Lecture 16 Exercise Solutions

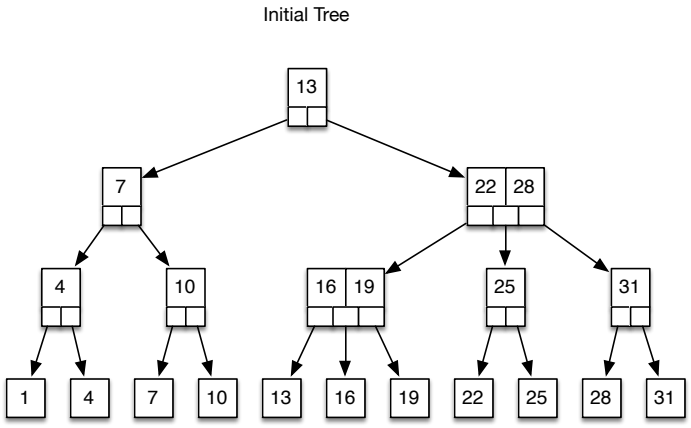
Mark Eramian

Exercise 1

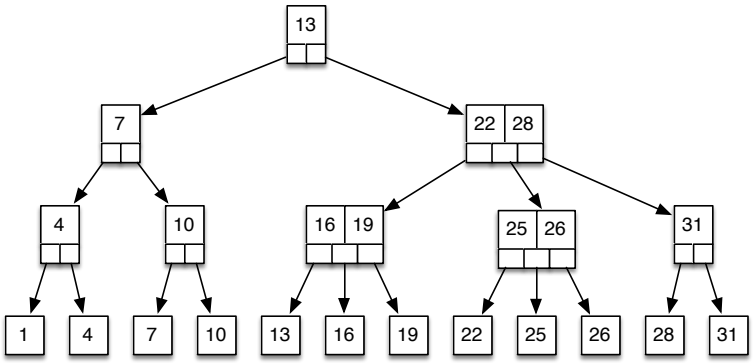
- When searching for 7, we must visit the nodes: 13 ⇒ 7 ⇒ 10 ⇒ 7.
- When searching for 19, we must visit: 13 ⇒ 22/28 ⇒ 16/19 ⇒ 19.
- When searching for 28, we must visit: 13 ⇒ 22/28 ⇒ 31 ⇒ 28.
- When searching for 1, we must visit: 13 ⇒ 7 ⇒ 4 ⇒ 1.
- When searching for 26, we must visit: 13 ⇒ 22/28 ⇒ 25 ⇒ 25.

Exercise 2

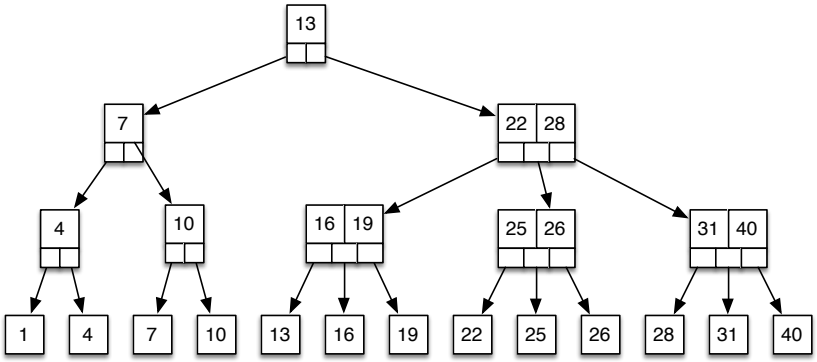
Here is the solution starting from the tree given on the slide:



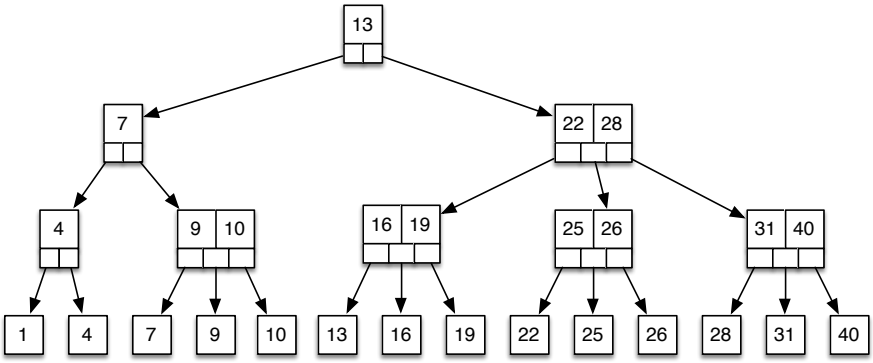
Insert 26



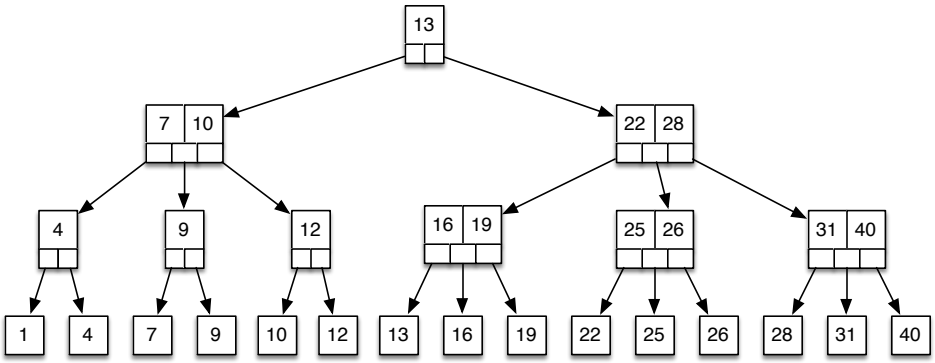
Insert 40



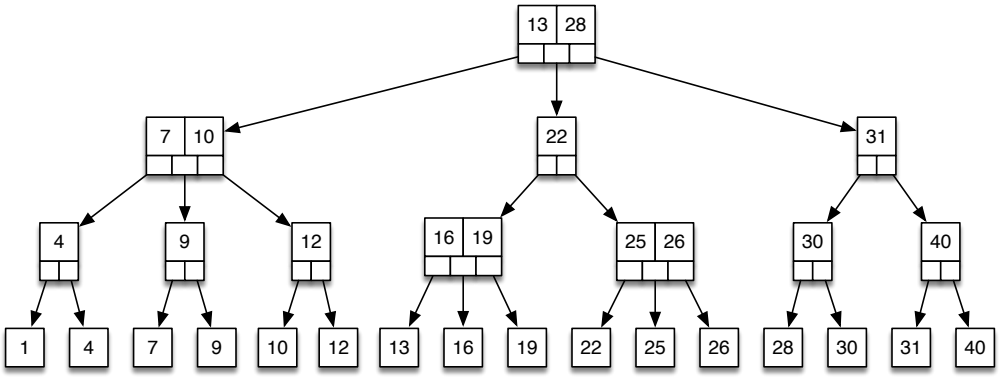
Insert 9



Insert 12



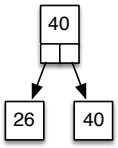
Insert 30



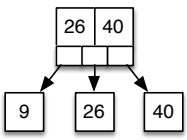
Here is the solution to exercise 2 starting from an empty tree:

Insert 26

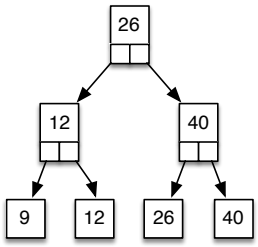
Insert 40



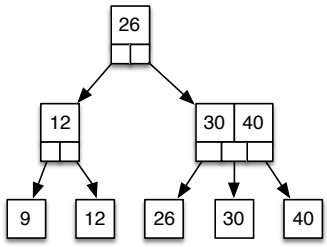
Insert 9



Insert 12



Insert 30



Exercise 3

We use an abbreviated statement-counting approach.

The time complexity for the base case is $O(1)$. There are no loops, just a sequence of statements.

The time expended before each recursive call is $O(1)$ because the base case can be checked in $O(1)$ time, and there are no loops prior to the recursive call in the recursive case, just an `if-elseif-else` staement with no other function calls.

The amount of time expended after the recursive call is again, just another series of conditional statements with no loops. However, there is a call to the `split()` function. One call to `split()` just adjusts some pointers locally, so it, also, would be just a sequence of statements in a some kind of nested if statement. Thus the amount of time expended after the recursive call is also $O(1)$. That means each recursive call does $O(1)$ work. All that is left to determine is the number of recursive calls.

