15 - AVL trees

An AVL tree is a balanced binary search tree; because it is balanced binary search is a $O(\log n)$ operation. It is named for G. M. Adelson-Velskii and E. M. Landis who invented it in 1962. AVL trees augment ordinary binary trees in two ways, first: every node has an extra piece of information called the balance factor and second: there are algorithms, called rotations, to rearrange the tree so that it is still a binary search tree but one where every node has a balance factor of one, zero or -1, corresponding to near perfect balance. The new node definition is given in Table 1 and the new constructor is given in Table 2.

The *height* of a tree is the maximum number of nodes from the root to a leaf travelling down the pointers. This is illustrated in Fig. 1. The *balance* of a node is a measure of the difference in height of the two subtrees the node has, the left subtree and the right subtree. Specifically, if height(**a_node**) gives the height of a node called **a_node** then

$$\textbf{a\_node}->\text{balance} = \text{height}(\ \textbf{a\_node}->\text{left}) - \text{height}(\ \textbf{a\_node}->\text{right}) \qquad (1)$$

In practise the balance factors will be updated as nodes are added, rather than calculated by explicitly computing heights; this will be looked at later, for now all that's important is that the balance is the difference in heights. A tree with the balance factors written in is given in Fig. 2. A tree is called *balanced* if all the balance factors are one, zero or -1; this makes sense since it is impossible to have a tree with all nodes having balance zero unless the number of nodes is $2^r - 1$, for some r. It should be remembered that the height is what is important for binary search and you can see from Fig. 3 that balanced trees don't always have the same number of nodes on each side.

Now, imagine adding a node causes one of its ancestors to have a balance factor of two or minus two. If the tree was balanced before the new node was added this is the worst case: adding a node changes balance factors by one or minus one at the most. Now to fix the lack of balance a rotation is used. There are actually four different rotations corresponding to four classes of cases, all cases fall into one of the four classes. We will examine two rotations here, it will then be clear the remaining two rotations are mirror images.

The two cases we look at are ones where the ancestor, which we will call **here**, has balance +2 and the two different rotations correspond to

$$\textbf{here}->\text{left}->\text{balance} = 1 \qquad (2)$$

and

$$\textbf{here}->\text{left}->\text{balance} = -1 \qquad (3)$$

```
1 struct node
2 {
3   int entry;
4   int balance;
5   struct node *left;
6   struct node *right;
7 };
```

Table 1: A node, it has a variable to store the entry and pointers to the left and right children. It also has a new int to keep track of how balanced the node is.

```
1
2 struct node * make_node(int new_entry)
3 {
4   struct node * a_node=(struct node *)malloc(sizeof(struct node));
5   a_node->entry=new_entry;
6   a_node->balance=0;
7
8   return a_node;
9 }
```

Table 2: Making a node, the new thing is that the balance is initialized to zero.

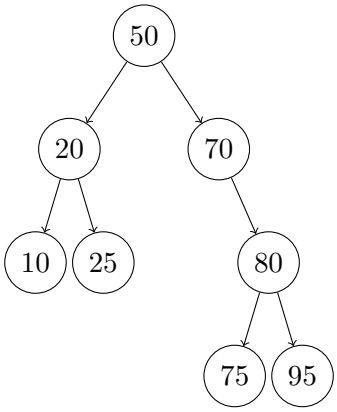


Figure 1: An example binary tree for looking at heights. The node 50 has height four, the distance down to 75 and 95. The node 20 has height two and the node 70 has height three.

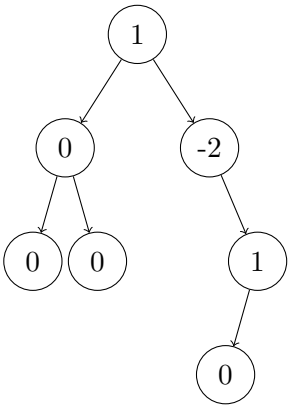


Figure 2: An example binary tree with the balance factors shown.

