2-3 Tree PseudoCode

Node Class

Allow it to have 3 item spots, 1 for a temporary location.
Allow it to have 4 children, 1 for a temporary location.
Needs to have an isLeaf field.
Needs to have a parent variable.
insertItem method needs to dynamically finds the correct spot and shifts the other elements out of the way.
deleteItem also returns the item at the same time. And does not shift indexes.
Have a maxItem find the biggest element in the list of children.
insertChild method use the maxItem method to locate where to place the child in the child array.
toString method for debugging.

Tree Class

findPosition method : node

Recursive method to find the spot in the tree, base case is if it is a leaf you return the node.
If the number we are trying to insert is less than the first item in the node, call to the first child.
If we have less than 2 items, or the number we are trying to insert is less than the second item, call to second child.
Else, we call to the third child.

Insert method

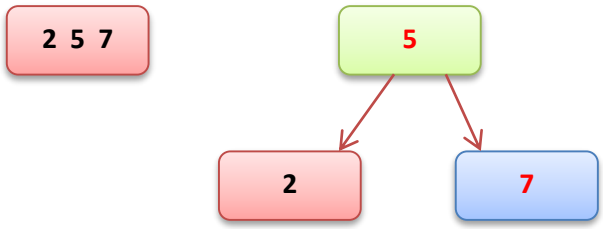
Use findPosition to the find the node
Use the node's insertion methods to insert at that leaf.
If we then have more than 2 items after the insertion, call the split method.

Split node method, Recursive method

First thing first, create a node which is the parent of the current node, and make a new node which is called sibling.

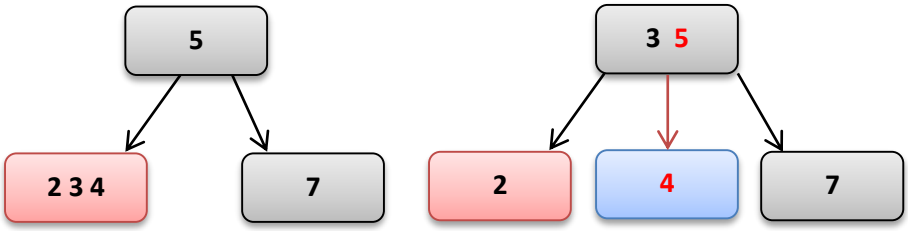
4 Cases for the node to be split

Case 1) It is the leaf and also the root. (parent is null and the node is leaf)



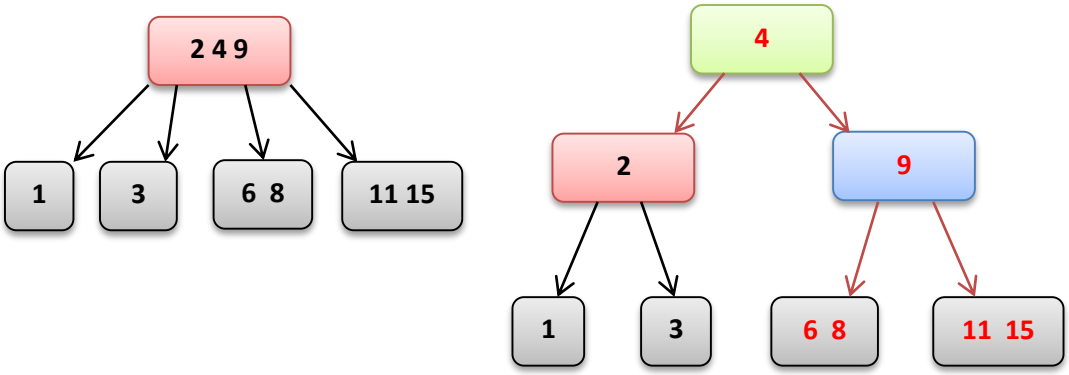
The new parent, set it to not be a leaf, set its item to be the 2nd item in old.
Set the children, node and sibling.
Sibling, it gets the 3rd item in the old node.
Set the parents of the node and the sibling

