Mannan Abdul

21801066

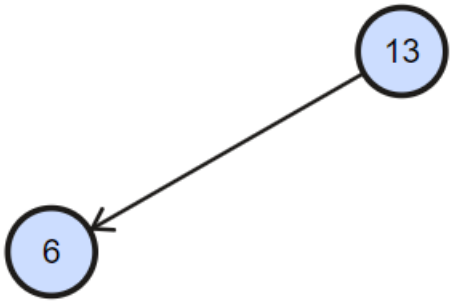
CS 202

HW 3

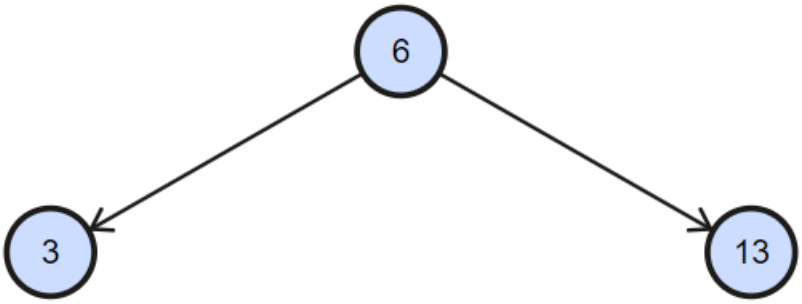
Q1) a)

Insert 13:

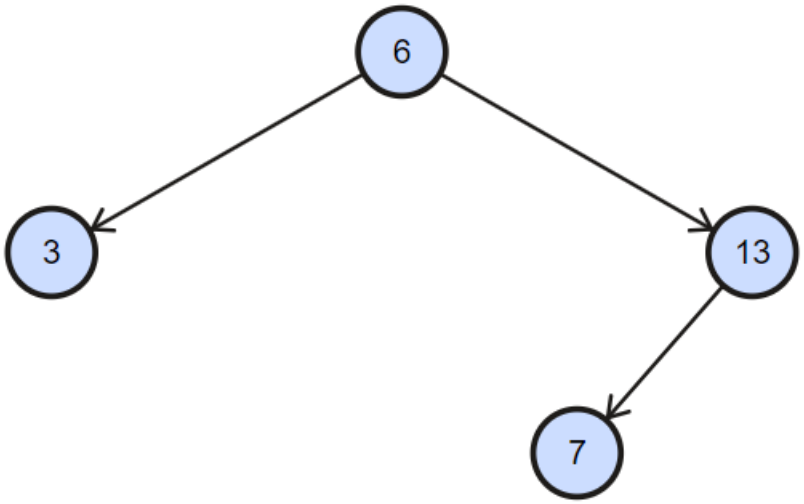
Insert 6:



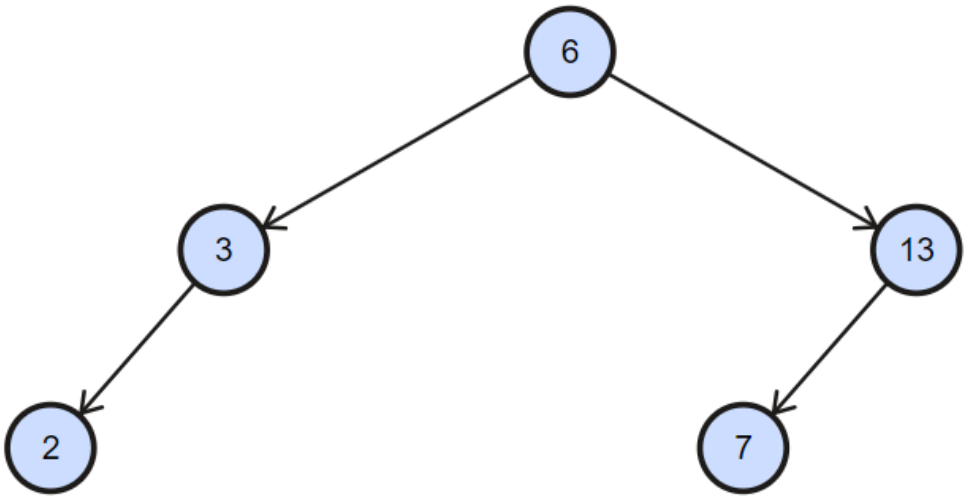
Insert 3 (Single Right Rotation):



