Solutions to Midterm 1 Practice Problems

Solution 1:

(a) The number of leaves is exactly \([n/2]\). 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 \([n/2]\).

(b) True: Every non-leaf node has at least one incoming thread, one from each child that is non-null. For example, if the left child of some node \(u\) is not null, then the rightmost leaf in the subtree \(u.left\) has \(u\) as its inorder successor, and so its forward inorder thread pointing to \(u\).

(c) 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.

(d) \(n - k\): Initially every element is in its own set. As stated in the problem, every time a union occurs, it merges two distinct sets. Each time two sets are merged, we get one fewer set. So after \(k\) merges, we have \(n - k\) sets.

(e) The sibling is immediately before or after each element. The left child is at an even index and the right child is at an odd index. So, we can always find the sibling by toggling the lowest-order bit. If \(\oplus\) denotes the exclusive-or operator, we have \(\text{sibling}(i) = i \oplus 1\). Alternatively, we can do this by cases with \(\text{sibling}(i) = i + 1\) if \(i \mod 2 = 0\) and \(i - 1\) otherwise.

(f) Min: 0, Max: \((\log n) \pm O(1)\). The minimum occurs when the right child of the root is null. The maximum happens for a left-complete binary tree. (I believe that the most accurate expression is \([\log(n + 1)] - 1\).)

(g) 4 (but partial credit for any value in the range 3–7): A naive argument is that the third smallest element cannot reside below depth two, so there are at most 7 nodes to consider. A more refined argument observes that the root can never be the third smallest element. It can be either of the children of the root (whichever child is larger), and it can also be either of the children of the smaller child of the root. So, the best you hope to achieve is to inspect the two children of the root, and the two children of the smaller child of the root, for a total of 4 nodes.

(h) \(h + 1\): The worst case occurs when all the nodes in the tree are 3-nodes. Any insertion induces splits all the way back up to the root. In a tree of height \(h\), there are \(h + 1\) nodes along any path. Thus, the maximum number of splits is \(h + 1\).

(i) Min: 0, Max: \(h + 1\): If all the nodes of a 2-3 tree are 2-nodes, there are no red nodes at all. If all the nodes are 3-nodes, then the rightmost path in the 2-3 tree, has \(h\) edges and \(h + 1\) nodes. Each of these becomes a black-red pair, so there are \(h + 1\) red nodes.
(j) 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.

**Solution 2:** First observe that each union takes $O(1)$ time, and since there are at most $m$ unions, the total cost for all the unions is $O(m)$. To bound the time spent in the finds we classify the tree links as being of two types. Links that go directly to a root are said to be *shallow*, and all others are said to be *deep*. Since there are at most $m$ unions, there are at most $m - 1$ links (of both types) in the tree.

Each find operation can traverse at most one shallow link, which implies that the total time spent traversing shallow links is bounded by the number of find operations, which is $O(m)$. Every time we traverse a deep link, it becomes shallow (due to path compression). Therefore, the total time spent traversing deep links is at most the total number of links, which is $m - 1 = O(m)$. Since the total time for traversing both short and deep links is $O(m)$, the total time spent in all find operations is $O(m)$.

**Solution 3:**

(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 (see Fig. 1). We can rebalance each of these subtrees recursively (but reversing left and right on the right subtree).

![Figure 1: Rotating a tree into balanced form.](image-url)

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)`.

```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)
    recur on subtrees
}
```
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 4: To compute the inorder successor of a node, we first check whether its right child is not null. If so, (as in finding the replacement for a deletion) we find the leftmost node in the right subtree (see Fig. 2(a)). Otherwise, we iteratively follow parent links until we first find an ancestor where we lie in the left subtree of this ancestor (see Fig. 2(b)). If no such ancestor is found, $p$ must be the last inorder node of the tree, and we return null.

```
Node inorderSuccessor(Node p) { // find p’s inorder successor
    if (p.right != null) { // p has a right subtree?
        Node q = p.right // go to its right subtree
        while (q.left != null) q = q.left // find its leftmost node
        return q
    }
    else { // follow p’s ancestor chain
        Node q = p.parent // until we are a left child
        while (q != null && p == q.right) { // until we are a left child
            p = q
            q = q.parent
        }
        return q
    }
}
```

Figure 2: Inorder successor.
Observe that the program visits at most one node for each level of the tree, therefore its running time is proportional to the tree’s height.

**Solution 5:** 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 $p$. The initial call is `atDepth(0, d, root)`.

```java
void atDepth(int currDepth, int d, AVLNode p) {
    if (p == null || currDepth > d) return
    else
        atDepth(currDepth + 1, d, p.left)
        if (currDepth == d) output p.key
        atDepth(currDepth + 1, d, p.right)
}
```

We assert that the running time is proportional to the number nodes of depth $d$ or lower. First observe that we only visit nodes at depth $d+1$ or smaller. This is more than what we want, but observe that the number of nodes at depth $d+1$ is at most twice the number of nodes at depth $d$ (one for the left child and one for the right child), thus the number of nodes visited is asymptotically the same.

**Solution 6:** In both parts, we assume we have access to a utility `swap(u, v)`, which swaps two node pointers. This is not possible in Java (since parameters are passed by value), but it is fine for pseudocode.

