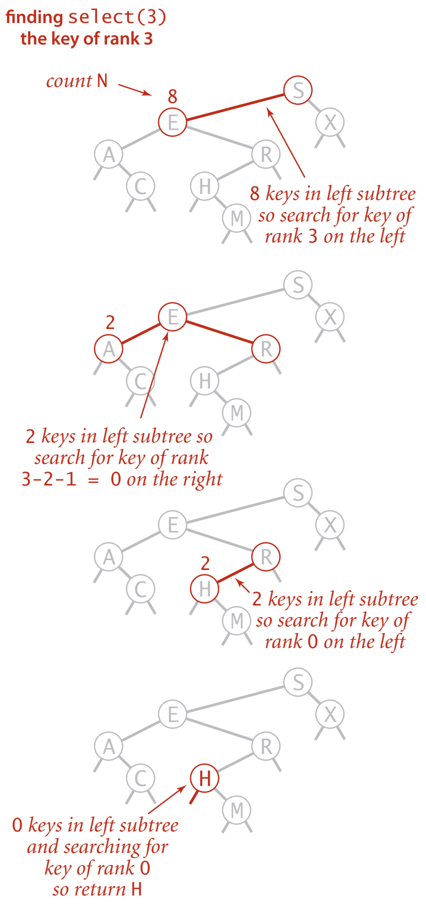
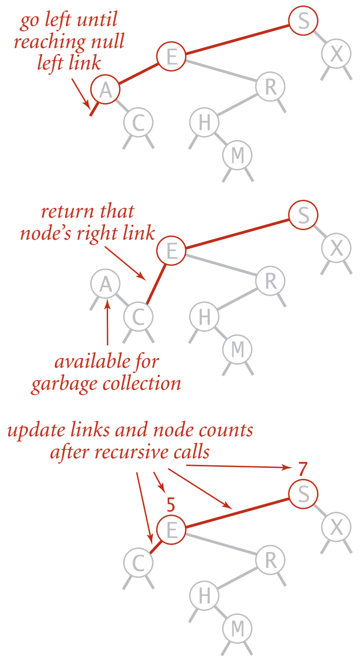
Suppose that we seek the key of rank *k* (the key such that precisely *k* other keys in the BST are smaller). If the number of keys *t* in the left subtree is larger than *k*, we look (recursively) for the key of rank *k* in the left subtree; if *t* is equal to *k*, we return the key at the root; and if *t* is smaller than *k*, we look (recursively) for the key of rank $k - t - 1$ in the right subtree. As usual, this description serves both as the basis for the recursive `select()` method on the facing page and for a proof by induction that it works as expected.


finding select(3)
the key of rank 3
count N
8
8 keys in left subtree so search for key of rank 3 on the left
2
2 keys in left subtree so search for key of rank 3-2-1 = 0 on the right
2
2 keys in left subtree so search for key of rank 0 on the left
0 keys in left subtree and searching for key of rank 0 so return H

Selection in a BST

Rank. The inverse method `rank()` that returns the rank of a given key is similar: If the given key is equal to the key at the root, we return the number of keys *t* in the left subtree; if the given key is less than the key at the root, we return the rank of the key in the left subtree (recursively computed); and if the given key is larger than the key at the root, we return *t* plus one (to count the key at the root) plus the rank of the key in the right subtree (recursively computed).


go left until reaching null left link
return that node's right link
available for garbage collection
update links and node counts after recursive calls
7
5

