Solutions to Practice Problems for Midterm 2

Disclaimer: These solutions have not been carefully proofread. If anything appears fishy, please let me know.

Solution 1:

(a) The symmetrical trees are (2) AVL trees, (3) red-black trees, (5) scapegoat trees, and (6) splay trees. Leftist heaps are not symmetrical because of the leftist condition, and AA trees are not symmetrical because the skew condition implies that left subtrees are never deeper than right subtrees.

(b) Splay tree: This sort of efficiency is called static optimality (where the access probabilities are non-uniform but do not change over time). Among the dictionary data structures we saw this semester, only the splay tree is efficient with respect to static optimality, since it restructures itself so that more frequently accessed keys are placed near the root of the tree.

(c) In $d$-dimensional space, each node in a point quadtree stores one point and has up to $2^d$ children. In dimension 3, this means 8 children. The root level has one node, level 1 has 8 nodes, level 2 has 64 nodes, and generally, level $i$ has $8^i$ nodes. Since each node stores a single point, the maximum number of points in a tree of height $h$ is

$$\sum_{i=0}^{h} 8^i = \frac{8^{h+1} - 1}{8 - 1} = \frac{8^{h+1} - 1}{7} \approx 8^h.$$ 

(d) The expected height is $O(\log n)$. The analysis is essentially the same as that of a standard (unbalanced) binary search tree, since, irrespective of the dimension, the next splitting chosen randomly from among the points to be inserted into the subtree.

(e) Remarkably, the expected height is still $O(\log n)$, although with about twice the constant factor. While each $x$-split insertion is entirely degenerate, each $y$-insertion is essentially random. Thus, there are in expectation $O(\log n)$ $y$-cutting levels, and hence the total height is roughly twice this. We can get a more formal analysis in the special case where the $y$-splits are perfectly balanced. Then with every two levels of the kd-tree, we decrease the points by half. Thus, the number of levels satisfies the recurrence $T(n) = 2 + T(n/2)$, and this solves to $T(n) = 2 \log n$.

(f) The scapegoat tree is preferred because its height is guaranteed to be $O(\log n)$, so the find search time is also guaranteed to be $O(\log n)$. In the splay tree, the search time is amortized $O(\log n)$.

(g) With standard binary search trees, the expectation was over all $n!$ insertion orders. With skip lists, the expectation was over all $n!$ orders of the random choices. The latter is preferred, because the data structure’s expected performance is not dependent on the insertion order.
Recall that in order to reach level $i$, a node must throw $i$ consecutive heads, which occurs with probability $1/2^i$. Therefore, there are $n/2^i$ such nodes in expectation, which yields $n/8$ for $i = 3$. (Observe that half of these nodes, $n/16$, terminate at this level and the rest continue to higher levels.)

Solution 2: Let us assume that we want to construct a tree for the array $A[0..size-1]$. There are two major differences with respect to the standard rebuilding process. First, the splitter is chosen as the mean element, that is, $s = (A[0] + A[size - 1])/2$. Second, rather than splitting at the median, we find the smallest element $A[m] \geq s$, and partition the array as $A[0..m-1]$ (for the left subtree) and $A[m..size-1]$ (for the right subtree). We assume we have a function $A.subList(0,m)$ and $A.subList(m, size)$ to extract these sub-lists. The construction is presented in the code block below. The initial call is $buildSubtree(A)$, where $A$ is the full array.

```java
Node buildSubtree(List<Key> A) { // geometrically balanced tree
    size = A.size
    if (size == 1) // one key?
        return new ExternalNode(A[i]) // ... external node
    else {
        Key s = (A[i] + A[size-1])/2 // split at midpoint
        m = 0
        while (m < size && A[m] < s) m++ // partition
        Node left = buildTree(list.subList(0,m)) // build subtrees
        Node right = buildTree(list.subList(m,size))
        return new InternalNode(s, left, right); // combine them
    }
}
```

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 $size(s) = 1$. Letting $d$ denote $s$'s depth, we have $1 = size(s) \geq (2/3)^d n$. 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 $size(s) \geq 3$, 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 $size(t) \geq 1$. Letting $d$ denote $t$'s depth, we have $1 \leq size(t) \leq (1/3)^d n$. Solving for $d$, we have $3^d \leq n$, which implies that $d \leq \log_3 n$, as desired.

