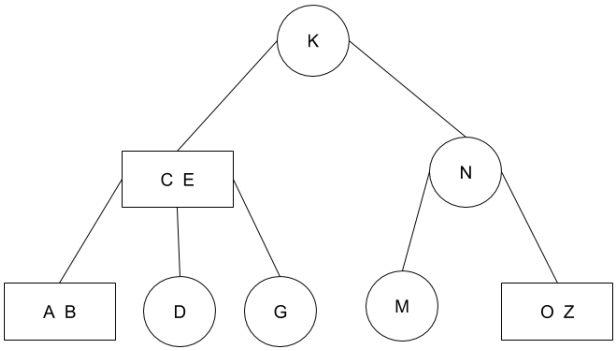
Balanced Search Trees

1 Balanced Search Trees: 2-3 Trees

In order to ensure that we don't get the worst case runtime for a Binary Search Tree, we can attempt to use a category of data structures called ***Balanced Search Trees***- these data structures will always have a logarithmic height regardless of how keys are inserted. The first such data structure we will discuss is called **2-3 Trees**. The difference between this and a standard Binary Search Tree is that we can have "2 nodes" which have 2 elements inside of them and have either 0 or 3 children- "1 nodes" will have either 0 or 2 children. To provide a visual representation, this is how a 2-3 tree looks.

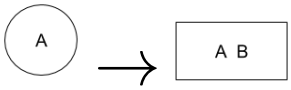


Now that we know what a 2-3 Tree is, how do we go about searching through one? Well, just like the Binary Search Tree, we start at the root; however, instead of immediately traversing the tree, we check to see if the node is a "2 node". If it is a "2 node", we check to see if the element we are searching for is less than the smallest element in the node, greater than the largest, or in between. If the element is less than the smaller element in the node, we traverse the left branch, and if the item is greater than the larger element we traverse the right branch- this process is similar to a Binary Search Tree search. However, if it is the case that the element is between the two elements, we go down the middle path. We recursively do this process until we find the element, or until there is no element to go to (the element is not in the tree).

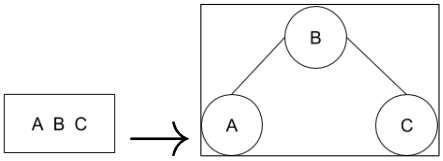
Now that we know how to do search on a 2-3 Tree, let's see if we can do insert. In order to insert an element into a 2-3 tree, we have to recognize a few things: A node can have no more than 2 elements within it and the number of children a node has depends on how many elements are within it. There are a few cases that we need to account for.

- Inserting into a node with 1 element.
- Inserting into a node with 2 elements.
- Inserting into the root.

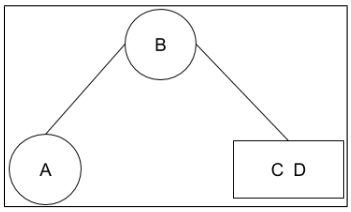
Let's begin to create a 2-3 tree. Following each image will be a brief description of what happened.



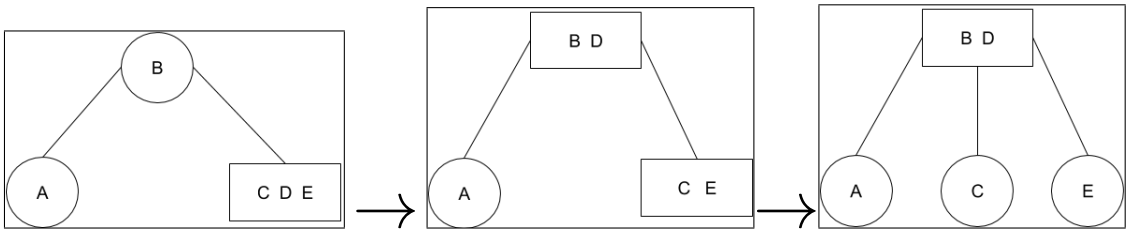
Initially, we created a 2-3 Tree composed just of A. In the next step, we added B. We then made it into a 2-node.



