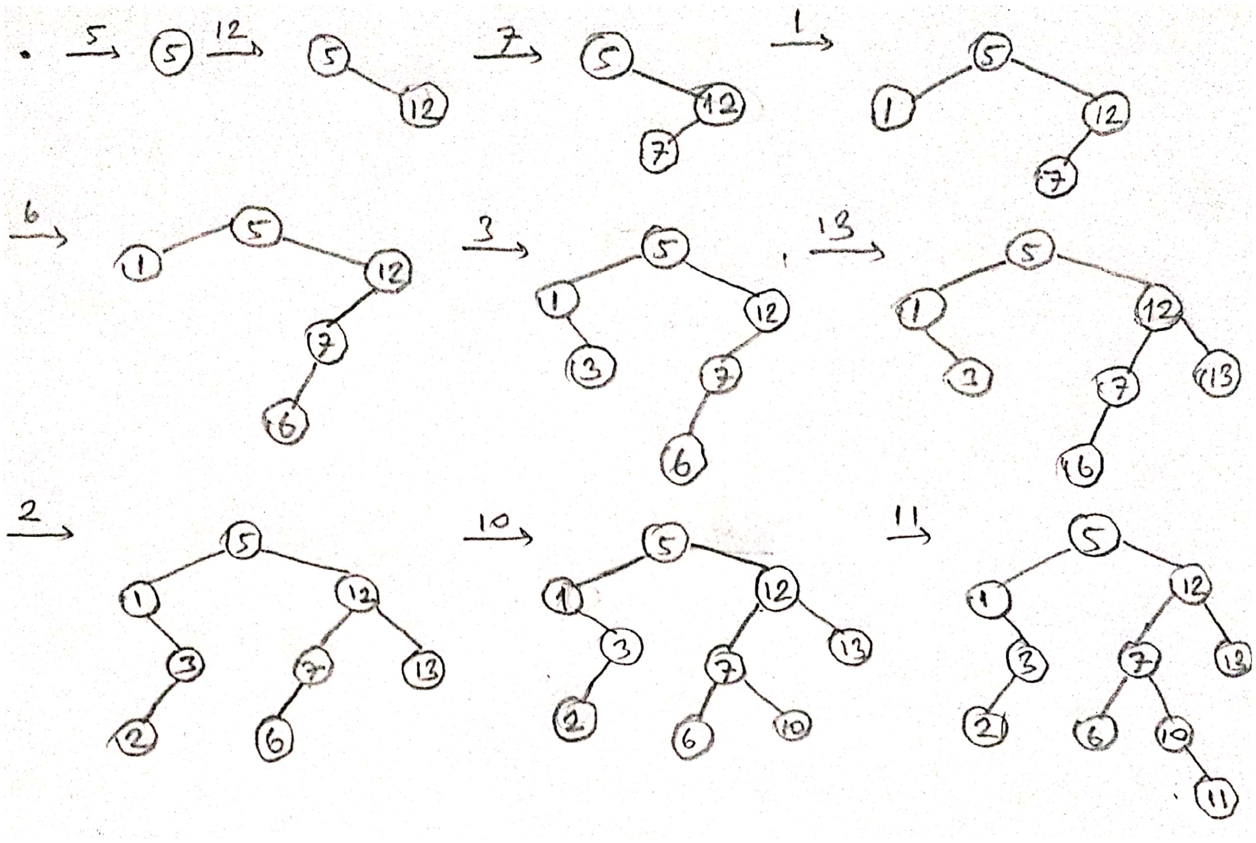
```
/**
* Title : Binary Search Trees
* Author : Munib Emre Sevilgen
* ID: 21602416
* Section : 1
* Assignment : 2
* Description : Part a, b, and c of the Question 1 and the Question 3
*/
```

Question 1:

(a) Input : 5, 12, 7, 1, 6, 3, 13, 2, 10, 11



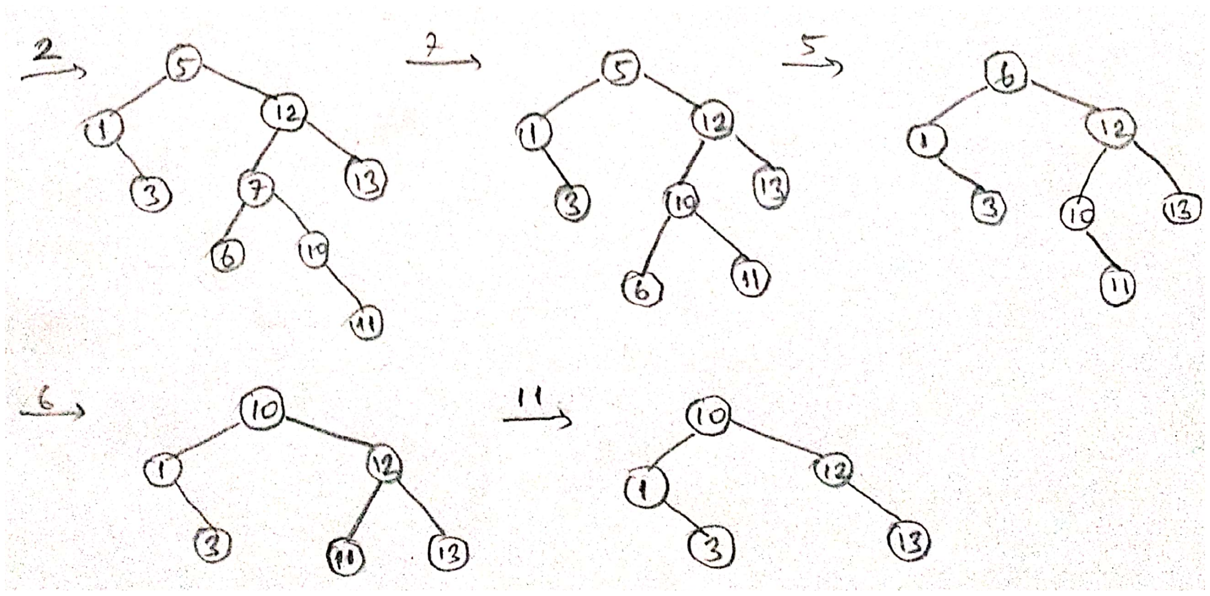
(b)