Solution 4: We start at level 0, since we know that every node exists at this level. The search involves two phases, called up-phase and down-phase. During the up-phase, we try to move up to
higher levels if possible. If not, we follow the next pointer. If we see that the next link leads us beyond the search key \( y \), we initiate the down-phase. This is the same as the standard skip-list search. It attempts to move forward if the move does not take us beyond the search key. If it would, we instead move down a level. The process terminates when the level number becomes negative. If the key is in the structure, it will be at this final node (see Fig. 4).

```java
Node fingerSearch(Node p, Key y) { // search from p to y
    int i = p.next.length - 1; // top level of p
    while (p.next[i].key <= y) { // up phase
        p = p.next[i]
        i = p.next.length - 1
    }
    while (i >= 0) { // down phase
        if (p.next[i].key <= y) {
            p = p.next[i]
        } else i--
    }
    return p; // return final node
}
```

![Figure 1: Finger search in a skip list.](image)

We will not analyze the running time, but intuitively, we expect to make a constant number of hops at each level, and each time we move up a level, we are doubling the distance traveled. Thus, if there \( m \) nodes between our starting and ending node, we expect to spend \( O(\log m) \) hops in the up-phase and \( O(\log m) \) hops in the down-phase.

**Solution 5:**

(a) We assume we have an object called Strip, where strip.lo and strip.hi are the strip’s left and right endpoints, respectively. Let’s assume we have two utility functions. The first tests whether a rectangular cell is disjoint from the strip, and the second determines whether a point is contained in the strip.

```java
boolean inStrip(Point pt, double lo, double hi) { return pt.x >= lo && pt.x <= hi }

boolean isDisjoint(Rect r, double lo, double hi) { return r.lo.x > hi || r.hi.x < lo }
```

We apply the standard approach for answering range searching queries. We visit nodes of the kd-tree recursively. Let \( p \) denote the node currently being visited. If we fall out of the tree or
if the cell is disjoint from the strip, we return the current best. If the node’s point is in the
strip and higher than the current best, it replaces best. Otherwise, we construct the two child
cells. If the node is a vertical splitter, we search both subtrees. If it is a horizontal splitter,
we try the right (upper) subtree first. If the left subtree is still relevant to the search, then we
visit it. The initial call at the root level is \text{partialMax}(\text{strip}, \text{root}, \text{bbox}, \text{null}), where
\text{bbox} is the kd-tree’s bounding box.

Point partialMax(double lo, double hi, KDNode p, Rect cell, Point best) {
    if (p == null || isDisjoint(cell, lo, hi)) // trivial cases
        return best
    if (inStrip(p.point, lo, hi) && (best == null || p.point.y > best.y))
        best = p.point // new best
    // children cells
    Rect leftCell = cell.leftPart(p.cutDim, p.point)
    Rect rightCell = cell.rightPart(p.cutDim, p.point)

    if (cutDim == 0) { // vertical cut?
        best = partialMax(lo, hi, p.left, leftCell, best) // try both sides
        best = partialMax(lo, hi, p.right, rightCell, best)
    } else { // horizontal cut?
        best = partialMax(lo, hi, p.right, rightCell, best) // try right first
        if (best == null || p.point.y > best.y) // is left relevant?
            best = partialMax(lo, hi, p.left, leftCell, best) // try left
    }
    return best
}

(b) The running time of the algorithm is $O(\sqrt{n})$ under the standard assumptions about kd-trees.
To see this, observe first that, by applying the standard orthogonal range search analysis for
dk-trees, the number of nodes whose cells are stabbed by the two vertical sides of the range is
$O(\sqrt{n})$. The remaining nodes have cells that are either entirely inside or outside the vertical
strip. If the cell is outside the strip, the search will return immediately.

