Solution 1:

(a) True: Every non-leaf node has at least one incoming thread, one from each child that is non-null. For example, if the left child of some node \( u \) is not null, then the rightmost leaf in the subtree \( u . \text{left} \) has \( u \) as its inorder successor, and so its forward inorder thread pointing to \( u \).

(b) (2) and (3): In an inorder traversal, internal and external node alternate with each other. The first node is the leftmost leaf. (4) is almost true, but fails only in the case where the tree consists of a single external node.

(c) 4 nodes (but partial credit for any value in the range 3–7): A naive argument is that the third largest element cannot reside below depth two, so there are at most 7 nodes to consider. A more refined argument observes that the root can never be the third largest element. It can be either of the children of the root (whichever child is smaller), and it can also be either of the children of the larger child of the root. So, the best you hope to achieve is to inspect the two children of the root, and the two children of the smaller child of the root, for a total of four nodes.

(d) (3) and (5): A replacement node is needed whenever the node containing the deleted key has two non-null children. (Replacements are never needed for leaves and may not be needed for the root if it has only one child.) Selecting the replacement only from the right subtree can lead to less balanced trees over time, and so even though it is a common convention, selecting exclusively from the right subtree is not an optimal strategy. It is a bit surprising to note that at most one replacement will be needed per deletion. A replacement node is either the largest key in the left subtree or the smallest key in the right subtree. Such a node can have at most one child. Hence, once a replacement is performed, the node to be recursively deleted has just a single child and does not need a replacement!

(e) 2 (assuming a double rotation is counted as two): In AVL tree-insertion, after the first rotation operation (single or double) the subtree to which the rotation is applied has exactly the same height it did prior to the insertion. It follows that this subtree and all the others in the tree are properly balanced with respect to the AVL height criteria.

(f) \( O(\log n) \): In AVL-tree deletion, rotations may propagate from the leaf to the root. Since the tree has \( O(\log n) \) height, this bounds the maximum number of rotations.

(g) Min: 7, Max: 13 (The minimum is a complete binary tree of height 2, and the other is a complete ternary tree of height 2.)

(h) 1: Once a key-rotation (adoption) is performed, the tree structure is restored

(i) Skew: The skew operation enforces the right-child constraint. (In contrast, the split operation is used to enforce the condition that if a node is red, then both its children are black.)
(j) $L$: You may need to perform a skew at every level of the tree.

(k) A finger search is one where, rather than starting at the root of the tree, the search starts from an existing entry in the tree. (This might be the last node visited in the previous search. For example, imagine that you want to look up “house” in a dictionary (book), but just prior to this you had looked up the word “hose”.) Ideally, the search should exploit the fact that you are already close to the target.

(l) If the second hash function and table size share a common factor, then the probe sequence may not visit every entry of the table, and hence insertion may fail even when there are available empty slots in the table. (For example, $m = 10$, $h(x) = 3$, and $g(x) = 5$, the probe sequence will consist of indices of the form $(3 + 5 \cdot i) \mod 10 = \langle 3, 8, 3, 8, \ldots \rangle$. If these two positions are filled, then the insertion fails.)

(m) Hashing does not support ordered dictionary operations. Operations such as finding the largest, smallest, next-larger/smaller, and range searching are not efficiently supported by hash tables, but almost all of our tree-based structures support these in $O(\log n)$ time.

(n) The reason for storing the size field is for the purpose of merging blocks together. When a used block becomes free, it needs to see whether the immediately preceding block is free. The prevInUse bit tells us whether it is free or not, but if it is we need to find its header. The size field tells us the block’s size, and by offsetting by that amount, we can find the previous block’s header.

(o) $m - 1$: The suffix tree has $m$ external nodes. Due to compression, each internal node has at least two children. The number of internal nodes is maximized when every internal node is binary, and we have shown that an extended binary tree with $m$ external nodes has $m - 1$ internal nodes.

(p) (1), (4), (5): A Bloom filter can suffer false positives, but no false negatives. So, if $x \in X$, it must report this correctly, but if $x \notin X$, it may (with low probability) report incorrectly that $x$ is in the set. The running time is $O(k)$ in the worst case. (Evaluate $k$ hash functions and check these bits.) The correctness is randomized, but the running time is worst-case.

**Solution 2:**

(a) Our helper function is `printMaxK(Node p, int k)`, which prints the largest $k$ nodes from the subtree rooted at $p$. If $k$ is not positive, we print nothing. The initial call is `printMaxK(root, k)`. Subtracting the size of the right subtree from $k$ leaves the number of nodes remaining to be printed. (The remainder may be negative, but if so, nothing is printed.)

