Solutions to the Final Exam

Solution 1:

1. Adoption can be performed if either sibling is a 3-node. Otherwise, you must merge.

2. In an AA tree, a node is red if its parent is at the same level, and otherwise it is black. (The root is always black.)

3. This is a consequence of the Working-Set Theorem. It states that if elements are accessed after a short delay, these accesses are efficient. (I’ll give partial credit for Static-Optimality, since if keys are accessed repeatedly after a small delay, their access probabilities will be high.)

4. It is used in range-reporting queries. Given an interval \([x_{\text{min}}, x_{\text{max}}]\), once you find the leftmost key of the range, you can just walk through the leaf nodes through the use of the next-leaf pointers, until you find an entry that is larger than \(x_{\text{max}}\).

5. Scapegoat trees are guaranteed to have \(O(\log n)\) height, so query operations are unconditionally efficient.

6. The two that guarantee finding a single empty slot are (a) linear probing and (f) double hashing when \(m\) and \(g\) are relatively prime. All the others may visit only a strict subset of entries and so may miss an empty slot. (In quadratic probing, even if the table size is prime, the probing sequence is only guaranteed to hit half of the entries.)

7. The \texttt{size2} field was needed when we were merging with the block before us. We needed to be able to locate the preceding block’s header, and the \texttt{size2} field tells us how large the block is, so we can find it.

Solution 2:

1. Preorder: 3, 2, 1, 7, 5, 4, 6, 14, 11, 9, 8, 10, 12, 15

2. Inorder: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 15

3. Postorder: 1, 2, 4, 6, 5, 8, 10, 9, 12, 11, 15, 14, 7, 3

4. See Fig. 1. The first rotation is a zig-zig (8–9–10), and then a zig-zag (8–14–7), and finally a zig (8–3).

Solution 3:

1. We present below pseudo-code for the helper. The initial call is \textbf{return minPriority}(x_{0}, x_{1}, root). If \(x_{1} < p.key\), the interval \([x_{0}, x_{1}]\) lies entirely to the left this node’s key and we recurse on that side. Similarly, if \(x_{0} > p.key\), the internal lies entirely on the right, and we recurse on that side. Otherwise, this key lies within the interval. If the priorities were arbitrary, then the answer would be obtained by recursing on both subtrees. However, this fails to exploit an important property of treaps. Because treap priorities are heap-ordered, we can infer that the current node has the minimum priority of all the nodes in its subtree, and we just return the current node’s priority.
int minPriority(Key x0, Key x1, TreapNode p) {
    if (p == null) return Integer.MAX_VALUE // fell out of the tree?
    else if (x1 < p.key) // range to left of p.key
        return minPriority(x0, x1, p.left) // ...search left
    else if (x0 > p.key) // range to right of p.key
        return minPriority(x0, x1, p.right) // ...search right
    else // p.key in range?
        return p.priority // this has lowest priority
}

(3.2) In the worst case, the algorithm traverses a single path in the tree, and so the running time is
proportional to the treap’s height, which is $O(\log n)$. The slower solution that recurses on the
two subtrees in the last case, runs in time $O(k + \log n)$, where $k$ is the number of points of $P$
that lie within the interval $[x_0, x_1]$. While this is fairly efficient when $k$ is small, in the worst
case, this is $O(n)$. (There is a totally naive solution which involves visiting both subtrees all
the time, irrespective of $x_0$ and $x_1$. This runs in $O(n)$ time all the time, regardless of the
number of points of $P$ lying within the range.)

**Solution 4:** The substring identifiers and final suffix tree (alternative Patricia-trie drawing)
are shown in Fig. 2. The standard Patricia-trie drawing is shown below. As an example of the
difference, consider the $aab$ grandchild of the root in the alternative drawing. Since the first three
characters $S[0..2]$ of the string have been processed, this node branches at $S[3]$, which is why the
corresponding node is labeled with 3 in the standard drawing.

**Solution 5:**

(5.1) This problem follows the standard form we have seen in other kd-tree range searching
problems. Computing the answer is equivalent to computing the point of $P$ whose $y$-coordinate
lies in the interval $[q_y, q_y + h]$ and whose $x$-coordinate is the smallest that is at least as large
as $q_x$. A point lying in $q$’s vertical stip to the right of the query segment is called a candidate,
and the best candidate is the one with the smallest $x$-coordinate.

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, and let $cell$ denote its cell.
The point best is the best (leftmost) candidate seen so far. The initial call at the root level is `segSlideRight(q, root, boundingBox, sentinel)`, where `boundingBox` is the root cell, and `sentinel` is the point $(+\infty, +\infty)$. If this point is returned from the search, we return `null` as the answer.

