Lecture 16
AVL Trees

15-122: Principles of Imperative Computation (Spring 2022)
Frank Pfenning

Binary search trees are an excellent data structure to implement associative arrays, maps, sets, and similar interfaces. The main difficulty is that they are efficient only when they are balanced. Straightforward sequences of insertions can lead to highly unbalanced trees with poor asymptotic complexity and unacceptable practical efficiency. For example, if we insert n entries with keys that are in strictly increasing or decreasing order, the complexity for n insertions will be $O(n^2)$. On the other hand, if we can keep the height to $O(\log n)$, as it is for a perfectly balanced tree, then the complexity is bounded by $O(n \log n)$.

The tree can be kept balanced by dynamically *rebalancing* the search tree during insert or search operations. We have to be careful not to destroy the ordering invariant of the tree while we rebalance. Because of the importance of binary search trees, researchers have developed many different algorithms for keeping trees in balance, such as AVL trees, red/black trees, splay trees, or randomized binary search trees. They differ in the invariants they maintain (in addition to the ordering invariant), and when and how the rebalancing is done.

In this lecture we use *AVL trees*, which is a simple and efficient data structure to maintain balance, and is also the first that has been proposed. It is named after its inventors, G.M. Adelson-Velskii and E.M. Landis, who described it in 1962.

Additional Resources

- Review slides (`https://cs.cmu.edu/~15122/slides/review/16-avl.pdf`)
- Code for this lecture (`https://cs.cmu.edu/~15122/code/16-avl.tgz`)

In terms of the learning objectives of the course, AVL trees make the following contributions:

Computational Thinking: We learn that the computational limitations of a data structure (here the possibility that binary search trees can develop a linear behavior) can sometime be overcome through clever thinking (here rebalancing).

Algorithms and Data Structures: We examine AVL trees as an example of self-balancing trees.

Programming: We use contracts to guide the implementation of code with increasingly complex invariants.

1 The Height Invariant

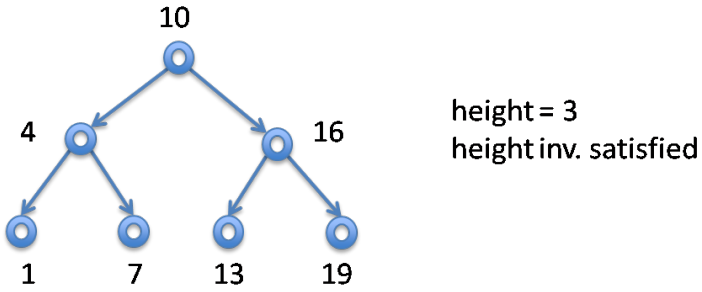
Recall the *ordering invariant* for binary search trees.

> **Ordering Invariant.** At any node with key k in a binary search tree, all keys of the entries in the left subtree are strictly less than k, while all keys of the entries in the right subtree are strictly greater than k.

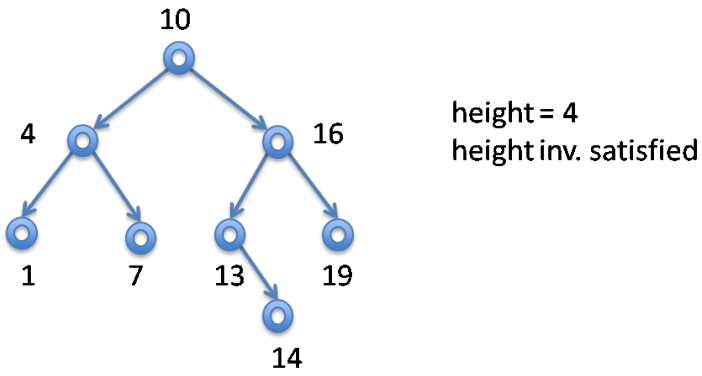
To describe AVL trees we need the concept of *tree height*, which we define as the maximal length of a path from the root to a leaf. So the empty tree has height 0, the tree with one node has height 1, a balanced tree with three nodes has height 2. If we add one more node to this last tree is will have height 3. Alternatively, we can define it recursively by saying that the empty tree has height 0, and the height of any node is one greater than the maximal height of its two children. AVL trees maintain a *height invariant* (also sometimes called a *balance invariant*).

> **Height Invariant.** At any node in the tree, the heights of the left and right subtrees differ by at most 1.

As an example, consider the following binary search tree of height 3.

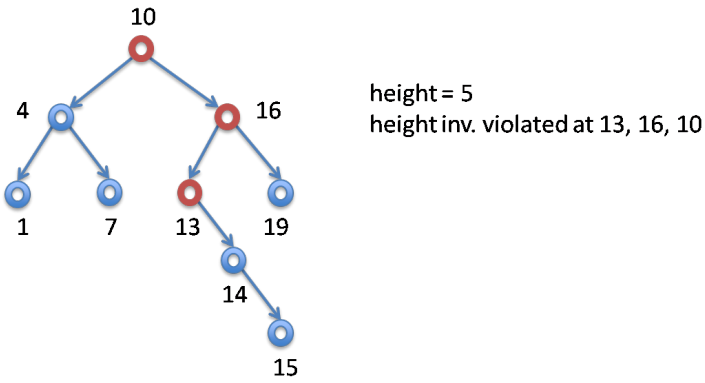


If we insert a new entry with a key of 14, the insertion algorithm for binary search trees without rebalancing will put it to the right of 13.



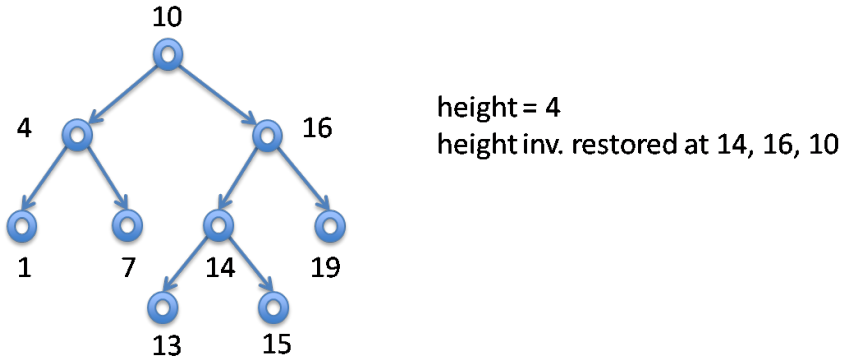
Now the tree has height 4, and one path is longer than the others. However, it is easy to check that at each node, the height of the left and right subtrees still differs only by one. For example, at the node with key 16, the left subtree has height 2 and the right subtree has height 1, which still obeys our height invariant.

Now consider another insertion, this time of an entry with key 15. This is inserted to the right of the node with key 14.



All is well at the node labeled 14: the left subtree has height 0 while the right subtree has height 1. However, at the node labeled 13, the left subtree has height 0, while the right subtree has height 2, violating our invariant. Moreover, at the node with key 16, the left subtree has height 3 while the right subtree has height 1, also a difference of 2 and therefore an invariant violation.

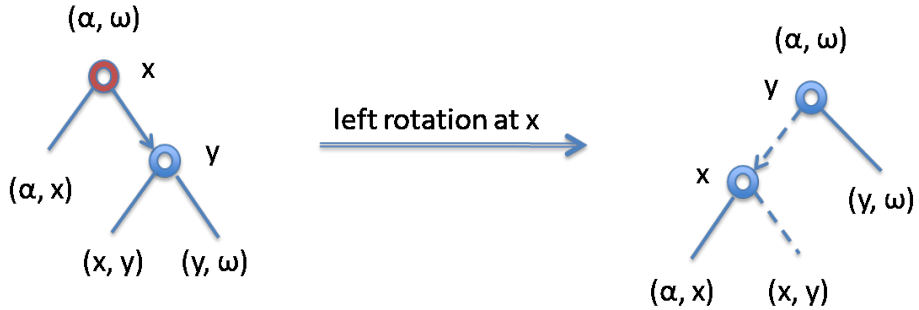
We therefore have to take steps to rebalance the tree. We can see without too much trouble that we can restore the height invariant if we move the node labeled 14 up and push node 13 down and to the left, resulting in the following tree.