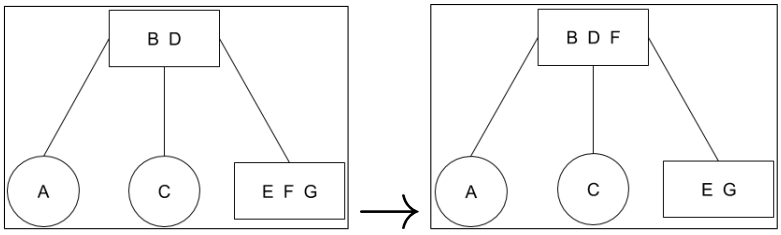
In the first step of this process, we inserted C. However, unlike the previous step, when we insert into this node, we have 3 elements in 1 node. By our definition of 2-3 Trees, we can only have 2 elements max per node. Since there is no parent to move an element to, we will make the middle element the root (in this case B) and make A its left child and C its right child.



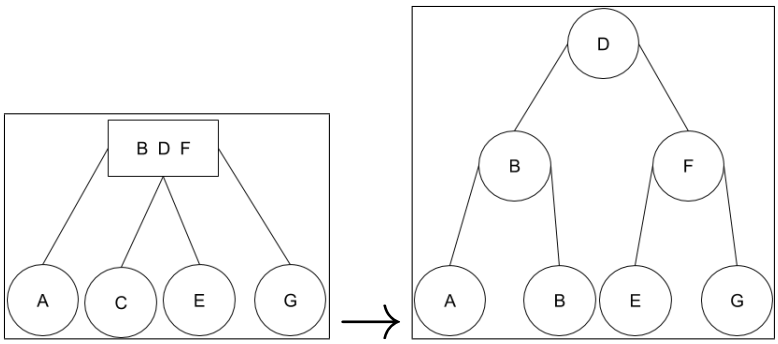
This step is trivially adding D, since it is greater than D, it goes to the right side. Since C is in a normal node, we transform it into a 2-node and add D.



In this step, we added E, once again we end up at the node with C and D. We add E to this node; however, it becomes a node of 3. We solve this by "promoting" the middle child- D. We move D to the root and make the root a 2-node. The new problem that arises is that the node BD only has 2 children. To solve this, we split the 2-node that we promoted D from. We know that both elements in the 2-node that D was promoted from are greater than B as they were to the right of B. We also know that there is one element less than D and one greater than it because D was the middle element of the node. As a result, from this node, we can make one node that is between B and D and one node that is greater than D.



In the initial step, we added F and G (adding F was trivial so I skipped that step). Once again, we end up with a 3-node. To deal with this, we promote F. However, once we promote F we have a 3-node in the root. We'll solve this in the next step.



To solve the problem of getting a 3 node in the root, we first split the 2-node that F was promoted to into 2 separate nodes. Now we have 1 node less than B, 1 between B and D, 1 between D and F, and one greater than F. We finish off by making D the root and the item less than it in the root (B) its left child, and the item greater than it in the root (F) its right child and make the corresponding old children of the 3-node their children.

Just about now, you are probably wondering "is there some sort of universal way to insert into a 2-3 tree? I have compiled a set of "steps" that can be followed in order to insert into a 2-3 Tree.

- No matter what element you are inserting, the item you insert will first go to a leaf.
- If the node that you insert into is a single node, you make it into a 2-node and you are done.
- If the node becomes a 3-node, you promote the middle element.
- You split the children so that the parent has $1 + amountofitemsinthenode$.
- If the node becomes a 2-node after the promotion, you are done, otherwise you promote the middle element again.
- This process keeps repeating unless the node is the root.
- If the root becomes a 3-node, then the middle child becomes the new root, and the item less than it in the node becomes the left child and the item greater becomes the right child.

All of a 2-3 trees operations will run in $\Theta(log(N))$ time in worst case. The reason behind this is that the height of the tree will never exceed $log(N)$.

