Solution 1:

(1.1) See Fig. 1. The key “X” arrives at index 13 and after three probes it is inserted at index 15. The key “Y” arrives at index 15 and the probe sequence wraps around and after 6 probes is inserted at index 4. The key Z arrives at index 9 and after 14 probes (and wrapping around) it is inserted at index 6. (If you counted the number of probes consistently one smaller, namely as 2, 5, and 13, we will give full credit.)

![Hashing with linear probing.](image1)

(1.2) See Fig. 2. The keys “M” and “D” follow are both successfully inserted after two and three probes, respectively. In the case of “Q” we attempted to insert the key at the locations

$$9 + \{0, 1, 4, 9, 16, 25, 36\} = (9 + \{0, 1, 4, 9, 6, 10\}) \mod 15,$$

and all were occupied. No other indices will be checked because these include all the quadratic residues of 15 (offset by 9), and so the insertion fails.

![Hashing with quadratic probing.](image2)

Solution 2:

(2.1) This follows by a straightforward application of the *pigeonhole principle*. If we distribute $nm$ keys over the $m$ entries of the hash table, at least one of the entries must contain at least $n$ keys. Select any such table entry, and let $S$ denote the subset of keys that hashed to this entry. This subset satisfies the desired properties.
(2.2) If we have the bad luck of the user inserting all the elements of $S$ into the hash table, they will all hash to the same location, and will all be added to the same linked list. When the $i$th element of $S$ is added, we need to take $O(i)$ time to traverse the linked list (which is necessary to verify that there are no duplicates). Thus, the total insertion time is proportional to $\sum_{i=1}^{n} i = O(n^2)$.

Solution 3:

(3.1) The helper function is described below. It follows the standard template for a range search in a kd-tree, except that whenever we see that a point lies within $S(q,r)$, we can immediately return false. The initial call is `emptySquare(q, r, root, root.cell)`, where `root.cell` is any rectangle that contains all the points of the tree, e.g., $(-\infty, -\infty)$ to $(+\infty, +\infty)$.

```java
boolean emptySquare(Point2D q, float r, KDNode p, Rectangle2D cell)
    if (p == null) return true // fell out of tree?
    else if (cell is disjoint from S(q,r)) return true
    else if (p.point lies within S(q,r)) return false
    else
        return emptySquare(q, r, p.left, cell.leftPart(p.cutDim, p.point)) &&
        emptySquare(q, r, p.right, cell.rightPart(p.cutDim, p.point))
```

Here are a few addition notes about the solution:

**No containment case?** We have omitted the case where the cell is contained in $S(q,r)$. This could certainly be added without affecting the asymptotic running time, but it is redundant. Because a node’s cell contains the point itself, the node’s cell is contained within $S(q,r)$ only if `p.point` is contained in $S(q,r)$, and the point test is a bit quicker.

**Left before right?** We could make the function a bit faster (in practice) by adjusting the order of the recursive calls in the final case. In particular, we would recurse first on the side that closer to $q$. (For example, if the splitter is vertical and $q.x > p.point.x$ we should first recurse first on the right child, then the left.) While this does not alter the worst-case time, it will likely run faster in practice because it favors the child that is more likely to provide a point in the square and hence lead to termination faster.

**Reduction to counting:** An alternative approach is to reduce this to orthogonal range counting. We could simply applying the orthogonal range counting algorithm from Lecture 13 to count the number of points in the query square and check that this is nonzero. While this perfectly correct and would not affect the worst-case query time, it could be significantly slower in practice, especially when the square contains many points. In the above function, as soon as any recursive call returns `false`, the entire chain of recursive calls leading here would return `false`, and we are done.

(3.2) The running time is basically the same as the $O(\sqrt{n})$ running time for orthogonal range search queries as presented in class. The principal difference in this case is that if the cell is contained within the range (even if the point itself is contained in the range) we can immediately return `false`. Thus, the running time is no greater.

(3.3) There are a number of approaches that can be taken for this problem. The solution given in (3.1) works just fine, but by replacing references to “$S(q,r)$” with “$L(q, w, h)$”. Since the shape has changed, the running-time analysis will need to be modified.
An alternative, which simplifies the analysis is to reduce this to emptiness queries on two semi-infinite rectangles. Let $R^\uparrow$ be the rectangle whose lower left corner is $q$ and whose upper right corner is $(q_x + w, +\infty)$, and let $R\rightarrow$ be the rectangle whose lower left corner is $(q_x + w, q_y)$ and whose upper right corner is $(+\infty, q_y + h)$ (see Fig. 3(b)). Clearly $L$ is empty if and only if both $R^\uparrow$ and $R\rightarrow$ are empty. Thus, rather than designing a new algorithm, we should instead modify the above $\text{emptySquare}$ function to work for arbitrary rectangles, and then invoke it twice. Clearly, the asymptotic running time will be the same.

