12.2-1

c, e

12.2-3

```
function treePredecessor(node)
    if node.left
        return tree-maximum(node.left)
    parent = node.parent
    while parent ≠ NIL and x == parent.left
        x = parent
        parent = parent.p
    return parent
```

12.3-3

Best Case: Elements from the set are selected in an order such that for every insertion, the tree remains balanced. Height (h) will always be $\log n$, thus tree-insert will always take $O(\log n)$. Building the tree calls tree-insert n times, so this operation takes $O(n \log n)$. Traversing the tree always takes n time, so the algorithm is $O(n \log n) + n = O(n \log n)$.

Worst Case: Elements from the set are already sorted such that the tree essentially becomes a list of elements on one side. Here $h = n$, so tree-insert takes $O(h) = O(n)$ time. With n calls to tree-sort, the algorithm will take $n\,O(n) = O(n^2)$ time to build the tree. Walking the tree will always take n time, so the overall runtime is $O(n^2) + n = O(n^2)$.