All that remains is to analyze the number of nodes whose cell lies entirely inside the strip.
For each horizontal splitter, we never need to visit the lower child (since the point stored in
this node will provide a larger $y$-coordinate than any point in this subtree). Therefore, we
will visit at most two out of the four grandchildren of any such node. It follows that the
total number of nodes of this last type that are visited by the search satisfies the recurrence
$T(n) = 2T(n/4) + 3$, which by the Master Theorem solves to $O(\sqrt{n})$.

Solution 6: A key observation is that this problem, which apparently has to do with line segments
can be answered by a data structure that just stores points! It is not hard to see that answering a
left-to-right horizontal ray-shooting query at point $q = (q_x, q_y)$ is equivalent to computing the point
of $P$ with the minimum $x$-coordinate that lies within the northeast quadrant of $q$ (see Fig. 2). To
see why, observe that in order to hit any segment, its topmost point must lie to $q$’s right and have
a higher $y$-coordinate, thus it lies in $q$’s northeast quadrant. We seek the first such point, that is,
the one with the lowest $x$-coordinate. A point lying in $q$’s northeast quadrant is called a candidate
(points \{p_8, p_9, p_{10}\} in the figure), and the best candidate is the one with the smallest \(x\)-coordinate, that is, \(p_8\).

We will apply the standard approach for answering range searching queries. We visit nodes of the kd-tree recursively. Let \(p\) denote the node currently being visited, and let \(cell\) denote its associated cell. Let \(best\) denote the best candidate seen so far. The initial call at the root level is \(ray\text{\_}Shoot(q, root, bbox, null)\), where \(bbox\) is the bounding box for the entire tree.

Let \(Q\) denote the region lying to the northeast of \(q\) (shaded in blue in Fig. 2). Let’s assume we have two utility functions. The first tests whether a rectangular cell is disjoint from \(Q\), and the second determines whether a point is contained within \(Q\).

boolean inQ(Point pt, Point q) { return pt.x >= q.x && pt.y >= q.y }

boolean isDisjoint(Rect r, Point q) { return r.hi.x < q.x || r.hi.y < q.y }

If we fall out of the tree or if the cell is disjoint from \(Q\), we return the current best. If the node’s point is in \(Q\) and left of the current best, it replaces best. Otherwise, we construct the two child cells. If the node is a horizontal splitter, we search both subtrees. If it is a vertical splitter, we try the left subtree first. If the right subtree is still relevant to the search, then we visit it. The initial call at the root level is \(ray\text{\_}Shoot(q, root, bbox, null)\), where \(bbox\) is the kd-tree’s bounding box.

Point rayShoot(Point q, KDNode p, Rect cell, Point best) {
    if (p == null || isDisjoint(cell, q)) // trivial cases
        return best
    if (inQ(p.point, q) && (best == null || p.point.x < best.x))
        best = p.point // new best
        // children cells
    Rect leftCell = cell.leftPart(p.cutDim, p.point)
    Rect rightCell = cell.rightPart(p.cutDim, p.point)
    if (cutDim == 1) { // horizontal cut?
        best = rayShoot(q, p.left, leftCell, best) // try both sides
        best = rayShoot(q, p.right, rightCell, best)
    } else {
        best = rayShoot(q, p.left, leftCell, best) // try left first
        if (best == null || p.point.x < best.x) // is right relevant?
            best = rayShoot(q, p.right, rightCell, best) // try right
    }
    return best
}
The running time of the algorithm is $O(\sqrt{n})$ under the standard assumptions about kd-trees. (The proof is similar that of the previous problem.)

