CS 228: Introduction to Data Structures
Lecture 28
Friday, April 3, 2015

`remove()`

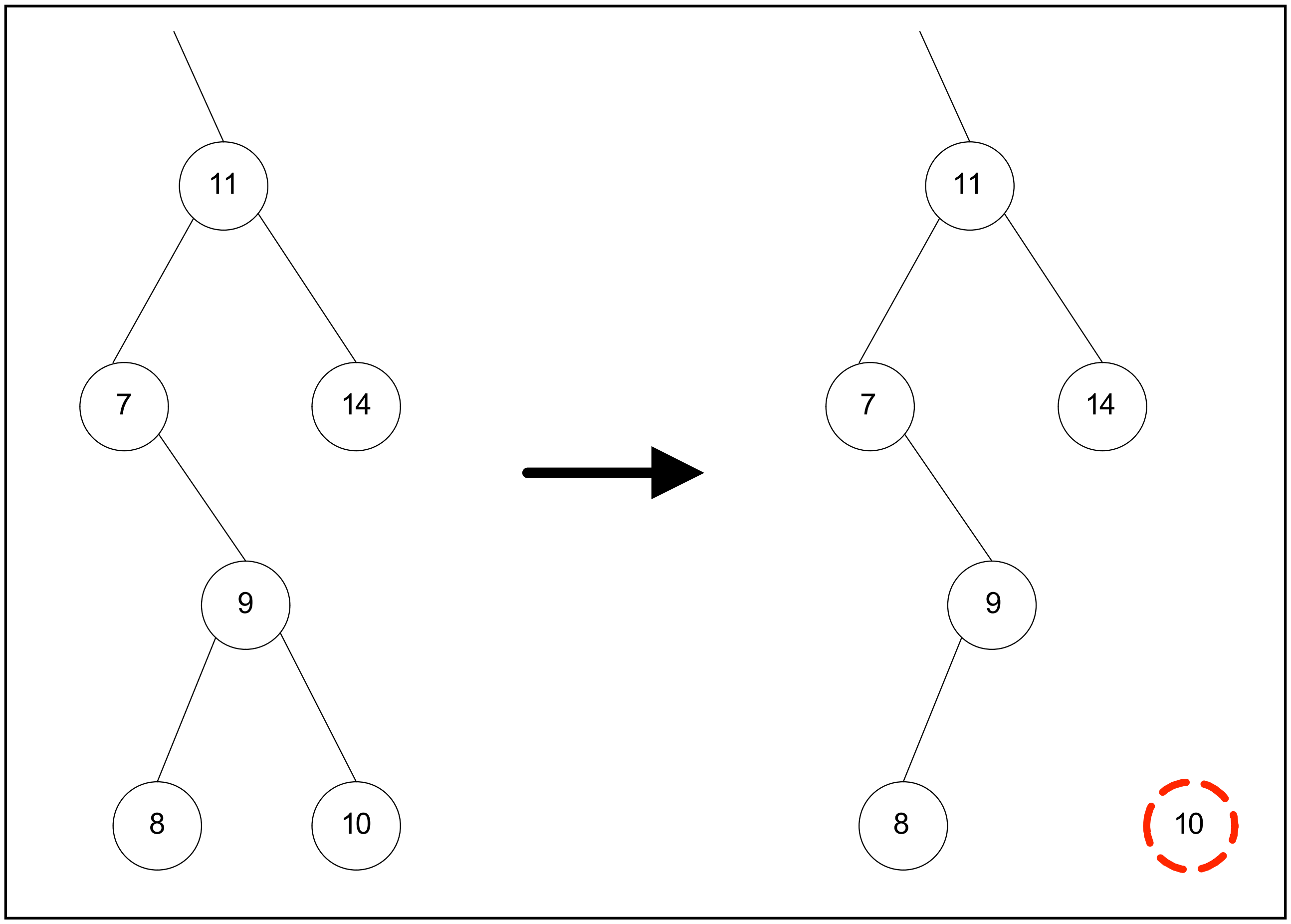
To delete a key, we first have to find it; `findEntry()` takes care of this. If the key is not in the tree, we just return `false`. Otherwise, `findEntry()` returns a reference to the node containing the key. We then leave the actual removal to a helper method called `unlinkNode()`, which we'll describe shortly. The code for `remove()` is now simply this.

```
public boolean remove(Object obj)
{
  E key = (E) obj;
  Node n = findEntry(key);
  if (n == null)
  {
    return false;
  }
  unlinkNode(n);
  return true;
}
```