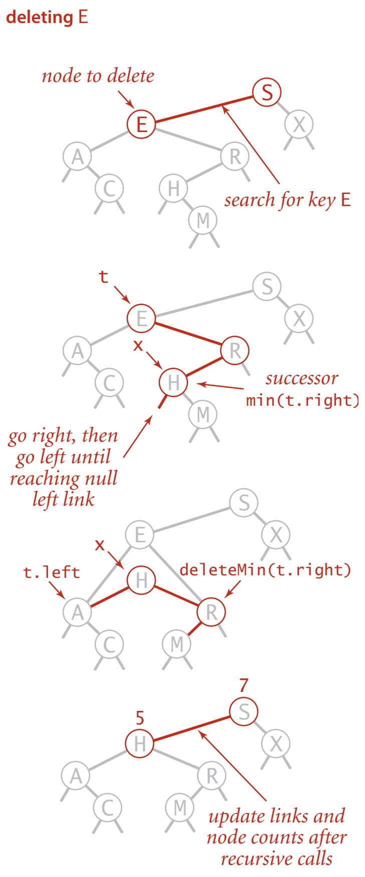
Deleting the minimum in a BST

Delete the minimum/maximum. The most difficult BST operation to implement is the `delete()` method that removes a key-value-pair from the symbol table. As a warmup, consider `deleteMin()` (remove the key-value pair with the smallest key). As with `put()` we write a recursive method that takes a link to a `Node` as argument and returns a link to a `Node`, so that we can reflect changes to the tree by assigning the result to the link used as argument. For `deleteMin()` we go left until finding a `Node` that has a null left link and then replace the link to that node by its right link (simply by returning the right link in the recursive method). The deleted node, with no link now pointing to it, is available for garbage collection. Our standard recursive setup accomplishes, after the deletion, the task of setting the appropriate link in the parent and updating the counts in all nodes in the path to the root. The symmetric method works for `deleteMax()`.

Delete. We can proceed in a similar manner to delete any node that has one child (or no children), but what can we do to delete a node that has two children? We are left with two links, but have a place in the parent node for only one of them. An answer to this dilemma, first proposed by T. Hibbard in 1962, is to delete a node `x` by replacing it with its *successor*. Because `x` has a right child, its successor is the node with the smallest key in its right subtree. The replacement preserves order in the tree because there are no keys between `x.key` and the successor's key. We can accomplish the task of replacing `x` by its successor in four (!) easy steps:

- Save a link to the node to be deleted in `t`.
- Set `x` to point to its successor `min(t.right)`.
- Set the right link of `x` (which is supposed to point to the BST containing all the keys larger than `x.key`) to `deleteMin(t.right)`, the link to the BST containing all the keys that are larger than `x.key` after the deletion.
- Set the left link of `x` (which was null) to `t.left` (all the keys that are less than both the deleted key and its successor).

Our standard recursive setup accomplishes, after the recursive calls, the task of setting the appropriate link in the parent and decrementing the node counts in the nodes on the path to the root (again, we accomplish the task of updating the counts by setting the counts in each node on the search path to be one plus the sum of the counts in its children). While this method does the job, it has a flaw that might cause performance problems in some practical situations. The problem is that the choice of using the successor is arbitrary and not symmetric. Why not use the predecessor? In practice, it is worthwhile to choose at random between the predecessor and the successor. See EXERCISE 3.2.37 for details.


deleting E
node to delete
search for key E
t
x
successor
min(t.right)
go right, then go left until reaching null left link
x
t.left
deleteMin(t.right)
7
5
update links and node counts after recursive calls
S
E
X
A
R
C
H
M

Deletion in a BST

ALGORITHM 3.3 (continued) Deletion in BSTs

```
public void deleteMin()
{
   if (isEmpty()) return;
   root = deleteMin(root);
}

private Node deleteMin(Node x)
{
   if (x.left == null) return x.right;
   x.left = deleteMin(x.left);
   x.N = size(x.left) + size(x.right) + 1;
   return x;
}

public void delete(Key key)
{  root = delete(root, key);  }

private Node delete(Node x, Key key)
{
   if (x == null) return null;
   int cmp = key.compareTo(x.key);
   if      (cmp < 0) x.left  = delete(x.left,  key);
   else if (cmp > 0) x.right = delete(x.right, key);
   else
   {
      if (x.right == null) return x.left;
      if (x.left == null) return x.right;
      Node t = x;
      x = min(t.right);  // See page 313.
      x.right = deleteMin(t.right);
      x.left = t.left;
   }
   x.N = size(x.left) + size(x.right) + 1;
   return x;
}
```

These methods implement eager Hibbard deletion in BSTs, as described in the text on the facing page. The `delete()` code is compact, but tricky. Perhaps the best way to understand it is to read the description at left, try to write the code yourself on the basis of the description, then compare your code with this code. This method is typically effective, but performance in large-scale applications can become a bit problematic (see EXERCISE 3.2.37). The `deleteMax()` method is the same as `deleteMin()` with right and left interchanged.