Lecture 16 Notes
AVL Trees

15-122: Principles of Imperative Computation (Spring 2016)
Frank Pfenning

1 Introduction

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 elements 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.

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 de-

velop 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.

2 The Height Invariant

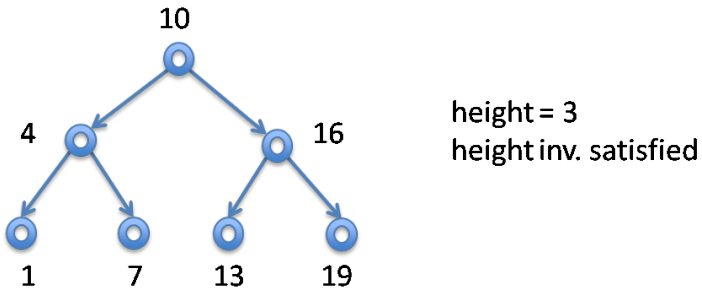
Recall the *ordering invariant* for binary search trees.

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

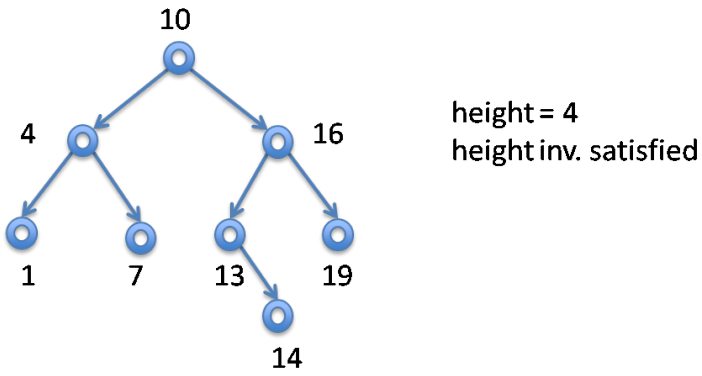
To describe AVL trees we need the concept of *tree height*, which we define as the maximal length of a path from the root to a leaf. So the empty tree has height 0, the tree with one node has height 1, a balanced tree with three nodes has height 2. If we add one more node to this last tree is will have 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 differs by at most 1.

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

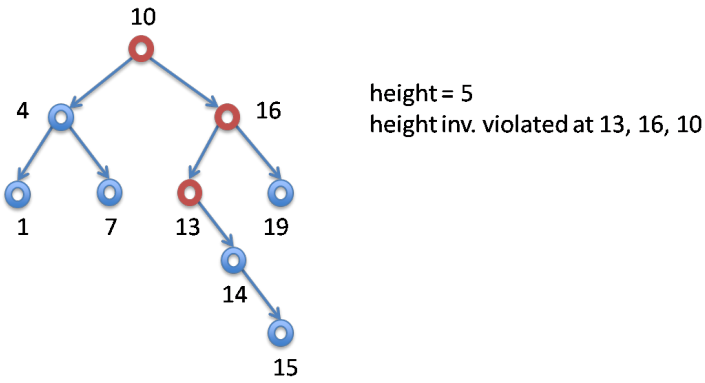


If we insert a new element with a key of 14, the insertion algorithm for binary search trees without rebalancing will put it to the right of 13.



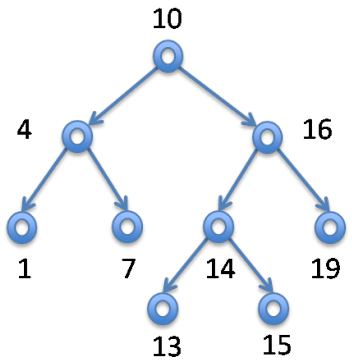
Now the tree has height 4, and one path is longer than the others. However, it is easy to check that at each node, the height of the left and right subtrees still differ only by one. For example, at the node with key 16, the left subtree has height 2 and the right subtree has height 1, which still obeys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

Now consider another insertion, this time of an element 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 right, resulting in the following tree.

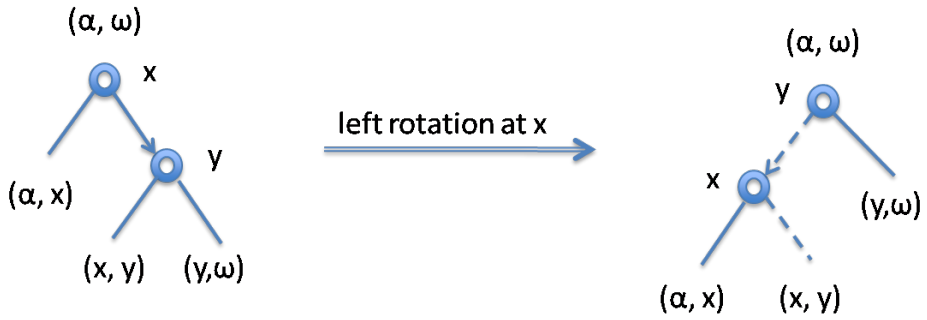


height = 4
height inv. restored at 14, 16, 10

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*.