(3.4) If we did not employ the trick of reducing to two empty rectangle queries, we can still adapt the analysis from the orthogonal range search queries. Recall that in that analysis, we argued that there are essentially three cases: cells that are disjoint from the query region (easy), cells that are contained in the query region (easy), and cells that overlap the boundary of the query region (that is, are “stabbed”). We argued in class that rather than count the cells that are stabbed by the actual query range, we could count the number stabbed by the infinite lines bounding the query range. There are four lines bounding a rectangle, so it sufficed to show that there $O(\sqrt{n})$ cells that are stabbed by any (infinite) line horizontal or vertical line. We then multiply by four, one for each side of the query rectangle.

In our case, the same argument applies. An “L”-shaped region is bounded by four lines (two horizontal and two vertical), and so the exact same argument could be applied. If the query region is bounded by any constant number of horizontal and vertical lines, the same analysis can be applied.

Solution 4:

(4.1) The intuition is that the first layer will filter out all the points except those lying in the horizontal strip $q_y \leq y \leq q_y + h$, and the second layer performs find-larger using $q_x$. Here are the details. The main tree is sorted based on $y$-coordinates. For each node of this tree, the auxiliary tree is sorted by $x$-coordinates and supports find-larger queries. Both the main and auxiliary trees can be any standard balanced binary search tree, and following the analysis given in class, the space is $O(n \log n)$.

Given the query $(q, h)$, we first apply range search to the main tree to obtain a collection of $O(\log n)$ nodes that cover the $y$ interval $[q_y, q_y + h]$. For each of these nodes, we then query the associated auxiliary tree by performing a find-larger query with $q_x$. Each tree will
either return null (there is no point larger than $q_x$ in this subtree) or it will return a viable candidate. If all the auxiliary trees return null, we return the same. Otherwise, among all the viable candidates, we return the one with the smallest $x$-coordinate as the final answer.

It takes $O(\log n)$ time to identify the subtrees from the first range tree. For each of these, it takes $O(\log n)$ time to perform the find-larger query, and hence the total query time is $O(\log^2 n)$.

(4.2) Two sides of V-shape are slanted. It will make a life simpler if we first rotate the space so that these sides are aligned with the axes. To do this, let’s rotate space by 45° clockwise (about the origin), so the V-shape is now the northeast quadrant of the rotated query point. From now on, let’s assume that the point set $P$ has been rotated in this manner. The data structure consists of a main tree that is sorted on $x$, auxiliary trees sorted on $y$, and finally, each node of the $y$-tree will have an auxiliary tree sorted on a new coordinate $z = x + y$. (That is, for each rotated point $p = (p_x, p_y)$, we will transform it to the point $p' = (p_x, p_y, p_z)$, where $p_z = p_x + p_y$, and we will build a 3-layer tree for these 3-dimensional points.)

This is a 3-layer range tree, and so by the result mentioned in class, the total storage is $O(n \log^2 n)$.

Given a query point $q$, the query is answered as follows. First, we rotate $q$ to the point $q'$. We apply the search to the main ($x$) tree to obtain a collection of $O(\log n)$ nodes that cover the semi-infinite interval $x \geq q'_x$ (keeping just the points east of $q'$). For each of these nodes, we then query the associated auxiliary ($y$) tree to obtain a collection of $O(\log n)$ nodes that cover the semi-infinite interval $y \geq q'_y$ (keeping just the points north of $q'$).

Now, all the points lying in the resulting subtrees lie in the northeast quadrant of $q'$, and so all are viable candidates for the answer to the query. For each of the selected nodes from the $y$-auxiliary trees, we invoke a find-larger query on the associated auxiliary ($z = x + y$) tree with the query value $q'_x + q'_y$. In each subtree, we either obtain null (there is no point in this subtree that is northeast of $q'$) or returns the first point northeast of $q'$ in the slanted direction. If the results of all these queries are null, we return null as the final answer. Otherwise, among all the non-null results, we return the one that has the smallest $z = x + y$ component as the final answer.

