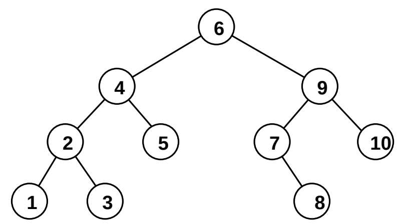
Computer Science 501
Data Structures & Algorithms
The College of Saint Rose
Fall 2015

Topic Notes: Binary Search Trees

Possibly the most common usage of a binary tree is to store data for quick retrieval.

Definition: A binary tree is a *binary search tree* (BST) iff it is empty or if the value of every node is both greater than or equal to every value in its left subtree and less than or equal to every value in its right subtree.



Note that a BST is an ordered structure. That is, it maintains all of its elements in order. Like our other ordered structures, this restriction will allow us to build more efficient implementations of the most important operations on the tree.

We will refer to the implementation of binary search trees in the structure package, the same package we have been using for graph implementations.

See Structure Source:
`/home/cs501/src/structure5/BinarySearchTree.java`

A note from the data structures implementation perspective. Note that the implementation does not expose the tree nodes, as stucture's `BinaryTree` implementation does. This is precisely because we need to be more restrictive to enforce the ordered nature of the structure.

We do, however, make use of the existing `BinaryTree` implementation to take care of the details of storing the tree. The public methods of `BinarySearchTree` then make use of the underlying `BinaryTree` as appropriate.

In addition to the underlying `BinaryTree`, whose root is stored as an instance variable, the implementation maintains a count of the number of nodes and remembers a `Comparator` (by default, the `NaturalComparator`).

Any interesting operation in a BST (`get`, `add`, `remove`) has to begin with a search for the correct place in the tree to do the operation. Our implementation has a helper method used by all of these to do just that. We want to find the `BinaryTree` whose root either contains the value, or, if the value is not to be found, the node whose non-existent child would contain the value.

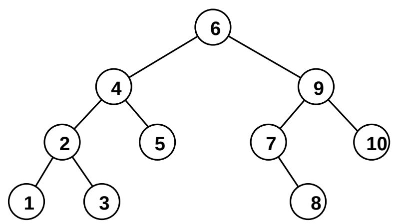
This is just a binary search, and is performed in the protected method `locate`.

Next, let's consider the `add` method. We will be creating a new `BinaryTree` with the given value, and setting parent and child links in the tree to place this new value appropriately.

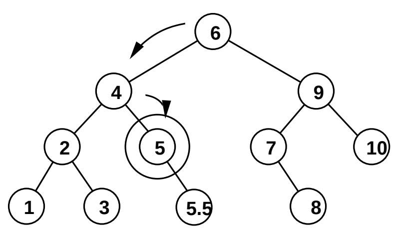
We need to consider several cases:

1. We are adding to an empty binary tree, in which case we just make the new node the root of the BST.

2. We add the new value as a new leaf at the appropriate place in the tree as the child of a current leaf. For example, add 5.5 to this tree:

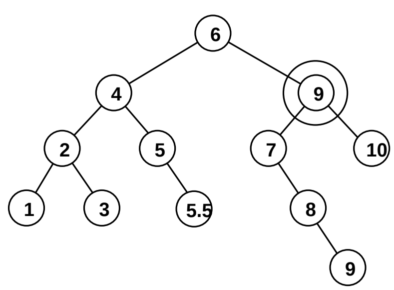


   to get



3. If the value is one that is already in the tree, the search may stop early, in which case we need to add the new item as a child of the predecessor of the node found.

   For example, adding 9 to the tree above:



   We can choose to add duplicate values as either left or right children as long as we're consistent. Our implementation chooses to put them at the left.

Our `add` implementation used a helper method `predecessor`. We have this, plus a helper method to find the `successor`.

These are pretty straightforward. Our predecessor is the rightmost entry in our left subtree, and the successor is the leftmost entry in the right subtree.

