Programming Assignment – 5

a) Discuss on Q1 & Q2:

Q1. Discuss the Best and Worst case time and space complexities of all the queries.

• insert(item) - Inserts item into the tree.

Insert operation insert elements in the BST tree and ensures BST property.

The best case scenario would be inserting only one element which results in below complexities.

Space complexity: O(1)
Time complexity : O(1)

In the worst case scenario, we would be inserting in the last leaf of the tree.

Space complexity: O(h) , as it has to save the nodes in root object
Time complexity : O(h) , as it has to traverse the height

• contains(item) - Returns True if item is in the tree, otherwise False.

Contains check the availability of the item in the tree, which is nothing but search operation. Search is similar to insert in terms of time complexity, whereas space complexity is constant.

In Best case scenario, the root item will be the expecting item

Space complexity: O(1)
Time complexity : O(1)

In worst case scenario, the leaf is the expecting item or item is not available

Space complexity: O(h) , as we are recursively calling the function and h is the height of the tree
Time complexity : O(h) , as it has to scan the tree to check the availability

• print() - Prints elements of the tree inorder. (python users should change the name of this function to something else)

Print operation prints the tree in inorder, i.e left, middle and right. Which will result in increasing order as it is a Binary search tree.

In Best case scenario, if the tree has only one element

Space complexity: O(1)
Time complexity : O(1)

In Worst case scenario, if the tree is a balanced binary tree

Space complexity: O(n) , where n is the number of elements, and recursion has to save objects by stack
Time complexity : O(n), as it has to print all the element (1*n)

• size() - Returns the number of nodes in the tree.

Size operation returns the size of the tree, which is similar to the print operation, in my program it has to traverse through all the element to find the size.

In Best case scenario, if the tree has only one element

Space complexity: O(1)
Time complexity : O(1)

In Worst case scenario, if the tree is a balanced binary tree

Space complexity: O(n) , where n is the number of elements, and recursion has to save objects by stack
Time complexity : O(n), as it has to traverse all the element (1*n)

• smallest() - Returns the smallest element in the tree.

Smallest function returns the smallest element in the tree, by the BST property, if we traverse the till the left most leaf, we will get the smallest element.

In Best case scenario, if the tree has only one element

Space complexity: O(1)
Time complexity : O(1)

In Worst case scenario, if all the elements are lower than the root element

Space complexity: O(n) , where n is the number of elements, and recursion has to save objects by stack
Time complexity : O(n), as it has to traverse all the element (1*n) till it reaches the left most leaf

• largest() - Returns the largest element in the tree.

Largest function returns the largest element in the tree, by the BST property, if we traverse the till the right most leaf, we will get the largest element.

In Best case scenario, if the tree has only one element

Space complexity: O(1)
Time complexity : O(1)

In Worst case scenario, if all the elements are larger than the root element

Space complexity: O(n) , where n is the number of elements, and recursion has to save objects by stack
Time complexity : O(n), as it has to traverse all the element (1*n) till it reaches the right most leaf

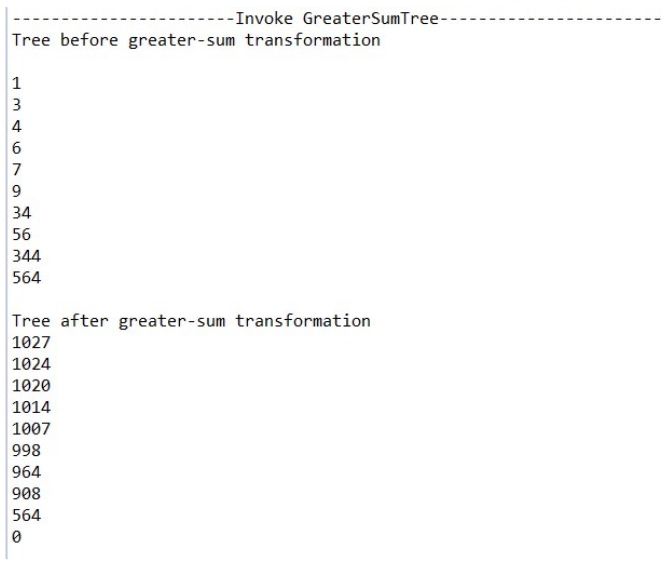
Q2. Analyze time and space complexities of greaterSumTree()

In the greaterSumtree problem, we are traversing from the largest element and saving it in a variable called "sum" and the sum is saved in a static variable "temp" and added with the key value of the current element. Likewise recursive call saves the sum of all the elements greater than current in the tree.