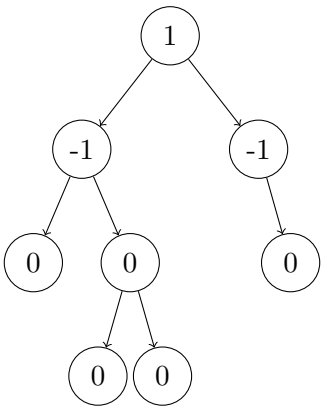


Figure 3: An example of a balanced tree that may not look very balanced. The numbers are the balance factors.

In the first case the disbalance is on the left of the left subtree of **here**. This is the left-left case and the corresponding rotation is called a left-left, or LL, rotation. In the second case it is on the right of the left subtree of here and the corresponding rotation is called a left-right, or LR, rotation. As a notational aside, the LL rotation is sometimes just called the left rotation since it looks more like a single rotation whereas the LR rotation looks like a double rotation.

Lets examine the LL rotation first. The tree at Fig. 4 is in need of such a rotation. The node **here**− >left is called **left**. The first step is to remove the subtree rooted in **left** from **here** and attach **left**− >right in its place. This is shown in Fig.5. Next, the subtree rooted in **here** is attached as the right subtree of **left**. The pointer to **here** would then be updated to point to **left**, the new root. This is shown in Fig. 6.

The same set of instructions would have applied in the simpler looking case Fig. 7, the only difference is that some of the pointers in this case are NULL rather than pointing to nodes. The rotation would also fix the more complicated case Fig. 8 in which some of the nodes in the case we looked at have been replaced by subtrees. Again the same instructions in terms of pointers to nodes and where they get moved to will work. Code to do the LL rotation is given in Table 3.

Now, we consider the LR case. The tree at Fig. 9 is in need of a LR rotation. Again, the disbalanced ancestor is called **here** and the left node of this ancestor is called **left**. It turns out we also need to keep track of **left**− >right and this node is called **left_right**. The new node has been added to **left_right**, the new node could be the left or the right node of **left_right**, in the figure both possibilities are shown, the left dashed and the right dotted, only one of these two will happen at one time.

Now, **left_right** is snapped off. One of its two nodes is the new node. If it is the left node this is attached at the right of **left**, where **left_right** used to be, that is the case illustrated in the figure as the dashed case in Fig. 10. Next, the subtree rooted in **left** is snapped off and attached at the right of **left_right**. If the new node was to the right of **left_right** this is attached to the left of **here**. This is the dotted case in the figure, which for step 2 is Fig. 11. Finally, the subtree rooted in **here** is attached to the right of **left_right** and the pointer that pointed to here is set pointing to **left_right**. This is shown in Fig. 12.

As with the LL rotation, the example we looked at is standing in for a whole class of

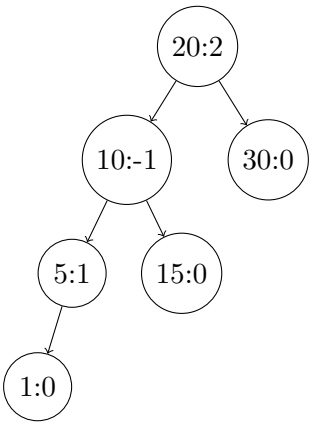


Figure 4: A tree in need of a LL rotation. Example data values are given before the colon, the balance factor after. The 20 node is the ancestor, called **here** in the text and the 1 node is the new node, the one that has just been added.

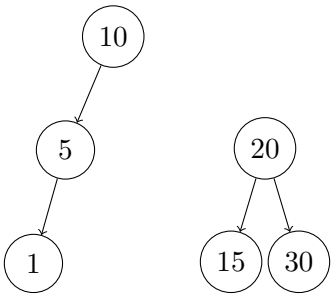


Figure 5: LL step 1. The left subtree has been snapped off and replaced by what was the right subtree of **left**.