The `get` method is very simple. It's just a `locate`, followed by a check to make sure the value was found at the location.

`remove` is more interesting.

Again, there are a number of possibilities:

1. We are removing a leaf, which is very straightforward.
2. We are removing the root of the entire tree, in which case we need to change the root of the BST.
3. We are removing an internal node, in which case we need to merge the children into a single subtree to replace the removed internal node.
4. The item is not found, so the tree is unmodified and we return `null`.

The implementation separates out three important cases: removing the root, removing a node which is a left subtree, or removing a node which is a right subtree. In each case, we replace the removed node with a tree equivalent to its two subtrees, merged together. This is accomplished through the `removeTop` method.

This method is also broken into a number of cases:

1. Easiest cases: If either child of top is empty, just use the other child as the new top. Done. These are illustrated in cases (a) and (b) in Figure 14.2 of Bailey (p. 352 of text, 368th page of the PDF).
2. Another easy case: top's left child has no right child. Here, just assign top's right to be the right child of left child and call the left child the new top. This is case (c) in Bailey Figure 14.2.
3. The more complex case remains: left child has a right child.

   Here, we will proceed by locating the predecessor of the top node, make that the new top, and rearrange the subtrees in the only way that retains the order of the entire tree. This is illustrated in Bailey Figure 14.3.

What about complexity of these operations?

`add`, `get`, `contains`, and `remove` are all proportional to the height of the tree. So if the tree is well-balanced, these are $\Theta(\log n)$, but all are $\Theta(n)$ in the worst case.

Tree Sort

One of many ways we can use a BST is for sorting: the *tree sort*.

We can build a BST containing our data and then do an inorder traversal to retrieve them in sorted order. Since the cost of entering an element into a (balanced) binary tree of size n is $\log n$, the cost of building the tree is

$(\log 1) + (\log 2) + (\log 3) + \cdots + (\log n) = \Theta(n \log n)$ compares.

The inorder traversal is $\Theta(n)$. The total cost is $\Theta(n \log n)$ in both the best and average cases.

The worst case occurs if the input data is in order. In this case, we're essentially doing an insertion sort, creating a tree with one long branch. This results in a tree search as bad as $\Theta(n^2)$.

In the average case, Tree Sort is good – plus it has the side effect of building an ordered structure that is potentially useful and interesting for other reasons.

Comparisons of advanced sorts

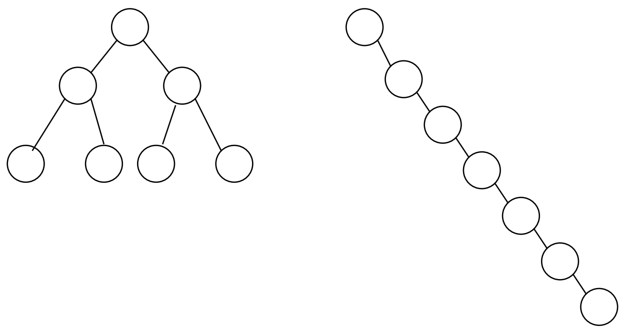
To recap what we know about our sorting routines.

- Quicksort is fastest on average – $\Theta(n \log n)$, but has worst case behavior of $\Theta(n^2)$ when we are unable to generate good partitions.
  - Low overhead makes it perform well on average.
- HeapSort takes $\Theta(n \log n)$ in average and worst case.
  - On random data, it is somewhat slower than Quicksort and MergeSort.
  - If you only need to retrieve the first few items of a collection rather than the entire set, it can be better since the initial heapify can be done in time $\Theta(n)$.
- MergeSort takes $\Theta(n \log n)$ in average and worst case, $\Theta(n)$ extra space.
  - On random data, it's likely to be somewhat slower than Quicksort.
  - It performs well on external files where all data will not fit into memory.
- Tree Sort is $(O(n \log n)$ on average, but is $\Theta(n^2)$ in the worst case, and takes $\Theta(n)$ extra space.
  - It does well on random data.
  - Has the side-effect of building an ordered structure which, when balanced, allows $\Theta(\log n)$ search capabilities.