The question is how to do this in general. In order to understand this we need a fundamental operation called a *rotation*, which comes in two forms, *left rotation* and *right rotation*.

2 Left and Right Rotations

Below, we show the situation before a left rotation. We have generically denoted the crucial key values in question with x and y. Also, we have summarized whole subtrees with the intervals bounding their key values. At the root of the subtree we can have intervals that are unbounded on the left or right. We denote these with pseudo-bounds $-\infty$ on the left and $+\infty$ on the right. We then write α for a left endpoint which could either be an integer or $-\infty$ and ω for a right endpoint which could be either an integer of $+\infty$. The tree on the right is after the left rotation.



From the intervals we can see that the ordering invariants are preserved, as are the contents of the tree. We can also see that it shifts some nodes from the right subtree to the left subtree. We would invoke this operation if the invariants told us that we have to rebalance from right to left.

We implement this with some straightforward code. First, recall the type of trees from last lecture. We do not repeat the functions `is_tree` that checks the basic structure of a tree and `is_ordered` that checks if a tree is ordered.

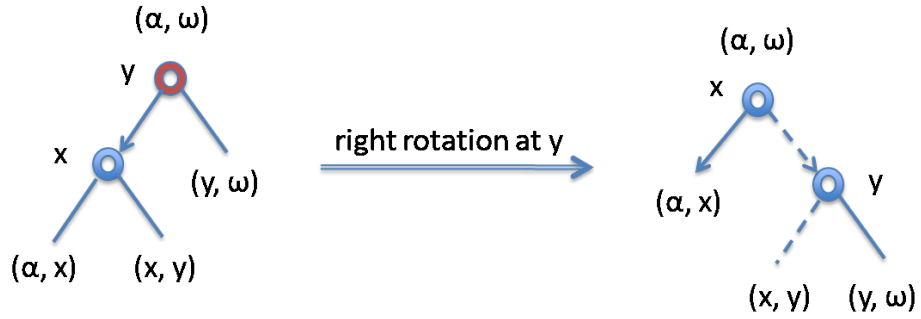
```
struct tree_node {
  entry data;
  struct tree_node* left;
  struct tree_node* right;
};
typedef struct tree_node tree;
bool is_tree(tree* T);
bool is_ordered(tree* T, entry lo, entry hi);
```

The main point to keep in mind is to use (or save) a component of the input before writing to it. We apply this idea systematically, writing to a location immediately after using it on the previous line.

```
tree* rotate_left(tree* T)
//@requires T != NULL && T->right != NULL;
{
  tree* R = T->right;
  T->right = T->right->left;
  R->left = T;
  return R;
}
```

These rotations work generically. When we apply them to AVL trees specifically later in this lecture, we will also have to recalculate the heights of the two nodes involved. This involves only looking up the height of their children.

The right rotation is exactly the inverse. First in pictures:



Then in code:

```
tree* rotate_right(tree* T)
//@requires T != NULL && T->left != NULL;
{
  tree* R = T->left;
  T->left = T->left->right;
  R->right = T;
  return R;
}
```

3 Searching for a Key

Searching for a key in an AVL tree is identical to searching for it in a plain binary search tree. We only need the ordering invariant to find the entry; the height invariant is only relevant for inserting an entry.

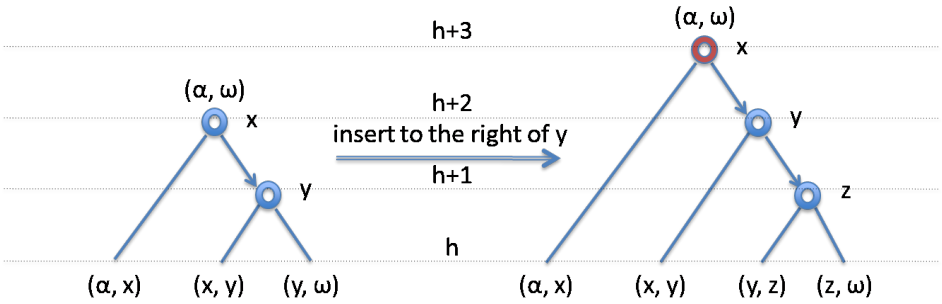
4 Inserting an Entry

The basic recursive structure of inserting an entry is the same as for searching for an entry. We compare the entry's key with the keys associated with the nodes of the trees, inserting recursively into the left or right subtree. When we find an entry with the exact key we overwrite the entry in that node. If we encounter a null tree, we construct a new tree with the entry to be inserted and no children and then return it. As we return the new subtrees (with the inserted entry) towards the root, we check if we violate the height invariant. If so, we rebalance to restore the invariant and then continue up the tree to the root.

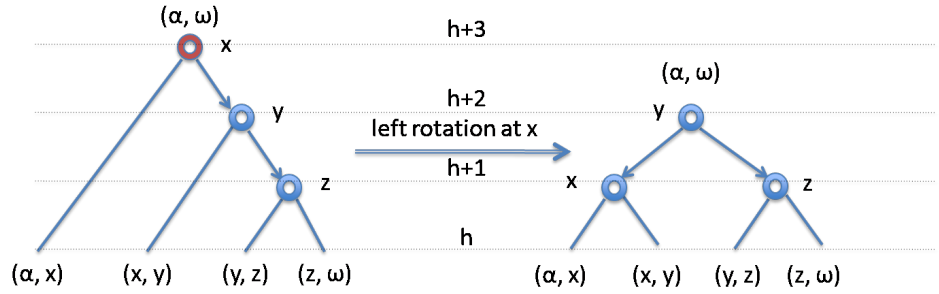
The main cleverness of the algorithm lies in analyzing the situations when we have to rebalance and need to apply the appropriate rotations to restore the height invariant. It turns out that one or two rotations on the whole tree always suffice for each insert operation, which is a very elegant result.

First, we keep in mind that the left and right subtrees' heights before the insertion can differ by at most one. Once we insert an entry into one of the subtrees, they can differ by at most two. We now draw the trees in such a way that the height of a node is indicated by the height that we are drawing it at.

The first situation we describe is where we insert into the right subtree, which is already of height $h + 1$ where the left subtree has height h. If we are unlucky, the result of inserting into the right subtree will give us a new right subtree of height $h + 2$ which raises the height of the overall tree to $h + 3$, violating the height invariant. This situation is depicted below. Note that the node we inserted does not need to be z, but there must be a node z in the indicated position.

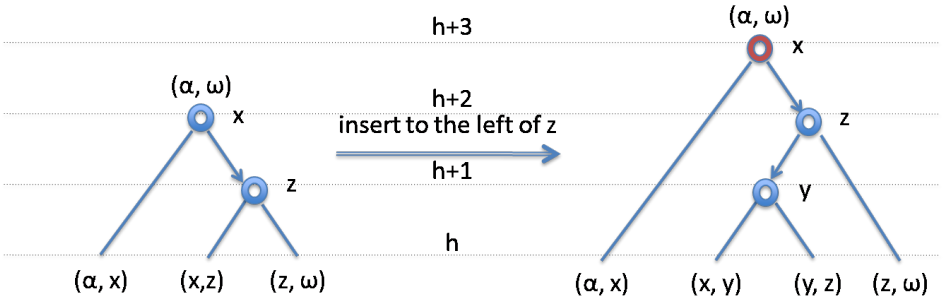


If the new right subtree has height $h + 2$, either its right or its left subtree must be of height $h + 1$ (and only one of them — think about why). If it is the right subtree we are in the situation depicted on the right above (and on the left below). While the trees (α, x) and (x, y) must have exactly height h, the trees (y, z) and (z, ω) need not. However, they differ by at most 1, because we are investigating the case where the lowest place in the tree where the invariant is violated is at x.

