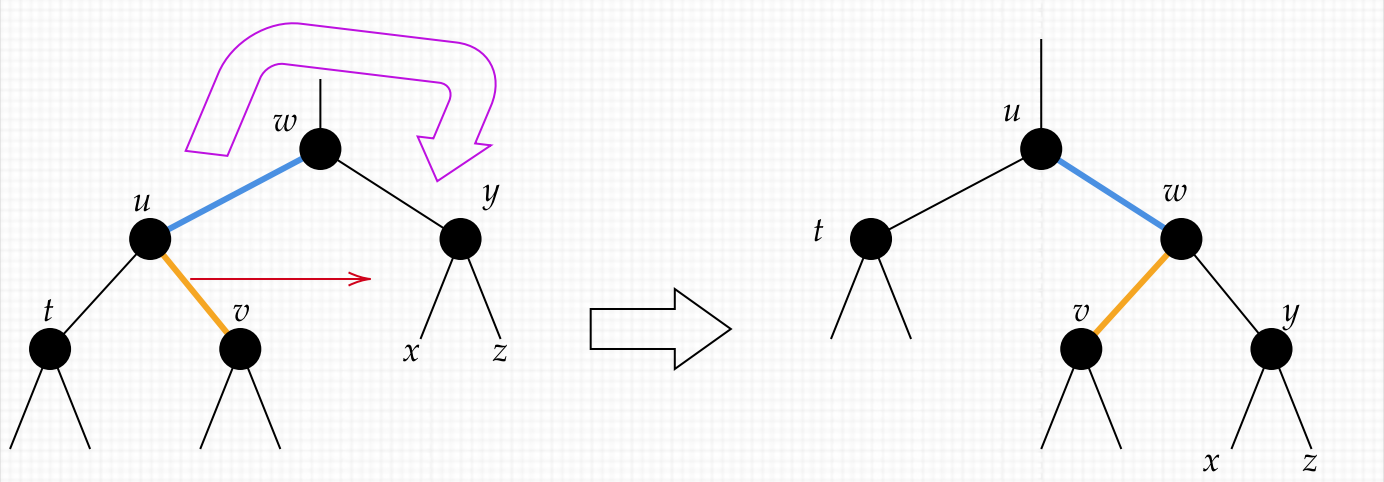
CSE250 Week 14: Balanced Binary Search Trees And Other Trees (text ch. 21; 19 & 20 parts)

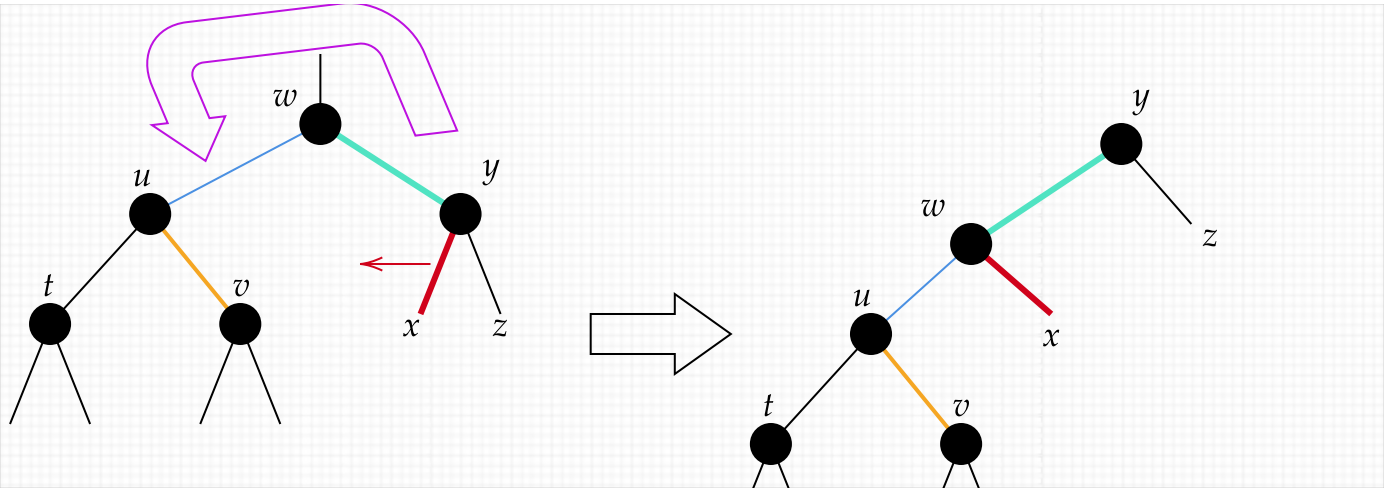
The idea of *tree balancing* is is to expend additional work on many insertions (less so on removals) to keep the tree compact on-the-fly. This increases the principal constant but the asymptotic order stays $O(\log n)$. Indeed, self-balancing BSTs guarantee $O(\log n)$ time for all of `find`, `insert`, and `remove`. Generally, they balance the *heights* of the left and right subtrees at any node. This is often, but not always, accompanied by a similar balance in the *size* of the subtrees, meaning their total number of nodes. The heights are what matters most to the `find` operation.