See Example:
`/home/cs501/examples/SortingComparisons`

Balanced Trees

The efficiency of the operations on binary search trees (and other binary trees, for that matter) relies on them being reasonably *balanced trees*. That is, the height is proportional to $\log n$, not n.



Just having the same height on each child of the root is not enough to maintain a $\Theta(\log n)$ height for a binary tree.

We can define a *balance condition*, some set of rules about how the subtrees of a node can differ.

Maintaining a perfectly strict balance (minimum height for the given number of nodes) is often too expensive. Maintaining too loose a balance can destroy the $\Theta(\log n)$ behaviors that often motivate the use of tree structures in the first place.

For a strict balance, we could require that all levels except the lowest are full.

How could we achieve this? Let's think about it by inserting the values 1,2,3,4,5,6,7 into a BST and seeing how we could maintain strict balance.

First, insert 1:

```
1
```

Next, insert 2:

```
1
 \
  2
```

We're OK there. But when we insert 3:

```
1
```

```
    \
     2
      \
       3
```

we have violated our strict balance condition. Only one tree with these three values satisies the condition:

```
   2
  / \
 1   3
```

We will see how to “rotate” the tree to achieve this shortly.

Now, add 4:

```
   2
  / \
 1   3
      \
       4
```

Then add 5:

```
   2
  / \
 1   3
      \
       4
        \
         5
```

Again, we need to fix the balance condition. Here, we can apply one of these rotations on the right subtree of the root:

```
   2
  / \
 1   4
    / \
   3   5
```

Now, add 6:

```
    2
   / \
  1   4
     / \
    3   5
         \
          6
```

It's not completely obvious how to fix this one up, and we won't worry about it just now. We do know that after we insert the 7, there's only one permissible tree:

```
      4
     / \
   2     6
  / \   / \
 1   3 5   7
```

So maintaining strict balance can be very expensive. The tree adjustments can be more expensive than the benefits.

There are several options to deal with potentially unbalanced trees without requiring a perfect balance.

Two are discussed in detail in the Bailey text and have implementations in the structure package. The third is not.

1. *Red-black trees* – nodes are colored red or black, and place restrictons on when red nodes and black nodes can cluster.

   See Structure Source:
   `/home/cs501/src/structure5/RedBlackTree.java`

   Red-black trees are described briefly in the text. We may discuss them later.

2. *AVL Trees* - Adelson-Velskii and Landis developed these in 1962. We will look at these.

3. *Splay trees* – every reference to a node causes that node to be relocated to the root of the tree.

   See Structure Source:
   `/home/cs501/src/structure5/SplayTree.java`

   This is very unusual! We have a `contains()` operation that actually modifies the structure.

   This works very well in cases where the same value or a small group of values are likely to be accessed repeatedly.

   We may talk more later about splay trees as well.

4. *2-3 Trees* – tree nodes can hold more than one key – described in the Levitin text.

AVL Trees

We consider *AVL Trees*, developed by and named for Adelson-Velskii and Landis, who invented them in 1962.

The balance condition for AVL trees (the *AVL condition)*: the heights of the left and right subtrees of any node can differ by at most 1.

To see that this is less strict than perfect balance, let's consider two trees:

```
        5
     /     \
   2        8
  / \      /
1    4   7
    /
   3
```

This one satisfies the AVL condition (to decide this, we check the heights at each node), but is not perfectly balanced since we could store these 7 values in a tree of height 2.

But...

```
        7
     /     \
   2        8
  / \
1    4
    / \
   3   5
```

This one does not satisfy the AVL condition – the root node violates it!

So the goal is to maintain the AVL balance condition each time there is an insertion (we will ignore deletions, but similar techniques apply).

When inserting into the tree, a node in the tree can become a violator of the AVL condition. Four cases can arise which characterize how the condition came to be violated. Let's call the violating node A.