Case 2) It is a leaf that has a parent. (node is leaf, parent != null)



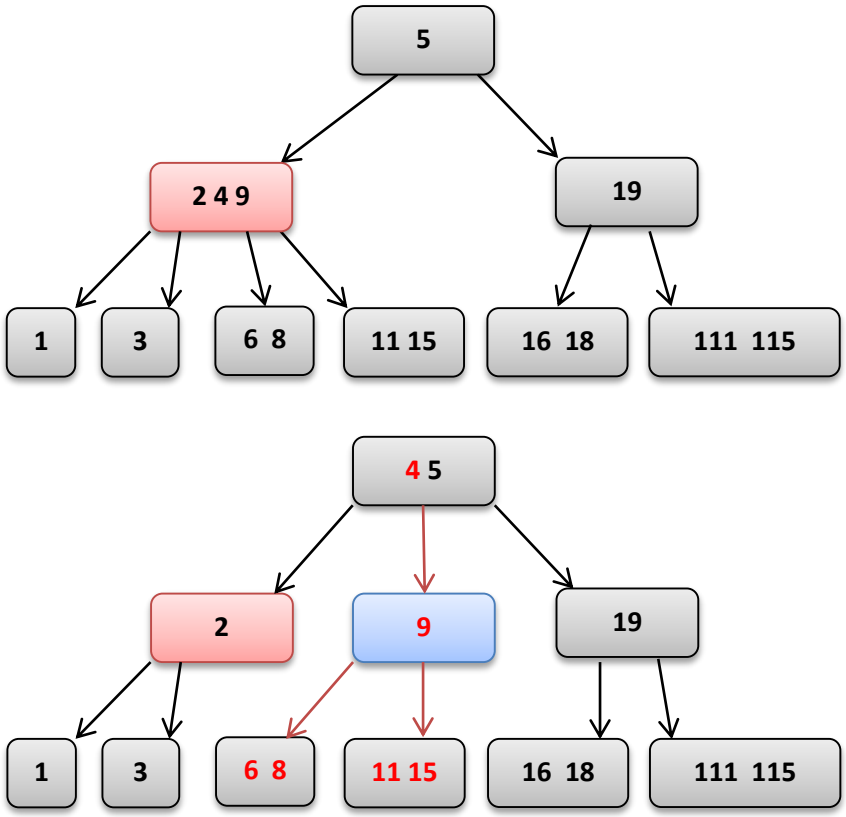
The process is, insert the middle item into the parent, and the 3rd item into the sibling
Set the siblings parent, and insert the sibling into the parent. (insertions are based on maxitem)

Case 3) It is the root with children (not a leaf, parent = null)



First create a new parent, its not a leaf. Insert into it the node's 2nd item.
Then insert the nodes 3rd item into the sibling, set the sibling's parent to the newParent, the sibling is not a leaf.
Then set the node's parent to the newParent, insert into the sibling the nodes 3rd and 4th child.
Update the sibling's new children's parent to the sibling itself.
Insert into the newParent the node and the sibling, the newParent is now the root.

Case 4) Internal node and not the root (not a leaf, and the node != root)



Add the node's 2nd item to the parent.
Add the node's 3rd item to the sibling, along with its 3rd and 4th child
Set these children to point to the sibling as their parent
Insert sibling into the parent.

Delete Method

Deletion will always begin at a leaf node.

Locate the node which has the item we want to delete.

If the node is not a leaf, we need to swap the item we want to delete with it's in-order successor.

However, the node is a leaf, we can just delete the item.

If leaf node is now empty after this deletion, we must now see if we can redistribute or merge.

Redistribute – Leaf Nodes (Example 2)

We can redistribute into an empty leaf node, if there is a sibling with 2 items.

Redistribute through the parent, maintaining the BST property.

