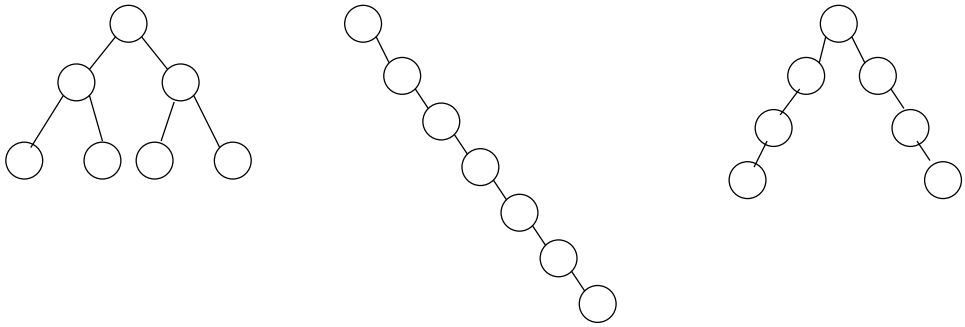
SIENA
COMPUTER SCIENCE

Computer Science 385
Design and Analysis of Algorithms
Siena College
Spring 2025

Topic Notes: Balanced Search 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.

(We will work through this in class)

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.

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

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.

   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.

4. *2-3 Trees* – tree nodes can hold more than one key – described in the Levitin Algorithms text and we will study them in class and lab.

AVL Trees

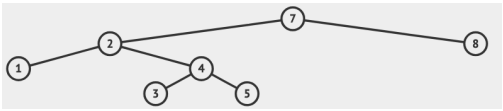
We consider *AVL Trees*, developed by and named for Adelson-Velskii and Landis, who invented them in 1962.

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:

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...



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.

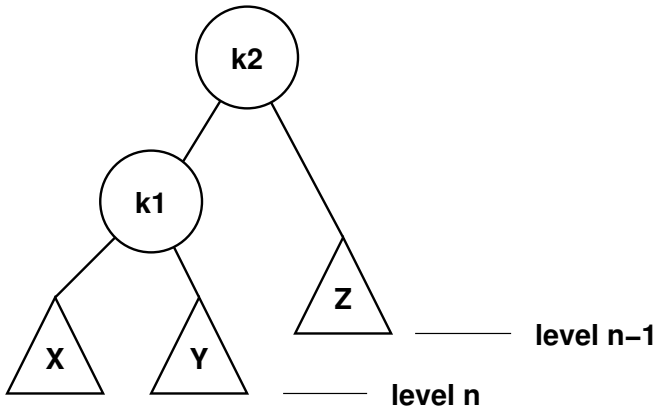
1. Insertion into the left subtree of the left child of A.
2. Insertion into the right subtree of the left child of A.
3. Insertion into the left subtree of the right child of A.

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

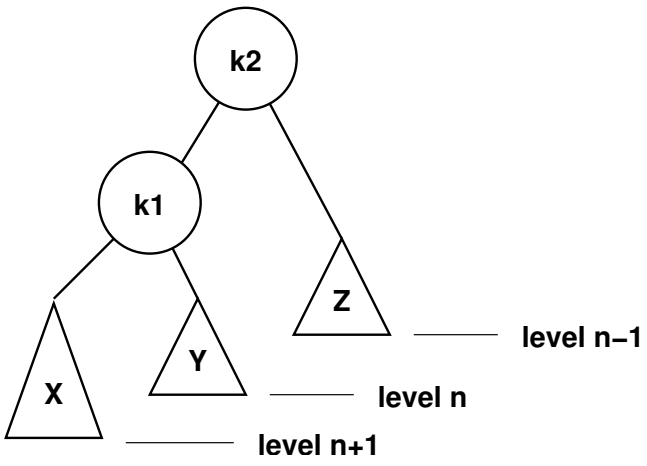
In reality, however, there are only two really different cases, since cases 1 and 4 and cases 2 and 3 are mirror images of each other and similar techniques apply.

First, we consider a violation of case 1.