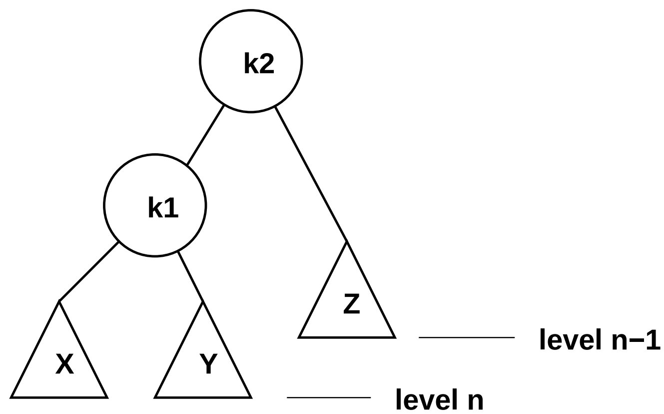
1. Insertion into the left subtree of the left child of A.
2. Insertion into the right subtree of the left child of A.
3. Insertion into the left subtree of the right child of A.

4. Insertion into the right subtree of the right child of A.

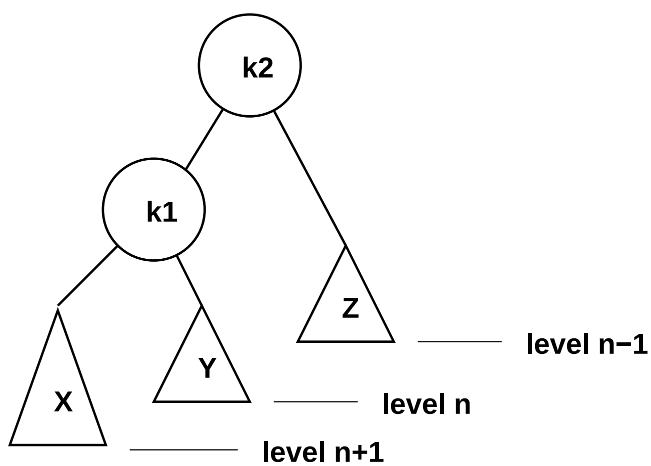
In reality, however, there are only two really different cases, since cases 1 and 4 and cases 2 and 3 are mirror images of each other and similar techniques apply.

First, we consider a violation of case 1.

We start with a tree that satisfies AVL:



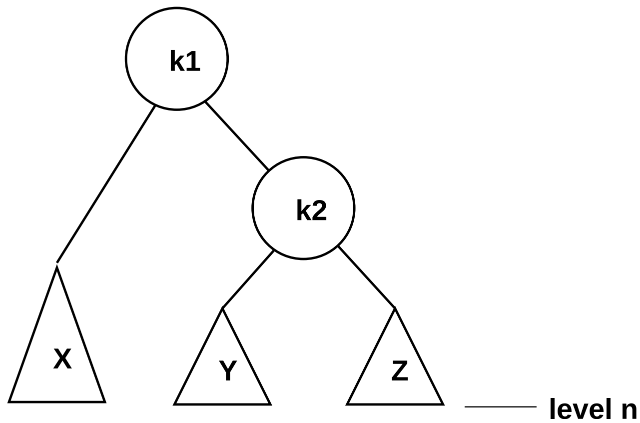
After an insert, the subtree X increases in height by 1:



So now node k_2 violates the balance condition.

We want to perform a *single rotation* to obtain an equivalent tree that satisfies AVL.

Essentially, we want to switch the roles of k_1 and k_2, resulting in this tree:



For this insertion type (left subtree of a left child – case 1), this rotation has restored balance.

We can think of this like you have a handle for the subtree at the root and gravity determines the tree.

If we switch the handle from k_2 to k_1 and let things fall where they want (in fact, must), we have rebalanced.

Consider insertion of 3,2,1,4,5,6,7 into an originally empty tree.