Merge – Leaf Nodes (Example 1)

If we cannot redistribute, then we must merge, this happens when no sibling has 2 items to share.

The empty leaf will get removed, and the parent and sibling will merge.

Redistribute – Internal Nodes (No example, but very similar to Example 2)

We can redistribute into an empty internal node once again, if there is a sibling with 2 items to share.

Redistribute through the parent, while maintaining the BST property.

Merge – Internal Nodes (Example 3)

If redistribution is not possible, we must merge.

Move the parent's item down into the siblings, and add the empty internal node's child to the sibling.

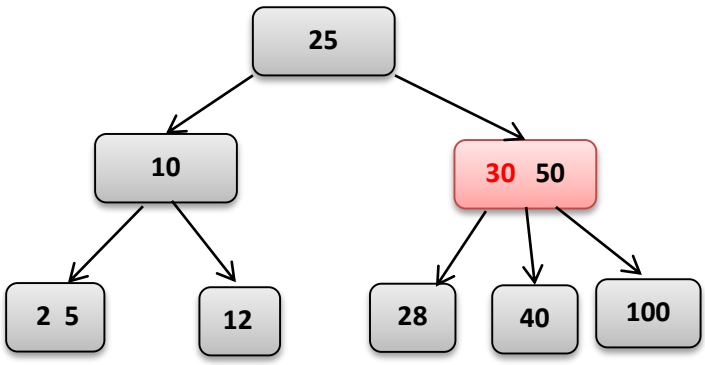
We can now remove the empty internal node.

If the parent ends up empty, it is also an internal node that requires redistribution or merging, this is a recursive property.

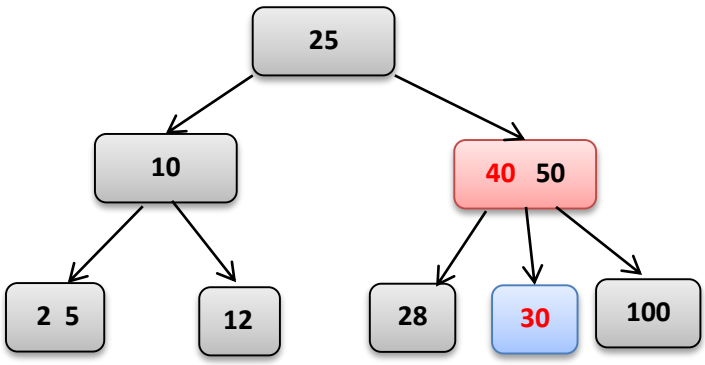
Merging Effects on the Root (Example 3)

If the root ever ends up without an item, we simply delete the root, and the merged child becomes the new root of the tree.

