By Julienne Walker

Andersson trees are simple and easy to implement balanced binary search trees that are based on the foundations of red black trees. Consequently, Andersson trees have similar performance and structuring properties as red black trees without the difficult implementation. Red black trees are an abstraction of the symmetric binary B-tree, which is a clever abstraction of a B-tree of order 4. Andersson trees are a simplification of the symmetric binary B-tree that use a B-tree of order 3 to remove several unpleasant cases in the implementation. If the last two sentences meant absolutely nothing to you, that's okay. This background isn't necessary to understand Andersson trees or implement them well. Andersson trees were introduced by Arne Andersson in his paper "Balanced Search Trees Made Simple". They were further studied by a few people, most notably Mark Allen Weiss, who discussed them briefly in his books on data structures and algorithm analysis.

This tutorial will cover everything that they did and more. Personally, I found Arne's paper to be somewhat incomplete and Mark's descriptions to add very little (and nothing that could not be easily gleaned from Arne's paper). So the intention of this tutorial is to cover Andersson trees more thoroughly and with practical implementations in mind. I will introduce the same implementation that both Andersson and Weiss described as well as several other implementations that I believe simplify the algorithms by using more conventional methods. I will also introduce a non-recursive implementation which has sadly been missing from descriptions and implementations of the Andersson tree.

Necessity

The current sad state of affairs in the world today is that efficient algorithms and data structures are not used in production code. I often see the dreaded bubble sort in real world applications, when the only acceptable place for bubble sort is as an introduction to sorting that should be forgotten shortly after understanding how it works. I also see basic binary search trees occasionally, if the author of the code was "working with performance in mind". Of course, in every case the expected input included degenerate cases, and a balanced binary search tree would have been a better choice if one really had performance in mind. This is one of the better cases. More often I see linked lists and arrays where balanced trees would have been far superior.

I blame two things for this problem. First, the overly clever armchair computer scientists with a PhD in mathematics and only a vague idea that not every programmer is as brilliant as they are. These people devise beautiful works of art that are theoretically optimal in both time and space, but in practice are virtually impossible to understand for most of the world's programmers. In practice, optimal time and space is nice, but often not worth the cost of initial implementation and subsequent maintenance, not to mention subtle bugs. I further blame these people because in their papers and reports on their creations, they invariably seem to forget that operations such as deletion are neither so trivial nor so unimportant as to be an "exercise for the reader". Especially when deletion is almost always the most complicated and difficult operation by far.

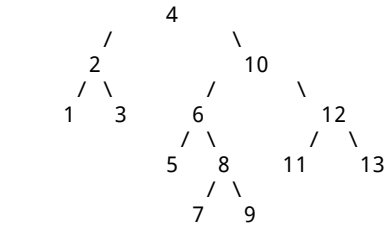
Second, I blame teachers for failing to emphasize the practical need for good algorithms and data structures. I also blame them for showing off their intelligence instead of stripping away complexity and bringing the concepts down to a level that their students will be able to follow more easily. If a CS student doesn't understand that a data structure is important and doesn't understand the concepts of the data structure, that student is highly unlikely to use the data structure out in the real world. As a result, we see sub-par applications that could

be improved drastically with nothing more than a better choice of algorithms and data structures.

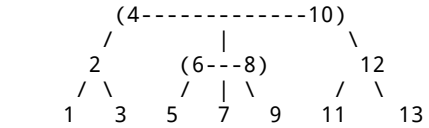
Andersson trees bring balanced binary search trees down to the masses by taking a complicated and difficult data structure and peeling back the complexity until the only thing left is an easy to understand and easy to implement balanced tree. Well, enough of my ranting. Let's take a look at the concepts.

