Solution 1:

(a) The next-leaf pointer in B+ trees 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 \( x_{\text{max}} \).

(b) 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) 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 \( O(\log n) \).

(d) 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

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

(e) 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.

(f) 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 \( k \)-d-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 \( T(n) = 2 \log n \).

(g) The two that guarantee finding a single empty slot are (1) linear probing and (6) 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.
Solution 2:

(a) We present below pseudo-code for the helper. The initial call is `return minPriority(x0, x1, root)`. If \( x_1 < p\text{.key} \), the interval \([x_0, x_1]\) lies entirely to the left this node’s key and we recurse on that side. Similarly, if \( x_0 > p\text{.key} \), the interval 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.

```java
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
}
```

(b) 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 3: To expose a node, we first apply a standard descent to find the exposed node, and we set the priority to \(-\infty\) (actually `Integer.MIN_VALUE`). We then walk back up the search path to the root. As we return from a call to expose, the exposed node has replaced the child. Thus, if we apply `expose` to the left subtree, a single right rotation suffices to move it to the current node. We then return this value. (The right side is symmetrical.) An example is shown in Fig. 1.

```java
TreapNode expose(Key x, TreapNode p) {
    if (p == null) // error - key not in tree
        throw Exception("Key not found");
    else if (x < p.key) { // x is smaller - search left
        p.left = expose(x, p.left);
        return rotateRight(p); // rotate the exposed node up
    }
    else if (x > p.key) { // x is larger - search right
        p.right = expose(x, p.right);
        return rotateLeft(p); // rotate the exposed node up
    }
    else { // found it
        p.priority = Integer.MIN_VALUE; // set priority to -infinity
        return p;
    }
}
```
Solution 4:

(a) As in the standard search, we never want to overshoot the key, we move forward only if the key in the next node is \( \leq y \) (\( p.\text{next}[i].\text{key} \leq y \)). In the standard search, the level index decreases monotonically. The difference here is that, whenever possible, we will move up a level. We only move down when needed. This implies that the search moves forward as fast as possible. The code is very similar to the standard find function, except that it starts at level zero, and we include the “move up” option.

\[
\text{Value forwardSearch}(\text{SkipNode} \ p, \text{Key} \ y) \{
\begin{align*}
\text{i} & = 0 \quad \text{// start at the lowest level} \\
\text{while (i >= 0)} \{ \quad \text{// (level will rise then fall)} \\
\quad \text{if (p.\text{next}[i].\text{key} \leq y)} \{ \quad \text{// can we move forward?} \\
\quad \quad \text{if (i+1 < p.\text{next}.\text{length}) i++} \quad \text{// move up, if possible} \\
\quad \quad \text{else p = p.\text{next}[i]} \quad \text{// move horizontal, if not} \\
\quad \quad \text{else i--} \quad \text{// drop down a level}
\}
\text{return (p.\text{key} == y ? p.\text{value} : null) \quad \text{// return value if found}
\}
\]

While it is not exactly apparent from the code, the algorithm has two phases, an up-phase and a down-phase (see Fig. 2). In the up-phase, whenever we have the opportunity to move up a level we do so. This phase continues until we reach a node \( p \) and level \( i \) such that the next link at this level takes us beyond \( y \), that is, \( p.\text{next}[i].\text{key} > y \). After this, we begin the down-phase, which operates in the same manner as the standard skip-list search.

(b) In class, we showed that, given a set of \( n \) nodes in a skip list, with high probability the maximum level of any node will be \( O(\log n) \). This did not depend on any properties of the keys stored in these nodes nor the global properties of the data structure, since each node of a skip list determines its height independent of all other nodes. Therefore, the analysis can be applied to any subset of skip-list nodes, in particular, the \( k \) nodes lying between \( p \) and \( y \).

(c) We will show that the algorithm visits a constant number of nodes at each level. In particular, in expectation, it visits two nodes on each level during the up-phase, and two nodes at each level during the down phase.
Up-Phase: During the up-phase, whenever we arrive at a node, we move up to the next higher level if the node contributes to this higher level. There is a 1/2 probability of this event occurring. Therefore, the expected number of nodes visited at this level is exactly the same as the expected number of coin tosses made until first seeing heads, which is 2.

Down-Phase: Because the down-phase is the same as the standard search process in skip lists, the analysis of the down-phase is the same as the one given in class (based on a backwards analysis), which shows that the expected number of nodes visited per level is 2.

By (b), the maximum level that we will visit in expectation is $O(\log k)$, and by (c), the number of hops at any level is $2 + 2 = 4$, so therefore, the total time for the search is $O(4\log k) = O(\log k)$, as desired.

