Heaps and AVL Trees

Maryam Shahid

21801344

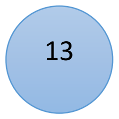
CS202

Assignment 3

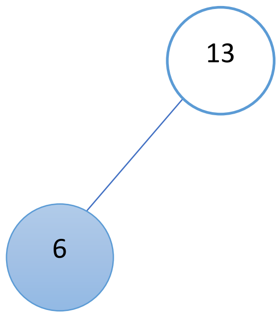
Question 1 – Part a

Initially: no node present; AVL tree is empty

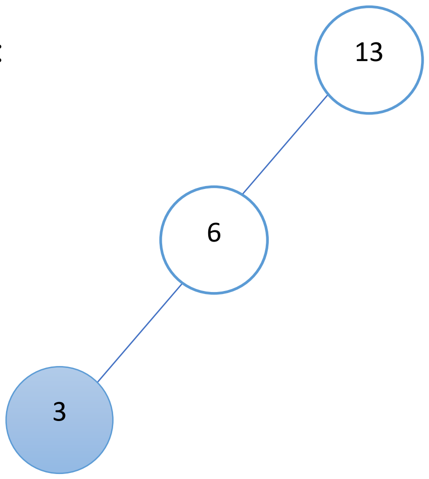
Insert 13:



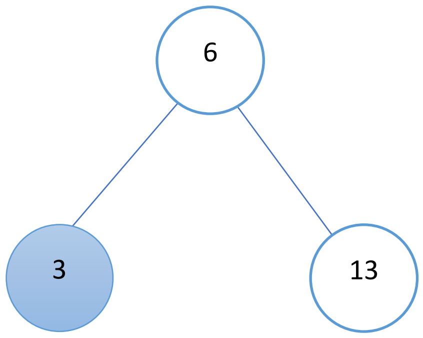
Insert 6:



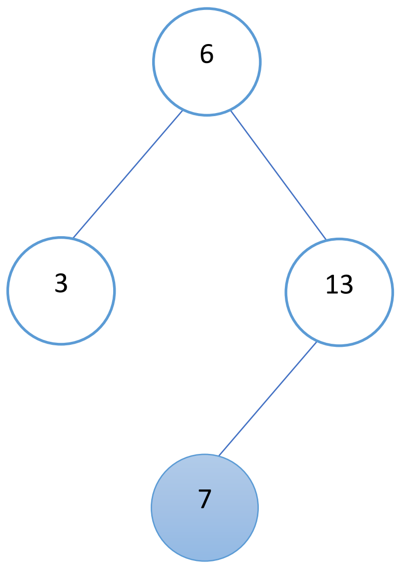
Insert 3:



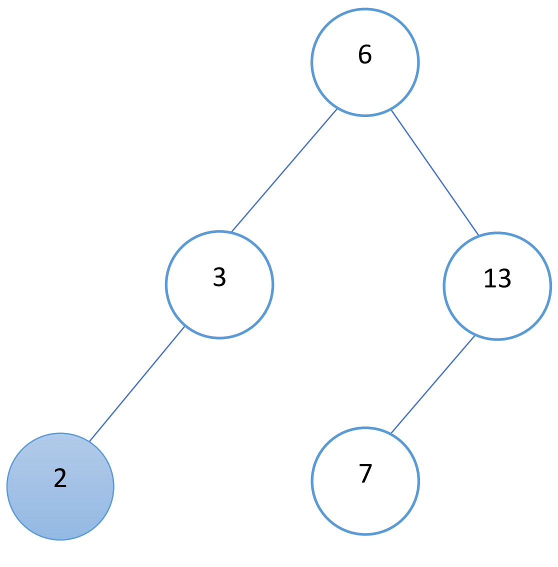
Perform single right rotation on 13



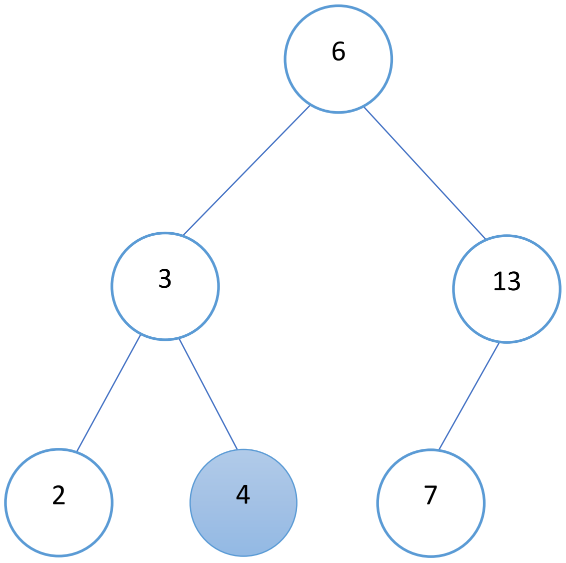
Insert 7:



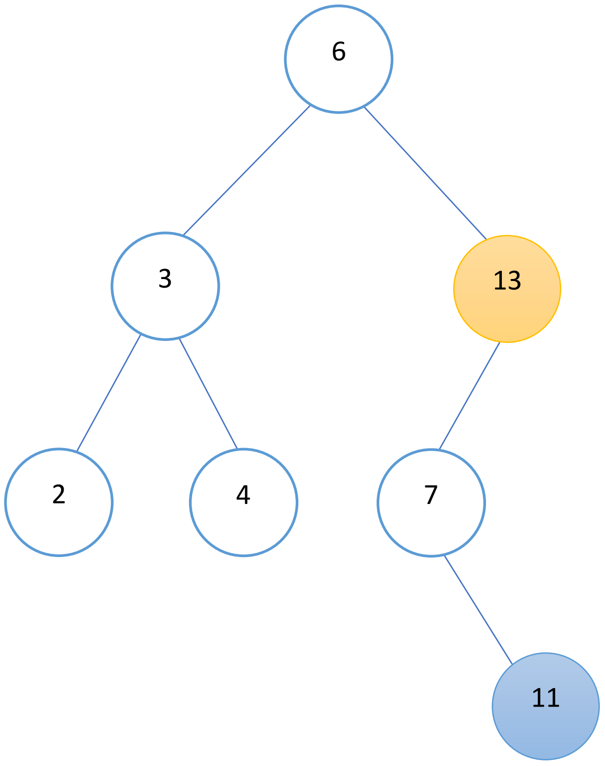
Insert 2:



