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.

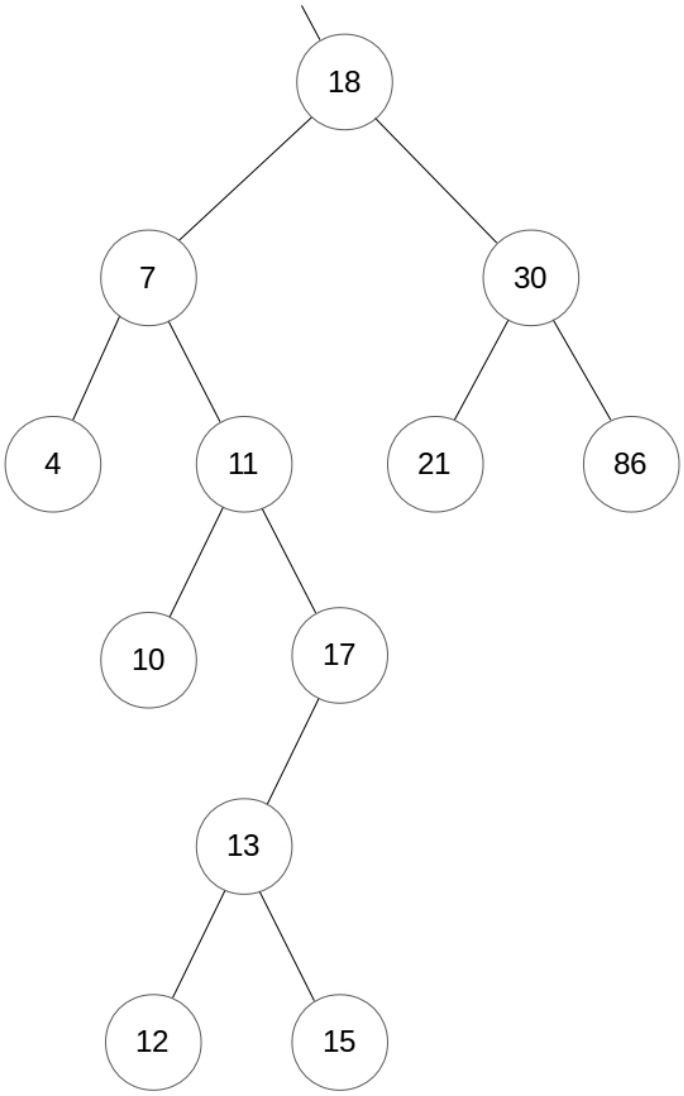
One option would be to simply move values around in the tree so that the structure remains more or less intact. Since we are deleting a value we will eventually need to delete a vertex, but possibly we could move values around in the tree so that the vertex we need to delete is a leaf – which would be 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 perhaps in order to restore a valid arrangement of values in the tree, we might have to move a lot of them.

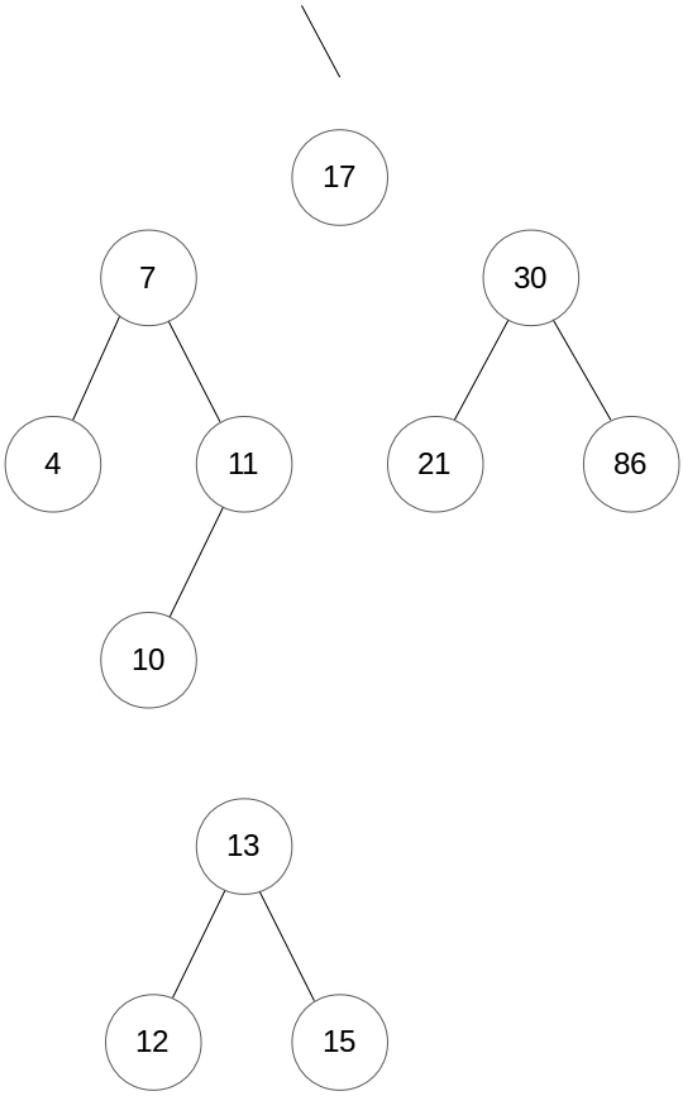
Our goal is to make as few changes as possible to the tree, so that our algorithm will run quickly. The standard approach is to replace the vertex we are removing with another existing vertex. The vertex we use as a replacement needs to contain a value that is $\geq$ all values in the left subtree and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

