# Problem 1

## Part a.

The resulting tree is as follows:

50

20 60

10 40 70

15 30 65 80

25 34 76

## Part b.

Printing with an **in-order traversal algorithm** (with each printout separated by a comma and a space) would result in the following output:  
*10, 15, 20, 25, 30, 34, 40, 50, 60, 65, 70, 76, 80*

Similarly, printing with a **pre-order traversal algorithm** (where the left subtree is processed before the right subtree) would result in the following output:  
*50, 20, 10, 15, 40, 30, 25, 34, 60, 70, 65, 80, 76*

Similarly, printing with a **post-order traversal algorithm** (where the left subtree is processed before the right subtree) would result in the following output:  
*15, 10, 25, 34, 30, 40, 20, 65, 76, 80, 70, 60, 50*

## Part c.

After deleting the nodes 30 and 20 (in order) following a simple deletion algorithm, we get the following tree:

50

25 60

10 40 70

15 34 65 80

76

# Problem 2

## Part a.

*struct Node {*

*Node\* m\_parent;*

*Node\* m\_leftChild;*

*Node\* m\_rightChild;*

*int data;*

*};*

## Part b.

Following is the desired pseudocode of a recursive function *insert*, that returns a pointer to a node and takes parameters *subTreePtr* and *newNodePtr*, both pointers to nodes. In each call of *insert*, the function returns either *newNodePtr* (if we’ve reached an empty leaf subtree) or just *subTreePtr* if node is to be inserted at some deeper level.

if *subTreePtr* is *nullptr*:

return *newNodePtr*

else:

create a *childPtr* variable to keep track of the resulting child of the subTree

if data in *subTreePtr* > data in *newNodePtr*:

find new child pointer insubTree’s left subtree with a recursive call of *insert*

set *childPointer* to subTree’s new child pointer

set subTree’s left child pointer to *childPtr*[[1]](#footnote-1)

else if data in *subTreePtr* < data in *newNodePtr*:

find new child pointer insubTree’s left subtree with a recursive call of *insert*

set subTree’s left child pointer to *childPtr*

set subTree’s right child pointer to *childPtr1*

if *childPtr* is *newNodePtr*:

set *newNodePtr*’s parent pointer to *subTree* // parent found

return *subTreePtr*

This algorithm traverses into the tree until the appropriate position for the new node is found, returns a pointer to that new node so its new parent may update its child pointer, update’s the new node’s parent pointer, and then all the other recursive calls return the same child pointers the sub-trees previously had. The end result is a modified subtree with the new node inserted in the appropriate position and two effective pointer changes: the parent pointer of the new node, and the appropriate child pointer of the parent.

# Problem 3

## Part a.

The resulting tree is as follows:

7

3 5

0 2 4

## Part b.

The heap from part a. can be represented in array form as follows: { 7, 3, 5, 0, 2, 4 }

## Part c.

After removing one more item, the heap can be represented in array form as follows: { 5, 3, 4, 0, 2 }

# Problem 4

1. O(C + S)
2. O(logC + S)
3. O(logC + logS) i.e. O(logSN)
4. O(logS)
5. O(1)
6. O(logC + S)
7. O(SlogS)
8. O(ClogS)

# Problem 5

## Part b.

A one-parameter overload of *listAll* wouldn’t have worked as recursive function since for each call to *listAll*, we not only need to print the name of the class passed in but also, before it, the names of its base classes in order. Without the ability to specify the names of such base classes in a recursive call (via, say, a string parameter), the function would only print out the name of each class one by one.

1. Note: this is either *newNodePtr* if it is to be inserted as a child of the current subtree, or a pointer to the same left/right child of subTree if the new node is to be inserted at some deeper level [↑](#footnote-ref-1)