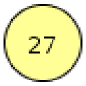
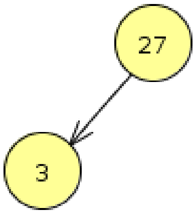
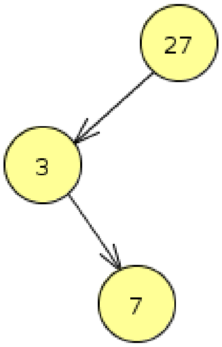
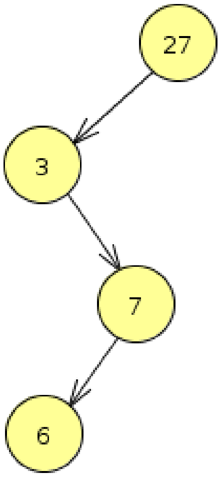
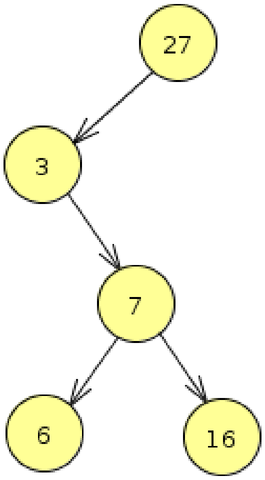
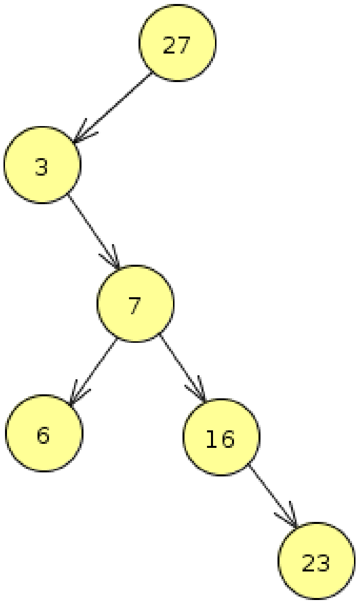
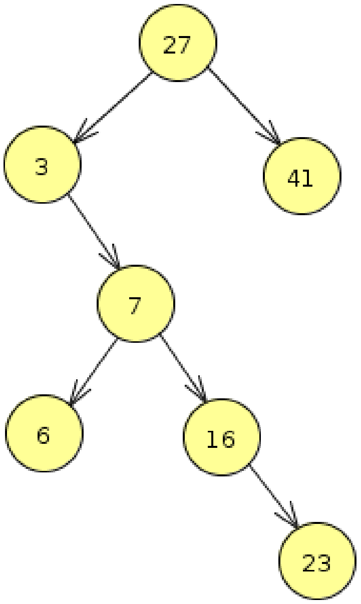
Name: Sabbir Ahmed
Section: 02
HW #: 3
Version: B
Username: sabbir1

CMSC 341 Homework 3 – Version B
Introduction to Trees

1. **Insertion:**
   Insert these numbers into a Binary Search Tree: 27, 3, 7, 6, 16, 23, 41
   Draw what the tree would look like after each addition and explain why it looks this way.

Inserted	What the tree looks like	Explain what happened
27		The first node is assigned as the root of the tree
3		The function would check if 3 is greater or less than the root, 27, and assign it as a left child if less than or right if greater than. Since 3 < 27, 3 is a left child
7		The function would check if 7 is greater or less than the root, and assign it as a left or right child respectively. Since 7 < 27, it would be assigned as a left child. But 3 is already there, so the function checks if 7 is less than or greater than 3 to assign it as its left or right child. Since 7 > 3, 7 is a right child.

6		The function would check if 6 is greater or less than the root, and assign it as a left or right child respectively. Since 6 < 27, it would be assigned as a left child. But 3 is already there, so the function would recursively check 3 and 7 if they are less than or greater than 6 to assign as their left or right child. Since 6 > 3, 6 is assigned as a right child. But 7 is already there, so it is assigned as 7’s left child since 6 < 7
16		The function would recursively check if 16 is less than or greater than the nodes similarly as 6, but since 16 > 7, it is assigned as 7’s right child

23		The function would recursively check if 23 is less than or greater than the nodes similarly as 16, but since 23 > 7, it is assigned as 7's right child. But 16 is already the right child of 7, so 23 is assigned as the right child of 16 since 23 > 16
41		The function would check if 41 is less than or greater than the root, and since 41 > 27, it is assigned as the first right child of the root

2. **Searching:**
Using the same tree above, show each step the algorithm takes as it searches the tree in order to find the value **6**

A typical tree searching algorithm would have processes similar to:

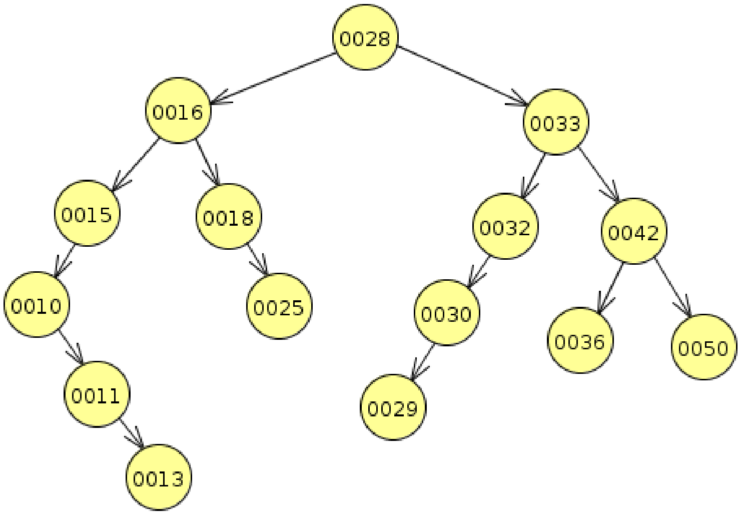
```
def search( left-subtree or right-subtree)

        var searchVal = 6;
        var nodeVal = key or value of the node;

        if nodeVal == NULL:
           return NULL

        else if nodeVal == searchVal:
           return nodeVal

        else if nodeVal < searchVal:
           search (right)

        else:
           search (left)
```

Using search(tree), the function would first check if the root is 6. When it’s done, it checks if 27 < 6, and returns false, so search() is called to focus on the left subtree of the root.