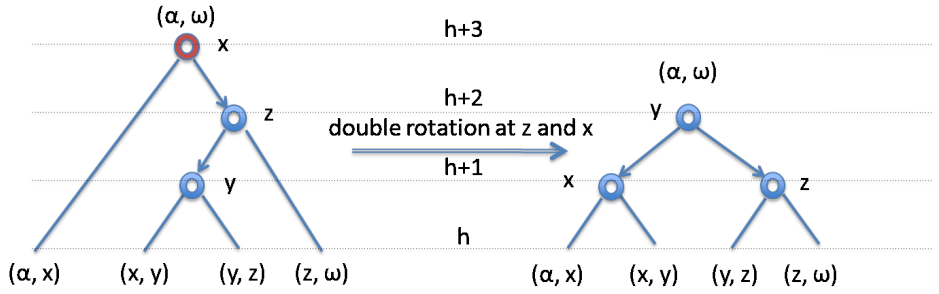


We fix this with a left rotation at x, the result of which is displayed to the right. Because the height of the overall tree is reduced to its original $h + 2$, no further rotation higher up in the tree will be necessary.

In the second case we consider, we insert to the *left* of the right subtree, and the result has height $h+1$. This situation is depicted on the right below.



In the situation on the right, the subtrees labeled (α, x) and (z, ω) must have exactly height h, but only one of (x, y) and (y, z) does. In this case, a single left rotation alone will not restore the invariant (see Exercise 5). Instead, we apply a so-called *double rotation*: first a *right* rotation at z, then a *left* rotation at the root labeled x. When we do this we obtain the picture on the right, restoring the height invariant.



There are two additional symmetric cases to consider, if we insert the new entry on the left (see Exercise 3).

We can see that in each of the possible cases where we have to restore the invariant, the resulting tree has the same height $h + 2$ as before the insertion. Therefore, the height invariant *above* the place where we just restored it will be automatically satisfied, without any further rotations.

5 Checking Invariants

The interface for the implementation is *exactly* the same as for binary search trees, as is the code for searching for a key. In various places in the algorithm we have to compute the height of the tree. This could be an operation of asymptotic complexity $O(n)$, unless we store it in each node and just look it up. So we have:

```
typedef struct tree_node tree;
struct tree_node {
  entry data;
  int height;          // New
  tree* left;
  tree* right;
};

/* height(T) returns the precomputed height of T in O(1) */
int height(tree* T) {
  return T == NULL ? 0 : T->height;
}
```

The conditional expression b `?` e_1 `:` e_2 evaluates to the result of e_1 if the boolean test b returns `true` and to the value of e_2 if it returns `false`.

When checking if a tree is balanced, we check that all the heights that have been computed are correct.

```
bool is_specified_height(tree* T) {
  if (T == NULL) return true;
  return is_specified_height(T->left)
      && is_specified_height(T->right)
      && T->height == max(height(T->left),
                          height(T->right)) + 1;
}
```

```
bool is_balanced(tree* T) {
  if (T == NULL) return true;
  return abs(height(T->left) - height(T->right)) <= 1
      && is_balanced(T->left) && is_balanced(T->right);
}
```

A tree is an AVL tree if it is both ordered (as defined and implemented in the BST lecture, and extended by our `is_specified_height` condition) and balanced.

```
bool is_avl(tree* T) {
  return is_tree(T) && is_ordered(T, NULL, NULL)
      && is_specified_height(T)
      && is_balanced(T);
}
```

Of course, if we store the height of the trees for fast access, we need to adapt it when rotating trees. After all, the whole purpose of tree rotations is to rebalance and change the height. For that, we implement a function `fix_height` that computes the height of a tree from the height of its children. Its implementation directly follows the definition of the height of a tree.

```
void fix_height(tree* T)
//@requires T != NULL;
//@requires is_specified_height(T->left);
//@requires is_specified_height(T->right);
{
  int hl = height(T->left);
  int hr = height(T->right);
  T->height = (hl > hr ? hl+1 : hr+1);
  return;
}
```

The implementation of `rotate_right` and `rotate_left` needs to be adapted to include calls to `fix_height`. These calls need to compute the heights of the children first, before computing that of the root, because the height of the root depends on the height we had previously computed for the child. Hence, we need to update the height of the child before updating the height of the root. For example, `rotate_left` is upgrades as follows

```
tree* rotate_left(tree* T)
//@requires T != NULL && T->right != NULL;
//@requires is_specified_height(T->left);
//@requires is_specified_height(T->right);
//@ensures is_specified_height(\result);
{
  tree* R = T->right;
  T->right = T->right->left;
  R->left = T;
  fix_height(T);
  fix_height(R);
  return R;
}
```

We use this, for example, in a utility function that creates a new leaf from an entry (which may not be `NULL`).

```
tree* leaf(entry e)
//@requires e != NULL;
//@ensures is_avl(\result);
{
  tree* T = alloc(tree);
  T->data = e;
  T->height = 1;
  return T;
}
```

Recall that the pointer fields are set to `NULL` by default when the structure is allocated.

6 Implementing Insertion

The code for inserting an entry into the tree is mostly identical to the code for plain binary search trees. The difference is that after we insert into the left or right subtree, we call a function `rebalance_left` or `rebalance_right`, respectively, to restore the invariant if necessary and calculate the new height.

```
tree* tree_insert(tree* T, entry e)
//@requires is_avl(T) && e != NULL;
//@ensures is_avl(\result);
{
  if (T == NULL) return leaf(e);

  //@assert is_avl(T->left);
  //@assert is_avl(T->right);
  int cmp = key_compare(entry_key(e), entry_key(T->data));
  if (cmp == 0) {        // Found
    T->data = e;
  } else if (cmp < 0) {  // Go left
    T->left = tree_insert(T->left, e);
    //@assert is_avl(T->left);
    T = rebalance_left(T);               // New
    //@assert is_avl(T);
  } else if (cmp > 0) {  // Go right
    T->right = tree_insert(T->right, e);
    //@assert is_avl(T->right);
    T = rebalance_right(T);              // New
    //@assert is_avl(T);
  }
  return T;
}
```

The pre- and post-conditions of this function are actually not strong enough to prove this function correct. We also need an assertion about how the tree might change due to insertion, which is somewhat tedious. If we perform dynamic checking with the contract above, however, we establish that the result is indeed an AVL tree. As we have observed several times already: we can test for the desired property, but we may need to strengthen the pre- and post-conditions in order to rigorously prove it.

We show only the function `rebalance_right`; `rebalance_left` is symmetric.

```
tree* rebalance_right(tree* T)
//@requires T != NULL && T->right != NULL;
//@requires is_avl(T->left) && is_avl(T->right);
//@ensures is_avl(\result);
{
  if (height(T->right) - height(T->left) == 2) {
    if (height(T->right->right) > height(T->right->left)) {
      // Single rotation
      T = rotate_left(T);
    } else {
      //@assert height(T->right->left) > height(T->right->right);
      // Double rotation
      T->right = rotate_right(T->right);
      T = rotate_left(T);
    }
  } else { // No rotation needed, but tree may have grown
    fix_height(T);
  }
  return T;
}
```

Note that the preconditions are weaker than we would like. In particular, they do not imply some of the assertions we have added in order to show the correspondence to the pictures. This is left as the Exercises 8–11. Such assertions are nevertheless useful because they document expectations based on informal reasoning we do behind the scenes. Then, if they fail, they may be evidence for some error in our understanding, or in the code itself, which might otherwise go undetected.

