


```

+--+
|18|
/--\---+
 12  |25|
 / \  +/\---+
4 15 25 |30|
/ / \ +-\++
1 13 17 |28|
 \ \ \ +-\+
 3 14 29
    
```

For instance, suppose we search for the key 27 in the tree at left. Along the way, we encounter the keys 25 and 28, which are the keys closest to 27 (below and above).

Here's how to implement a method `smallestKeyNotSmaller(k)`: search for the key `k` in the tree, just like in `find()`. As you go down the tree, keep track of the smallest key not smaller than `k` that you've encountered so far. If you find the key `k`, you can return it immediately. If you reach a null pointer, return the best key you found on the path. You can implement `largestKeyNotLarger(k)` symmetrically.

```

[2] Entry min();
    Entry max();
    
```

`min()` is very simple. If the tree is empty, return null. Otherwise, start at the root. Repeatedly go to the left child until you reach a node with no left child. That node has the minimum key.

`max()` is the same, except that you repeatedly go to the right child. In the tree above, observe the locations of the minimum (1) and maximum (30) keys.

```

[3] Entry insert(Object k, Object v);
    
```

`insert()` starts by following the same path through the tree as `find()`. (`find()` works because it follows the same path as `insert()`.) When it reaches a null reference, replace that null with a reference to a new node storing the entry (`k`, `v`).

Duplicate keys are allowed. If `insert()` finds a node that already has the key `k`, it puts the new entry in the left subtree of the older one. (We could just as easily choose the right subtree; it doesn't matter.)

```

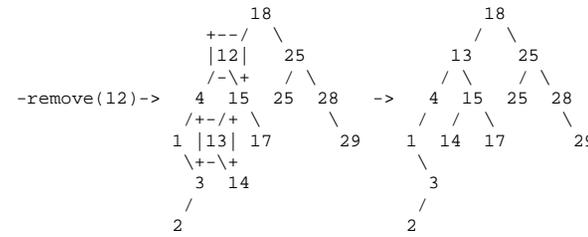
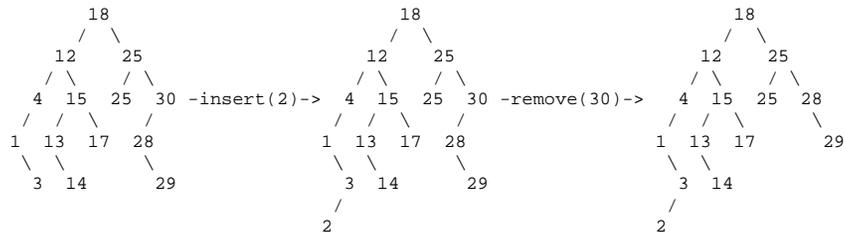
[4] Entry remove(Object k);
    
```

`remove()` is the most difficult operation. First, find a node with key `k` using the same algorithm as `find()`. Return null if `k` is not in the tree; otherwise, let `n` be the first node with key `k`.

If `n` has no children, we easily detach it from its parent and throw it away.

If `n` has one child, move `n`'s child up to take `n`'s place. `n`'s parent becomes the parent of `n`'s child, and `n`'s child becomes the child of `n`'s parent. Dispose of `n`.

If `n` has two children, however, we have to be a bit more clever. Let `x` be the node in `n`'s right subtree with the smallest key. Remove `x`; since `x` has the minimum key in the subtree, `x` has no left child and is easily removed. Finally, replace `n`'s entry with `x`'s entry. `x` has the key closest to `k` that isn't smaller than `k`, so the binary search tree invariant still holds.



To ensure you understand the binary search tree operations, especially `remove()`, I recommend you inspect Goodrich and Tamassia's code on page 446. Be aware that Goodrich and Tamassia use sentinel nodes for the leaves of their binary trees; I think these waste an unjustifiably large amount of space.

Running Times of Binary Search Tree Operations

In a perfectly balanced binary tree (left) with height h , the number of nodes n is $2^{(h+1)} - 1$. (See Footnote 2.) Therefore, no node has depth greater than $\log_2 n$. The running times of `find()`, `insert()`, and `remove()` are all proportional to the depth of the last node encountered, so they all run in $O(\log n)$ worst-case time on a perfectly balanced tree.

On the other hand, it's easy to form a severely imbalanced tree like the one at right, wherein these operations will usually take linear time.

There's a vast middle ground of binary trees that are reasonably well-balanced, albeit certainly not perfectly balanced, for which search tree operations will run in $O(\log n)$ time. You may need to resort to experiment to determine whether any particular application will use binary search trees in a way that tends to generate somewhat balanced trees or not. If you create a binary search trees by inserting keys in a randomly chosen order, or if the keys are generated by a random process from the same distribution, then with high probability the tree will have height $O(\log n)$, and operations on the tree will take $O(\log n)$ time.

Unfortunately, there are occasions where you might fill a tree with entries that happen to be already sorted. In this circumstance, you'll create the disastrously imbalanced tree depicted at right. Technically, all operations on binary search trees have $\Theta(n)$ worst-case running time.

For this reason, researchers have developed a variety of algorithms for keeping search trees balanced. Prominent examples include 2-3-4 trees (which we'll cover next lecture), splay trees (in one month), and B-trees (in CS 186).

Footnote 1: When we search for a key `k` not in the binary search tree, why are we guaranteed to encounter the two keys that bracket it? Let `x` be the smallest key in the tree greater than `k`. Because `k` and `x` are "adjacent" keys, the result of comparing `k` with any other key `y` in the tree is the same as comparing `x` with `y`. Hence, `find(k)` will follow exactly the same path as `find(x)` until it reaches `x`. (After that, it may continue downward.) The same argument applies to the largest key less than `k`.

Footnote 2: A perfectly balanced binary tree has 2^i nodes at depth i , where

$$0 \leq i \leq h. \text{ Hence, the total number of nodes is } \sum_{i=0}^h 2^i = 2^{h+1} - 1.$$