3 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 function `is_ordered` that checks if a tree is ordered.

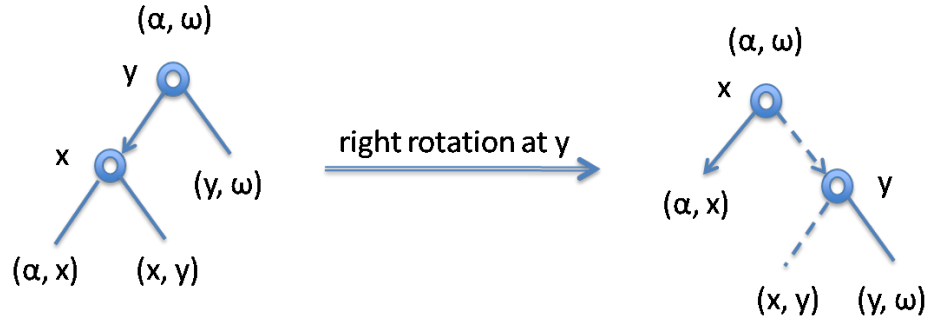
```
1 struct tree_node {
2   elem data;
3   struct tree_node* left;
4   struct tree_node* right;
5 };
6 typedef struct tree_node tree;
7 bool is_ordered(tree *T);
```

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. We repeat the type specification of tree from last lecture.

```
9  tree* rotate_left(tree* T)
10 //@requires T != NULL && T->right != NULL;
11 {
12   tree* R = T->right;
13   T->right = T->right->left;
14   R->left = T;
15   return R;
16 }
```

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:

```
18 tree* rotate_right(tree* T)
19 //@requires T != NULL && T->left != NULL;
20 {
21   tree* R = T->left;
22   T->left = T->left->right;
23   R->right = T;
24   return R;
25 }
```

4 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 element; the height invariant is only relevant for inserting an element.

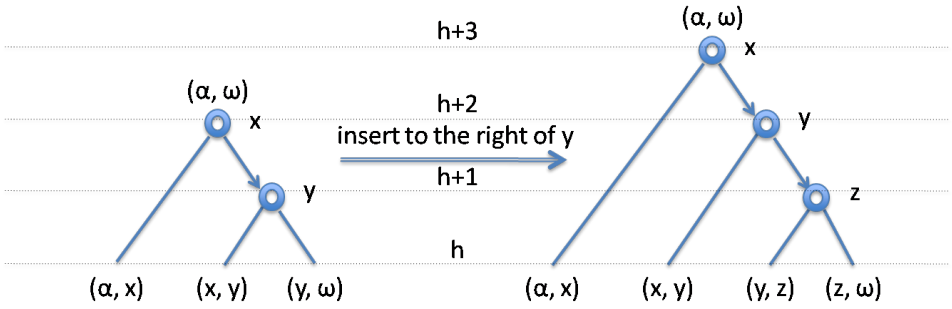
5 Inserting an Element

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

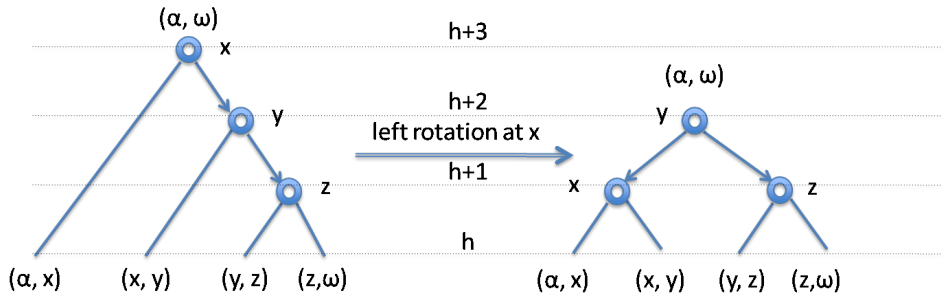
The main cleverness of the algorithm lies in analyzing the situations when we have to rebalance and need to apply the appropriate rotations to restore the height invariant. It turns out that one or two rotations on the whole tree always suffice for each insert operation, which is a very elegant result.

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 element 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.

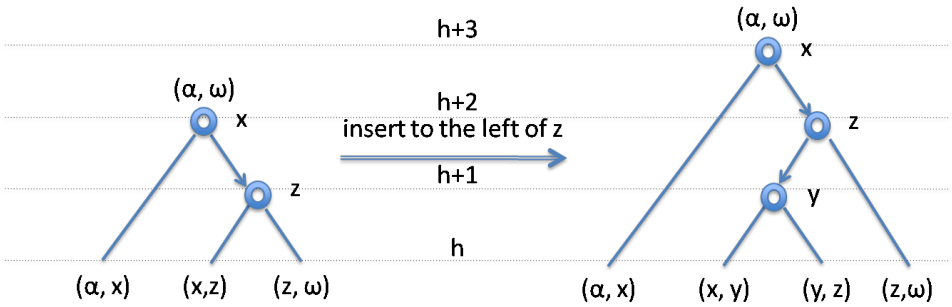


If the new right subtree has height $h + 2$, either its right or its left subtree must be of height $h + 1$ (and only one of them — think about why). If it is the right subtree we are in the situation depicted on the right above (and on the left below). While the trees (α, x) and (x, y) must have exactly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