In Worst case scenario, it has to traverse, all the elements (n) and save n-1 elements in the memory with recursive call. Hence,

Space complexity: O(n) , where n is the number of elements, and recursion has to save objects by stack Time complexity : O(n), as it has to traverse all the element (1*n) till it reaches the left most leaf

<u>b)</u> <u>Output print of Q2:</u>



```
-----------------------Invoke GreaterSumTree-----------------------
Tree before greater-sum transformation

1
3
4
6
7
9
34
56
344
564

Tree after greater-sum transformation
1027
1024
1020
1014
1007
998
964
908
564
0
```

c) Source code: (attached separately)

```
class Node
        {
         int key;                              //Retained value
         Node right,left;              //linking the nodes


        //@SuppressWarnings("null")
public Node() {
                        //key=(Integer) null;
                        right=null;
                        left=null;
                       //System.out.println("Node "+ key + "left" +left + "right" +right);
                // TODO Auto-generated constructor stub
        }



}


public class BinarySearchTree {


        Node root;                                      //initialize a node
        static int temp=0;                      //Static variable to externally handle the sum in greater sum tree traversal


        public BinarySearchTree() {}


        //----------------------------------------------------//
        void insert(int item)
        {                                                                       //Wrapper for insert
root=insert_wrap(root,item);
        }


        public Node insert_wrap(Node root,int item){
        if (root==null)
                {
                        root=new Node();
                        root.key=item;
                        return root;
                }
                else if (root.key>item)
                {
                        root.left=insert_wrap(root.left, item);
                }
                else if (root.key<item)
                {
                        root.right=insert_wrap(root.right, item);
                }
                return root;
    }

        //----------------------------------------------------//
        void contains(int item)
        {                                       //Wrapper for contains fuction
                boolean exist=contains_wrap(root,item);
                System.out.println("Item->"+item+" available ? "+exist);
        }


                private boolean contains_wrap(Node root,int item){
                        if (root==null)
                                {
                                return false;
                                }
                else if         (root.key==item){
                        return true;
                }
                else if(root.key>item){
                        return contains_wrap(root.left, item);
                }
                else {
                        return contains_wrap(root.right, item);
                }
                }

        //----------------------------------------------------//
        void print (){
                print_wrap(root); //Wrapper for inorder print
        }

        public void print_wrap(Node root){                              // in order printing function of
BST             if (root!=null){                print_wrap(root.left);
System.out.println(root.key);
                print_wrap(root.right);
                }


        }

        //----------------------------------------------------//
        public void size(){
                int sz=size_wrap(root);
                System.out.print("Size "+sz+"\n");
        }
        public int size_wrap(Node root){
                int count=0;
                if (root!=null){
```

```
                                count=size_wrap(root.left)+size_wrap(root.right)+1;
                                }
                        return count;
                }

                //----------------------------------------------------//
                void smallest()
                { int small=smallest_wrap(root);
                        System.out.print("Smallest
                        "+small+"\n");


                }
                public int smallest_wrap(Node root){
                        if (root.left!=null){
                                return smallest_wrap(root.left);
                                }
                        else
                                return root.key;
                }

                //----------------------------------------------------//
                void largest()
                {
                        int large=largest_wrap(root);
        System.out.print("Largest "+large+"\n");
                }
                        public int largest_wrap(Node root){
                                if (root.right!=null)
                                        {
                                        return largest_wrap(root.right);
                                        }
                        else
                                        return root.key;
                }

                //----------------------------------------------------//


                public void greaterSumTree(){
                        System.out.print("Tree before greater-sum transformation"+"\n"+"\n");
                                print();
                                int sum=0;
                                greaterSumTree_wrap(root,sum);
                                System.out.print("\n"+"Tree after greater-sum transformation"+"\n");
                                print();
                                //return root;
                        }
                        public void greaterSumTree_wrap(Node root,int sum){
                                if (root==null){
                                        return;
                                }
                                greaterSumTree_wrap(root.right,sum);
                                sum=temp+root.key;
                                temp=sum;
                                root.key=sum-root.key;
                                greaterSumTree_wrap(root.left,sum);
                }
                //----------------------------------------------------//
public static void main(String[] args) {
                                // TODO Auto-generated method stub
int[] array = {56,34,6,1,4,7,9,564,344,3};
BinarySearchTree a=new BinarySearchTree();
                                System.out.print("-----------------------BST Insert-----------------------"+"\n");
                for (int i=0;i<array.length;i++){
                                                a.insert(array[i]);
                                }
                                System.out.print("-----------------------Invoke inorder print-----------------------"+"\n");
                                a.print();
                                System.out.print("-----------------------Invoke contains-----------------------"+"\n");
                                a.contains(4);
                                a.contains(100);
                                a.contains(564);
                                System.out.print("-----------------------Invoke size-----------------------"+"\n");
                                a.size();
                                System.out.print("-----------------------Invoke smallest-----------------------"+"\n");
                                a.smallest();
                                System.out.print("-----------------------Invoke largest-----------------------"+"\n");
                                a.largest();
                                System.out.print("-----------------------Invoke GreaterSumTree-----------------------"+"\n");
                                a.greaterSumTree();

                }

}
```