CSci 4041: Algorithms and Data Structures
Balanced Binary Search Trees (AVL trees). Part 2

Last revision March 21, 2020

This is part two of a two-part series about balanced binary search trees. Here's the agenda.

Announcements.
Balance factors.
Inserting key-value pairs into an AVL tree.
A big complicated example.
A big complicated algorithm.

Announcements. First, a comment about names and terminology. Some students who have read Part 1 of the AVL tree lectures tell me that the names I've used for transformations aren't consistent with the names which other authors use. For example, I said *left* where other authors say *right,* etc. Possibly. The names I've used are consistent with those used by Niklaus Wirth in *Algorithms + Data Structures = Programs,* the textbook I mentioned last time. It's possible Wirth got the names wrong (unlikely because he's very smart). But even if he did, I'm going to follow his naming conventions even if they're different from everyone else's. If I ask about AVL transformations on an assignment, or an examination, I'll phrase the question in such a way that the names of the transformations don't matter.

Second, what should you remember from this lecture? The algorithms for AVL trees are very complex—that doesn't mean they're difficult, it just means they're detailed. For this course, you need to know only how the four rotation transformations work, and how they're used to reduce the height of a binary search tree. You don't have to know how the algorithms work. However, for students who want to go deeper, I've included pseudocode for an algorithm that adds a key-value pair to an AVL tree while keeping the tree balanced.

Balance factors. We saw in the last lecture that AVL trees have the *AVL tree property,* which says every node has subtrees whose heights differ by at most 1. (We measure heights either by counting links or by counting nodes.) However, inserting a new node into an AVL tree may violate this property, by increasing a subtree's height. If the property is violated, then we say the tree is *unbalanced,* so we must *rebalance* it, by applying the transformations discussed in the previous lecture.

How do we know which transformations to apply? How do we know where in the tree to apply them? We keep a *balance factor* in each node. The balance factor is defined as the height of the node's right subtree, minus the height of its right subtree. Since these heights can differ by at most 1, the balance factor is always either −1, 0, or +1. It could therefore be represented as a two-bit binary number, as shown in the following table.

BITS	BALANCE
`11`	−1
`00`	0
`01`	+1

Recall that each transformation lowers the height of a node's left or right subtree. If we look at a node's balance factor, then we can tell which of its subtrees has too great a height. Then we can apply the transformation to lower that height.

An algorithm for inserting key-value pairs. We'll demonstrate how this works by sketching an algorithm that inserts a key k and its value v into an AVL tree. The insertion algorithm will be described first in English, and later in pseudocode. Here's the English version.

1. Starting at the root, descend the tree using the usual BST insertion algorithm (such as BST-PUT from an earlier lecture), looking for a node with key k. Remember the path we took through the tree, because we may have to retrace it later.

2. If we find a node with key k, then change its value to v, and stop. No rebalancing is needed.

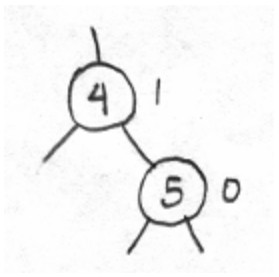
3. If we don't find the node, then make a new node with k and v, whose left and right subtrees are empty. Insert the new node into the tree according to the insertion algorithm. It always goes under an existing node, in either the left or right subtree.

4. Rebalance the tree. Visit nodes in reverse order, along the path we remembered from step 1. Use the balance factors of the nodes along that path to decide which transformations should be applied to rebalance the tree, and apply them. Stop when the balance factors say that no more rebalancing is needed.

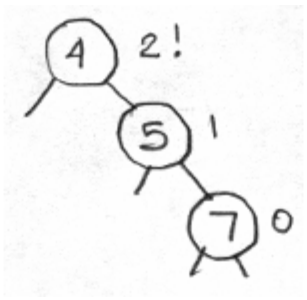
It's also possible to design a deletion algorithm for AVL trees using the same principles as the insertion algorithm, but we won't do that here.