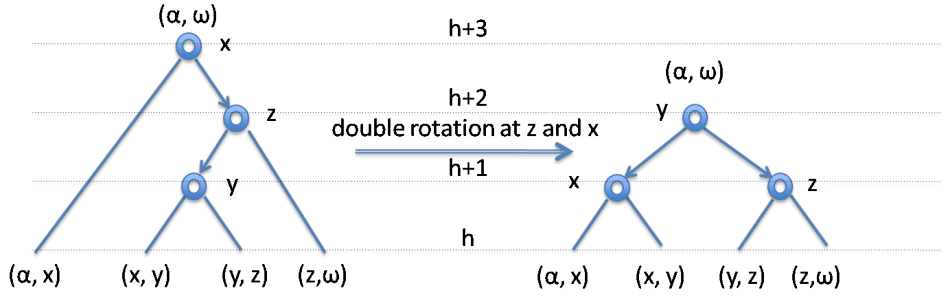


We fix this with a left rotation at x, the result of which is displayed to the right. Because the height of the overall tree is reduced to its original $h + 2$, no further rotation higher up in the tree will be necessary.

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). In this case, a single left rotation alone will not restore the invariant (see Exercise 1). 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 element on the left (see Exercise 4).

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.

6 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:

```
1  typedef struct tree_node tree;
2  struct tree_node {
3    elem data;
4    int height;
5    tree* left;
6    tree* right;
7  };
8
9  /* height(T) returns the precomputed height of T in O(1) */
10 int height(tree *T) {
11   return T == NULL ? 0 : T->height;
12 }
```

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`.

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. 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. Look at the code for details.

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

```
13 bool is_specified_height(tree* T) {
14   if (T == NULL) return true;
15   int computed_height = max(height(left), height(right)) + 1;
16   return computed_height == T->height
17     && is_specified_height(T->left)
18     && is_specified_height(T->right);
19 }
20
21 bool is_balanced(tree* T) {
22   if (T == NULL) return true;
23   return abs(height(T->left) - height(T->right)) <= 1
24     && is_balanced(T->left) && is_balanced(T->right);
25 }
```

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

```
27 bool is_tree(tree *T) {
28   return is_ordered(T, NULL, NULL)
29     && is_specified_height(T)
30     && is_balanced(T);
31 }
```

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

```
1 tree* leaf(elem e)
2 //@requires e != NULL;
3 //@ensures is_tree(\result);
4 {
5   tree *T = alloc(tree);
6   T->data = e;
7   T->height = 1;
8   return T;
9 }
```

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

7 Implementing Insertion

The code for inserting an element 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.

```
33 tree* tree_insert(tree* T, elem x)
34 //@requires is_tree(T) && x != NULL;
35 //@ensures is_tree(\result);
36 {
37   if (T == NULL) return leaf(x);
38
39   int r = elem_compare(x, T->data);
40   if (r == 0) {        // x == T->data
41     T->data = x;
42   } else if (r < 0) {  // x < T->data
43     T->left = tree_insert(T->left, x);
44     T = rebalance_left(T);
45   } else if (r > 0) {  // x > T->data
46     T->right = tree_insert(T->right, x);
47     T = rebalance_right(T);
48   }
49
```

```
50  return T;
51 }
```

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.

```
27 tree* rebalance_right(tree* T)
28 //@requires T != NULL && T->right != NULL;
29 //@requires is_tree(T->left) && is_tree(T->right);
30 //@ensures is_tree(\result);
31 {
32  if (height(T->right) - height(T->left) == 2) {
33   if (height(T->right->right) > height(T->right->left)) {
34    T = rotate_left(T);
35   } else {
36    //@assert height(T->right->left) > height(T->right->right);
37    T->right = rotate_right(T->right);
38    T = rotate_left(T);
39   }
40  } else {
41   fix_height(T);
42  }
43  return T;
44 }
```

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 (difficult) Exercise 5. 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.

8 Experimental Evaluation

We would like to assess the asymptotic complexity and then experimentally validate it. It is easy to see that both insert and search 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 elements 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 elements 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.

Exercises

Exercise 1. *Show that in the situation on page 8 a single left rotation at the root will not necessarily restore the height invariant.*

Exercise 2. *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 3. *Show that left and right rotations are inverses of each other. What can you say about double rotations?*

Exercise 4. *Show the two cases that arise when inserting into the left subtree might violate the height invariant, and show how they are repaired by a right rotation, or a double rotation. Which two single rotations does the double rotation consist of in this case?*

Exercise 5. *Strengthen the invariants in the AVL tree implementation so that the assertions and postconditions which guarantee that rebalancing restores the height invariant and reduces the height of the tree follow from the preconditions.*