Because we invoke the function on left, then this node, then right, the keys will be printed in ascending order.

```java
void printMaxK(Node p, int k) { // print max k
    if (p != null && k > 0) // something to print?
        int rightSize = (p.right == null ? 0 : p.right.size) // size of p.right
        int remainder = k - rightSize // remainder after p.right
        if (remainder > 0)
```
printMaxK(p.left, remainder - 1) // print left keys
print(p.key) // print this node
printMaxK(p.right, k) // print right keys
}

(b) We assert that the running time is $O(k + \log n)$. To see this, observe that there are two ways we might visit a node. First, we visit it to print its key. The number of such nodes is $k$, and (since we do $O(1)$ work in each node) the time spent visiting all these nodes is $O(k)$. Otherwise, we visit the node but do not print its contents. This happens when the right subtree has $k$ or fewer keys. If so, we make a recursive call on its right subtree only. Since the tree's height is $O(\log n)$, the number of times we can do this is $O(\log n)$. So, the total running time is $O(k + \log n)$.

(c) The helper function is called printEvenOdd(Node p, int index), where index indicates the index of this key in the sequence. We print a key if the index value is odd, and we increment the index each time we visit a node. We return the updated index after visiting a subtree (which is a bit sneaky). The initial call is printEvenOdd(root, 1). It easy to see that this runs in $O(n)$ time.

```java
int printEvenOdd(Node p, int index) {
    if (p == null) return index // nothing to print
    else {
        index = printEvenOdd(p.left, index) // print left subtree
        if (index % 2 == 1) print(p.key) // print current if odd
        index += 1
        return printEvenOdd(p.right, index) // print the right subtree
    }
}
```

Solution 3: While this could be done by making multiple passes over the tree (which is what most people did), we'll present a very simple procedure that works using essentially a single postorder traversal of the tree.

As usual, we will make use of a recursive helper function boolean validAVL(AVLNode p, int lo, int hi). In order to verify that the nodes are in order, we will provide our helper function an (open) interval $(lo, hi)$ such that all the keys in the subtree must lie strictly within the interval. Whenever we visit a node with key value, say $x$, the keys of the left subtree must lie in the subinterval $(lo, x)$ and the keys of the right subtree must lie in $(x, hi)$.

Why do we need this interval? Note that it is not sufficient to just compare parent-child relationships alone, e.g., $p.left.key < p.key < p.right.key$. To see why, suppose that $p.key = 2$, $p.left.key = 1$, and $p.left.right.key = 3$. This combination passes all the parent-child tests, but it is not valid because $p.left.right.key$ should be smaller than $p.key$, since it is in $p$'s left subtree.

The helper function height(p) returns the height field if $p$ is non-null and $-1$ otherwise. The function idealHeight(p) returns the ideal height of $p$ based on the heights of its children.

The main helper validAVL(p, lo, hi) makes the following checks:

- $p$'s height equals the ideal height (else return false)
- $lo < p.key < hi$ (else return false)
• The absolute height difference between \( p \)'s subtrees is at most 1 (else return false)

• If the above tests are passed, then we recursively check that \( p \)'s left and right subtrees are valid

The initial call is validAVL(root, -INFINITE, +INFINITE). The pseudocode is presented in the following code block.

```java
int height(AVLNode p) { return (p == null ? -1 : p.height); } // height utilities
int idealHeight(AVLNode p) { return 1 + max(height(p.left), height(p.right)); }

boolean validAVL(AVLNode p, int lo, int hi) {
    if (p == null) { // empty tree?
        return true; // ... is always valid
    } else if ((p.height != idealHeight(p)) || // bad height?
               (abs(height(p.left) - height(p.right)) > 1)) || // imbalanced?
               (p.key <= lo || p.key >= hi)) { // key out of range?
        return false;
    } else { // check our subtrees
        return validAVL(p.left, lo, p.key) &&
               validAVL(p.right, p.key, hi);
    }
}
```

