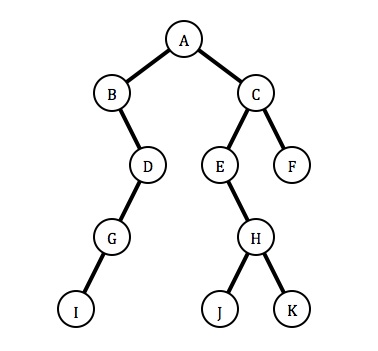
**Vipul Krishna**

**Assignment 4**

**INFO 6205 – Program Structures and Algorithms**

Question 1:

Given binary tree:



Inorder Traversal - left, root, right = B, I, G, D, A, E, J, H, K, C, F

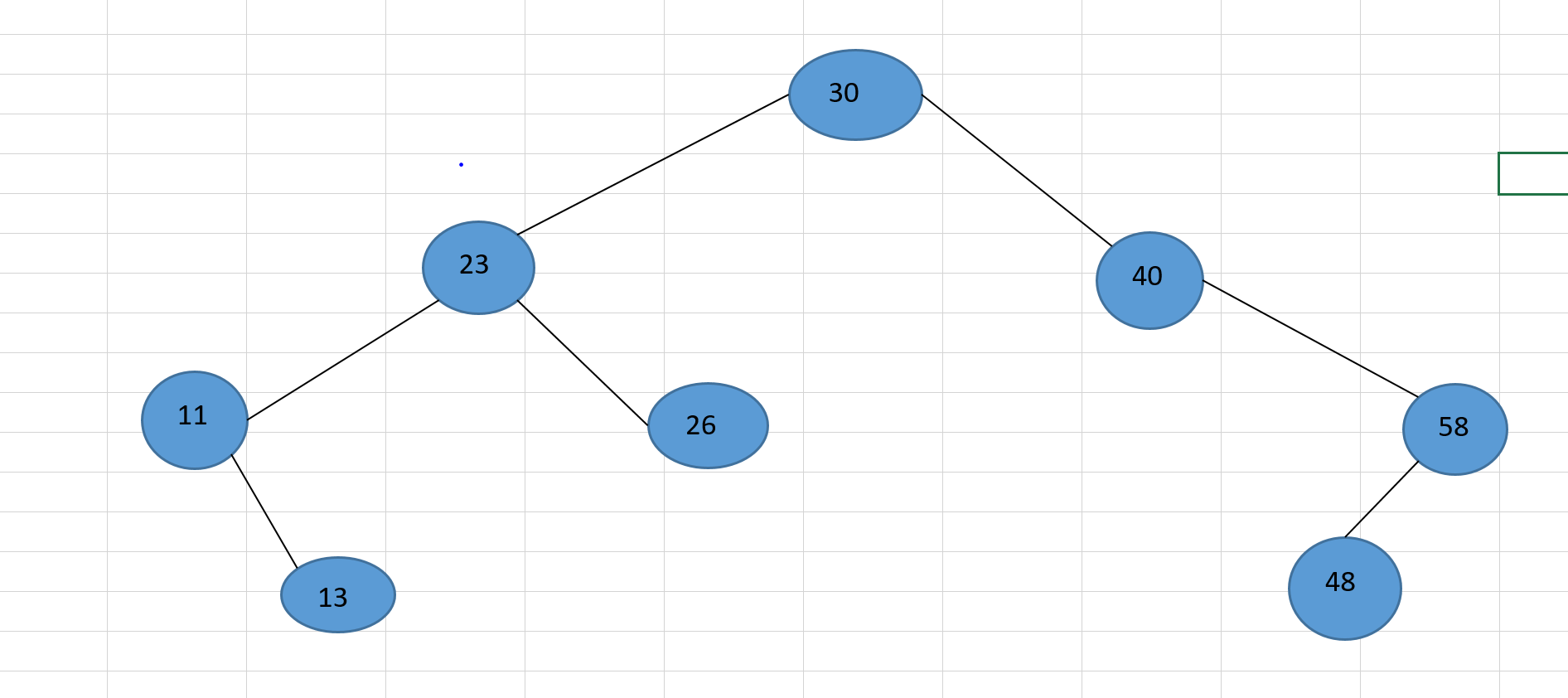
Preorder Traversal – root, left, right = A, B, D, G, I, C, E, H, J, K, F