2 Balanced Search Trees: Red Black Trees

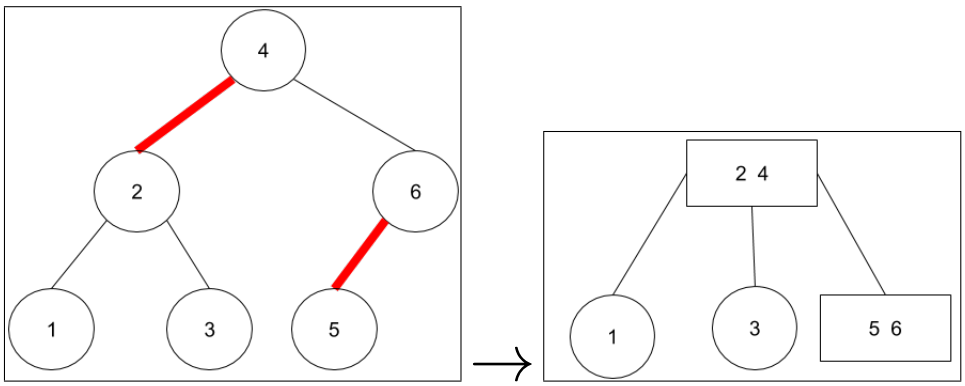
There is also another type of Balanced Search Tree called the ***Red Black Tree***. Red Black Trees are isometric with 2-3 trees, that is there is a red black tree which can be represented as a 2-3 tree. Similar to a 2-3 tree, Red Black trees have a guarantee of $lg(n)$ height, so what is the benefit of it over a 2-3 tree? Well in practice, Red-Black Trees are easier to implement, especially the that we will go over, Left-Leaning Red Black Trees (LLRB). With 2-3 trees, you would need various classes for different types of nodes and would constantly need to modify a node based off insertions and deletions. With LLRB's, you have a standard class that can be used regardless of any conditions. Let's now discuss the structure of red black trees.

Instead of having "nodes" that have more than 1 element, Left Leaning Red Black Trees have 2 distinct kinds of edges. A black edge between two nodes indicates a parent child relationship while a red link shows a "same node" relationship. For a Left Leaning Red Black Tree to be valid, it must follow certain criteria:

- Red Links can only lean left
- No Node has 2 red links connected to it

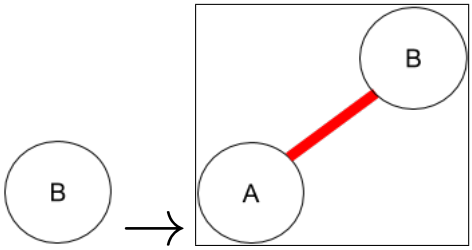
- Every leaf in the tree has the same amount of blank links above it.

Here is an example of an LLRB, and it's equivalent 2-3 Tree:

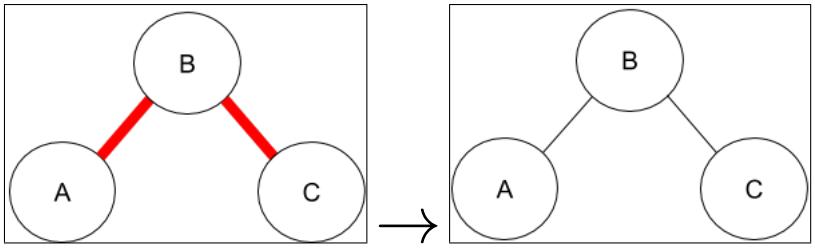


Search is relatively self explanatory in the LLRB, you just look to see if a node is less than the current, if so go left otherwise go right- you continue this until you find the element. Inserting; however, can be a bit tricky.

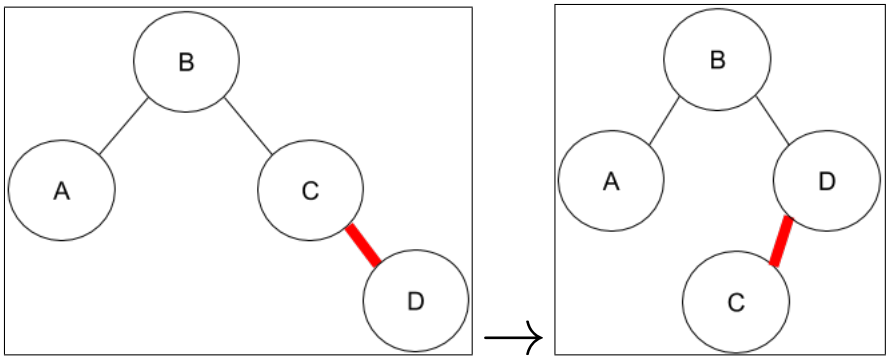
There are 2 broad ways that we can insert into a LLRB- either we can convert the tree into a 2-3 tree, the usual preferred approach as it is simple, or we can do a series of color flips and rotations. The 2-3 tree method is straightforward, so we won't discuss how to do it; however, we will go over the way to insert into an LLRB. Whenever inserting into a LLRB, the node that you insert is inserted with a red link. After the element is inserted, we must make sure that it follows our above rules. We will follow a similar process of creating a red-black tree from scratch in order to show all the possible cases.



