Solutions to Midterm Exam 1

Solution 1:

(1.1) The key 5 is inserted into the node containing (6 : 8), which causes this node to become overfull (see Fig. 1). We split the node, which results in two children (5) and (8) and the key 6 is promoted to the parent. The promotion of 6 causes the parent node to contain (3 : 6 : 11), which is now overfull. By the algorithm given in class, this node splits, resulting in the two nodes (3) and (11), and 6 is promoted to the root node, which now contains (6 : 15). (Note that a rotation is also possible, but the algorithm given in class performs splits rather than rotations when doing insertions.)

(1.2) The deletion of 4 causes its leaf node to become underfull. Its only sibling 1 cannot spare a key, so we perform a merge. This demotes the parent’s key 3 resulting in the leaf node (1 : 3) (see Fig. 2). The parent has lost its key is now underfull. We look to its sibling (8 : 12), which can spare a key. We perform an adoption (key rotation) which pulls the 8 up to the parent, moves the 5 down to the underfull child, and transfers the child 6 to the left.

Solution 2:

(a) Answer: Yes it is possible; \( n + 1 \) threads. Explanation: We showed in class that any binary tree with \( n \) nodes has \( n + 1 \) null pointers. All of these null pointers become threads, which results in \( n + 1 \) threads. (Note that the two null threads at the leftmost and rightmost edges of the tree are counted.)
(b) Answer: (1) Definitely full, (5) Height larger by exactly 1. Explanation: Because the keys are inserted in arbitrary order, the tree containing the seven keys \{1, 3, \ldots, 13\} can have pretty much any possible structure. This tree has exactly eight null pointers, and (because they alternate with the original keys) the newly inserted keys \{0, 2, \ldots, 14\} create one new node in each of these null pointers. (This fact is not hard to prove by induction.) As a consequence, the new tree is full (since the internal nodes hold the odd keys and the even keys fill in all the null pointers), and the height has increased by exactly one. Since the original tree is arbitrary, the final tree need not be complete.

(c) Min: 1, Max: \(n/2\). Explanation: The problem stipulates that the number of keys is even. In such a case it is not possible for all the nodes to be black, because the tree is effectively a complete binary tree, and any complete binary tree of height \(h\) has \(2^{h+1} - 1\) nodes, which is always odd. Thus, there needs to be at least one red node. At the other extreme, every node is a red-black combination, which implies that exactly half of the nodes are red.

(d) Answer: The number of roots per level is \(\langle 1, 0, 1, 1, 0 \rangle\). Explanation: Because of the balanced nature of merging, the number of leaves in any tree following merge-trees is a power of 2. Since \(13 = 1 + 4 + 8 = 2^0 + 2^2 + 2^3\), the roots are at levels 0, 2, and 3. (We did not ask for the number of nodes per level, but this would be \(\langle 13, 6, 3, 1, 0 \rangle\).)

(e) Answer: (1) Smallest. Explanation: The nodes of a treap are ordered like a heap, with smaller priorities closer to the root. So, if a node appears at the root, it has the smallest priority.

(f) Answer: Not an issue. Explanation: This would be an issue if they hacker could actually see the internals of the data structure. However, without access to the random number sequence nor to the private members of the data structure, the hacker has no way of knowing the heights of the nodes. Thus, such an attack is not possible.

Solution 3:

(a) Our strategy is to move \(p\) and \(q\) up the tree in a coordinated manner until they converge at the LCA. If they are at different levels, the lower of two is moved up. Otherwise, it doesn’t matter which one we move up.

```java
Node LCA(Node p, Node q) {
    while (p != q) {
        if (p.level <= q.level) p = p.parent
        else q = q.parent
    }
    return p
}
```

The problem can also be solved recursively:

```java
Node LCA(Node p, Node q) {
    if (p == q) return p
    else if (p.level <= q.level) return LCA(p.parent, q)
    else return LCA(p, q.parent)
}
```
Assuming that $p$ and $q$ are valid inputs, there should be no need to check for null pointers. Clearly, the time is $O(h)$, where $h$ is the height of the tree.

(b) This is made trickier by the fact that two nodes may be at the same level, and yet one may be slightly lower because it is a red node. If we determine that the two nodes are both at the same level, but are not equal to each other, we advance whichever is a red node. If they are both black, we advance both.

```java
Node AA_LCA(Node p, Node q) {
    while (p != q) { // repeat until meeting at the LCA
        if (p.level < q.level) { // p is lower?
            p = p.parent // ... move p up
        } else if (q.level < p.level) { // q is lower?
            q = q.parent // ... move q up
        } else if (p.parent.level == p.level) { // p is red?
            p = p.parent // ... move p up
        } else if (q.parent.level == q.level) { // q is red?
            q = q.parent // ... move q up
        } else { // both are black
            p = p.parent; q = q.parent // ... move both up
        }
    }
    return p // now: p == q == LCA(p,q)
}
```

The recursive version has a very elegant alternative. The idea is to assign each node a semi-level. The black node in any red-black pair is assigned a level number that is +0.5 higher. This way, we can pretend as if levels numbers behave in a purely monotonic fashion.

```java
float semiLevel(Node p) { // same as p.level, but black node is +0.5 higher
    return p.level + (p.parent.level == p.level ? 0 : 0.5)
}

Node AA_LCA(Node p, Node q) {
    if (p == q) return p
    else if (semiLevel(p) <= semiLevel(q)) return AA_LCA(p.parent, q)
    else return AA_LCA(p, q.parent)
}
```

There is one small bug in both the above algorithms. If either $p$ or $q$ is the root, then $p.parent.level$ is undefined (and same for $q.parent.level$). As we did with nil, we could create a sentinel node to act as the root’s parent, whose level is higher by one.