Postorder Traversal – left, right, root = I, G, D, B, J, K, H, E, F, C, A

Question 2:

Question 3:

Given items = {30, 40, 23, 58, 48, 26, 11, 13}



Note: The java code for this insertion is also included in the java project folder

Question 4:

1. The maximum height of the binary tree = The number of edges between the root of the tree and its farthest leaf.

Therefore, in this case the height ‘h’ = 3 because the number of edges between the root 30 and the farthest leaf(Leaf 13 and Leaf 48) is 3

Height of a binary search tree is also given by the expression: h = Log(n) where n is the number of nodes in the tree.

Here n = 8; So, h = Log (8)

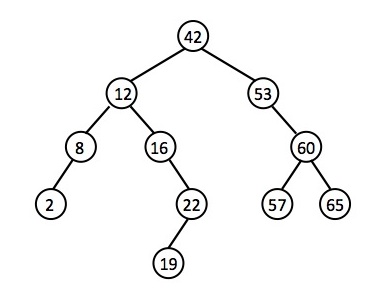
1. Time complexity of the tree = O(Height of the tree) = O(h) = O(Log(n)) = O(Log(8)).

This is because for any operation, we would be dealing with either the left half of the tree or the right half and not all the elements of the tree.

Note – There are other constants involved in calculating the time complexity.

Question 5:

Given BST:



While deleting nodes from a BST, 3 conditions can arise:

1. Deleting a leaf node
2. Deleting a node with one child
3. Deleting a node with two children (scenario of deleting the root node covered in this case)

Main Methods Used for Deletion:

1. Public void deleteKey(int key){

Root = deleteRec(root, key);

}

This method mainly calls the deleteRec method passing the root and key as parameter.

1. Public Node deleteRec(Node root, int key){

if (root == null) return root;

if (key < root.key)

root.left = deleteRec(root.left, key);

else if (key > root.key)

root.right = deleteRec(root.right, key);

else{

if (root.left == null)

return root.right;

else if (root.right == null)

return root.left;

root.key = minValue(root.right);

root.right = deleteRec(root.right, root.key);

}

return root;

}

1. Public int minValue(Node root){

While(root != null{

Root = root.left;

}

Return root.key; 🡪 This will be the minimum value of the tree with the passed root

