Binary Search Tree Insert

We saw that when we search a binary tree $T$ for a value $x$, the search path is completely deterministic – for each possible value of $x$, we know exactly where it should be. So the basic idea of our insert algorithm is to put the new value in the location where it will be found on a subsequent search. This means that our insert algorithm is basically a modified version of our insert algorithm.

In this discussion we will assume that it is legal to have multiple copies of a value in the BST – thus every insertion will result in the tree gaining a new vertex. When inserting a value $x$, if we find an existing instance of $x$ already in the tree, we just keep going. Thus we will inevitably reach the bottom of the tree and "fall off". The point at which we fall off the tree is the unique correct location for the new leaf containing the new value.

We will examine the iterative version of the algorithm first. In order to attach a new vertex containing the new value, we need to keep track of the existing vertex that will be the parent of the new vertex, and we need to know whether to add the new vertex as the left child or the right child of its parent.

```
BST_Search(T, x):
    if T.root == nil:
        T.root = new BST_Vertex(x)
    else:
        current = T.root
        parent = nil
        side = ""
        while current != nil:
            parent = current
            if current.value < x:
                current = current.right_child
                side = "right"
            else:
                current = current.left_child
                side = "left"
            if side == "left":
                parent.left_child = new BST_Vertex(x)
            else:
                parent.right_child = new BST_Vertex(x)
```
We can eliminate the side variable because all we need to know is whether x is $\leq$ its parent (in which case it goes on the left) or $>$ its parent (in which case it goes on the right). This changes the algorithm to

```python
BST_Search(T, x):
    if T.root == nil:
        T.root = new BST_Vertex(x)
    else:
        current = T.root
        parent = nil
        while current != nil:
            parent = current
            if current.value < x:
                current = current.right_child
            else:
                current = current.left_child
            if x <= parent.value:
                parent.left_child = new BST_Vertex(x)
            else:
                parent.right_child = new BST_Vertex(x)
```

We can also eliminate the parent variable – it’s not clear that this is worth the trouble, but here is one way to do it:
BST_Search(T,x):
    if T.root == nil:
        T.root = new BST_Vertex(x)
    else:
        current = T.root
        Not_Done = True
        while Not_Done:
            parent = current
            if current.value < x:
                if current.right_child == nil:
                    current.right_child = new BST_Vertex(x)
                    Not_Done = False
                else:
                    current = current.right_child
            else:
                if current.left_child == nil:
                    current.left_child = new BST_Vertex(x)
                    Not_Done = False
                else:
                    current = current.left_child

As you might expect, I wouldn't use any of these ... I prefer a recursive version. I find it cleaner and easier to understand. As usual, I will present the algorithm in a form of a “wrapper” function that effectively hides the details of the implementation, and a recursive function that actually does all the work.

My philosophy on algorithms that modify a tree structure is that they should always return a pointer to the root of the modified tree. This allows us to treat trees and their subtrees in exactly the same way. I'll expand on this a bit after we look at the algorithm.
BST_Insert(T,x):
    T.root = rec_BST_Insert(T.root,x)

rec_BST_Insert(current,x):
    if current == nil:
        return new BST_Vertex(x)
    else:
        if current.value < x:
            current.right_child = rec_BST_Insert(current.right_child, x)
        else:
            current.left_child = rec_BST_Insert(current.left_child, x)
        return current

You might want to trace through the operation of this on a small set of values to see that it does what it should. One of the key things to note is that – as described above – the recursive function does not need to make a special case for dealing with an empty tree: when it reaches the point of insertion it creates the new vertex and returns it. The proper attachment of the vertex is managed by the next level up in the recursion sequence.

Note how this algorithm follows the philosophy stated above. If the new value belongs in the right subtree of the current vertex, we recurse down into that subtree. Whatever is the result of that recursive call, we attach it as the right child of the current vertex. Most of the time this will be exactly the same vertex as it was before the recursive call so this may seem like a wasted operation, but this is more than made up for by the elimination of extraneous variables and extra “if-then” statements.

