Solution to Midterm Exam 1

Solution 1:

(1.1) An extended binary tree with \( n \) internal nodes has \( n + 1 \) external nodes, and hence a total of \( 2n + 1 \) nodes of both types.

(1.2) The danger of always using the inorder successor is that this introduces a systematic bias into the structure of the tree. Over a long sequence of insertions and deletions, the tree’s height can grow to \( O(\sqrt{n}) \).

(1.3) A treap’s structure is uniquely determined by the keys and their priority values (assuming no duplicate priorities). Thus, the choice of the replacement key has no effect on the asymptotic performance of the treap.

(1.4) Each of the \( h + 1 \) levels of the tree may give rise to a red node (if the path goes through a 3-node) or not (if the path goes through a 2-node). Thus, the number of red nodes along any path ranges from a minimum of 0 up to a maximum of \( h + 1 \).

(1.5) The expected number of hops per level is \( 1/p \). The analysis is analogous to the previous one. With probability \( p \), we change levels and with probability \( 1 - p \), we remain at the same level. Thus, the number of hops at level \( i \) satisfies the recurrence \( E(i) = 1 + pE(i-1) + (1 - p)E(i) \). After some manipulations, this becomes \( E(i) = 1/p + E(i - 1) \). This implies \( 1/p \) hops are expected before jumping to the next level.

(1.6) True: If \( n = 1 \), there is only one node of depth 0 = \( \lceil \lg n \rceil \). Otherwise, one of the two subtrees has no more than \( n/2 \) external nodes. By induction, this subtree has an external node at depth at most \( \lceil \lg(n/2) \rceil = \lceil \lg n \rceil - 1 \), and therefore its depth in the full tree is at most \( \lceil \lg n \rceil \).

Solution 2:

(2.1) See Fig. 1 for all the intermediate steps. The rightmost is the final tree.

Figure 1: AVL-tree insertion.
(2.2) See Fig. 2 for all the intermediate steps. The rightmost is the final tree.

Solution 3: In the process of doing the rotation, in addition to \( p \) and \( q \), the following nodes are affected: \( s = p.\text{sibling} \), \( r = q.\text{right} \), and \( q.\text{left} \) and \( p.\text{right} \) (see Fig. 2). After doing the rotation itself, we make \( q \) and \( s \) siblings, we make \( p \) and \( q.\text{left} \) siblings, and we make \( r \) and \( p.\text{right} \) siblings. Except for \( p \) and \( q \), any of these could be null.

```java
BinaryNode rotateRight(BinaryNode p) { // right rotation at p
    BinaryNode q = p.left;
    s = p.sibling; // p's old sibling
    r = q.right; // q's old right child
    p.left = q.right; // do the rotation
    q.right = p;
    makeSiblings(s, q); // make s and q siblings
    makeSiblings(q.left, p); // make q.left and p siblings
    makeSiblings(r, p.right); // make r and p.right siblings
    return q;
}

void makeSiblings(BinaryNode p, BinaryNode q) {
    if (p != null) p.sibling = q;
    if (q != null) q.sibling = p;
}
```
**Solution 4:** 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 parent pointers up towards the root. When we arrive at the first ancestor \(q\) such that \(p\) is in \(q\)’s left subtree and \(q.right \neq null\), we return \(q.right\). If we never find such a node, then we return null.

```java
BinaryNode nextPreorder(BinaryNode p) { // p’s preorder successor
    if (p.left != null) // p has a left child?
        return p.left; // ...return it
    else if (p.right != null) // p has a right child?
        return p.right; // ...return it
    else { // p is a leaf
        BinaryNode q = p.parent; // follow parent chain
        while (q != null && (q.right == p || q.right == null)) {
            p = q;
            q = q.parent;
        }
        if (q != null) return q.right; // next non-ancestor on right
        else return null; // no preorder successor
    }
}
```

**Solution 5:**

(5.1) In a zig-zag tree, all rotations are zig-zag rotations. The result (with all intermediate trees) is shown in Fig. 4.

![Figure 4: Splaying the deepest node in a zig-zag tree.](image)

(5.2) In looking at the figure, it is evident that the nodes whose original level was odd (\(a, b, \text{etc.}\)) are mapped to depth \((k + 1)/2\), and nodes whose original level was even (\(i, h, \text{etc.}\)) are mapped to depth \(1 + k/2\). (We ignore the splayed node itself, which is mapped to depth 0.) In summary, we have

\[
\text{depth}(k) = \begin{cases} 
(k + 1)/2 & \text{if } k \text{ is odd} \\
1 + k/2 & \text{if } k \text{ is even.}
\end{cases}
\]

This provides some intuition for why splaying is good. The depth of every node along the search path decreases by roughly half.
(5.3) To prove the correctness of the above formula, observe that whenever three nodes are involved in a zig-zag rotation, the depth of the topmost node (which is an even-level node) increases by one, and the depth of its child (which is an odd-level node) remains unchanged. Rotations below these nodes do not affect their depth. By the nature of zig-zag rotations, whenever such a rotation takes place above these nodes, the depth of the subtree containing them decreases by one. Thus, if these nodes are at levels $k' = 2\ell$ and $k'' = 2\ell + 1$, there are $\ell$ such rotations that occur above them.

Therefore, the final depth of the upper (even level) node is its original depth $(2\ell)$, plus 1 (when it is rotated), and minus 1 for each subsequent zig-zag rotation above it $(-\ell)$:

$$(2\ell) + 1 - \ell = 1 + \frac{\ell}{2} = 1 + \frac{k'}{2} = \text{depth}(k').$$

Similarly, the final depth of the lower (odd level) node is

$$(2\ell + 1) + 0 - \ell = 1 + \ell = \frac{(2\ell + 1) + 1}{2} = \frac{k'' + 1}{2} = \text{depth}(k''),$$

as desired.