Solution 1:

(a) To perform \texttt{insert((80,20))}, we trace the search path to this point, falling out of the tree on the right child of (70,30). We create a new node at this point. Since the parent splits vertically (cutting dimension is \textit{x}), the new node splits horizontally (cutting dimension is \textit{y}).

(b) To perform \texttt{delete((40,10))}, we find the point at the root node. We search for the replacement point in the right subtree, which returns (50,70) (see the figure below). We copy its contents to the root and recursively delete (50,70) from the right child. We then search for the replacement point in the right subtree, which is clearly (60,90). We copy its contents to the (50,70) node, and recursively delete (60,90).
Solution 2:

(a) To perform \texttt{insert(7)}, we first search for 7 in the tree, falling out at node 6. We first perform a zig-zag rotation, followed by a zig-zig, followed by a zig to bring 6 to the root (see Fig. 1).

(b) To perform \texttt{delete(15)}, we first search for 15 in the tree. We first perform a zig-zig rotation, followed by a zig-zag, followed by a zig to bring 15 to the root (see Fig. 2). Next, we perform \texttt{splay(15)} on the root’s right subtree, but since we fall out of the tree at node 17 itself, so there is no change. Finally, we replace the root with 17, moving 15’s left child to be 17’s left child.

In both cases, in addition to the final tree, show the result after each splay. Indicate which node was splayed on and the tree that resulted after splaying. (Intermediate results may be shown for partial credit.)

Solution 3: Let $T(m)$ denote the running time for \texttt{findMin(Node p, int i)}, where $m$ is the number of points in $p$’s subtree. At levels where the cutting dimension equals $i$, we recurse on one side and at the other levels we recurse on both children. This means that for every two levels of descent, we visit at most two out of the four possible grandchildren. Since the tree is balanced, each of these recursive calls involves at most $m/4$ points. The running time at each node is $O(1)$.
Thus, up to constant factors, the running time is given by the recurrence $T(m) = 2T(m/4) + 1$. By the Master Theorem (where $a = 2$ and $b = 4$), the running time is $O(m^{\log_b a}) = O(m^{\log_4 2}) = O(m^{1/2}) = O(\sqrt{m})$. Therefore, the asymptotic running time of \texttt{findMin} is $O(\sqrt{m})$, where $m$ is the number of points in the subtree.

**Solution 4:** Our approach to answering upper-right queries is similar to that used in nearest-neighbor searching. We will design a recursive helper function. The function is given the query point $q = (q.x, q.y)$, the current node $p$ being visited, the cell associated with this node, and the current best point seen so far in the search.

The helper first checks the various trivial cases. This includes any of the following cases (see Fig. 3(a)):

- We have fallen out of the tree ($p == \text{null}$)
- The current cell is fully to the left or fully below the query point ($\text{cell.hi.x} < q.x$ || $\text{cell.hi.y} < q.y$)
- The current cell lies entirely above best ($\text{cell.lo.y} \geq \text{best.y}$)

If any of these conditions hold, the current node cannot offer a better solution, so we just return best.

If the current point $p\.point$ lies to the northeast of $q$ and lies below best, then we update best to $p\.point$ (see Fig. 3(b)). Finally, we compute the cells associated with the left and right subtrees, and we make recursive calls on each of these. If the cutting dimension is $y$ (horizontal) it is more likely that the left (lower) subtree will contribute a better answer, so we invoke the left-side helper first.

The final code is shown in the code block below. The initial call at the root level is \texttt{upperRight(q, root, rootCell, (+\infty, +\infty))}, where \texttt{rootCell} is the bounding box for the entire point set. The initial best point is set to $(+\infty, +\infty)$ so any other point will be preferred to this.

```java
Point upperRight(Point q, Node p, Rectangle cell, Point best) {
    if (p == null) return best // handle trivial cases
    if (cell.hi.x < q.x || cell.hi.y < q.y) return best // cell is disjoint?
    if (cell.lo.y >= best.y) return best // cell is too high?
        // p's point better?
    if (p\.point.x >= q.x && p\.point.y >= q.y && p\.point.y < best.y))
```
best = p.point