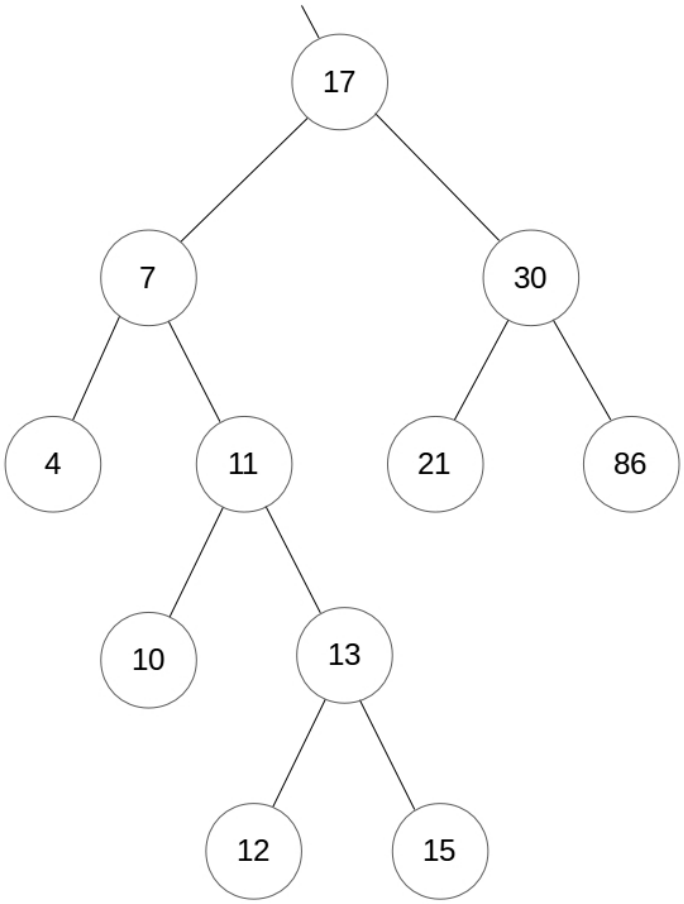
Consider this BST. Note the partial edge coming in to the vertex containing 18. This indicates that *something* points to this vertex. It may be a parent vertex, or it may be the root pointer of the tree. Our goal here is to delete the "18".



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 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:

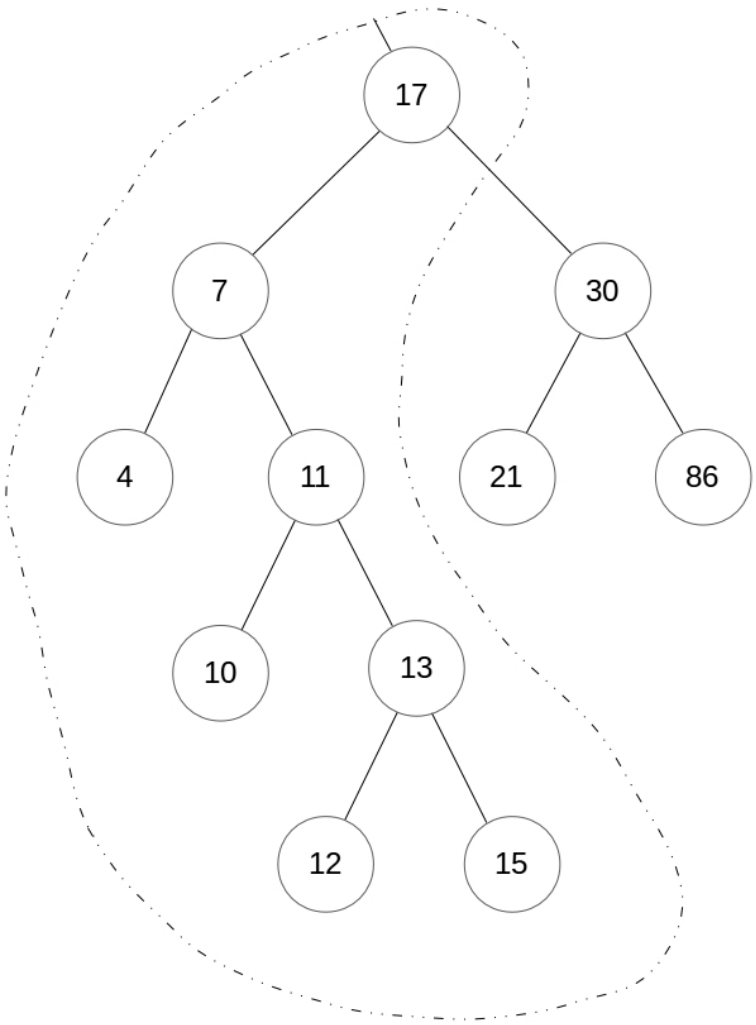


I have moved the 17 up in the diagram – of course there is no actual movement of the information in memory – that is what we are trying to avoid. Now we need to put these pieces back together. But that is easy! Whatever it was that used to point to 18 should now point to 17. 17's left_child and right_child 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:

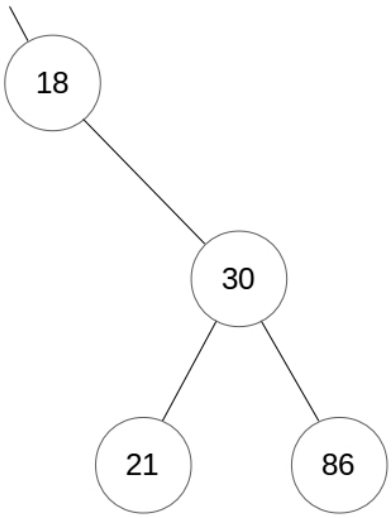


Note that the material within the dotted line consists entirely of vertices that were in the left subtree of the vertex we originally deleted (the 18). We can describe what we have done to that subtree very simply: we modified it so that its largest value was at the root. If we treat that modification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

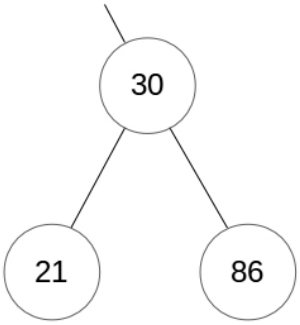
```
let p be the thing that points to 18
let tmp be a pointer to the root of the subtree that results
     from fixing 18's left subtree
make tmp.right_child point to 18's right subtree
make p point to tmp
```

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:



simply becomes



which we can express as

```
let p be the thing that points to 18
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

```
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:

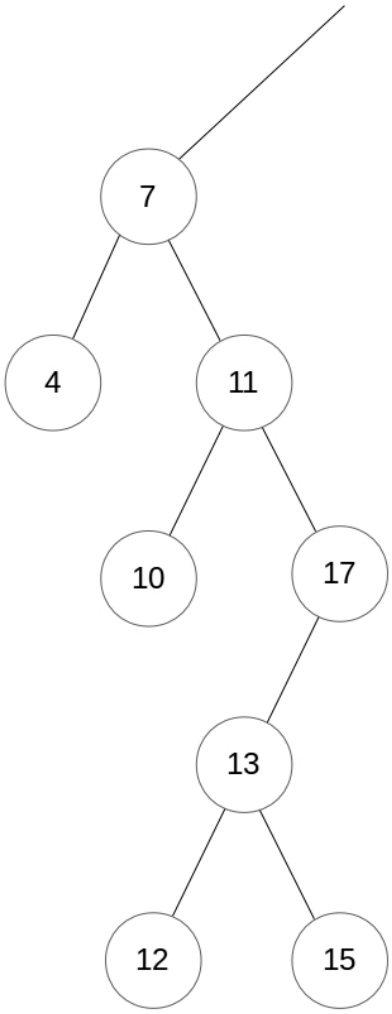
- the largest value in the left subtree is at the root of the subtree, or
- it isn't