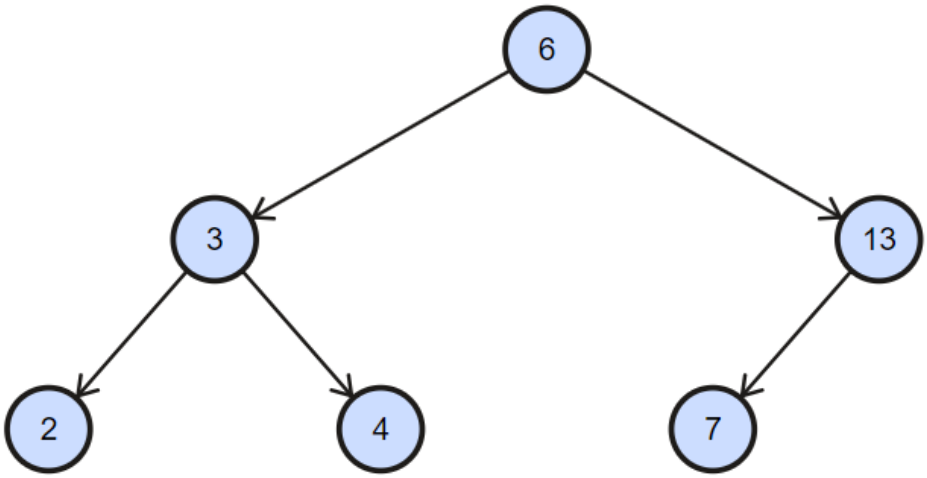
Insert 7:



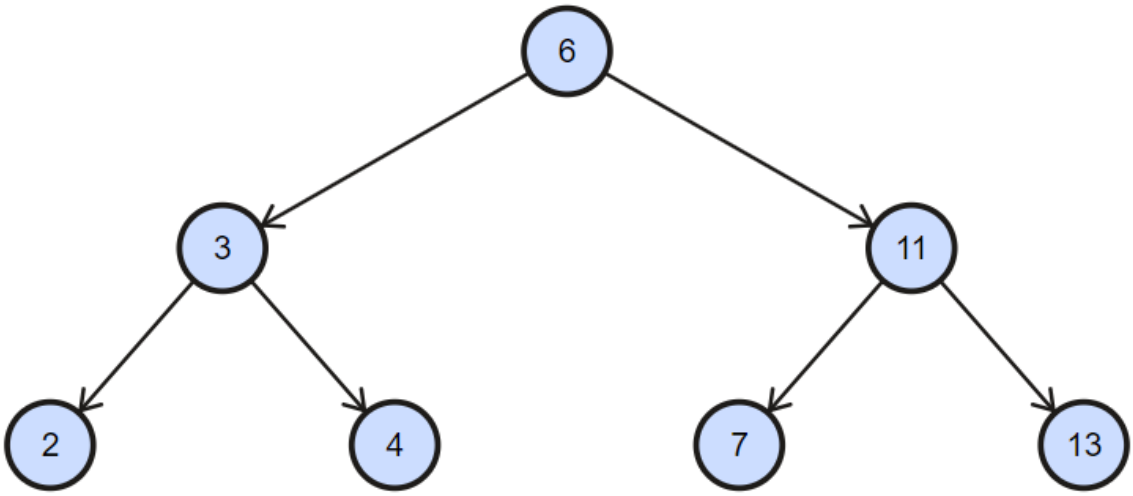
Insert 2:



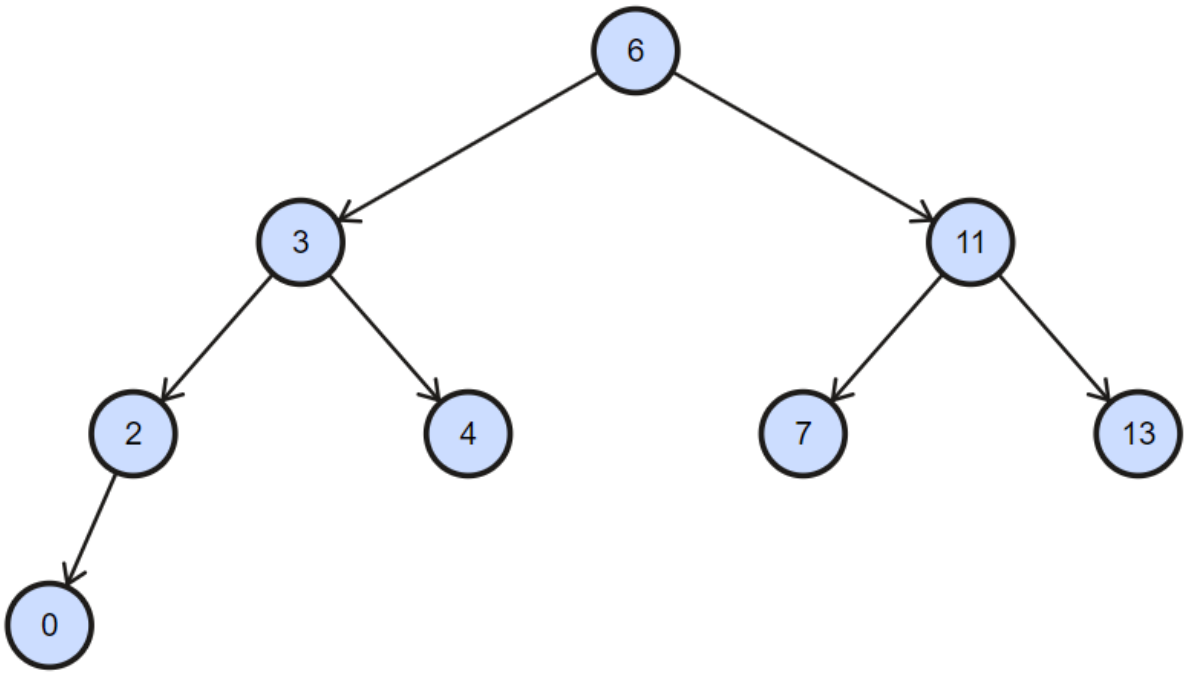
Insert 4:



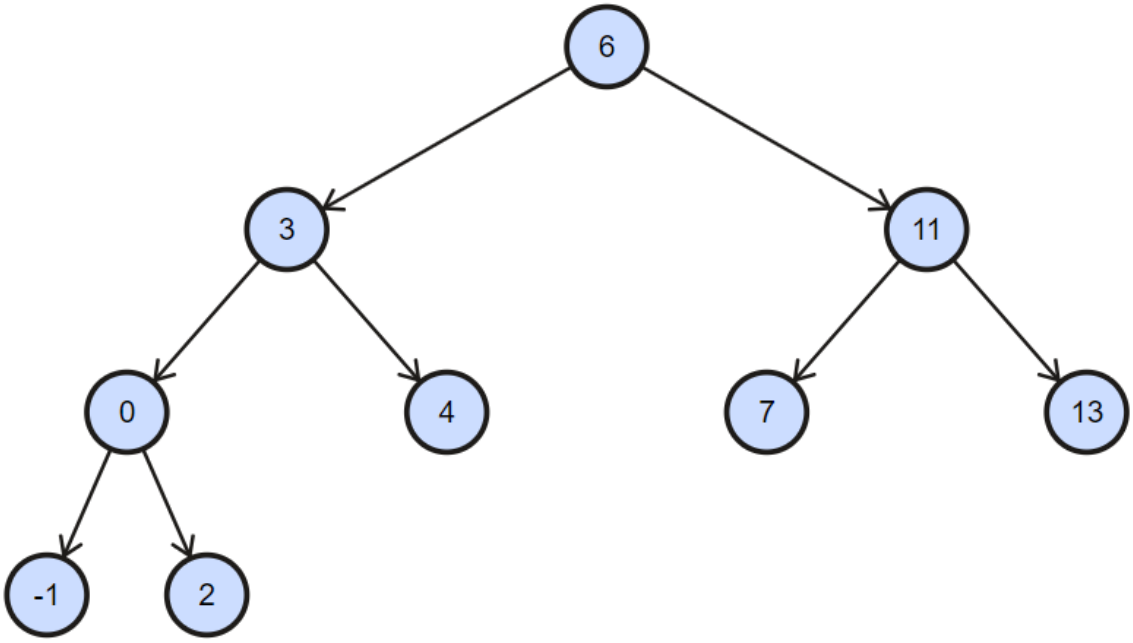
Insert 11 (Double Right-Left Rotation):