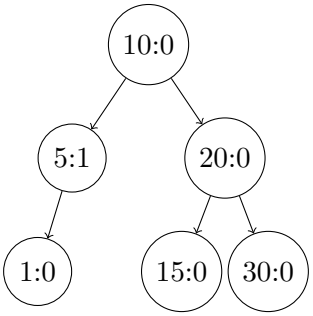
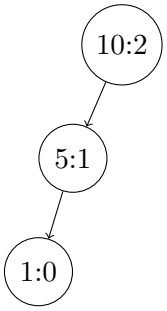


Figure 6: LL step 2. The **here** subtree has been attached as the right subtree of **left**. The updated balance factors are given after the colons.

A



B

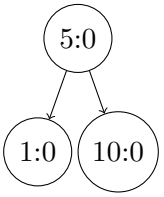
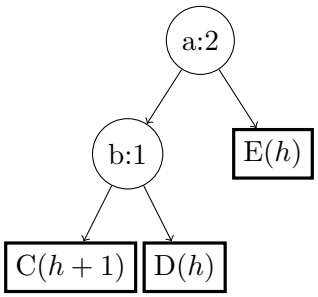


Figure 7: Another LL rotation. **A** represents before and **B** after. Here the 5 node has no right node, so the right pointer of 10 remains set to NULL, but following the same instructions gives the rotation.

```
1  struct node * rotate_ll(struct node * here)
2  {
3    struct node * left = here->left;
4    here->left=left->right;
5    left->right=here;
6    here->balance=0;
7    here=left;
8    here->balance=0;
9
10   return here;
11 }
```

Table 3: The LL rotation. In line 3 a new pointer is made to keep track of **here**− >left. This is now snapped off and **here**− >left set to point at **left**− >right instead in line 4, **left**− >right is then set pointing to **here** in line 5. **left** is now the root, so in line 7 the **here** pointer is set to **left**.

A



B

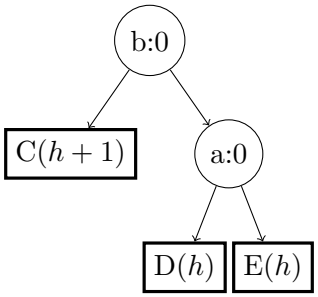


Figure 8: **A**: A tree in need of a LL rotation. Circles correspond to node and rectangles to whole subtrees, the number in brackets in the rectangles gives the height of the corresponding subtree, h is some integer. The subtree at C is has height one greater than the subtree at D and the whole subtree rooted in B is has height two greater than the subtree at E. **B** shows the same tree after the LL rotation.

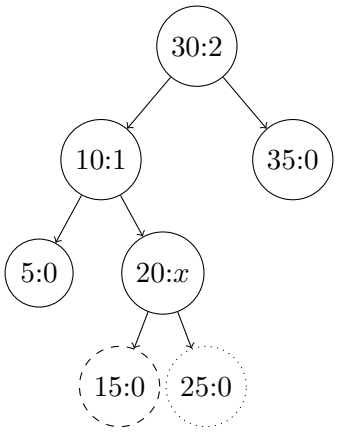


Figure 9: A tree in need of a LR rotation. The 30 node is the one called **here**, the 10 node is **left** and the 20 node is **left_right**. The new node is 15 or 25, the 15 is shown dashed and the 25 dotted since only one of 15 and 25 can be present, the other is NULL. The balance of 20 depends on which case we are dealing with, for the case where the new node is 15, the dashed case, $x = 1$, for the dotted case, $x = -1$.

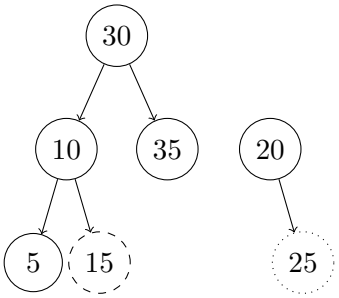


Figure 10: The first step of LR rotation. The 20 node is **left_right**, it has been snapped off the tree. If the new node is the 15 node it takes the place of the **left_right** node on the right of **left**. If the new node is the 25 node, that is attached on the left of **here**.

