Solution 1: (1.1): See Fig. 1(a). (1.2): See Fig. 1(b).

(1.3): Fig. 2 shows the insertion. First, the new key is inserted at its appropriate location in the tree, namely as the right child of 6. As we walk up the search path, the first violation of the AA conditions is at node 5, which has a right-right grandchild at the same level. This results in a call to \texttt{split}(5), which performs a left rotation at 5 and promotes its right child 6 up to the next higher level. Next, when we get to node 8, it sees that its left child 6 is at the same level. This results in a call to \texttt{skew}(8), which performs a right rotation. After this, the AA conditions are all satisfied and we are done.

Solution 2:

(2.1) True: Generally, given an inorder threaded tree, it is possible to travel from any node to its inorder predecessor or successor. By repeating this, we can reach any node from any other.

(2.2) (c): If \texttt{u.right} is a thread, then it points to \texttt{u}'s inorder predecessor. If a node in a binary tree has no right child (as must be true here), then it inorder successor must be a proper ancestor. Therefore, its depth is strictly smaller than \texttt{u}'s. (It was pointed out that this is technically incorrect if \texttt{u} is the rightmost leaf of the tree, since \texttt{u.right == null}. If we make the convention that \texttt{depth(null) = -1}, then this case works as well.)
(2.3) Only (c): In order to have a thread in a full binary tree, the node \( u \) must be a leaf. In a full binary tree, leaves and internal nodes alternate according to an inorder traversal. Therefore, \( u.\text{right} \) must be an internal node.

(2.4) The node storing the smallest key \( x_1 \) is guaranteed to be black. This is due to the AA-tree constraint that each red node is the right child of its parent, and hence its key must be larger than its parent.

(2.5) (a): Tracing through the code reveals that no changes will be made to the tree. This is due to the fact that \( \text{nil}.\text{left} = \text{nil}.\text{right} = \text{nil} \), and all pointer assignments just replace one pointer to \( \text{nil} \) with another pointer to \( \text{nil} \). So, the initial test that \( p == \text{nil} \) is not needed.

(2.6) (a) A node's priority is set when the node is created, and it is never changed after that.

(2.7) I'll accept two answers: \( n/8 \) and \( n/16 \), since the term “contributes to level \( i \)” is somewhat ambiguous. Recall that in order to reach level \( i \), a node must throw \( i \) consecutive heads, which occurs with probability \( 1/2^i \). Therefore, there are \( n/2^i \) such nodes in expectation, which yields \( n/8 \) for \( i = 3 \). (This was the answer I was expecting.) Among the nodes that reach level \( i \), there is a probability of \( 1/2 \) that they will rise to the next higher level. Thus, the number of nodes whose maximum level is level \( i \) is \( n/2^{i+1} \). This yields \( n/16 \) in the case when \( i = 3 \).

**Solution 3:**

(3.1) The algorithm performs an inorder traversal of the tree. If it falls out of the tree or arrives at a node of height less than \( h \), it returns. Otherwise, it invokes itself on the left subtree, then processes the current node by checking if the height matches \( h \), and then invokes itself on the right subtree. We present the recursive helper below, which takes as arguments the target height \( h \), the current node \( p \), and the current list \( L \). The initial call is \( \text{listAtHeight}(h, \text{root}, L) \), where \( L \) is an empty list.

```java
void listAtHeight(int h, AVLNode p, List L) {
    if (p == null || p.height < h) return
    else
        listAtHeight(h, p.left, L)
        if (p.height == h) add p.key to L
        listAtHeight(h, p.right, L)
}
```

We assert that the running time is proportional to the number of nodes at height \( h \) or higher. Each recursive call takes constant time, and so the running time is proportional to the number of nodes on which the function is called. Every call is made to a node of height \( h - 1 \) or larger, so the running time is clearly proportional to the number of nodes of height \( h - 1 \) or more. A node of level \( h - 1 \) is the child of a node of height at least \( h \), so this number is proportional to the number of nodes of height \( h \) or higher, as desired.