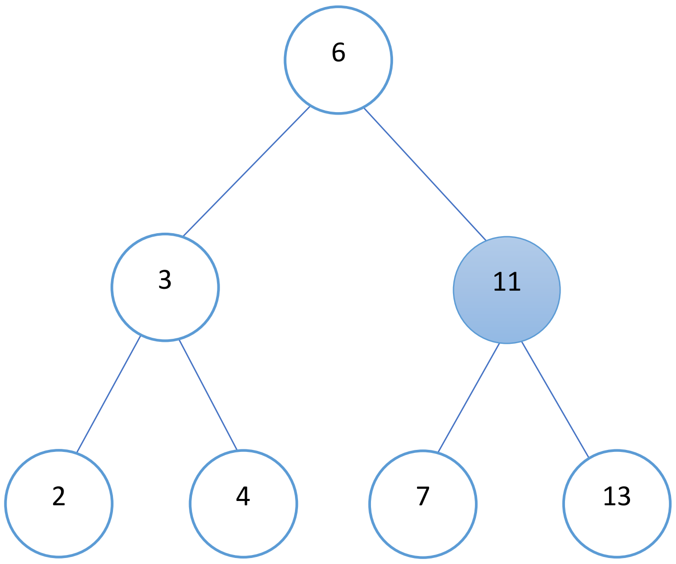
Insert 4:



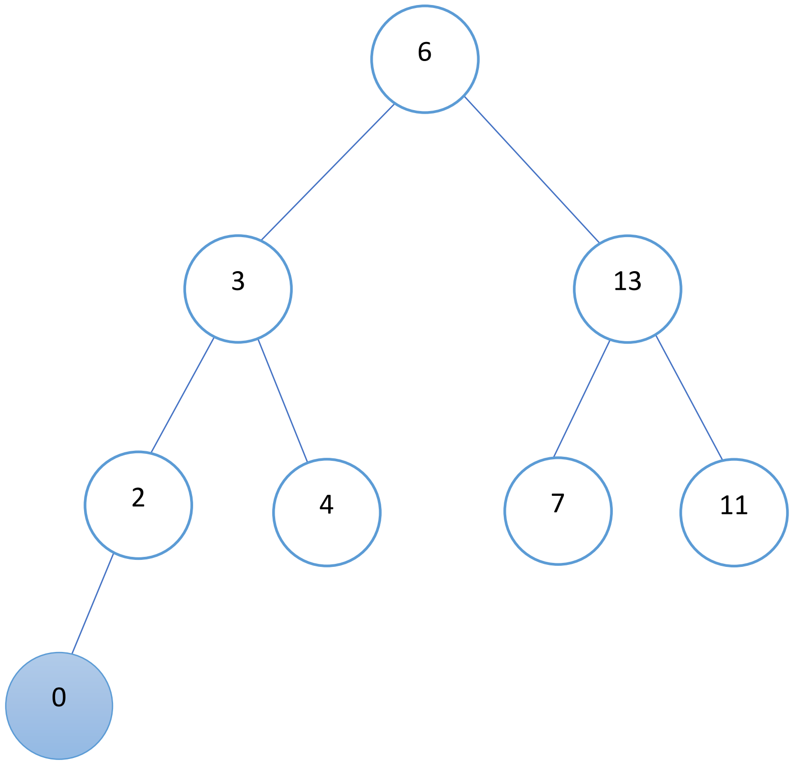
Insert 11:



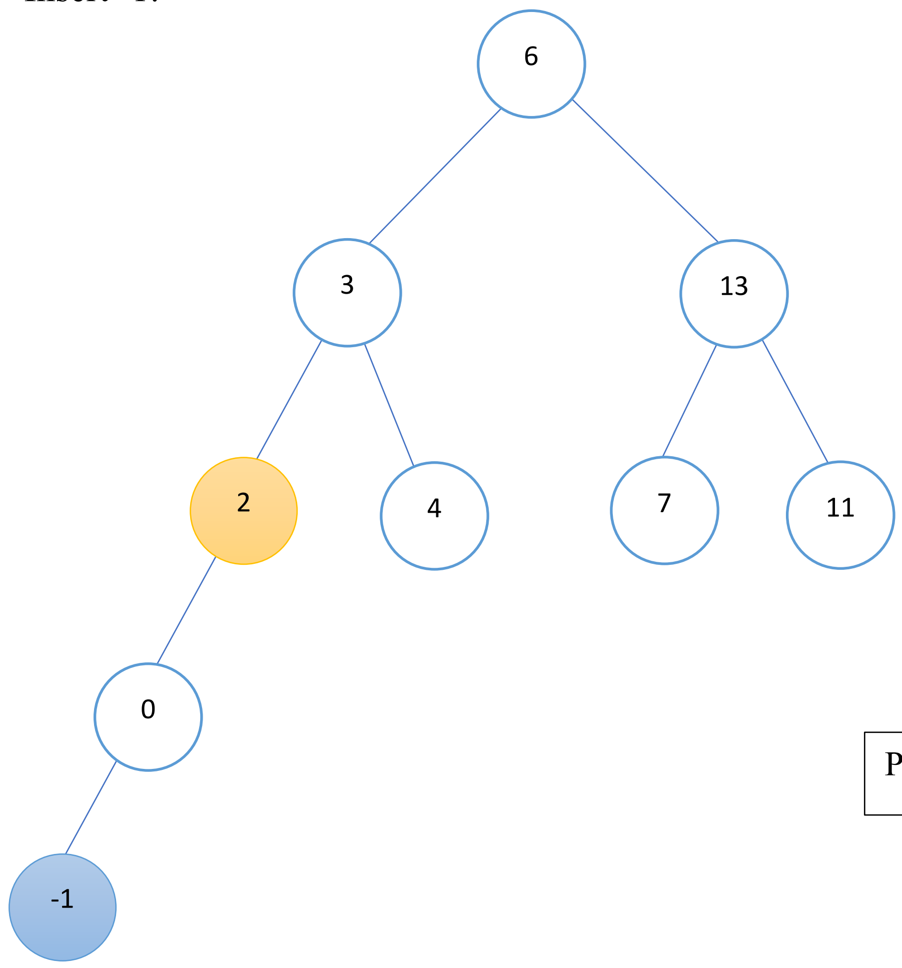
Perform double left - right rotation on 13



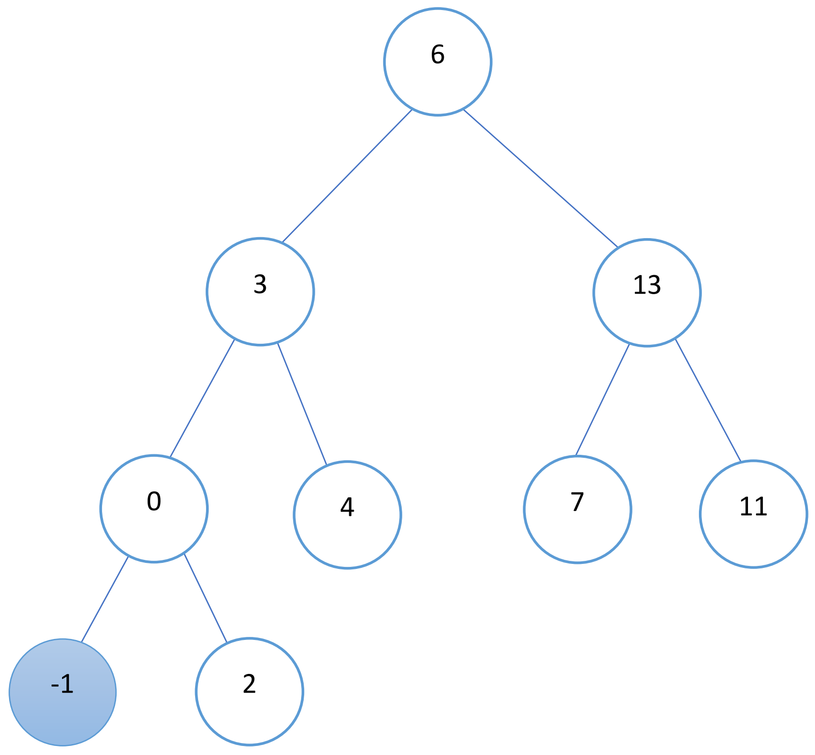
Insert 0:



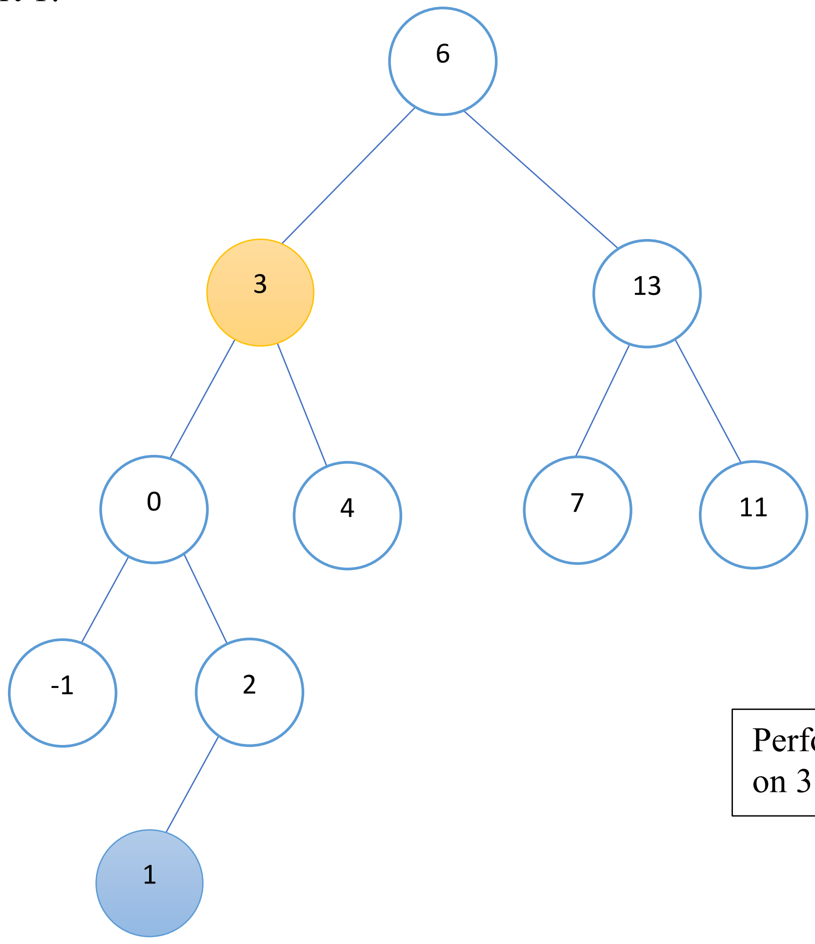
Insert -1:



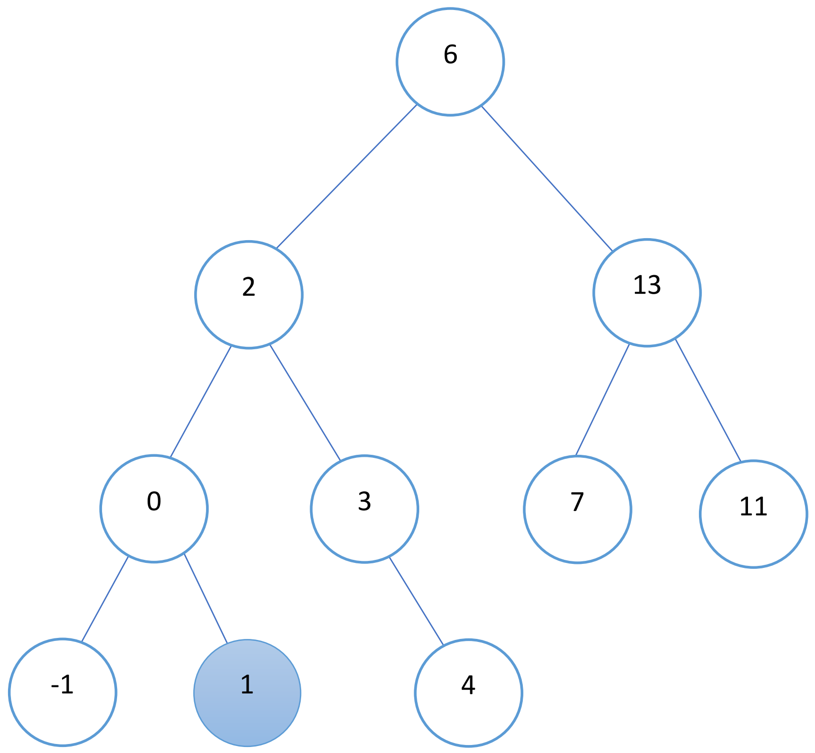
Perform single left rotation on 2



Insert 1:



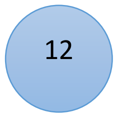
Perform double left - right rotation on 3



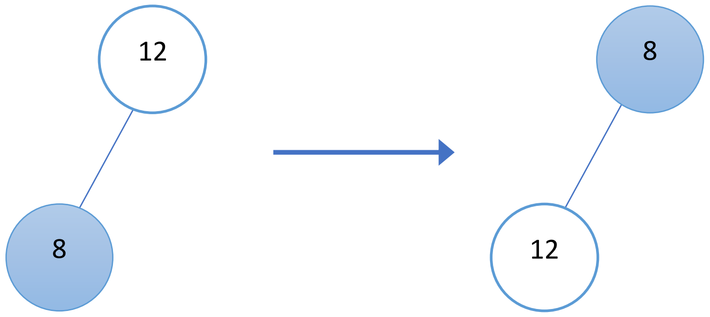
Part b

Initially: no nodes present; binary min heap is empty

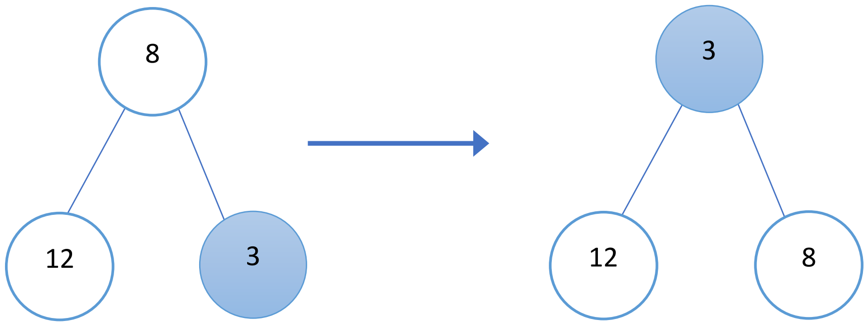
Insert 12:



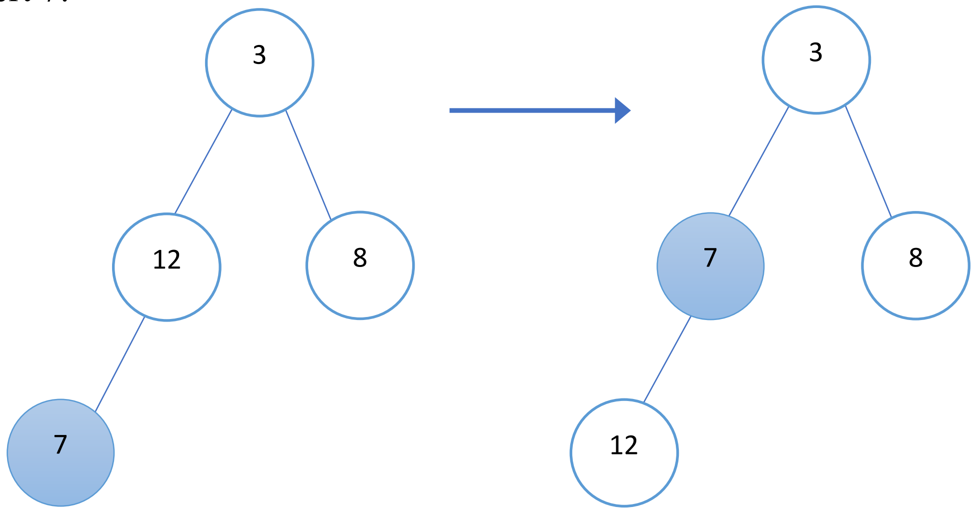
Insert 8:



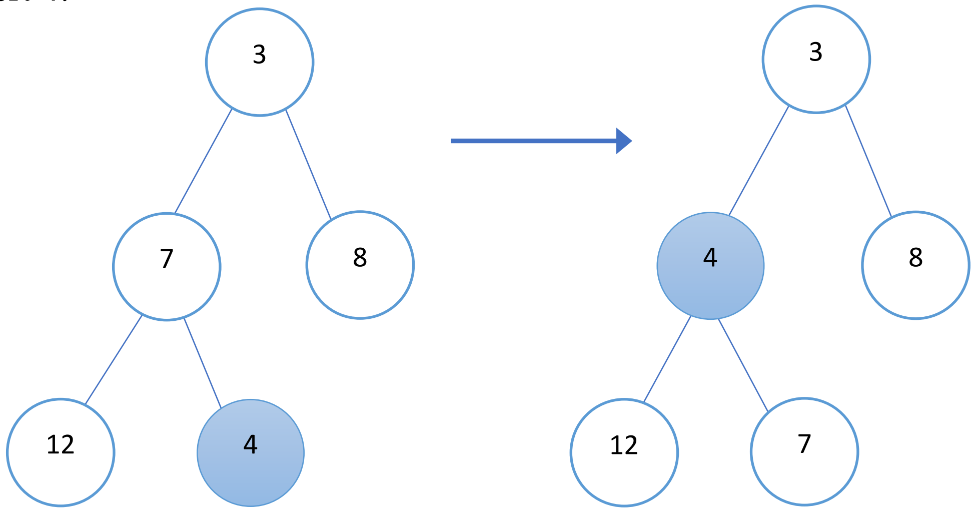
Insert 3:



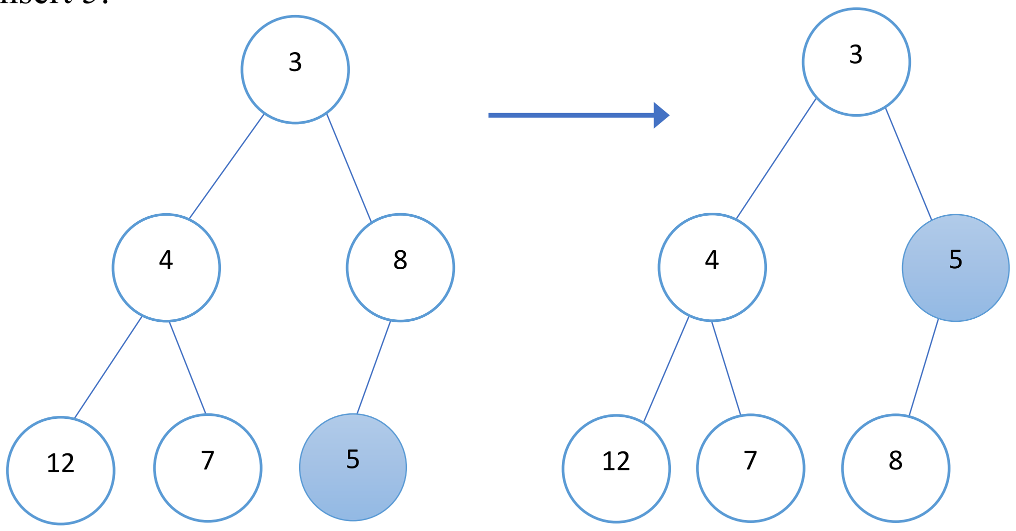
Insert 7:



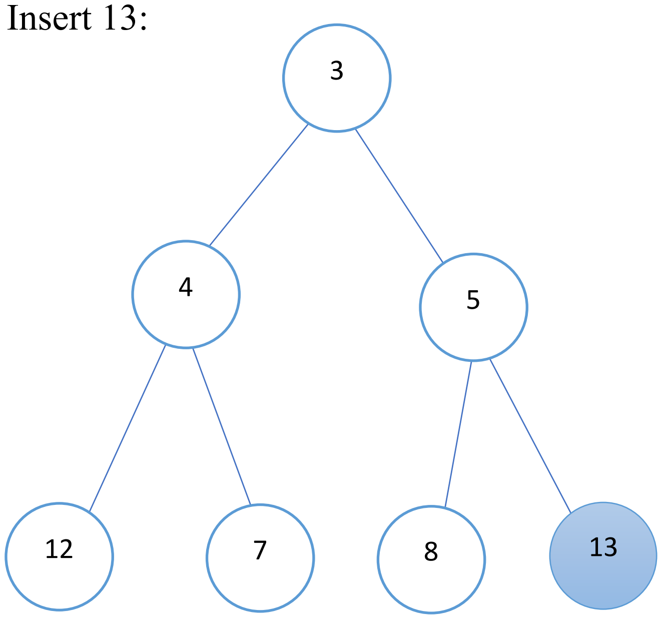
Insert 4:



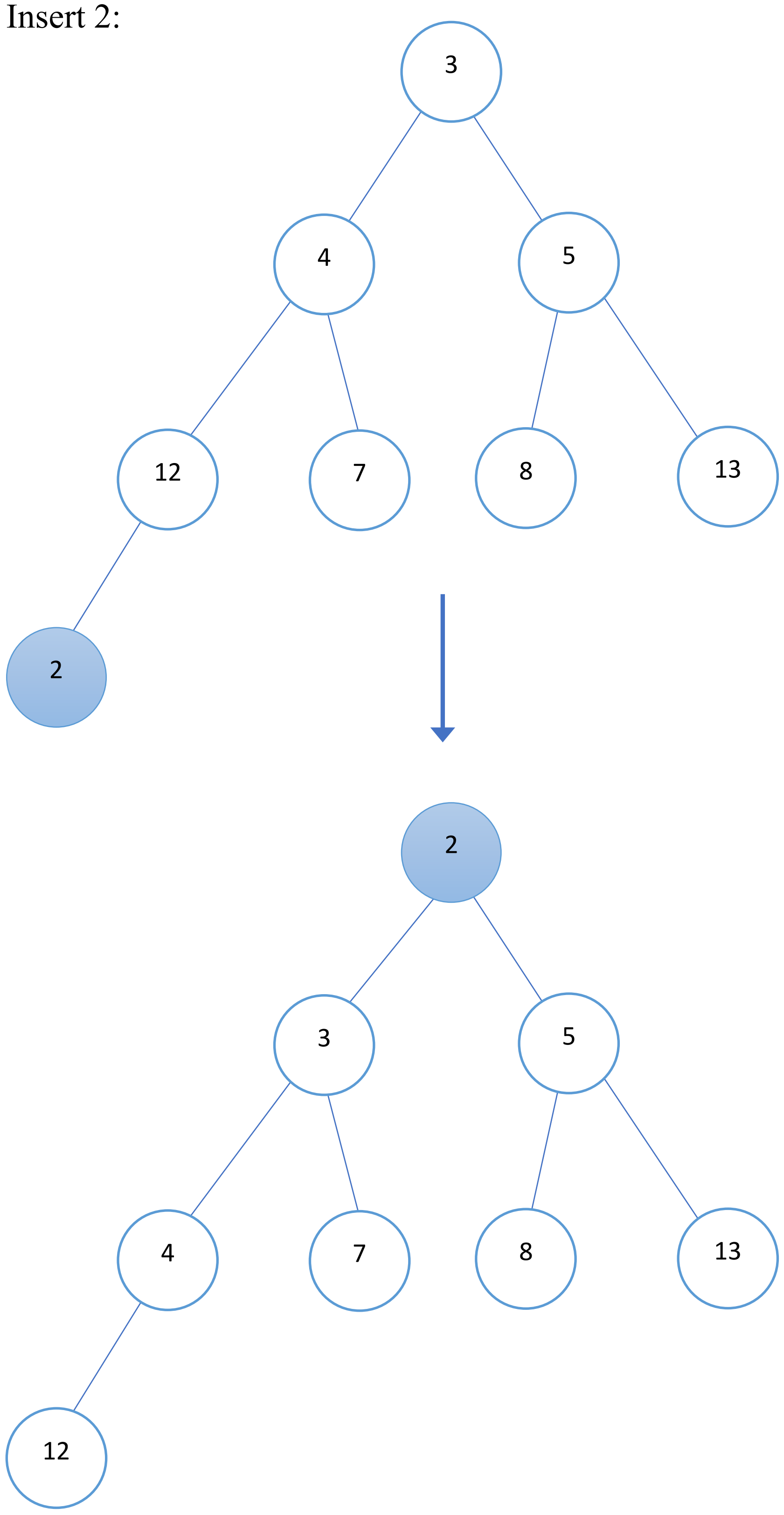
Insert 5:



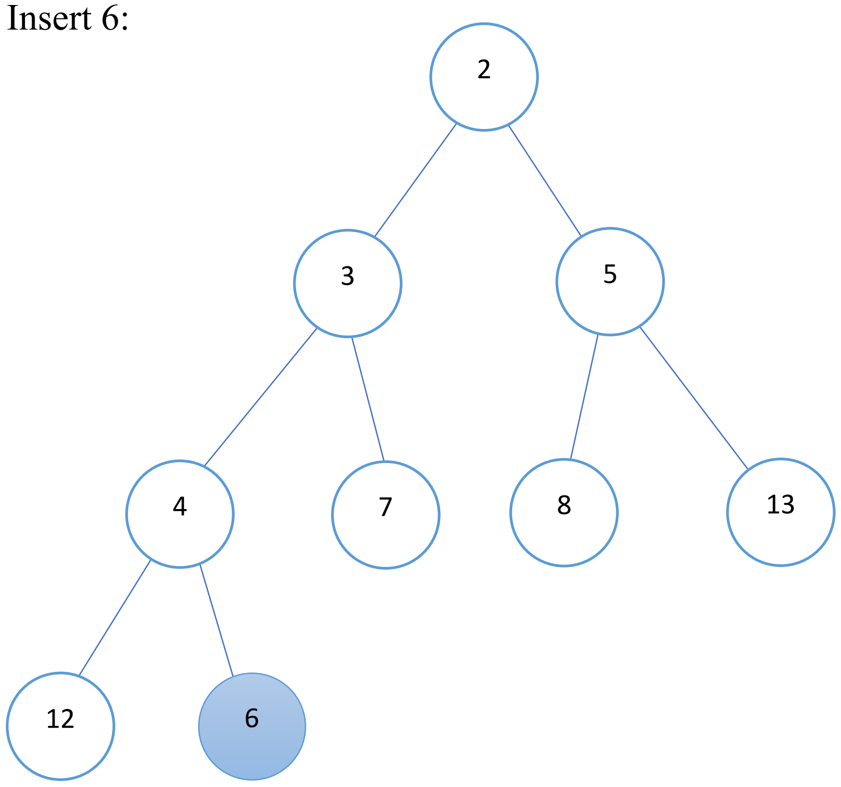
Insert 13:



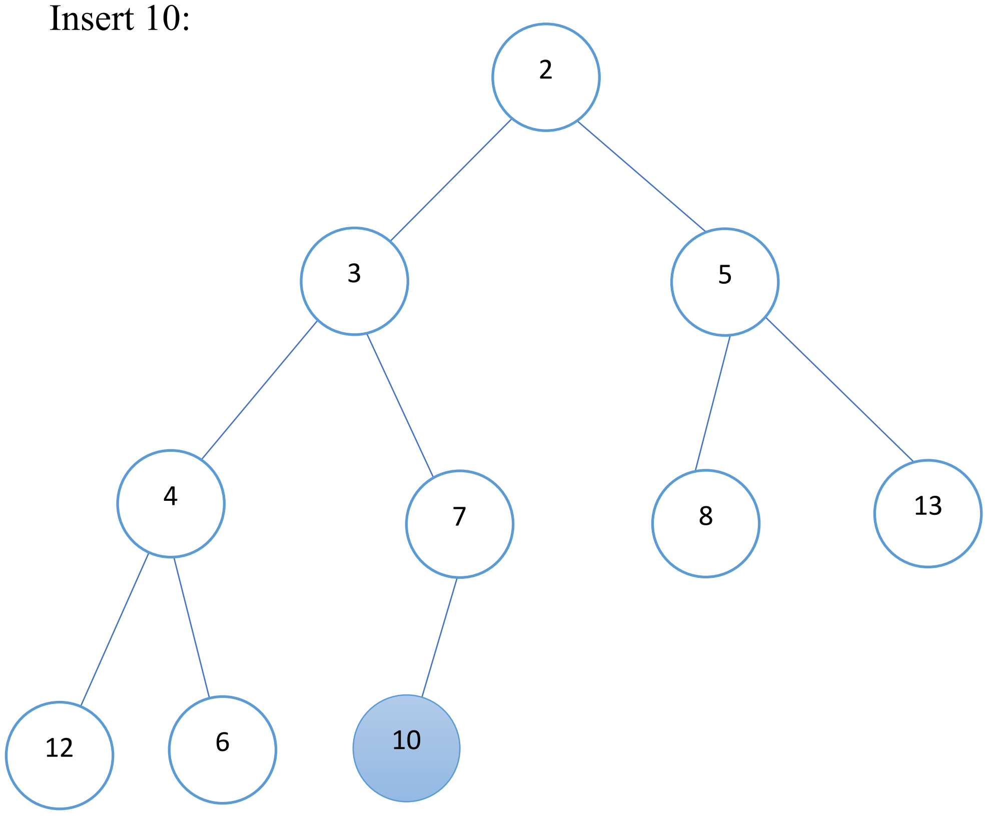
Insert 2:



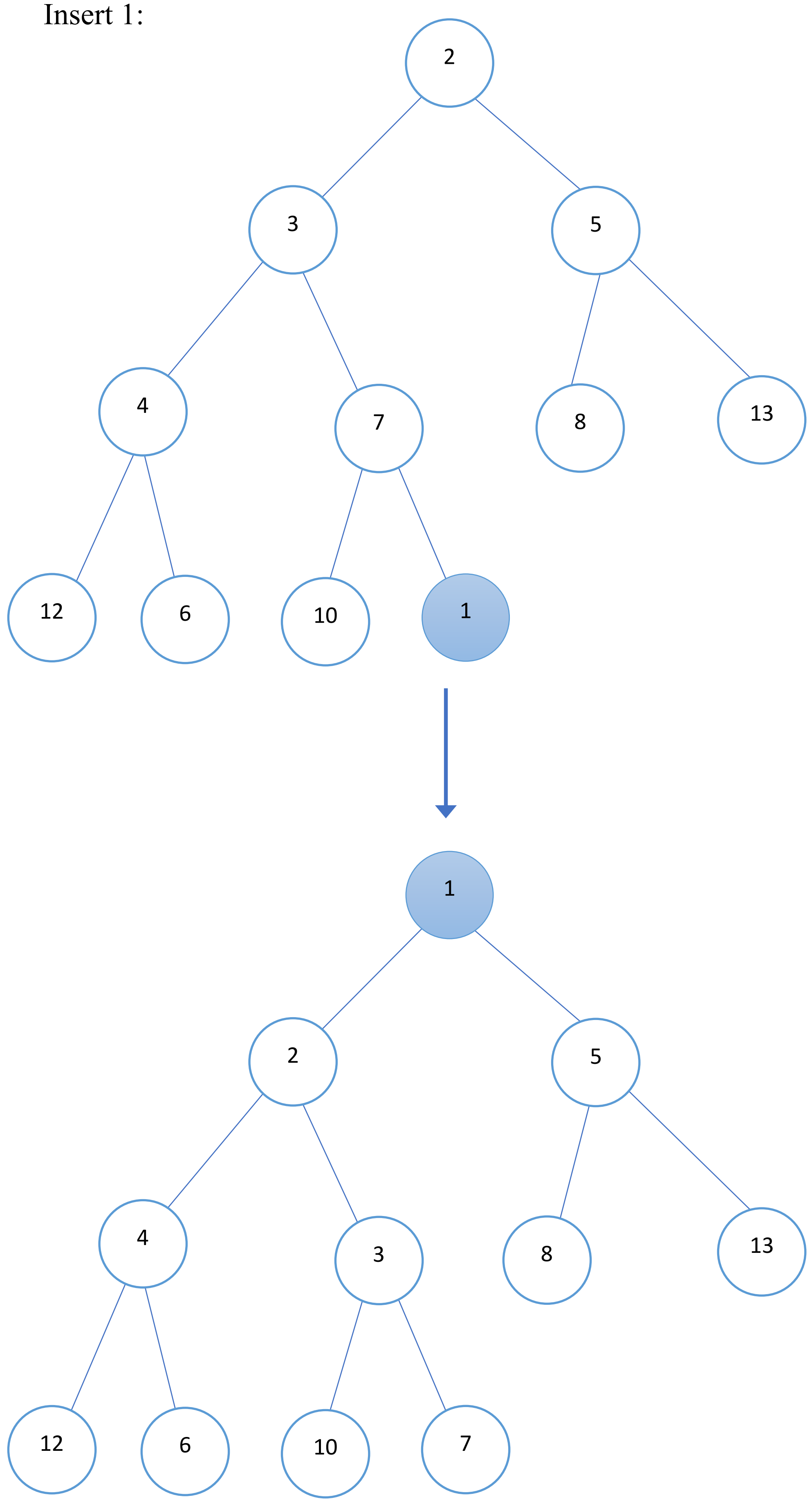
Insert 6:



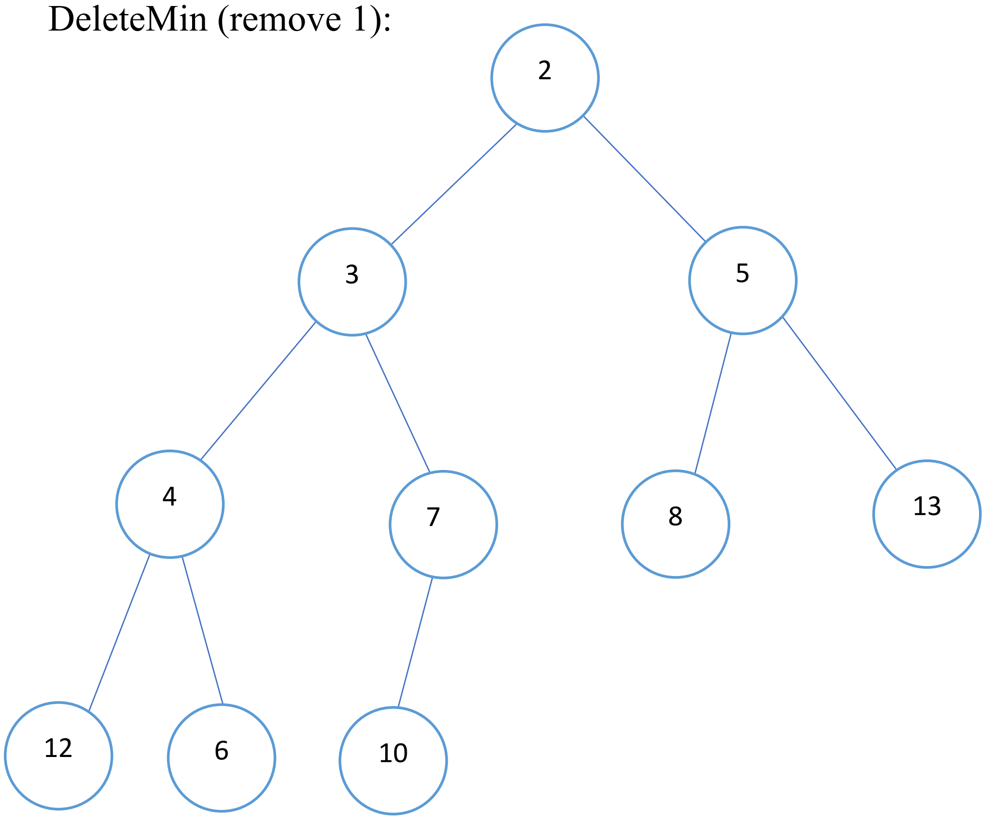
Insert 10:



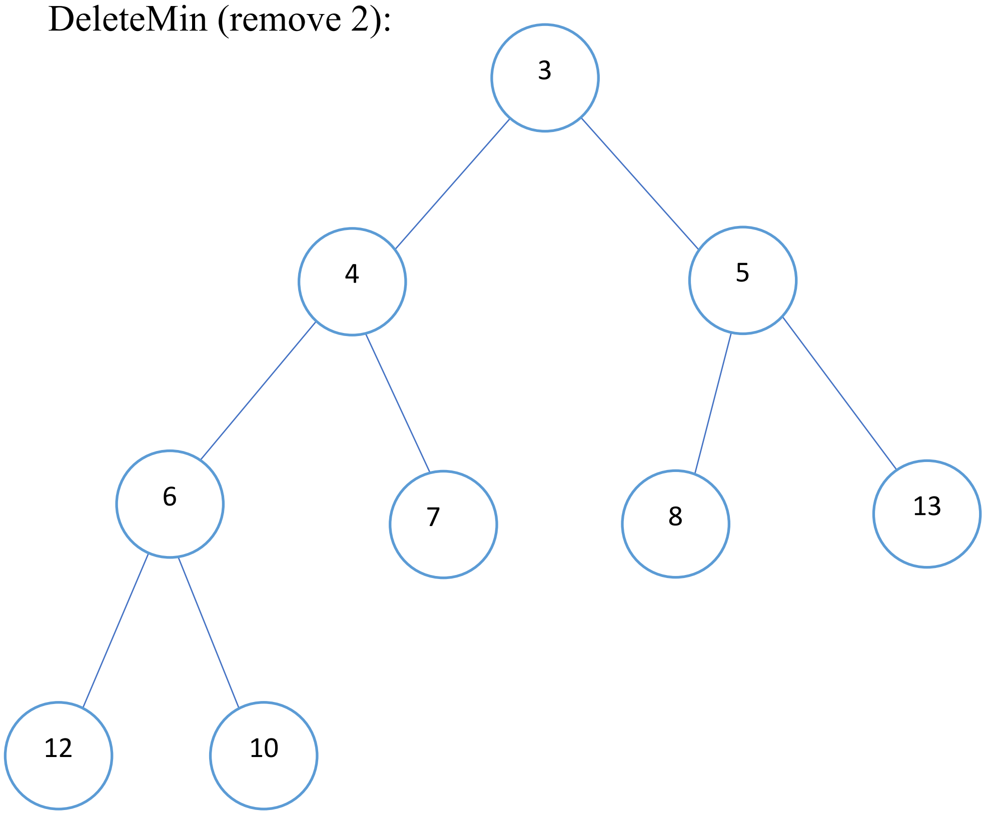
Insert 1:



DeleteMin (remove 1):

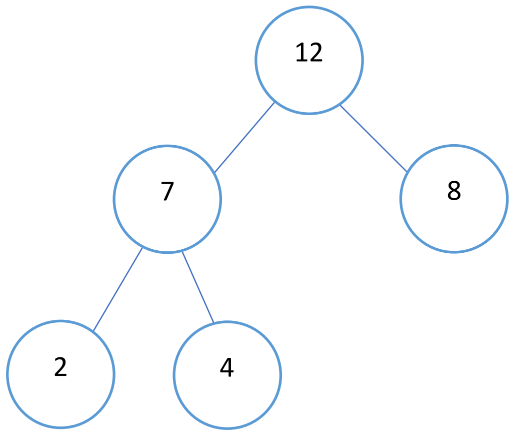


DeleteMin (remove 2):



Part c

Consider a binary max heap:



Preorder traversal: 12, 7, 2, 4, 8
Inorder traversal: 2, 7, 4, 12, 8
Postorder traversal: 2, 4, 7, 8, 12

In preorder traversal, the output is not entirely sorted. However, since it is a max heap, each parent node contains a value larger than its children. Thus, for items at index i, the parent node (if it exists) is at (i – 1) / 2. Thus each node at this position for i = 1, 2, … , n will be greater than its children, and also partially sorted in a descending order. Hence, the output can be said to be partially sorted.

Similarly, for inorder and postorder traversal, the output is again not sorted. Although, the parent node still has a greater value than that of its children, for these traversals it has a different position and index number.

Conclusively, the output printed by traversing a binary heap is not entirely sorted, regardless of the traversal order.

Part d

Let $M(h)$ be the minimum number of nodes in an AVL tree of height h. Each subtree of an AVL tree with minimum number of nodes must also have minimum number of nodes. Similarly, at least one of the two subtrees – left or right, is an AVL tree of height $h - 1$. Since the height of the left and right subtrees differ by at most 1, the other subtree must have height $h - 2$. Thus, we get the following recurrence relation:

$$M(h) = M(h-1) + M(h-2) + 1 \qquad (1)$$

Consider the bases cases where: $M(0) = 1$ and $M(1) = 2$

This recurrence relation can be solved by guessing the solution and proving it through induction. Observe that the recurrence relation in equation (1) is similar to the recurrence relation in the Fibonacci numbers. Looking at the first few numbers of the sequence in $M(h)$, we can infer the following equation $M(h) = F(h+3) - 1$ We can prove the equation by using induction.

Base cases:

$M(0) = 1$, $F(3) = 2$. So, $M(0) = F(3) - 1$
$M(1) = 2$, $F(4) = 3$. So, $M(1) = F(4) - 1$

Inductive hypothesis:

Assume that the hypothesis for $M(h) = F(h+3) - 1$ is true for $h = 1, \ldots, k$

Inductive Step:

Prove that it is also true for $h = k + 1$

$$\begin{aligned} M(k+1) &= M(k) + M(k-1) + 1 \\ &= F(k+3) - 1 + F(k+1) - 1 + 1 \\ &= F(k+4) - 1 \end{aligned}$$

Replacing $M(k)$ and $M(k - 1)$ with their equivalence from the hypothesis gives $M(k + 1) = F(k + 4) - 1$. The hypothesis is also true for $h = k + 1$. Thus, it is true for all values of h.