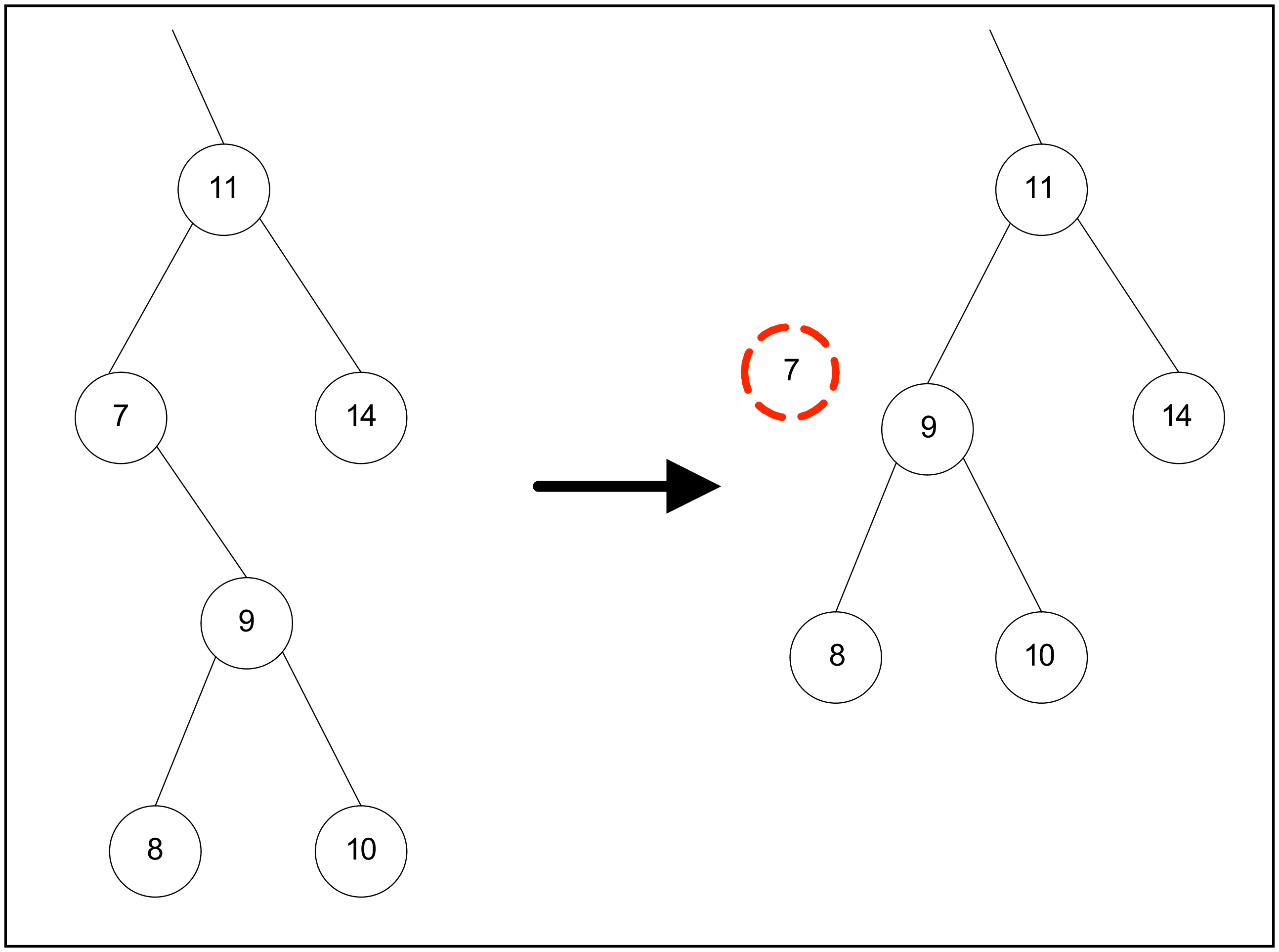
`unlinkNode()`

There are three possibilities.

Case 1: The node to be removed has no children. Then, we simply delete the node.



