Solution 1: There are a number of approaches that lead to a valid splay tree, but to get full credit, you must use the algorithm given in class.

(a) To insert 5, we first perform `splay(5)`. Since 5 is not in the tree, we fall out on visiting node 4. Thus, we splay at 4. This performs a zig-zag rotation, which brings 4 to the root. Since 4 < 5, we insert 5 in between 4 and 6 on the right side of 4.

(b) To delete 6, we first perform `splay(6)`. This performs a zig rotation to bring 6 to the root. Next, we perform `splay(6)` but on 6’s right child. Since 6 is not in this tree, we fall out on visiting node 7. We perform a single zig rotation at 7 (recall that we are limited to 6’s right subtree), which brings 7 to the root of this subtree. Finally, we remove the node 6 from the root and replace it with 7.

Solution 2:

(a) (4): There are no significant consequences. The priorities affect search times, and if just two priorities are the same, the effect is minimal.

(b) $O(n)$ worst case. This is a consequence of the Scanning Theorem for splay trees. (Partial credit for $O(n \log n)$, since this is bound given by the Balance Theorem.)

(c) A non-root node in a B-tree of order $m$ has from $\lceil m/2 \rceil$ to $m$ children, and one fewer key than children. Thus, for $m = 13$, we have from 7 up to 13 children and from 6 up to 12 keys per node.

(d) True: Following an insertion, an arbitrary number of nodes on the search path might satisfy the scapegoat condition. Even though insertions are vigilant against scapegoat nodes, deletions can wreak havoc with the tree’s structure (up until the point that they induce a rebuild of the entire tree).

(e) False: The scapegoat insertion algorithm stops as soon as it performs its first rebuild.
(f) The main disadvantage of quadtrees is that each node has $2^d$ children. So, in 10 dimensional space, each internal node would have over a thousand children!

(g) $O(\sqrt{n})$: Any axis-parallel line can intersect at most $O(\sqrt{n})$ cells.

(h) $O(n)$: If the line is not axis parallel, it can intersect all of the cells.

Solution 3:

(a) Let $s$ denote the leftmost node in the tree. The size of the root is $n$. By left-heaviness, the size of its left child is at least $(2/3)n$, the size of its left-left grandchild is $(2/3)((2/3)n) = (2/3)^2n$, and generally the size of its $d$-fold left descendant is at least $(2/3)^d n$.

By left-heaviness, we may assume that $s$ is a leaf, and therefore $\text{size}(s) = 1$. Letting $d$ denote $s$'s depth, we have $1 = \text{size}(s) \geq (2/3)^d n$. Solving for $d$, we have $(3/2)^d \geq n$, which implies that $d \geq \log_{3/2} n$, as desired. (This proof is not formally correct, since the left-heaviness condition only applies if $\text{size}(s) \geq 3$, but we can correct this by adjusting the constant $c$.)

(b) Let $t$ denote the rightmost node in the tree. The size of the root is $n$. By left-heaviness, the size of its left child is at least $(2/3)n$, and therefore the size of its right child is at most $n - 1 - (2/3)n \leq n/3$. The size of its right-right grandchild is at most $(1/3)((1/3)n) = (1/3)^2n$, and generally the size of its $d$-fold right descendant is at most $(1/3)^d n$.

We have $\text{size}(t) \geq 1$. Letting $d$ denote $t$'s depth, we have $1 \leq \text{size}(t) \leq (1/3)^d n$. Solving for $d$, we have $3^d \leq n$, which implies that $d \leq \log_3 n$, as desired.

Solution 4:

(a) If the cell lies entirely above or entirely to the right of $q$, then no point can be hit by the platform. That is, $\text{cell}.lo.x > q.x \lor \text{cell}.lo.y > q.y$. A practical (but not essential) enhancement would be to check whether the cell lies entirely below $\text{best}$, that is, $\text{cell}.hi.y < \text{best}.y$. If so, no point in the cell can offer a better solution.

(b) In order to provide a better solution, $\text{p}.point$ must lie below the falling platform, that is, $\text{p}.point.x <= q.x \&\& \text{p}.point.y <= q.y$, and it must be above $\text{best}$, that is, $\text{p}.point.y > \text{best}.y$.