Concept

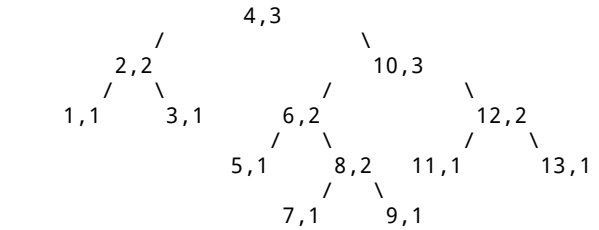
A balanced binary search tree is easy to understand, even if the traditional algorithms to implement one are not. A balanced binary search tree is simply a binary search tree where every path is never much shorter the longest path. In other words, a balanced tree guarantees that every path is logarithmic. The most common balanced trees ensure this by keeping balance information in every node, such as the difference in height of the left and right subtrees. Another common abstraction is to view one or more nodes as pseudo-nodes, thereby using a binary search tree to fake a multi-way search tree. This is the concept behind red-black trees. Removing the red black abstraction and simply looking at nodes as pseudo-nodes with two, three, or four links, you can then consider "red" links as being horizontal and "black" links as being vertical. Horizontal links create a single pseudo-node and vertical links connect two pseudo-nodes. Consider the following binary search tree:



By converting 4 and 10 into a single node, and 6 and 8 into a single node, you have the following 3-way tree:



To view this 3-way tree as a binary search tree of pseudo-nodes, simply use a flag or some other method to signify that a link is horizontal. For example, by supplying each node with a level, where leaves are level 1, two nodes that are linked together with the same level represent a single pseudo-node:



Because 4 and 10 have the same level and are directly linked, they are viewed as a single node even though a binary search tree does not allow more than a single item and two links per node. This pseudo-node has two items and three links. 6 and 12 also have the same level, but they are not directly linked together, so they are not viewed as a single pseudo-node. The three trees shown above are

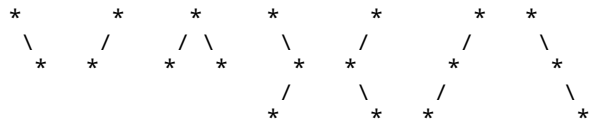
Andersson trees because there are no horizontal left links. Let's look at the rules for an Andersson tree:

- 1) Every path contains the same number of pseudo-nodes.
- 2) A left child may not have the same level as its parent.
- 3) Two right children with the same level as the parent are not allowed.

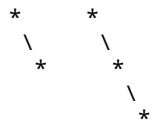
The last two rules can be written in with more abstraction because it is possible, though more complicated, to implement an Andersson tree without using levels as the balance information:

- 2) Horizontal left links are not allowed.
- 3) Two consecutive horizontal links are not allowed.

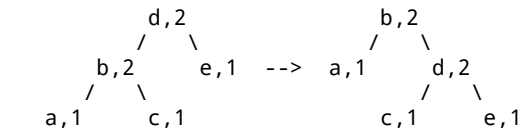
This concept is different from a red black tree which is based on the same idea because a red black tree allows left horizontal left links. The result is that an Andersson tree fakes a B-tree of order 3 and a red black tree fakes a B-tree of order 4. The good news is that an Andersson tree is vastly simpler because if left horizontal links are allowed, there are manny different patterns to look for. With a red black tree, the following shapes must be considered to maintain balance, resulting in several complicated cases:



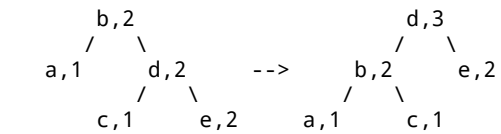
With an Andersson tree, those seven shapes are reduced to two, and the cases required to maintain balance are both small in number and simple:



To maintain balance in an Andersson tree, you need to test the levels and make sure that none of the rules are broken. Because there are only two cases to test for, they can be broken into two operations called skew and split. A skew removes left horizontal links by rotating right at the parent. No changes are needed to the levels after a skew because the operation simply turns a left horizontal link into a right horizontal link:



Unfortunately, a skew could create two consecutive right horizontal links. A split removes consecutive horizontal links by rotating left and increasing the level of the parent. A split needs to change the level of a single node because if a skew is made first, a split will negate the changes made by doing the inverse of a skew. Therefore, a proper split will force the new parent to a higher level:



For any rebalancing operation in an Andersson tree, you only need to follow the right path of the tree and skew, then do the same thing and split. This implies that skew and split need to be implemented recursively, which is both doable and trivial if you know how to perform a rotation:

```
struct jsw_node *skew ( struct jsw_node *root )
{
  if ( root->level != 0 ) {
    if ( root->link[0]->level == root->level ) {
      struct jsw_node *save = root;
      root = root->link[0];
      save->link[0] = root->link[1];
      root->link[1] = save;
    }

    root->link[1] = skew( root->link[1] );
  }

  return root;
}

struct jsw_node *split ( struct jsw_node *root )
{
  if ( root->link[1]->link[1]->level == root->level && root->level != 0 )
    struct jsw_node *save = root;
    root = root->link[1];
    save->link[1] = root->link[0];
    root->link[0] = save;
    ++root->level;
    root->link[1] = split ( root->link[1] );
  }

  return root;
}
```

It might seem at first glance that a recursive skew and split all the way down the tree would be inefficient, but notice that both skew and split are conditional. Structural changes are only made if the balancing rules are violated. Each rotation serves to bring the tree more into balance, so it makes sense that rotations will not be made as often as it originally appears. In practice, for insertions an Andersson tree will only make a few more rotations than a red black tree, but for deletions an Andersson tree will probably make about 25% more rotations than a red black tree. Both of these are often acceptable when you consider that even though more rotations are made, the algorithms are simpler and faster than red black algorithms. So it all evens out rather nicely.

Skew and split can also be implemented non-recursively as standalone operations. This requires more work in the deletion algorithm because both skew and split need to be called more than once for each node. This is a slightly more efficient implementation because only one function call is made at a time rather than stacking them up with recursion. The implementation is identical to the recursive implementation except without the recursive calls:

```
struct jsw_node *skew ( struct jsw_node *root )
{
  if ( root->link[0]->level == root->level && root->level != 0 ) {
    struct jsw_node *save = root->link[0];
    root->link[0] = save->link[1];
    save->link[1] = root;
    root = save;
  }

  return root;
}

struct jsw_node *split ( struct jsw_node *root )
{
  if ( root->link[1]->link[1]->level == root->level && root->level != 0 )
    struct jsw_node *save = root->link[1];
    root->link[1] = save->link[0];
    save->link[0] = root;
    root = save;
    ++root->level;
```

```
21   }
22
23   return root;
24 }
```

A third alternative is to write skew and split inline and avoid function calls altogether. My Andersson tree library uses this approach by implementing skew and split as macros. However, this is strictly an implementation optimization, and the rest of this tutorial will conveniently ignore the inline approach in favor of easier to understand solutions.

```
 1 #define skew(t) do {                                          \
 2   if ( t->link[0]->level == t->level && root->level != 0 ) { \
 3     struct jsw_node *save = t->link[0];                       \
 4     t->link[0] = save->link[1];                               \
 5     save->link[1] = t;                                        \
 6     t = save;                                                 \
 7   }                                                           \
 8 } while(0)
 9
10 #define split(t) do {                                                \
11   if ( t->link[1]->link[1]->level == t->level && root->level != 0 ) { \
12     struct jsw_node *save = t->link[1];                               \
13     t->link[1] = save->link[0];                                       \
14     save->link[0] = t;                                                \
15     t = save;                                                         \
16     ++t->level;                                                       \
17   }                                                                   \
18 } while(0)
```

Insertion

To simplify the insertion and deletion algorithms, Andersson trees typically use a sentinel node to terminate the nodes rather than null pointers. This way there is no need to test for a null pointer and the sentinel can be given a level lower than all other levels. All in all it makes for simpler algorithms, so I recommend it and will assume a sentinel in every implementation for this tutorial. The sentinel will simply be a global pointer to a node called nil, and must be fully initialized before any other node in the tree:

```
 1 struct jsw_node {
 2   int data;
 3   int level;
 4   struct jsw_node *link[2];
 5 };
 6
 7 struct jsw_node *nil;
 8
 9 int jsw_init ( void )
10 {
11   nil = malloc ( sizeof *nil );
12
13   if ( nil == NULL )
14     return 0;
15
16   nil->level = 0;
17   nil->link[0] = nil->link[1] = nil;
18
19   return 1;
20 }
```

Since leaf nodes have a level of 1, nil will have a level of 0 so that it does not adversely affect how restructuring algorithms work. The sentinel must also point to itself for the left and right links. That way we can walk as much as we want beyond a leaf and still have predictable behavior.

