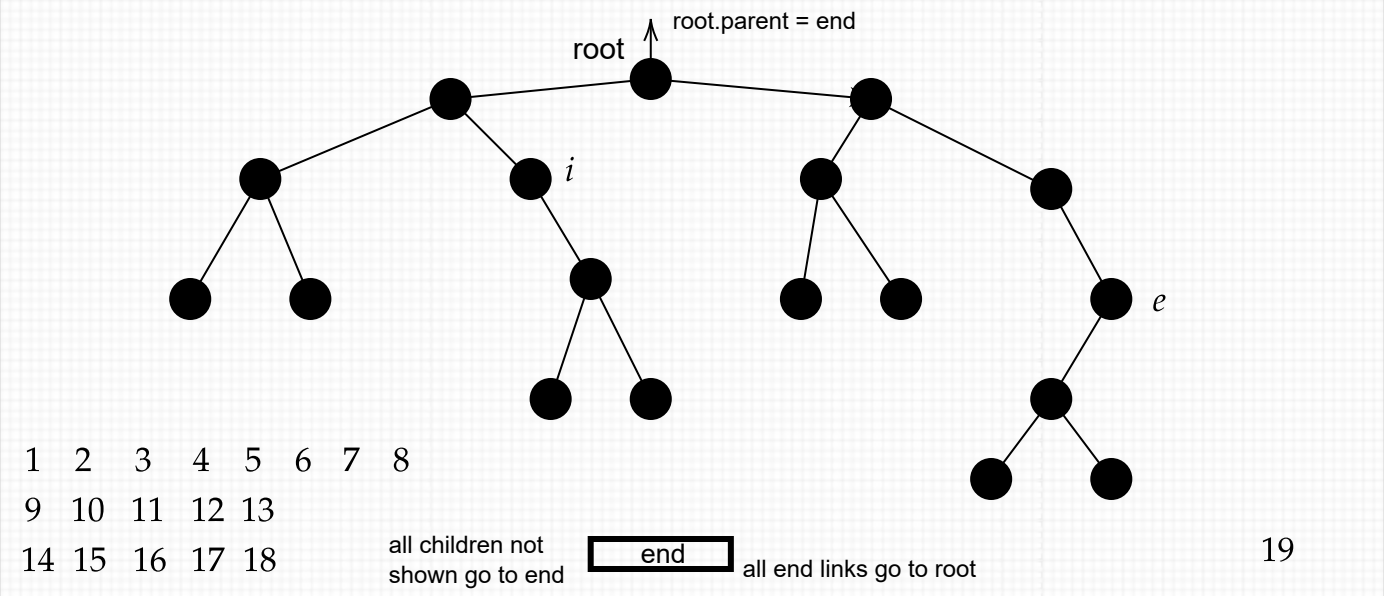
CSE250 Week 11: Binary Search Trees

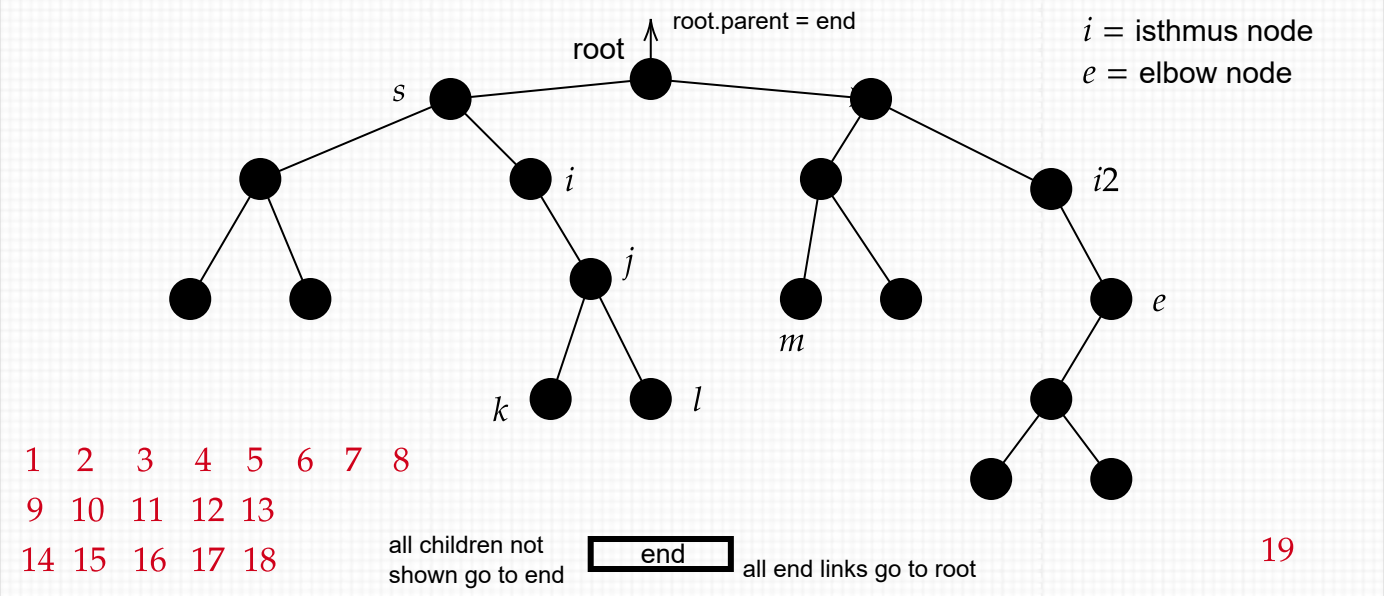
All binary search trees (BSTs) maintain the invariant that the keys of the items they store are sorted by inorder. Given a set of n items, a BST holding those items may have many possible shapes. But once the tree is laid out, the keys must appear in sorted order of the **inorder transversal**. (The only exception is that items with equal keys can appear in any order so long as they stay consecutive in the transversal. Applications like Assignment 6 would rely on this.)