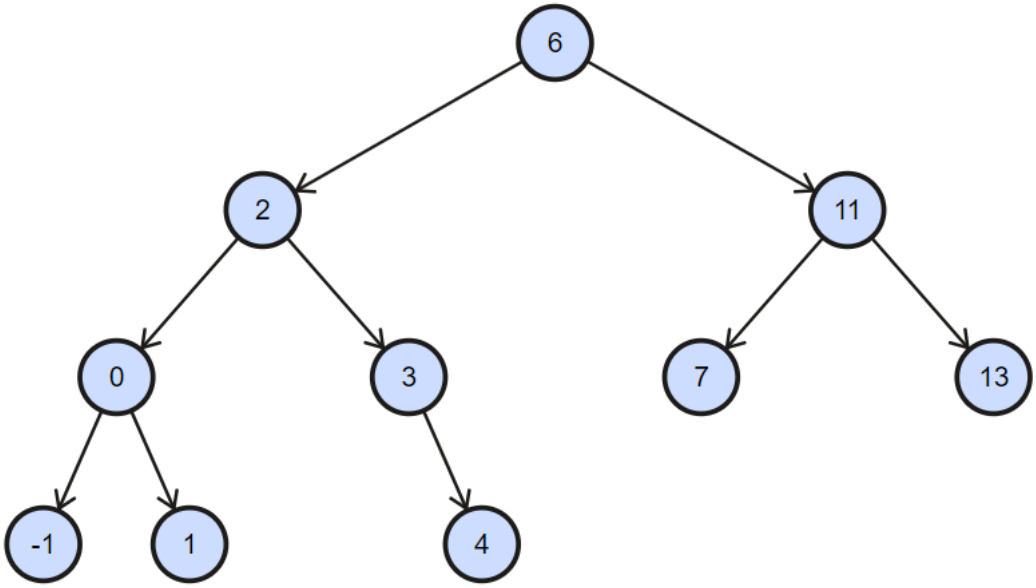
Insert 0:



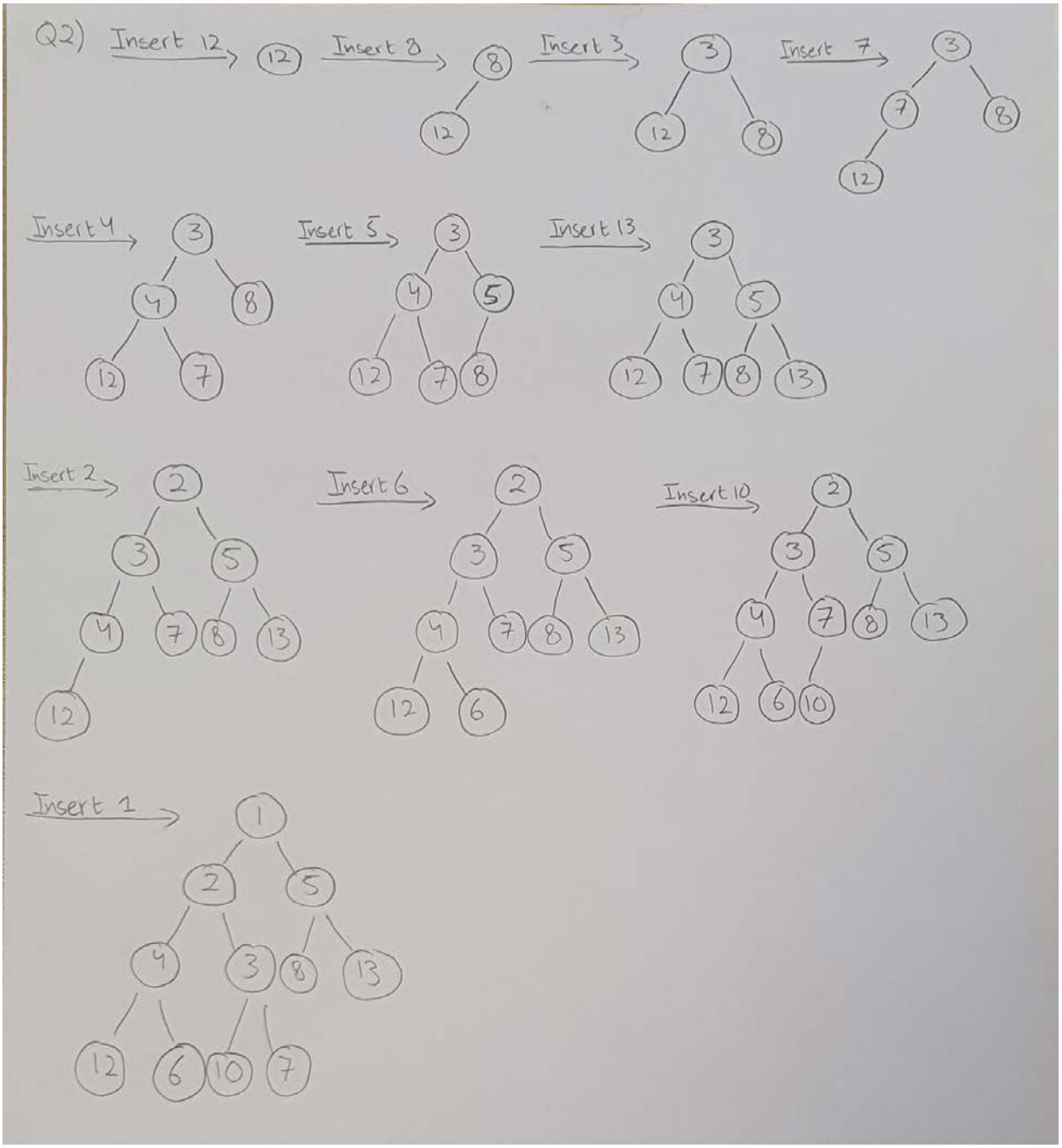
Insert -1 (Single Right Rotation):