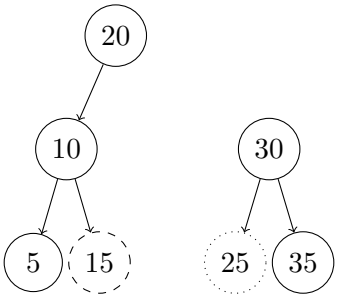


Figure 11: The second step of LR rotation. The 10 node, which is called **left** has been snapped off the tree and attached to the left of **left_right** instead.

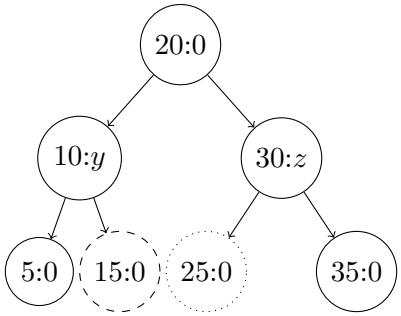


Figure 12: The third step of LR rotation. **here** is attached to the right of **left_right** and balance is restored. The balance factors y and z depend on whether this is the dotted or dashed case, for dashed $y = 0$ and $z = -1$ and for dotted $y = 1$ and $z = 0$.

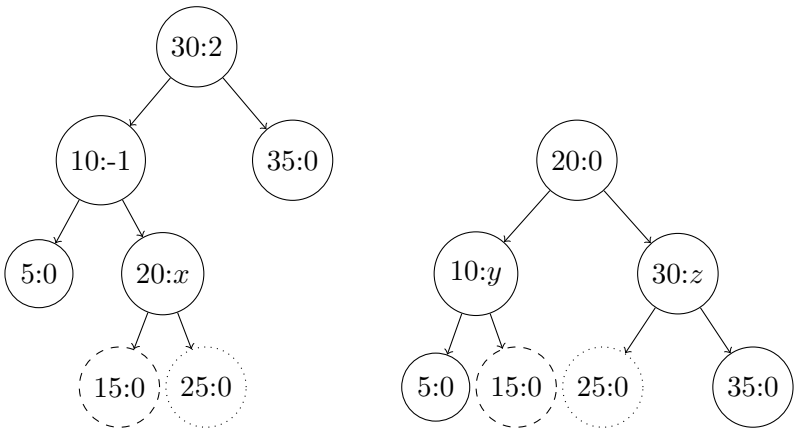


Figure 13: A before and after picture of the LR rotation.

exampled. Of course, for the LR rotation we actually looked at two examples at once, the dotted example and the dashed example, but there is a simpler case, given in Fig. 14 where some of the nodes are replaced with NULLs, but, more importantly, there is a more general case where some of the nodes are replaced by subtrees; this is given in Fig. 15.

This leaves the issue of how the balance factors are updated. This is done by a process of, in a sense, pushing the potential trouble upwards until it either sorts itself out or becomes actual trouble and gets fixed.

Imagine adding a new node, called **new** to the left of an existing node, which we'll call **here**. If the existing node has balance factor one then adding **new** would change it to two and a rotation would be needed. Alternatively, the node might have a balance factor of zero, in which case its new balance factor would be one and the subtree coming from **here** is now one longer than it used to be; this needs to be addressed by considering what happens to **here**'s parent and so the process iterates with the balance factor of **here**'s parent being looked at. Finally, **here** might have had a balance factor of -1, in which case its balance factor become zero and, since the length of the substree rooted in **here** has been unchanged there is nothing more to do. Obviously, as far as iterating this process is concerned **new** can either be a new node, or a subtree rooted in **new** that is one longer than it used to be and what applied to adding to the left also works for adding to the right, except one is taken from the balance factor of **here** instead of one being added. The code that does this can be seen in Table 5.

