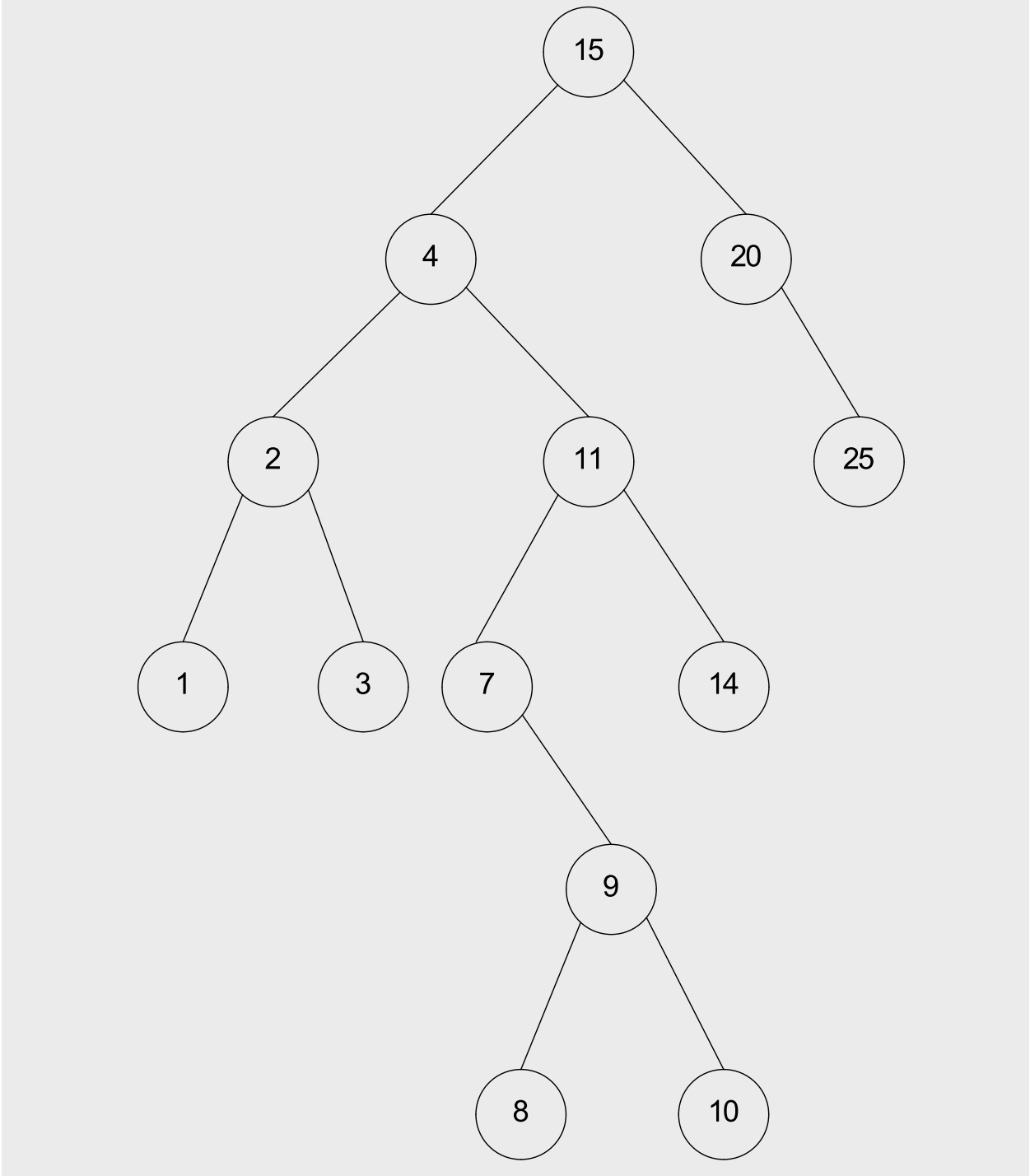
CS 228: Introduction to Data Structures
Lecture 31

`successor()`

Given a reference `n` to a node in a BST `T`, `successor()` returns a reference to the node that contains the successor of `n.data` (according to the natural ordering) in `T`. The method has two cases (see the example on the next page):

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

- **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.

Splay Trees

The rest of today's lecture dealt with splay trees, a kind of self-adjusting BST. The lecture was based on slides by Professor Jia, posted on Blackboard.