Rotations

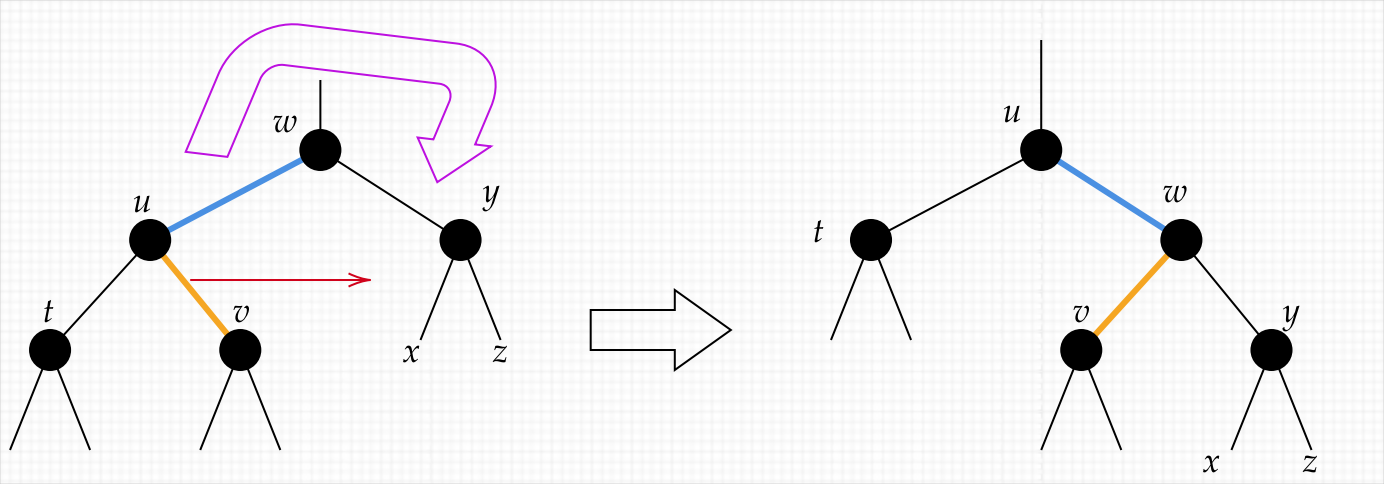
The key extra operation is **rotation**, which can be **left** or **right** at a node w. Rotations are employed by several balancing schemes. The basic idea is that if u is the left child of w, then the right subtree of u would be legal as the left subtree of w. So we can do a **right rotation** as follows:



A **left rotation** at u in the right-hand diagram would invert the process. It is more interesting to see the effect of a left rotation at w in the original tree:



For code, let's look again at the first diagram.



