Solutions to Homework 1: Basics, Union-Find, and Heaps

Solution 1:

(a) See Fig. 1(a)

(b) Preorder: \{b, g, e, h, c, d, i, f, \ell, k, a, m\}

(c) Postorder: \{h, j, e, g, c, f, i, k, a, m, \ell, d, b\}

(d) See Fig. 1(b)

Solution 2:

(a) See Fig. 2 (bottom left).
(b) See Fig. 2. Since trees 4 and 10 have the same ranks, we follow the convention of the code given in class and link the first tree under the second, and the rank of the resulting tree is higher by 1. Since 9 is of lower rank compared to 16, so we link 9 as a child of 16, and the rank of 16 does not change.

(c) See Fig. 2. The path from 5 leads through 7, 4, before arriving at the root 10. We compress the path by linking 5 and 7 directly to 10. The rank of the tree does not change (even though its height decreases.) The final answer is 10.

Solution 3:

(a) The NPL values were already shown for heap $H_1$. For heap $H_2$, see Fig. 3

(b) We’ll follow the intuitive approach given in the lecture notes, rather than tracing the formal algorithm. We begin by merging them along their rightmost paths (see Fig. 4(a)). Next, we update the NPL values, but only node 6 changes its NPL value, which is now 2. Finally, we check whether the nodes on rightmost path satisfy the leftist property by comparing the NPLs of their children. Nodes 2, 3, and 14 violate this (right NPL exceeds left NPL), and we swap their left and right subtrees (see Fig. 4(b)).

Solution 4: The process consists of the following steps. First, we locate $v$ node by searching through the children of the root (see Fig. 5(a)).

Next, we unlink $v$ from the tree. There are two cases depending on whether it is the first child of the root or one of the other children. If its the first child of the root, we just advance the root’s first child link to point to its second child. Otherwise, we find the node $c$ that immediately precedes
null

Figure 5: Promoting a node v to be the root.

v in the child list, and adjust c’s next-sibling link to skip over v (see Fig. 5(b)). Once v is removed, we insert the old root node as v’s first child, and copy v’s old first-child link to become the old root’s next-sibling link. Finally, we update the root pointer to point to v (see Fig. 5(c)). The pseudocode is presented below.

```c
void promote(Node v) { // make v the new root
    Node c = root.firstChild
    if (c == v) { // v is the first child of root
        root.firstChild = v.nextSibling // remove v from root child list
    } else {
        while (c.nextSibling != v) c = c.nextSibling // find v’s predecessor sibling
        c.nextSibling = v.nextSibling // unlink v
    }
    root.nextSibling = v.firstChild // link root into v’s child list
    v.firstChild = root // make old root v’s first child
    v.nextSibling = null // v now has no sibling
    root = v // make v the new root
}
```

Solution 5:

(a) As remarked in the problem description, when the expansion takes place, each of the arrays has $n_L + n_U = m'/3$ available empty positions (see Fig. 6(a)).

We have just inserted a new element into one of these stacks, implying that this stack has $m'/3 - 1$ available entries remaining (and the other has one more). The fastest we can induce
the next overflow is to push elements into this stack (see Fig. 6(b)). This happens after \( m'/3 \) pushes \((m'/3 - 1)\) to fill the stack, and one more to induce the overflow.

(b) Because the expansion cost depends on the number of entries that are copied, the worst-case occurs when both stacks are as full as possible, and we perform one additional push. The worst case occurs when we are using every element of the current array, implying that there are \( m' \) elements in the current stack, leading to an expansion cost of \( m' \) (see Fig. 6(c)).

(c) We will show that the amortized cost is at most 4. Our proof will involve a token-based approach. To avoid begging the question of why 4 is the correct answer, let’s start by assuming that the amortized cost is some value \( \tau \), and we will show that \( \tau = 4 \) does the job. We will show that for any sequence of \( m \) operations, the total cost, denoted \( T(m) \), is at most \( \tau m \).

To make the analysis simpler, we break the sequence of \( m \) operations into runs. Each run starts just after the prior expansion and ends with the next expansion. We ignore the first and last runs, since they don’t follow the pattern, but they are easy to account for. (The first run involves only a constant number of operations, and the final run may not end in an expansion, which only makes the total cost smaller.)

Each time we perform an operation, we will use one token to pay for the operation, leaving \( \tau - 1 \) tokens to put in our bank account. In (a) we argued that there are at least \( m'/3 \) operations until the next expansion, implying that we have banked at least \((\tau - 1)m'/3\) tokens by the end of the run. From (b), we know that the worst-case expansion cost is \( m' \).