The minimum number of nodes in an AVL tree of height 15 can be given as:

$M(15) = F(18) - 1$

$M(15) = 2584 - 1$

$M(15) = 2583$

(head is not included in height)

Part e

For a binary tree to be a min heap, it must satisfy the following conditions:

- It should be a complete tree
- Every node's value should be lesser than or equal to the value in each of its children

The pseudocode that determines whether a given binary tree is a min heap will be as follows:

```
// returns true if binary tree is a min heap
bool isMinHeap ( root: integer, index: integer, itemCount: integer )
    // base case
    if ( root is nullptr )  {
        return true;
    }
    // not complete binary tree, false - if index range is not valid
    if  ( index is greater than or equal to itemCount )  {
        return false;
    }
    // false - if current node has a lesser value than its left or right child
    if (left child exists and leftChild <= root or right child exist and rightChild <= root) {
        return false;
    }
    // recursively check the same for left and right subtree
    // return true if both subtrees are min heaps
    return isMinHeap (root->left , 2 * i + 1 , itemCount)  and
        isMinHeap ( root->right , 2 * i + 2 , itemCount)
```

Question 2

The heap data structure defined in heap.h and heap.cpp constructs a heap using a maximum value. The destructor deletes the array to prevent memory leak. Using the insert(const int a) function, it adds a value to the heap. In case size of the array is not enough, the maximum value is doubled and a larger heap is created. The value is inserted at the bottom of the heap, and then trickled upwards to its correct position. At worst, insert has to swap entries through the entire height of the tree which gives it the worst case time complexity of O (log n).

Similarly, the popMaximum( ) function removes the maximum value of the tree. The function maximum( ) returns this value, which is stored in the root since it is a maxheap. This is done by swapping the last value of the heap with the root, and then deleting the root from the heap (which is now a leaf), and returning its value. Afterwards, the heap is rebuilt and the current root is trickled down to its correct position. At worst, this function also has to swap entries through the entire tree which gives this function the time complexity of O (log n).

The functions getCompCount( ) returns the number of key comparisons and the function isEmpty( ) returns true when the heap is empty.

Heapsort makes use of all these functions. Integers in the input file are inserted in the heap using the insert function. Then, by repeatedly calling popMaximum, we remove the items from the heap in a descending order and place them into a sequential position in the output file. This results in a sorted list of integers in ascending order. The program outputs the number of key comparisons using getCompCount for each data point.

In the worst case, the time complexity for the number of key comparisons in heap sort is equal to $2 * n * \log\ (n)$

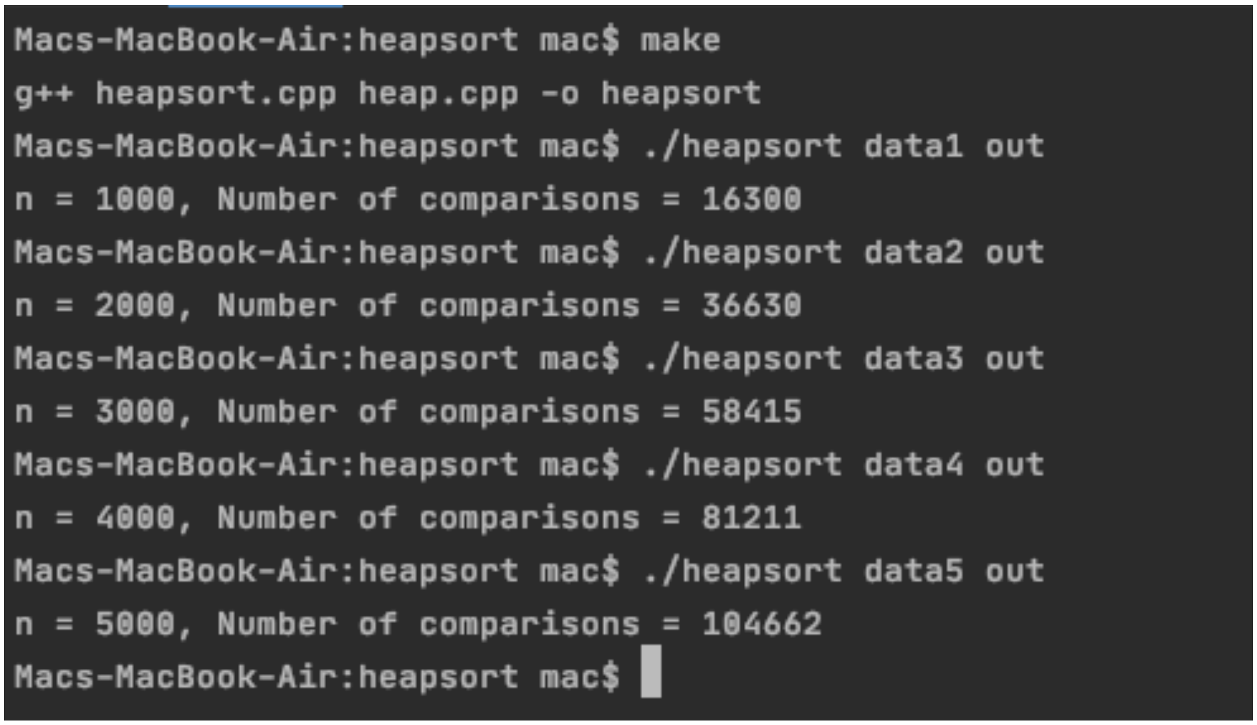
This is because the functions insert and popMaximum are called one after the other, and both have the time complexities of O (log n) as stated above.

The input files used for this program contained integers based on an average case. This is the major reason why the actual number of key comparisons is lesser than the theoretical number.

The following table shows the difference between the theoretical and actual number of key comparisons for various data points:

Data points (n)	Theoretical key comp.	Actual key comp.
1000	20000	16300
2000	44000	36630
3000	69300	58415
4000	96000	81211
5000	123000	104662

Output observed:



```
Macs-MacBook-Air:heapsort mac$ make
g++ heapsort.cpp heap.cpp -o heapsort
Macs-MacBook-Air:heapsort mac$ ./heapsort data1 out
n = 1000, Number of comparisons = 16300
Macs-MacBook-Air:heapsort mac$ ./heapsort data2 out
n = 2000, Number of comparisons = 36630
Macs-MacBook-Air:heapsort mac$ ./heapsort data3 out
n = 3000, Number of comparisons = 58415
Macs-MacBook-Air:heapsort mac$ ./heapsort data4 out
n = 4000, Number of comparisons = 81211
Macs-MacBook-Air:heapsort mac$ ./heapsort data5 out
n = 5000, Number of comparisons = 104662
Macs-MacBook-Air:heapsort mac$
```