Solution 1:

(a) (3) Saves space: The multilist only stores non-zero entries, so it saves space for sparse matrices. Operations generally take slightly longer to perform.

(b) (2) and (5): Replacements are computed when a node has two non-null children. (Replacements are never computed 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 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!

(c) Skew: The skew operation enforces the right-child constraint. (Split is used to enforce the condition that if a node is red, then both its children are black.)

(d) (1): Exactly $2^h$. Each time a link is performed, two trees of height $k$ are merged to form a tree of height $k + 1$. By induction, the number of leaves in a tree of height $h$ built in this manner is exactly $2^h$. The number can be smaller if cuts are performed within this subtree (but the problem stipulated that this does not happen). It can never be bigger.

(e) Key rotation (adoption) is preferred because it does not require the allocation of new nodes, and once performed, we are done—it cannot propagate to higher levels of the tree. It is not needed when doing insertions, and hence this option is often omitted just to keep the code simpler.

(f) 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.

(g) The reason for storing the size field twice is for the purpose of merging blocks together. When a used block becomes free, it accesses the block immediately before and after, and will merge if they are free. If a merge is possible, we need to know the size of each block. The size field tells us the size of the block before us and the size field tells us the size of the block after us.

(h) (2) and (3): The spatial index stores all the points that are not in the current spanning tree (so there is exactly one entry for each point not in the current spanning tree). For each point in the current spanning tree, the priority queue stores its closest point outside the spanning tree. There may be redundant entries, however, so the number may be larger than than the number of points in the current spanning tree.
Solution 2: The answer is presented in Fig. 1. The first rotation is a zig-zag from 5’s grandparent 4. The next is a zig-zig from 5’s new grandparent 10, and the final is a single zig about 5’s new parent 2.

Figure 1: Splaying.

Solution 3: The answers for both parts are presented in Fig. 2. In part (a), the insertion of "Z" loops infinitely, and so the insertion fails. In part (b), the deletion of "B" puts a special “placeholder” in "B”’s position so that when "Z" is inserted, it can be placed in this location. (The placeholder is needed, because if we simply erased "B", then an attempt find "X" would fail when we see that table[h("X")]=table[4] is empty.)

Figure 2: Hashing.

Solution 4: The answers to (a) and (b) are shown in Fig. 3. We first list the substring identifiers by their suffix index, and we then list them in lexicographical order. To answer the query of how many occurrences of of the substring "ba" occur in S, we follow the link labeled "ba" from the root. The node we arrive at is labeled with 3, which indicates that there are three instances of this subtree (starting at indices 2, 0, and 5 in particular, as indicated by the leaves descended from this node).

Solution 5:
(a) $q_y^- \leq y_i \leq q_y^+$: To intersect the query segment defined by $x$, $y^-$, and $y^+$, the $y$-coordinate of segment $s_i t_i(y_i)$ must lie between the upper and lower endpoints of the vertical segment.
Figure 3: Substring identifiers and the associated suffix tree.

(b) \( x_i^- \leq q_x \): To intersect the query segment the left endpoint \( (x_i^-) \) must lie to the left of the vertical segment.

(c) \( x_i^+ \geq q_x \): To intersect the query segment the right endpoint \( (x_i^+) \) must lie to the right of the vertical segment.

(d) We will build a 3-layer range-tree structure, to test each of these conditions. (The order does not matter, since we are just counting. Note that we cannot just build a single layer for the \( x \)-conditions. This is because there are two independent \( x \) values, \( x_i^- \) and \( x_i^+ \) and each needs to be filtered independently against \( q_x \).)

Let us treat each horizontal line segment as if it is a point \( (y_i, x_i^-, x_i^+) \) in \( \mathbb{R}^3 \). We can model the query segment as a 3-dimensional range, given by the three constraints

\[
 y_i \in [q_y^- , q_y^+], \quad x_i^- \in [-\infty, q_x], \quad \text{and} \quad x_i^+ \in [q_x , +\infty].
\]

This is just a standard 3-dimensional range tree (where two of the intervals are semi-infinite).

(e) Queries are answered as they would be for any standard 3-dimensional range tree:

1. We search the first layer (sorted by \( y_i \)) to identify a set of \( O(\log n) \) canonical nodes such that the points in these subtrees have \( y_i \) coordinates that define a disjoint cover of \( [q_y^- , q_y^+] \).
2. For each node from the first layer, we access the 2nd layer auxiliary trees to identify a set of \( O(\log n) \) canonical nodes such that the \( x_i^- \) coordinates of these points form a disjoint cover of \( [-\infty, q_x] \).
3. Finally, for each of the nodes from the 2nd-layer search, we access the 3rd-layer auxiliary trees to identify a set of \( O(\log n) \) canonical nodes such that the \( x_i^+ \) coordinates of these points form a disjoint cover of \( [q_x , +\infty] \). We sum the sizes of all these subtrees and return the result as the final answer.

(f) This is just a standard 3-layer range tree for \( n \) points, which has space \( O(n \log^2 n) \). The query time is \( O(\log^3 n) \).
Solution 6:

(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 $\lambda m - \lambda m = \lambda (\beta - 1)m$. 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 $\lambda m$. Therefore, $\alpha$ must satisfy:

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

(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.

Solution 7: 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)
    }
}
```