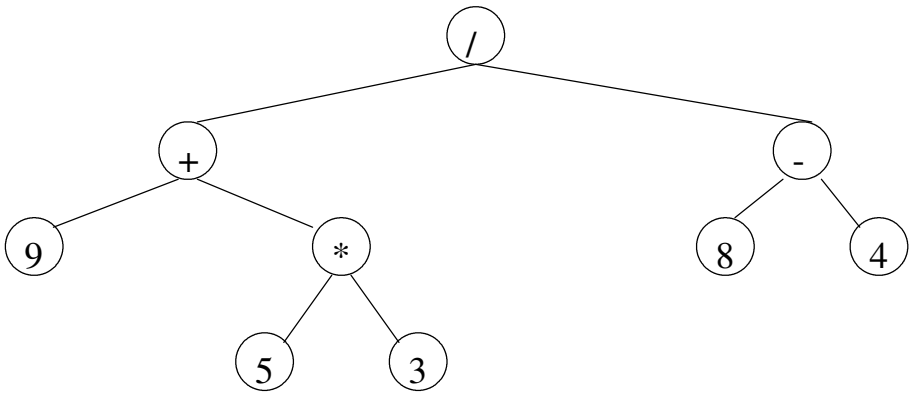
Data Structures (810:052) Name:_____

1. Consider the parse tree for (9 + (5 * 3)) / (8 - 4):



Indentify the following items in the above tree:

a. *node* containing "*"

b. *edge* from node containing "-" to node containing "8"

c. *root* node

d. *children* of the node containing "+"

e. *parent* of the node containing "3"

f. *siblings* of the node containing "*"

g. *leaf* nodes of the tree

h. *subtree* who's root is node contains "+"

i. *path* from node containing "+" to node containing "5"

j. *branch* from root node to "3"

k. mark the *levels* of the tree (level is the number of edges on the path from the root)

l. What is the *height* (max. level) of the tree?

2. A *traversal* iterates through all the nodes in the tree. The four common binary tree traversals are:

Preorder	**Inorder**	**Postorder**	**Level order**
visit the root node Preorder traverse left subtree Preorder traverse right subtree	Inorder traverse left subtree visit the root node Inorder traverse right subtree	Postorder traverse left subtree Postorder traverse right subtree visit the root node	visit nodes at each level from left-to-right starting with level 0

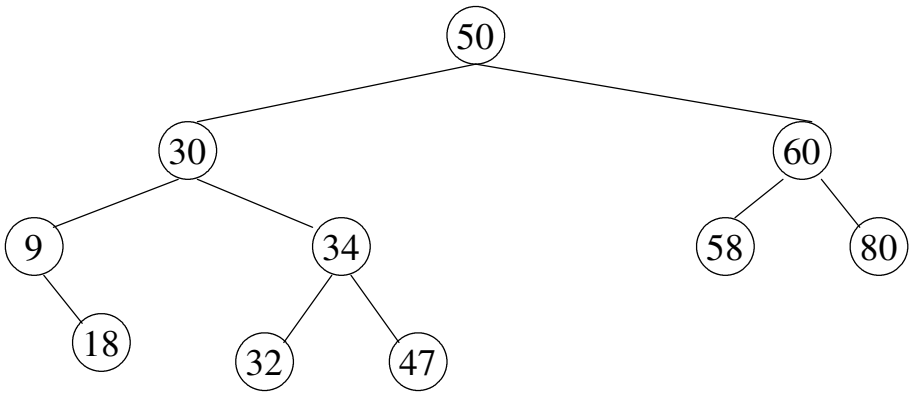
a. What is the preorder traversal of the above expression tree?

b. What is the inorder traversal of the above expression tree?

c. What is the postorder traversal of the above expression tree?

d. What is the level-order traversal of the above expression tree?

3. Consider the Binary Search Tree (BST). For each node, the values in the left-subtree are < the node and the values in the right-subtree are all > the node.



a. What would be the result of an inorder traversal?

b. Starting at the root, how would you find the node containing "32"?

c. Starting at the root, when would you discover that "70" is not in the BST?

d. Starting at the root, where would be the "easiest" place to add "70"?