**Solution 7:** We create a line with slope of $-1$ and place the points either on or very close to this line. To keep the tree balanced, we insert them in a balanced manner, recursively inserting the median point (see Fig. 3). It is easy to see that the line will stab all the leaf cells of the kd-tree, and hence it stab all the cells (ancestors as well).

![Figure 3: A line that stabs all the cells of a kd-tree.](image)

**Solution 8:** This is true, as shown in the following theorem.

**Theorem:** Given a splay tree $T_0$ and any two keys $x, y \in T$, the trees $T_1$ resulting from $\text{splay}(x); \text{splay}(y)$ and $T_2$ resulting from $(\text{splay}(x); \text{splay}(y))^2$ are identical.

**Proof:** We may assume that $x < y$, since the other case is left-right symmetrical. (This is because all the splay operations are left-right symmetrical.) We assert that, after performing $\text{splay}(x); \text{splay}(y)$, $T_1$ has one of two possible structures:

(a) The root node is $y$ and its left child is $x$ (see Fig. 4(a)).
(b) The root node is $y$ and its left-left grandchild is $x$ (see Fig. 4(b)).

To see this, observe that after $\text{splay}(x)$, $x$ is at the root of the tree. The final rotation of $\text{splay}(y)$ is either Zig (implying that $x$ is now the left child of $y$), Zig-Zig (implying that $y$ was the right-right grandchild and now $x$ is its left-left grandchild), or Zig-Zag (implying that $y$ was $x$’s right-left grandchild, and now $x$ is $y$’s left child).

In case (a), the next $\text{splay}(x); \text{splay}(y)$ will right rotate and then left rotate the root (see Fig. 4(a)), which leaves the tree unchanged. In case (b), the next $\text{splay}(x); \text{splay}(y)$ will Zig-Zig $x$ back up to the root and then Zig-Zig $y$ back up to the root (see Fig. 4(a)). In either case, we wind up back where we started.

Here is an interesting question, which is suggested by the above problem. Consider any sequence of distinct $k \geq 1$ keys, $\langle x_1, \ldots, x_k \rangle$ in a splay tree $T$. Are the trees resulting from $(\text{splay}(x_1) \ldots \text{splay}(x_k))$ and $(\text{splay}(x_1) \ldots \text{splay}(x_k))^2$ always the same? (Honestly, I don’t know the answer.)
Solution 9: In both parts, let us consider what happens during a run that starts with a tree of size $n$, and ends when the number of active entries falls below $n/2$.

(a) To see that find operation takes $O(\log n_a)$, consider any find that takes place during the run. Thus, we have $n/2 \leq n_a \leq n$. The original tree was balanced, so (even with all the inactive nodes clogging things up) any find operation takes time $O(\log n)$. Since $n/2 \leq n_a$, it follows that $\log(n/2) \leq \log n_a$, and since $\log(n/2) = (\log n) - 1$, we have $\log n \leq (\log n_a) + 1$. Therefore, a running time of $O(\log n)$ is also $O(\log n_a)$, as desired.

(b) We assert that the amortized running time of delete is $O(\log n)$. To see this, consider the run. The actual running time of each delete and each find is $O(\log n)$. Initially, all the entries are active, and to reduce the number of active entries below $n/2$, we need to perform at least $n/2$ deletions. If we assess a charge of $t = O(\log n)$ tokens for each operation, we use half of this to pay for the actual cost of the operation, and bank the other half. Thus, we bank a total of at least $t(n/2) = O((n/2) \log n) = O(n \log n)$ tokens during the run. Clearly, we have accumulated enough tokens to pay the $O(n)$ cost to rebuild the tree.

Note that we had quite a bit of excess “slack” in our analysis. We would have collected enough tokens if we had charged just one token for every delete operation. Unfortunately, we cannot assert that the amortized running time is $O(1)$, because the amortized time can never be smaller than the actual time. (After all, it is the average of the actual costs.) The actual time for each delete and each find is $O(\log n)$.