Solutions to Homework 3

Solution 1: See the sequence of operations shown in Fig. 1. We first find the node 8 in the tree by a standard binary-search tree descent. The node 8 is a left-left grandchild, so we perform a zig-zig operation performing a right rotation at the grandparent 11 followed by a right rotation at its parent 9. Now, 8 is a right-left grandchild, so we perform a zig-zag operation by performing a right rotation at its parent 12 followed by a left rotation at its (former) grandparent 7. Now, 8 is the right child of the root, and we perform a single zig operation by performing a left rotation at the root. Finally, 8 is the new root and we are done.

Figure 1: Splaying.

Solution 2:

(a) The insertion of “p” results in a node at depth 7, and a tree of size $n = 16$. This triggers a rebuild event since $7 > \log_{3/2} 16 \approx 6.8$ (see Fig. 2).

(b) Tracing the search path towards the root, the ratios $\text{size(p.child)}/\text{size(p)}$ shown next to the nodes along the search path (shown with heavy blue edges). The first node to exceed the $2/3$ limit is “k”, where $\text{size(p.child)}/\text{size(p)} = 9/12 > 2/3$.

(c) We traverse the subtree rooted at “k” inorder. This results in a sorted list of twelve keys $(i, j, k, \ell, m, n, o, p, q, r, s, t)$. The first median is “o” at index $\lfloor 12/2 \rfloor = 6$, which becomes the root of this subtree. This produces two sublists $(i, j, k, \ell, m, n)$ and $(p, q, r, s, t)$ of sizes 6 and 5, respectively. The former is split at the median “l” at index $\lfloor 6/2 \rfloor = 3$, and the latter at median “r” at index $\lfloor 5/3 \rfloor = 2$. We continue in this manner until we produce the tree shown in Fig. 2, which we link in to replace the subtree rooted at “p”.

Solution 3: The answer to [“You fill this in”] is $1/\alpha$. Here is the full lemma.
Lemma: Given a binary search tree of \( n \) nodes and any constant \( \alpha > 1 \), if there exists a node \( p \) such that \( \text{depth}(p) > \log_\alpha n \), then \( p \) has an ancestor (possibly \( p \) itself) such that
\[
\frac{\text{size}(u.\text{child})}{\text{size}(u)} > \frac{1}{\alpha}.
\]

Proof: The proof is by contradiction. Suppose to the contrary that no node from \( p \) to the root satisfies the above inequality. This means that for every ancestor node \( u \) from \( p \) to the root, we have \( \text{size}(u.\text{child}) \leq \frac{1}{\alpha} \cdot \text{size}(u) \).

We know that the root has a size of \( n \). Therefore, its child on the search path has size at most \((1/\alpha)n\), its grandchild has size at most \((1/\alpha)((1/\alpha)n) = (1/\alpha^2)n\), and generally the node at depth \( i \) along the search path as size at most \((1/\alpha)^i n\).

Let \( d \) denote the depth of \( p \). We know what its subtree rooted at \( p \) must have at least one node (namely \( p \) itself), and therefore
\[
1 \leq \text{size}(p) \leq \left(\frac{1}{\alpha}\right)^d n.
\]

Solving for \( d \), we have
\[
\alpha^d \leq n \implies d \leq \log_\alpha n.
\]

However, this violates our hypothesis that \( p \)'s depth exceeds \( \log_\alpha n \), yielding the desired contradiction.

Solution 4: Given the scalar \( x_0 \), we seek the point of the kd-tree with the smallest \( x \)-coordinate that lies on or to the right of \( x_0 \). Below we give a simple recursive solution. We recursively traverse the nodes of the kd-tree, keeping track of each node's cell. (In fact, the problem can be solved without reference to a node's cell, but we will express the answer in this form, since this is illustrative of common kd-tree query algorithms.) Let \( p \) denote the current node and let \( \text{cell} \) denote the associated rectangular cell, which is the region of space in which all of \( p \)'s descendant points reside. Our recursive helper will have the signature \text{Point \ vertLineSlide(Scalar \ x0, KDNode

![Figure 2: Insertion into a scapegoat tree.](image-url)
p, Rectangle cell), and the initial call is Point vertLineSlide(x0, root, bbox), where root is the tree’s root and bbox is the bounding rectangle for the entire point set.