Preorder: 5, 1, 3, 2, 12, 7, 6, 10, 11, 13

Inorder: 1, 2, 3, 5, 6, 7, 10, 11, 12, 13

Postorder: 2, 3, 1, 6, 11, 10, 7, 13, 12, 5

(c) The order of the deletion: 2, 7, 5, 6, 11



Question 3:

- insertItem:

  In this method, the right place that the node will be added is found recursively. By comparing with the root of the sub-tree, the method chooses one of the sub-trees and calls itself for this sub-tree. If the key is greater than the root than it calls itself for right sub-tree, otherwise calls for left sub-tree. The worst-case is when the binary search tree is a chain of n nodes. Therefore, the worst-case complexity is $O(n)$. When the tree is more balanced, the height of the tree is $\log(n)$. Then, the average-case complexity is $O(\log(n))$. Therefore, the time complexity of the method is $O(\log(n))$. This method does not use any extra space. However, because of the recursion, the function calls use extra space.

- deleteItem:

  In this method, the node will be deleted is found recursively. By comparing with the root of the sub-tree, the method chooses one of the sub-trees and calls itself for this sub-tree. If the key is greater than the root than it calls itself for right sub-tree, otherwise calls for left sub-tree. After we found the node that will be deleted, there are three different cases. If the node is a leaf, then it is deleted directly. If the node has just one child, then we connect the child to the parent and delete the node. The last case is that the node has two children. In this case we find the left-most node of the right sub-tree and change it with the root. Then, connect the right sub-tree of the left-most node to the parent of the left-most node. The worst-case is when the binary search tree is a chain of n nodes and the deleted node at the end. Therefore, the worst-case complexity is $O(n)$. When the tree is more balanced, the height of the tree is $\log(n)$. The time complexity of finding the left-most node of the right sub-tree can be ignored because we get the summation of $O(\log(n)+\log(n)-1)$. Then, the average-case complexity is $O(\log(n))$. Therefore, the time complexity of the algorithm is $O(\log(n))$. This method does not use any extra space. However, because of the recursion, the function calls use extra space.

- retrieveItem:

  In this method, the node is found recursively. By comparing with the root of the sub-tree, the method chooses one of the sub-trees and calls itself for this sub-tree. If the key is greater than the root than it calls itself for right sub-tree, otherwise calls for left sub-tree. The worst-case is when the binary search tree is a chain of n nodes and the node that is desired at the end of the chain. Therefore, the worst-case complexity is $O(n)$. When the tree is more balanced, the height of the tree is $\log(n)$. Then, the average-case complexity is $O(\log(n))$ Therefore, we can say that the time complexity of the algorithm is $O(\log(n))$. This method does not use any extra space. However, because of the recursion, the function calls use extra space.

- inorderTraversal:

  In this method, the keys of the are saved to the array recursively. Firstly, this method calls itself for the left sub-tree. Then it saves the key at the root. Then it calls itself for the right sub-tree. However, due to the fact that we traverse the whole tree, the average-case and the worst-case the time complexities of the algorithm is $O(n)$. Therefore, the time complexity of the algorithm is $O(n)$. This method uses space for storing the values in the array, but does not use any extra space. However, because of the recursion, the function calls use extra space.

- containsSequence:

  In this method, whether the array is sub-array of the inorder traversal of the BST is found recursively. Firstly, it tries to find the first element of the array in the tree. The worst case is that the tree is a chain of n nodes and the first element of the array is at the end of the chain. At the average case, when the tree is more balanced, the height of the tree is $\log(n)$. Therefore, the average-case and the worst-case time complexity of this operation is $O(\log(n))$, $O(n)$ respectively. Then, it tries to find the next elements of the array by moving from the last element that is found before. If the length of the array is k , then both the average-case and the worst-case time complexity of this operation is $O(k)$. Therefore the average-case and the worst-case time complexities of the algorithm is $O(\log(n)+k)$, $O(n+k)$ respectively. Therefore, the time complexity of the algorithm is $O(\log(n)+k)$. This method does not use any extra space. However, because of the recursion, the function calls use extra space.

- countNodesDeeperThan:

  In this method, the number of the nodes that are deeper than the given level is found recursively. It adds one to the minimum of the heights of the left and right sub-trees if the node at deeper level. Due to the fact that we traverse the whole tree, the worst-case and the average-case time complexities of the method is $O(n)$. Therefore, the time complexity of the algorithm is $O(n)$. This method does not use any extra space. However, because of the recursion, the function calls use extra space.

- maxBalancedHeight:

  In this method, the maximum height of the balanced tree is found recursively. It adds one to the minimum of the heights of the left and right sub-trees. By adding another one to the sum it gets the maximum height of the tree. Due to the fact that we traverse the whole tree, the worst-case and the average-case time complexities of the method is $O(n)$. Therefore, the time complexity of the algorithm is $O(n)$. This method does not use any extra space. However, because of the recursion, the function calls use extra space.