1. Master Method Practice (6 pts)

Use the Master Method to give tight θ bounds for the following recurrence relations. Show *a*, *b*, and *f(n)*. Then explain why it fits one of the cases, choosing ϵ where applicable. Write and *simplify* the final θ result.

(a) $T(n) = 3T(\frac{n}{9}) + n$

Let $a = 3, b = 9, f(n) = n^3$
By comparing n^3 to $\theta(n_9^{log3}) = \theta(n^1/2)$ and letting $= 1/2$, you get:
$f(n) = n^3 = \Omega(n_9^{log3}) = \Omega(n)$
By using the Master Method, this follows case 3 by checking for regularity, given the conditions are satisfied:
$T(n) = \theta(n^3)$

(b) $T(n) = 7T(\frac{n}{3}) + n$

Let $a = 7, b = 3, f(n) = n$
By using ϵ and letting $\epsilon = log_3 7 - 1$, we can input this into $f(n)$ as follows:
$f(n) = n = O(n_3^{log7} - \epsilon) = O(n^1) = O(n)$
By using the Master Method, this follows Case 1:
$T(n) = \theta(n_3^{log7})$

(c) $T(n) = 2T(\frac{n}{4}) + \sqrt{n}$

Let $a = 2, b = 4, f(n) = \sqrt{n}$
By comparing $n^1/2 = \sqrt{n}$ to $\theta(n_4^{log2})$, you get $\theta(n^1/2)$
By using the Master Method, this follows Case 2:
$T(n) = \theta(\sqrt{n} lgn)$

2. Solving a Recurrence (10 pts)

Solve the following recurrence by analyzing the structure of the recursion tree to arrive at a "guess" and using substitution to prove it. (Do not try to use the Master Method.) Justify all steps to show that you understand what you are doing.

$$T(n) = T(\sqrt{n}) + c \quad ifn > 2$$
$$T(n) = c \quad ifn \leqslant 2$$

Recursion Tree

$T(n)$
|

$$\begin{array}{c} T(n^{\frac{1}{2}}) \\ | \\ T(n^{\frac{1}{4}}) \\ ... \\ T(2) \end{array}$$

By substitution method, you can deduce the following:

$$T(n) = T\sqrt{n} + c$$

From inductive hypothesis you have the equations as followed:

Let a guess be $c(log(log(n)))$, where c is arbitrarily a constant. Thus you have:

$$T(n) \leqslant c * loglog\sqrt{n}$$
$$T(n) \leqslant c * loglogn^{\frac{1}{2}}$$
$$T(n) = c * \frac{1}{2}loglogn$$
$$T(n) = \frac{1}{2} * cloglogn$$
$$T(n) = c * loglogn$$
$$\therefore T(n) = \theta(loglogn)$$

3. Binary Search Tree Proof (11 pts

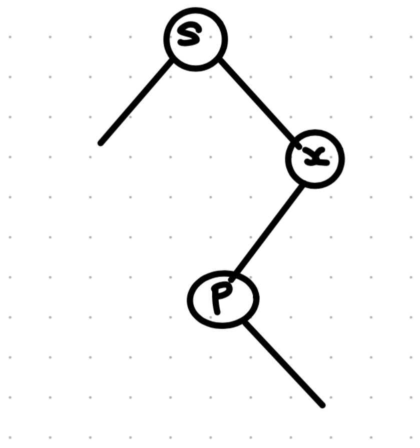
The procedure for deleting a node in a binary search tree relies on a fact that was given without proof in the notes:

Lemma: If a node X in a binary search tree has two children, then its successor S has no left child and its predecessor P has no right child.

In this exercise you will prove this lemma. Although the proof can be generalized to duplicate keys, for simplicity assume no duplicate keys. The proofs are symmetric, so we start by proving for the successor. We rule out where the successor cannot be to narrow down to where it must be. Drawing pictures may help.

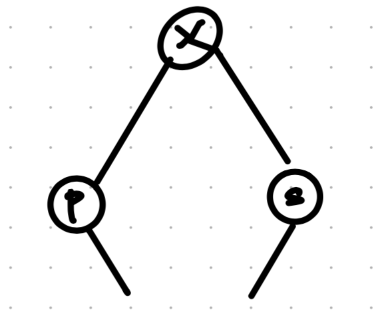
(a) Prove by contradiction that the successor S cannot be an ancestor or cousin of X, so S must be in a subtree rooted at X.

It is given that the success of X has no given left child and that the predecessor of X has no given right child. Therefore you can deduce that S.key is the smallest key, and P.key is the largest key. Knowing these facts allows you to deduce that there is a violation in the Binary Search Tree property. X can therefore neither be an ancestor or a cousin.



(b) Identify and prove the subtree of X that successor S must be in.

Given S is a successor of X, and S > X, it is assumed that S is within the left subtree. This means that the above answer violates the properties of the Binary Search Trees, and would need to be arranged again so that S would be the right subtree rooted at X. The basic reconstruction of this would require that S be in the left subtree, implying that X > S.



(c) Show by contradiction that successor S cannot have a left child.

