Solution 1:

(a) To perform `insert((6,6))`, we trace the path to this node, falling out on the left child of `(8,8)` and we insert the new node there. By alternation, it is a horizontal \(y\) splitter (see Fig. 1(a)).

(b) To perform `delete((3,6))`, we find it in the tree. Since its right subtree is null, we take the replacement as the node as the point with the smallest \(y\)-coordinate in the left subtree \((1,1)\). We copy \((1,1)\) to \((3,6)\), and recursively delete \((1,1)\) from the left subtree. Then we swap the left and right subtrees of this node (see Fig. 1(b)).

Solution 2:

(a) \(3m + 1\): A single node has four null pointers. Each time we add a new node, we get four new null pointers but lose one, for a net increase of 3. So, the total is \(4 + 3(m - 1) = 3m + 1\).

(b) (1) (larger height) and (4) (rebuild less often): Decreasing the log base from \(3/2\) to \(10/9\) increases the allowed tree height. Also, increasing the scapegoat ratio from \(2/3\) to \(9/10\) means that the larger subtree may have roughly 9 times as many nodes as the smaller one, so we rebuild less often.

(c) (1) 1: The scapegoat update procedure stops after the first subtree is rebuilt.

(d) \((3/5)^2 n = (9/25)n\): To get to level two (or higher), you need to throw two consecutive heads, which occurs with probability \((3/5)^2\). To get the expected number of occurrences, we multiply by \(n\).

(e) (2) \(O(1)\): The expected number of nodes visited in any level in the skip-list search is \(O(1)\).

(f) An important feature of splaying is that it tends to make unbalanced trees more balanced. But if you perform single rotations on a degenerate tree, the height of the tree does not decrease.
(g) (1) \(O(1)\) amortized cost: Since there are only two accesses between access to the popular key, it can drop only a constant number of levels. So, after the first splay, all remaining splays are guaranteed to take \(O(1)\) time.

**Solution 3:**

(a) Every node stores a value \textbf{maxRating}, which stores the maximum rating of all the points in its subtree. If the current node \(p\)’s cell is disjoint from the query range, it does not contribute and we return 0. If it is contained, then every point is considered, and we return \(p.\text{maxRating}\). Otherwise, we check whether \(p\) lies within the range, and if so include it. Finally, we recurse on the two children. The initial call is \texttt{rangeMax(Q, root, rootCell)}.

```java
int rangeMax(Rectangle Q, KDNode p, Rectangle cell) {
    if (p == null) return 0 // fell out of the tree?
    else if (Q.isDisjointFrom(cell)) // no overlap with range?
        return 0
    else if (Q.contains(cell)) // the range contains our entire cell?
        return p.maxRating // include the total rating
    else { // check point, left, and right
        pMax = (Q.contains(p.point) ? p.point.rating : 0)
        lMax = rangeMax(Q, p.left, cell.leftPart(p.cutDim, p.point))
        rMax = rangeMax(Q, p.right, cell.rightPart(p.cutDim, p.point))
        return max(pMax, lMax, rMax)
    }
}
```

The code’s structure, and hence its running time is identical to that of the range counting on kd-trees, which is \(O(\sqrt{n})\).

(b) When the insert function falls out of the tree, we create a new node whose \textbf{maxRating} is set to the newly inserted point’s rating. Otherwise we insert it recursively into the appropriate subtree and update our max rating maxing it with the newly inserted point’s rating.

```java
KDNode insert(Point pt, KDNode p, int cutDim) {
    if (p == null) {
        p = new KDNode(pt, cutDim, pt.rating) // <-- set new node’s rating
        return p
    } else if (p.point.equals(pt)) { // throw Exception("Error - duplicate point")
        throw Exception("Error - duplicate point")
    } else if (pt[cutDim] < p.point[cutDim]) {
        p.left = insert(pt, p.left, (cutDim + 1) % 2)
    } else {
        p.right = insert(pt, p.right, (cutDim + 1) % 2)
    }
    p.maxRating = max(p.maxRating, p.point.rating)) // <-- update this node’s rating
    return p
}
```

**Solution 4:** It is possible to delete all the points between \(a\) and \(b\) with just two splays. We first splay on \(a\), which brings \(a\) to the root and all larger keys are in the right subtree. We then splay on
Node bulkDelete(Key a, Key b) { // removes keys between a and b exclusive
    root.splay(a) // splay a in the entire tree
    root.right.splay(b) // splay b in the right subtree
    root.right.left = null // remove b’s left subtree from the tree
}

Solution 5: We maintain a variable count, which counts the remaining number of items to visit. The algorithm operates essentially in a similar manner as the find operation for skip lists, but rather than using the key as the criteria for moving down a level, it uses the count of the remaining number of elements to be considered. We do not traverse a link if the span exceeds the remaining count. Whenever we traverse a link, we decrement the count by the span. Otherwise we drop down a level. The running time is the same as the standard skip list of $O(\log n)$ expected.

