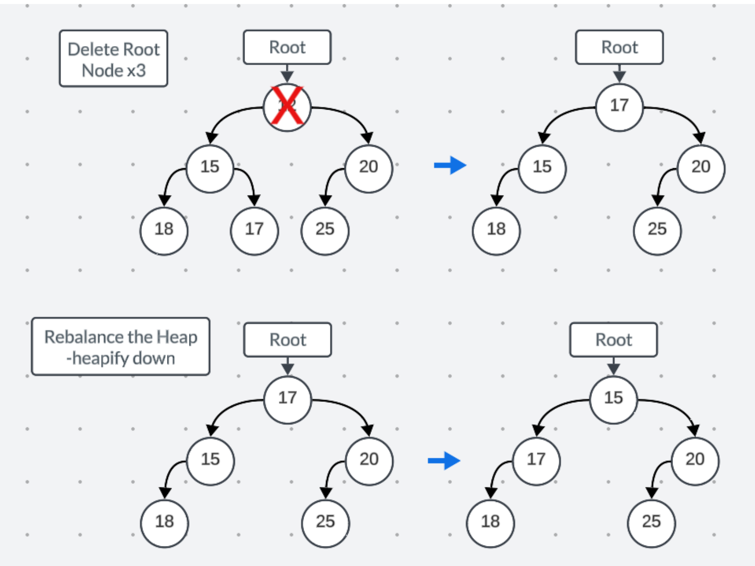
We need to swap the root (18) with the child of the lowest key value (10). Then we need to check if it is in the correct spot, but (18) is still greater than the child nodes, so we need to swap (18) and (15).

Keiffer Tan



The 2nd delete of the root node occurs and we take the “last right child” leaf node which is (25).

Then we need to heapify down to preserve the heap property. Swapping (12) and (25) then swapping (25) with (20) so that all children key values are larger than the parent key values.



The 3rd delete of the root node occurs and we take the “last right child” leaf node which is 17 and place it at the root.

Then heapifying down the tree, we swap (17) and (15) and then no further action needs to be made as the heap property of the tree is restored.

The structure of the heap is interesting with inserting and deleting to preserve the height and balance of the heap. We don’t add to the first available leaf position, but fill the left child leaf positions of the nodes and then the right child leaf positions. While in a BST, you just add to the child node, which can result in unbalanced trees. In an AVL tree, when adding, there is rebalancing and rotations made so that the tree is always balanced.