Case 2: The node to be removed has one child. This is easy too. The BST property is preserved by replacing the deleted node with its child.



Case 3: The node to be removed has two children. For instance, suppose we want to remove the node “4” in the tree in p. 8. Unlike cases 1 and 2, we cannot just delete the node; we must replace it with something else. If we put either of 4’s children in its place, we then have to move one of the child’s children, and so on. This looks like too much restructuring.

The solution is to leave the node where it is, and find some other element in the tree that can go there without violating the BST property. The ***successor*** of 4 in the tree ordering, which is 7, is a good choice. This is because of the following.

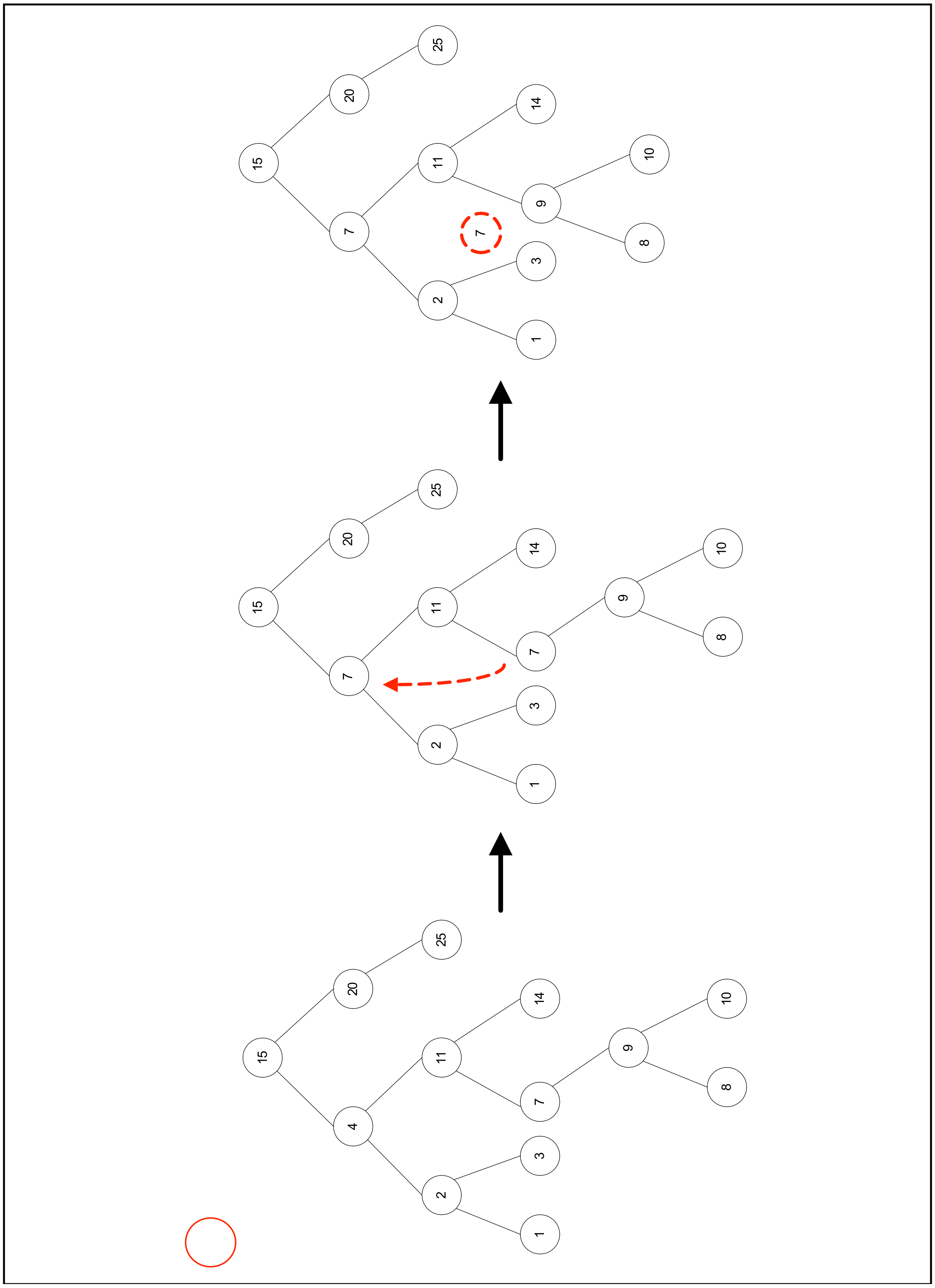
Fact. *If `t` is a node with two children and node `s` contains the successor of `t`, then `s` cannot have a left child*.