A big complicated example. The English description of the algorithm is somewhat vague, but it might be made clearer by an example, based on one in Wirth's book. The example uses all four transformations, so you can see how they work. Recall that the single transformations need two nodes, *A* and *B,* along with three subtrees, α, β and γ. The double transformations need three nodes, *A, B,* and *C,* along with four subtrees, α, β, γ, and δ.

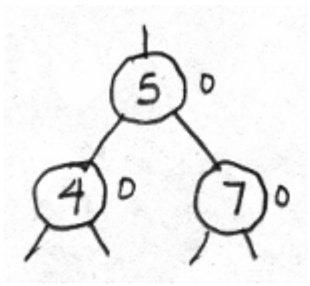
For simplicity, we'll assume keys are integers, and we won't show values at all. We'll start with an AVL tree that has two nodes. The root node has key 4, and the node in its right subtree has key 5. The numbers next to the nodes are balance factors, which in this case show that the tree is balanced.



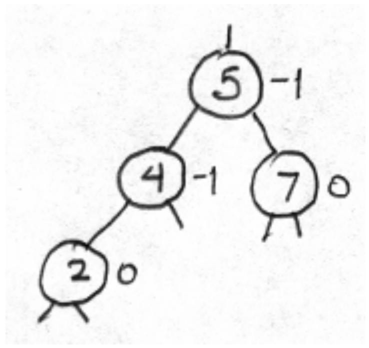
If we add a node with key 7, then it must go into the right subtree of the node with key 5. That gives the node with key 4 a balance factor of 2, which is impossible, because balance factors must be either −1, 0, or +1. It has an exclamation point (!) to indicate this.



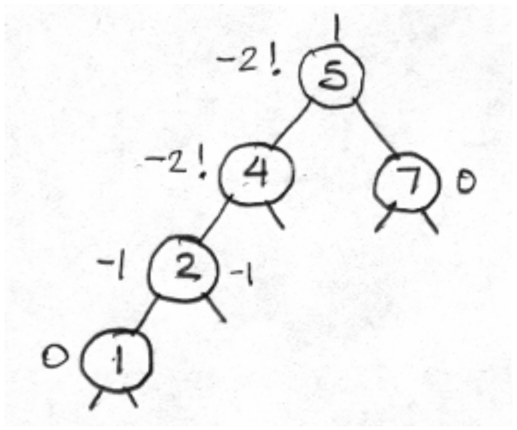
We can restore the balance by doing a single right right rotation, where *A* is node 5, *B* is node 4, α is node 4's empty left subtree, β is node 5's empty left subtree, and γ is the subtree rooted at node 7. The result is this balanced tree.



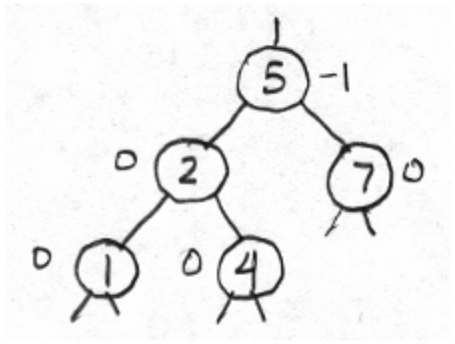
Now suppose we insert a node with key 2. It must go into the left subtree of the node with key 4. This doesn't alter the tree's balance, because all its balance factors are either −1, 0, or +1.



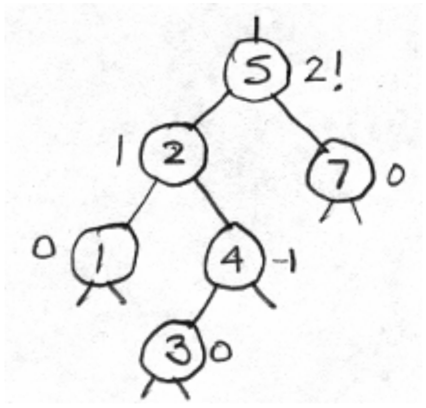
Next, we'll insert a node with key 1, in the left subtree of the node with key 2. The tree is unbalanced again, as we can see from the balance factors of node 4 and node 2.