7 Experimental Evaluation

We would like to assess the asymptotic complexity and then experimentally validate it. It is easy to see that both insert and lookup operations take time $O(h)$, where h is the height of the tree. But how is the height of the tree related to the number of entries stored, if we use the balance invariant of AVL trees? It turns out that h is $O(\log n)$. It is not difficult to prove this, but it is beyond the scope of this course.

To experimentally validate this prediction, we have to run the code with inputs of increasing size. A convenient way of doing this is to double the size of the input and compare running times. If we insert n entries into the tree and look them up, the running time should be bounded by $c \times n \times \log n$ for some constant c. Assume we run it at some size n and observe $r = c \times n \times \log n$. If we double the input size we have $c \times (2 \times n) \times \log(2 \times n) = 2 \times c \times n \times (1 + \log n) = 2 \times r + 2 \times c \times n$, we mainly expect the running time to double with an additional summand that roughly doubles as n doubles. In order to smooth out minor variations and get bigger numbers, we run each experiment 100 times. Here is the table with the results:

n	AVL trees	increase	BSTs
2^9	0.129	–	1.018
2^{10}	0.281	$2r + 0.023$	2.258
2^{11}	0.620	$2r + 0.058$	3.094
2^{12}	1.373	$2r + 0.133$	7.745
2^{13}	2.980	$2r + 0.234$	20.443
2^{14}	6.445	$2r + 0.485$	27.689
2^{15}	13.785	$2r + 0.895$	48.242

We see in the third column, where $2r$ stands for the doubling of the previous value, we are quite close to the predicted running time, with a approximately linearly increasing additional summand.

In the fourth column we have run the experiment with plain binary search trees which do not rebalance automatically. First of all, we see that they are much less efficient, and second we see that their behavior with increasing size is difficult to predict, sometimes jumping considerably and sometimes not much at all. In order to understand this behavior, we need to know more about the order and distribution of keys that were used in this experiment. They were strings, compared lexicographically. The keys were generated by counting integers upward and then converting them to strings. The distribution of these keys is haphazard, but not random. For example, if we start counting at 0

```
"0" < "1" < "2" < "3" < "4" < "5" < "6" < "7" < "8" < "9"
    < "10" < "12" < ...
```

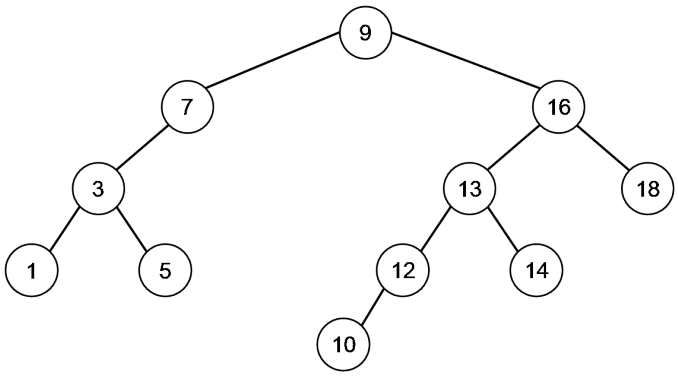
the first ten strings are in ascending order but then numbers are inserted between `"1"` and `"2"`. This kind of haphazard distribution is typical of

many realistic applications, and we see that binary search trees without rebalancing perform quite poorly and unpredictably compared with AVL trees.

The complete code for this lecture can be found on the course website.

8 Exercises

Exercise 1 (sample solution on page 17)**.** *In the following tree, which nodes violate the height invariant?*



Exercise 2 (sample solution on page 17)**.** *Draw all the rotations that you must perform and the final AVL tree after the following elements are inserted in the given order starting from an empty tree.*

$$1, \quad 10, \quad 5, \quad 7, \quad 3, \quad 13, \quad 6, \quad 4, \quad 8, \quad 9$$

Exercise 3 (sample solution on page 19)**.** *Draw a valid AVL tree and give two keys to be inserted on the left subtree of this tree. Either insertion should cause a violation of the height invariant at the root, but fixing them should require different rotations below the root. Indicate which rotations and draw the updated trees.*

Exercise 4 (sample solution on page 20)**.** *We are inserting the following five keys into an initially empty tree:*

$$6, \quad 20, \quad 45, \quad 80, \quad 96$$

Determine an insertion order that will trigger a double rotation at some point (and no other rotations).

Exercise 5 (sample solution on page 21)**.** *Consider the situation where* `tree_insert` *in Section 4 performs a double rotation. Show that a single rotation at the root of the tree may not necessarily restore the height invariant.*

Exercise 6 (sample solution on page 21)**.** *Draw pictures that demonstrate that left and right rotations are inverses of each other. What can you say about double rotations?*

Exercise 7 (sample solution on page 22)**.** *Show, in pictures, that a double rotation is a composition of two rotations. Discuss the situation with respect to the height invariants after the first rotation.*

Exercise 8 (sample solution on page 24)**.** This is the first of a four-part exercise to show that `tree_insert` meets its postconditions.

Using point-to reasoning, show that the code for `tree_insert` *in Section 6 satisfies its postcondition. You may assume that the contracts of all the functions called by* `tree_insert` *are correct (including* `tree_insert` *itself).*

Exercise 9 (sample solution on page 26)**.** This is the second of a four-part exercise to show that `tree_insert` meets its postconditions.

Once `tree_insert` *steps into* `rebalance_right`*, the current contracts of* `rotate_left` *and* `rotate_right` *are not strong enough to prove the correctness of the* `rebalance_right` *function. Explain why. Then, strengthen the pre- and post-conditions of the* `rotate_left` *function such that the postconditions of* `rebalance_right` *holds when it performs a single rotation. Strengthen the contracts of* `rotate_right` *similarly.*

Exercise 10 (sample solution on page 28)**.** This is the third of a four-part exercise to show that `tree_insert` meets its postconditions.

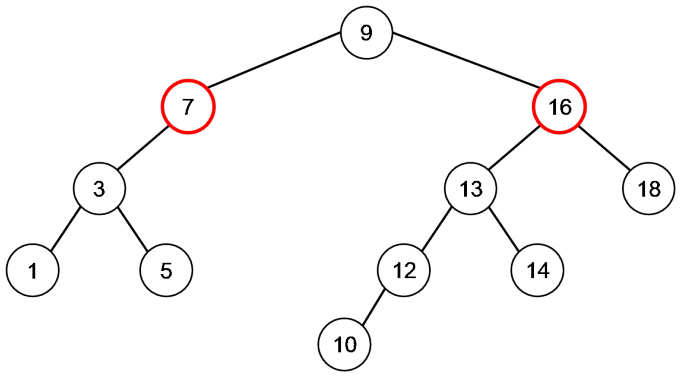
The strengthened contracts for the rotate functions now support the postcondition of `rebalance_right` *when dealing with a single rotation. Things don't work though in the case of a double rotation. Explain what goes wrong and propose a fix (you don't need to implement this fix).*

Exercise 11 (sample solution on page 29)**.** This is the fourth of a four-part exercise to show that `tree_insert` meets its postconditions.