Also used will be a function to create and initialize a single node. Technically this isn't needed since that code can be written inline, but it makes the algorithms shorter and seemingly simpler by factoring out unnecessary clutter:

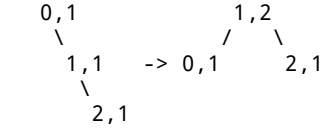
```
 1 struct jsw_node *make_node ( int data, int level )
 2 {
 3   struct jsw_node *rn = malloc ( sizeof *rn );
 4
 5   if ( rn == NULL )
 6     return NULL;
 7
 8   rn->data = data;
 9   rn->level = level;
10   rn->link[0] = rn->link[1] = nil;
11
12   return rn;
13 }
```

With these preliminaries out of the way, we can look at the recursive insertion algorithm described by Andersson and Weiss. The bad news is that it's somewhat anticlimactic. The good news is that it's somewhat anticlimactic, because the algorithm is insertion for a basic binary search tree with skew and split called on the return path. It's bad news because we're talking about a balanced tree, and everyone expects long and smart looking code for a balanced tree. It's good news because this is about as simple as it gets for a balanced tree, and looking smart gets old after you have to implement and debug it a few times:

```
 1 struct jsw_node *jsw_insert ( struct jsw_node *root, int data )
 2 {
 3   if ( root == nil )
 4     root = make_node ( data, 1 );
 5   else {
 6     int dir = root->data < data;
 7     root->link[dir] = jsw_insert ( root->link[dir], data );
 8     root = skew ( root );
 9     root = split ( root );
10   }
11
12   return root;
13 }
```

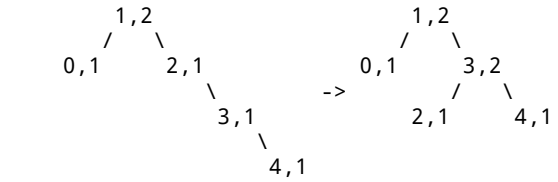
A new node is always placed at the bottom of the tree, so it will always be given a level of 1. Remember that nil has a level of 0, so skew and split won't be affected at the leaves because 0 is only one less than 1 (but you already knew that). Notice that the return value of skew and split is being assigned to root, which suggests that I'm using one of the functions rather than the macro. The macro would not require assignment of a "return value". Also notice that either the recursive or the non-recursive version of skew and split would work equally well here. The difference is only noticeable in the deletion algorithm.

Let's walk through the building of an Andersson tree. First we will insert 0 into an empty tree. This hits the first case immediately and a new node is inserted with a level of 1. Then 1 is inserted. On the way back up, neither skew nor split will do anything because there is only one horizontal link, and it's a right link. Then we insert 2 and the fun begins. Once again, skew does nothing because we only have right horizontal links, but because there are two right horizontal links in a row, split must perform a left rotation and increase the level of 1:

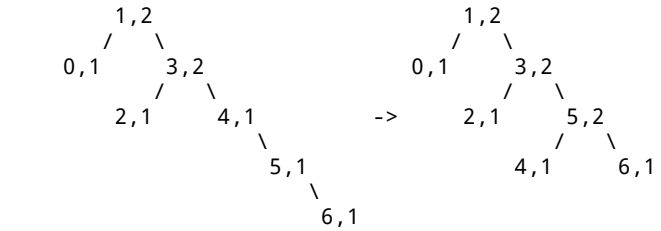


Now 3 is added and nothing is done because none of the rules are broken. 1 no longer has a right horizontal link, but now 2 does have one, so the next insertion