When we visit a node p, we consider the relationship of p’s cell to the strip to the right of the segment. If p is null or if its cell does not overlap the strip because it is left of (cell.high.x < q.x) or below (cell.high.y < q.y) or above (cell.low.y > q.y + h), we may ignore this node and its contents since it cannot possibly provide a candidate. Otherwise, we visit the node. If its point lies in range, we check whether it is better (to the left of) the current best, and if so we update the best.

```java
Point segSlideRight(Point q, float h, KDNode p, Rectangle cell, Point best) {
    if (p == null) return best // fell out of tree?
    else if (cell.high.x < q.x || cell.high.y < q.y || cell.low.y > q.y + h)
        return best // no overlap?
    else { // is p.point a candidate?
        if (p.point.x >= q.x && p.point.y >= q.y && p.point.y <= q.y + h) {}
        if (p.point.x < best.x) // better than current best?
            best = p.point
    }
    Rectangle leftCell = cell.leftPart(p.cutDim, p.point) // child cells
    Rectangle rightCell = cell.rightPart(p.cutDim, p.point)
    best = segSlideRight(q, h, p.left, leftCell, best) // recurse on left
    if (cutDim == 1 || best.x > p.point.x) // is right side relevant?
        best = segSlideRight(q, h, p.right, rightCell, best) // recurse on right
    return best
}
```
Rather than blindly making two recursive calls to the left and right subtrees, we first apply the query on the left side, and if the right side is still relevant to the search, we invoke the procedure on this side. (This last check for relevance is not required for full credit, but if you add it, it can be shown the procedure has a running time of $O(\sqrt{n})$.)

(5.2) While this can be solved from scratch, it is much easier to reuse the solution to (5.1). First, by symmetry it is easy to modify the algorithm to slide vertical segments either left or right and to slide horizontal segments either up or down. (We won’t bother giving the details.) Our solution involves invoking this function four times, one for each side of $Q$. We see how far we can slide each of $Q$’s sides, and then take the rectangle defined by the resulting sides. Let us assume that we enhance our Rectangle class to include a few useful functions for returning various corners (e.g., $Q.LR$ is the lower-right corner) and the height and width (e.g., $Q.hgt$ is the height). If there are no points of $P$ within $Q$, the sliding queries will slide over each other. To test this, we check whether the left-side segment has slid beyond the right side of $Q$.

```java
Rectangle minBox(Rectangle Q) {
    // Define and initialize points
    Point pLeft = slideRight(Q.LL, Q.hgt, root, bddBox, Point(+inf, +inf))
    Point pRight = slideLeft (Q.LR, Q.hgt, root, bddBox, Point(-inf, -inf))
    Point pBottom = slideUp (Q.LL, Q.wid, root, bddBox, Point(+inf, +inf))
    Point pTop = slideDown (Q.UL, Q.wid, root, bddBox, Point(-inf, -inf))
    if (pLeft.x > Q.hi.x) return null // no points of Q are inside
    else return Rectangle(Point(pLeft.x, pBottom.y), Point(p.Right.x, p.Top.y))
}
```

Solution 6:

(6.1) The solution is a variant of a problem from HW 3. There are two wrinkles. First, we change the primary range from $x$ to $y$, and second, we slant the discriminating line used in minimization. We will make use of the operation getMin, which was described in the second programming assignment. It runs in time $O(\log n)$, assuming that the tree is balanced.

The range tree is constructed from the original point set $P$, but we will modify how the trees are sorted. The primary tree is sorted based on $y$-coordinates. For each node $p$ of this tree, let $S(p)$ denote the set of points stored in this tree. We generate an auxiliary tree for these points that is sorted by $x + y$. Other than these changes, the range tree is standard, and can be constructed in $O(n \log n)$ time and space. (An alternative is to apply the transformation $T(x, y) = (y, x + y)$.)

Given an SIM query $(y_0, y_1)$, we access the primary tree to identify a collection of $O(\log n)$ subtrees containing the all the points of $P$ that lie within the range $y_0 \leq y \leq y_1$. For each of these subtrees, we apply the operation getMin to the auxiliary tree. (This returns the point $p_i$ of the subtree that minimizes the sum $x_i + y_i$.) Among the $O(\log n)$ resulting points (some of which may be null), we return the one that minimizes the value $x_i + y_i$. 

As we have seen earlier this semester, \texttt{getMin} queries can be answered in \(O(\log n)\) time from a tree of size \(n\). Since each auxiliary tree has size at most \(n\), and there are \(O(\log n)\) auxiliary trees, the overall query time is the product of these, which is \(O(\log^2 n)\).