Solution 5:

(a) The algorithm operates essentially the same as the find operation for skip lists. The main difference is that, whenever we follow a link, we count the number of elements that the link spans. The other modification is that, rather than advancing when $p\text{.next}[i]\.key \leq x$, we instead use $p\text{.next}[i]\.key < x$. (Alternatively, you could leave the condition the same, but decrement the count by 1 if the key is found.)

```c
int countSmaller(Key x) {
    int i = topmostLevel // start at topmost nonempty level
    SkipNode p = head // start at head node
    int count = 0 // number of smaller elements
    while (i >= 0) { // while levels remain
        if (p.next[i]\.key < x) {
            count += p.span[i] // count number of skipped items
            p = p.next[i] // advance along same level
        } else i-- // drop down a level
    }
    return count // return final count
}
```
(b) The algorithm operates essentially in a similar manner as the find operation for skip lists, but rather than using the key as the criteria for moving down a level, it uses the count of the remaining number of elements to be considered. We maintain a variable `count`, which counts the remaining number of items to visit. Whenever we traverse a link, we decrement the count by the number of nodes that this link spans. If the link spans more than the remaining number, we drop down a level.

```java
Value getMinK(int k) {
    int i = topmostLevel // start at topmost nonempty level
    SkipNode p = head // start at head node
    int count = k // number of items remaining
    while (count > 0) { // repeat until exhausting the span
        if (p.span[i] <= count) {
            count -= p.span[i] // decrement count by the number spanned
            p = p.next[i] // advance along same level
        } else i-- // drop down a level
    }
    return p.key // return final element
}
```

**Solution 6:** 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. 3(a)).

(b) The root node is $y$ and its left-left grandchild is $x$ (see Fig. 3(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. 3(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. 3(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, $(x_1, \ldots, x_k)$ 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 7:**

(a) We go up to the parent and determine which of its children is $p$. We then respond with the next child, if this child exists. Clearly, this takes constant time.

```java
Node23 rightSibling(Node23 p) {
    q = p.parent;
    if (q == null) return null; // root node has no sibling
    else {
        if (p == q.child[0]) { // p is child #1?
            return q.child[1]; // answer is child #2
        } else if (q.nChildren >= 3 && p == q.child[1]) { // p is child #2?
            return q.child[2]; // answer is child #3
        } else
            return null; // no child following p
    }
}
```

(b) We walk back towards the root, as long as we are the rightmost child of our parent. We then go to our right sibling and walk down along the leftmost child the same number of levels. We ascend the tree and then descend, so the running time is proportional to the tree’s height, which is $O(\log n)$. There is an elegant recursive implementation of this idea. If a node has a right child, then its right child is its level successor. If not, its level successor is the leftmost child of the level successor of its parent. (By our assumption that all leaves are at the same level, if the parent’s level successor is non-null, its leftmost child exists.)

```java
Node23 levelSuccessor(Node23 p) {
    if (p == null) return null;
    else if (rightSibling(p) != null) return rightSibling(p);
    else {
        q = levelSuccessor(p.parent);
        if (q == null) return null
        else return q.child[0]
    }
}
```
There are at most $n$ nodes on any level and each invocation of `levelSuccessor` takes $O(\log n)$ time, so $O(n \log n)$ is an obvious upper bound. However, it is not a tight bound. Suppose we consider the worst-case of starting at the leftmost leaf node. The various invocations of `levelSuccessor` visit every edge of the tree twice, once moving up the edge and once moving down. (Trace the code and you will see this easily.) Since a tree with $n$ nodes has $n-1$ edges, it follows that the running time is just $O(n)$.

**Solution 8:** 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. Since we do not store a node’s cell with each node, we pass the cell into the function as the parameter `cell`. We maintain a point `best`, which among all candidates seen so far, has the largest $x$-coordinate. The recursive function is given in the code block below.

```java
Point partialMax(double x1, double x2, KDNode p, Rectangle cell, Point best) {
    if (p == null) // fell out of tree?
        return best;
    else if (cell.low.x > x2 || cell.high.x < x1) // no overlap with strip
        return best;
    else {
        if (p.point.x >= x1 && p.point.x <= x2) // p.point in the strip?
            if (p.point.y > best.y) best = p.point; // p.point is better?

        // get children cells
        Rectangle leftCell = cell.leftPart(p.cutDim, p.point);
        Rectangle rightCell = cell.rightPart(p.cutDim, p.point);

        best = partialMax(q, p.left, leftCell, best); // search left subtree
        best = partialMax(q, p.right, rightCell, best); // search right subtree
        return best;
    }
}
```

There are a couple of further refinements we could make to the above algorithm to improve its efficiency. First, if the cell’s cutting dimension is $y$ (horizontal), we should recurse on the high child before the low child, since it is more likely to yield a point with a higher $y$-coordinate. Second, if cell lies entirely below `best` (that is, `cell.high.y < best.y`), there is no need to visit this cell, since any point it can provide will be worse than the current best.

With these two refinements, the running time of the algorithm is $O(\sqrt{n})$ under the assumption that the tree is balanced. To see this, observe first that, by applying the standard orthogonal range search analysis for kd-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. So, all that remains is to analyze the number of nodes whose cell lies entirely inside the strip. For each $y$-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 $O(\sqrt{n})$.