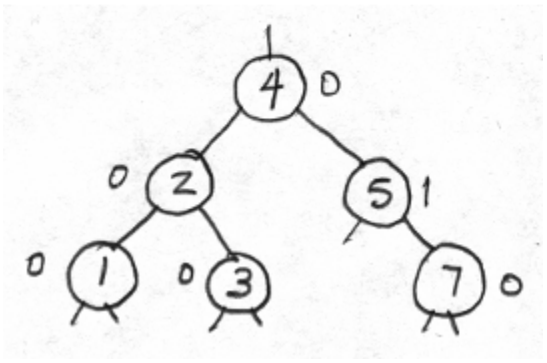
This time we'll restore the balance by a single left left rotation, where *A* is node 2, *B* is node 4, α is the subtree rooted at node 1, β is the empty right subtree of node 2, and γ is the empty right subtree of node 4. The rotation gives us this balanced tree.



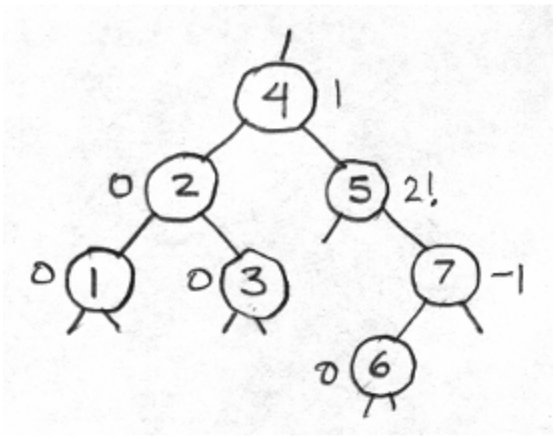
Just to be difficult, let's add a node with key 3. It goes into the left subtree of the node with key 4. This makes the tree unbalanced again.



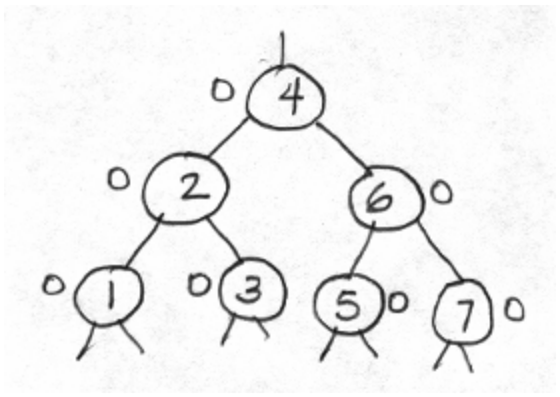
To restore the balance, we use a double left right rotation, where *A* is node 2, *B* is node 4, *C* is node 5, α is the subtree rooted at node 1, β is the subtree rooted at node 4, γ is node 4’s empty right subtree, and δ is the subtree rooted at node 7. We get the following balanced tree.



Finally, we’ll insert a node with key 6, which must go into the left subtree of the node with key 7. This makes the subtree rooted at node 5 unbalanced.



It takes a double right left rotation to fix this, where *A* is node 7, *B* is node 6, *C* is node 5, α is node 5’s empty left subtree, β is node 6’s empty left subtree, γ is node 6’s empty right subtree, and δ is node 7’s empty right subtree. Here’s the result. By coincidence, it’s perfectly balanced!



It's important to realize, however, that the AVL insertion algorithm we've just discussed doesn't guarantee to give us a perfectly balanced tree in the end—we just got lucky this time. All it guarantees is that no node has subtrees whose heights differ by more than 1.

A big complicated algorithm. We can also write the insertion algorithm in pseudocode. The procedures shown here from from Wirth's book, where they were written in the (dead) programming language Pascal; I translated them. You don't have to know how this pseudocode works, but I'll talk about it a little anyway.

Each AVL tree node has five slots. The slot *bal* is the node's balance factor: either −1, 0, or +1. The slot *key* is the node's key, and may be anything that can be compared by '<', '==', or '>'. The slot *value* is the value that corresponds to the key, and may be anything. The slots *left* and *right* point to nodes, or to NIL.