In order to pay for this, it suffices to set \( \tau \) large enough so that

\[
(\tau - 1)m'/3 \geq m' \quad \text{or equivalently} \quad \tau \geq m' \frac{3}{m'} + 1 = 4.
\]

Therefore, setting \( \tau = 4 \) is sufficient, which implies that the amortized cost is at most 4.

Is this tight? No. To see why, see Challenge Problem 2.

**Solution to Challenge Problem 1:**

(a) (iii): The results are correct, but the running time of merge may be not be \( O(\log n) \).

Alice used “max” rather than “min” in her NPL computation. Notice that this is equivalent to using height rather than NPL to decide which subtree goes on the right. Unfortunately for Alice, having the lower height tree on the right side does not guarantee that the number of nodes in the rightmost chain is \( O(\log n) \). To see why, consider the example shown in Fig. 7.

- \( n = 1 + 2 + 3 + \ldots + r = r(r + 1)/2 \)
- \( r \approx \sqrt{2n} = \Omega(\sqrt{n}) \)

![Figure 7: Using height rather than NPL does not work.](image)

This tree satisfies the property that the height of the left subtree is never smaller than the height of the right subtree. Let \( r \) denote the number of nodes along the rightmost path.
of the tree. By adding up the nodes in each of the left subtrees of these nodes, we have
\[ n = 1 + 2 + \cdots + r. \]
This is an arithmetic progression, and by the standard formula, we have
\[ n = r(r + 1)/2. \]
For large \( r \), this implies that \( r \approx \sqrt{2n} = \Omega(\sqrt{n}) \). This is asymptotically larger than \( O(\log n) \). Thus, the time to merge two such trees is worse than \( O(\log n) \) (but it is still better than \( O(n) \)). The valid heap order will still be maintained by the merging procedure.

(b) (iii): The results are correct, but the running time of merge may be not be \( O(\log n) \).

Bob’s mistake is more serious than Alice’s. If the right subtrees have higher NPL values, then in the worst case, the tree may degenerate to a chain of \( n \) nodes along the right side. In this case, the merge process will take \( \Omega(n) \) time. The valid heap order will still be maintained by the merging procedure.

Solution to Challenge Problem 2: Recalling the analysis of Problem 5, we know that at the start of the run there are \( m'/3 \) elements in the stack and each stack has \( m'/3 \) available entries. We need at least \( m'/3 \) pushes to induce another expansion, but we may push as many as \( 2m'/3 \) entries. To determine where the worst-case resides, let the actual number of pushes be \( (1 + \alpha)m'/3 \), for \( 0 \leq \alpha \leq 1 \). Let \( \tau \) denote the amortized cost. As before, each cheap operation costs +1 unit, and the remaining \( \tau - 1 \) are banked with each operation. The total number of banked tokens until the next rebuild is \( (1 + \alpha)m'/3 \). The expansion involves copying these elements plus the original \( m'/3 \), for a total cost of \( (1 + \alpha)m'/3 + m'/3 = (2 + \alpha)m'/3 \). In order to pay for this, it suffices to set \( \tau \) large enough so that
\[
(\tau - 1)(1 + \alpha)m'/3 \geq (2 + \alpha)m'/3 \quad \text{or equivalently} \quad \tau \geq \frac{2 + \alpha}{1 + \alpha} + 1 = \left( 1 + \frac{1}{1 + \alpha} \right) + 1.
\]
This is maximized when \( \alpha \) is minimized. Setting \( \alpha = 0 \), it follows that \( \tau \geq (1 + 1) + 1 = 3 \). Setting \( \tau = 3 \) satisfies this, implying that the amortized cost is at most 3.

---

\(^1\)This is also an upper bound. Here’s a sketch. Let \( T(r) \) denote the minimum number of nodes in a tree whose rightmost chain has \( r \) nodes. If we remove the root, the left subtree needs to have at least \( r - 1 \) nodes in order to match the height of the right subtree, which can be arrayed as a left chain of \( r - 1 \) nodes. The right subtree has \( r - 1 \) nodes on its rightmost chain, and therefore by induction it has at least \( T(r - 1) \) nodes. Thus, we have \( T(r) = 1 + (r - 1) + T(r - 1) \), which solves to \( T(r) = r(r + 1)/2 \).