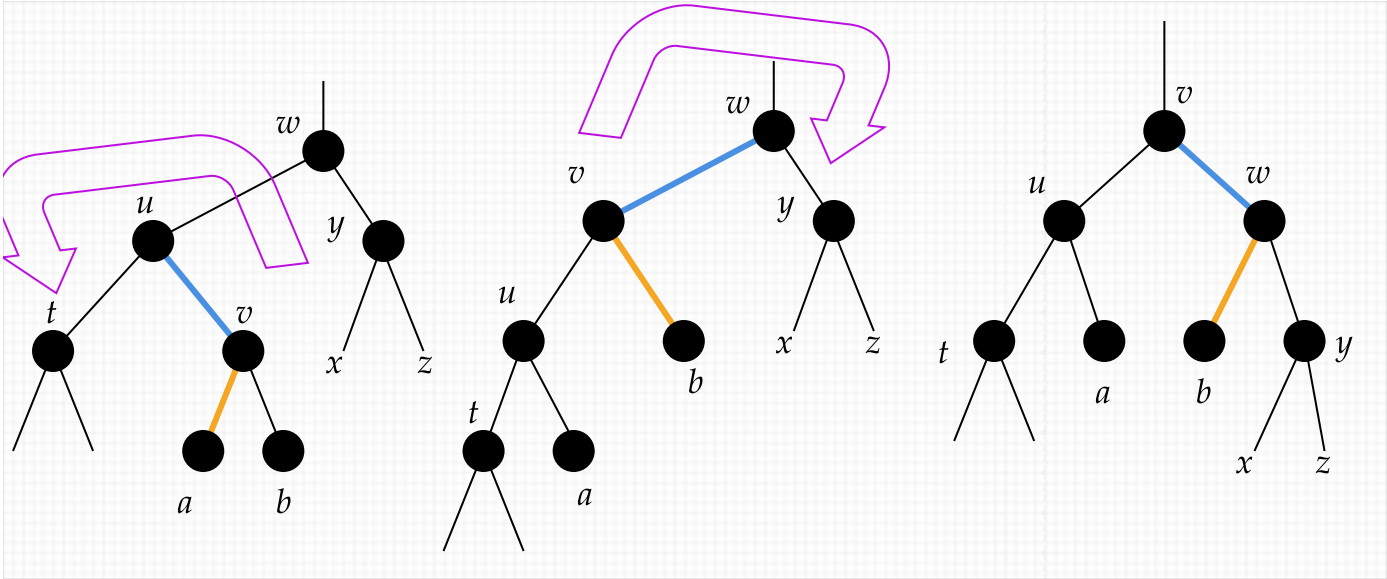
Here u, v, and w all change their `parent` links, `w.parent` changes whichever link(s) went to `w` to go to `u` instead, `u.right` becomes `w`, and `w.left` becomes `v`. Only `u` and `w` can change their heights. The code simply executes this:

```
private def rotateRight(w: Node): Unit = {
   assert(w != endSentinel, "Attempt to rotate at end")
   var u = w.left
   assert(u != endSentinel, "Attempt to updraft an empty left child")
   var v = u.right   //OK if this is endSentinel
   var p = w.parent  //OK if w is root and p is endSentinel
   //first link u to p, twice if it becomes new root.
   if (w == p.left) { p.left = u }
   if (w == p.right) { p.right = u }
   u.parent = p
   u.right = w
   w.parent = u
   w.left = v
   v.parent = w
   w.height = 1 + Math.max(v.height,w.right.height)
   u.height = 1 + Math.max(u.left.height, w.height)
   root = endSentinel.left
}
```

The last line is curious, but it is possible that w was the root and u got rotated into its place. The earlier code will then take p to be the end sentinel and make u to be both its left and right child. Since the invariant that `root` is `endSentinel.left` (as well as `endSentinel.right`) is not otherwise disturbed, we might as well just re-enforce it rather than test for `w ≡ root`.