(6.2) The L-shaped region can be expressed as the union of two rectangles, one forming the upright part of the shape, with lower-left corner \(q\) and upper-right corner \((q_x + w, q_y + \ell)\), and the other forming the horizontal part of the shape with lower-left corner \((q_x + w, q_y)\) and upper-right corner \((q_x + \ell, q_y + w)\) (see Fig. 3(a)). To avoid double counting points that lie on the boundary between these two shapes, we can slightly modify the orthogonal range counting algorithm for the horizontal part so it excludes points on the left edge of the rectangle.

Given a set \(P\) of \(n\) points, we can compute the transformed set \(T(P)\) in \(O(n)\) time, and we can then build the range tree in time and space \(O(n \log n)\). Given a query, we transform it into axis-aligned form in constant time, and then the query can be answered (by the standard range-tree algorithm) in \(O(\log^2 n)\) time.

![Figure 3: Range queries.](image)

(6.3) If you didn’t get this. Don’t feel bad. I threw this on as challenge, knowing that very few people would solve it. But, this problem can be solved using the techniques that we have covered. Here are two solutions:

\textbf{2D range search with binary search:} The idea is to apply binary search to obtain the value of \(s\). For each probe, we generate a trial value for \(s\) and apply an orthogonal range counting query with the box of the given size centered at \(q\). For a given trial value \(s\), we invoke a 2D range search to determine whether the square with corners \((q_x - s, q_y - s)\) and \((q_x + s, q_y + s)\) contains no points in its interior. If so, we know that the box is empty, and \(s\) is too small and if not, the box has at least one point, and \(s\) is too large (or perhaps perfect). We can then apply binary search on \(s\) to get successively more accurate estimates on the final value of \(s\).

The issue with this, however, is binary search with respect to what set of values? You cannot just use arbitrary real numbers, since this might get you arbitrarily close to the answer without arriving at the exact solution. Fixing this problem, is not trivial. (Think about this!) To drive the binary search, you need a discrete set of sorted values, one of which is guaranteed to yield the desired value of \(s\). Here is how to find such a set.

Because we know the optimal square will contain a point of \(P\) on its boundary, we know that \(s\) will have one of the forms \(|q_x - p_{i,x}|\) or \(|q_y - p_{i,y}|\) for some \(p_i \in P\). Our solution
to finding $s$ is to create two 1-dimensional arrays, one storing all the $x$-coordinates of $P$ sorted from left to right, and the other storing $y$-coordinates of $P$ sorted from bottom to top. In $O(\log n)$ time, we can locate $q_x$ and $q_y$ in the respective sorted array. Then we can perform four binary searches, one for the $x$-coordinates lying to the left of $q_x$, one for the $x$-coordinates lying to the right of $q_x$, one for the $y$-coordinates lying below $q_y$, and one for the $y$-coordinates lying above $q_y$. Whichever of these searches yields the largest value of $s$ such that the square of side length $2s$ centered at $q$ contains no points in its interior is the desired answer. This approach involves building a range-tree in 2-dimensional space with $O(n \log n)$ space, and the query time involves four binary searches, each involving a range search. The range search takes $O(\log_2 n)$ time, and so the overall time is $O(\log^3 n)$. (Whew!)

**3D getMin:** This approach is inspired by the challenge problem from HW 3. The idea is to first apply a $45^\circ$ rotation to the plane, which converts the square into a diamond. Now, we can interpret the square as consisting of four $45\text{-}45$ right triangles that share a vertex at $q$ (one for each quadrant) (see Fig. 3(b)). For each, we can apply something like the skewed minimum query of part (6.1). However, we need to filter two, not just one, coordinate.

Let’s assume that the rotation has already been applied. We consider each of the four quadrants about $q$, and for each, we will compute the point of $P$ that minimizes the diagonal distance to $q$. (More formally, we compute the point $q$ within the quadrant that minimizes $|(px - q_x) + (py - q_y)|$.) Let’s consider how to do this for just the northeast quadrant, since the others are symmetrical. Our first-level range tree is sorted on $x$. The second-level range trees are sorted on $y$, and the third level range trees are sorted by the diagonal distance $x + y$. Given $q$, we use the first-level tree to filter out all the points except those whose $x$-coordinates are at least as large as $q_x$. We then use the second-level tree to filter out all the points except those whose $y$-coordinates are at least as large as $q_y$. Finally, we apply a getMin query to each of the third-level subtrees. This generates the closest point with respect to diagonal distance in this quadrant. In order to obtain the final diamond, we take the minimum diagonal distance over all four quadrants. Let $s$ be this distance. We return $s$ and the answer to the query.

Since we are applying a 3-layered search structure, the space needed is $O(n \log^2 n)$ and the query time is $O(\log^3 n)$. 