Insert 3:

```
        3
```

Insert 2:

```
        3
       /
      2
```

Insert 1:

```
        3                  2
       /                  / \
      2           --->   1   3
     /
    1
```

Here, we had to do a rotation. We essentially replaced the root of the violating subtree with the root of the taller of its children.

Now, we insert 4:

```
     2
    / \
   1   3
        \
         4
```

Then insert 5:

```
     2
    / \
   1   3  <-- AVL violated here (case 4)
        \
         4
          \
           5
```

and we have to rotate at 3:

```
     2
    / \
   1   4
      / \
     3   5
```

Now insert 6:

```
     2   <-- AVL violated here (case 4)
    / \
   1   4
      / \
     3   5
          \
           6
```

Here, our rotation moves 4 to the root and everything else falls into place:

```
     4
    / \
   2   5
  / \   \
 1   3   6
```

Finally, we insert 7:

```
      4
     / \
    2   5  <-- AVL violated here (case 4 again)
   / \   \
  1   3   6
           \
            7
```

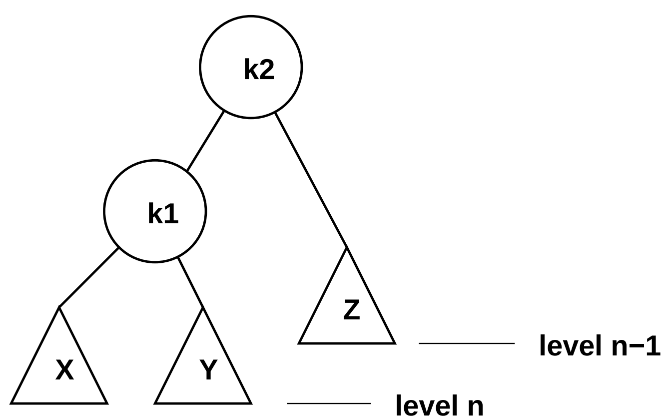
```
      4
     /   \
    2     6
   / \   / \
  1   3 5   7
```

We achieve perfect balance in this case, but this is not guaranteed in general.

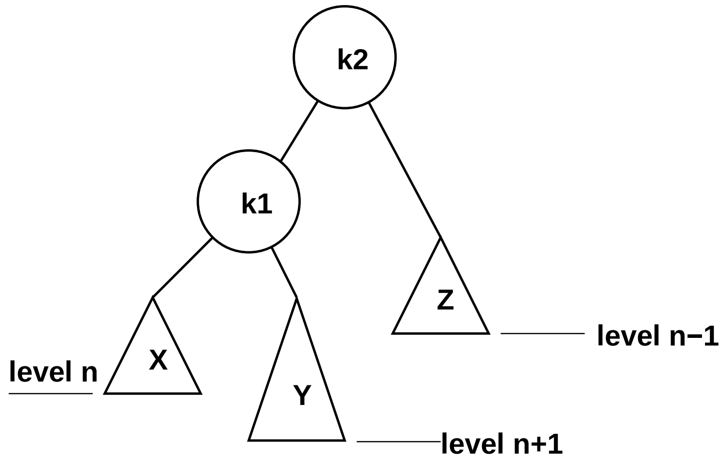
This example demonstrates the application of cases 1 and 4, but not cases 2 and 3.

Here's case 2:

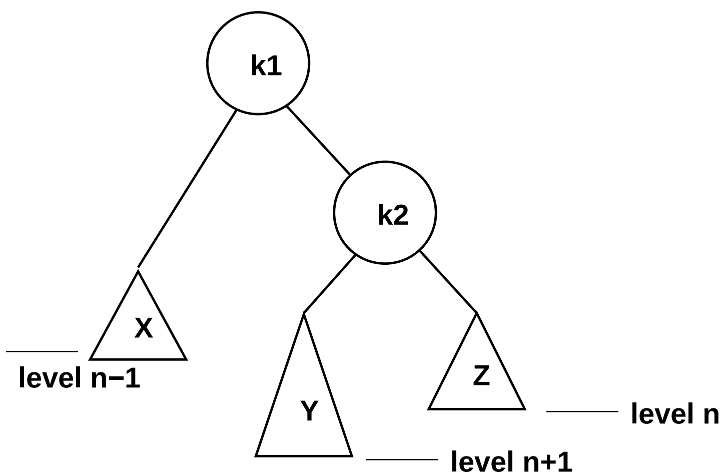
We start again with the good tree:



But now, our inserted item ends up in subtree Y:

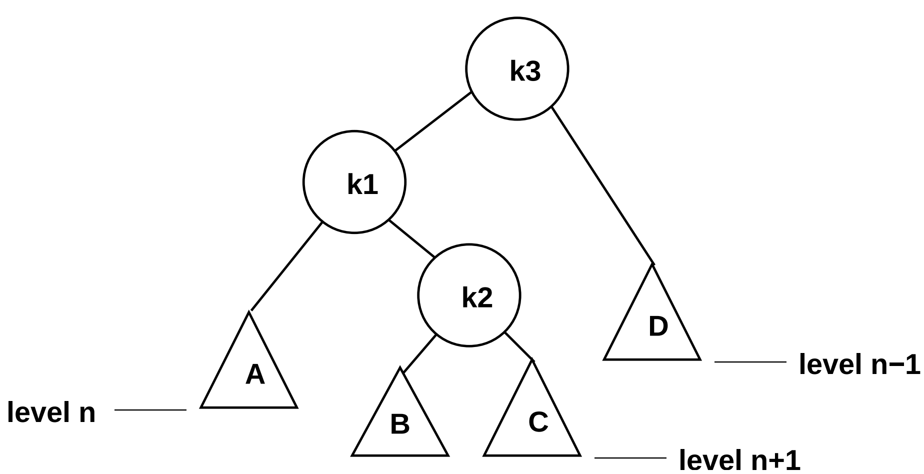


We can attempt a single rotation:



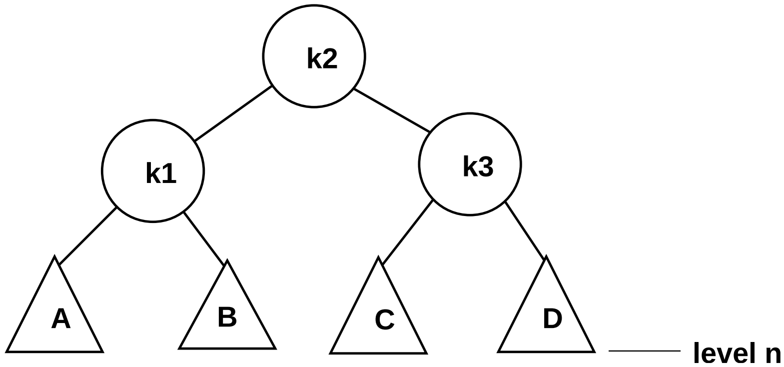
This didn't get us anywhere. We need to be able to break up Y.

We know subtree Y is not empty, so let's draw our tree as follows:



Here, only one of B or C is at level $n+1$, since it was a single insert below k_2 that resulted in the AVL condition being violated at k_3 with respect to its shorter child D.

We are guaranteed to correct it by moving D down a level and both B and C up a level:



We're essentially rearranging k_1, k_2, and k_3 to have k_2 at the root, and dropping in the subtrees in the only locations where they can fit.

In reality, only one of B and C is at level n – the other only descends to level $n-1$.

Case 3 is the mirror image of this.

To see examples of this, let's pick up the previous example, which had constructed a perfectly-balanced tree of the values 1–7.

```
      4
    /   \
   2     6
  / \   / \
 1   3 5   7
```

At this point, we insert a 16, then a 15 to get:

```
      4
    /   \
   2     6
  / \   / \
 1   3 5   7
            \
             16
            /
          15
```

Node 7 violates AVL and this happened because of an insert into the left subtree of its right child. Case 3.