The recursive insert algorithm is called once for every level of the tree except the bottom level (because we stop recursing one level above the leaves). That means that the number of recursive calls is equal to one less than h, the height of the tree. So what is the height of a 2-3 tree with m nodes? In the worst case, each node has 3 children, so each level has no more than 3 times the number of nodes as the one above it. Thus, the height of a tree with m nodes is at most $O(\log_3 m)$. Recall that

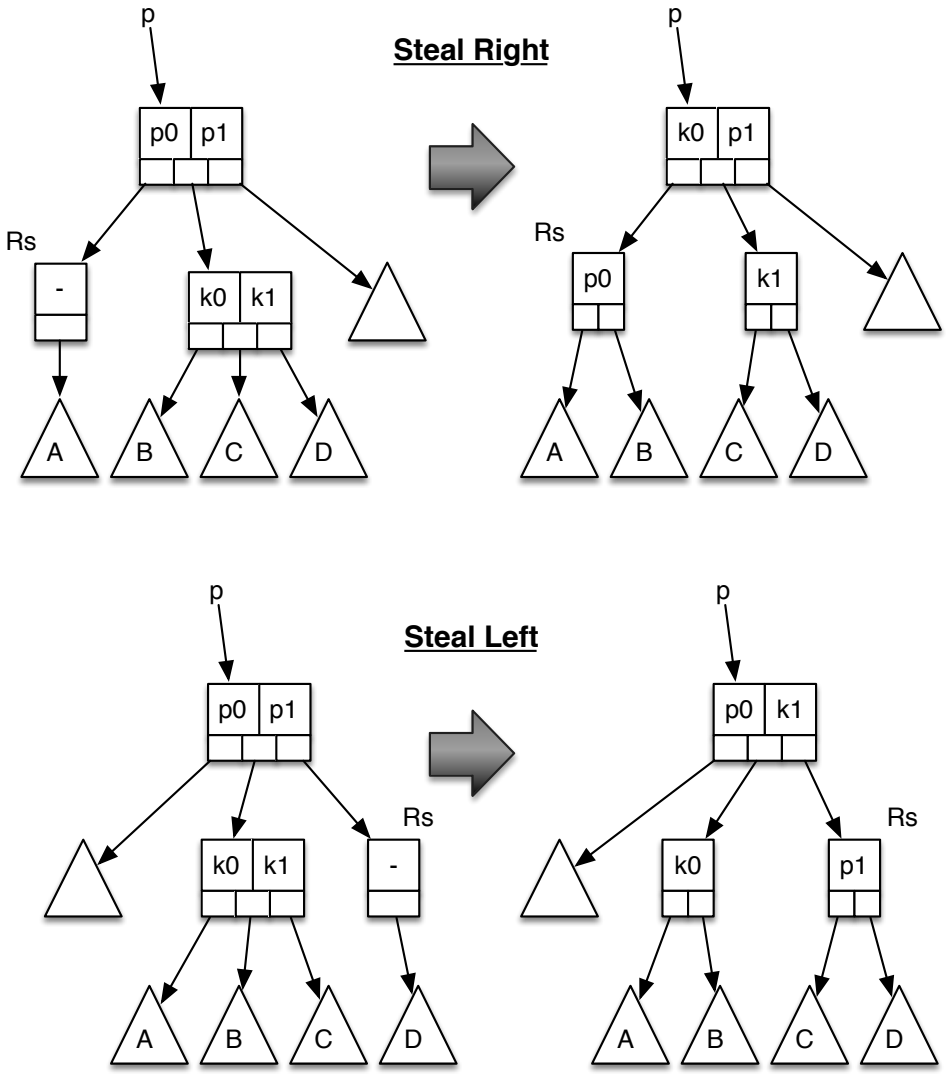
$$\log_3 m = \frac{\log_2(m)}{\log_2 3} = \frac{1}{\log_2 3}\log_2 m$$

This means that the number of recursive calls is $O(h-1) = O(\frac{1}{\log_2 3}\log_2 m - 1) = O(\log_2 m)$ where m is the number of nodes. But this isn't quite what we need. We want to know the time