And now ... deleting from a Binary Search Tree!
BST Deletion

Deleting a value from a Binary Search Tree is a bit more complicated than inserting a value, but we will deal with the steps one at a time.

First, we need to find the value – which is easy because we can just use the method we developed for BST_Search.

Once the value is found, we have a problem. When we delete an element from a linked list, we just make its predecessor point to its successor to fix the list. But with a tree, if we delete a vertex v we may have two children to reconnect, and only one link from v’s parent to connect to them.

One option would be to simply move values around in the tree so that the structure remains more or less intact. Since we are deleting a value we will eventually need to delete a vertex, but possibly we could move values around in the tree so that the vertex we need to delete is a leaf – which would be simple because it has no children to re-connect.

There are two potential problems with this idea. One is that each vertex may contain a massive amount of data – copying the data from one vertex to another might be time-consuming. Another, more significant problem is that it is possible that in order to restore a valid arrangement of values in the tree, we might have to move a lot of them.

Our goal is to make as few changes as possible to the tree, so that our algorithm will run quickly. The standard approach is to replace the vertex we are removing with another existing vertex. The vertex we use as a replacement needs to contain a value that is ≥ all values in the left subtree and > all values in the right subtree. One candidate is the vertex that contains the largest value in the left subtree.

We will delete this vertex from the left subtree and “plug it in” to replace the vertex v (the one we are trying to delete). But doesn’t this just present us with another “delete a vertex” problem? Yes, but it turns out that this problem is very easy to solve.

Consider this BST. Note the partial edge coming in to the vertex containing 18. This indicates that something points to this vertex. It may be a parent vertex, or it may be the root pointer of the tree. Our goal here is to delete the “18”.
To delete the 18, we find the largest value in its left subtree – in this case, that is 17 (we will talk about how to find this a bit later). If we delete the vertex containing 18 and pull the vertex containing 17 out of its current location, the pieces of the tree look like this:
I have moved the 17 up in the diagram – of course there is no actual movement of the information in memory – that is what we are trying to avoid. Now we need to put these pieces back together. But that is easy! Whatever it was that used to point to 18 should now point to 17. 17’s left_child and right_child pointers should now point to the subtrees that used to be 18’s left and right children. Whatever it was that used to point to 17 (in this example, 11) should now have 17’s old left child as its right child. The result is this:
Now that may have seemed like a bunch of ad hoc fixes ... but it turns out that we do exactly the same things every time (with one special case that we will deal with later).

Here’s that same “fixed” tree, but I have drawn a line around the new “top” vertex and its left subtree:
Note that the material within the dotted line consists entirely of vertices that were in the left subtree of the vertex we originally deleted (the 18). We can describe what we have done to that subtree very simply: we modified it so that its largest value was at the root. If we treat that modification as a function (I will call it “fixing the left subtree”) then our method of deleting 18 gets even easier to express:

let p be the thing that points to 18
let tmp be a pointer to the root of the subtree that results from fixing 18’s left subtree
make tmp.right_child point to 18’s right subtree
make p point to tmp

How do we know that tmp.right_child isn’t already in use? Because when we fix the left subtree, we move the largest value to the top ... so its right subtree will be empty!
Alarm bells ringing? They might be. What happens if 18’s left subtree is empty? We obviously can’t pull out the largest value in an empty subtree. Fortunately this special case is extremely easy to resolve:

\[
\begin{align*}
18 & \quad \quad \quad 30 \\
30 & \quad \quad \quad 21 \quad 86
\end{align*}
\]

simply becomes

\[
\begin{align*}
30 & \quad \quad \quad 21 \quad 86
\end{align*}
\]

which we can express as

\[
\begin{align*}
\text{let } p \text{ be the thing that points to } 18 \\
\text{make } p \text{ point to } 18\text{'s right subtree}
\end{align*}
\]

Similarly, if 18 has no right child, we can simply make p point to 18’s left child.
Now we can put all the pieces together. We can do it iteratively or recursively ... guess which one I am going to choose.