Implement the function `rotate_right_left(T)` *that performs a right-left double rotation on tree* `T` *without relying on single rotations. When used in* `rebalance_right`*, the contracts of this function should be strong enough to allow proving that* `rebalance_right` *meets its postconditions when performing a double rotation.*

Sample Solutions

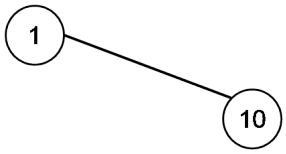
Solution of exercise 1 Nodes 7 and 16 violate the height invariant:



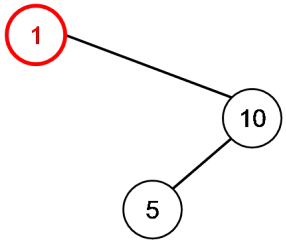
Node 7: The left subtree has height 2, but the right subtree has height 0 since it is empty. Since $|2 - 0| = 2 > 1$, there is a violation.

Node 16: The left subtree has height 3 while the right subtree has height 1. Since $|3 - 1| = 2 > 1$, there is a violation.

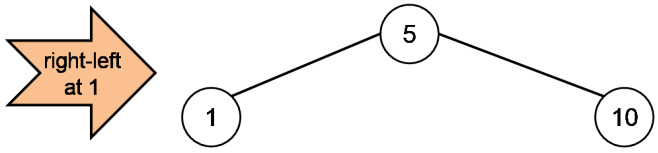
Solution of exercise 2 We first add 1 and 10 without making any rotations:



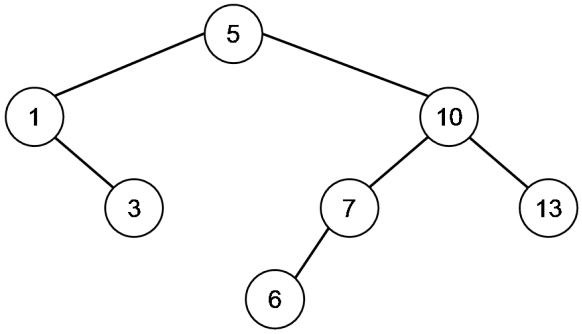
After adding 5 as in a binary search tree, the height invariant is violated since the height of the right subtree rooted at 1 is two more than the height of the left subtree.



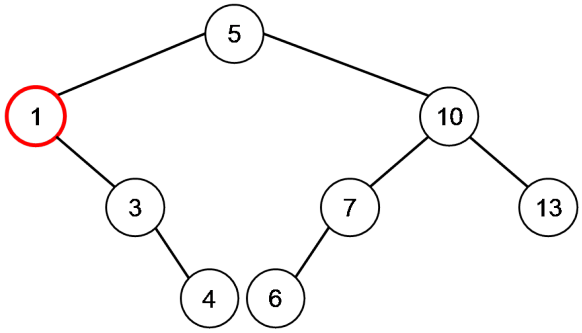
In order to fix this, we apply a double rotation at 1:



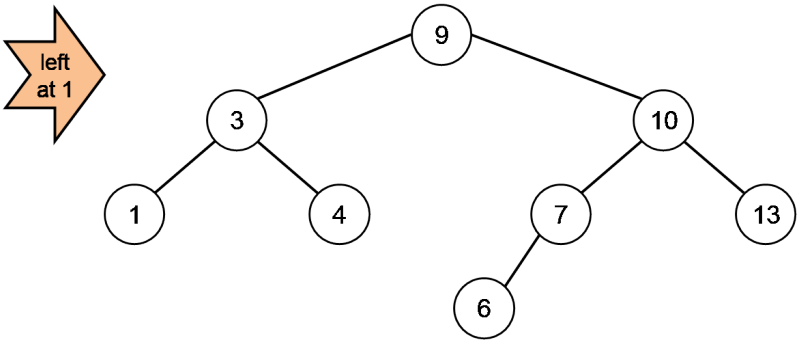
Inserting 7, 3, 13 and 6 does not require performing any rotations:



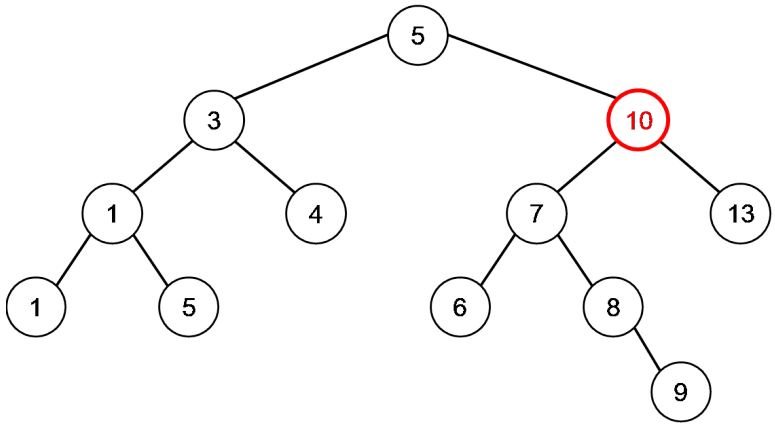
After inserting 4, the height invariant at node 1 is violated.



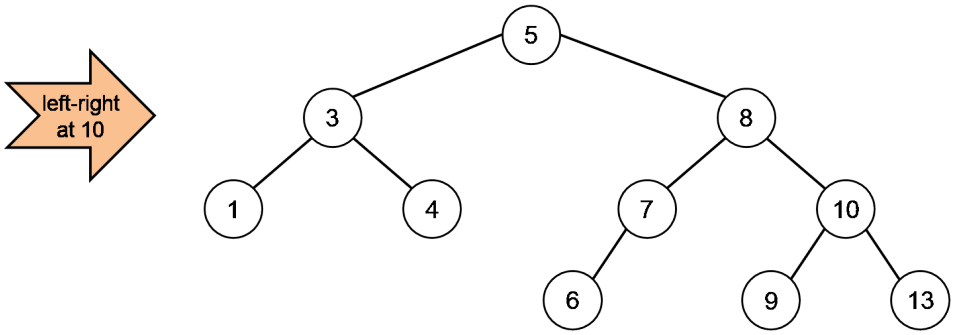
It takes a single left rotation at 1 to fix it:



We then insert 8 without doing any rotations, but when we insert 9, the height invariant at node 10 is violated.

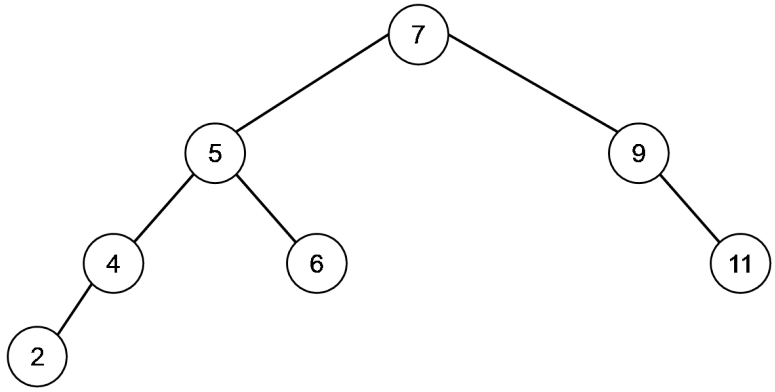


We fix this with a double right rotation at nodes 10:

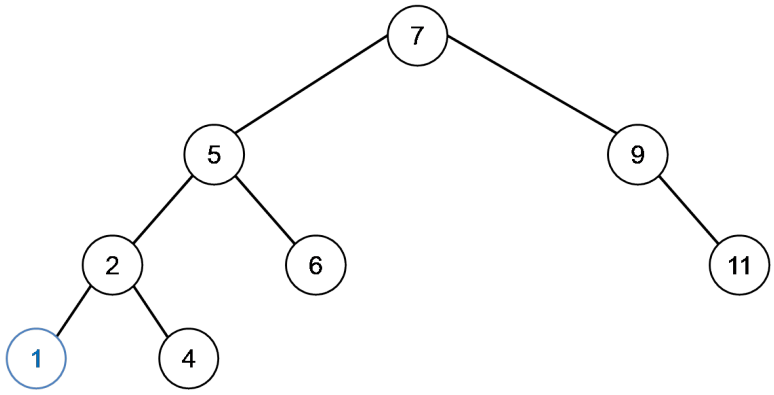


This is our final tree.