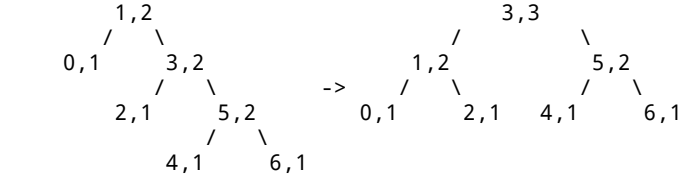
of 4 will violate the rule for consecutive right horizontal links and must be split:



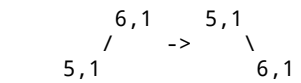
Adding 5 does nothing, but 6 will violate the same rule (notice a trend for ascending value insertion?) and a split is required. But this time a single split will not be enough because splitting 4, 5, and 6 will cause 5 to be increased to level 2, resulting in consecutive horizontal links at 1, 3, and 5:



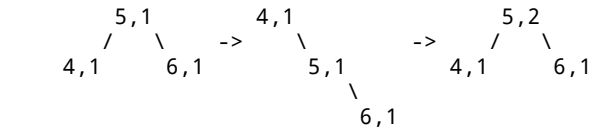
So the rebalancing must propagate up the tree and split at 1:



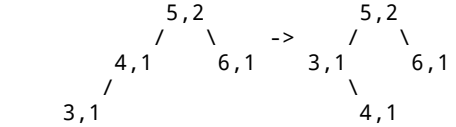
In this example there was never a skew because we were only inserting values in ascending order. Ascending order is a degenerate case, and as you can see, the end result was quite balanced. However, let's look at another degenerate case that will force a skew, descending order insertion. We start with adding 6 to an empty tree, and nothing happens, then 5 is inserted. At this point we have a left horizontal link because both 6 and 5 have a level of 1, so a skew is required to turn the left horizontal link into a right horizontal link:



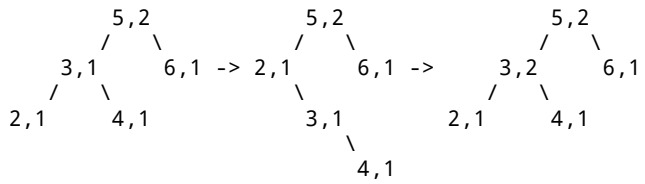
Now we insert 4, which creates another horizontal left link, so another skew is required. But the skew creates consecutive right horizontal links, so a split needs to fix that. This is why skew is called before split, because a skew can create a case that requires a split:



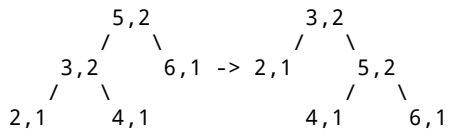
This is an interesting case because it seems to unnecessarily rotate twice. Can you think of a way to implement an Andersson tree where this case could simply increase the level of 5 and avoid the two rotations? Moving on, we add 3 and skew, but no split is needed:



When we add 2, however, both a skew and a split is required:



This is still not enough because now 5 has a left horizontal link. So the rebalancing must propagate and skew again to restore balance:



A lot of people try to avoid recursion, for many good reasons. The good news is that an Andersson tree is really no harder to implement iteratively than a basic binary search tree. Here is the code for non-recursive insertion into an Andersson tree:

```
 1 struct jsw_node *jsw_insert ( struct jsw_node *root, int data )
 2 {
 3   if ( root == nil )
 4     root = make_node ( data, 1 );
 5   else {
 6     struct jsw_node *it = root;
 7     struct jsw_node *up[32];
 8     int top = 0, dir = 0;
 9
10     for ( ; ; ) {
11       up[top++] = it;
12       dir = it->data < data;
13
14       if ( it->link[dir] == nil )
15         break;
16
17       it = it->link[dir];
18     }
19
20     it->link[dir] = make_node ( data, 1 );
21
22     while ( --top >= 0 ) {
23       if ( top != 0 )
24         dir = up[top - 1]->link[1] == up[top];
25
26       up[top] = skew ( up[top] );
27       up[top] = split ( up[top] );
28
29       if ( top != 0 )
30         up[top - 1]->link[dir] = up[top];
31       else
32         root = up[top];
33     }
34   }
35
36   return root;
37 }
```