(3.2) The algorithm performs an inorder traversal of the tree, keeping track of the depth of the nodes visited. If it falls out of the tree or arrives at a node of depth greater than \( d \), it returns. Otherwise, it invokes itself on the left subtree (incrementing the current depth by one), then
processes the current node by checking if the depth matches \( d \), and then invokes itself on the right subtree. We present the recursive helper below, which takes as arguments the current depth of the node, the target depth \( d \), the current node \( p \), and the current list \( L \). The initial call is \texttt{listAtDepth(0, d, root, L)} where \( L \) is an empty list.

```java
void listAtDepth(int currDepth, int d, AVLNode p, List L) {
    if (p == null || currDepth > d) return
    else
        listAtDepth(currDepth + 1, d, p.left, L)
    if (currDepth == d) add p.key to L
    listAtDepth(currDepth + 1, d, p.right, L)
}
```

We assert that the running time is proportional to the number nodes of depth \( d \) or lower. The argument is similar to that of (3.1). As in (3.1), we actually invoke the algorithm on nodes of depth \( d + 1 \), but their number is within a constant factor of the number of nodes of depth \( d \) or lower.

(3.3) This is easy to prove by induction. There is at most one node at depth 0, the root. Assume inductively that there are \( 2^{d-1} \) nodes at depth \( d - 1 \). Each node at level \( d - 1 \) gives rise to at most two nodes at level \( d \), so the number of nodes at level \( d \) is at most \( 2 \cdot 2^{d-1} = 2^d \).

(3.4) This is proved by induction. For the basis cases, observe that an AVL tree of height 0 or 1 is full at level 0 only. Otherwise, an AVL tree of height \( h \) is formed from two AVL trees, one of height exactly \( h - 1 \) and the other of height either \( h - 1 \) or \( h - 2 \). By the induction hypothesis, both these subtrees are all full up to depth at least \( \lfloor (h-2)/2 \rfloor \). Therefore, the original tree is full at depth

\[
\left\lfloor \frac{h-2}{2} \right\rfloor + 1 = \left\lfloor \frac{h}{2} - 1 \right\rfloor + 1 = \left\lfloor \frac{h}{2} \right\rfloor - 1 + 1 = \left\lfloor \frac{h}{2} \right\rfloor,
\]

as desired.

\textbf{Solution 4: } It will simplify the proof to assume that \( k \) and \( n \) are very large, so we can ignore terms like \( \pm 1 \) and floors and ceilings. Let \( c \) denote the amortized cost, and we will determine the best value for \( c \).

(4.1) (Expansion case) Amortized cost = 5: Let \( k \) denote the size of the array at the start of the run. Irrespective of whether we got here via expansion or contraction, there are very nearly \( k/2 \) elements stored in this array. In order to overflow this array as fast as possible, we need to perform at least \( k - k/2 = k/2 \) operations (all pushes). We collect \( c(k/2) \) tokens over the run. One of these tokens goes to the standard push cost, and so we have at least \( c - 1 \) tokens in our bank account by the end of the run. The reallocation produces an expanded array of size \( 2k \), which means that the actual reallocation cost is \( 2k \). We wish to select \( c \) so that we have enough in our bank account to pay for the operations, that is \( (c - 1)k/2 \geq 2k \).

The smallest value satisfying this is \( c = 5 \).

(4.2) (Contraction case) Amortized cost = 3: Let \( k \) denote the size of the array at the start of the run. Observe that at the end of each run, the number of elements in the array is roughly
half the array size, that is, there are $k/2$ elements in the array at the start of the run. In order to underflow this array as fast as possible, we need to perform at least $k/2 - k/4 = k/4$ operations (all pops). We collect $c(k/4)$ tokens over the run. One of these tokens goes to the standard pop cost, and so we have at least $(c - 1)k/4$ tokens in our bank account by the end of the run. The reallocation produces a contracted array of size $k/2$, which means that the actual reallocation cost is $k/2$. We wish to select $c$ so that we have enough in our bank account to pay for the operations, that is $(c - 1)k/4 \geq k/2$. The smallest value satisfying this is $c = 3$. (A common error is to assume that there are $k$ entries in the array at the start of the run. If you work though the rest of the analysis, this yields a final amortized cost of $5/3$.)

From (4.1) we have $c \geq 5$ and from (4.2) we $c \geq 3$. Therefore, setting $c = \max(5, 3) = 5$ satisfies both requirements.

(4.3) Contracting when $n \leq k/2$ does not yield a constant amortized cost. The reason is that we can generate a very expensive oscillation. For concreteness, let’s imagine that the array is full, say, $n = k = 100$. The next push overflows and it costs us 200 units to allocate the new array of size $k = 200$. Now, if we immediately pop this array, we have $n = 100$ and $k = 200$, triggering an immediate contraction, at a cost of 100. So, just two operations cost us 300 work units, and we are back where we started. (There is nothing magical about $k/4$. To get a constant amortized cost, the contraction should be triggered when $n$ is any constant fraction of $k$ that is strictly smaller than $1/2$. The amortized cost gets better as the fraction gets smaller, but then so does the amount of space that is wasted.)