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! 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 old 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.

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 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.

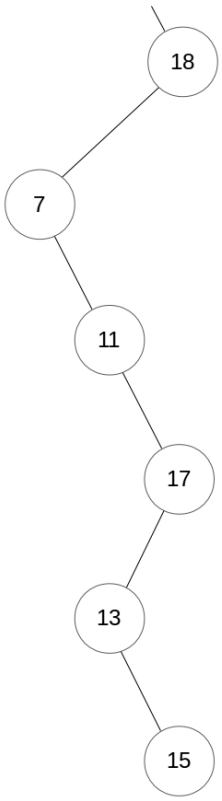
Thus for each of these algorithms, the maximum number of iterations or recursive calls is bounded by the height of the tree. In other words, if T has height h, then each of our algorithms runs in O(h) time.

The problem is that a BST with n values can have n levels. 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.

Red-Black Trees

As we have seen, the ideal Binary Search Tree has height approximately equal to log n, where n is the number of values stored in the tree. Such a BST guarantees that the maximum time for searching, inserting and deleting values is always $\in O(\log n)$. However, if we have no control over the order in which the values are added and/or deleted, the BST may end up looking like a linked list, with each vertex having just one child. In this case the maximum time for searching, inserting and deleting values is in O(n), which is no better (and in the case of searching, much worse) than the time required for these operations if we store the set in a sorted array.



If the values are inserted in the order 18, 7, 11, 17, 13, 15 the binary tree ends up looking like a linked list: each vertex has only one successor. The number of levels is equal to the number of values in the set.

In order to claim that BSTs are better than sorted arrays, we need to find a way to always attain that desirable O(log n) time for the three operations. One way would be to rebuild the tree from scratch after each insertion or deletion ... but that would be a lot of work.

A better option would be to establish a limit on the number of levels – for example, we might choose 2*log n. Then whenever the tree exceeds this number of levels, we could rebuild the tree from scratch and make it as compact as possible.

This is a very interesting idea. The "rebuild from scratch" operation is time-consuming (although not **too** bad – we can rebuild the tree in "perfect" form in O(n) time … I strongly recommend figuring out how!) but we probably don't do it very often. With luck, after rebuilding the tree we could do a lot of inserts before we had to rebuild again. Thus the average amount of extra work we do for each insert would be quite small. This would be a reasonable solution if we don't mind a significant delay, once in a while.

What we will see now is that it is possible to take another approach: keep the number of levels small by doing a little bit of extra work fairly often.

In the 1960's, people started to use the term "**balanced**" to describe trees where each vertex has the property that its left subtree and right subtree are "about the same height" ... of course "about the same height" can be interpreted in different ways.

Red-Black trees were invented in 1972 in an effort to create a binary search tree structure that maintains O(log n) height while requiring relatively few re-organizations of the tree. In a Red-Black tree, the idea of balance is "at each vertex, neither subtree is more than twice as tall as the other". For your own interest you may want to read about AVL trees, which have similar properties but a much stricter balance rule: at each vertex, the two subtrees must have heights that differ by no more than 1.

A Red-Black tree is a binary search tree in which each vertex is coloured either Red or Black. In practice all that is required is a single bit to indicate if the vertex is Red or Black, but for learning purposes we can imagine that the vertices are physically painted.

Red-Black trees are usually described as obeying the following rules :

1. All vertices are coloured Red or Black
2. The root is Black
3. All leaves are Black, and contain no data (ie data values are only stored in internal vertices)
4. Every Red vertex has 2 children, both of which are Black
5. At each vertex, all paths leading down to leaves contain the **same number** of Black vertices

… and we will pick up at that point in our next class!