This implementation simulates the down and up movement of recursion by saving the path down the tree on a stack and then popping nodes off of the stack and rebalancing until the stack is empty. Care must be taken to ensure that the parent nodes properly reset their children if a rotation is made, and that the root of the tree is updated at the bottom of the stack. Aside from these subtle issues, the code is straightforward and surprisingly simple for a balanced tree.

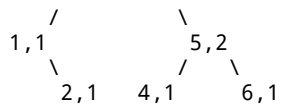
Deletion

Removing a node from an Andersson tree is, not surprisingly, more difficult than insertion. Fortunately, it's not much more difficult than deletion from a basic binary search tree. One of the most important adjustments that needs to be made is to decrease the level of a node is there is a break in the values. For example, if a node has a level of 4 then if its child has a lower level, that level must be 3. Then a recursive skew and split is performed. Let's look at my recursive code for deletion, and then step through an example:

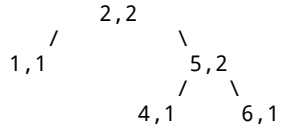
```
 1 struct jsw_node *jsw_remove ( struct jsw_node *root, int data )
 2 {
 3   if ( root != nil ) {
 4     if ( data == root->data ) {
 5       if ( root->link[0] != nil && root->link[1] != nil ) {
 6         struct jsw_node *heir = root->link[0];
 7
 8         while ( heir->link[1] != nil )
 9           heir = heir->link[1];
10
11         root->data = heir->data;
12         root->link[0] = jsw_remove ( root->link[0], root->data );
13       }
14       else
15         root = root->link[root->link[0] == nil];
16     }
17     else {
18       int dir = root->data < data;
19       root->link[dir] = jsw_remove ( root->link[dir], data );
20     }
21   }
22
23   if ( root->link[0]->level < root->level - 1
24     || root->link[1]->level < root->level - 1 )
25   {
26     if ( root->link[1]->level > --root->level )
27       root->link[1]->level = root->level;
28
29     root = skew ( root );
30     root = split ( root );
31   }
32
33   return root;
34 }
```

This is identical to recursive deletion from a basic binary search tree with the extra test for a break in the levels. If there's no break then there was no deletion and no rebalancing needs to be done, which is why skew and split are inside the conditional test along with the actual code to decrease a level. Let's look at a step by step example of this algorithm on the final tree of the ascending insertion example:

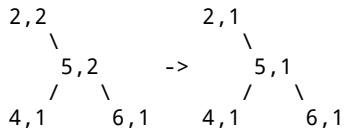
Removing 0 will cause a break in the levels between 1 and nil, so the level of 1 is decreased to 1. Then the break is between 1 and 3, so the level of 3 is decreased to 2. This is a simple case because no skews or splits are needed to restore balance:



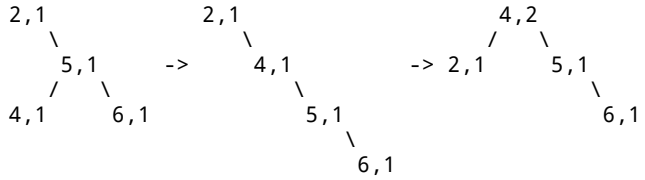
Now we will remove 3:



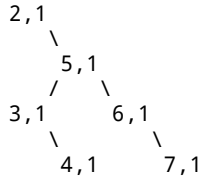
3 is replaced with its inorder predecessor 2, and the node for 2 is replaced with nil. In this case, no rebalancing is required at all because there is no break in the levels and the tree remains balanced. However, when we remove 1, this creates a break between 2 and nil. The level of 2 is decreased to 1 along with the level of 5:



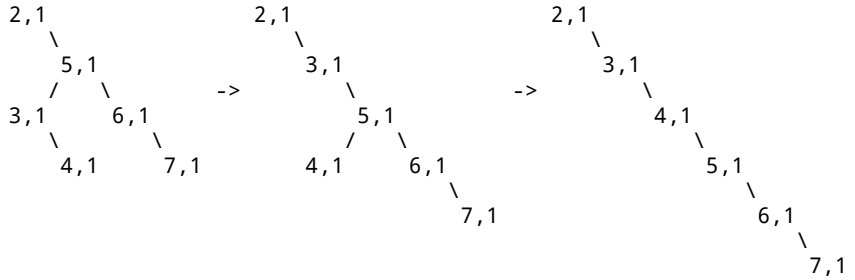
This is a somewhat nasty case that shows why skew and split need to be called recursively. If skew is not recursive, the violation at 5 would not be fixed. But since skew and split are recursive, everything works out nicely:



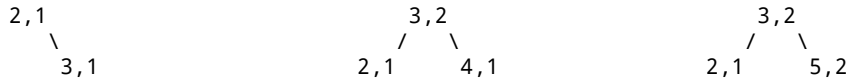
In reality, only 3 skews down the tree are needed at most, and only 2 splits. Why? Because the worst possible imbalance after a deletion would be the following structure (only relevant nodes shown):

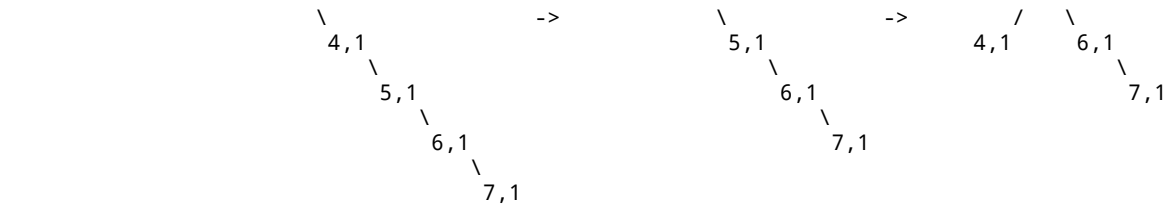


To fix this, a skew is made at 2, then at 5, and at 5 again:



This result is just a tad unbalanced, but two splits will fix it. The first split is at 2, then the second is at 4:





With all of this in mind, the non-recursive versions of skew and split can be used as long as three calls to skew and two calls to split are made:

```
 1 struct jsw_node *jsw_remove ( struct jsw_node *root, int data )
 2 {
 3   if ( root != nil ) {
 4     if ( data == root->data ) {
 5       if ( root->link[0] != nil && root->link[1] != nil ) {
 6         struct jsw_node *heir = root->link[0];
 7
 8         while ( heir->link[1] != nil )
 9           heir = heir->link[1];
10
11         root->data = heir->data;
12         root->link[0] = jsw_remove ( root->link[0], root->data );
13       }
14       else
15         root = root->link[root->link[0] == nil];
16     }
17     else {
18       int dir = root->data < data;
19       root->link[dir] = jsw_remove ( root->link[dir], data );
20     }
21   }
22
23   if ( root->link[0]->level < root->level - 1
24     || root->link[1]->level < root->level - 1 )
25   {
26     if ( root->link[1]->level > --root->level )
27       root->link[1]->level = root->level;
28
29     root = skew ( root );
30     root->link[1] = skew ( root->link[1] );
31     root->link[1]->link[1] = skew ( root->link[1]->link[1] );
32     root = split ( root );
33     root->link[1] = split ( root->link[1] );
34   }
35
36   return root;
37 }
```