Recall that the tree is associated with a bounding box cell that includes all the points of the kd-tree. This is the root’s cell. The initial call at the root level is \texttt{partialMax(q, root, boundingBox, initialBest)}, where \texttt{initialBest} can be taken to be a sentinel point with the \texttt{y-coordinate} $-\infty$.

**Solution 9:** 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. 4). It is easy to see that the line will stab all the leaf cells of the kd-tree, and hence it stabs all the cells (ancestors as well).

![Figure 4: A line that stabs all the cells of a kd-tree.](image)

**Solution 10:** A cool 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. 5). 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.

![Figure 5: Horizontal ray shooting via the kd-tree.](image)

We make the simplifying assumption that no point in the kd-tree has the same $x$- or $y$-coordinate.
as q. 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. We pass in p's cell into the function as the parameter cell. The point best is the best among all candidates seen so far. The initial call at the root level is rayShoot(q, root, boundingBox, sentinel), where sentinel is the point (+∞, +∞). 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 q's northeast quadrant. If p is null or if its cell does not overlap the quadrant (cell.high.x < q.x or cell.high.y < q.y), we may ignore this node and its contents, since it cannot possibly provide a candidate. If p's point is eligible (p.point.x >= q.x && p.point.y >= q.y) and it's x-coordinate is smaller than the current best (p.point.x < best.x), it replaces the new best.

Before applying the recursive calls, we make one additional observation. Suppose that the cell lies entirely above q and it is a vertical splitter (cell.lo.y >= q.y && p.cutDim == 0). Then we do not need to visit both of p's children. If p lies to the right of q (p.point.x > q.x) there is no need to search the right child, since we will hit p.point before any point on the right side. On the other hand, if p lies to the left of q (p.point.x <= q.x) there is no need to search the left child, since it does not overlap the search region. The final code is given below.

```java
Point rayShoot(Point q, KDNode p, Rectangle cell, Point best) {
    if (p == null) // fell out of tree?
        return best
    else if (cell.high.x < q.x || cell.high.y < q.y) // no overlap
        return best
    else {
        if (p.point.x >= q.x && p.point.y >= q.y && p.point.x < best.x)
            best = p.point // p.point is new best
        // get child cells
        Rectangle leftCell = cell.leftPart(p.cutDim, p.point)
        Rectangle rightCell = cell.rightPart(p.cutDim, p.point)

        if (cell.lo.y >= q.y && p.cutDim == 0) {
            if (p.point.x > q.x) // no need to search right
                best = rayShoot(q, p.left, leftCell, best)
            else // no need to search left
                best = rayShoot(q, p.right, rightCell, best)
        } else {
            best = rayShoot(q, p.left, leftCell, best)
            best = rayShoot(q, p.right, rightCell, best)
        }
        return best;
    }
}
```

The running time of the algorithm is $O(\sqrt{n})$ under the assumption that the tree is balanced. This follows from the observation that in order for us to recurse on a node, it’s cell must be stabbed either by the vertical line passing through q or the horizontal line passing through q, and as shown in lecture, at most $O(\sqrt{n})$ nodes have this property.
Solution 11: We will give just a high-level explanation of the “hole-free” method of deletion for hashing with linear probing. The insertion process works just as for standard linear-probing hashing. When we wish to delete a key $x$, we first compute the hash value $h(x)$ and then start searching the probe sequence $h(x), h(x) + 1, h(x) + 2, \ldots$ until we find the key to be deleted. (If we don’t find it, we throw an exception.) Suppose that the cell holding the deleted key is at $table[i]$. We set $table[i] = empty$ to indicate that the key is deleted, but this leaves a hole in our table, which needs to be filled.

```
// assume that table[i] is to be deleted
int p = i // index of empty slot to fill
int q = p+1 // index of candidate to fill it
while (table[q] != empty) {
    while (table[q] != empty && h[table[q]] > p)) q += 1
    if (table[q] != empty) {
        table[p] = table[q] // fill the hole
        p = q // the hole moves to q
        q = p+1
    }
}
table[p] = empty
```

To fill the hole, we consider the subsequent probe sequence $i + 1, i + 2, \ldots$ (wrapping around if necessary) until we first find an empty cell, say at index $j$. (To simplify the presentation, let’s assume that we didn’t need to wrap around, implying that $j > i$.) An overly simple idea is to shift each entry in the subarray $table[i+1 \ldots j-1]$ back one position to fill in the hole. There is a problem with this, however. We never want to move a key to a position that lies before its ideal location (that is, the position indicated by the hash function). So, we will maintain two indices, $p$, which stores the index of the hole to fill and $q$, which stores the index of the element to move into the hole. If $h[table[q]] > p$, then we cannot move this entry back to slot $p$, and so we increment $q$ and try again. The pseudocode is given in the code block. We assume that there is no wrap-around.