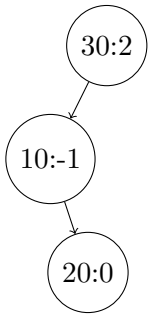
```
1  struct node * rotate_lr(struct node * here)
2  {
3    struct node * left=here->left;
4    struct node * left_right=left->right;
5
6    left->right=left_right->left;
7    left_right->left=left;
8
9    here->left=left_right->right;
10   left_right->right=here;
11
12   if(left_right->balance==1)
13     here->balance=-1;
14   else
15     here->balance=0;
16
17   if(left_right->balance==-1)
18     left->balance=1;
19   else
20     left->balance=0;
21
22   here=left_right;
23
24   here->balance=0;
25
26   return here;
27 }
```

Table 4: The LR rotation. In lines 3 and 4 news pointer are made to keep track of **here**− >left and **here**− >left− >right. The new balance factors at the end depend on the balance factor **left_right** had, just as the balance factors y and z depended on x.

```
1  struct node * add_node_r(struct node * here,int new_entry,int * work_needed)
2  {
3    if(here==NULL)
4      {
5        *work_needed = 1;
6        return make_node(new_entry);
7      }
8
9    if(new_entry<here->entry)
10     {
11       here->left = add_node_r(here->left ,new_entry,work_needed);
12       if(*work_needed)
13         {
14           switch(here->balance)
15             {
16             case -1:
17               here->balance=0;
18               *work_needed=0;
19               return here;
20             case 0:
21               here->balance=1;
22               return here;
23             case 1:
24               if(here->left->balance==1)
25                 here=rotate_ll(here);
26               else
27                 here=rotate_lr(here);
28                 *work_needed=0;
29                 return here;
30             }
31         }
32     }
33   else
34     {
35       [RIGHT CASE]
36   }
37   return here;
38
39 }
```

Table 5: Updating the balance factor. This recursive function uses an int called work_needed to decide if an addition is resolved. It works down the tree until it finds where to add the new node, this happens at line 3-7, as it returns up the recursion it updates the balance factors if work_needed is one, this uses the switch command from lines 14-30, if the balance factor is one then it should be changed to two and a rotation is done to avoid that, if it is zero it is changed to one and if it is -1 it is changed to zero and work needed is set to zero. The right version is similar and can be seen in the overall program `AVL_tree.c`.

A



B

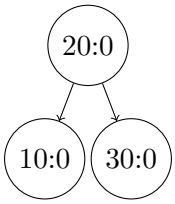
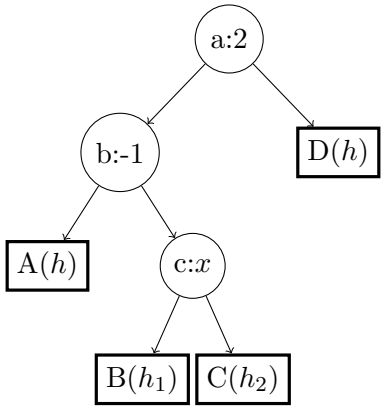


Figure 14: Another LR rotation. **A** represents before and **B** after. Here **left_right** is also a leaf, but following the same instructions gives the rotation.

A



B

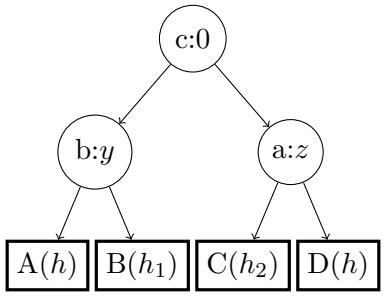


Figure 15: A general LR rotation. As before, the rectangles correspond to subtrees and the number in brackets to heights. The values of h_1 and h_2 can vary between two cases, $h_1 = h$ and $h_2 = h - 1$ or $h_1 = h - 1$ and $h_2 = h$ with $x = 1$ or $x = -1$ respectively, corresponding, roughly, to the dashed and dotted cases consider before. **A** is before, **B** is after; as before the values of y and z depend on the value of x, if x was one then $y = 0$ and $z = -1$, if x was -1 then $y = 1$ and $z = 0$.