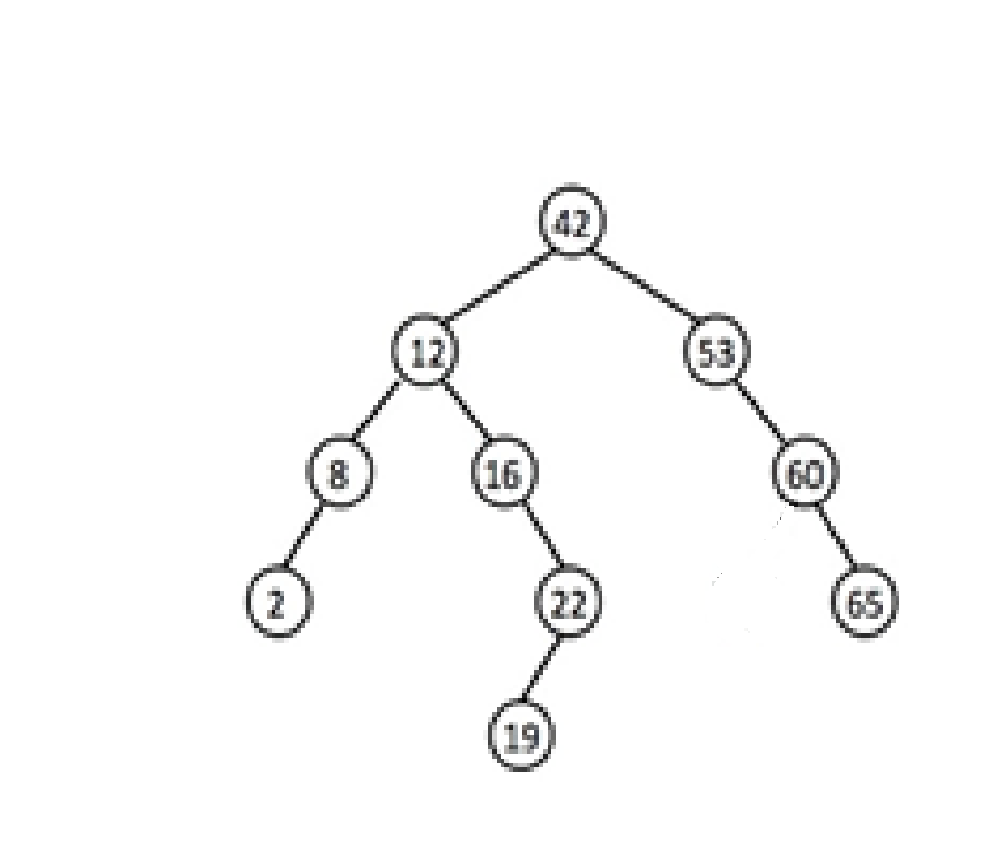
}

Delete Operations:

Case 1:

Delete 57 – 57 is a leaf node. The concept here is that we just need to detach this node from the tree. For this, first we should traverse the tree so that we reach this node and then just detach it from the tree.

1. Pass root and 57 to the deleteRec method.
2. Traversal occurs till we reach this node(by recursive calls of the same method) and enter the else condition of the above method.
3. Since this is a leaf node, its left node will be null. According to the condition, if the left node is null, we return the right node. Notice the right node is also null because it is a leaf node.
4. Hence, this returned null value gets assigned to the left of node of 60 because of the statement: root.left = deleteRec(root.left, key);
5. By this way, the leaf node 57 is removed from the given tree.



Case 2:

Delete 60 – Since 57 is already deleted, 60 is a node with just one child now. The concept here would be to detach node 60 from node 53, and attach node 65 to the right of node 53.

1. Pass root and 60 to the deleteRec method.
2. Traversal occurs till we reach this node(by recursive calls of the same method) and enter the else condition of the above method.
3. Node 60 has one right child which is node 65. So this statement will be executed

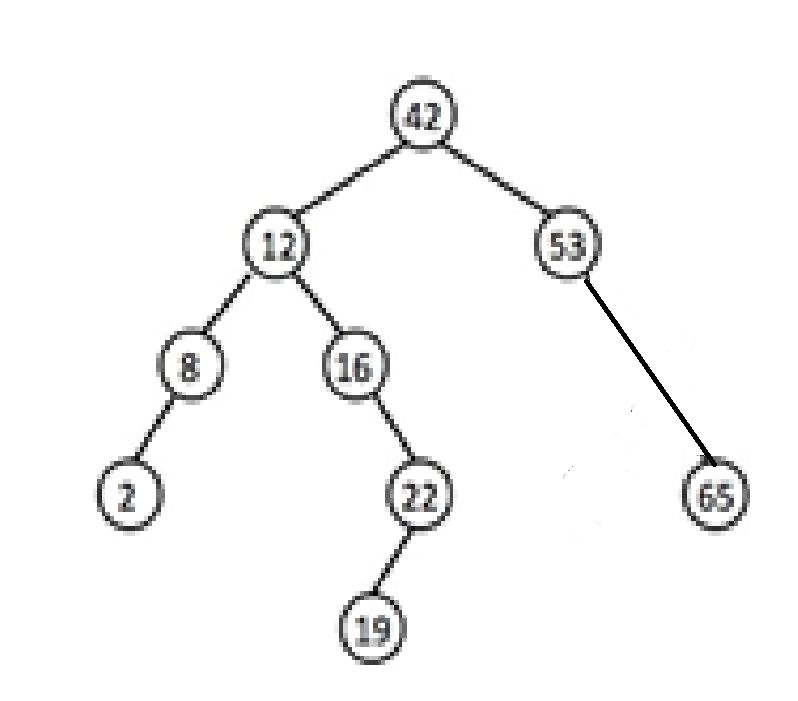
if (root.left == null) return root.right;

Here root right is node 65.

1. This returned node 65 will be assigned to the right of node 53 by the following statement

root.right = deleteRec(root.right, key);

1. Thus, node 60 is detached from node node 53 and node 65 now becomes the right node of node 53. Hence node 60 is removed from the tree.



