Question 1

Code to insert a value into binary search tree is

```
public static Node insert(Node tree, int value){
        if(tree == null){
                Node node = new Node(value, null, null);
                return node;
        }
        if(value < tree.value){
                tree.left = insert(tree.left, value);
        }
        else{
                tree.right = insert(tree.right, value);
        }
        return tree;
}
```

Case-I: Tree is Balanced

When Tree is balanced, it will be like a complete tree where each level other than last level is completely filled. So at each recursive call we reduce the size of tree by half and we either go to left subtree or right subtree but not both since it is a binary search tree. So, the recurrence relation will be

$T(n) = T(n / 2) + C$

We can apply Master's Theorem since it is of form $T(n) = a * T(n / b) + f(n)$ form.

$a = 1$

$b = 2$

$f(n) = C = Theta(1) = Theta(n^0 log^0 n)$

So, $d = 0$ and $k = 0$

Now, $log_b a = log_2 1 = 0$ which is equal to d. So, the given recurrence relation follows Case-III of Master's Theorem.

According to Case-III of Master's Theorem,

If $f(n) = Theta(n^d log^k n)$ where $d = log_b a$

Then, $T(n) = Theta(n^d log^{(k+1)} n)$

Therefore, $T(n) = Theta(n^d log^{(k+1)} n)$

$T(n) = Theta(n^0 log^{0+1} n)$

$T(n) = Theta(logn)$

According to Definition, $T(n) = Theta(g(n))$ if there exists C1, C2 and n0 where

$C1 * g(n) <= T(n) <= C2 * g(n)$ for all $n >= n0$

i.e; $C1 * logn <= T(n) <= C2 * logn$

If we consider only right part,

$T(n) <= C2 * logn$

According to Definition, $T(n) = O(g(n))$ if there exists C and n0 where

$T(n) <= C * g(n)$ for all $n >= n0$

Therefore, **$T(n) = O(logn)$**

Case-II: Tree is UnBalanced

Let's consider a skewed tree where each node has only 1 child from root to last node. So, at each recursive call we reduce size of tree by 1 and we either go to left subtree or right subtree but not both since it is a binary search tree. So, the recurrence relation will be

T(n) = T(n - 1) + C and T(0) = C1

Now, T(n - 1) = T(n - 2) + C [Since T(m) = T(m - 1) + C and m = n - 1 here]

So, T(n) = T(n - 1) + C + C

Now, T(n - 2) = T(n - 3) + C [Since T(m) = T(m - 1) + C and m = n - 2 here]

So, T(n) = T(n - 3) + C + C + C

Now, if we repeat for k times

T(n) = T(n - k) + C + C + … k times

We reach base case when n - k = 0 ===> k = n

At base case

T(n) = T(0) + C + C + …. n times

We know T(0) = C1

So, T(n) = C1 + C * n

We know that C1 + C * n <= C1 * n + C * n for all n >= 1

So, T(n) <= (C1 + C) * n for all n >= 1

According to Definition, T(n) = O(g(n)) if there exists C and n0 where

T(n) <= C * g(n) for all n >= n0

Therefore, **T(n) = O(n)**

Question 2

Code to search for a value in heap is

```
public static int search(int[] array, int index, int value){
    if(index >= array.length) return -1;
    if(array[index] == value) return index;

    int leftValue = search(array, 2 * index + 1, value);
    if(leftValue != -1) return leftValue;

    int rightValue = search(array, 2 * index + 2, value);
    return rightValue;
}
```

In Heap, the tree will be always a complete tree where each level other than last level is completely filled. For a max heap the only relation between a parent and its 2 children is that parent will be always greater than both the children. But, there is not guarantee that all values less than parent will be on left subtree and all values greater than parent will be on right subtree like we have in binary search tree. So, if we are searching for a value we have no choice but to search both the left and right subtrees for that value. At each recursive call we reduce the heap size by half. So, the recurrence relation will be

T(n) = 2 * T(n / 2) + C

We can apply Master's Theorem since it is of form T(n) = a * T(n / b) + f(n) form.

a = 2

b = 2

f(n) = C = Theta(1) = Theta(n^0)

So, d = 0

Now, $\log_b a = \log_2 2 = 1$ which is greater than d. So, the given recurrence relation follows Case-I of Master's Theorem.

According to Case-I of Master's Theorem,

If f(n) = Theta(n^d) where $d < \log_b a$

Then f(n) grows asymptotically slower than $\log_b a$

Therefore, T(n) = Theta($n^{\log_b a}$)

Therefore, T(n) = Theta($n^{\log_2 2}$)

T(n) = Theta(n)

According to Definition, T(n) = Theta(g(n)) if there exists C1, C2 and n0 where

C1 * g(n) <= T(n) <= C2 * g(n) for all n >= n0

i.e; C1 * n <= T(n) <= C2 * n

If we consider only right part,

T(n) <= C2 * n

According to Definition, T(n) = O(g(n)) if there exists C and n0 where

T(n) <= C * g(n) for all n >= n0

Therefore, **T(n) = O(n)**