How long does the query take? The top level ($x$) search identifies $O(\log n)$ nodes. For each of these, we invoke the search on its auxiliary ($y$) tree to obtain $O(\log n)$ nodes, now a total of $O(\log^2 n)$ nodes. For each of these, we invoke a find-larger query in its auxiliary ($z = x + y$) tree. Each of these queries takes $O(\log n)$ time. So, the total time is $O(\log^3 n)$. (Whew! If you got this one, congratulations! You can consider yourself a range-tree pro.)

Solution to the Challenge Problem: How to fake the initialization of an $m$-element table in constant time? The problem is that the table’s initial contents themselves may have been set up by a adversary, who is trying to trick us. So, we cannot trust any of its contents.

Here is the trick. We will create an auxiliary array which we will fill up like a stack. This stack will indicate which entries of the table have been set a value. For example, whenever we perform

---

1A simpler equivalent solution is to skip the rotation, an just map each point $(p_x, p_y)$ directly to the triple $(p_y - p_x, p_y + p_x, p_y)$. This has exactly the same effect and works for integer coordinates. Also it does not require rotating the query point.
the operation \texttt{set}(i,x) we can push \( i \) onto our stack. The stack starts out empty (e.g., \( \text{top} = -1 \)), and we can always trust the contents of \( \text{stack}[j] \), for \( 0 \leq j \leq \text{top} \). Note as well that we don't need to initialize the entire stack array. It suffices to set \( \text{top} = -1 \), which can be done in \( O(1) \) time.

So, whenever we wish to perform an the operation \texttt{get}(i), we just need to check whether \( i \) appears somewhere in our stack. If so, we know the entry is defined and we return \( \text{table}[i] \). Otherwise, it isn't and we return the default initial value \( f(i) \). Thus, the stack entries serve to certify which elements of the table are defined.

Hey, this is way too slow! Obviously, we cannot take the time to search the stack with each access. Here we employ a trick we learned in Programming Assignment 0. We create an array \( \text{locator}[m] \). Whenever we perform an operation \texttt{set}(i,x), we know that index \( i \) will be pushed on our stack. The purpose of \( \text{locator}[i] \) is to give us the index in the stack where this certification is stored, that is, if \( \text{stack}[j] = i \) then \( \text{locator}[i] = j \), or equivalently, \( \text{stack}[\text{locator}[i]] = i \). The locator entries are initially garbage, so we should check that \( \text{locator}[i] \) lies in the range from 0 to \( \text{top} \). If so, it points into the stack. If the corresponding stack entry points back to the locator, we know that the locator value can be trusted. So, the locator gives us an efficient way to find the corresponding entry in the stack.

So, here is a more formal definition of our operations. Let us assume for simplicity that \( 0 \leq i \leq m - 1 \) and that each entry is set only once. (It is easy to add the changes to make this fully general.)

**Initialization:** Set \( \text{top} = -1 \).

\texttt{set}(i,x): Push \( i \) on the stack (say it is at index \( j \)), set \( \text{locator}[i] = j \) and \( \text{table}[i] = x \).

\texttt{get}(i): Check that \( \text{locator}[i] \) lies in the interval \([0, \text{top}]\). If this is the case, next check that \( \text{stack}[\text{locator}[i]] = i \). If so, we have certified that this entry of the table has been set, and we return \( \text{table}[i] \). Otherwise, we return the default value \( f[i] \).

In all, in addition to the table we needed on parallel array \( \text{locator} \) and one stack of integer. Pretty slick!

There are alternative solutions, by the way. You could create a hash table in which to store the table entries. However, initializing the hash table will take time. So, instead you could use the rehashing approach described in class. Start with a constant sized hash table and build up gradually. This will be efficient on average (assuming you use a good randomized hash function) and amortized (due to rehashing). You might also consider creating a bit vector indicating which entries are set. But notice that the initialization time is still too high, because the space needed is \( m/b \), where \( b \) is the number of bits (e.g., 32) in a computer word. Yet another idea is to create a parallel certifying array, which you will set to some hash value. For example, when you set \( \text{table}[i] = x \), set \( \text{certify}[i] = h(i) \), where \( h(i) \) is a randomized hash function. While it is possible that the garbage value stored at this entry of memory matches \( h(i) \), it is quite unlikely, so the probability of making an error (returning garbage when the entry has not been defined) is small, but still nonzero.