IMHO, the most valuable helper function for the tree is the next() function of this transversal. We code it first as a helper function `inorderSuccessor` using nodes, and then the iterator simply uses that helper. The rules for the inorder successor v of a node u are:

1. If u has a right subtree, then v is the minimum node (a.k.a. the leftmost node or the "minChild") in that subtree.
2. Else, v is the first ancestor of u reached by going up a `left`-child link in the upward path.

The first node in the inorder transversal is the min-node of the tree. Is it obtained by applying Rule 1 to $u =$ the root? *No*, because the root usually has both left and right subtrees and then the successor is not in the right subtree. The root is usually somewhere in the middle of the inorder transversal anyway. The proper jumping-off point is the `end` node, and then Rule 1 does work: the "CL&R Tree" is rigged so that the right child of `end` is `root`, and the `begin` node is indeed the leftmost node of the "sub"tree rooted at `root`.

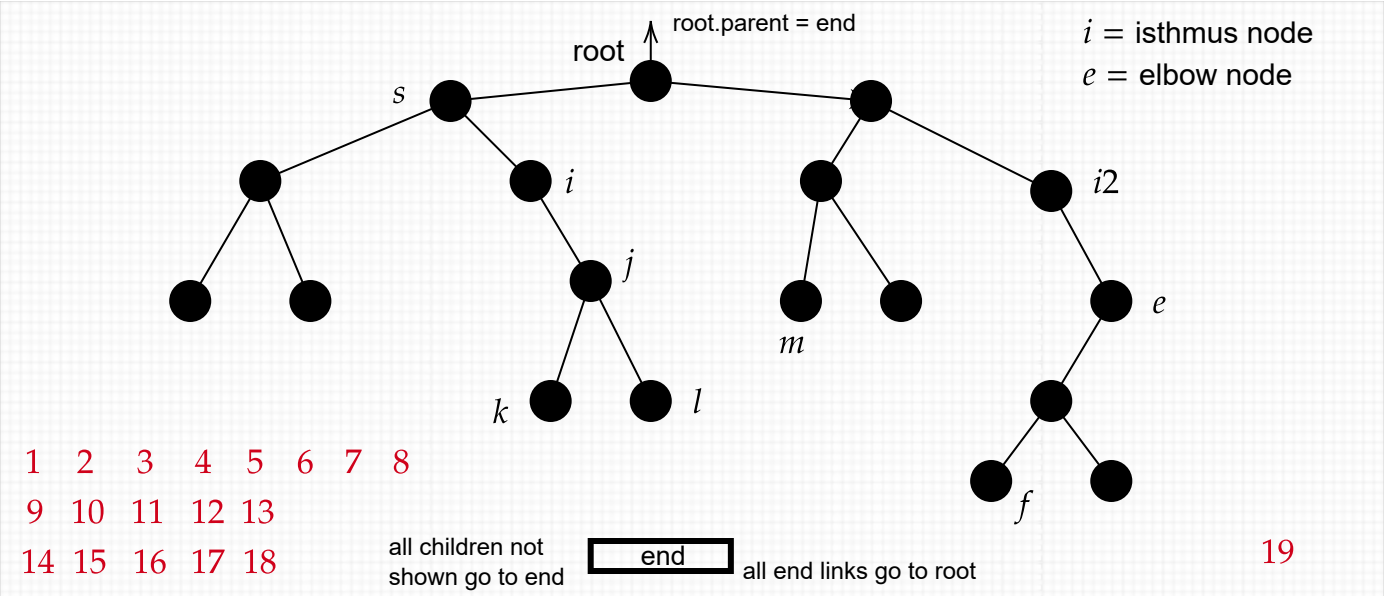


When we reach node s, we have a further dilemma: Is the leftmost node in its right subtree the **isthmus node** i or the leaf k? Well, k is in the right subtree of i, so its key does not compare less-than the key of i, so the correct answer must be i. How can we determine that based just on the tree? It helps to have another helper function (note: `BSTISR.scala` makes it `private`):

```
def leftmostBelow(x: Node) = if (x.left == endSentinel) x else leftmostBelow(x.left)
```

Since `i.left = endSentinel`, the call `leftmostBelow(s.right)`, which is the same as `leftmostBelow(i)`, immediately returns `i`. Next, `leftmostBelow(i.right)` returns `k`.

Thus the inorder transversal proceeds ... $s \rightarrow i \rightarrow k \rightarrow j \rightarrow l$ ... What about the successor of the leaf l? It has no right subtree, so we go up. We follow right-child links until we step from s to the root. Thus the successor is the root. The next node is the leaf m, quite a swoop in two calls.