So we let k_1 be 7, k_2 be 15, and k_3 be 16 and rearrange them to have k_2 at the root of the subtree, with children k_1 and k_3. Here, the subtrees A, B, C, and D are all empty.

We get:

```
     4
   /   \
  2     6
 / \   / \
1   3 5   15
         /  \
        7    16
```

Now insert 14.

```
     4
   /   \
  2     6
 / \   / \
1   3 5   15
         /  \
        7    16
         \
          14
```

This violates AVL at node 6 (one child of height 0, one of height 2).

This is again an instance of case 3: insertion into the left subtree of the right child of the violating node.

So we let k_1 be 6, k_2 be 7, and k_3 be 15 and rearrange them again. This time, subtrees A is the 5, B is empty, C is the 14, and D is the 16.

The *double rotation* requires that 7 become the root of that subtree, the 6 and the 15 its children, and the other subtrees fall into place:

```
      4
    /   \
  2       7
 / \     / \
1   3  6    15
      /    /  \
     5   14    16
```

Insert 13:

```
        4
      /   \
    2       7
   / \     / \
  1   3   6    15
         /    /  \
        5   14    16
           /
          13
```

What do we have here? Looking up from the insert location, the first element that violates the balance condition is the root, which has a difference of two between its left and right child heights.

Since this is an insert into the right subtree of the right child, we're dealing with case 4. This requires just a single rotation, but one done all the way at the root. We get:

```
             7
           /   \
        4          15
      /   \       /  \
    2       6   14    16
   / \     /   /
  1   3   5  13
```

Now adding 12:

```
             7
           /   \
        4          15
      /   \       /  \
    2       6   14    16
   / \     /   /
  1   3   5  13
            /
           12
```

The violation this time is at 14, which is a simple single rotation (case 1):

```
             7
           /   \
        4          15
      /   \       /  \
    2       6   13    16
   / \     /   /  \
  1   3   5  12   14
```

Inserting 11:

```
           7
         /   \
      4         15
    /   \      /  \
  2      6   13    16
 / \     /  /  \
1   3   5  12  14
          /
         11
```

Here, we have a violation at 15, case 1, so another single rotation there, promoting 13:

```
           7
         /   \
      4         13
    /   \      /  \
  2      6   12    15
 / \     /  /     /  \
1   3   5  11   14    16
```

(Almost done)

Insert 10:

```
           7
         /   \
      4         13
    /   \      /  \
  2      6   12    15
 / \     /  /     /  \
1   3   5  11   14    16
          /
         10
```

The violator here is 12, case 1:

```
           7
         /   \
      4         13
    /   \      /   \
  2      6   11     15
 / \     /  / \    /  \
1   3   5  10  12 14  16
```

Then we finally add 8 (no rotations needed) then 9:

```
           7
         /   \
      4         13
    /   \     /   \
  2       6   11     15
 / \     /  / \    /  \
1   3   5  10  12 14  16
          /
         8
          \
           9
```

Finally we see case 2 and do a double rotation with 8, 9, and 10 to get our final tree:

```
           7
         /   \
      4         13
    /   \     /   \
  2       6   11     15
 / \     /  / \    /  \
1   3   5  9  12 14  16
          / \
         8   10
```

This tree is not strictly balanced – we have a hole under 6's right child, but it does satisfy AVL.

You can think about how we might implement an AVL tree, but we will not consider an actual implementation. However, AVL insert operations make excellent exam questions, so keep that in mind when preparing for the final.

The whole point of considering AVL trees is to maintain a reasonable balance, and hopefully, a tree height that looks like $\log n$. We will not do a detailed analysis, but the height n of an AVL tree is guaranteed to satisfy the inequality:

$$\lfloor \log_2 n \rfloor \leq h < 1.44.05 \log_2(n+2) - 1.3277.$$

We have log factors on both sides, leading to $\Theta(\log n)$ worst case behavior of search and insert operations.