We start with a tree that satisfies AVL:



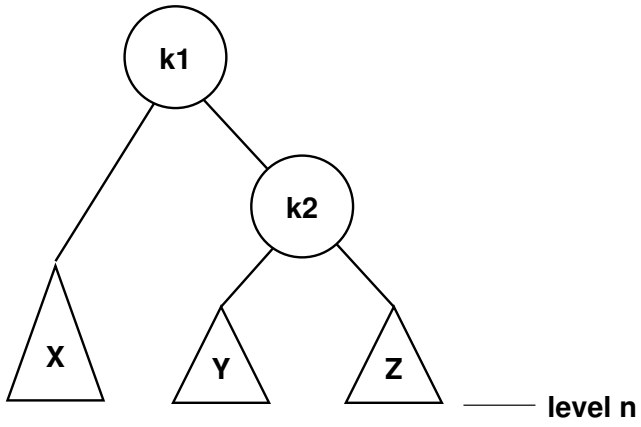
After an insert, the subtree X increases in height by 1:



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:

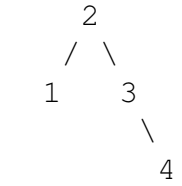
```
    3
   /
  2
```

Insert 1:

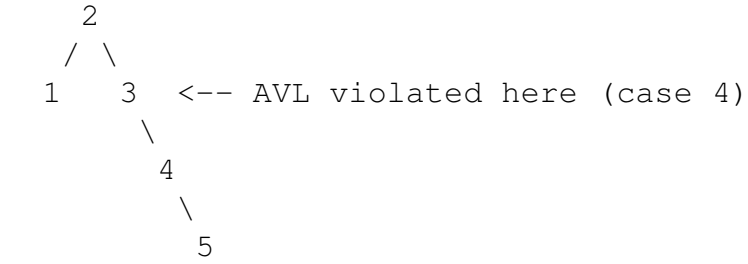
```
      3                    2
     /                    / \
    2          --->      1   3
   /
  1
```

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.

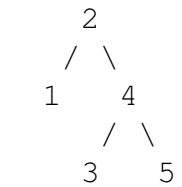
Now, we insert 4:



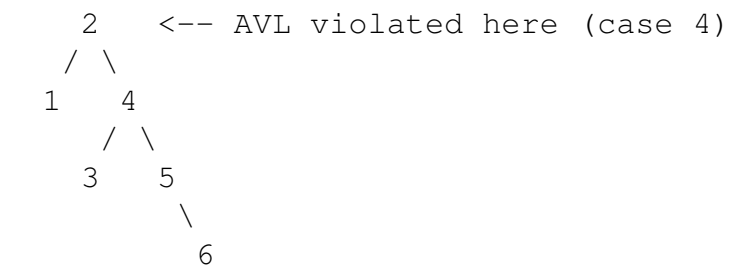
Then insert 5:



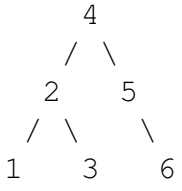
and we have to rotate at 3:



Now insert 6:



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



Finally, we insert 7:

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

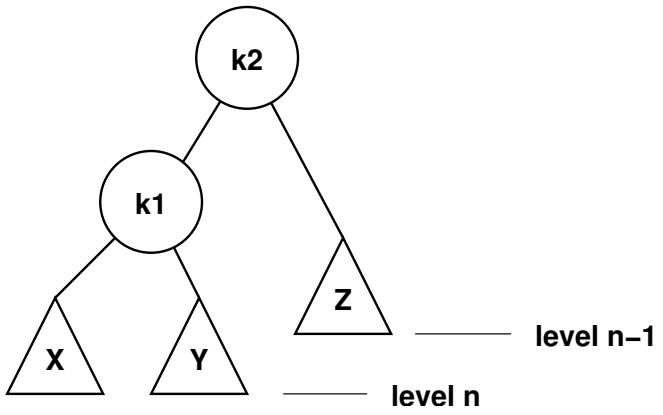
```
      4
    /   \
   2     6
  / \   / \
 1   3 5   7
```