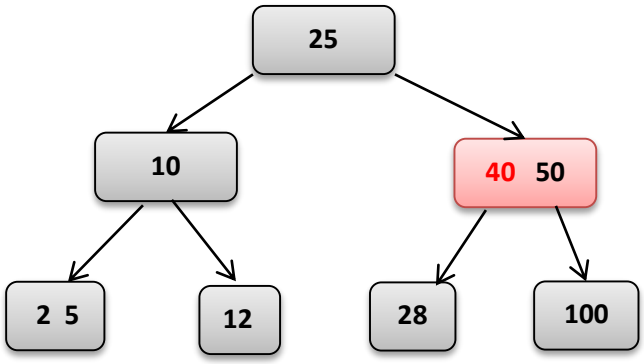
Example 1 – Delete 30



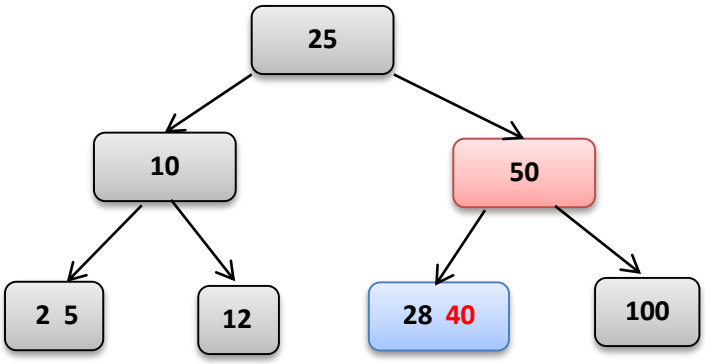
Locate Node



Swap Items

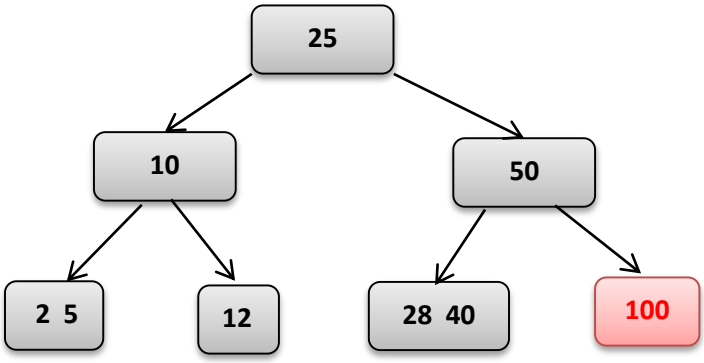


Delete Leaf

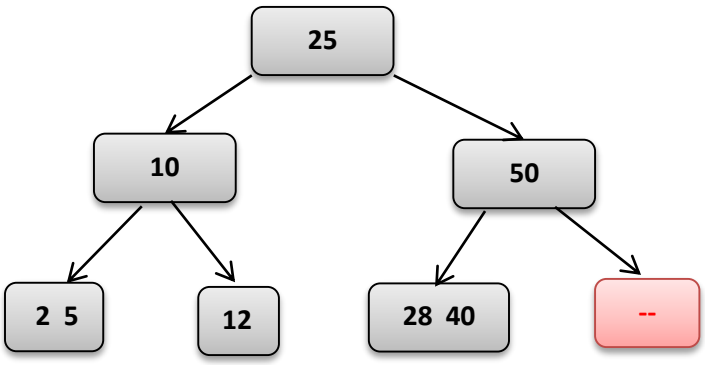


Merge Down

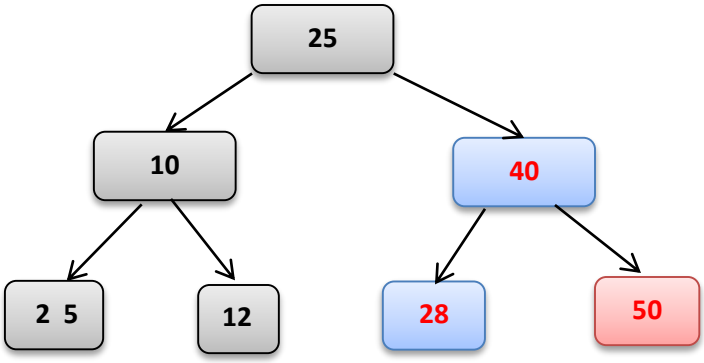
Example 2 – Delete 100



Locate Node

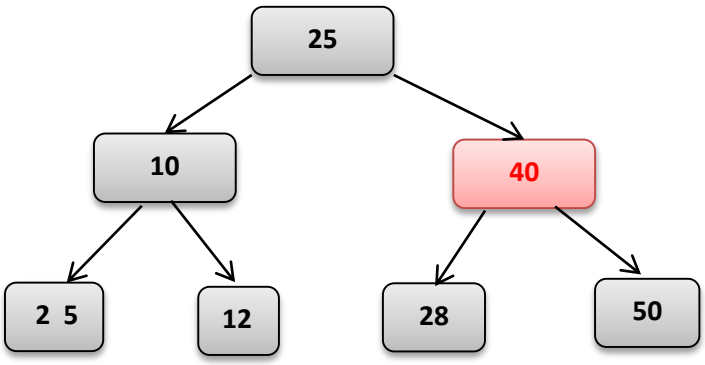


Remove Item

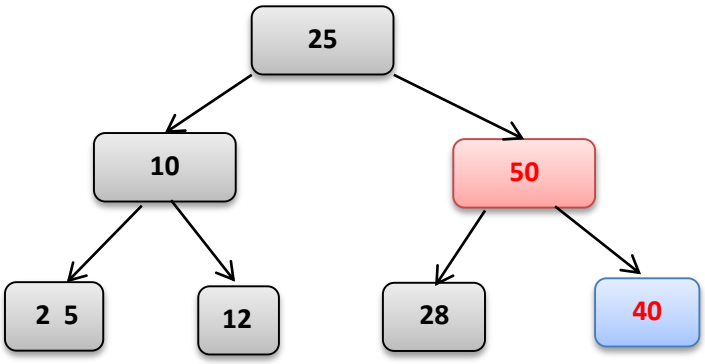


Redistribute

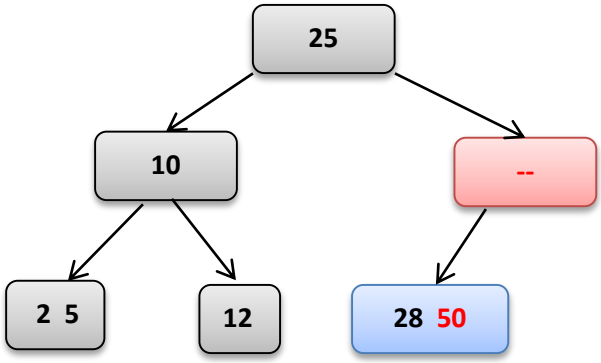
Example 3 – Delete 40



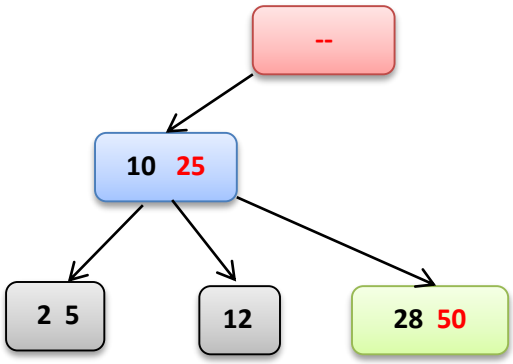
Locate Node



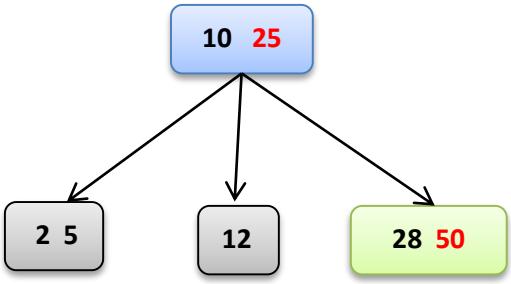
Swap Items



Merge Down



Merge Parent Down, Adopt Child



Delete the Root

Definition of Full:

A tree is full all of it's non-leaf nodes have the maximum amount of children it can have, it is not when the lowest level has the maximum number of nodes, that is a tree that is both complete and full. A B tree (2-3 tree with degree 3) will always be full due to it's balancing nature, it will only split when it needs to, and because of this will always have maximum children.

About Debugging:

At this point, not learning how to use a debugger is a very big disadvantage, in problems like these, it makes finding small errors such as in insertion or deletion much much easier. The era of using print lines to debug a program is pretty much over, especially in a language like Java where every IDE has a competent debugger, you should get comfortable using it. Eclipse's isn't the best I've seen, but I can't recommend IntelliJ's debugger enough. Additionally, using a debugger will help you out a lot when you take Assembly language or other classes which throw you into a language you aren't that strong in, the debugger will allow you to not only see your problems, but understand how the language works.