Solution 1:

(a) We first insert 25 in the tree as the right child of 23 (see Fig. 1). We then back out of the recursive calls, updating heights and recomputing balance factors as we go. When we reach 20, we find that it has a balance factor of +2. We see that it is right-right heavy, so we perform a single left rotation at 20, which brings the 23 up and makes 20 its left child. We update balance factors. No further updates are needed to the tree.

(b) We first insert 24 as a (red) left child of 25 at level 1 of the tree (see Fig. 2). On returning to 25, it performs a skew which brings up 24, which becomes the new current node. We then apply split at 24, which performs a split and brings 25 up to level 2, where it becomes the right (red) child of 22. We continue up the tree, but no further modifications are performed.

Solution 2:

(a) \( n + 1 \): The threads replace the null child pointers in the binary tree. In class we showed that any binary tree with \( n \) nodes has \( n + 1 \) null child pointers.

(b) \( \log(k + 1) \): Since the data structure starts with \( n \) individual singleton set, after \( k \) operations the largest any tree can be is \( k + 1 \) nodes (assuming that all \( k \) operations go to build the same tree). In class we showed that any tree in the union-find structure that has \( m \) nodes has height at most \( \log m \).
(c) $O(\log n)$: We showed that the NPL value of the root of any leftist tree is $O(\log n)$. Any tree can be converted into a leftist tree by swapping the left and right children of nodes that violate the leftist property. Swapping child nodes does not alter the NPL values, so this bound holds for all binary tree, leftist or not.

(d) $O(n)$: Even the rightmost path of a leftist heap is of $O(\log n)$ length, this restriction does not apply to any other path. A path can be as long as $O(n)$, and if you perform a sift-up that goes from the leaf to root level along such a path, the running time is $O(n)$.

(e) True: A postorder traversal does not visit a node until after both its subtrees are visited, so the first node visited must have no children and hence is a leaf.

(f) Let’s assume keys are sorted. Advantage of arrays: Faster search using binary search, less storage space (no need for pointers), better cache behavior for modern memory systems. Disadvantage: Need to reallocate periodically when space runs out. Also, insertion/deletion require shifting elements around.

(g) Min: 0, Max: 13: You can visit one per level, but there need not be any red nodes in the tree at all. A few people attempted to answer a much more challenging question. What is the maximum total number of red nodes in the tree? At each level of the tree, each black-red cluster gives rise to 3 more black-red clusters at the next level, so the number grows roughly as $\sum_{i=0}^{12} 3^i = (3^{13} - 1)/2$. This was a harder problem, so I gave credit if you got this answer.

(h) $O(n)$: The total time for $n$ operations is $T(n) = \sum_{i=1}^{n} i = O(n^2)$ and so the amortized time is $T(n)/n = O(n)$.

Solution 3:

(a) We can compute the leftmost leaf recursively as follows. If $p$ has a child, then return the leftmost leaf of its first child. Otherwise, return $p$ itself.

```java
Node leftLeaf(Node p) { // get leftmost leaf in p's subtree
    if (p.firstChild == null) { // p is a leaf?
        return p // it is its own left leaf
    } else {
        return leftLeaf(p.firstChild) // get the left leaf of the left child
    }
}
```

(b) To compute the rightmost leaf recursively, we will use a helper function. We invoke the helper on our first child. The helper function will first move to the rightmost sibling by following next-sibling links until it can go no further. Then it drops down the first child and repeats the process. When we can go neither right nor down, we are at the rightmost leaf.

```java
Node rightLeaf(Node p) { // get rightmost leaf in p's subtree
    if (p.firstChild == null) return p
    else return rlHelper(p.firstChild)
}
```
Node rlHelper(Node p) {
    if (p.nextSibling != null) { // can go right?
        return rlHelper(p.nextSibling) // go to the right
    } else if (p.firstChild != null) { // can go down?
        return rlHelper(p.firstChild) // go down
    } else { // no further?
        return p // this is it
    }
}

Solution 4: Throughout, let $k$ denote the number of rows and columns in the matrix.