The code for rotateLeft is mirror-image: just switch uses of `left` and `right`. There must at least be the two real nodes w and u in the tree, else an exception is raised.

It is interesting to see the effect of a left rotation at u followed by a right rotation at w:

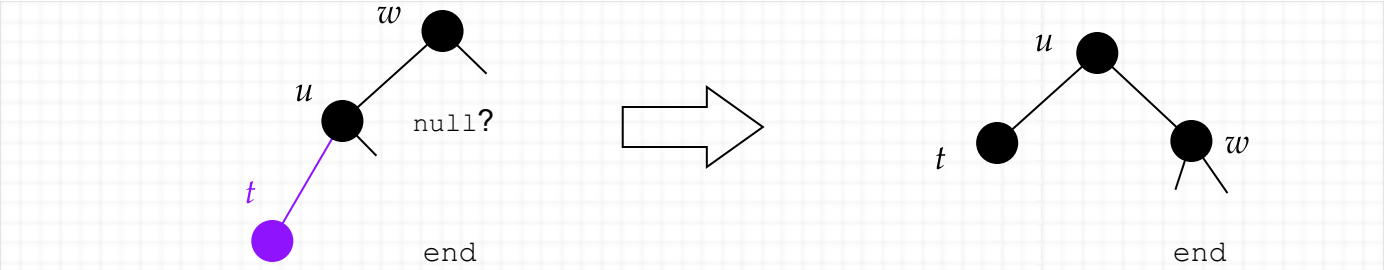


Visually, what happened is the "inner grandchild" v of the root w got hoisted up to be the new root, and the children of v become its inner grandchildren. Some animations, e.g. David Galles at the University of San Francisco's Data Structure Visualizations, show this sequence as one operation. You could call this "left hoist" and code it separately in a way that saves a couple of link changes over code that composes two rotations, and similarly code "right hoist" in mirror image. But this was not considered a big deal even before we had parallel-execution processor cores.

AVL Trees

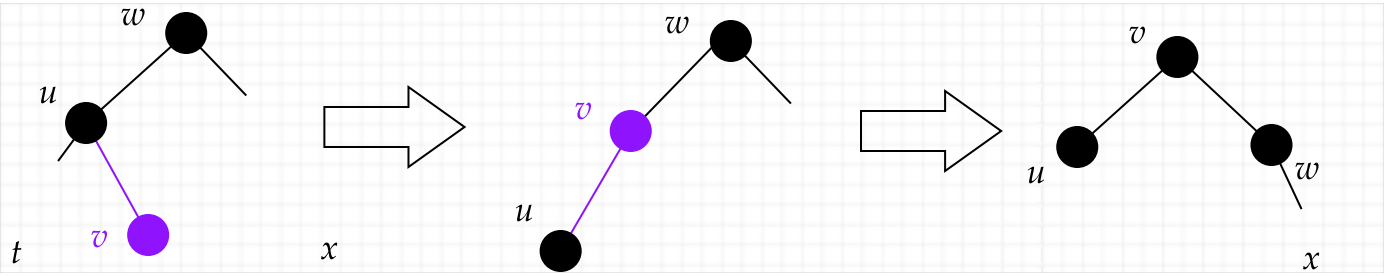
The first scheme for using rotations to preserve balance in trees was devised by the Soviet mathematicians Georgy Adelson-Velsky (who was also a chess programmer) and Evgenii Landis in 1962. It enforces at each node u the **balance condition** $B(u)$ that the heights of the left and right subtrees of u differ by at most 1. This ensures that the height of a whole tree of n nodes is at most about $1.44 \log_2 n$, which is well within our $2 \log_2(n)$ limit for being "balanced."