Exercise. Prove this.

Thus, deleting the successor of a node with two children falls under the (easy) Cases 1 or 2. In our example, the node `s` containing 7 only has one child. We copy the 7 into the node t containing 4, and then delete node `s`.

Note. In the preceding example, instead of replacing 4 by its successor, we could have replaced it by its ***predecessor*** in the tree ordering, which is 3. The predecessor of a node with two children cannot have a right child, so deleting it is as easy as deleting the successor.

The pseudocode for `unlinkNode` on pp. 9-10 assumes that we have a `successor()` method, which, given a node n, returns the successor of n or `null`, if n has no successor.


15
4
20
25
2
11
1
3
7
14
9
8
10
15
7
20
25
2
11
1
3
7
14
9
8
10
15
7
20
25
2
11
1
3
7
14
9
8
10

```
unlinkNode(n):
    if n has two children
        // Copy the successor's data into n; then arrange
        // to delete the successor.
        s = successor(n)
        n.data = s.data
        n = s

    // Now n has at most one child, so we delete n and
    // replace it with the child (or possibly null).

    // First, figure out whether the replacement node
    // should be the left child, right child, or null.
    replacement = null
    if n has a left child
        replacement = n.left
    else if n has a right child
        replacement = n.right

    // Now, link the replacement node to n's parent.
    if n is the root
        root = replacement
    else
        if n is a left child of its parent (*)
            n.parent.left = replacement
        else
            n.parent.right = replacement
    if replacement != null
        replacement.parent = n.parent

    --size
```

