Binary Search Trees - Part 10

Robin Dawes

February 17, 2021

Reveals that in Binary Search Trees as in life, getting rid of something we don't is often harder than adding something new.

.

Deleting a Value from a BST

Deleting a value from a set stored in 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 a value from an array, we can move all the following values one space towards the beginning of the array to close up the gap. 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 value x, the vertex containing x may have two children to reconnect, and there is 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 value we want to delete is in a leaf! But in order to maintain a valid arrangement of values in the tree, we might have to move a lot of them - effectively rebuilding the tree.

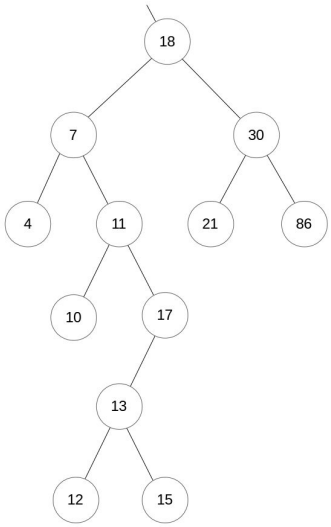
Our goal is to find a fast algorithm that will remove the undesired value (and the vertex that contains it) and maintain the essential properties of the Binary Search Tree, while making as few changes as possible to the tree.

The standard approach is to replace the vertex we are removing with another existing vertex, and attach the children of the deleted vertex to its replacement. Of course that requires finding a suitable vertex. The vertex we use as a replacement needs to contain a value that is $\geq$ all values in the left subtree and $<$ all values in the right subtree.

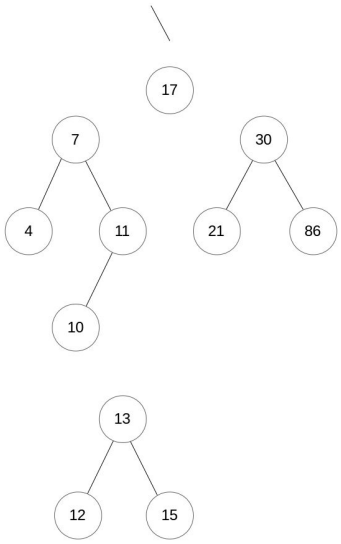
One candidate is the vertex that contains the largest value in the left subtree.

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".



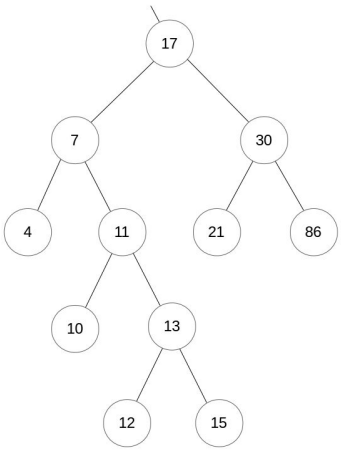
To delete the 18, we find the largest value in its left subtree - in this case, that is 17. (We will talk about how to locate this value a bit later.) If we delete the vertex containing 18 and pull the vertex containing 17 out of its current location, the pieces of the tree look like this. I have moved 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!

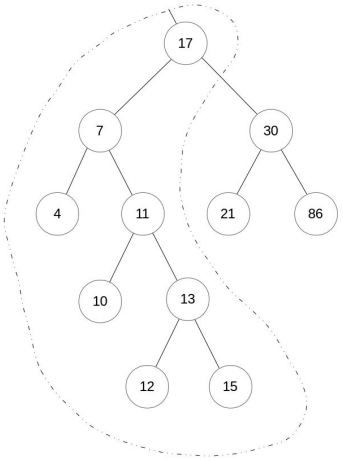
1. Whatever it was that used to point to 18 should now point to 17.
2. 17's left_child and right_child pointers should now point to the subtrees that used to be 18's left and right children.
3. Whatever it was that used to point to 17 (in this example, 11) should now have 17's old left child as its right child.

The result is shown at the right.



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:



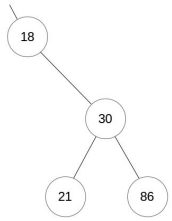
Note that the material within the dotted line consists entirely of vertices that were in the left subtree of the vertex we originally deleted (the 18). We can describe what we have done to that subtree very simply: we modified it so that its largest value was at the root. If we treat that modification as a function (I will call it "fixing the left subtree") then our method of deleting 18 gets even easier to express:

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

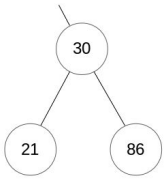
How do we know that tmp.right_child isn't already in use? We know that because when we fix the left subtree, we move the largest value to the top ... so the right subtree of the vertex holding this value will be empty!

... because its right subtree could only contain values that are larger, and we have chosen the largest value in this subtree.

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:



simply becomes:



which we can express algorithmically as

1. let p be the thing that points to 18
2. make p point to 18's right subtree

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 I am going to choose! As usual, we will assume this is an instance method within our Binary_Search_Tree class. The algorithm recursively searches for the desired value, then starts applying the fixes we have just developed.

```
def Delete(x):
    this.root = rec_Delete(this.root,x)

def rec_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_delete(current.right_child, x)
        return current
    else if current.value > x:
        current.left_child = rec_delete(current.left_child, x)
        return current
    else:                     # found it!
        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 re-attach its right subtree, then return the root of the rebuilt subtree (which automatically gets properly attached at the next level up).

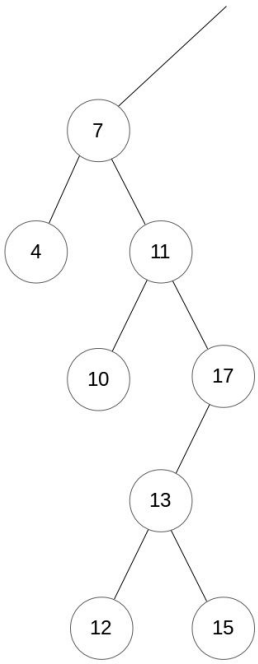
Except that is not quite it ... we haven't looked at the problem of fixing the left subtree. There are two cases to consider:

- the largest value in the left subtree is at the root of the subtree
- 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 (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! The **right_child** pointer of the subtree's root is nil if and only if the largest value is at the root.

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 previous parent of the vertex we are moving up.



So given this as the left subtree:

we see that the root (7) 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:

```
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.

Was That Worth the Pain?

WHY DID WE GO through the agonizing 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 takes 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 lower 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 a BST with n values can potentially have n levels. This happens when each vertex has only one child. 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! Now we will begin our study of Red-Black trees, a cleverly designed type of BST that solves this problem.