An important base case happens with just three nodes:



The text in chapter 21 seems to switch away from how in chapter 16 it defined the height of a leaf to be 0 (see Fig. 1 on page 494, say). If the height of the leaf t is 0 then the height of u is 1. Then in order to say that the node w is unbalanced (i.e., $B(w)$ is false), we would have to say that the "height of null" is -1. The text also (wisely, IMPHO) switches to using a bottom sentinel node it calls `AVLEmpty`, which is much the same as how we've been already using `endSentinel` in the "CL&R tree." It gives this node a `height` of 0 and thus switches over to giving leaves a `height` of 1. We will do the same. Then u counts as height 2, and the right child of w---which exists in the form of `endSentinel`---has height 0. So $B(w)$ does indeed fail, and we fix it by the right rotation shown.

There is a further wrinkle that the text and several other sources I find seem to duck. (I've always taught **red-black trees** before even when the text did AVL. In fact, RBTs are the implementations used by Scala too, but the fact is heralded more in the C++ Standard Template Library.) Suppose we just now inserted t. The height of w has become 3 in our new reckoning, but before we inserted t, it was 2. After the rotation, the height of the new root u is again 2. This implies that *the heights of all nodes p above u will not change from before the insertion*, nor those of any other nodes. On the presumption that the balance condition $B(p)$ held in all other nodes, we do not need to do any further work. If t were a right child, we get the double-rotation scenario:



Here again, the height of the root reverts to what it was before the insertion, so we can stop. This also holds when the first imbalance is discovered higher up in the tree. We can state:

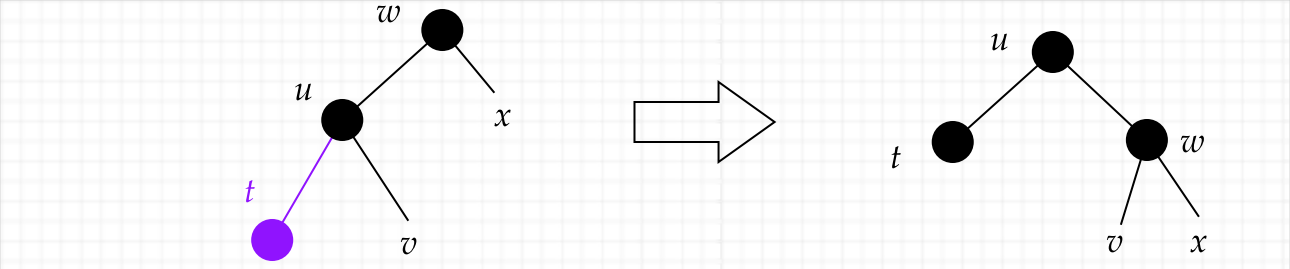
AVL Insertion Lemma. One insertion into an AVL tree requires at most two rotations to re-balance the tree. (Alternatively you can say: one rotation or one "hoist.")

Proof Sketch: The base case reasoning extends to general cases because the heights of the (subtrees
rooted at the) other child x of w, the other child t of u, and any children of v, will not change in either rotation. The text shows this by abstracting those trees as triangles in the diagrams. Suppose the new insertion went left at w (the case of going right is mirror-image). Then we need to rotate when