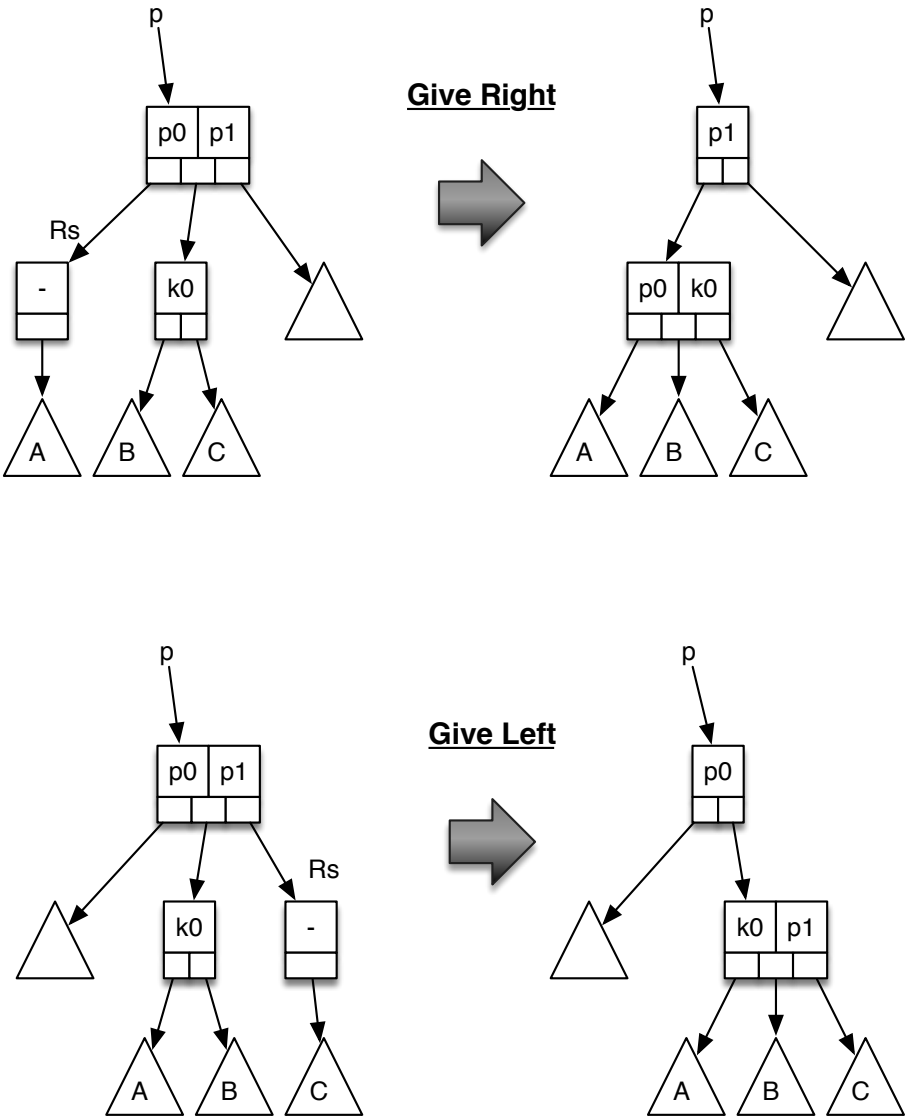
complexity in terms of n, the number of **elements** in the tree. In a 2-3 tree there are always **more** leaf nodes than internal nodes. Thus $n > m/2$, which implies $\log_2 n > \log_2 m - 1$, which means $\log_2 n + 1 > \log_2 m$, and therefore $\log m$ is $O(\log_2 n + 1) = O(\log_2 n)$. Therefore the number of recursive calls is also $O(\log_2 n)$.

Each recursive call is $O(1)$, so the recursive portion of the algorithm for insertion into a 2-3 tree is $O(\log n)$ where n is the number of elements in the tree. Finally, there is a constant amount of time to check the special cases in the non-recursive portion of the insertion algorithm, which means the overall complexity is $O(\log n)$.

Exercise 4



Exercise 5



Exercise 6

Deletion is $O(\log n)$ where n is the number of **elements** in the tree. The argument is very similar to the insertion case. There are $O(1)$ statements before and after the recursive call because the steal and give operations are $O(1)$. The argument for determining the number of recursive calls begin $O(logn)$ is identical to that for insertion. Therefore the recursive portion of the deletion algorithm is $O(\log n)$. The non-recursive portion is $O(1)$, so deletion is $O(\log n)$ overall.

Exercise 7

See `TwoThreeNode280.java`, `InternalTwoThreeNode280.java` and `LeafTwoThreeNode280.java`.