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 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.

(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

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

(c) 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 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 $T(n) = 2 \log n$.

Solution 2:

(a) We present below pseudo-code for the helper. The initial call is return minPriority(x0, x1, 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
}
```

(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.

```
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) The size of a parent is strictly larger than the size of either of its children, so

$$\frac{\text{size(u.child)}}{\text{size(u)}} < 1,$$
which implies that \( \alpha < 1 \). We also claim that we cannot have \( \alpha < 1/2 \). Suppose we did. If \( n \) were even, then excluding the root, one child must at least \( \lceil (n - 1)/2 \rceil \) of the remaining nodes. Since \( n - 1 \) is odd, we have \( \lceil (n - 1)/2 \rceil = n/2 > \alpha n \), implying that this subtree would violate the size condition, a contradiction.

(b) If every node satisfies the balance condition, then the height of the tree would be at most \( \log_{1/\alpha} n \). To see this, let \( h \) denote the height of the tree. Consider any path in the tree from the root to a node of depth \( h \). The root has size \( n \). By the balance condition, the child of this node along this path has size at most \( \alpha n \), the grandchild along the path has size at most \( \alpha^2 n \), and generally a node at level \( h \) has size at most \( \alpha^h n \). This leaf node defines a subtree with 1 node. Therefore, we have \( \alpha^h n \geq 1 \). Equivalently, we have \( (1/\alpha)^h \leq n \), and taking logs base \( 1/\alpha \) on both sides, we have \( h \leq \log_{1/\alpha} n \).

**Solution 5:** 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 \( \text{table}[i] \). We set \( \text{table}[i] = \text{empty} \) to indicate that the key is deleted, but this leaves a hole in our table, which needs to be filled.

```cpp
// 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 \( \text{table}[i+1 \ldots j-1] \) 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 \( h[\text{table}[q]] > p \), 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.

**Solution 6:** 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 \( \text{cell} \). We maintain a point \( \text{best} \), which among all candidates seen so far, has the largest \( x \)-coordinate. The recursive function is given in the code block below.

```python
def partialMax(x1, x2, p, cell, 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
            leftCell = cell.leftPart(p.cutDim, p.point);
            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 \( \text{best} \) (that is, \( \text{cell.high.y < best.y} \)), 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 `partialMax(q, root, boundingBox, initialBest)`, where `initialBest` can be taken to be a sentinel point with the $y$-coordinate $-\infty$.

**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. 2). 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 2: A line that stabs all the cells of a kd-tree.](image)

**Solution 8:** 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. 3). 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 3: 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 \( \text{cell} \). The point \( \text{best} \) is the best among all candidates seen so far. The initial call at the root level is \( \text{rayShoot}(q, \text{root}, \text{boundingBox}, \text{sentinel}) \), where \( \text{sentinel} \) is the point \((+\infty, +\infty)\). If this point is returned from the search, we return \( \text{null} \) 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 \((\text{cell.high.x < q.x \text{ or cell.high.y < q.y)}\), we may ignore this node and its contents, since it cannot possibly provide a candidate.

```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's point is in quadrant?
            if (p.point.x < best.x) best = p.point; // p's point is better?

            Rectangle leftCell = cell.leftPart(p.cutDim, p.point);
            Rectangle rightCell = cell.rightPart(p.cutDim, p.point);

            best = rayShoot(q, p.left, leftCell, best); // search left subtree
            best = rayShoot(q, p.right, rightCell, best); // search right subtree

            return best;
    }
}
```

Otherwise, we first consider whether the point stored in this node offers a better choice (that is, \( p \).point lies within the quadrant and has a smaller \( x \)-coordinate than \( \text{best} \)). If so, we update \( \text{best} \). Finally, we recurse on \( p \)'s two children, and keep the point with the smaller \( x \)-coordinate. We use the utility functions \( \text{leftPart} \) and \( \text{rightPart} \) to compute the cells associated with the left and right children.

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 \( x \) (vertical), we should recurse on the left child before the right child, since it is more likely to yield a point with a lower \( x \)-coordinate. Second, if cell lies entirely to the right of \( \text{best} \) (that is, \( \text{cell.low.x > best.x} \)), there is no need to visit this cell, since any candidate it provides cannot be better 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. (See Solution 6 for an explanation of why.)