**Solution 4:**

(a) The easiest approach is to develop the recurrence two levels at a time. We claim that the number of nodes increases by a factor of 6. Each 2-node at level $i-2$ gives rise to two 3-nodes at level $i-1$, each of which gives rise to three 2-nodes at level $i$, for a total of 6. Similarly,
each 3-node at level \( i - 2 \) gives rise to three 2-nodes at level \( i - 1 \), each of which gives rise to two 3-nodes at level \( i \), for a total of 6.

Because of skewness, this pattern does not kick in until levels 3 and higher. We provide depths 0, 1, and 2 as basis cases, yielding:

\[
 n(i) = \begin{cases} 
 1 & \text{if } i = 0, \\
 2 & \text{if } i = 1, \\
 5 & \text{if } i = 2, \\
 6n(i - 2) & \text{otherwise.} 
\end{cases}
\]

An alternative is to work one level at a time. Unfortunately, we cannot easily predict how many the nodes will be 2-nodes and 3-nodes, so we should track these quantities separately. Let \( n_2(i) \) be the number of 2-nodes at level \( i \), \( n_3(i) \) be the number of 3-nodes. Each 3-node on level \( i - 1 \) gives rise to three 2-nodes on level \( i \), and each 2-node on level \( i - 1 \) gives rise to two 3-nodes on level \( i \). Including the basis cases for levels 0 and 1, we have

\[
 n_2(0) = 1 \quad n_3(0) = 0, \\
 n_2(1) = 1 \quad n_3(1) = 1, \\
 n_2(i) = 3n_3(i - 1) \quad n_3(i) = 2n_2(i - 1) \quad \text{for } i \geq 2.
\]

Finally, have \( n(i) = n_2(i) + n_3(i) \).

(b) This is made tricky by the fact that depth 0 does not follow the trend. Starting at the basis cases for \( i = 1 \) (\( n(i) = 2 \)) and \( i = 2 \) (\( n(i) = 5 \)), we grow by a factor of 6 with every two levels. Therefore, for even levels (2 and higher), we have \( n(i) = 5 \cdot 6^{(i-2)/2} \) and for odd levels (1 and higher) we have \( n(i) = 2 \cdot 6^{(i-1)/2} \). We need to treat level 0 as a special case.

\[
 n(i) = \begin{cases} 
 1 & \text{if } i = 0, \\
 2 \cdot 6^{(i-1)/2} & \text{if } i \text{ is odd,} \\
 5 \cdot 6^{(i-2)/2} & \text{if } i \geq 2 \text{ and even.} 
\end{cases}
\]

**Solution 5:**

(a) 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, and yields the amortized cost.

(b) The expanded array has size \( 2m \) of which \( m \) are occupied, so the next reallocation occurs after at least \( 2m - m = m \) 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 \( (t - 1)m \) tokens. We need these to pay the copying cost of \( (2m)/\lg(2m) \). Therefore, we select \( t \) so that \( (t - 1)m \geq (2m)/\lg(2m) \). We may choose any \( t \) such that \( t \geq 1 + 2/\lg(2m) \). In the limit, as \( m \to \infty \), this converges to an amortized cost of 1.
Solution 4:  (Section 0201)

(a) The easiest approach is to develop the recurrence three levels at a time. We claim that the number of nodes increases by a factor of $2 \cdot 2 \cdot 3 = 12$ with each three levels. No matter which level we start at, the next three levels expand by $2 \cdot 2 \cdot 3 = 2 \cdot 3 \cdot 2 = 3 \cdot 2 \cdot 2 = 12$ nodes. We provide basis cases for depths 0, 1, and 2, yielding:

$$n(i) = \begin{cases} 
1 & \text{if } i = 0, \\
2 & \text{if } i = 1, \\
4 & \text{if } i = 2, \\
12n(i - 3) & \text{otherwise.}
\end{cases}$$

(b) There are three cases, depending on whether $i \mod 3$ is 0, 1, or 2. For the basis cases ($i = 0, 1, 2$) the answer comes from part (a). Following this, the value grows by factors of 12 with every three levels. Thus we have:

$$n(i) = \begin{cases} 
1 \cdot 12^{\lfloor i/3 \rfloor} & \text{if } i \equiv 0 \mod 3, \\
2 \cdot 12^{\lfloor i/3 \rfloor} & \text{if } i \equiv 1 \mod 3, \\
4 \cdot 12^{\lfloor i/3 \rfloor} & \text{if } i \equiv 2 \mod 3,
\end{cases}$$

There is an even cuter way to express this, avoiding the cases. $n(i) = 2^{(i \mod 3)} \cdot 12^{\lfloor i/3 \rfloor}$.

Solution 5:  (Section 0201)

(a) We will show that the amortized cost is $t$ for some constant $t$. The expanded array has size $\alpha m$ of which $m$ are occupied, so the next reallocation occurs after at least $\alpha m - m = (\alpha - 1)m$ 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 $(t - 1)(\alpha - 1)m$ tokens. We need these to pay the copying cost of $\beta \alpha m$, and therefore, we select $t$ so that $(t - 1)(\alpha - 1)m \geq \beta \alpha m$. Setting $t = 1 + \beta \alpha / (\alpha - 1)$ satisfies this, and yields the amortized cost.

(b) The expanded array has size $2m$ of which $m$ are occupied, so the next reallocation occurs after at least $2m - m = m$ 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 $(t - 1)m$ tokens. We need these to pay the copying cost of $\sqrt{2m}$. Therefore, we select $t$ so that $(t - 1)m \geq \sqrt{2m}$. We may choose any $t$ such that $t \geq 1 + 1/\sqrt{2m}$. In the limit, as $m \to \infty$, this converges to an amortized cost of 1.