(c) If $\text{cell}$ lies entirely to the left of $q$ (that is, $\text{cell}.hi.x <= q.x$) and $\text{p}$'s cutting dimension is 1 (horizontal cut), then we assert that we need visit only one of $\text{p}$'s children. If $\text{p}$'s point lies below $q$, that is, $\text{p}.point.y < q.y$, then the falling segment will hit $\text{p}.point$ before any point in its left (lower) subtree. So we don’t need to recurse on the left child. On the other hand, if $\text{p}$’s point lies above $q$, then the right (upper) subtree is entirely out of range, and we don’t need to recurse on the right child.

(d) The entire helper function is given below.

```java
Point platformDrop(Point q, KDNode p, Rectangle cell, Point best) {
    if (p == null) return best;
    if (cell.lo.x > q.x || cell.lo.y > q.y) return best;
    ...
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)
Rectangle rightCell = cell.rightPart(p.cutDim, p.point)
if (cell.hi.x <= q.x && p.cutDim == 1) {
    if (p.point.y > q.y) best = platformDrop(q, p.left, leftCell, best)
    else best = platformDrop(q, p.right, rightCell, best)
} else {
    best = platformDrop(q, p.right, rightCell, best)
    best = platformDrop(q, p.left, leftCell, best)
}
return bests
}

(e) The initial call is platformDrop(q, root, bbox, (-infinity, -infinity), where root is
the root of the kd-tree, bbox is the kd-tree’s bounding box.

Solution 5:

(a) We consider two cases. First, if p.left != null, then the inorder predecessor is the right-
most node in p’s left subtree (see Fig. 2(a)). We take one step to the left and then follow
right-child links as far as possible. On the other hand, if p.left is null, then the inorder
predecessor is found by walking up along left-child links as far as possible. When we can go
no farther, the inorder predecessor is the parent of this node (see Fig. 2(b)). If the parent is
null, we have hit the root, and there is no inorder predecessor.

Node inorderPred(Node p) {
    if (p.left != null) { // p as a left subtree
        p = p.left
        while (p.right != null) // find rightmost node in p’s left subtree
            p = p.right
        return p
    } else { // no left subtree
        while (p.parent != null && p == p.parent.left) {
            p = p.parent // move up the left-child chain
        }
        return p.parent
    }
}

(b) The cases are analogous to the binary tree, but a bit messier. First, we consider whether p is a
leaf node (p.child[0] == null). If so, but this is not the leftmost key in the leaf(i > 0), we
return the key immediately to its left in the same node (p.key[i-1]). (See Fig. 2(c).) Other-
wise, we travel up along the chain of leftmost-children (while p == p.parent.child[0]).
When we can go no further, we return the parent key just before us (see Fig. 2(d)). To find
this key we go up to the parent and enumerate its children.

On the other hand, if p is not a leaf, then we access the subtree immediately to its left
(p.child[i]), and then walk down the rightmost chain in this subtree (p.child[p.nc-1])
until we can go no further, and we return the rightmost key is this leaf node (p.key[p.nc-2]).
Figure 2: Inorder predecessor.

The code is given below. We provide utility functions to return the rightmost child and rightmost key in a node.

BNode rightmostChild(p) { return p.child[p.nc-1] } // p's rightmost child
Key rightmostKey(p) { return p.key[p.nc-2] } // p's rightmost key

Key inorderPred(BNode p, int i) {
    if (p.child[0] == null) {
        // p is a leaf
        if (i > 0) // this is not the leftmost key
            return p.key[i-1] // return the prior key
        else { // we are the leftmost key in a leaf?
            while (p.parent != null && p == p.parent.child[0])
                p = p.parent // travel up leftmost child chain
            if (p.parent == null) // hit the root?
                return null // no prior key
            else {
                q = p.parent // key lies in our parent
                return {the largest key in q that is < p.key[i]}
            }
        }
    } else { // not a leaf?
        p = p.child[i] // go to our left subtree
        while (rightmostChild(p) != null) // descend to the leaf level
            p = rightmostChild(p) // along the rightmost chain
        return rightmostKey(p) // return the rightmost key
    }
}