(*) To test if n is a left child of its parent, we do

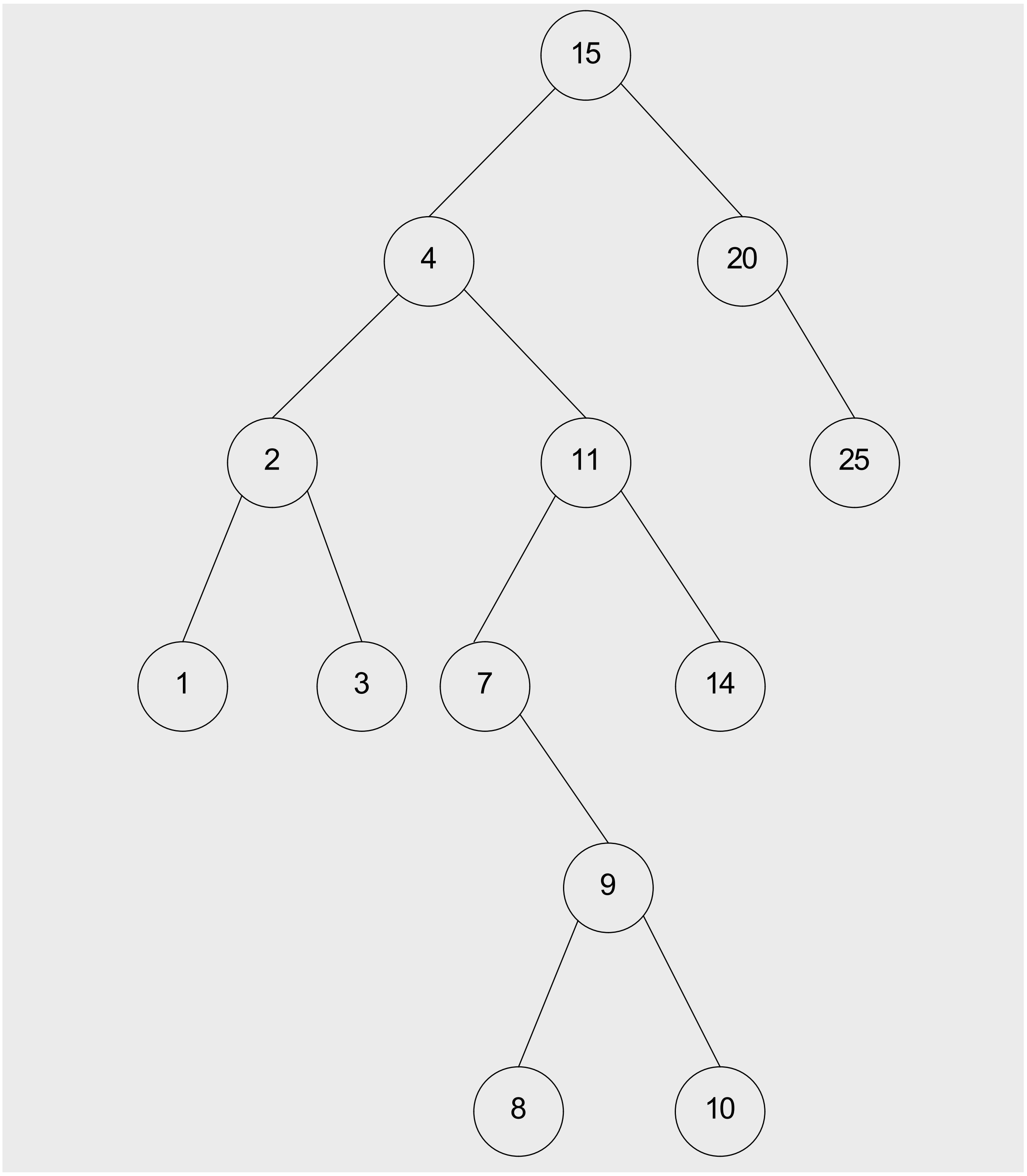
```
n == n.parent.left
```

`successor()`

There are two cases (see the example on the next page):

1. **The node has a right subtree.** Go down to leftmost element in right subtree. (E.g., the successor of 4 is 7).

2. **The node does not have a right subtree.** Then the node is the rightmost element of some subtree. If there actually is a successor, then it must be the root `r` of this subtree. (E.g., the successor of 10 is 11). To find `r`, go up the tree following parent pointers, stopping when the previous node in the path is a left child of the current node.


15
4
20
2
11
25
1
3
7
14
9
8
10

Running Times of `successor()` and `remove()`

The running time of `successor()` is O(height(T)) because the algorithm either moves up or down in the tree, but not both, doing O(1) work per step.

The running time of `remove()` is also O(height(T)). This is clear for the case where the node to be removed has at most one child, since the work is dominated by the time for `findEntry()`. When the node to be removed has two children, the work essentially consists of `findEntry()` followed by `successor()`. Since both of these are O(height(T)) operations, the running time of `remove()` in this case is also O(height(T)).

Iterators for BSTs

Suppose we want an iterator that goes through the keys of a BST following the natural order (which is the in-order sequence). The iterator should start at the minimum — or leftmost — element of the tree, and proceed all the way to the maximum — or rightmost — element. We can implement such an iterator as follows.

- The state of the iterator is given by a cursor (reference) to the element that would be returned by the next call to `next()`. To initialize it, we go down the tree, following left child pointers until we find a node with no left child. This must be the minimum. (Can you see why?)

- `hasNext()` and `next()` are easily implemented using `successor()`.

- `remove()` is implemented using `unlinkNode()`, with a little twist: Suppose we remove a node `x` with two children. Then `x` is not actually removed; instead, its successor is copied into it. This successor is exactly what a call to `next()` should return. Thus, we reassign the cursor to reference that node.

Performance of the iterator. The time to advance the iterator depends on the length of the path we traverse to find the successor. This, in turn, depends on the height of the tree. In the worst case, the time is O(n) (although more typically it is closer to O(log n)). This suggests that iterating over all n elements takes O(n^2) time in the worst case. The situation is actually a lot better.

Iterating through the keys with an iterator is essentially an in-order traversal. This implies that iterating over all elements is O(n). Another way to think about it is to notice that when iterating over all elements using `successor()`, each link —`left`, `right`, and `parent`— is used exactly once. The total number of such links is 3n. Thus, the total cost of iterating over the whole tree using the `successor()` method is O(n), even in the worst case. That is, the ***amortized*** cost of `successor()` is O(1) per call, even though any particular invocation might take O(n) time. Note that this O(1) amortized bound is independent of the height: it holds whether the tree is balanced or not.