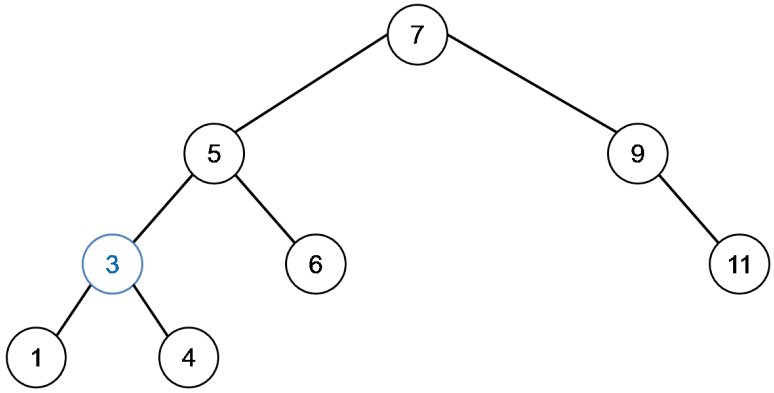
Solution of exercise 3 Consider the following AVL tree:



Inserting 1 as in a binary search tree will place it to the left of 2, causing violations at nodes 4, 5 and 7. A (single) right rotation at the lowest of these violations, node 4, fixes it, producing the following tree:



If instead we insert 3 in the original tree, it ends up to the right of 2, causing identical violations. This time, we need a double left-right rotation at node 4 to fix it. The resulting tree is as follows:

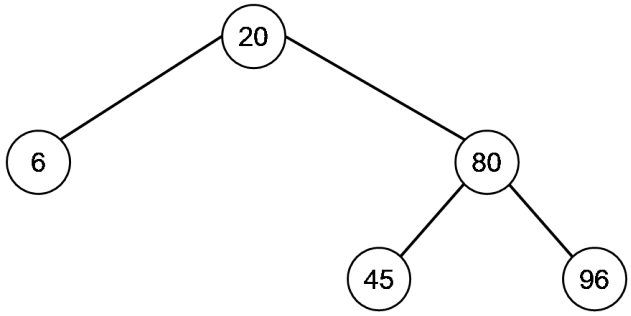


Notice that a single rotation was applied when the new node was initially inserted in an outer subtree of the tree rooted at the lowest violation, while a double rotation was used when the insertion occurred in an inner subtree. This observation applies generally.

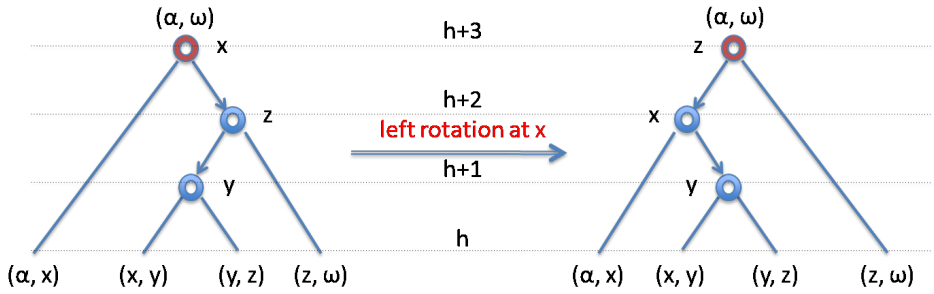
Solution of exercise 4 If we insert the given keys in the following order

$$20, \quad 6, \quad 45, \quad 96, \quad 80$$

the last insertion will result in a violation at node 45. Because 80 ends up in an inner subtree, we need to apply a double rotation, specifically a right-left rotation at 45. Here's the resulting AVL tree:



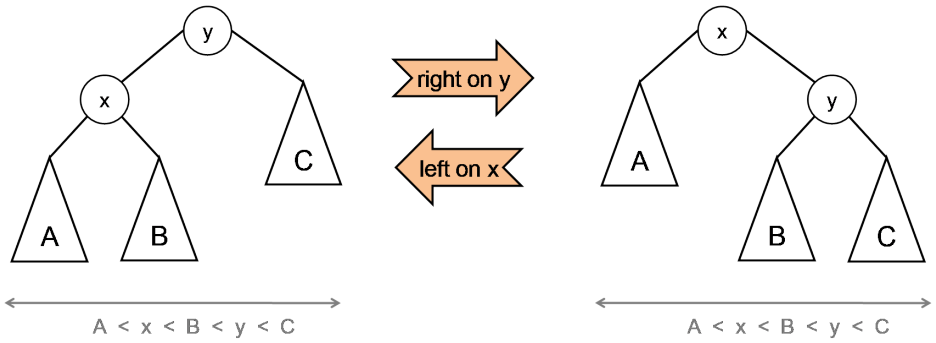
Solution of exercise 5 In the starting situation, on the left-side of the figure below, the lowest violation is at node x, meaning that its left subtree has height at most h. As noted in Section 4, the right subtree of z must also have height h. The outcome of performing a left rotation at x is displayed on the right.



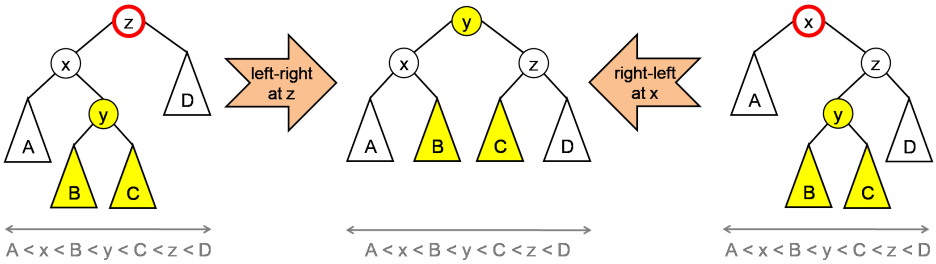
The left subtree of x still has height h since it was not modified. Its right subtree is the original subtree rooted at y, which has height $h+1$. Therefore the updated tree rooted x has now height $h+2$, which forces its parent, z, to be the root of a tree of height $h+3$. Thus, the left subtree of z has height $h+2$. However, its right subtree still has height h since it was not modified. We have a violation at z.

Thus, applying a left rotation at the root in this situation does not resolve the original violation: it just changes the node at this violation site.

Solution of exercise 6 Starting from the tree on the left of the figure below, if we apply a right rotation at node y, we obtain the tree on the right of the figure. From it, applying a left rotation gets us back to the tree on the left. The same happens if we start from the right subtree. This shows that the two single rotations are inverse of each other.