We continue in this manner, until (in this example tree) we get to the second isthmus $i2$, whose right child is the "elbow" e. That node has a left subtree, so iterating `leftmostBelow(i2.right)` finds the leaf m. Then after three more calls, we find that e itself is the last node in the transversal. What is the successor of e? We follow right-child links all the way up to the root, but in the CL&R tree, `root` counts as a left-child of `end`. So the successor of e is `end`. This is actually what we want: the iteration goes to the `end` position, and then (if we deem it safe), we can cycle around again. Here is the code (because root is also a right-child of end, we word a helper negatively):

```
private def isNotLeftChild(u: Node) = (u != u.parent.left)

private def inorderSuccessor(x: Node): Node = {
   if (x.right != endSentinel) {
      return leftmostBelow(x.right)
   } //else
   var y = x
   while (isNotLeftChild(y)) {
      y = y.parent
   }
   return y.parent
}
```

The text does this a different way, inside the anonymous `iterator` implementation at the bottom of page 507. It uses a `Stack` to save upcoming upward steps on the fly while executing the downward step `pushRunLeft` in place of my `leftmostBelow`. This is recursive code; the above is iterative code. Besides the limitation that the text builds the whole traversal---you can't easily place an iterator into the middle of the tree---it seems not to realize that this code can be *re-used* by the removal method.

Inside the Iter class, we call it by:

```
   def next(): A = {
      assert(hasNext, "Attempt to advance past end")
      val ret = at.item   //this needs a temporary
      at = Outer.inorderSuccessor(at)
      return ret
   }
```

The need for `Outer.` here is a Scala technicality going with the "`{ Outer = >`" ending to the opening line of the `BSTISR` class.

You could also code a `prev()` method to iterate in the other direction with the help of an `inorderPredecessor` method. The rules are mirror-image, e.g., if a node u has a left subtree, then