Case 3:

Delete 42 – This node is the root of the tree. It falls in the category of deleting a node with two children. This case needs more adjustments than the first two cases. Here first we traverse to the node with the minimum value in the right subtree (It can be done by getting the max value from left subtree and replacing the node value with this max value from left subtree) and return the minimum value using the minValue method. Then we assign this value to the root value and detach the node with the minimum value on the right subtree from the BST. The detailed steps are:

1. Pass root and 60 to the deleteRec method.
2. Since we are deleting the root itself, the execution goes directly to the else part of the method and the first line executed is root.key = minValue(root.right);
3. The above method returns the value 53 and assigns it to the key of the root node
4. The next step is root.right = deleteRec(root.right, root.key);

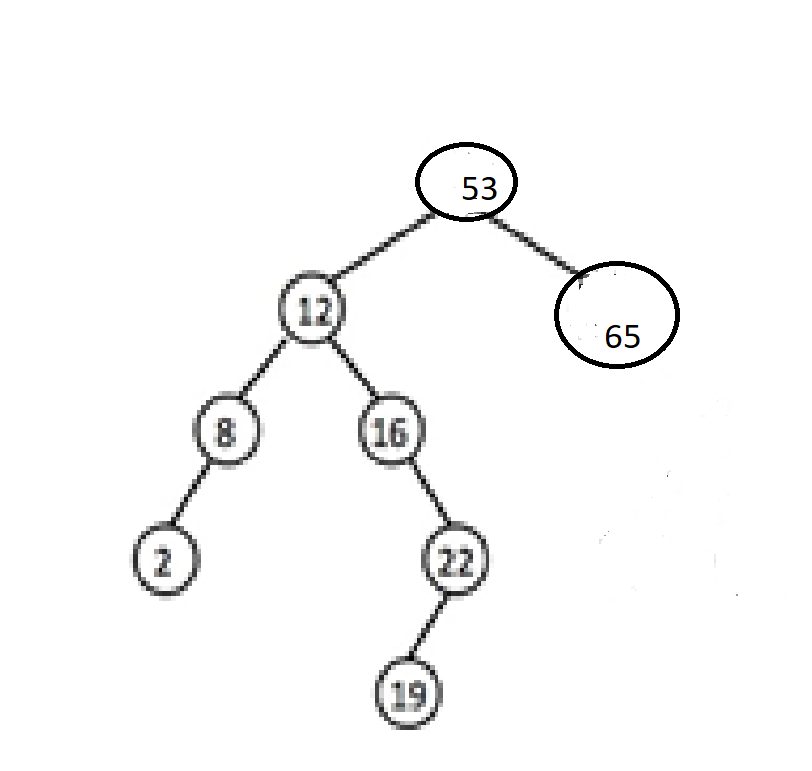
Here we pass node 53 and the value 53 to delete rec method. The same method is called and the execution enters the else part directly since both the values are equal.

1. The next execution occurs on

if (root.left == null)

return root.right; // The left node of node 53 is null. Right node is node 65 which is being returned.

1. The returned node 65 now becomes the right node of the root(see step iv. This is the step from where this method was called).
2. Hence the root key changes to 53(step iii) and node 65 becomes the right node of the root.
3. Thus 42 is deleted.



Question 6:

Show step-by-step Bubble sort for {5, 1, 12, -4, 20, -9}

Solution:

Bubble sort is the simplest sorting algorithm. In this algorithm, we start the iteration from the very first element.

We compare the element to the next element. If the next element is smaller, we swap the indices of the two elements under comparison. We continue the comparison for all the elements under consideration till we get the sorted array.

Given array = {5, 1, 12, -4, 20, -9}

First Iteration:

1. (5, 1, 12, -4, 20, -9) 🡪 (1, 5, 12, -4, 20, -9); 5>1, so the elements are swapped
2. (1, 5, 12, -4, 20, -9) 🡪 (1, 5, 12, -4, 20, -9); 5<12, so no change
3. (1, 5, 12, -4, 20, -9) 🡪 (1, 5, -4, 12, 20, -9); 12>-4, so the elements are swapped
4. (1, 5, -4, 12, 20, -9) 🡪 (1, 5, -4, 12, 20, -9); 12<20, so no change
5. (1, 5, -4, 12, 20, -9) 🡪 (1, 5, -4, 12, -9, 20); 20>-9, so elements are swapped