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

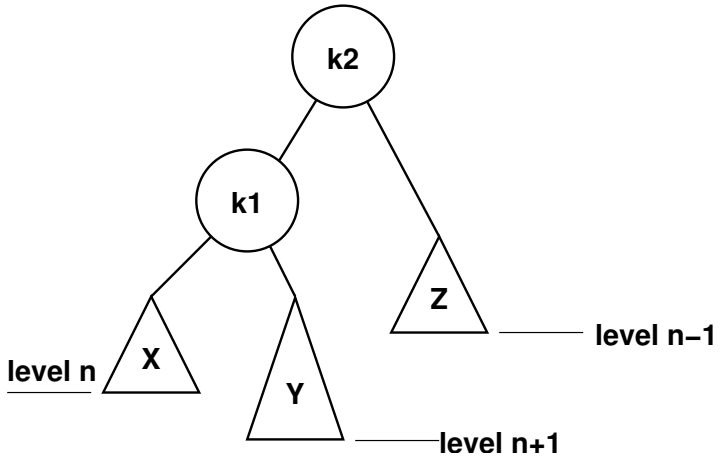
This example demonstrates the application of cases 1 and 4, but not cases 2 and 3.

Here's case 2:

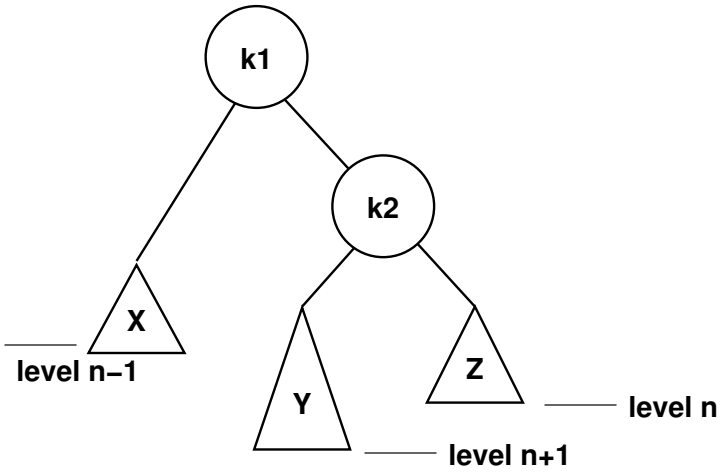
We start again with the good tree:



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

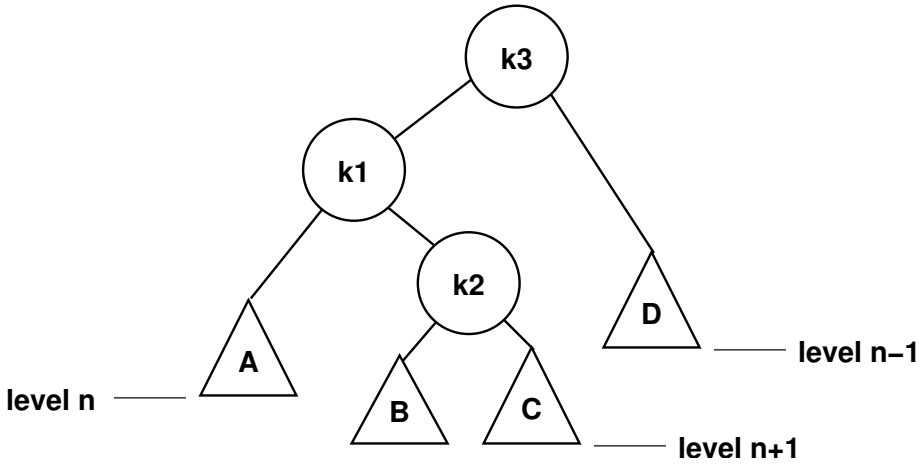


We can attempt a single rotation:



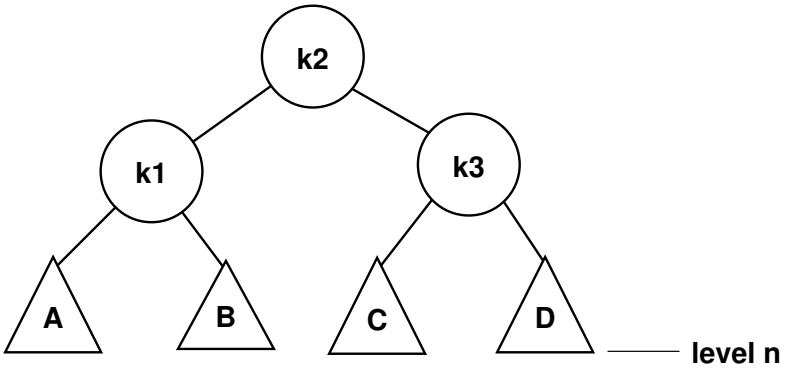
This didn't get us anywhere. We need to be able to break up Y.

