In the AVL trees below, keys are shown but values are not. Following our standard algorithm, insert the key `5` into the following AVL tree and restore balance.

Answer: [After insertion, node `2` is out of balance in the RR sense; a left rotation at this node restores the entire tree to balance.]

Following our standard algorithm, delete the key `8` from this AVL tree and restore balance.

Answer: [We put the contents of node `9` into node `8` and then tell the right subtree to delete node `9`. After it deletes the node, node `10` is out of balance in the RL sense; this is fixed by a right rotation at node `12` followed by a left rotation at node `10`. Then node `9` is out of balance in the LR sense; this is fixed by a left rotation at node `3` followed by a right rotation at node `9`. The resulting tree, rooted at node `6`, is in AVL balance.]

Write a function `parse` that parses postfix algebraic expressions such as `3 x + 15 *`. You may assume that there are no parentheses in the input, that the user makes no syntax errors, and that all operators are binary. You may assume the existence of a function `isOperator`, that for any token returns `True` or `False` indicating whether the token is an operator.

Answer:

```
from stack import Stack
from binarytreenode import BinaryTreeNode
def parse(tokens):
    myStack = Stack()
    for token in tokens:
        if isOperator(token):
            node = BinaryTreeNode(token)
            node.setRightChild(myStack.pop())
            node.setLeftChild(myStack.pop())
            myStack.push(node)
        else:
            myStack.push(BinaryTreeNode(token))
    return myStack.pop()
```

Describe the running time of `__setitem__` in `Dictionary` implemented atop a binary search tree (with no balancing). You may want to comment on several possible cases.

Answer: Let n be the number of key-value pairs in the dictionary. The time cost of `__setitem__` arises from (A) searching through a binary search tree of height to find the hashed key and (B) searching through that hashed key's chain to find the key. If the hash function is terrible, then all keys will hash to the same hash value, so the tree will have a single node with

a chain of length n. Because this chain must be searched linearly, `__setitem__` will be $\mathcal{O}(n)$. If the hash function is perfect, then the tree will have n nodes. However, the tree may be so unbalanced that its height h is $n-1$; then `__setitem__`, which is always $\mathcal{O}(h)$, will be $\mathcal{O}(n)$. On the other hand, in the case of a perfect hash function and a tree that happens to be balanced, there are n nodes in the tree and height is logarithmically related to the number of nodes, so that `__setitem__` is $\mathcal{O}(h) = \mathcal{O}(\log n)$.

Describe the running time of `__setitem__` in `Dictionary` implemented atop a AVL-balanced binary search tree. You may want to comment on several possible cases.

Answer: Similar remarks apply. If the hash function is terrible, then `__setitem__` will be $\mathcal{O}(n)$ due to a chain of length n. If the hash function is perfect, then the number of nodes in the tree will be n, and due to the AVL balancing the height h is logarithmically related to the number of nodes, so that `__setitem__` is $\mathcal{O}(h) = \mathcal{O}(\log n)$.

This question has two ingredients. First, our textbook's parsing algorithm stores operands as values (e.g. the integer `3`) rather than as tokens (e.g. the string `"3"`) in the parse tree. Second, the epilogue of the Interpretation assignment talks about user-defined functions stored in the environment as parse trees. Now suppose that we're trying to add user-defined functions to our interpreter, and we have a user who has volunteered to help us test the interpreter as we work on it. Somewhere in the middle of using our interpreter, the user issues the following command.

```
>> myFunction = (x return (x + y))
```

Should `x` be stored in the parse tree as a token or as a value? Should `y` be stored in the parse tree as a token or as a value? Discuss.

Answer: In the syntax being used here, `x` is the formal parameter (the input) to the function. We want to bind `x` to a new value every time the function is called. Therefore `x` must be stored as a token, rather than as a value.

On the other hand, `y` is not a parameter of the function; if it is ever to have a value, that value must come from an environment. If we store `y` as a value in the parse tree, then we are pulling the value of `y` out of the environment at the time of the definition of `myFunction`. (If `y` is undefined at the time of definition, then an error occurs.) If we store `y` as a token, then the value of `y` can change, and thus the behavior of the function can change. For example, the value of `y` can be pulled from the calling environment on each function call.

So the value-vs.-token choice for `y` depends on which semantics are desired. This is a key aspect of programming language design. For example, in the Mathematica language both semantics described above are available. If a function is defined using the `=` syntax, then `y` is stored as a value; if the function is defined using the `:=` syntax, then `y` is stored as a token and evaluated from the environment on each function call.