Rectangle leftCell = cell.leftPart(p.cutDim, p.point) // child cells
Rectangle rightCell = cell.rightPart(p.cutDim, p.point)
best = upperRight(q, p.left, leftCell, best) // try both sides
best = upperRight(q, p.right, rightCell, best)
return best
}

Is the cell needed? You might wonder whether it is necessary to maintain the cell in the helper. I believe that in this case it is possible to produce an efficient algorithm without using a cell, but proving efficiency is a bit subtler. As an example, consider the two nodes u and v shown in Fig. 3(c)). Both nodes have the same cutting dimension, the same cutting value, and the same points (up to translation). However, while you need to visit u.right, you do not need to visit v.right. If your algorithm does not use a cell but successfully distinguishes between these two cases, then you did a good job! (The trick is to update best using v.point before recursing on the right.) Having a cell makes it easier to see that v.right does not need to be visited.

Solution 5: There are a couple of ways to solve the range searching problem for an interval [lo, hi]. The simpler approach is to design a function that computes the total number of keys that are strictly smaller than a key x. Call this smallerThan(x).

Given this, we can determine the number of keys in any half-open interval [lo, hi) by computing the difference the number of elements strictly smaller than hi and the number strictly smaller than lo, that is, smallerThan(hi) - smallerThan(lo). Unfortunately, this will not count the element hi itself if it is in the skip list. We fix this by explicitly searching for hi and adding it to the count if present. Thus, the function rangeCount(Key lo, Key hi) can be implemented as

smallerThan(hi) - smallerThan(lo) + (find(hi) ? 1 : 0)

The function smallerThan(Key x) is essentially the same as find(x), except we avoid visiting x itself. We maintain a count, ct, of the number of entries we skip over with each jump. We initialize ct to zero. Whenever the search traverses a next link, we add the span of this link to ct (see Fig. 4 (top)). Note that this will not count the very last node visited, but this is what we want since we only count elements strictly smaller than x. Because the search process is essentially the same as for any skip list, the running time is O(log n) in expectation.

```c
int smallerThan(Key x) { // count keys < x
int i = topmostLevel // start at topmost nonempty level
SkipNode p = head // start at head node
int ct = 0 // number of smaller elements
while (i >= 0) { // while levels remain
  if (p.next[i].key < x) {
    ct += p.span[i] // count number of skipped items
    p = p.next[i] // advance along same level
  }
  else i--; // drop down a level
}
return ct // return final count
}
```
It is also possible to write a single function that performs the entire range search, but it is quite a bit messier. It involves three phases. The first phase performs attempts to find the last node that is less than \( lo \) (see Fig. 4 (bottom)). This is a straightforward modification of find. We initialize the counter \( ct \) to zero and begin the counting process.

We could just walk along the lowest level until reaching \( hi \), but this will be too slow. (It may take up to \( O(n) \) time if the range includes almost all the entries.) To do the search efficiently, we need to prioritize making large jumps.

This is done in two phases, called the up phase and the down phase. During the up phase, we repeatedly follow the next link at this highest level in this node, provided that this link does not go beyond \( hi \). Once we arrive at a node whose highest level next link jumps past \( hi \), we begin the down phase. This is essentially the same as the standard skip-list search. During both the up phase and the down phase, to count the number of elements that we skip over, for each next link traversed, we add the link’s span to a counter \( ct \). The code is given below.

```c
int rangeCount(Key lo, Key hi) { // count keys in [lo, hi]
    SkipNode p = head
    int i = maxLevel
    while (i >= 0) { // find last node < lo
        if (p.next[i].key < lo) p = p.next[i]
        else i--
    }
    int ct = 0 // initialize count
    i = p.next.length - 1 // top level of p
    while (p.next[i].key <= hi) { // up phase
        ct += p.span[i] // count nodes skipped
        p = p.next[i]
        i = p.next.length - 1 // go to top level
    }
    while (i >= 0) { // down phase
        if (p.next[i].key <= hi) {
            ct += p.span[i] // count nodes skipped
            p = p.next[i]
        }
    }
}
```