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:

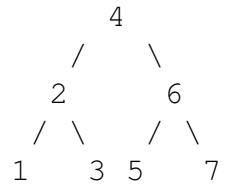


We're essentially rearranging k_1, k_2, and k_3 to have k_2 at the root, and dropping in the subtrees in the only locations where they can fit.

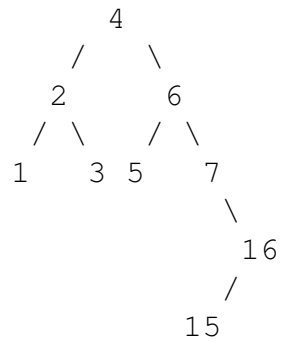
In reality, only one of B and C is at level n – the other only descends to level $n - 1$.

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.



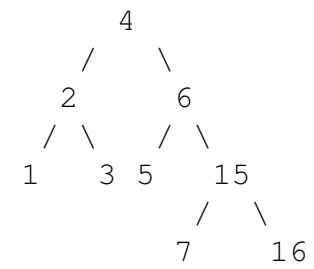
At this point, we insert a 16, then a 15 to get:



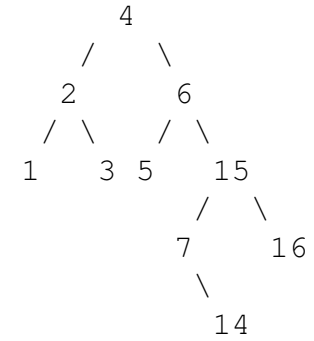
Node 7 violates AVL and this happened because of an insert into the left subtree of its right child. Case 3.

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:



Now insert 14.

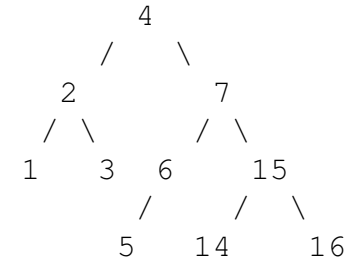


This violates AVL at node 6 (one child of height 0, one of height 2).

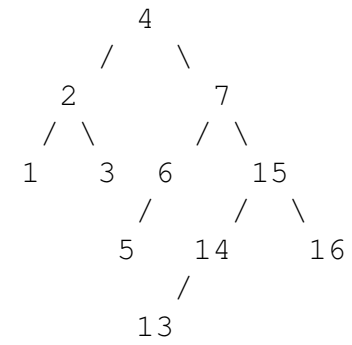
This is again an instance of case 3: insertion into the left subtree of the right child of the violating node.

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:

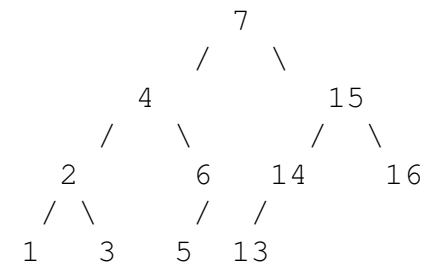


Insert 13:

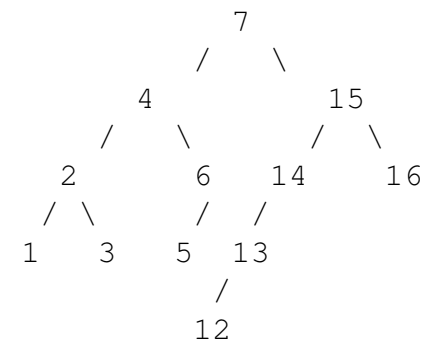


What do we have here? Looking up from the insert location, the first element that violates the balance condition is the root, which has a difference of two between its left and right child heights.

