Solution 0:

(a) Inserting 6 involves first inserting the key into the appropriate node, which now contains $(5 : 6 : 7)$ and is overfull. It splits, resulting in two child nodes 5 and 7, and the 6 is promoted to the parent. The parent now contains $(6 : 8 : 12)$ and is overfull. So we split this node, resulting in two child nodes 6 and 12, and the 8 is promoted to the parent. Since the parent 4 can absorb the additional key/child, becoming $(4 : 8)$, we are done.

(b) Deleting 20 causes its leaf node to become underfull. Since its only sibling 17 cannot give up a key, we perform a merge, which demotes the key 18 from the parent, resulting in a new leaf node containing $(17 : 18)$. The parent of this node is underfull. Its only sibling 24 cannot give up a key, so we again perform a merge, we demotes the key 21 from the parent, resulting in a node containing $(21 : 24)$. The new parent containing 26 is fine, so we are done.

![Figure 1: 2-3 tree insertion.](image)

![Figure 2: 2-3 tree deletion.](image)

Solution 1:

(a) In class we showed that an extended binary tree with $m$ internal nodes has $m + 1$ external nodes. Every full tree can be viewed as an extended binary tree, where leaves are external nodes. Thus, a full tree with $n = m + (m + 1) = 2m + 1$ total nodes has $m + 1 = (n + 1)/2$ leaves. Observe that $n$ is always odd, so this can also be written as $\lceil n/2 \rceil$.

(b) **True**: External and internal nodes alternate in an inorder traversal. This can be proved by induction. Observe that in the inorder traversal of any extended binary tree, the first and
last nodes visited must be external. So, by induction, the nodes of the left subtree alternate
(ending in an external node), then the root is visited (internal), and then the nodes of the
right subtree alternate (starting with an external node).

(c) **True**: There is always an external node at depth at most \( d = \lceil \lg n \rceil \). If this were not true,
then the first \( d \) levels would all be internal. It follows that the number of external nodes must
be at least \( 2^{d+1} > 2\lg n = n \), contradicting the hypothesis that there are \( n \) external nodes.

(d) Given a 2-3 tree with \( \ell \) levels, there are at least \( n_{\text{min}}(\ell) = \sum_{i=0}^{\ell-1} 2^i \) nodes and at most
\( n_{\text{max}}(\ell) = \sum_{i=0}^{\ell-1} 3^i \) nodes. By the formula for the geometric series, we have \( n_{\text{min}}(\ell) = 2^\ell - 1 \)
and \( n_{\text{max}}(\ell) = (3^\ell - 1)/2 \). Solving for \( \ell \) in each case, we have \( \ell = \log_2(n_{\text{min}}(\ell) + 1) \) and
\( \ell = \log_3(2n_{\text{max}}(\ell) + 1) \). Thus, the number of levels \( \ell \) is:

\[
\log_3(2n + 1) \leq \ell \leq \log_2(n + 1).
\]

(e) It was observed in class that in the insertion process, an AVL tree may perform either a single
rotation or a double-rotation. After this, the subtree height is the same as in the original
tree, so no further rotations are needed. Thus, the number of rotations following an insertion
is **at most two**.

(f) It was observed in class that deletions from the AVL may propagate up to the root. Thus,
the number of rotations can be proportional to the height, or \( O(\log n) \).

(g) Min: 0, Max: \( h + 1 \). A 2-3 tree of height \( h \) yields an equivalent AA tree with \( h + 1 \) levels.
Each node at a given level may give rise to a single red node (if the path goes through a
3-node) or not (if the path goes through a 2-node).

(h) With standard binary search trees, the expectation was over all \( n! \) insertion orders. With
treaps, the expectation was over all \( n! \) orders of the priority values. The latter is preferred,
because the data structure’s expected performance is not under the influence of the access
distribution.

(i) (i) \( O(\log n) \): Because all the keys were randomly permuted, the expected number of keys
from the second half of the list that can lie between any two consecutive elements from the
first half of the list is a constant (in fact, at most 2). The expected height of the tree after
inserting the first \( n/2 \) keys is \( O(\log n) \), and after this, there are at most a constant number
of insertions that are expected to fall out through any null pointer of this tree.

(j) False: While the root will be balanced, other nodes of the tree (e.g., the root’s right and left
children) may not be.

(k) (v) No negative effects: If just two keys have the same priority, their parent/child relationship
might be affected, but the rest of the treap’s structure will be fine.

**Solution 2:**

(a) Since \( n \) is of the form \( 2^k - 1 \), it follows that in a complete binary tree each subtree of the root
has exactly \( (n - 1)/2 \) nodes. If we start with a left chain and do \( (n - 1)/2 \) right rotations,
then we have a tree in which the median is now at the root, the left subtree is a left chain and the right subtree is a right chain. We can rebalance each of these subtrees recursively (but reversing left and right on the right subtree).

To keep track of whether we are fixing a left chain or right chain, we pass in a parameter `direc` which is either `LEFT` or `RIGHT`. The initial call is `balance(root, n, LEFT)`.