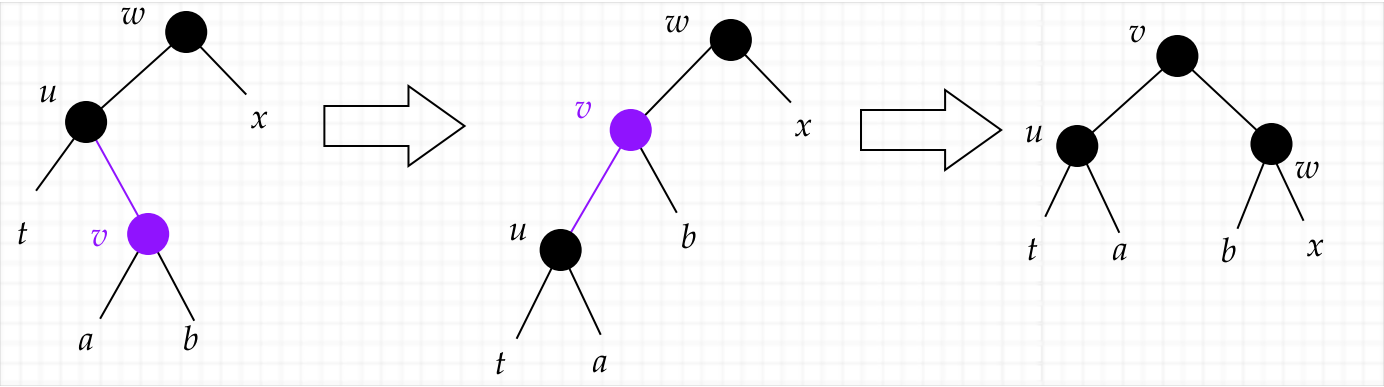
$u.height = x.height + 1$ before the insertion *and* when $u.height$ will increase after it. There are two subcases, one giving the single rotation and one for the hoist. In both subcases, the goal completing the proof is:

> **The height of the node (u or v) replacing w after the insertion is the same as what the height of w was before the insertion. Thus no further heights of subtrees are changed.**

1. After the insertion, $t.height \geq v.height$. Since w is the first unbalanced node, this means $t.height = v.height$ or $t.height = v.height + 1$. Either way, if the insertion caused the height of t to go up too, then the height of the root above t is unchanged from before the insertion, and the height of the root above v remains the same. If the insertion went below v, the fact that the height of the root u above v being the same as w above v works here too.



2. After the insertion, $v.height = t.height + 1$. This is when we need the double rotation. But the essence is that the first of these two rotations sets up subcase 1 with v in place of u, u in place of t, and the right child b of v playing the new "inner grandchild" role. So the logic of subcase 1 finished the job. ☒



AVL Removal

Unfortunately, the case of removing an element is not so nice---it can require $O(\log n)$ rotations. A concrete example where rotations percolate all the way to the root is the tree obtained by inserting 8, 5, 9, 3, 6, 11, 2, 4, 7, 10, 12, 1 in that order and then removing 10 (using the inorder predecessor as the "victim" as the Galles visualization does).

The text's code for insertion is recursive on immutable nodes, so it uses the policy of computing $O(\log n)$ new nodes along the path of insertion and stitching them into the tree. It implicitly uses this theorem by how it needs to change the height only for the new editions of those nodes that are involved in the one or two rotations. It is IMPHO hard to read at first go. The way removals can need $O(\log n)$ rotations is apparent in the text *only* from the recursive use of `rotate` in recursive calls to `removeMax` in line 107 on page 592.

Thus, when rotating after a removal, one needs to continue checking balance going up the tree via parent links. Although this is unnecessary after an insertion, the steps of checking balances are cheap---the fact that further imbalances won't happen still saves considerable time. Hence, we can code a single "fix-up" routine to use both after insertion and after deletion. This makes the following code mode modular than in the text.

Code for AVL Rebalancing

The following is taken from the functioning file `BSTAVL.scala`, our last addition to the "ISR" repository.

```
/** AVL rebalancing routine.  Code works from bottom up procedurally.
    Math fact: after insert, an "else" branch with rotation will be taken only once.
    But after a deletion of an elbow or leaf, rotations can need to propagate.
 */
private def fixUp(w: Node): Unit = {
   if (w == endSentinel) return     //we fixed the root and recursed on root.parent
   //else
   w.height = 1 + Math.max(w.left.height, w.right.height)   //now propagate updates
   if (Math.abs(w.left.height - w.right.height) <= 1) {     //nothing to do here
      fixUp(w.parent)
   } else if (w.left.height - w.right.height > 1) {
      val u = w.left
      if (u.left.height >= u.right.height) {  //single-rotation case
         rotateRight(w)                       //makes u the new root of subtree
         fixUp(u.parent)
      } else {                                //double-rotation case
         val v = u.right
         rotateLeft(u)
         rotateRight(w)                       //now v is root where w was
         fixUp(v.parent)
      }

   //--------------Rest of the code is merely mirror-image-----------------------

   } else {                                   //w.right.height - w.left.height > 1
```

```
      val u = w.right
      if (u.right.height >= u.left.height) {  //single-rotation case
         rotateLeft(w)                         //makes u the new root of subtree
         fixUp(u.parent)
      } else {                                 //double-rotation case
         val v = u.left
         rotateRight(u)
         rotateLeft(w)                        //now v is root where w was
         fixUp(v.parent)
      }
   }
}
```

