Problem 1

Describe algorithms to insert and delete points from a kd-tree. In your algorithm you do not need to take care of rebalancing the structure.

Algorithm 1 AddPoint(v: root node, p: new point, d: current depth)

```
if $v$ is a leaf node then
    Determine current hyperplane direction from $d$
    $L \leftarrow$ Hyperplane in this direction at $v$
    Replace $v$ with $L$
    if $p$ is left of $L$ then
        Add $v$ to right of $L$ and $p$ to left of $L$
    else
        Add $v$ to left of $L$ and $p$ to right of $L$
    end if
else
    if $p$ is left of $v$ then
        AddPoint($v_{\text{left}}, p, d+1$)
    else
        AddPoint($v_{\text{right}}, p, d+1$)
    end if
end if
```

Algorithm 2 RemovePoint(v: root node, p: point)

```
if $v$ is a leaf node then
    Delete $v$ and replace with the other child of the parent node
else
    if $p$ is left of $v$ then
        RemovePoint($v_{\text{left}}, p$)
    else
        RemovePoint($v_{\text{right}}, p$)
    end if
end if
```

Both algorithms are $O(\log n)$ (assuming the tree remains balanced) since both are traversals and do constant work at each point.

Problem 2

Describe an algorithm to construct a d-dimensional kd-tree for the points in P. Prove that the tree uses linear storage and that your algorithm takes $O(n \log n)$ time.

Algorithm 3 BuildTree(P: points, D: current depth)

```
if $|P| = 1$ then
    return leaf node storing unique point in $P$
else
    $D \leftarrow D \pmod{d}$
    $L \leftarrow$ hyperplane based on median of $D$'th coordinate in $P$
    Split $P$ into $P_L$ and $P_R$ via $L$
    $v_{\text{left}} \leftarrow$ BuildTree($P_L$, $D + 1$)
    $v_{\text{right}} \leftarrow$ BuildTree($P_R$, $D + 1$)
    return node containing $L$ with $v_{\text{left}}$ and $v_{\text{right}}$ as children
end if
```

The running time of the algorithm can be recursively expressed as

$$T(1) = O(1)$$
$$T(n) = 2T\left(\frac{n}{2}\right) + O(1)$$

This recursive running time is then simply $T(n) = O(n \log n)$. For storage complexity, the number of leaf nodes will be n and the number of inner hyperplanes will be $n - 1$. Therefore the storage complexity is $O(n + n - 1) = O(n)$.

Problem 3

In some applications one is interested only in the number of points that lie in a range rather than in reporting all of them. Such queries are often referred to as range counting queries. In this case one would like to avoid having an additive term of $O(k)$ in the query time.

Part A

Describe how a 1-dimensional range tree can be adapted such that a range counting query can be performed in $O(\log n)$ time. Prove the query time bound.

To adapt a 1-dimensional range tree, add a count variable to every node and initialize it to 0 when constructing the tree. Then after the tree is made, each point in the tree can be queried and, while passing through the tree, increment the count variables of nodes that were passed through. By doing this, the count variable will equal the number of leaf nodes below that node.

To perform the counting query, first find the split node v_{split}. Then we can simply traverse down both sides of v_{split}, accumulating the sizes of every subtree to the

> opposite side of our traversal until we hit a leaf. That is, we continually go left of v_{split}, adding the size of the right node to our total each iteration and vice versa. The final total is the number nodes within the query bounds.
>
> Adding in the count variable to the construction phase does not affect the preprocessing time complexity since it takes $O(n \log n)$ time which is the same as before. Since the query only has to do exactly 2 full root to leaf traversals, the time complexity is $O(\log n + \log n) = O(\log n)$.

Part B

Using the solution to the 1-dimensional problem, describe how d-dimensional range counting queries can be answered in $O(\log^d n)$ time. Prove the query time.

> We endow the d-dimensional tree with the same count variable for all the nodes. A similar pre processing step can then be used in which each node's count tracks the number of node's in contains. Importantly, for each subtree it should only keep track of the number of nodes that it is sorting by as determined by the depth of the subtree compared to the original tree. We can then, like the original d-dimensional query algorithm, pick $O(\log n)$ canonical subsets of points in the correct first coordinate range. This can then be done again after finding the split node, leading to another $O(\log n)$ subsets. This process is done recursively for every coordinate except the last. For the last coordinate, the counting query outlined in (A) can be done to the get desired answer. In total then the time complexity for the counting query is $O(\log^d n)$.

Part C

Describe how fractional cascading can be used to improve the running time with a factor of $O(\log n)$ for 2-and higher-dimensional range counting queries.

> Fractional cascading allows replacing the second to last range tree query followed by 1-d counting query with a 2-d range counting query that can be done in $O(\log n)$ time. This means that the time complexity decreases from $O(\log^2 n)$ to $O(\log n)$. That is, it decreases it by a factor of $O(\log n)$.

Problem 4

Let S_1 be a set of n disjoint horizontal line segments and let S_2 be a set of m disjoint vertical line segments. Give a plane-sweep algorithm that counts in $O((n+m)\log(n+m))$ time how many intersections there are in $S_1 \cup S_2$.

If we keep track of the lines intersecting the sweep line, once a vertical line is reached all intersecting horizontal lines can be found by doing a 1-dimensional range query based on the y-coordinates of the vertical line. This is because every horizontal line currently intersecting the plane sweep line is guranteed to intersect the vertical line if its y-coordinates are within the range of the current vertical line. Therefore the plane sweep algorithm is as follows

Algorithm 4 CountIntersections(S_1, S_2)

```
1. Initialize an intersection count to 0
2. Sort all endpoints of horizontal lines in $S_1$
3. Sort all vertical lines in $S_2$ by their $x$-coordinate
4. Add all sorted points into the event queue $Q$
5. Initialize the status structure $S$ and add the first event point
6. Do the following while $Q$ is not empty
    (a) Pop the next event point $p$ from $Q$
    (b) If $p$ is the start of a horizontal segment, track the $y$-coordinate of the
        segment in $S$
    (c) Else if $p$ is the end of a horizontal segment, remove its $y$-coordinate from
        $S$
    (d) Otherwise, $p$ is a vertical line segment so do a range-query on $S$ be-
        tween the $y$-values of its endpoints. Add these to the running total of
        intersections.
```

As for time complexity, handling either type of segment takes $O(\log(n+m))$ time since modifying the BST for horizontal segments takes that much time and the range query for vertical segments does as well. The plane sweep is also processing $O(n+m)$ points so the total running time is simply $O((n+m)\log(n+m)$.

Problem 5

Let S be a set of n axis-parallel rectangles in the plane. We want to be able to report all rectangles in S that are completely contained in a query rectangle $[x : x'] \times [y : y']$. Describe a data structure for this problem that uses $O(n \log^3 n)$ storage and has $O(\log^4 n + k)$ query time, where k is the number of reported answers.

The important thing to note is that since each rectangle is axis aligned, they can each be identified by just two corner points. Therfore each pair of corner points (x_1, y_1) and (x_2, y_2) can be identified as a point in 4-dimensions of the form (x_1, x_2, y_1, y_2). We can then construct a 4-dimensional range tree that sorts in the x-direction first for x_1 and x_2 and then in the y-direction for y_1 and y_2. The original range query is then transformed into

$$[x : x'] \times [x : x'] \times [y : y'] \times [y : y'].$$

A d-dimensional range tree can be constructed to use $O(n \log^{d-1} n)$ storage and $O(\log^d n + k)$ query time. Therefore in this instance the storage complexity is $O(n \log^3 n)$ and time complexity is $O(\log^4 n + k)$ which was to be shown.