1 Question 1

1.1 a.

1.2 b.

1.3 c.

2 Question 2

2.1 a.

The successor of x is the smallest element in the right subtree of x in a BST. Let r be x's successor. If r has no right child, it will have 0 children because if it had a left child, it would not be the successor of $x/$. If r has a right child, that child will be r's only child for the reason just stated. Therefore, x's successor can have either 0 or 1 children.

2.2 b.

The predecessor of x is the largest element in the left subtree of x. Let l be x's predecessor. Then, if l has no right child, it will have 0 children because a right child would imply that l is not x's successor. If l has a left child, it will still be the predecessor to x because any child in the left subtree of l will be strictly less than l. Therefore, l can have 0 or 1 children.

3 Question 3

3.1 a.

funA:
Let x be an arbitrary node in a BST. In every instance of comparing x's children to x, each invocation ensures that no left child be greater than $nodeVal(x)$ and no right child can be less than $nodeVal(x)$. This implies that funA is a correct algorithm.

funB:
I will give an example to show that the algorithm is incorrect. Suppose we take an arbitrary node x in the BST. Let l be the left child of of x and l_r be the right child of l. Given this, the algorithm will not check if $nodeVal(l_r) < nodeVal(x)$ because it is only comparing children nodes to their respective parent nodes. So, if $nodeVal(l) < nodeVal(x)$ but $nodeVal(l_r) > nodeVal(x)$, funB will still return true.

3.2 b.

funA:
At each check of a node we are using $tree-get-max()$ and $tree-get-min$ functions which have a runtime of $O(h)$. Additionally, since we are calling both functions for each node, we have a runtime of $2h * n$. Since h is a constant, we drop it to get a complexity of $\theta(hn)$. In the best case, where the BST properties do not hold, funA will have a complexity of $\Omega(h)$ because it will still call both $get - max/min$ functions in order to determine the tree is not a BST.

funB:
For each node in the BST, we are compare it to its parent, left child, and right child which, implies that each node is compared to 3 times. Since we compare every node in the tree, with a size of n, we have a runtime of $3n$. Since 3 is a constant we can drop it, which yields a complexity of $\theta(n)$.

4 Question 4

4.1 a.

From the book, we have that the expected upper bound for the number of probes of an unsuccessful search is $\frac{1}{1-\alpha}$. Using $\alpha = \frac{6}{7}$ we have $\frac{1}{1-\frac{6}{7}} = 7$ probes.

4.2 b.

Also from the book, we have that the expected upper bound for the number of probes of a successful search is $\frac{1}{\alpha}ln(\frac{1}{1-\alpha})$. Using $\alpha = \frac{6}{7}$ we have $\frac{1}{\frac{6}{7}}ln(\frac{1}{1-\frac{6}{7}}) = \frac{7}{6}ln(7) = 2.27$ probes.

5 Question 5

From (4), we know the equations of the expected number of probes. For this problem, we can set the equation for the unsuccessful search equal to 3.5 times the equation for the successful search. This gives $\frac{1}{1-f} = \frac{3.5}{f}ln(\frac{1}{1-f})$. We can then graph these two equations and see where they intersect. When graphed, both lines intersect when $x = 0.882$. Thus, a load factor f of 0.882 will ensure that the expected number of probes in an unsuccessful search is 3.5 times the expected number of probes in a successful search.