That's it. It may seem weird that there is no statement updating the height of any node. The essence is that the calls of `rotateRight` and `rotateLeft` already update the heights of the pivot node of the rotation and its left or right child that rotates with it, while no other node changes its height. In fact, the code does not need to maintain the total numerical height, only whether the difference $u.left.height - u.right.height$ at a node u is -1, 0, $+1$, or "unbalanced". Thus, an AVL tree needs 2
extra bits of "state" information per node.

The changes to `insert` and `remove` are straightforward, but the former has a subtle point. The latter only needs showing the part of the code that de-links the "victim" node, and we just need to put in calls to `fixUp`:

```
def remove(loc: Iter): A = {
   ...
   if (loc.at.left == endSentinel) {   //its right subtree -> parent's new subtree
      val parent = loc.at.parent
      if (loc.at.right != endSentinel) { loc.at.right.parent = parent }
      if (loc.at == parent.left) {
         parent.left = loc.at.right    //overwrites loc.at, so loc becomes invalid
      } else {
         parent.right = loc.at.right
      }
      fixUp(parent)
   } else if (loc.at.right == endSentinel) { //left subtree -> parent's new subtree
      val parent = loc.at.parent
      if (loc.at.left != endSentinel) { loc.at.left.parent = parent }
      if (loc.at == parent.left) {
         parent.left = loc.at.left
      } else {
         parent.left = loc.at.left
```

```
      }
      fixUp(parent)
   } else {
      ...
   }
}
```

The call to `insert` returns an iterator, and the burning question is whether the iterator remains valid after nodes might get shifted around by `fixUp`:

```
/** REQ: Node of loc is a leaf or elbow, and insertion will not violate sortedness
    Note that we make this private so that the REQ need only be enforced internally.
 */
private def insert(item: A, loc: Iter, leftward: Boolean): Iter = {
   assert((leftward && loc.at.left == endSentinel)
             || (loc.at.right == endSentinel && !leftward),
            "Attempt to munge an existing node while inserting below " + loc())
   if (leftward) {
      loc.at.left = new Node(item, endSentinel, endSentinel, loc.at, 1)
      val itr = new Iter(loc.at.left)
      if (loc.at.right == endSentinel) { loc.at.height = 2 }    //base case
      fixUp(loc.at)
      return itr      //is this robust if the parent rotates right?
   } else {
      loc.at.right = new Node(item, endSentinel, endSentinel, loc.at, 1)
      val itr = new Iter(loc.at.right)
      if (loc.at.left == endSentinel) { loc.at.height = 2 }     //base case
      fixUp(loc.at)
      return itr      //still pointing to the new item?
   }
}
```

The answer is *yes*, because the rotations do not ***re-allocate*** any nodes. The objects for the nodes preserve their place in memory. The nodes move about in our diagrams because their links change but they do not actually migrate to new locations---not even like what happens to an array when it changes its capacity. So they do not invalidate the iterator `itr` created before the call. (This still feels more "squishy" in Scala, IMPHO, compared to C++ where the address-of operator `&` is explicit.)