Figure 4: Smaller-than queries in a skip list (top) and range counting (bottom).
We claim that each of three phases visits $O(\log n)$ nodes in expectation. Thus, the overall running time is $O(3\log n) = O(\log n)$. To prove the claim, observe that the initial search for $lo$ and the final down phase are both structurally identical to the standard skip-list search, so the running time for each is $O(\log n)$. The proof that the up phase takes $O(\log n)$ time is essentially symmetrical to the skip-list analysis given in class. We claim that, in expectation, we make two hops per level. The reason is that whenever we arrive at a node, there is a $1/2$ probability that it will take us up a level. Thus, the number of nodes visited at any given level is equal to the expected number of coin tosses until throwing tails, which (as shown in class) is $O(1)$. Since there are $O(\log n)$ levels with high probability, the expected time for the up phase is $O(\log n)$.

Solution to the Challenge Problem: Rather than designing a special-purpose function for answering this query, we can reduce the problem to solving a single upper-right query. The first simplification we can make is to observe that the right-directed horizontal ray from $q$ hits a rectangle if and only if it hits the right vertical side of any rectangle (see Fig. 5(a)). (Since we assume that $q$ lies outside all the rectangles, this is also true for the left vertical side.) We’ll design a solution to this query problem.

![Figure 5: Answering ray-shooting queries.](chart)

Let us first construct a point set $P$ of size $2n$ consisting of all the lower-right and upper-right corners of the rectangles. We claim that we can determine the answer to any horizontal ray-shooting query emanating from a point $q$ from information stored within $q$’s lowest upper-right point.

What information do we store? First, we label all endpoints as being upper or lower segment endpoints. If the point is a segment lower endpoint, we shoot two rays horizontally to the left and right from this endpoint, and we record the $x$-coordinates where these rays first hit any other vertical segment (see Fig. 5(b)). If the no segment is hit, we set this $x$-coordinate to $-\infty$ or $+\infty$. (We will ignore the cost of this preprocessing, but it is possible to compute it for all the points in $O(n \log n)$ total time.)

Next, we construct an upper-right data structure (as in Problem 4) for these $2n$ points. We answer a ray-shooting query for a point $q = (q_x, q_y)$ as follows:

1. Using the upper-right data structure, find the lowest point $p$ that lies to the upper right of $q$.
2. If $p$ is an upper endpoint, we report that the ray hits some segment (see Fig. 5(c), top).
(3) If \( p \) is a lower endpoint, let \( \ell_x \) and \( r_x \) be the \( x \)-coordinates of the left and right rays shot from \( p \).

(3a) If \( r_x \neq +\infty \), we report that the ray hits some segment (see Fig. 5(c), center).
(3b) If \( \ell_x \geq q_x \), we report that the ray hits some segment (see Fig. 5(c), bottom).

(4) Otherwise, we report that the ray does not hit any segment.

To prove the correctness of this algorithm, we will show that the horizontal ray emanating from \( q \) hits some segment if and only if one of the conditions of the lemma is satisfied. Suppose first that the horizontal ray emanating from \( q \) hits some segment. Translate the ray upwards until it first hits a segment endpoint. If the first endpoint it hits is an upper endpoint, then we satisfy condition (2). If it hits a lower endpoint, there are two cases. If the \( x \)-coordinate of the segment endpoint is to the left of the ray intersection point, then we satisfy condition (3a). Otherwise, if the \( x \)-coordinate of the segment endpoint is to the right of the ray intersection point, then we satisfy condition (3b). Therefore, we will detect the ray intersection. Conversely, suppose that any of the three conditions holds. In all three cases, there is a segment lying to the upper-right of \( q \) that clearly hit (as seen in Fig. 4) provided that the endpoint of this segment lies below \( q_y \). But if the lower endpoint were above \( q_y \), then the upper-right query would have returned this other endpoint instead. This implies that the algorithm is correct.

Because the query algorithm involves a single application of the upper-right query, it has the same running time.