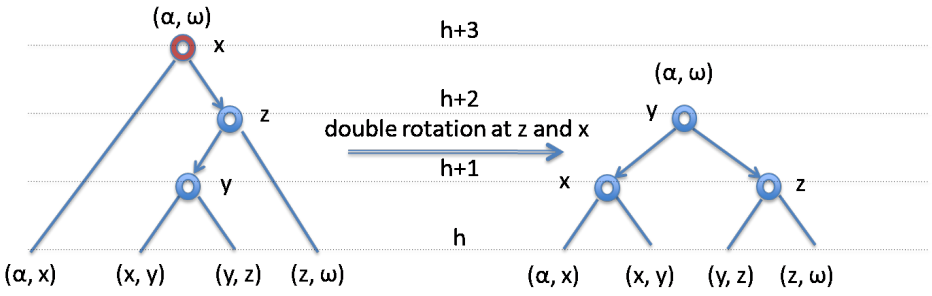


The double rotations are not inverse of each other: starting from a given tree and applying either of them yields a tree, we cannot go back to the original tree by applying the other double rotation.



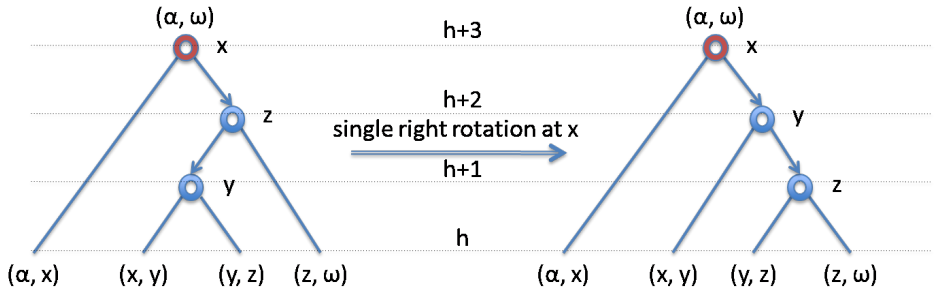
The two double rotations are however symmetric to each other, as are the single rotations.

Solution of exercise 7 Consider the case of a right-left double rotation from Section 4 (reproduced in the next figure):



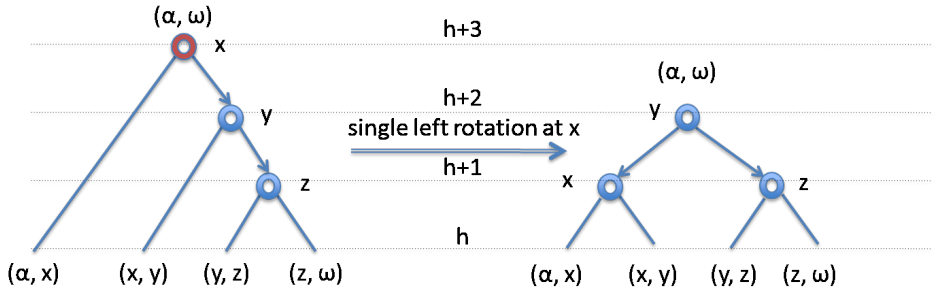
Recall that left subtree of node x and the right subtree of z must have height h, and at least one of the subtrees of y also has height h — the other could have height $h - 1$.

From the tree on the left, let's perform a single right rotation at node z (even if there is no violation there):



The tree rooted at z has height $h+1$ since its right subtree has height h and its left subtree has height at most h. The tree rooted at y has height $h+2$ since its right subtree has height $h+1$ and its left subtree has height at most h. Note that there could be a violation at y. The tree rooted at x still has height $h+3$ since its right subtree has height $h+2$ and its left subtree has height h — there is definitely a violation at x still although it may not be the lowest any more.

Next, let's perform a single rotation at node x:



The tree rooted at z is unchanged and therefore its height remains $h+1$. The tree rooted at x now has height $h+1$ because its left subtree has height h and its right subtree has height at most h. This makes y the root of two subtrees of height $h+1$ and therefore the overall tree rooted at y has height $h+2$.

This shows that a right-left double rotation can be viewed as a sequence of a single right rotation applied one node below the lowest violation, followed by a single left rotation applied on the node where the lowest violation was originally.

The case of a left-right double rotation is symmetric.

Solution of exercise 8 Here is the code for `tree_insert` from Section 6 together with the prototype of all the functions it calls.

```
1  tree* leaf(entry e)
2  /*@requires e != NULL;                              @*/
3  /*@ensures is_avl(\result);                         @*/ ;
4
5  tree* rebalance_left(tree* T)
6  /*@requires T != NULL && T->left != NULL;           @*/
7  /*@requires is_avl(T->left) && is_avl(T->right);    @*/
8  /*@ensures is_avl(\result);                         @*/ ;
9
10 tree* rebalance_right(tree* T)
11 /*@requires T != NULL && T->right != NULL;          @*/
12 /*@requires is_avl(T->left) && is_avl(T->right);    @*/
13 /*@ensures is_avl(\result);                         @*/ ;
14
15 tree* tree_insert(tree* T, entry e)
16 //@requires is_avl(T) && e != NULL;
17 //@ensures is_avl(\result);
18 {
19   if (T == NULL) return leaf(e);
20
21   //@assert is_avl(T->left);
22   //@assert is_avl(T->right);
23   int cmp = key_compare(entry_key(e), entry_key(T->data));
24   if (cmp == 0) {        // Found
25     T->data = e;
26   } else if (cmp < 0) {  // Go left
27     T->left = tree_insert(T->left, e);
28     //@assert is_avl(T->left);
29     T = rebalance_left(T);                // New
30     //@assert is_avl(T);
31   } else if (cmp > 0) {  // Go right
32     T->right = tree_insert(T->right, e);
33     //@assert is_avl(T->right);
34     T = rebalance_right(T);               // New
35     //@assert is_avl(T);
36   }
37   return T;
38 }
```

We need to show `is_avl(\result)` assuming the preconditions hold

and the functions that `tree_insert` calls work as advertised in their contracts. The function returns in two places, so we need to show that the postcondition holds in both.

The call returns on line 19:

A.	`\result== leaf(e)`	by line 19
B.	`is_avl(leaf(e))`	by line 3
C.	`is_avl(\result)`	by (A) and (B)

The call returns on line 37: If the execution makes it past line 19, we know that the left and right subtrees of `T` are themselves valid AVL trees, as noted in the assertions on lines 21 and 22.

We need to consider three cases depending on the result of the comparison on line 23. We will only pursue the case where `cmp < 0` (line 26) as the case where `cmp > 0` is symmetric and the case where `cmp == 0` is trivial since the tree structure does not change.

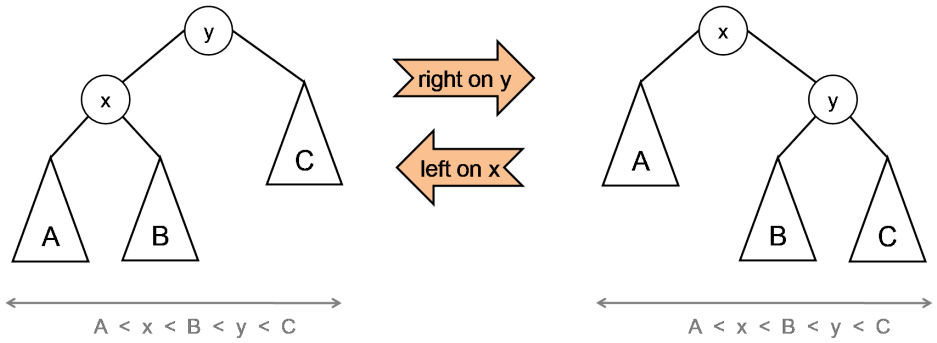
A.	`T->left != NULL`	by line 21
B.	`e != NULL`	by line 16
C.	`is_avl(T->left)`	by line 27 and (A–B)
D.	`rebalance_left`'s preconditions hold	by lines 16, 22 and (C)
E.	`is_avl(T)`	by lines 29 and 8
F.	`is_avl(\result)`	by line 37 and (D)

