Solutions to Midterm 1 Practice Problems

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 neighboring sibling 9 can actually accept another key, so an adoption/rotation would be possible here. However, the insert algorithm given in class only performs splits.) 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.

Solution 1:

(a) The number of leaves is exactly $\lceil n/2 \rceil$. 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 $m + 1 = \frac{n + 1}{2}$ leaves. Observe that $n$ is always odd, so this can also be written as $\lceil n/2 \rceil$. 

Figure 1: 2-3 tree insertion.

Figure 2: 2-3 tree deletion.
(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. (Notice that the problem asks whether *there exists* a node of depth at most \( \lceil \lg n \rceil \). This does not preclude the possibility of nodes whose depths are larger.)

(d) **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.

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

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

(f) Min: 0, Max: \( h + 1 \): If all the nodes of a 2-3 tree are 2-nodes, there are no red nodes in the corresponding AA tree, and hence the minimum is 0. To get the maximum, consider a 2-3 tree of height \( h \) in which all the nodes are 3-nodes. Along the far right chain there are \( h \) edges and hence \( h + 1 \) nodes. When encoded as an AA tree, each of these becomes a black-red pair, which implies that there are up to \( h + 1 \) red nodes along the rightmost chain.

(g) 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 dependent on the insertion order.

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

(i) (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.

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

(k) \( n/8 \): 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 \( i = 3 \). (Observe that half of these nodes, \( n/16 \), terminate at this level and the rest continue to higher levels.)

**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 \texttt{direc} which is either \texttt{LEFT} or \texttt{RIGHT}. The initial call is \texttt{balance(root, n, LEFT)}.

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

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

(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:** 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. 4(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. 4(b)).

\[
\begin{align*}
\text{p} &\rightarrow \text{p.parent} & \text{p.parent.left} &\rightarrow \text{p.parent} & \text{p.parent.left} &\rightarrow \text{null} \\
&\rightarrow \text{null} & & & & \\
(a) & & (b) & & (c) \\
\text{return last preorder node}
\end{align*}
\]

Figure 4: Cases arising in computing the preorder predecessor.

Otherwise, \( p \)'s parent has a left child. Let \( q \) be this child (see Fig. 4(c)). The desired node is the 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 \( \text{preorderLast} \) in the code block below).

\[
\begin{align*}
\text{Node preorderPred(Node p) \{ &// p's preorder predecessor} \\
&\text{if (p.parent == null) \} // p is the root?} \\
&\quad \text{return null; \} // \ldots no predecessor} \\
&\text{else if (p == p.parent.left) \} // p is a left child?} \\
&\quad \text{return p.parent; \} // \ldots parent is predecessor} \\
&\text{else \} // p must be a right child} \\
&\quad \text{if (p.parent.left == null) \} // no left sibling?} \\
&\quad \quad \text{return p.parent; \} // \ldots parent is predecessor} \\
&\quad \text{else \} \} // p.parent.left \neq null} \\
&\quad \quad \text{return preorderLast(p.parent.left); // preorder last of parent’s left} \\
&\}
\end{align*}
\]

\[
\begin{align*}
\text{Node preorderLast(Node q) \{ &// preorder last in q’s subtree} \\
&\text{if (q.right != null) \} // right subtree is non-empty?} \\
&\quad \text{return preorderLast(q.right); \} // \ldots look for it here} \\
&\text{else if (q.left != null) \} // left subtree is non-empty?} \\
&\quad \text{return preorderLast(q.left); \} // \ldots look for it here} \\
&\text{else \} // arrived at a leaf} \\
&\quad \text{return q; \} // \ldots this is it!} \\
&\}
\end{align*}
\]

To see that the running time is \( O(h) \), where \( h \) is the height of the tree, observe that each recursive call to \( \text{preorderLast} \) takes constant time and makes a single recursive call on one of its children. We can make \( h \) such calls before hitting a leaf.

Solution 4:

(a) 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.

(b) 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 \( \text{listAtDepth}(0, d, \text{root}, L) \), 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.

(c) 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 \( 2 \cdot 2^{d-1} = 2^d \).

(d) 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

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

**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.

```c
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.)

```c
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 \texttt{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 \texttt{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.