Solutions to the Practice Problems for Midterm 1

Disclaimer: These solutions have not been carefully checked. If anything seems to be fishy, please check with me.

Solution 1:

(a) The induction argument given in class for extended binary trees shows that a full binary tree with \( m \) internal nodes has \( m + 1 \) leaves. Thus, a 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) 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:

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

(c) 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.

(d) It was observed in class that deletions from the AVL may propagate up to the root. Thus, the number of rotations is \( O(\log n) \).

(e) In a skip list, nodes of variable sizes are allocated, because once allocated, the number of pointers in the node does not change. In contrast, in a B-tree, the number of keys stored in a node can vary as keys are inserted and deleted, and thus we always allocate nodes of the maximum possible size.

(f) Among almost all the search structures we studied, performance was measured in terms of worst-case performance. The exception was the splay tree. We showed that, over a sequence of operations, the time complexity is within a constant factor of the entropy of the access distribution, and this is known to be theoretically optimal.

(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 under the influence of the access distribution.

(h) A finger search means that, rather than starting the search at the root node, it starts from a given position in the search structure, for example, from the location of the most recently accessed node. Finger searches are important when a sequence of queries are being performed, and each query object is expected to be close to its predecessor in the sequence.
(i) We assert that node $y$ is at maximum depth 2 in the resulting splay tree. Consider the last splay rotation (zig, zig-zag, or zig-zig) just before $z$ was brought to the root. At this point $x$ is already at the root. Since $x$, $y$, and $z$ are consecutive in order, $y$ is either the right child of $x$ (zig-zig case) or $y$ is the left child of $z$ (in the zig or zig-zag cases). In the first case $y$ becomes the left child of $z$ (depth 1) and in the other case $x$ becomes the left child of $z$ and $y$ becomes the right child of $x$ (depth 2).

![Figure 1: Solution to Problem 1(i).](image)

**Solution 2:** We appeal to Fig. 2 as a justification of the following observations. (A proof by induction is straightforward in each case.)

![Figure 2: Solution to Problem 3.](image)

(a) A preorder traversal of $T'$ is equivalent to a preorder traversal of $T$: $\langle a, b, e, c, f, i, j, k, d, g, h \rangle$.
(b) An inorder traversal of $T'$ is equivalent to a postorder traversal of $T$: $\langle e, b, i, j, k, f, c, g, h, d, a \rangle$.
(c) A postorder traversal of $T'$ is not equivalent to any of the standard traversals of $T$. In the example shown above the order is $\langle e, k, j, i, f, h, g, d, c, b, a \rangle$.

**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 $\text{par}$. 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 (\texttt{par.left}) and its inorder successor is its parent (\texttt{par}). (See Fig. 3(a).) If we fall out on a right child link, then the newly created node’s inorder successor is its parent’s inorder successor (\texttt{par.right}) and its inorder predecessor is its parent (\texttt{par}).

We assume that the \texttt{BinaryNode} constructor is given four arguments: the key, the value, and the two threads. It sets both thread indicators to \texttt{true}.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{problem3a.png}
\caption{Solution to Problem 3(a).}
\end{figure}

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

(b) If \texttt{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. 3(b)). The right child of this node is the preorder successor. If this chain ends in a null pointer, then we return null (since there is no preorder successor). To start the process, the initial node is the root.

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

Solution 4:

(a) We first retrieve the number of keys \( m \) in the left subtree. If \( k \leq m \), we search the left subtree recursively. If \( k = m + 1 \), then the current node is the \( k \)-th smallest. Finally, if \( k > m + 1 \), we search the right subtree for the \((k - m - 1)\)-th smallest key (since we need to exclude the \( m + 1 \) keys from the current node and left subtree). The initial call is `findKth(root, k)`.

```java
Key findKth(BinaryNode p, int k) {
    int m = (p.left == null ? 0 : p.left.size);
    if (k <= m) return findKth(p.left, k);
    else if (k == m+1) return p.key;
    else return findKth(p.right, k-m-1);
}
```

(b) We first define two helper functions `countLessThan(p, x)` and `countLessEqual(p, x)`. The first counts of the number of keys of \( p \)'s subtree that are strictly less than \( x \), and the second counts the number that are less than or equal to \( x \).

The first function is computed as follows. If \( p \) is null, then the count is zero. Otherwise, if \( p.key \leq x \), then both \( p \) and its left subtree are all included in the count, and we recur on \( p \)'s right child. Otherwise, all the keys to be counted lie in \( p \)'s left subtree, and we recur on \( p.left \). The other function is similar.

```java
int size(BinaryNode p) { return (p == null ? 0 : p.size) }

int countLessThan(BinaryNode p, Key x) {
    if (p == null) return 0
    else if (p.data < x) return 1 + size(p.left) + countLessThan(p.right, x)
    else return countLessThan(p.left, x)
}

int countLessEqual(BinaryNode p, Key x) {
```
// ...analogous but replace "p.data < x" with "p.data <= x"

The running time of both functions are clearly proportional to the height of the tree, since it makes one recursive call at each level of the tree. Finally, the answer to the range query is the number of keys that are less or equal to \(x_2\) minus the number that are strictly less than \(x_1\).

\[
\text{return countLessEqual(root, } x_2) - \text{countLessThan(root, } x_1)\]

**Solution 5:** In Fig. 4, we show all the intermediate results. The final tree is on the right.

![Solution to Problem 5](image)

Figure 4: Solution to Problem 5.

**Solution 6:** In Fig. 5, we show all the intermediate results. The final tree is on the right.

![Solution to Problem 6](image)

Figure 5: Solution to Problem 6.

**Solution 7:** In Fig. 6, we show all the intermediate results. The final tree is on the right.

**Solution 8:**
(a) We start with $n$ nodes at level 0, on average $pm$ nodes survive to level 1, $p^2n$ survive to level 2, and in general we expect $p^i n$ to survive to level $i$.

(b) Let $h$ denote the number of levels in the skip list. (We actually don’t care what this value is.) Summing the number of nodes that contribute to each level of the skip list, and employing the fact that, for $0 < c < 1$, $\sum_{i=0}^{\infty} c^i = 1/(1 - c)$, the total expected number of links is

$$\sum_{i=0}^{\infty} p^i n \leq n \sum_{i=0}^{\infty} p^i = \frac{n}{1 - p};$$

Observe that for any constant $p$, this is $O(n)$.

**Solution 9:** Let $x[1] < \cdots < x[n]$ denote the keys being stored in the splay tree. We will show that for $1 \leq i \leq n$, that after the call to $\text{splay}(x[i])$, the tree will have $x_i$ as the root, and keys $x_1, \ldots, x_i$ will form a left chain (see Fig 7.)

Assuming by induction that this is true after the first $i - 1$ splays, let’s consider the result of $\text{splay}(x[i])$. At this point $x[i]$ is the smallest key in the right subtree of the root. The last rotation of the operation is either zig, zig-zig, or zig-zag. We assert that it cannot be zig-zig, since this would mean that there is a key smaller than $x[i]$ in the right subtree. The remaining two cases are illustrated in Fig. 7. It is easy to see that, in either case, $x[i]$ (shown as 4 in the figure) is rotated to the root, and the previous keys form a left chain beneath it.
Solution 10:

(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 $\lfloor n/2 \rfloor$ nodes. If we start with a left chain and do $\lfloor n/2 \rfloor$ 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 8: Solution to Problem 10.](image)

```
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.)