Second Iteration

1. (1, 5, -4, 12, -9, 20) 🡪 (1, 5, -4, 12, -9, 20); 1<5, so no change
2. (1, 5, -4, 12, -9, 20) 🡪 (1, -4, 5, 12, -9, 20); 5>-4, so elements swapped
3. (1, -4, 5, 12, -9, 20) 🡪 (1, -4, 5, 12, -9, 20); 5<12, so no change
4. (1, -4, 5, 12, -9, 20) 🡪 (1, -4, 5, -9, 12, 20); 12>-9, so elements swapped
5. (1, -4, 5, -9, -12, 20) 🡪 (1, -4, 5, -9, 12, 20); 12<20, so no change

Third Iteration

1. (1, -4, 5, -9, 12, 20) 🡪 (-4, 1, 5, -9, 12, 20); 1>-4, so elements swapped
2. (-4, 1, 5, -9, 12, 20) 🡪 (-4, 1, 5, -9, 12, 20); 1<5, so no change
3. (-4, 1, 5, -9, 12, 20) 🡪 (-4, 1, -9, 5, 12, 20); 5>-9, so elements swapped
4. (-4, 1, -9, 5, 12, 20) 🡪 (-4, 1, -9, 5, 12, 20); 5<12, so no change
5. (-4, 1, -9, 5, 12, 20) 🡪 (-4, 1, -9, 5, 12, 20); 12<20 so no change

Fourth Iteration

1. (-4, 1, -9, 5, 12, 20) 🡪 (-4, 1, -9, 5, 12, 20); -4<1, so no change
2. (-4, 1, -9, 5, 12, 20) 🡪 (-4, -9, 1, 5, 12, 20); 1>-9, so elements swapped
3. (-4, -9, 1, 5, 12, 20) 🡪 (-4, -9, 1, 5, 12, 20); 1<5, so no change
4. (-4, -9, 1, 5, 12, 20) 🡪 (-4, -9, 1, 5, 12, 20); 5<12, so no change
5. (-4, -9, 1, 5, 12, 20) 🡪 (-4, -9, 1, 5, 12, 20); 12<20, so no change

Fifth Iteration

1. (-4, -9, 1, 5, 12, 20) 🡪 (-9, -4, 1, 5, 12, 20); -4>-9, so elements swapped
2. (-9, -4, 1, 5, 12, 20) 🡪 (-9, -4, 1, 5, 12, 20); -4<1, so no change
3. (-9, -4, 1, 5, 12, 20) 🡪 (-9, -4, 1, 5, 12, 20);1<5, so no change
4. (-9, -4, 1, 5, 12, 20) 🡪 (-9, -4, 1, 5, 12, 20); 5<12, so no change
5. (-9, -4, 1, 5, 12, 20) 🡪 (-9, -4, 1, 5, 12, 20); 12<20, so no change

Sixth Iteration

1. (-9, -4, 1, 5, 12, 20) 🡪 (-9, -4, 1, 5, 12, 20); -9<-4, so no change
2. (-9, -4, 1, 5, 12, 20) 🡪 (-9, -4, 1, 5, 12, 20); -4<1, so no change
3. (-9, -4, 1, 5, 12, 20) 🡪 (-9, -4, 1, 5, 12, 20);1<5, so no change
4. (-9, -4, 1, 5, 12, 20) 🡪 (-9, -4, 1, 5, 12, 20); 5<12, so no change
5. (-9, -4, 1, 5, 12, 20) 🡪 (-9, -4, 1, 5, 12, 20); 12<20, so no change

The sixth iteration did not have any swapping done. There now the array is sorted.

Sorted Array = {-9, -4, 1, 5, 12, 20}

Sample Code:

Method sort(array){

len = array.length

Boolean swap = false;

for(i=0; i<len-1; i++){

for(j=0; j<len-1-I; j++){

if (array[j] > array[j+1]{

temp = array[j];

array[j] = array[j+1]

array[j+1] = temp;

swap = true;

}

}

If(swap == false)

break;

}

}

Using a Boolean flag customizes the sort operation. If the array gets sorted at any iteration, then there is no need to iterate further. This is checked by the Boolean flag. The time complexity in this case gets lowered.