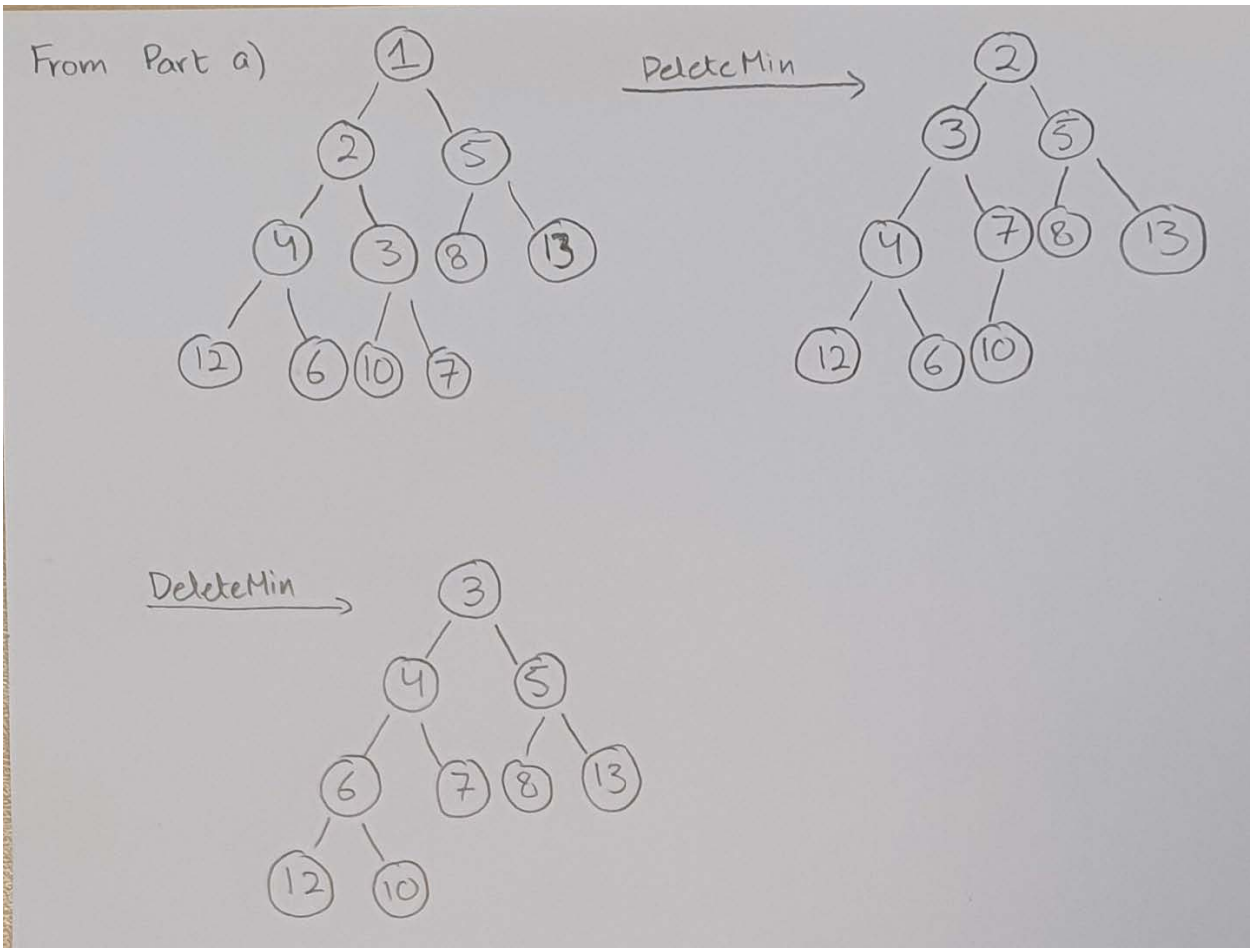


Insert 1 (Double Left-Right Rotation):



b)


Q2)
Insert 12
12
Insert 8
8
12
Insert 3
3
12
8
Insert 7
3
7
8
12
Insert 4
3
4
8
12
7
Insert 5
3
4
5
12
7
8
Insert 13
3
4
5
12
7
8
13
Insert 2
2
3
5
4
7
8
13
12
Insert 6
2
3
5
4
7
8
13
12
6
Insert 10
2
3
5
4
7
8
13
12
6
10
Insert 1
1
2
5
4
3
8
13
12
6
10
7



c)

The traversal of a heap in the pre-order, in-order or post-order will not give us a sorted output, this is because when insertion in the heap is done, the aim is not to sort the new input but rather to maintain a complete binary tree which is why no matter the traversal method, the output would never be a sorted one. A heap while ordered in a weak sense is not sorted.

If we look at the min heap from Q2, we can see the traversals of the heap produce the following, none of which are sorted outputs.

Pre-Order: 1, 2, 4, 12, 6, 3, 10, 7, 5, 8, 13

In-Order: 12, 4, 6, 2, 10, 3, 7, 1, 8, 5, 13

Post-Order: 12, 6, 4, 10, 7, 3, 2, 8, 13, 5, 1

d)

The minimum number of nodes in an AVL tree of height h can be found recursively by using the basic property of an AVL tree, which is the height of the 2 subtrees of a node cannot differ by more than 1.

$N(h) = N(h - 1) + N(h - 2) + 1$ where $n > 2$ and $N(1) = 1$, $N(2) = 2$ and $N(3) = 4$

This calculates the heights of the 2 subtrees while accounting for the node and the AVL conditions.