(a) Our solution operates recursively. If $u$ is null, we return. Otherwise, we recursively invoke `swapRight` on its right subtree, and when it returns, we swap $u$’s children. The initial call is `swapRight(root)`. (By the way, the function does not change the root of the subtree, so there is no harm in ignoring the return value.)

```java
Node swapRight(Node u) {
    if (u != null) {
        swapRight(u.right) // apply to u's right subtree
        swap(u.left, u.right) // swap u's subtrees
    }
    return u
}
```

(b) Our solution operates recursively. The initial call is `root = swapMerge(root1, root2)` where `root` is the root of the resulting tree.

While the code is similar to the leftist-heap function `merge`, there are a few significant differences. First, there are no NPL values, and the left-right swap always takes place. There are three statements from the leftist-heap algorithm that must be changed.
(a) if (u == null) return v: We still need to perform swaps along v’s right chain, so it should be “return swapRight(v)”.

(b) if (v == null) return u: Symmetrically, it should be “return swapRight(u)”.

(c) if (u.left == null) u.left = v: Again, we still need to swap v’s right chain, so it should be “u.left = swapRight(v)”. In our code, we simply skip this step, since this small optimization is not useful here.

Node swapMerge(Node u, Node v) {
    if (u == null) return swapRight(v) // if either is empty, swap the other
    if (v == null) return swapRight(u)
    if (u.key > v.key) swap(u, v) // swap so that u has smaller key
    u.right = swapMerge(u.right, v) // recursively merge u’s right subtree
    swap(u.left, u.right) // swap the subtrees after merging
    return u // return the root of final tree
}

As a side comment, there is a variant of the leftist heap, called a skew heap, which is based on swapMerge. This variant does not store NPL values, and so does not guarantee worst-case efficiency. However, Sleator and Tarjan proved that it is efficient in the amortized sense.

Solution 7: This is proved by induction on the height of the tree. For the basis cases, observe that an AVL tree of heights $h = 0$ or 1 has a root node, and so it is full at depth $\lfloor h/2 \rfloor = 0$.

Let us make the (strong) induction hypothesis that for any $h \geq 2$, an AVL tree of strictly smaller height $h' < h$ is full at level $\lfloor h'/2 \rfloor$, and we will use this to prove the result for $h$ itself.

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, by including the root level, the entire tree is full at one higher depth, that is, $\lfloor h - 2 \rfloor + 1$. Using the identity that $\lfloor x - 1 \rfloor = \lfloor x \rfloor - 1$, we conclude that the entire tree is full up to 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.

Solution 8:

(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
    
}
(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.)

```java
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 9:

(a) For $i \geq 0$, let $n(i)$ denote the number of nodes at depth $i$ in an alternating 2-3 tree (where the root is at depth 0. Clearly, $n(0) = 1$, and for $i \geq 1$:

$$n(i) = \begin{cases} 2n(i - 1) & \text{if } i \text{ is odd} \\ 3n(i - 1) & \text{if } i \text{ is even.} \end{cases}$$

By expanding two levels of this recurrence, it is easy to see that for any $n \geq 2$ (irrespective of $i$’s parity) $n(i) = 6n_{i - 2}$. By repeatedly expanding this (or induction, if you prefer), it is easy to see that $n(2k) = 6^k n(0) = 6^k$. Also, since $2k + 1$ is odd, we have $n(2k + 1) = 2 \cdot 6^k$.

Therefore, we have the following general formula for the number of nodes at level $i$ of the alternating 2-3 tree:

$$n(i) = \begin{cases} 2 \cdot 6^{(i-1)/2} & \text{if } i \text{ is odd} \\ 6^{i/2} & \text{if } i \text{ is even.} \end{cases}$$

This can also be expressed without resorting to cases with the following equivalent formula

$$n(i) = 2^{[i/2]} 3^{[i/2]}.$$
(b) We can use the number of nodes to derive the number of keys. If \( i \) is even, all the nodes are 2-nodes, and since each contains a single key, we have \( k(i) = n(i) \). If \( i \) is odd, all the nodes are 3-nodes, and each contains two keys, so we have \( k(i) = 2n(i) \). In summary, we have

\[
k(i) = \begin{cases} 
4 \cdot \frac{6^{(i-1)/2}}{\sqrt{2}} & \text{if } i \text{ is odd} \\
\frac{6^{i/2}}{\sqrt{2}} & \text{if } i \text{ is even}.
\end{cases}
\]

**Solution 10:** We will show that the amortized cost is \( t \) for some constant \( t \). The expanded array has size \( \gamma m \) of which \( m \) are occupied, so the next reallocation occurs after at least \( \gamma m - m = (\gamma - 1)m \) operations. If we charge \( t \) tokens for each operation, and use one for each push, we accrue \( t - 1 \) tokens per operation, for a total of at least \( (t - 1)(\gamma - 1)m \) tokens. We need these to pay the copying cost of \( \delta \gamma m \). (A common error is to take the cost to be \( \delta m \), but note that the size of the array being copied is \( \gamma m \), not \( m \), which increases the copying cost by a factor of \( \gamma \).) Therefore, we select \( t \) so that \( (t - 1)(\gamma - 1)m \geq \delta \gamma m \). Setting \( t = 1 + \delta \gamma / (\gamma - 1) \) satisfies this.