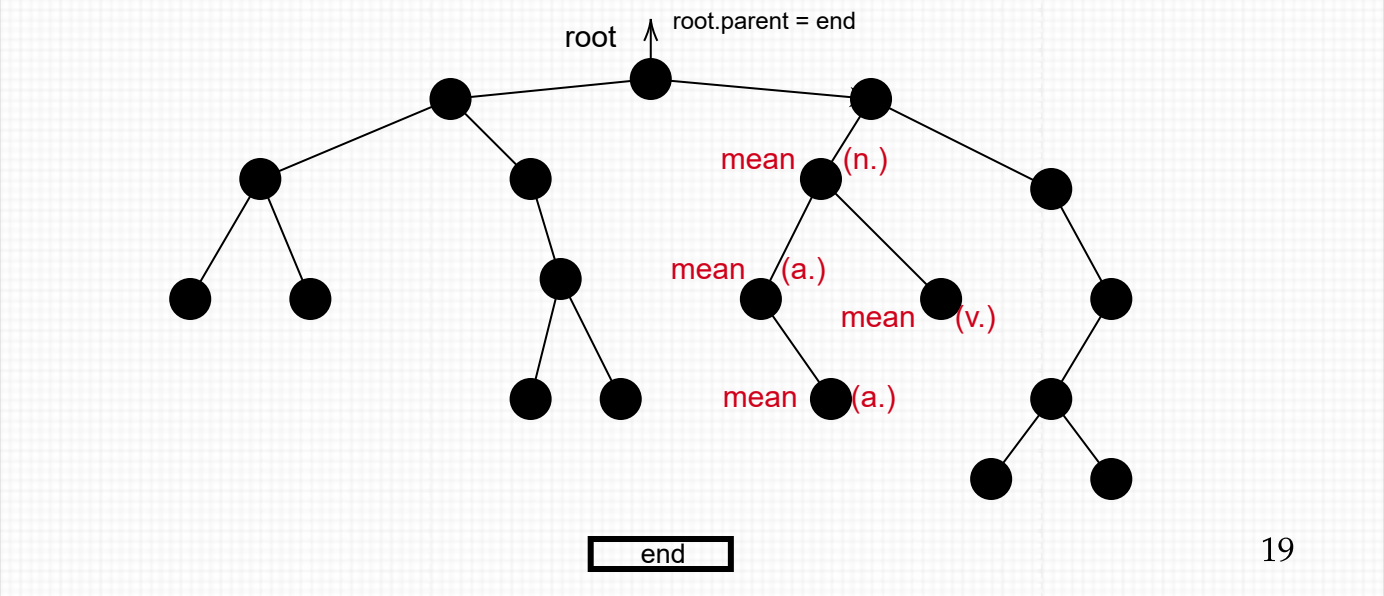
the predecessor is the ***maximum*** child in that subtree. Actually, this is what the text does while coding the routine for removal (which it calls as the `-=` operator). But we can do removal in mirror image---many other texts say to remove the right-subtree's `minChild` instead, and that is exactly what we can *re-use* `inorderSuccessor` for.

[Wednesday's lecture ended here with some Q&A and a preview demo of the BST code.]

But let's see the code for insertion (`+=` in the text) and `find` first. The following code for `find` is fairly similar to that in the text, modulo the difference that the text explicitly separates the key and value in its `TreeMap` class. (IMPHO this creates lots of "tuple clutter" in the code.) One global similarity is that the text uses a `comp`arator-type function rather than a `lessThan` (in either `<` or `<=` forms), and both our code exploits this to finish right when an equal comparison is found:

```
def find(item: A): Iter = {
   if (root.item == null) {  //i.e., tree is empty
      return end
   } //else
   var rover = root
   while (rover != endSentinel) {
      val c = keyComp(item, rover.item)
      if (c == 0) {
         return new Iter(rover)
      } else if (c < 0) {
         rover = rover.left
      } else {
         rover = rover.right
      }
   } //control here means not found
   return end
}
```

Nothing could seem more natural---and yet there is a drawback for our desire to iterate on equal-key items. Suppose our tree came out like this:



The above code stops at the noun form of mean. Then the iterator going forward will catch the verb form, but will miss the entries for the adjectival forms---and there are two of them. The idea for the fix is that the first occurrence of a key will always be leftward of the uppermost one found. So we change the code to read:

```
if (c == 0) {
   while (rover.left != endSentinel && keyComp(item,rover.left.item) == 0) {
      rover = rover.left
   }
   return new Iter(rover)
} ...
```

Finally, note that the text's `get` method uses the convention of returning `None` if the item `i` is not found, `Some(i)` when found. This code instead returns an iterator `itr` such that `itr.hasNext` means the item was found (so `itr()` will return it), otherwise `itr` is `end`.