Solution 4:
(a) Every node stores double field weight, which stores the total weight of all the points in this cell. The initial call is weightedRange(R, root, bbox). 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
double weightedRange(Rectangle R, KDNode p, Rectangle cell) {
    if (p == null) return 0; // fell out of the tree?
    else if (R.isDisjointFrom(cell)) // no overlap with range?
        return 0;
    else if (R.contains(cell)) // the range contains our entire cell?
        return p.weight; // include the weight of p's subtree
    else { // the range stabs this cell
        int result = 0;
        if (R.contains(p.point)) // consider this point
            result += p.point.weight; // include p's point's weight
        // apply recursively to children
        result += rangeCount(R, p.left, cell.leftPart(p.cutDim, p.point))
        result += rangeCount(R, p.right, cell.rightPart(p.cutDim, p.point));
    }
    return count;
}
```

4
(b) 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.

**Solution 5:** The approach follows the standard nearest-neighbor search, but we add the additional condition that we do not visit nodes whose cell lies outside the disk’s radius, that is, if the distance between $q$ and the cell exceeds the disk radius $r$.

The helper is the same as for the standard nearest-neighbor search, but it is also given the disk radius $r$. Note that the best point may be null if no point has been found within the query disk. Otherwise, it contains the closest point seen so far that is within the query disk.

The initial helper call is `frnn(q, r, root, bbox, null)`, where `root` is the root of the tree, `bbox` is the bounding cell for the entire tree, and `null` is the initial best point. We define the *viability region* to be the disk centered at $q$ whose radius is the smaller of $r$ and the best point seen so far. If we fall out of the tree or our cell is outside the viable region, we return `best`. Otherwise, we check the point in this node, and update `best` appropriately. Finally, we recurse on the children, favoring the child that is closer to $q$.

```java
Point frnn(Point q, double r, KDNode p, Rectangle cell, Point best) {
    double bestDist = (best == null ? INFINITY : distance(q, best))
    double viableDist = min(r, bestDist) // distance to be viable
    if (p == null || distance(q, cell) >= viableDist) // not viable
        return best
    if (dist(q, p.point) < viableDist) // p.point is better?
        best = p.point // it's the new best
    Rectangle leftCell = cell.leftPart(cd, p.point) // child cells
    Rectangle rightCell = cell.rightPart(cd, p.point)
    if (q[cd] < p.point[cd]) { // q is closer to left
        best = frnn(q, r, p.left, leftCell, best) // try left then right
    } else { // q is closer to right
        best = frnn(q, r, p.right, rightCell, best) // try right then left
    }
    return best
}
```

**Solution 6:** To expose a node, we first apply a standard descent to find the exposed node, and we set the priority to $-\infty$ (or more practically, something like `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) {
    p.left = expose(x, p.left) // x is smaller - search 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
}

Figure 1: Expose operation in a treap.

Solution 7:

(a) Remember that the priorities in a treap are heap ordered, so that a parent has a lower priority than its children. Any node p where \(x_0 \leq p.key \leq x_1\) is a candidate answer, and by heap ordering, the answer is the highest such node in the treap, that is, the first such node we encounter.

We present below pseudo-code for the helper. The initial call is \(\text{return minPriority}(x_0, x_1, \text{root})\). If p.key > \(x_1\), we recurse on the left subtree, and if p.key < \(x_0\), we recurse on the right subtree. Otherwise, we are in the interval and return the current key.

```java
int minPriority(Key x0, Key x1, TreapNode p) {
    if (p == null) return Integer.MAX_VALUE // fell out of the tree?
    else if (p.key > x1) // range to left of p.key
        return minPriority(x0, x1, p.left) // ...search left
    else if (p.key < x0) // 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 \(O(\log n)\) in expectation.

Solution 8: See Fig. 2. The substring identifiers are shown (in suffix order) in the upper left. They are sorted lexicographically in the lower left. The final suffix tree is shown on the right.
Solution 9: We maintain two pointers \( p \) (source) and \( q \) (destination). When we encounter an allocated block (\( p.inUse \)) we copy this block’s contents to the destination. We set the \( prevInUse \) to 1, since we assume there will be no gaps after compression. We then increment the pointers to the source and destination by the block size. When we are done, we have one huge block leftover free block at the end. We set its \( inUse \) to 0, its \( prevInUse \) to 1, set its block sizes, and we return a pointer to the head of this block.

```c
(void*) compact(void* start, void* end) { // compact memory from start to end-1
    void* p = start; // p points to source block
    void* q = start; // q points to destination block
    while (p < end) {
        if (p.inUse) { // allocated block?
            memcpy(q, p, p.size); // copy to destination
            q.prevInUse = 1; // previous block is in-use
            q += p.size; // increment destination pointer
            // (no need to set q.size or q.inUse, since they are copied from p)
        }
        p += p.size; // advance to the next block
    }
    // everything copied - now q points to the remaining available block
    q.inUse = 0; // this block is available
    q.prevInUse = 1; // previous block is in-use
    int blockSize = p - q; // size of this final block
    q.size = blockSize; // set q.size
    *(q + q.size -1) = blockSize; // ... and q.size2
    return q; // return pointer to this block
}
```
Solution 10:

(a) Worst-case: $n + 1$. In the worst case, the user performs $n$ pushes and erases them all. In this case the pop operation skips over all $n$ of the erased elements and returns $null$, for a running time of $n + 1$.