![Figure 3: Rotating a tree into balanced form.](image)

```java
balance(BinaryNode p, int n, Direction direc) {
    if (n <= 1) return // one node?---done
    if (direction == LEFT) // subtree is left chain
        for (i = 0; i < n/2; i++) p = rotateRight(p)
    else // subtree is right chain
        for (i = 0; i < n/2; i++) p = rotateLeft(p)
    balance(p.left, n/2, LEFT) // rebalance left subtree
    balance(p.right, n/2, RIGHT) // rebalance right subtree
}
```

(b) Let $R(n)$ denote the number of rotations needed to rotate an $n$-node tree into balanced form. After performing $n/2$ rotations, we then invoke the function on two subtrees, each with roughly $n/2$ nodes. The total number of rotations satisfies the following recurrence:

$$R(n) = \begin{cases} 
1 & \text{if } n = 1 \\
2R(n/2) + (n/2) & \text{otherwise.}
\end{cases}$$

This is essentially the same recurrence that arises with sorting algorithms like MergeSort. By applying any standard method for solving recurrences (e.g., the Master Theorem or expansion) it follows that the total number of rotations is $O(n \log n)$. (Note by the way that it is possible to modify this proof to show that it is possible to convert any $n$-node binary tree into any other with $O(n \log n)$ rotations.)

Solution 3:

(a) The insertion code is similar to that of a standard binary search tree, but since we need access to the node’s parent, we have two arguments, the current node `p`, and its parent `par`.
To insert a node we begin with the usual descent used by the standard insertion algorithm. When we fall out of the tree, there are two cases. If we fall out on a left child link, then the newly created node’s inorder predecessor is its parent’s inorder predecessor \((\text{par.left})\) and its inorder successor is its parent \((\text{par})\). (See Fig. 4(a).) If we fall out on a right child link, then the newly created node’s inorder successor is its parent’s inorder successor \((\text{par.right})\) and its inorder predecessor is its parent \((\text{par})\).

Because of the use of threads, we cannot detect falling out of the tree using the standard test \(p == \text{null}\). Instead, we will add an additional parameter, \(\text{fellOut}\), which is set to true whenever we attempt to advance \(p\) through a thread. We assume that the \BinaryNode\ constructor is given four arguments: the key, the value, and the two threads. It sets both thread indicators to \text{true}.

```java
BinaryNode insert(Key x, Value v, BinaryNode p, BinaryNode par, boolean fellOut) {
    if (fellOut) { // fell out of tree
        if (par == null) // new node is the root
            p = new BinaryNode(x, v, null, null);
        else if (x < par.data) // new leaf on left
            p = new BinaryNode(x, v, par.left, par);
        else if (x > par.data) // new leaf on right
            p = new BinaryNode(x, v, par, par.right);
    }
    else if (x < p.data) { // insert in left subtree
        p.left = insert(x, p.left, p, p.leftIsThread);
        p.leftIsThread = false;
    }
    else if (x > p.data) { // insert in right subtree
        p.right = insert(x, p.right, p, p.rightIsThread);
        p.rightIsThread = false;
    }
    else throw DuplicateKeyException;
    return p;
}
```

Figure 4: Insertion into a threaded binary tree
(b) If \( p \) has a left child, then its preorder successor is this child. Otherwise, if it has a right child, then the preorder successor is this right child. If it has neither (that is, this node is a leaf), we follow right threads until reaching the first node whose right-child link is not a thread (see Fig. 4(b)). The right child of this node is the preorder successor. If this chain ends in a null pointer, then we return \texttt{null} (since there is no preorder successor). To start the process, the initial node is the root.

\[
\text{BinaryNode nextPreorder(BinaryNode } p) \{ \quad \text{// preorder successor of } p \\
\quad \text{if (!p.leftIsThread) } \quad \text{// has a left child?} \\
\quad \quad \text{return p.left; } \quad \text{// ...return this} \\
\quad \text{else } \{ \quad \text{// no left child} \\
\quad \quad \text{BinaryNode q = p; } \quad \text{// start here and} \\
\quad \quad \quad \text{do } \{ \quad \text{// ...follow right threads} \\
\quad \quad \quad \quad \text{boolean isThread = q.rightIsThread;} \\
\quad \quad \quad \quad \text{q = q.right; } \quad \text{// until null or child} \\
\quad \quad \quad \quad \text{while (q != null && isThread)} \quad \text{// return the result} \\
\quad \quad \quad \text{return q;} \\
\quad \text{\}} \\
\}\}
\]

\textbf{Solution 4:} There are a number of cases to consider. First, if \( p \) is the root, it has no predecessor. Otherwise, if \( p \) is a left child, then its preorder predecessor is its parent (see Fig. 5(a)). If \( p \) is a right child, there are two cases. If its parent has no left child, then its preorder predecessor is its parent (see Fig. 5(b)).

\[
p == p.parent.left \\
\quad \text{return } p.parent \\
\]

\[
p.parent.left == null \\
\quad \text{return } p.paren \\
\]

\[
p.parent.left != null \\
\quad \text{return last preorder node} \\
\]

Figure 5: Cases arising in computing the preorder predecessor.