Insertion

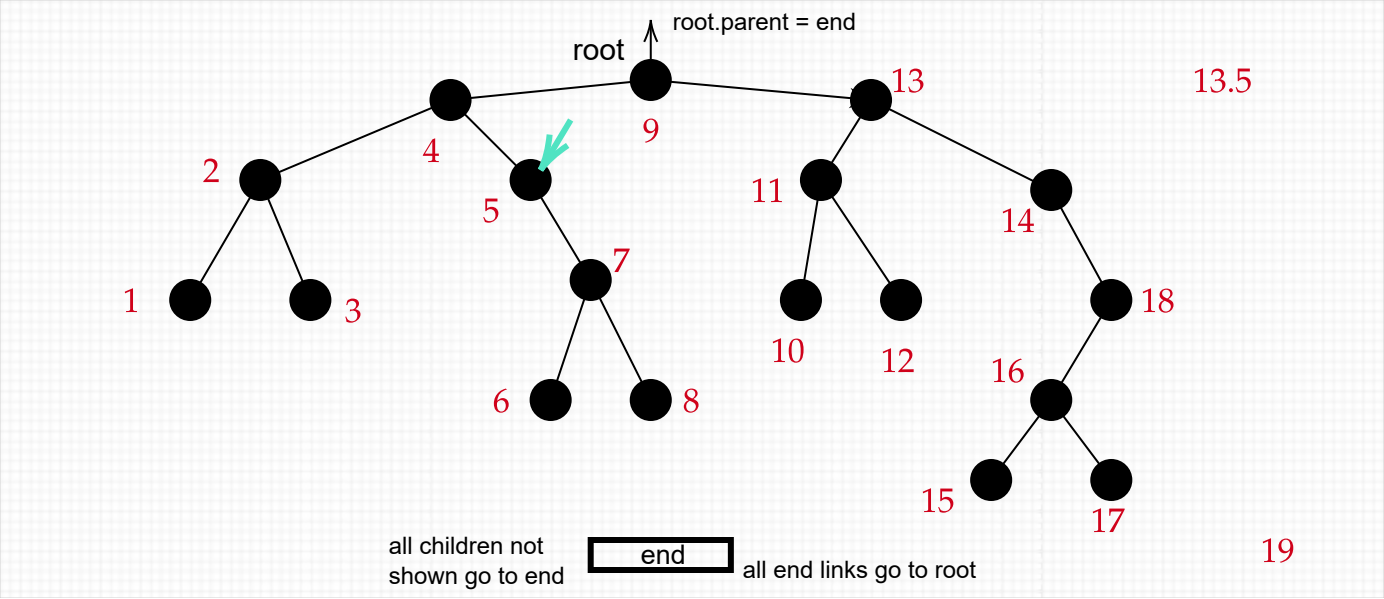
Insertion actually benefits from an iterator helper function that, however, is unsafe to use publicly:

```
type I = Iter
def begin = new Iter(inorderSuccessor(endSentinel))  //per above discussion
def end: Iter = new Iter(endSentinel)   //And double links help here: O(1) time


/** REQ: Node of loc is a leaf or elbow, and insertion will keep sortedness
    Note that we make 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)
      return new Iter(loc.at.left)
   } else {
      loc.at.right = new Node(item, endSentinel, endSentinel, loc.at)
      return new Iter(loc.at.right)
   }
}

/** Public version needed for consistency with ISR trait, but use insert(item)
 */
def insert(item: A, loc: Iter):Iter = insert(item, loc, loc.at.left==endSentinel)
```

The new node is always a leaf, and the `leftward` argument tells whether it is intended to be a left child or right child of the parent node.

The reason this is unsafe is that inserting an arbitrary element can break the sortedness invariant. Let's suppose our tree currently holds integer keys. Suppose we have an iterator to the isthmus node with key 5. We cannot insert a right-child there. The only left-children we can insert must have keys between 4 and 5. If we want to insert, say, 13.5 as a key, it must go elsewhere.



The insertion routine must find the place where the key of the given item can go. There is a special case when the tree is empty because the root node pre-exists with a null item.