Value getKth(int k) { // get the kth smallest
    int i = topmostLevel // start at topmost nonempty level
    SkipNode p = head // start at head node
    int count = k // number of items remaining
    while (i >= 0) {
        if (p.span[i] <= count) { // don't pass the kth?
            p = p.next[i] // skip forward
            count -= p.span[i] // decrement by number skipped
        }
        else i-- // drop down a level
    }
    return p.key // return final element
}
Solution 6:

(a) The proof is by (strong) induction on the weight. If \( n = 1 \), then the tree consists of a single external node and has height 0. But \( 0 \leq \log_{3/2} 1 = 0 \). For the induction step, we assume the induction hypothesis holds for trees of weight strictly smaller than \( n \), and we will prove it for \( n \). Given a tree of total weight \( n \), let \( n' \) and \( n'' \) denote the weights of its left and right subtrees, and let \( h' \) and \( h'' \) denote their respective heights. Without loss of generality, we may assume that \( n' \geq n'' \). Since the tree is 2-weight balanced, \( n' \leq 2n'' \). The total weight of the tree is \( n = n' + n'' \geq n' + n'/2 = (3/2)n' \), or equivalently \( n' \leq 2n/3 \). Each subtree has strictly smaller weight than \( n \), so by the induction hypothesis \( h' \leq \log_{3/2} n' \) and since \( n'' \leq n' \), the same bound holds for \( h'' \). The entire tree is one level higher than either subtree, and therefore

\[
\begin{align*}
    h &= 1 + \max(h', h'') \\
    &\leq 1 + \log_{3/2} n' \\
    &\leq 1 + \log_{3/2} \frac{2n}{3} \\
    &\leq 1 + \log_{3/2} \frac{2}{3} + \log_{3/2} n.
\end{align*}
\]

Since \( \log_{3/2}(2/3) = -1 \), this implies that \( h \leq \log_{3/2} n \), as desired.

An alternative (but equally valid) approach is to show that any 2-balanced tree of height \( h \) has weight at least \( n \geq (3/2)^h \). This is proved by induction. For the basis case, a tree of height 0 has 1 external node, and so its weight satisfies \( 1 \geq (3/2)^0 \). For the induction step, let us assume this holds for all trees of height less than \( h \), and we will show it for a tree of height \( h \). Let \( n' \) and \( n'' \) denote the weights of the two subtrees, where \( n' \) is the subtree with the greater height. Thus, the tree with weight \( n' \) has height \( h - 1 \), implying that \( n' \geq (3/2)^{h-1} \). By the 2-balance condition, the weight of the other subtree is at least half this, so \( n'' \geq (1/2)(3/2)^{h-1} \). The overall weight is their sum, that is,

\[
n = n' + n'' \geq \left( 1 + \frac{1}{2} \right) \left( \frac{3}{2} \right)^{h-1} = \left( \frac{3}{2} \right) \left( \frac{3}{2} \right)^{h-1} = \left( \frac{3}{2} \right)^h,
\]

as desired.

(b) \( \beta = 1 + 1/\alpha \): In the analysis above, the value 3/2 arises from the fact that the smaller subtree has at least half the weight of the larger, and therefore both subtrees combined have a total weight of at least 3/2 times the larger. For general \( \alpha \), the smaller subtree has at least an \( 1/\alpha \) fraction of the weight of the larger, and therefore the total tree has weight of \( 1 + 1/\alpha \) times the weight of the larger, and so \( \beta = 1 + 1/\alpha \). (Note that when \( \alpha = 2 \), this yields \( \beta = 3/2 \).)

(c) True: The proof is the same as for scapegoat trees. The weight of each subtree decreases by a factor of at least 2/3 with every level of descent in the tree, and thus you cannot descend more than \( \log_{3/2} n \) levels before the weight is smaller than 1.