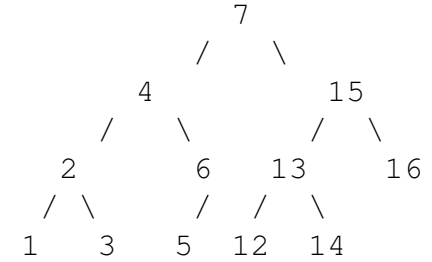
Since this is an insert into the right subtree of the right child, we're dealing with case 4. This requires just a single rotation, but one done all the way at the root. We get:



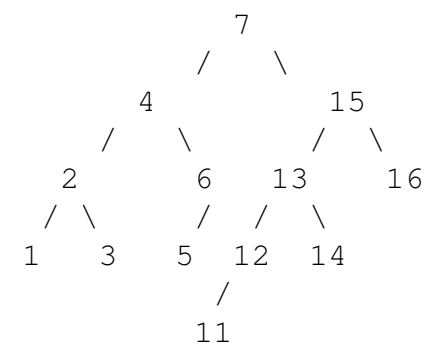
Now adding 12:



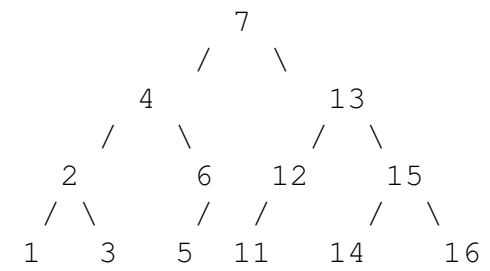
The violation this time is at 14, which is a simple single rotation (case 1):



Inserting 11:

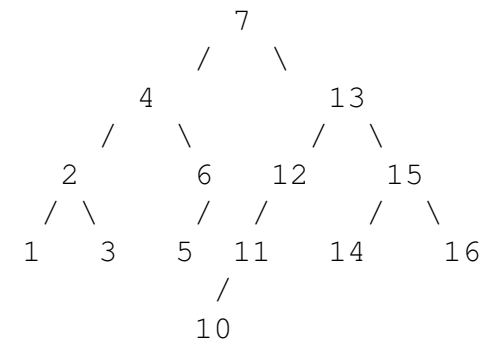


Here, we have a violation at 15, case 1, so another single rotation there, promoting 13:

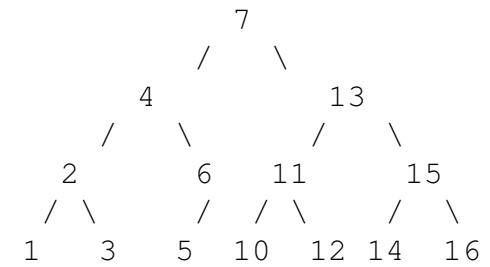


(Almost done)

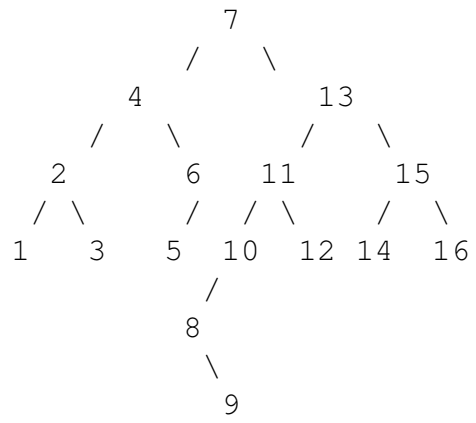
Insert 10:



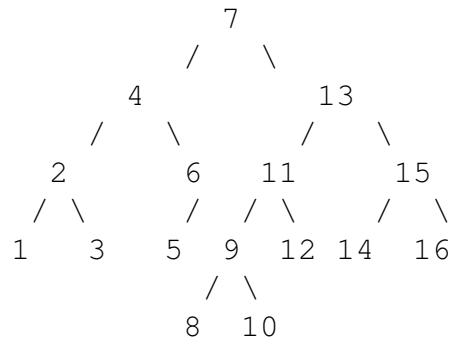
The violator here is 12, case 1:



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



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



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:

$$\lfloor \log_2(n+1) \rfloor \leq h < 1.44 \log_2(n+2) - 0.328.$$

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

2-3 Trees

You are familiar with the idea of a *binary search tree* and with the importance of maintaining a *balance condition* in those binary search trees to maintain the $\Theta(\log n)$ behavior of many operations on those trees.