(a) $n = k(k + 1)/2$: The total number of elements $n$ is obtained from adding up the number of elements in each column, which is $\sum_{i=1}^{k} i$, which by the standard formula is $k(k + 1)/2$. Note that for large $n$ and $k$, we have $n \approx k^2/2$ or equivalently $k \approx \sqrt{2n}$.

(b) The final column $\langle 5, 9, \ldots, 15 \rangle$ forms a right chain of nodes. Following this, each successive column adds a new left child to each of the nodes from the previous column (see Fig. 3).

(c) $h = k - 1$: All the leaves of the tree are at the same level, which is the side length of the matrix. Because heights start counting from zero, the tree’s height is $k - 1$. (This can be expressed as a function of $n$, but it is quite messy. We get $h = (\sqrt{1 + 8n} - 3)/2 \approx \sqrt{2n}$.)

(d) $d(i) = i + 1$: The number of nodes per level increases linearly from 1. Since depths start with zero, we have $d(i) = i + 1$.

(e) $T(n) = O(n^{3/2})$: The total insertion time $T(n)$ is the sum of the product $(i + 1)d(i)$ for $i$ ranging from zero up to the height of the tree. Thus, we have

$$T(n) = 1 \cdot 1 + 2 \cdot 2 + \ldots + k \cdot k = \sum_{i=1}^{k} i^2.$$  

By the standard formula for the quadratic summation, this is $(2k^3 + 3k^2 + k)/6 = O(k^3)$. Since asymptotically, $k = O(\sqrt{n})$, we have $T(n) = O(n^{3/2})$.

Solution 5:
(a) Amortized cost is $\approx k/2$ or equivalently $\sqrt{m}/2$: At the start of the run the array has $(k-1)^2$ elements. We overflow when we hit $k^2$ elements, so we can perform at least $k^2-(k-1)^2 = 2k-1$ operations until the next expansion. Let us charge the $\tau$ work tokens for each operation. We take one work token to pay for the actual operation, and we bank the remaining $\tau - 1$ tokens. When the next expansion occurs, we have saved at least $(2k-1)(\tau - 1)$ tokens. The expansion cost is the number of elements copied, which is $m = k^2$. To pay for the expansion, we need to set $\tau$ large enough so that 

$$(2k - 1)(\tau - 1) \geq k^2$$

or equivalently

$$\tau \geq 1 + \frac{k^2}{2k-1} \approx \frac{k}{2} = \frac{\sqrt{m}}{2},$$

assuming that $k$ is large.

(b) Amortized cost is $O(\sqrt{n})$: We break any sequence of operations of (large) length $n$ into runs between expansion events. We have shown in (a) that the amortized cost of any run is $O(\sqrt{m})$ where $m$ is the size of the current array. After $n$ operations, the largest the array can be is $n$ (assuming nothing but pushes), so the amortized cost cannot be larger than $O(\sqrt{n})$.

This is a good first-order estimate (and would be good enough for full credit). Here is a more detailed analysis. Since we’re looking for an asymptotic bound, there is no harm in rounding $n$ up to the next larger perfect square. Let $n = k^2$. This means that we have gone through $k$ runs to get here (with arrays of size $1, 4, 9, \ldots, k^2$). As argued in part (a), each run is dominated by the copying cost, which is the current size of the array. Thus, the cost of the $i$th run is roughly $i^2$. Applying the quadratic summation formula, we see that the total cost is therefore

$$\sum_{i=1}^{k} i^2 = \frac{2k^3 + 3k^2 + k}{6} \approx \frac{k^3}{3} \quad \text{for large } k.$$

Since $n = k^2$, the total cost is roughly $T(n) = n^{3/2}/3$. To get the amortized cost, we divide by the number of operations $n$, which yields an amortized cost of $T(n)/n = \sqrt{n}/3$, which is $O(\sqrt{n})$, exactly as given by our quick analysis.