15-122: Principles of Imperative Computation

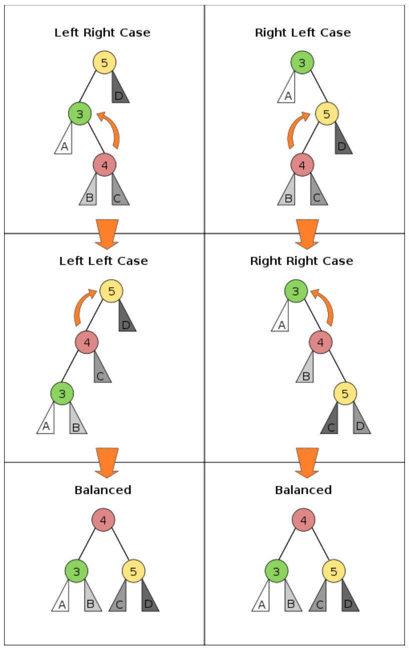
Recitation 19 Solutions — **Nivedita Chopra, Josh Zimmerman**

AVL Trees

- AVL trees are balanced binary search trees
- The insert and delete operations are $O(\log n)$ on AVL trees, since they are balanced. This is unlike regular binary search trees, where the worst case is $O(\log n)$
- Height of a tree can be defined in two ways. They are equivalent, so you should keep in mind just the one that makes more sense to you.
  (a) The height of a tree is the maximum number of nodes from the root to a leaf, inclusive.
  (b) (recursive) The empty tree has height 0 and any other tree has height $1 + \max(hl, hr)$ where hl and hr are the heights of the left and right children of the tree, respectively.
- AVL trees maintain a height invariant. The height invariant states that for every node in the tree, the height of the left and right subtrees differ by at most 1, or in other words, if the left and right subtrees have height hl and hr, respectively, $|hl - hr| \leq 1$. This lets us maintain balance of the tree, and so ensures that we'll have at worst $O(\log n)$ lookup time.
- In an AVL tree, we insert an element much like we would in a BST, but we then check to see if the height invariant is violated and rebalance the tree if necessary.

Rotations

- Rebalancing the tree is done by rotations. Since we rebalance immediately when the height invariant is violated, the difference between the heights of the children of a node is at most 2
- Thus there are exactly four cases for rotations (the figure below is from Wikipedia).



- Two of the cases (the ones the diagram calls "Left Right" and "Right Left") can be transformed into the other two with a single rotation, and those two can be made balanced with a single rotation.
- This means that we *never* need more than 2 rotations to restore balance an AVL tree after inserting an element.
- Since rotation is a constant time operation, insertion into an AVL tree is only at worst a constant amount slower than insertion into a BST
- Note that we do these rotations at the *lowest* violation of the height invariant in the AVL tree.

Playing with AVL trees!

Use the visualization at `http://www.cs.usfca.edu/~galles/visualization/AVLtree.html` to insert these keys into the tree in the following order:

$$1, 2, 5, 3, 4$$

Then delete the keys 2 and 4.

Discussing the code for rotations

Now that we have a conceptual understanding of AVL trees, let's talk about the implementation details.

To keep track of the height in an efficient manner, we add a `height` field to the data structure:

```
struct tree_node {
  elem data;
  int height;
  struct tree_node *left;
  struct tree_node *right;
};
```

Note that we now need to do more work in the `is_balanced` function, since we need to check that `height` accurately describes the height of each node of the tree (if it didn't, we wouldn't be able to guarantee that the tree was balanced, since we use the `height` variable when checking balance in our `rebalance_right` and `rebalance_left` functions).

First, let's look at `rotate_right`. (`rotate_left` is exactly symmetric)

```
tree *rotate_right(tree *T) {
  REQUIRES(is_ordtree(T));
  REQUIRES(T != NULL && T->left != NULL);
  tree *root = T->left;
  T->left = root->right;
  root->right = T;
  fix_height(root->right); /* must be first */
  fix_height(root);
  ENSURES(is_ordtree(root));
  ENSURES(root != NULL && root->right != NULL);
  return root;
}
```

Let's go through some sketches to help illustrate what this function is doing. If you're reading through this after recitation, I encourage you to sketch some diagrams on a piece of paper or a whiteboard to help you understand the code. (Note that `rotate_left` is symmetric to `rotate_right`, so if you understand one you should be able to understand the other.)