Popular ways to maintain balance in binary search trees include *AVL Trees* and *red-black Trees*, each of which places rules when parts of the tree will need to be "rotated" to restore balance after an operation that modifies the tree has caused it to violate its balance condition.

Here, we will consider another variation on the search tree that maintains balance during construction by allowing nodes of the tree to contain multiple keys. Therefore, these are not **binary** search trees, but they are still **balanced** search trees. The specific construct will will study is called a *2-3 Tree*.

Key features and properties of a 2-3 tree include:

- Each node in the tree stores either 1 or 2 key values
- Nodes that are storing 1 key value are called *2-nodes* and have either 0 (when a leaf node) or 2 (when an interior node) children
- Nodes that are storing 2 key values are called *3-nodes* and have either 0 (when a leaf node) or 3 (when an interior node) children
- A valid 2-3 tree is always *perfectly balanced*, in the sense that every node in the tree's bottom level is a leaf, and every other node in the tree is an interior node

In class, we will work through examples of building 2-3 trees. The idea is very similar to that of inserting into an AVL tree or other type of balanced tree:

- We insert the value by visiting the interior nodes and deciding whether the new value should be added to its left or right subtree (for a 2-node) or its left, middle, or right child (for a 3-node). 2-nodes work just like nodes in a traditional binary search tree in that values smaller than its key go to the left subtree, and values larger than its key go to the right subtree. In 3-nodes, we have 3 choices. Values smaller than both keys go to the left subtree, values between the two keys go to the middle subtree, and values larger than both keys go to the right subtree.
- Once a leaf node is encountered, the value is added to that leaf. If the leaf was a 2-node, it becomes a 3-node, and we are done. If the leaf was a 3-node, it would then be a 4-node (one containing 3 keys and with potentially 4 children). This is not permitted in a 2-3 tree, so the tree must now be reconfigured to be a 2-3 tree once again. This is accomplished by splitting the node into two new 2-nodes, one with the node's smallest value and one with the node's largest. The middle value is promoted to the parent. If the node was already the root of the entire tree, the promoted value will become the new root. Otherwise, the middle value is added as a new key to the parent. If it was a 2-node, it is now a 3-node and we are done. Otherwise, it is now a 4-node, and must again be split. This process continues back up the tree until either a 2-node becomes a 3-node and we can stop, or the entire tree gets a new root.

We will run through one or two examples of the construction of 2-3 trees in class.

It is fairly straightforward to see that the height of a 2-3 tree containing n keys falls between $\log_3 n$ (for a tree consisting of only 3-nodes) and $\log_2 n$ (for a tree consisting only of 2-nodes).

From this, we can see that an insertion is $\Theta(\log n)$, because finding the leaf for insertion will involve either 1 or 2 comparisons at each level, of which there are no more than $\log_2 n$. And then if the tree needs configuration, the changes can only propagate back up through those same number of levels, possibly introducing a new level in the case of a new root being created.

Let's consider an algorithm to check if a 2-3 tree contains a given value:

ALGORITHM CONTAINS23TREE(T, k)
//Input: a 2-3 tree node T, a search key k
//Output: a boolean indicating whether k is stored anywhere in the 2-3 tree rooted at T
if T is a 2-node **then**
  if $T.key1 = k$ **then**
    return true
  if T is a leaf **then**
    return false
  // it's not a leaf, we need to search in a child
  if $k < T.key1$ **then**
    return $Contains23Tree(T.left, k)$
  else
    return $Contains23Tree(T.right, k)$
else
  // T is a 3-node
  if $T.key1 = k$ **or** $T.key2 = k$ **then**
    return true
  if T is a leaf **then**
    return false
  // it's not a leaf, we need to search in a child
  if $k < T.key1$ **then**
    return $Contains23Tree(T.left, k)$
  else if $k > T.key2$ **then**
    return $Contains23Tree(T.right, k)$
  else
    return $Contains23Tree(T.middle, k)$

We can quickly note that this algorithm has no loops, and in the worst case it needs to recurse to a leaf node. Since the height of the tree is $\Theta(\log n)$, the entire algorithm is $\Theta(\log n)$.

We will write additional algorithms to operate on 2-3 trees.