The call *insert(p, k, v)* changes the AVL tree rooted at *p* so it has a node with key *k* and value *v.* It does this either by adding a new node to *p* (which may require rebalancing), or by changing an existing node with key *k* so it has value *v* (which doesn't require rebalancing). Parameters with **var** prefixes are *variables,* so changes to them are seen by the caller. (Cormen's pseudocode doesn't use **var** parameters, but we introduced them in the lecture on BST's.)

```
insert(var p, k, v)
  h = false
  inserting(h, p, k, v)
```

The procedure *inserting* does all the work for *insert.* The call *inserting*(*h, p, k, v*) changes the AVL tree rooted at *p* so it has a a node with key *k* and value *v.* Its parameter *h* is a Boolean variable, set to *true* if *p*'s height has changed, so *p* may require rebalancing. It's set to *false* otherwise. The call *newNode*(*b, k, v*) returns a pointer to a new node, whose *bal* slot is *b,* whose *key* slot is *k,* whose *value* slot is *v,* and whose *left* and *right* slots are NIL.

```
inserting(var h, var p, k, v)
  if p == nil
    p = newNode(0, k, v)
    h = true
  else if k < p.key
    inserting(h, p.left, k, v)
    if h
      leftRebalance(h, p)
  else if k > p.key
    inserting(h, p.right, k, v)
    if h
      rightRebalance(h, p)
  else
    p.value = v
    h = false
```

Note that after each recursive call to *inserting,* we check if *h* is *true,* and call a procedure to rebalance the tree if it is. This trick uses the recursion stack to trace a path back through the tree in reverse—step 4 in the English version of the algorithm.

The remaining procedures are *inserting*'s helpers. The **case** statement acts like a switch statement in C-like languages, but it doesn't need break statements. The procedures on the left are symmetric with respect to those on the right. Each +1 on one side corresponds to a −1 on the other side. Each slot *left* on one side corresponds to a slot *right* on the other side. If you know how one side works, then you also know how the other side works.

```
leftRebalance(var h, var p)
  case p.bal
   +1: p.bal = 0
       h = false
    0: p.bal = −1
   −1: if p.left.bal == −1
         singleLeftLeftRotation(p)
       else
         doubleLeftRightRotation(p)
       p.bal = 0
       h = false
```

```
rightRebalance(var h, var p)
  case p.bal
   −1: p.bal = 0
       h = false
    0: p.bal = +1
   +1: if p.right.bal == +1
         singleRightRightRotation(p)
       else
         doubleRightLeftRotation(p)
       p.bal = 0
       h = false
```

```
singleLeftLeftRotation(var p)
  p1 = p.left
  p.left = p1.right
  p1.right = p
  p.bal = 0
  p = p1
```

```
singleRightRightRotation(var p)
  p1 = p.right
  p.right = p1.left
  p1.left = p
  p.bal = 0
  p = p1
```

```
doubleLeftRightRotation(var p)
  p1 = p.left
  p2 = p1.right
  p1.right = p2.left
  p2.left = p1
  p.left = p2.right
  p2.right = p
  if p2.bal == −1
    p.bal = +1
  else
    p.bal = 0
  if p2.bal == +1
    p1.bal = −1
  else
    p1.bal = 0
  p = p2
```

```
doubleRightLeftRotation(var p)
  p1 = p.right
  p2 = p1.left
  p1.left = p2.right
  p2.right = p1
  p.right = p2.left
  p2.left = p
  if p2.bal == +1
    p.bal = −1
  else
    p.bal = 0
  if p2.bal == −1
    p1.bal = +1
  else
    p1.bal = 0
  p = p2
```

I'll repeat: *you don't have to know how this works!* It's included here only to show that we really can write a procedure that changes an AVL tree and rebalances it. We can show that this algorithm needs O($\log_2 n$) key comparisons for a tree with *n* nodes, but I won't demonstrate that either.

This is the last lecture on AVL trees. In the next lecture, we'll begin talking about *dynamic programming,* starting with *memoization.* We'll eventually see a dynamic programming algorithm that constructs *optimal binary search trees,* or BST's whose keys have probabilities.