Otherwise, \( p \)'s parent has a left child. Let \( q \) be this child (see Fig. 5(c)). The desired node is the \textit{last preorder node} in \( q \)'s subtree. Computing this correctly takes a bit of thought. The key observation is that such a node must be a leaf (since an internal node comes earlier in preorder than either of its children). If a node has a single right child, the last preorder node comes from this child. If it has just a left child, it will come from there. We will give a recursive function to implement this (see the function \texttt{preorderLast} in the code block below).

\[
\text{Node preorderPred(Node } p) \{ \quad \text{// } p \text{'s preorder predecessor} \\
\quad \text{if (p.parent == null) } \quad \text{// } p \text{ is the root?} \\
\quad \quad \text{return null; } \quad \text{// ...no predecessor} \\
\quad \text{else if (p == p.parent.left) } \quad \text{// } p \text{ is a left child?} \\
\quad \text{\}} \\
\]

5
return p.parent; // ...parent is predecessor
else { // p must be a right child
if (p.parent.left == null) // no left sibling?
    return p.parent; // ...parent is predecessor
else {
    return preorderLast(p.parent.left); // preorder last of parent’s left
}
}

Node preorderLast(Node q) { // preorder last in q’s subtree
    if (q.right != null) // right subtree is non-empty?
        return preorderLast(q.right); // ...look for it here
    else if (q.left != null) // left subtree is non-empty?
        return preorderLast(q.left); // ...look for it here
    else // arrived at a leaf
        return q; // ...this is it!
}

Solution 5:

(a) We go up to the parent and determine which of its children is p. We then respond with the next child, if this child exists. Clearly, this takes constant time.

Node23 rightSibling(Node23 p) {
    q = p.parent
    if (q == null) return null // root node has no sibling
    else {
        if (p == q.child[0]) // p is child #1?
            return q.child[1] // answer is child #2
        else if (q.nChildren >= 3 && p == q.child[1]) { // p is child #2?
            return q.child[2] // answer is child #3
        else
            return null // no child following p
    }
}

(b) We walk back towards the root, as long as we are the rightmost child of our parent. We then go to our right sibling and walk down along the leftmost child the same number of levels. We ascend the tree and then descend, so the running time is proportional to the tree’s height, which is O(\log n). There is an elegant recursive implementation of this idea. If a node has a right child, then its right child is its level successor. If not, its level successor is the leftmost child of the level successor of its parent. (By our assumption that all leaves are at the same level, if the parent’s level successor is non-null, its leftmost child exists.)

Node23 levelSuccessor(Node23 p) {
    if (p == null) return null;
    else if (rightSibling(p) != null) return rightSibling(p);
    else {
        q = levelSuccessor(p.parent)
        if (q == null) return null
else return q.child[0]
}
}

(c) There are at most \( n \) nodes on any level and each invocation of `levelSuccessor` takes \( O(\log n) \) time, so \( O(n \log n) \) is an obvious upper bound. However, it is not a tight bound. Suppose we consider the worst-case of starting at the leftmost leaf node. The various invocations of `levelSuccessor` visit every edge of the tree twice, once moving up the edge and once moving down. (Trace the code and you will see this easily.) Since a tree with \( n \) nodes has \( n - 1 \) edges, it follows that the running time is just \( O(n) \).

**Solution 6:**

(a) Min: \( m/3 \), Max: \( m/2 \): The minimum number of bits that are set to 1 is arises when we have a pattern of the form \( \ldots 001001001001 \ldots \), which implies we have roughly \( m/3 \) 1-bits. The maximum number of bits arises when we have the alternating pattern \( \ldots 01010101 \ldots \), which implies we have roughly \( m/2 \) 1-bits.

(b) \( 2m/3 \) ops: The new array is of size \( 3m \). The number of 1-bits copied over is at most \( m/2 \), so at most \( 2(m/2) = m \) entries have been rendered unavailable after these bits have been copied over. Thus, there are \( 3m - m = 2m \) remaining entries to be filled. Assuming we fill them in the most inefficient manner, it will take at least \( 2m/3 \) set operations to saturate these remaining elements.

(c) \( 3m \): The next reallocation event replaces the array of size \( 3m \) with an array of size \( 9m \), at a cost of \( 3m \).

(d) We use a token-based approach to prove that the amortized cost is \( t \). Let us assess a charge of \( t \) for each operation. As seen in 5.2, at least \( 2m/3 \) operations have occurred until the next reallocation, meaning that we have collected at least \( (2m/3)t \) tokens. We use one token to pay for each operation, meaning that we can bank the remaining \( (2m/3)(t - 1) \) tokens. The reallocation cost is \( 3m \), and so to pay for this, we must set \( t \) so that

\[
(2m/3)(t - 1) \geq 3m
\]

By simple manipulations, we see that \( t \geq 9/2 + 1 = 5.5 \) will do the job. So the amortized cost is 5.5.