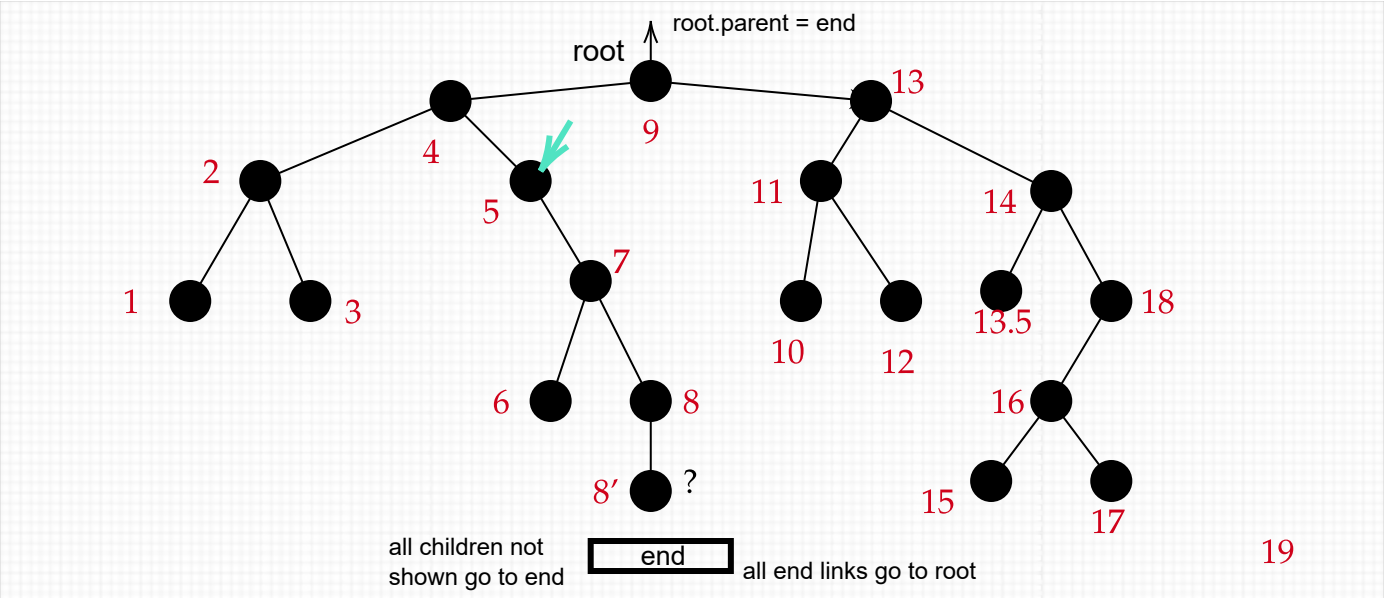
```
def insert(item: A): Iter = {
```

```
      if (isEmpty) {
         assert(root.item == null, "Null not replaced in BST")
         root.item = item
         _size += 1
         return new Iter(root)
      } //else
```

The next problem compared to `find` is that the sequence of comparisons is terminated by `rover` becoming the end sentinel (or becoming `null` in the text), but then it forgets the node it came from, which is where we want to insert. So we run a "trailer pointer" called `parent` (in the text too):

```
      var parent = root  //can't trust parent link of rover: could be endSentinel
      var wasLeft = keyComp(item, root.item) < 0      //dupe keys "tend to append"
      //var wasLeft = keyComp(item, root.item) <= 0   //dupe keys "tend to prepend"

      var rover = (if (wasLeft) root.left else root.right)
      while (rover != endSentinel) {
         parent = rover
         wasLeft = (keyComp(item, rover.item) < 0)
         //wasLeft = (keyComp(item, rover.item) <= 0)
         rover = (if (wasLeft) rover.left else rover.right)
      }
      _size += 1    //we have found a place to insert and whether left or right
      return insert(item, new Iter(parent), wasLeft)
   }
```

There is one more wrinkle related to, but not as critical as, the issue above with `find` and the issue with binary search in arrays with intervals of equal-key items. When we are inserting an item with a key $k_2 = k_1$ that already exists in the tree, we have a choice of placing k_2 physically before or after k_1 in the tree. If we move left only on strict inequality, we will place equal-key items rightward. Moving left on equals will always make the new item first in the range.



Removal

The first logical contrast to note is a general fact about sorted sequences---not necessarily from trees:

- Inserting an arbitrary element into a sorted sequence can break the sortedness.
- But removing an element from a sorted sequence still always leaves it sorted.