BST_Delete(T, x):
    T.root = rec_BST_Delete(T.root, x)

rec_BST_Delete(current, x):
    if current == nil:
        return current  # takes care of case where x is not present in T
    else:
        if current.value < x:
            current.right_child =
            rec_BST_delete(current.right_child, x)
            return current
        else if current.value > x:
            current.left_child =
            rec_BST_delete(current.left_child, x)
            return current
        else:
            if current.left_child == nil:
                return current.right_child
            else if current.right_child == nil:
                return current.left_child
            else:
                tmp = fix_left_subtree(current)
                tmp.right_child = current.right_child
                return tmp

And that’s it. At each stage of the search we enter the appropriate subtree, and then use whatever comes back as the new subtree on that side. When we find the value to delete, we fix its left subtree and reattach its right subtree, then return the root of the rebuilt subtree (which automatically gets properly attached at the next level up).

Except that is not quite it … we haven’t looked at the problem of fixing the left subtree. There are two cases to consider:
- the largest value in the left subtree is at the root of the subtree, or
- it isn’t

If the largest value in the left subtree is already at the root of the subtree, we don’t have to do anything (we know its right_child pointer is nil) so we just return it. How do we recognize that the largest value in the left subtree is at the root? By checking its right_child pointer!

If the largest value in the left subtree is not at the root of the subtree, it must be in the root’s right subtree. We step down to the root’s right child. If this is the largest value in the subtree, it will have no right child, and conversely if it has a right child then it is not the largest value. Extending this logic we can see that to find the largest value we can simply continue stepping
down to the right until we reach a vertex with no right child. This is the vertex we will move
to the top of this subtree. In its new position, its left child will be the original root of the
subtree and its right child will be nil. Down where it came from, its left child needs to be
reattached ... which it can be, as the right child of the old parent of the vertex we are moving
up.

So given this as the left subtree:

```
  7
 /   \
4   11
 /  \
10 17
   /\  \
  13
 /\  \  \
12 15
```

we see that the root has a right child so we step down to the right until we can’t go any
further (at the 17). We make 17’s left_child pointer point to the original root (7) and we
make the vertex we looked at just prior to 17 (17’s parent, which is 11) point to 17’s left
child (13).

This is simple coding:
```python
def fix_left_subtree(v):
    temp = v.left_child  # temp is the root of v’s left subtree
    if temp.right_child == nil:
        return temp  # no fix needed
    else:
        parent = nil
        current = temp
        while current.right_child != nil:
            parent = current
            current = current.right_child
        parent.right_child = current.left_child
        current.left_child = temp
        return current
```

And now, at long last, we are really done with the deletion algorithm.

**What was the Point?**

Why did we go through the painful exercise of working out the precise details of the insert and delete operations on binary search trees? There are several reasons:

- It’s really good exercise for our brains
- It deepens our understanding of the BST data structure
- It strengthens our coding chops
- It is a good warm up for what comes next (Red-Black Trees)
- It gives us a basis for discussing the computational complexity of these operations

Let’s focus on the last of these. Our whole reason for looking at Binary Search Trees was to provide a better alternative to a sorted array when the required operations are Search, Insert and Delete. What we have seen is that there are algorithms for these operations that first find the appropriate location in the tree (two locations, for Delete) and then do a small sequence of actions that take constant time. Furthermore, we saw that “finding the appropriate location” consisted of making comparisons, and that each time we made a comparison we either recognized that we were at the proper location, or we moved down to a specific vertex, one level down in the tree. None of the algorithms required us to back-track and go down a different branch of the tree than we were already in.
Thus for each of these algorithms, the maximum number of iterations or recursive calls is bounded by the height of the tree. In other words, if T has height h, then each of our algorithms runs in $O(h)$ time.

The problem is that the height of a BST with $n$ values can be $n-1$. This means that our algorithms have a worst-case complexity of $O(n)$, which is no better than an array. So it seems that all of our work has been for naught.

But don’t despair! Starting on Monday we will begin our study of Red-Black trees, a cleverly designed type of BST that solves this problem.