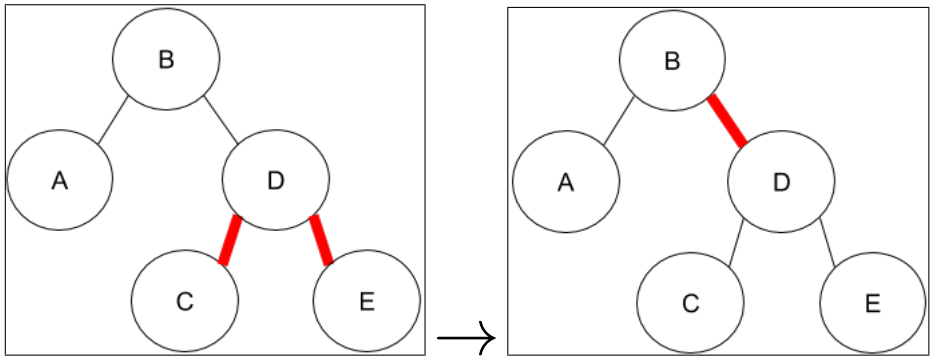
In the above image, we inserted the node B, following this, we inserted A. Whenever we insert into a red black tree, we insert with a red link. Since A is less than B, we make it the left child of B.



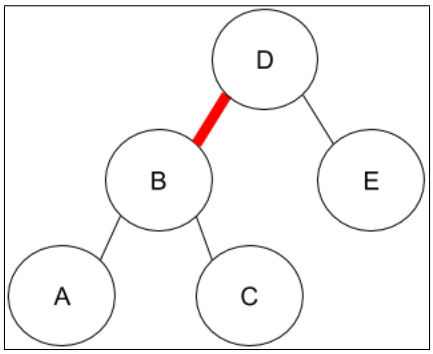
In this example we inserted C, and since it is greater than B, it is inserted as a right child of B. Now the problem is that both of B's children have red links, to circumvent this problem, we "flip the colors" so that only the parent B's parent link is red, and its children links are black. Since B has no parent links, the only color flips are from the links to A and C from B.



In this image, we inserted D; however, now we have a red link "leaning right". Since this is never allowed, we simply rotate the node D to the left. The link between C and D stays as red, and it is now a valid Red Black Tree.



In this step, we inserted E into our LLRB. We now have a similar situation to the step when we inserted C. We do a similar process of color flipping and D's parent link becomes red. Keep in mind that this is a valid trick because the amount of black links will remain the same (If the two children have red links making both links black and turning the parent's red keeps each leaf's amount of black links the same since 2 links are changing per leaf). However, after this we realize that we created a new problem, there is a right leaning link from B to D, we'll fix it now!



In order to fix this problem, we simply rotated the D node to the left. Keep in mind when we do this, that the children less than D, in this case just the node C, must become the right child of the old root. If we want to logic this out it is because C was less than D but greater than B.

I compiled a "cheat sheet" of rules for the rotation of Left Leaning Red Black Trees and they are as follows:

- A node is ALWAYS inserted with a red link.

- When you insert an item, and it causes a problem, take care of the small subproblem, which will only contain the inserted node's parent and potential other child.
- If the item was inserted to the left of the parent, there is no problem.
- If the item was inserted to the right of the parent and the parent does not have a second child, perform a rotation so the link becomes left leaning. Remember to move any children properly to make sure the rotation is valid.
- If the item was inserted to the right of the parent and the parent has a second child, perform a color flip.

The steps for inserting into a red-black tree are much fewer than those for inserting into a 2-3 tree which is partially why the implementation is easier.

We can see from the insertion steps that we did for both 2-3 trees and Red-Black Trees that the height of the tree never gets beyond $log(N)$. The runtime for a red-black tree is the same as the runtime for 2-3 trees at $log(N)$ time, which can be derived from its maximal height