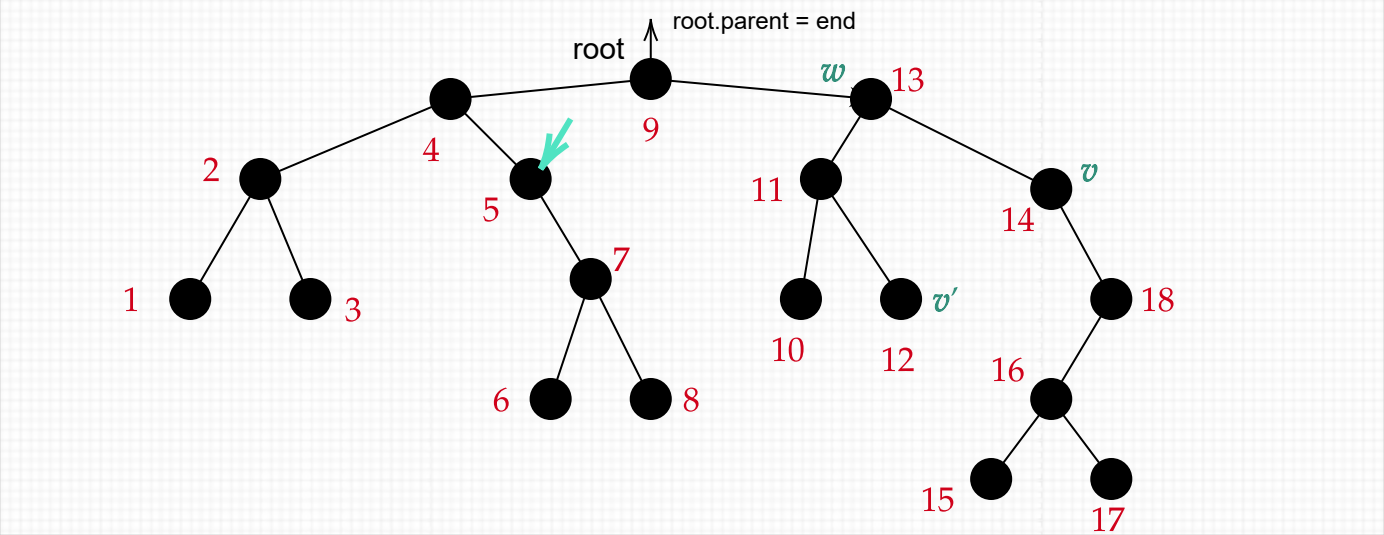
So it is fine to have an iterator version of `remove`, with the iterator having been placed by a call to `find`

(more code re-use).

```
/** Return full item removed, if "item" is a dummy.
    If not found, return original dummy item.
 */
def remove(item: A): A = {
   val itr = find(item)
   if (!itr.hasNext) { return item }  //and do nothing, == how text treats "-="
   //else
   return remove(itr)
}
```

Now we encounter a physical fact about trees:

- It is super-easy to remove an item in a leaf---just delete the leaf node.
- It is easy to remove an item in an isthmus or elbow node v. The single subtree of v becomes the
  new subtree of the parent node u of v.
- But a full binary node w cannot be snipped out.



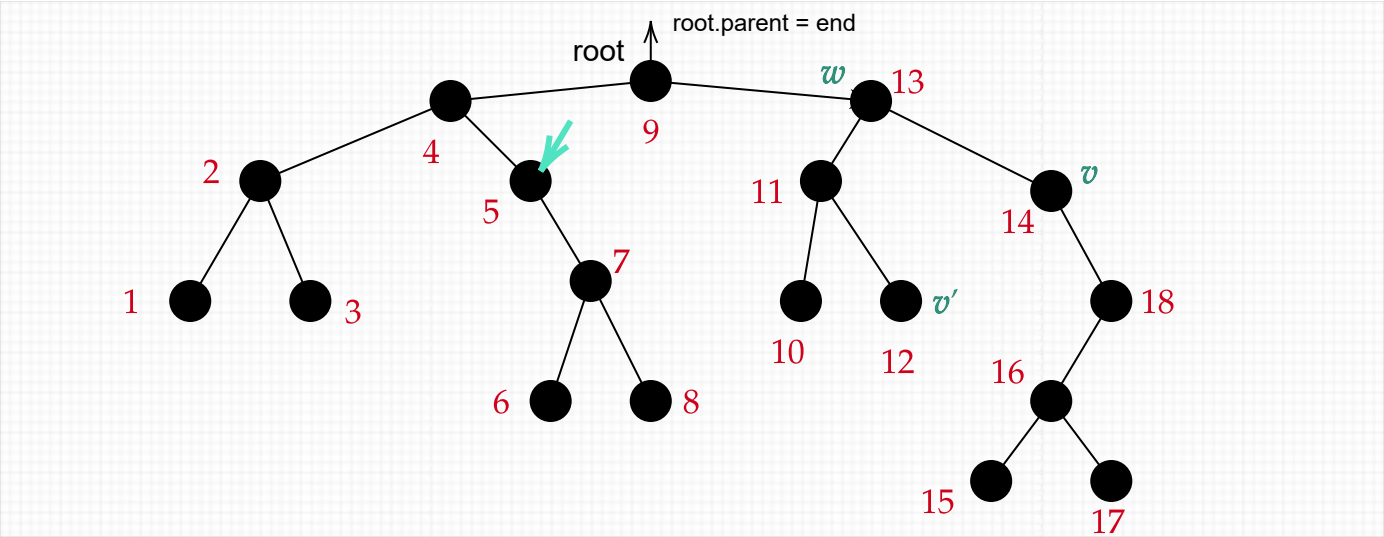
However, a fact we encountered about the inorder transversal comes to our aid: The successor v of w is always a leaf or isthmus or elbow node---in particular, it never has a left subtree. If it did have a left subtree, the actual successor would have been in there. The text actually uses a mirror-image fact: the inorder predecessor v of w always has an empty right subtree. (If it did, the successor of v would have been in that subtree, and so wouldn't be w.) In either case we can:

1. Swap the items of nodes w and v.
2. Delete the "victim" node v, which now has the item that was in w which we want to remove.

The call to `inorderSuccessor` does the work of `findVictim` in the text; the body of `findVictim` in the text actually does what you could code as `inorderPredecessor`. The code below needs more lines because of the special-case treatment of `root` as well as the presence of `endSentinel` and `parent` links. It recurses on itself once the victim node is found, because the victim node has an empty subtree so that the recursion cannot fall through to the bottom again.

```
/** Removing any Node is always legal by the sortedness invariant, but full binary
    nodes requires swapping value with the inorderSuccessor before deleting its node.
    Note that the operation of searching for the node with item to remove is in other code.
    The code has to do extra work because it uses endSentinel not null links as in the text.
 */
def remove(loc: Iter): A = {
   assert(loc.hasNext, "Attempt to remove past-end Node of unfound item")
   _size -= 1        //we will always delete something
   val tmp = loc()   //we will remove this item but not necessarily its node
   if (loc.at == root) {   //we treat it specially
      if (_size == 0) {
         root.item = null.asInstanceOf[A]
         return tmp
      } else if (root.left == endSentinel) {
```

```
            root = root.right
            root.parent = endSentinel
            endSentinel.parent = root
            endSentinel.left = root
            endSentinel.right = root
            return tmp
         } else if (root.right == endSentinel) {
            root = root.left
            root.parent = endSentinel
            endSentinel.parent = root
            endSentinel.left = root
            endSentinel.right = root
            return tmp
         } else {
            //we do nothing now because the root will not be moved.
         }
      }     //no "else" here: we want the fall-through if loc.at == root == a full internal node
            //the next two tests are redundant but handling needs loc.at.parent to be real node.
      if (loc.at.left == endSentinel) {   //its right subtree becomes 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    //and this overwrites loc.at, so loc becomes invalid
         } else {
            parent.right = loc.at.right
         }
      } else if (loc.at.right == endSentinel) {  //its left subtree becomes 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
         }
      } else {    //loc.at is a full binary node.  But this means its successor is not.
                  //So after swapping in that node's value, we can remove it with one more call.
         val u = inorderSuccessor(loc.at) //which is "findVictim" in the text
         loc.at.item = u.item             //tree being mutable is very helpful here
         return remove(new Iter(u))       //recursion is safe because it won't fall thru to else
      }
      //control here means we were in one of the easier cases and loc.at is already spliced out.
      return tmp
   }
```



Issues With Simple Binary Search Trees

The main issue is similar to that with Quicksort: if the data arrives already in (nearly-)sorted order, you will get an unbalanced tree giving at-worst $O(n)$ performance per call to `find` (likewise `insert` and `remove`), and hence possibly needing $O(n^2)$ time to build a tree of n elements. Traversal would still be $O(n)$ time total, and amortized $O(1)$ time per call, but calling `find` at-will n times could take the worst-case $O(n^2)$-time total. (Note that the list of synonyms can have words from all over the alphabet, so that part of the task really does require *random access* to the whole data structure. It is not a neatly streaming task...)

The text does not actually define exactly when a tree is balanced. The colloquial definition is that its height is "$O(\log n)$" in regard to the total size n. I prefer specifying the principal constant in the "O" to be 2: A binary search tree of n nodes is **balanced** if its height is at most $2 \log_2 n$.

There are two main strategies for preserving well-balanced trees.

1. Maintain balance after each insertion and removal by doing extra work in the form of **rotations** as-needed.
2. Periodically `refresh` the tree. This is a painful $O(n)$ time extra operation when needed, but is like the resize policy already used for arrays: it is amortized by being needed only after linearly many quick operations.

The text covers strategy 1 in chapter 21 in the form of **AVL Trees**. I've put the code for strategy 2 into `BSTISR.scala`, but have not yet specified how to manage it.