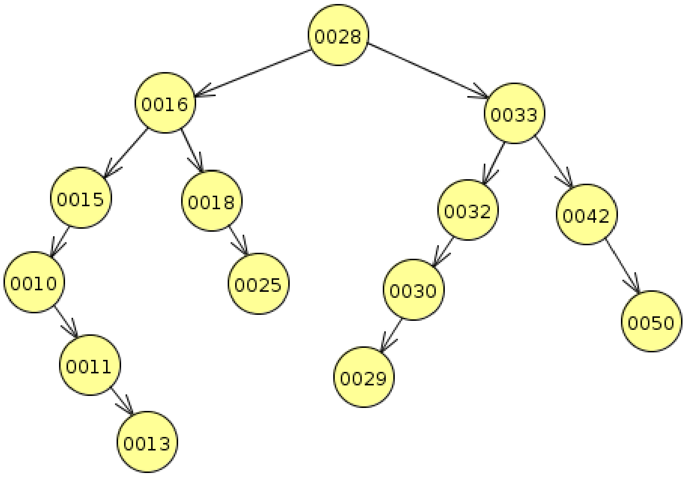
The function now checks if the root of the left subtree, 3, is 6. It returns false, and then checks if 3 < 6. Since it returns true, search() now traverses through the right subtree.
The function now checks if the root of the right subtree, 7, is 6. It returns false, and then checks if 7 < 6. Since it returns false, search() now traverses through the left subtree.

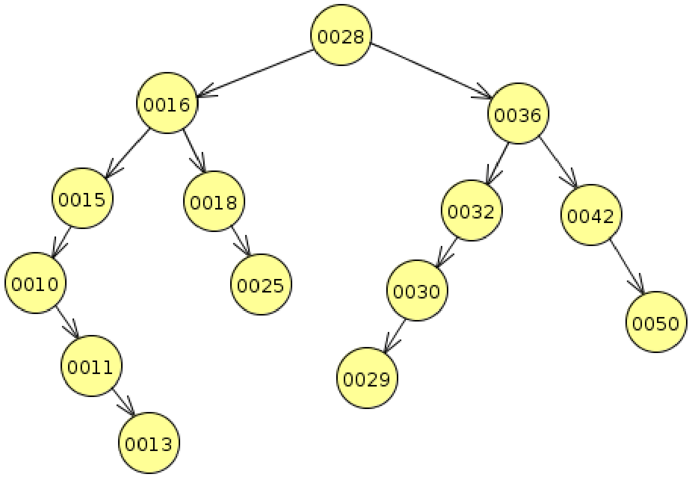
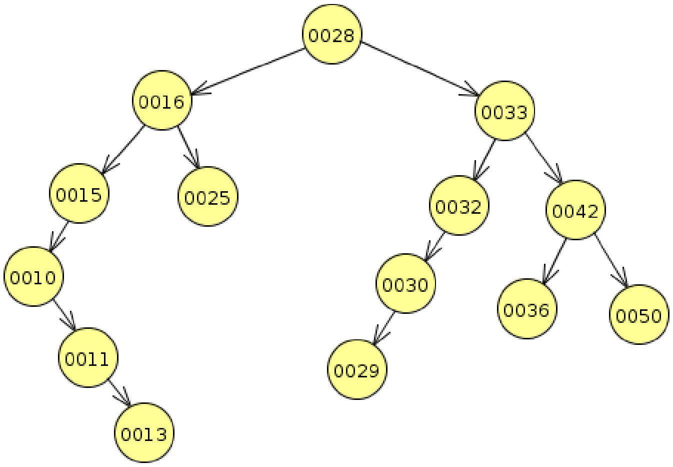
The function now checks if the root of the left subtree, 6, is 6. It returns true, and search returns the location of 6.

3. **Deletion:**
You will be given a node # to delete. Show what the tree will look like AFTER the complete deletion and why the tree looks like this as a result.



Original tree given

Node # to delete	**Draw final tree after deletion**	**Why does it look this way?**
0036	 0028 0016 0033 0015 0018 0032 0042 0010 0025 0030 0050 0011 0029 0013	The delete algorithm would first search for the value, and then once it finds it, attempts to remove it while keeping the structure of the BST. Since 0036 does not have any children, it can be safely removed without having to worry about any dependencies

0033	 0028 0016 0036 0015 0018 0032 0042 0010 0025 0030 0050 0011 0029 0013	The delete algorithm searches for the value first, and then attempts to remove it while keeping the structure of the BST. Since 0033 if the topmost right child of the root, it has several children nodes to worry about. After 0033 is removed, the algorithm looks for its successor, 0042. Since that node also has 2 children, it looks for its smaller child that can now replace 0033's position to maintain the directory structure
0018	 0028 0016 0033 0015 0025 0032 0042 0010 0030 0036 0050 0011 0029 0013	The delete algorithm searches for the value first, and then attempts to remove it while keeping the structure of the BST. Since 0018 only has one successor, it is simply replaced by it to maintain the BST structure