Solution 1:

(1.1) See Fig. 1(a). Key “X” is inserted successfully after 3 probes. Key “Y” is inserted successfully after 4 probes after wrapping around the end of the table. Key “Z” fails, since it goes into an infinite loop.

(1.2) See Fig. 1(a). Key “R” is inserted successfully after 6 probes. Key “B” is deleted successfully after 7 probes. We mark the deleted spot with the special symbol “∅”. Key “D” is inserted successfully after 4 probes, reusing the empty slot.

Solution 2:

(2.1) (a) Recall that we keep growing the size of a skip-list tower as long as the coin keeps coming up heads. So, if the probability of heads increases, the towers will tend to be higher, and this will result in more storage. If the increase in probability is by a constant factor, the increase in the tower size will be by an additive constant, but multiplying this times all the $n$ keys in the data structure results in a constant factor increase in the space requirements.

(2.2) (b) This is a tricky one to reason about without getting into the details. One effect that tends to reduce the query time is fact that we will visit fewer nodes at each level of the skip list. In the query-time analysis, we argued that the probability of remaining at the same level was proportional to $1 - p$, where $p$ was the probability of throwing a heads. So, increasing the heads probability implies that we are decreasing the probability of staying at the current level, which implies that there are fewer visits on average per level. This leads to faster query times. On the other hand, another effect the maximum level in the tree increases, which will increase the highest level in the tree. The latter increase is just by an additive amount, and so the first effect is more significant, since it applies at every level of the search.

(2.3) Because B-trees have variable-width nodes (depending the degree) it is possible to set the degree so that each node fits within a page of external memory. Since accessing each page involves considerable latency delay, the fewer page accesses the faster the query time.
2.4) 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 $O(\log n)$.

2.5) The treap is preferred because the variation in performance is due to the random number generator, not the order in which keys are inserted or deleted.

2.6) Hashing supports dictionary operations in $O(1)$ time in expectation (assuming that the load factor is balanced). So, it would be preferred over the AVL tree for the standard dictionary operations find, insert and delete.

2.7) The AVL tree is an ordered dictionary. So, it supports operations such as getMin and find-Larger, which would require $O(n)$ time to perform using a hash table.

Solution 3:

(3.1) See Fig. 2. The first operation is a zig-zig, the second is a zig-zag, and the third is a zig-zig.

![Splay trees](image)

Figure 2: Splay trees

(3.2) The depths of nodes $a$ and $b$ have both increased by $+2$.

(3.3) We will show that, given any splay tree $T$ and any node $x$, after performing $\text{splay}(x)$, if the depth of a node increases, the increase is at most $+2$. To show this, we will consider the effect that each rotation operation (zig-zig, zig-zag, or zig) has on each of the nodes of the tree (see Fig. 3). Let $p$ denote the node that is being splayed. The proof is based on the following observations:

- The descendants of $p$ either remain at the same depth or their depths decrease.
- The descendants of $p$’s parent and grandparent either remain the same or increase by at most $+2$, but after the operation these nodes are now descendants of $p$.
- The depths of the other nodes in the tree are unaffected.

It follows that there only one chance in the course of the splay for a node’s depth to increase, and when this happens, the increase is by at most $+2$. After this, the node is a descendant of the splayed node and so its depth can only decrease. Thus, the maximum increase in depth is $+2$. 

2
Solution 4:

(4.1) Every node stores float field weight, which stores the total weight of all the points in this cell. The initial call is weightedRange(R, root, rootCell). The principal difference over the standard range counting query is that whenever the cell or point lies within the range, we add its weight (not just count) to the result.

```java
def weightedRange(Rectangle R, KDNode p, Rectangle cell):
    if p == null:
        return 0  # fell out of the tree?
    else if R.isDisjointFrom(cell):
        return 0
    else if R.contains(cell):
        return p.weight  # include the total weight
    else:
        result = 0
        if R.contains(p.point):
            result += p.weight
        result += rangeCount(R, p.left, cell.leftPart(p.cutDim, p.point))
        result += rangeCount(R, p.right, cell.rightPart(p.cutDim, p.point))
        return result
```

(4.2) The code is structurally equivalent to the standard range-counting query. Thus, it visits exactly the same nodes as the standard range-counting query. Thus, the $O(\sqrt{n})$ analysis applies here as well.

(4.3) The helper is passed in the current best, and returns the updated best. The initial call is frnn(q, r, root, rootCell, null). If we fall out of the tree or our cell is outside the query disk, we return best. Otherwise, we compute how close we need to be to be the new best, called viableDist. If the point in this cell is close enough, it replaces best. Finally, we recurse on our two children, updating best along the way.

```java
Point frnn(Point q, float r, KDNode p, Rectangle cell, Point best):
    if p == null or cell.distanceTo(q) >= r:
        return best  # not viable
    else:
        viableDist = computeViableDist(p, q, r)
        if cell.contains(q):
            result = frnn(q, r, p.left, cell.leftPart(p.cutDim, p.point), best)
        else:
            result = frnn(q, r, p.right, cell.rightPart(p.cutDim, p.point), best)
        return result
```

Figure 3: Splay tree rotations.
else {
    float bestDist = (best == null ? INFINITY : best.distanceTo(q))
    float viableDist = min(r, bestDist)  // distance to be viable
    if (dist(q, p.point) < viableDist)  // p.point is better?
        best = p.point  // it’s the new best
                // recurse on children
    best = frnn(q, r, p.left, cell.leftPart(p.cutDim, p.point), best)
    best = frnn(q, r, p.right, cell.rightPart(p.cutDim, p.point), best)
    return best
}

A further enhancement would be to order the recursive calls on the children favoring the side on which q lies.

Solution 5:

(5.1) The simplest solution is to count number of points in $S(q, r_1)$ and $S(q, r_2)$ using standard range counting and check that these counts are equal. If so, the annulus is empty. This is a standard 2-level query, so the space is $O(n \log n)$ and the query time is $O(\log^2 n)$.

(5.2) It suffices to perform two ray-shooting queries. One against the A segments and one against the B segments, and then return the closest to q. Let’s just describe the B case, since the A case is symmetrical.

The main tree is sorted on y, and the auxiliary trees are sorted on x and can answer find-larger queries. To answer a query, we apply the main tree to identify $O(\log n)$ subtrees that cover all the points $(p_x, p_y)$ such that $p_y \geq q_y$. Any of these segments will be hit by the query ray. We then invoke the x-auxiliary tree for to compute find-larger of $q_x$, which returns the first segment hit by the ray. (Note that we first filtered based on y and performed a find-larger on x. If we had filtered first on x and then on y, we would need to make an additional pass to determine which of the surviving points has the smallest x-coordinate. Given that the final trees are sorted by y-coordinate, this is nontrivial. Our solution avoids the need for this third pass.)

In the end, we return the answer (from A and B, which the smallest x-coordinate. This is a standard 2-level structure, so the space is $O(n \log n)$ and the query time is $O(\log^2 n)$.