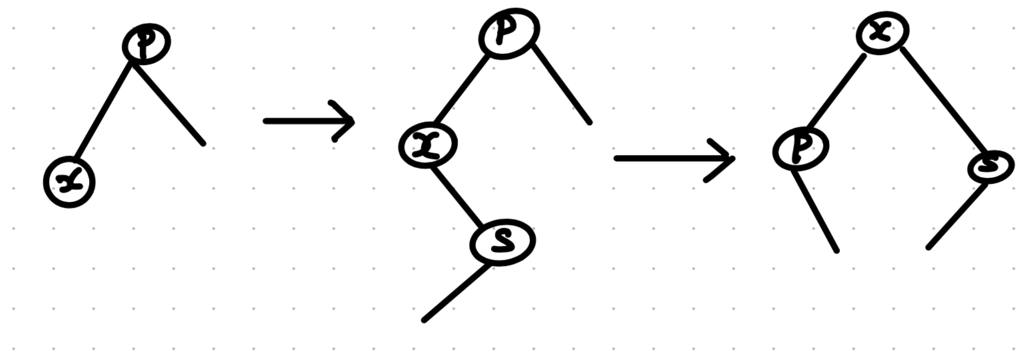
The successor is otherwise known as having no left child, with S.Key being the smallest > X.key. S would otherwise not be the smallest key in the node X's right subtree if there was not a left child for node S. Therefore, for S to be the successor of X, it is proven that S ultimately cannot have a left child.

(d) Indicate how this proof would be changed for the predecessor.

It is given that the successor of X has no given left child and that the predecessor of X has no given right child in part (a).

By adding a successor that has no left child to the node, you know that the P.key would be the largest key that is < X.key, and the smallest key > X.key. Since this is a violation of the binary search tree property, there would need to be rearrangements in this data structure.

A predecessor is otherwise defined as having no right child. Since P.Key is the largest key that is < X.key, if it is true that there would be a right child for P, then P would not be the largest node in X's left subtree. In order for P to be a predecessor of its parent X, P cannot have a right child, because anything in the right subtree of P would mean that it is larger than P itself.



4. Deletion in Binary Search Trees (5 pts)

Consider Tree-Delete (CLRS page 298), where x.left, x.right, and x.p access the left and right children and the parent of a node x, respectively.

Algorithm 1 TREE-DELETE (T,z)

```
if z.left == NIL
    TRANSPLANT (T,z,z.right)
elseif z.right == NIL
    TRANSPLANT(T,z,z.left)
else y = TREE-MINIMUM(z.right) //successor
    if y.p != z
        TRANSPLANT (T,y,y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT (T,z,y)
    y.left = z.left
    y.left.p = y
```

(a) How does this code rely on the lemma you just proved in problem 3? Be specific, referring to line numbers. Be careful that you are using the actual lemma above, and not a similar fact proven elsewhere.

At the end of the code, z.left is replacing y.right with its sufficient subtrees. Given the above lemma that was proven in part 3 of this assignment, it is understood that no subtree is overwritten in the node y.left of this data structure.

(b) When node z has two children, we arbitrarily decide to replace it with its successor. We could just as well replace it with its predecessor. (Some have argued that if we choose randomly between the two options we will get more balanced trees.) Rewrite Tree-Delete to use the predecessor rather than the successor. Modify this code just as you need to and underline or boldface the changed portions.

Algorithm 2 TREE-DELETE (T,z)

```
if z.left == NIL
    TRANSPLANT (T,z,z.right)
elseif z.right == NIL
    TRANSPLANT(T,z,z.left)
else y = TREE-MINIMUM(z.left) //successor
    if y.p != z
        TRANSPLANT (T,y,y.left)
        y.left = z.left
        y.left.p = y
    TRANSPLANT (T,z,y)
    y.right = z.right
    y.right.p = y
```

5. Constructing Balanced Binary Search Trees (8 pts)

Suppose you have some data keys sorted in an array A and you want to construct a balanced binary search tree from them. Assume a tree node representation TreeNode that includes instance variables key, left, and right. (No p needed in this problem.)
(a) Write pseudocode (or Java if you wish) for an algorithm that constructs the tree and returns the root node. (We won't worry about making the enclosing BinaryTree class instance.) You will need to use methods for making a new TreeNode, and for setting its left and right children.

Algorithm 3 BALBST (int arr[], int start, int end)

```
if (start >end)
    return null
else
int mid = Math.floor(start + end)/2
Node node = new Node(arr[mid])
node.left = BALBST(arr, start, mid-1)
node.right = BALBST(arr, mid+1, end)
return new TreeNode(A[mid], node.left, node.right)
```

Hints: Think about how BinarySearch works on the array. Which item does it access first in any given subarray it is called with? How can we set up the tree so that a search sequence for a given key examines the same keys as it does in the array?

(b) What is the θ time cost to construct the tree, assuming that the array has already been sorted? Justify your answer.

Assuming that the array has already been sorted, the runtime cost to construct this given tree is $\theta(n)$. The basic functions of the tree denotes that it is a constant time to either add or remove certain elements within an array that has n elements. Although it is true that there is a constant time to delete the elements, the n number of elements needed to construct the tree dominates, leaving the runtime to be $\theta(n)$.

(c) Compare the expected runtime of BinarySearch on the array to the expected runtime of BST TreeSearch in the tree you just constructed. Have we saved time?
The expected runtime of BinarySearch given in our lectures notes is $O(logn)$. In the tree that was constructed, we have not saved that much time.