Now, let's look at our `rebalance_right` code, on an intuitive level. (The `rebalance_left` code is similar, so we're leaving it out here. But we highly encourage you to work with it and make sure you understand it, as the code for AVL trees is tricky.)

```
 1 tree *rebalance_right(tree *T) {
 2   REQUIRES(T != NULL);
 3   REQUIRES(is_avl(T->left) && is_avl(T->right));
 4   /* also requires that T->right is result of insert into T */
 5
 6   tree *l = T->left;
 7   tree *r = T->right;
 8   int hl = height(l);
 9   int hr = height(r);
10   if (hr > hl + 1) {
11     ASSERT(hr == hl + 2);
12     if (height(r->right) > height(r->left)) {
13       // We're in the "right right" case in the diagram from Wikipedia
14       ASSERT(height(r->right) == hl + 1);
15       T = rotate_left(T);
16       // Note that hl + 2 == hr here
17       ASSERT(height(T) == hl+2);
18     }
19     else {
20       // We're in the "right left" case in the diagram from Wikipedia
21       ASSERT(height(r->left) == hl + 1);
22       /* double rotate left */
23       T->right = rotate_right(T->right);
24       T = rotate_left(T);
25       // Note that hl + 2 == hr here
26       ASSERT(height(T) == hl+2);
27     }
28   }
29   else {
30     // the tree is already balanced, so just update the height
31     ASSERT(!(hr > hl+1));
32     fix_height(T);
33   }
34   ENSURES(is_avl(T));
35   return T;
36 }
```

Checkpoint 0

Give an informal explanantion of why the rebalance right function works. (We're missing a precondition, as described by the comment on line 4, so we won't do a formal proof here)

Solution:

NOTE: This is *NOT* a formal proof and we would not accept it as a proof. It is solely to help you get the intuition for how AVL rotations work.

If we enter the branch of the conditional that starts on line 10, we know the tree is unbalanced, since the height of the right subtree is more than 1 larger than the height of the left subtree.

Now, we have two cases: either the right sub-subtree has a larger height than the left sub-subtree, or not.

If the right sub-subtree is longer, then we simply need to rotate left to even things out, as we can see intuitively in the wikipedia picture, since the left sub-subtree doesn't cause any problems by being where it is but the right sub-subtree does. Note that this means the height of the whole tree is now equal to the height of the right subtree, since the height of the left subtree was less than the height of the right subtree.

Otherwise, the left sub-subtree is longer, and so it's causing the problems. If we rotate the right subtree right, then we're in the "right right" case, and the left sub-subtree is no longer causing a problem, so the same reasoning as above applies.

Discussing the code for insertion

For reference, the AVL insertion function calls `rebalance_left` and `rebalance_right` in exactly the same place where the height-tracking BST implementation called `fix_height`. The assignment back to `T` is needed because we might be changing the root of the tree with a rotation or double rotation.

```
tree *tree_insert(tree *T, elem e)
{
   REQUIRES(is_ordtree(T));
   REQUIRES(is_specified_height(T));
   REQUIRES(e != NULL);

   if (T == NULL) {
      /* create new leaf with the data e */
      T = xmalloc(sizeof(struct tree_node));
      T->data = e;
      T->height = 1;
      T->left = NULL;
      T->right = NULL;
   }
   else {
      int r = key_compare(elem_key(e), elem_key(T->data));
      if (r == 0) {
         T->data = e;  /* modify in place */
      }
      else if (r < 0) {
         T->left = tree_insert(T->left, e);
         T = rebalance_left(T); /* also fixes height */
      }
      else {
         ASSERT(r > 0);
         T->right = tree_insert(T->right, e);
         T = rebalance_right(T); /* also fixes height */
      }
   }

   ENSURES(is_ordtree(T));
   ENSURES(is_specified_height(T));
   return T;
}
```