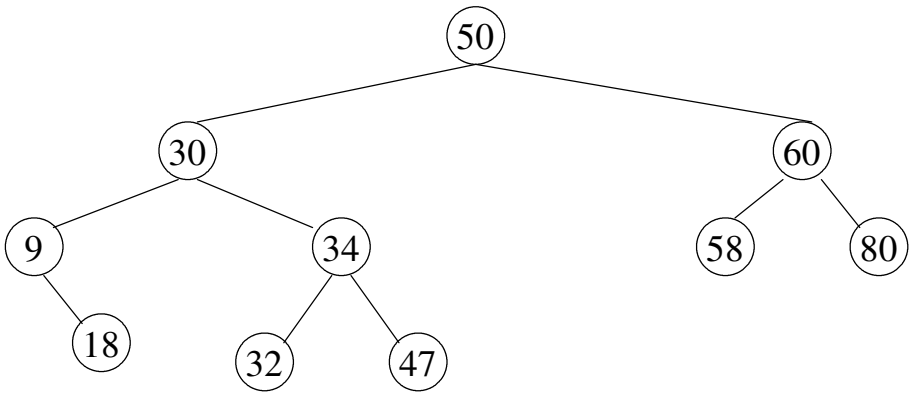
4. a) If a BST contains n nodes and we start searching at the root, what would be the worst-case theta Θ( ) notation for a successful search? (Draw the shape of the BST leading to the worst-case search)

b) We could store a BST in an array like we did for a binary heap, what would be the worst-case storage needed for a BST with n nodes?

5. a) If a BST contains n nodes, draw the shape of the BST leading to best, successful search in the worst case.

b) What is the worst-case theta Θ( ) notation for a successful search in this "best" shape BST?

6. Consider the Binary Search Tree (BST):



a. What would be the result of deleting 58 from the BST?

b. What would be the result of deleting 9 from the BST?

c. What would be the result of deleting 50 from the BST? (Hint: One technique when programming is to convert a hard problem into a simpler problem. Deleting a BST node that contains two children is a hard problem. Since we know how to delete a BST node with none or one child, how could we convert"deleting a node with two children" problem into a simpler problem?)

BINARY TREE METHOD	WHAT IT DOES
`T = BinaryTree(item)`	Creates a new binary tree with `item` as the root and empty left and right subtrees. This is essentially a leaf node.
`T.__str__()`	Same as `str(T)`. Returns a string representation of the tree that shows its structure.
`T.isEmpty()`	Returns `True` if `T` is empty, or `False` otherwise.
`T.preorder(aList)`	Performs a preorder traversal of `T`. *Postcondition*: the items visited are added to `aList`.
`T.inorder(aList)`	Performs an inorder traversal of `T`. *Postcondition*: the items visited are added to `aList`.
`T.postorder(aList)`	Performs a postorder traversal of `T`. *Postcondition*: the items visited are added to `aList`.

continued

`T.levelorder(aList)`	Performs a level order traversal of `T`. *Postcondition*: the items visited are added to `aList`.
`T.getRoot()`	Returns the item at the root. *Precondition*: `T` is not an empty tree.
`T.getLeft()`	Returns the left subtree. *Precondition*: `T` is not an empty tree.
`T.getRight()`	Returns the right subtree. *Precondition*: `T` is not an empty tree.
`T.setRoot(item)`	Sets the root to `item`. *Precondition*: `T` is not an empty tree.
`T.setLeft(tree)`	Sets the left subtree to `tree`. *Precondition*: `T` is not an empty tree.
`T.setRight(tree)`	Sets the right subtree to `tree`. *Precondition*: `T` is not an empty tree.
`T.removeLeft()`	Removes and returns the left subtree. *Precondition*: `T` is not an empty tree. *Postcondition*: the left subtree is empty.
`T.removeRight()`	Removes and returns the right subtree. *Precondition*: `T` is not an empty tree. *Postcondition*: the left subtree is empty.

[TABLE 18.3] The operations on a binary tree ADT