Performance and Comparison With Other Trees

AVL Theorem: The height of an AVL tree of n nodes is at most $1.44... \log_2 n$. Thus `find`, `insert`, and `remove` all run in **guaranteed** $O(\log n)$ time, and an AVL tree can be built from n items in $O(n \log n)$ time regardless of the order of inserting the items.

This avoids the worst-case danger with basic BSTs. But there is basically no performance gain over the average-case performance of BSTs, such as if you "shuffle" chunks of data points before inserting them. If the time to build the data structure really matters (such as when it is trying to compete concretely with making a heap in $O(n)$ time), or if a lot of removals and insertions will alternate, then the extra time for rotations will be noticeable.

Scala's `TreeSet` and `TreeMap`, which double as the implementations of `SortedSet` and `SortedMap`, use a **red-black tree**, as does the C++ Standard Template Library. A red-black tree (RBT) needs only 1 extra bit of state information per node u: whether u is "red" or "black". The RBT maintains the following invariants:

1. For any node u, every path from u down to a leaf has the same number of black nodes. Put another way, the "black heights" of the two children of u are *exactly* balanced.
2. No red node can be a parent of a red node. (The root is always black.)

These invariants imply that a path down from the root can never have more red nodes than black nodes. Thus the worst imbalance is having one path having just k black nodes, and another alternating k black and k red. In consequence:

RBT Theorem: The height of a red-black tree of n nodes is never more than $2 \log_2(n+1)$.

This is right at the limit of our concrete definition of "balanced" and means that the worse-case RBT performance of `find` has a larger principal constant than in an AVL tree. The appeal of RBTs is that insertions and removals typically need fewer rotations to maintain the two invariants. In particular, inserting a red leaf of a black node needs no further action. (The code is a little more complicated than for AVL, however.)

The **splay tree** does *more* rotations than either AVL or RBT and does them even after `find`. The point is that those rotations move frequently-accessed elements closer to the root, so that the time to consult them again via find will move toward being $O(1)$ rather than merely $O(\log n)$. (Their main co-inventor, Daniel Sleator of CMU, went on to found the Internet Chess Club 30 years ago.)

The **tango tree** tries to work concretely faster in a streaming environment. There are other similar attempts to improve on AVL and RBT, but they all share a common drawback of linked structures.

Paging Improvements on Linked Structures... (text ch. 19 is only skin-deep)

Pun here on "paging" meaning "calling for" and on *memory paging*. The basic points is that linked lists, linked trees, and even hash tables do not assure **contiguity** of large chunks of data in memory. This affects the frequency of needing to load new memory pages while providing what clients see as the ability to demand "full random access" to data.

- A linked list might give good paging performance when freshly created with consecutive nodes and their items being allocated in adjacent memory cells. But as soon as nodes from other areas of system memory are inserted and/or some nodes get spliced out, performance can degrade. (Until, that is, "mirrored" garbage collection copies all reachable data afresh.)
- A BST subtree at a node u has a consecutive subsequence of items in sorted order. That subtree might all fit in one page.
- Similar considerations apply to a chunk of buckets in a hash table with chaining, so long as one does not rely on a predetermined order of the elements, and to open addressing with linear probing. But with quadratic probing, a probe sequence of a "bucket" can quickly jump out of an interval of the hash array.

Simply using a sorted array gives optimal paging performance when taking chunks of it at a time. But having a single large sorted array is uweildy for insertions and deletions. The popular middle-grounds for giving good scalable memory performance are hybrids of linked structures and *pretty large* pieces of arrays---way larger than what textbooks describe for simplicity.

- "AIOLI" and "BALBOA" are such hybrids. (Well, they will be if we have time to cover their cases with the target size r of their linked lists being greater than 1.)
- **B-Trees**. Covered only as a sidebar in the text at the very end of chapter 19. Hmmmmm....

Trees for Spatial Data (Ch. 20, if time allows)

- **Quadtrees** for efficiently abstracting 2D image data.
- Cousins for quadtrees for higher-dimensional images.
- ...