Solution of exercise 9 Here is the code for `rebalance_right` together with the prototype of the rotate functions:

```
1  tree* rotate_left(tree* T)
2  /*@requires T != NULL && T->right != NULL;  @*/ ;
3
4  tree* rotate_right(tree* T)
5  /*@requires T != NULL && T->left != NULL;   @*/ ;
6
7  tree* rebalance_right(tree* T)
8  //@requires T != NULL && T->right != NULL;
9  //@requires is_avl(T->left) && is_avl(T->right);
10 //@ensures is_avl(\result);
11 {
12   if (height(T->right) - height(T->left) == 2) {
13     if (height(T->right->right) > height(T->right->left)) {
14       // Single rotation
15       T = rotate_left(T);
16     } else {
17       //@assert height(T->right->left) > height(T->right->right);
18       // Double rotation
19       T->right = rotate_right(T->right);
20       T = rotate_left(T);
21     }
22   } else { // No rotation needed, but tree may have grown
23     fix_height(T);
24   }
25   return T;
26 }
```

Consider the situation where `rebalance_right` applies a single rotation by calling `rotate_left` on line 15. This call is safe by the preconditions of `rebalance_right` on line 8. However, `rotate_left` has no postconditions, which means we have no information on the updated value of the tree `T` that is returned by `rebalance_right`.

To fix this issue, let's look again at the diagrams for single rotations:



We know that single rotations are applied when the lowest violation is at the root of the input tree, with the effect that the rotated tree does not have a violation any more. This suggests equipping the rotate functions with a postcondition that says that the tree they return is indeed an AVL tree. However, adding just this contract does not allow us to prove that the rotate functions are correct. This is because being an AVL tree is a recursive property, and the current contract say nothing about the subtrees of the input tree `T`. But we know that both subtrees of `T` must be valid AVL trees. We can incorporate this as an additional precondition.

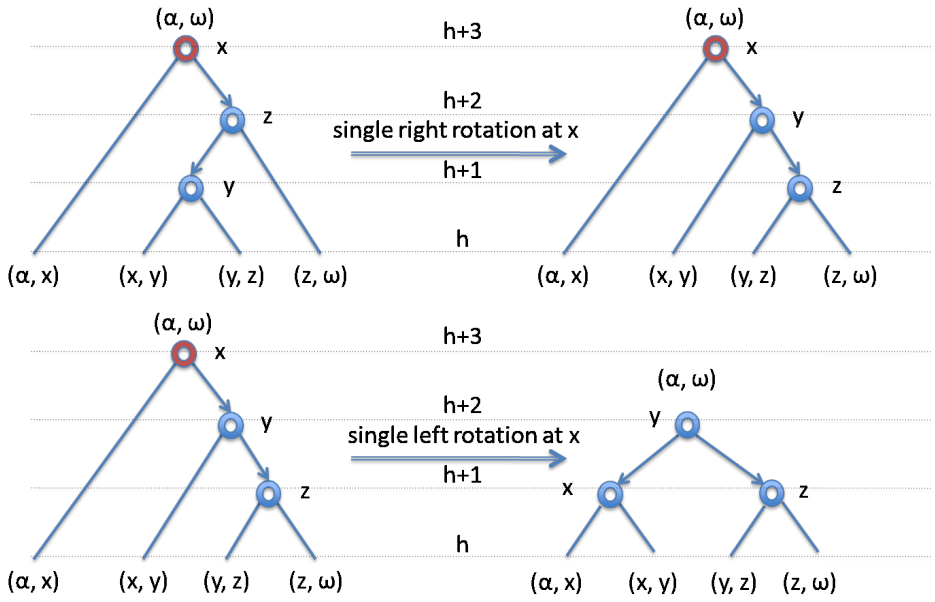
Altogether, the updated contracts for the rotate functions are:

```
tree* rotate_left(tree* T)
/*@requires T != NULL && T->right != NULL;        @*/
/*@requires is_avl(T->left) && is_avl(T->right);  @*/
/*@ensures is_avl(\result);                       @*/ ;

tree* rotate_right(tree* T)
/*@requires T != NULL && T->left != NULL;         @*/
/*@requires is_avl(T->left) && is_avl(T->right);  @*/
/*@ensures is_avl(\result);                       @*/ ;
```

These more precise contracts for `rotate_left` allow us to conclude that the call on line 15 returns a valid AVL tree. Since this same tree is returned by `rebalance_right`, the postconditions of this latter function are met in this case.

Solution of exercise 10 The function `rebalance_right` implements a right-left double rotation by making two single rotations, a right rotation on the root of the right subtree and then a left rotation on the original root. Here's what it looks like schematically:



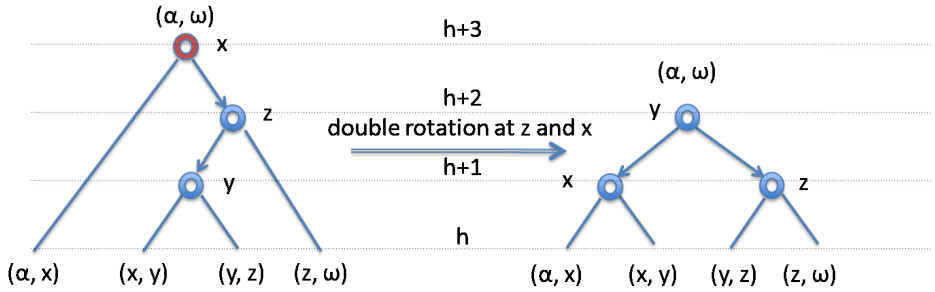
and here is the relevant portion of `rebalance_right`:

```
T->right = rotate_right(T->right);
T = rotate_left(T);
```

As observed earlier in Exercise 7, right after the first of these two single rotations, there may be a violation at node y. If there is, the call to `rotate_right(T->right)` will return a tree that is not an AVL tree. Said differently, it will fail the postcondition we extended it with in the last exercise.

How can we fix this? One easy way to do so is to implement double rotations as their own functions instead of as a composition of two single rotations. Double rotations take as input a tree whose two subtrees are valid AVL trees and return a valid AVL tree. We can record this insight as contracts for these functions. By modifying `rebalance_right` to use one such function that implements a right-left double rotation, we can then use these contracts to prove that it achieves its postcondition when performing a double rotation.

Solution of exercise 11 Here is the schematic form of a right-left double rotation from section 4:



To apply this rotation, the nodes x, y and z must exist. Furthermore, since it is applied at the lowest violation, both the right and left subtree of x are valid AVL trees. Since it fixes the violation, the tree rooted at y is a valid AVL tree. This defines the contracts of `rotate_left_right`.

The rest of the implementation is simple pointer manipulation. As we implement it, we need to be careful not to lose track of parts of the tree, and of course to update pointers so that the tree is modified correctly. Since we moved nodes x, y and z to different heights in the tree, we update their stored height using the function `fix_height` . We must fix the height of x and z before that of y since y is their parent in the returned tree.

```
tree* rotate_right_left(tree* T)
//@requires T != NULL && T->right != NULL;
//@requires T->right->left != NULL;
//@requires is_avl(T->left) && is_avl(T->right);
//@ensures is_avl(\result);
{
  tree* X = T;               // pointer to x
  tree* Y = T->right->left;  // pointer to y
  tree* Z = T->right;        // pointer to z

  X->right = Y->left;
  Z->left = Y->right;
  Y->left = X;
  Y->right = Z;

  fix_height(X);
  fix_height(Z);
  fix_height(Y);
  return Y;
}
```