Our recursive helper works as follows. First, if we fall out of the tree (p == null) we return null. Otherwise, if p.point.x ≥ x0, we consider p.point to be a possible candidate. Next, we consult p’s cutting dimension. If the cutting dimension is 0 (a vertical splitter), we consider the relationship between p.point and the vertical line x = x0. If p.point lies to the left of the line, we know that its left subtree lies entirely to the left of the vertical line, and so it may be safely ignored, but we will recurse on the right child (see Fig. 3(a)). If p.point lies on or to the right of the vertical line, then we assert that its right subtree may be safely ignored, since every point in this subtree lies to the right of p.point (which itself is a valid answer), and so no point in this subtree can be the answer. We recurse only on the left subtree (see Fig. 3(b)). In either case, we visit only one of its two children.

![Figure 3: Sliding line queries.](image)

On the other hand, if the cutting dimension is 1 (a horizontal splitter), then we cannot generally determine whether the answer lies in the lower cell or the upper cell, so we will try both and take the best of them (see Fig. 3(c)). To help us out, let’s assume we have access to a helper function min[x](Point q1, Point q2), which returns the non-null point among its arguments that has the minimum x-coordinate. If both points are null, it returns null. The pseudocode is given below.

```plaintext
Point vertLineSlide(double x0, KDNode p, Rectangle cell) {
    if (p == null) return null // fell out of tree - ignore
    Point q = (p.point.x >= x0 ? p.point : null) // initial candidate point
    if (p.cutDim == 0) { // vertical splitter?
        if (p.point.x < x0) // p.point is left?
            return min[x](q, // ... search right subtree
                vertLineSlide(x0, p.right, cell.rightPart(p.cutDim, p.point)))
        else { // p.point is right?
            return min[x](q, // ... search left subtree
                vertLineSlide(x0, p.left, cell.leftPart (p.cutDim, p.point)))
        }
    } else { // horizontal splitter?
        return min[x](q, // ... search both sides
            vertLineSlide(x0, p.left, cell.leftPart (p.cutDim, p.point)),
            vertLineSlide(x0, p.right, cell.rightPart(p.cutDim, p.point)))
    }
}
```

Hey, we never made use of the node’s cell! Yes, this is true. We could eliminated all reference to the cell, and it would be completely correct. This is not always the case, however, and
so it was useful to derive the pseudocode in its general form as practice for other kd-tree query algorithms.

We claim that the query time is $O(\sqrt{n})$, assuming that the kd-tree is balanced and we alternate cutting dimensions between $x$ and $y$ with each level. This follows directly from the analysis given in the lecture notes and the fact that we only recurse on nodes whose cells are stabbed by the vertical line $x = x_0$.

For completeness, here is a more detailed analysis of the query time. If a node is an $x$-splitter (vertical), then we visit one of its two children. If the node is a $y$-splitter (horizontal), then we make recursive calls on both of its children. Thus, for every two levels of descent in the recursion tree, we will make a total of two recursive calls. Assuming that the tree is balanced, the sizes of the subtrees decrease by a factor of $1/4$ with every two levels of descent. We spend a constant time at each node visited. Therefore, up to constant factors, the query time is given by the recurrence:

$$T(n) = 2T(n/4) + 1.$$ 

As shown in class (or using the Master Theorem for recurrences), this solves to $T(n) = O(\sqrt{n})$.

**Solution to the Challenge Problem:**

(a) Let $p$ denote the root node initially, but as the following operation proceed, $p$ always points to the same node. We repeatedly apply the operation $s \leftarrow S.pop()$. If $s = \perp$, we exit the operation. Otherwise, we apply the following rotation depending on what $s$ is:

**LL:** Let $q \leftarrow p.right$. Do a left rotation at $p$ followed by a left rotation at $q$. (The RR case is symmetrical.)

**LR:** Do a left rotation at $p$ following by a right rotation at $p$. (The RL case is symmetrical.)

**L:** Do a left rotation at $p$.

(b) The reason that $\perp$ is important in part (a) is that it signals when to end the rotations. However, if we know that every node starts from the leaf level, we can keep popping elements off the stack, perform the rotations as indicated in (a), and stop when node $p$ becomes a leaf. Thus, there is no need for the $\perp$ symbol.