All of this is fun, and in my opinion this recursive version is easy to understand, but both Andersson and Weiss described a more efficient variation of the deletion algorithm. Based on earlier work by Andersson, a minimal comparison tree search is used rather than the usual three way comparison. The trick is to use two variables that point to the heir whose value will replace the deleted item, and the item to be replaced. The code is simple, but unconventional enough to be confusing:

```
 1 struct jsw_node *jsw_remove ( struct jsw_node *root, int data )
 2 {
 3   static struct jsw_node *item, *heir;
 4
 5   /* Search down the tree */
 6   if ( root != nil ) {
 7     int dir = root->data < data;
 8
 9     heir = root;
10     if ( dir == 0 )
11       item = root;
```

```
12
13      root->link[dir] = jsw_remove ( root->link[dir], data );
14    }
15
16    if ( root == heir ) {
17      /* At the bottom, remove */
18      if ( item != nil && item->data == data ) {
19        item->data = heir->data;
20        item = nil;
21        root = root->link[1];
22      }
23    }
24    else {
25      /* Not at the bottom, rebalance */
26      if ( root->link[0]->level < root->level - 1
27        || root->link[1]->level < root->level - 1 )
28      {
29        if ( root->link[1]->level > --root->level )
30          root->link[1]->level = root->level;
31
32        root = skew ( root );
33        root->link[1] = skew ( root->link[1] );
34        root->link[1]->link[1] = skew ( root->link[1]->link[1] );
35        root = split ( root );
36        root->link[1] = split ( root->link[1] );
37      }
38    }
39
40    return root;
41  }
```

When we think of efficiency, recursion typically doesn't come to mind. Of course, in a practical implementation it really depends on what is needed, but I always like to have both recursive and non-recursive options when working with binary search trees. So I came up with a quick non-recursive version of deletion. Amazingly enough, it corresponds to deletion in a basic binary search tree, with only a small amount of balancing and bookkeeping code added:

```
 1  struct jsw_node *jsw_remove ( struct jsw_node *root, int data )
 2  {
 3    if ( root != nil ) {
 4      struct jsw_node *it = root;
 5      struct jsw_node *up[32];
 6      int top = 0, dir = 0;
 7
 8      for ( ; ; ) {
 9        up[top++] = it;
10
11        if ( it == nil )
12          return root;
13        else if ( data == it->data )
14          break;
15
16        dir = it->data < data;
17        it = it->link[dir];
18      }
19
20      if ( it->link[0] == nil || it->link[1] == nil ) {
21        int dir2 = it->link[0] == nil;
22
23        if ( --top != 0 )
24          up[top - 1]->link[dir] = it->link[dir2];
25        else
26
27          root = it->link[1];
28      }
29      else {
30        struct jsw_node *heir = it->link[1];
31        struct jsw_node *prev = it;
```

```
32
33        while ( heir->link[0] != nil ) {
34          up[top++] = prev = heir;
35          heir = heir->link[0];
36        }
37
38        it->data = heir->data;
39        prev->link[prev == it] = heir->link[1];
40      }
41
42      while ( --top >= 0 ) {
43        if ( top != 0 )
44          dir = up[top - 1]->link[1] == up[top];
45
46        if ( up[top]->link[0]->level < up[top]->level - 1
47          || up[top]->link[1]->level < up[top]->level - 1 )
48        {
49          if ( up[top]->link[1]->level > --up[top]->level )
50            up[top]->link[1]->level = up[top]->level;
51
52          up[top] = skew ( up[top] );
53          up[top]->link[1] = skew ( up[top]->link[1] );
54          up[top]->link[1]->link[1] = skew ( up[top]->link[1]->link[1] );
55          up[top] = split ( up[top] );
56          up[top]->link[1] = split ( up[top]->link[1] );
57        }
58
59        if ( top != 0 )
60          up[top - 1]->link[dir] = up[top];
61        else
62          root = up[top];
63      }
64    }
65
66    return root;
67  }
```

Conclusion

Andersson trees are a very simple alternative to the more traditional balanced binary search trees. The performance properties are very close to that of red black trees, and the effort required in implementing them is minimal for anyone who is comfortable writing basic binary search trees. This tutorial discussed the background and concepts of Andersson trees, walked through several examples of how the rebalancing operations work, and looked at several variations of working code.

From the twisted mind of Julienne Walker

Top