Minimum number of Nodes in AVL tree of height 15:

$N(15) = N(14) + N(13) + 1 \Rightarrow 986 + 609 + 1 = 1596$

$N(14) = N(13) + N(12) + 1 \Rightarrow 609 + 376 + 1 = 986$

$N(13) = N(12) + N(11) + 1 \Rightarrow 376 + 232 + 1 = 609$

$N(12) = N(11) + N(10) + 1 \Rightarrow 232 + 143 + 1 = 376$

$N(11) = N(10) + N(9) + 1 \Rightarrow 143 + 88 + 1 = 232$

$N(10) = N(9) + N(8) + 1 \Rightarrow 88 + 54 + 1 = 143$

$N(9) = N(8) + N(7) + 1 \Rightarrow 54 + 33 + 1 = 88$

$N(8) = N(7) + N(6) + 1 \Rightarrow 33 + 20 + 1 = 54$

$N(7) = N(6) + N(5) + 1 \Rightarrow 20 + 12 + 1 = 33$

$N(6) = N(5) + N(4) + 1 \Rightarrow 12 + 7 + 1 = 20$

$N(5) = N(4) + N(3) + 1 \Rightarrow 7 + 4 + 1 = 12$

$N(4) = N(3) + N(2) + 1 \Rightarrow 4 + 2 + 1 = 7$ Back Substituting

Therefore min number of nodes in an AVL tree of height 15 would be 1596.

e)

```
boolean isHeap
   index = 0
   return isHeap(root, index)


boolean isHeap(root, index)
   if root is equal to NULL
      return TRUE


   if index is equal to or bigger than number of nodes
      return FALSE


   if left or right child of root is less than or equal to root
      return FALSE


   return isHeap(left child of root, 2 * index + 1) AND isHeap(right child of root, 2 * index + 2)
```

Q2)

Heap Data Structure:

The heap data structure is implemented as in the lecture slides, using an array to keep the index of the items in the Heap. It is a max heap because it makes it easier to implement the max and popMaximum functions while making the it faster at O(1) time.

For the Heap, we have an array named items. This array will store all the numbers in the Heap. We have a variable named size to store the size of the heap and a nother variable named compCount that counts the number of comparisons made during a sort, I will explain further in the sorting section.

heap(): the normal constructor for the heap, sets both size and compCount to 0

bool heapIsEmpty() const: This function checks if the heap is empty or not by checking the size variable

void insert(const int a): This function adds a new integer to the heap, it checks whether the heap is full or not beforehand, a value that can be modified in the heap class. Then it inserts the value to the last index and then bubbles it to its proper position.

int maximum(): returns the max value or root of the heap

int popMaximum(): returns the max value or the root of the heap but then also deletes it creating 2 disjoint trees, it then places the value at last index to the root and then calls on the help rebuild function to sort it to its appropriate place.

void heapRebuild(int root): this is a recursive function. First the index of the left child of the root is identified, after that we find the index of the right child, we then compare to check which value is bigger. We then compare the bigger value with the root, it is smaller, the function ends here and increments compCount by 2 for the comparisons made, else we swap the values of the root and the greater child and recursively call the function again to solve a smaller sub problem.

int getSize(): returns the size of the heap

void displayHeap(): outputs the heap to standard output by index, used for testing purposes

void resetCompCount(): resets compCount

int getCompCount(): returns compCount

Heapsort:

The heapsort file has a main driver that takes 2 arguments, the name of the input and output file.

void generateHeap(string filename, heap& maxHeap): this function reads all the integers from the input file and inserts them into the heap one after another.

void heapsort(heap maxHeap, const int n, int* sorted): Using the pseudocode form the slides, this heapsort is implemented. Because of the way the insert function is implemented, after the file is read we have a heap already, so we don't need to repeat the step of making the array a heap again. After that, we use the popMaximum function to put the max value into an array called sorted, the way the function is implemented, it will automatically rebuild the heap after the pop. We repeat until the heap is empty and all the values are now sorted. To remove any errors in number of comparisons, the function also resets the compCount before and after all the work is done.

void writeOutput(const int size, const string filename, int* sorted): writes the sorted array into another file called output.txt in an ascending order.

Comparisons:

In theory, the worst case for the number of comparisons a heapsort makes is O(nlogn), but the actual number of comparisons required is higher. This is because for every call of the heaprebuild function, where the comparisons take place, there are 2 comparisons, one comparison between the right and left child of the root and the other between the bigger child and the root, and regardless of whether we execute the if statements, a comparison is still conducted.

Input Samples:

1. N = 5, No of comparisons = 8.
2. N = 10, No of comparisons = 28.
3. N = 15, No of comparisons = 54.
4. N = 20, No of comparisons = 86.
5. N = 25, No of comparisons = 124.