(b) Amortized: 1.5. Before giving the formal proof, here is an intuitive argument. The expensive operations are skips of erased elements performed during a pop operation. In order to skip an erased node, it must first be pushed and then erased. If we charge an additional $\frac{1}{2}$ token for each push and erase, we have enough tokens accumulated to pay for each skip of an erased elements.

We will employ a standard token-based analysis. We charge 1.5 tokens for each operation. Each push and erasure takes 1 unit of actual time, and this means that we place half a token in the bank for each. Whenever a pop comes along, we skip over some number of elements. In order to skip over an element, it must have been pushed (depositing half a token) and it must have been erased (depositing half a token), and together, the $\frac{1}{2} + \frac{1}{2} = 1$ token pays for the time needed to skip this one element. We also use one token for the pop of the final unerased item.

Is 1.5 tight? Yes. This can be seen if you push $n$ entries (for a huge value $n$), erase them all, and do a single pop. The total number of operations is $m = n + n + 1 = 2n + 1$. The total work is $n + n + (n + 1) = 3n + 1$. Averaging over the $m$ operations, the amortized cost is $(3n + 1)/m = (3n + 1)/(2n + 1)$. If $n$ is large, this is $\approx 1.5$.

(c) Expected: $O(m/(m - k))$. The probability that any element was erased is $k/m$. Therefore, the probability that any accessed element is not erased is $p = 1 - k/m = (m - k)/m$. Basic probability theory teaches us that if a coin has probability $p$ of coming up heads, then in expectation, you will need to flip the coin $1/p$ times before seeing heads. In our case, this means that we expect to visit $1/p = m/(m - k)$ entries before finding an unerased entry.

Solution 11:

(a) We will show that the amortized time $\alpha = 7/3 = 2.333\ldots$ Each time we perform an insertion, we receive $\alpha$ tokens. One of these tokens will be used to pay for the insertion, and the remaining $\alpha - 1$ are put in a bank account to pay for the next expansion. Let us assume that we have just expanded a table of size $m$ resulting in a new table of size $m' = 4m$, which contains $3m/4$ entries. In order to induce the next expansion, the total number of entries must grow to $(3/4)m' = (3/4)(4m) = 3m$. This means that the number of new insertions is at least $3m - (3m/4) = (9/4)m$. Through these insertions we have collected $(9/4)m(\alpha - 1)$ tokens. We need to have enough tokens to pay the expansion cost, which is $3m$. Therefore, $\alpha$ must satisfy:

$$\frac{9m}{4}(\alpha - 1) \geq 3m \implies \alpha \geq 1 + \frac{4}{3} = \frac{7}{3},$$

as desired.

Aside: We can generalize this. Let $0 < \lambda < 1$ denote the load factor when the expansion is triggered, and let $\beta > 1$ denote the expansion factor. Let us assume that we have just expanded a table of size $m$ resulting in a new table of size $m' = \beta m$, which contains $\lambda m$
entries. In order to induce the next expansion, the total number of entries must grow to $\lambda m' = \lambda(\beta m)$. This means that the number of insertions is at least $\lambda \beta m - \lambda m = \lambda(\beta - 1)m$. Through these insertions we have collected $\lambda(\beta - 1)m(\alpha - 1)$ tokens. We need to have enough tokens to pay the expansion cost, which is $\lambda \beta m$. Therefore, $\alpha$ must satisfy:

$$\lambda(\beta - 1)m(\alpha - 1) \geq \lambda \beta m \quad \implies \quad \alpha \geq 1 + \frac{\beta}{\beta - 1} = \frac{2\beta - 1}{\beta - 1}.$$ 

(It is interesting that the amortized cost does not depend on $\lambda$. When $\beta = 4$, this yields $\alpha = 7/3$, as expected.)

(b) To decrease the amortized cost, we should increase the expansion factor, since this reduces the frequency with which expansions take place (but does not increase their cost). This increase has the negative side effect that we may waste more space if we never fill up the expanded table. For example, if we expanded the table by a factor of 400 instead of 4, expansions would be very infrequent, but the final expansion could potentially waste a lot of space.