CMSC 420: Fall 2022

Solutions to Homework 1: Trees and More

Solution 1:

(a) See Fig. 1(a)

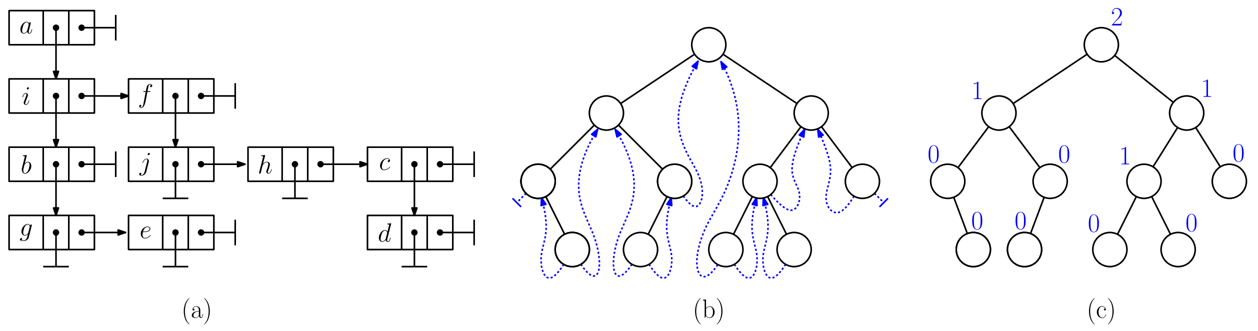

Figure 1: Rooted trees.

(b) Preorder: $\langle d, j, c, g, e, a, f, b, i, h, k\rangle$

(c) Inorder: $\langle c, g, j, e, a, d, i, b, h, f, k\rangle$

(d) Postorder: $\langle g, c, a, e, j, i, h, b, k, f, d\rangle$

(e) See Fig. 1(b)

(f) See Fig. 1(c)

Solution 2: This is a bit tricky, so please check out the proof carefully. The recursive structure is very similar to the one given in the lecture on union-find, but that proof was simpler because rank and height are basically identical. In contrast, with size-based merging, we cannot infer whether the taller tree has more nodes nor that the tree with more nodes is taller. We need to consider all the possibilities.

Lemma: If the above size-based merging process is used, a tree of height h has at least 2^h elements.

Proof: Given any T, let h denote the height of T, let n denote its size. We will show that $n \geq 2^h$.

Our proof is based on induction on the number of unions u used to build the tree. For the basis ($u = 0$) we have a tree consisting of a singleton, that is, $n = 1$. This tree has height $h = 0$. Since $1 = 2^0$, we have $n \geq 2^h$,as desired.

For the induction step, we will make the (strong) induction hypothesis that any tree built with strictly fewer than u unions satisfies the lemma, and we will prove it for a tree built with exactly u unions. Consider any such tree T. Its last union involved merging two trees, denoted T' and T'', with respective sizes n' and n'', and respective heights h' and h''. Since

T' and T'' each were constructed using fewer unions than T, we may apply the induction hypothesis, which implies that $n' \geq 2^{h'}$ and $n'' \geq 2^{h''}$. Following the merge we have a total $n = n' + n''$ elements. We may assume without loss of generality that $h' \leq h''$. (If not, swap the trees T' and T''.)

There are two possibilities. If $h' = h''$, then the resulting tree has height $h = h' + 1 = h'' + 1$. The number of elements in the final tree is

$$n \;=\; n' + n'' \;\geq\; 2^{h'} + 2^{h''} \;=\; 2^{h-1} + 2^{h-1} \;=\; 2 \cdot 2^{h-1} \;=\; 2^h.$$

On the other hand, suppose that $h' < h''$. There are two subcases.[1] If $n'' \geq n'$, then T' is linked as a child of T'', and the final tree has height equal to $h = h''$. The number of elements is

$$n \;=\; n' + n'' \;\geq\; n'' \;\geq\; 2^{h''} \;=\; 2^h.$$

On the other hand, if $n'' < n'$, then T'' is linked as a child of T', and the final tree has height $h = \max(h', h'' + 1) = h'' + 1$. The number of elements is

$$n \;=\; n' + n'' \;>\; 2n'' \;\geq\; 2 \cdot 2^{h''} \;=\; 2^{h''+1} \;=\; 2^h.$$

In any case we obtain the desired conclusion, which completes the proof.

Solution 3:

(a) We create a helper function `preDepth(BSTNode u, int d)`, which is given a node `u` and its depth `d`. The initial call is `preDepth(root, 0)`. Assuming that `u` is not `null`, it prints the node's information and then invokes itself on the node's left and right children, each deeper by one level.

```
void preDepth(BSTNode u, int d) {
    if (u == null) return           // fell out of the tree
    print(u.key, d)                 // print this node's key and depth
    preDepth(u.left, d+1)           // traverse the left subtree
    preDepth(u.right, d+1)          // traverse the right subtree
}
```

(b) We create a helper function `int postNPL(BSTNode u)`, which is given a node `u` and returns its NPL value. The initial call is `postNPL(root)`. Assuming that `u` is not `null`, we invoke the procedure recursively on each child, and save the returned NPL values. We then compute our own NPL value from these and print it.

```
void postNPL(BSTNode u) {
    if (u == null) return           // fell out of the tree
    leftNPL = postNPL(u.left)       // traverse the left subtree
    rightNPL = postNPL(u.right)     // traverse the right subtree
    myNPL = 1 + min(leftNPL, rightNPL)
    print(u.key, myNPL)             // print this node's key and NPL
}
```

[1] It is tempting here to make the same inference as in the lecture, namely that the height of the final tree will be h'', but this is *incorrect*. The reason is that T' may have more elements, in which case we merge T'' into T' and the final height will be $\max(h', h'' + 1) = h'' + 1$. Be sure you understand this.

(c) We exploit the same ideas of parts (a) and (b) by passing in the depth and passing back the height. (The height computation is essentially the same as the NPL computation, just replacing `min` with `max`). The initial call is `postAll(root, 0)`.

```
void postAll(BSTNode u, int d) {
    if (u == null) return               // fell out of the tree
    leftHgt = postAll(u.left, d+1)      // traverse the left subtree
    rightHgt = postAll(u.right, d+1)    // traverse the right subtree
    myHgt = 1 + max(leftHgt, rightHgt)
    print(u.key, d, myNPL)              // print this node's stats
}
```

Solution 4:

(a) When the expansion takes place, the array is full, meaning that $m = n_1 + n_2$. The new array has size $m' = 3 \cdot \max(n_1, n_2)$. Subject to the constraint that $m = n_1 + n_2$, $\max(n_1, n_2)$ is minimized when $n_1 = n_2 = m/2$. (If we make one smaller, then the other must be larger, and hence the max increases.) This implies that $m' \geq 3m/2$. The number of new slots created in the expansion is $m' - m \geq 3(m/2) - m = m/2 = m'/3$. It takes at least $m'/3$ operations to fill all these entries (and this will happen if user does nothing but pushes).

(b) The next expansion occurs when the new array is entirely full. The reallocation cost is equal to the number of elements copied, which is m'. (Notice that this does depend on the relative sizes of the two stacks at the time of the expansion.)

(c) We assert that the amortized cost is $c = 4$. We will prove this by a token-based argument. In particular, we will charge 4 tokens for each operation, and we will show that there are always enough tokens to pay for all the operations.

We will break up the execution sequence into a series of runs, where a run starts just after the previous expansion and ends with the next expansion. We will show the above assertion holds for any run, and hence it holds for the entire execution sequence. Because our focus is on large n, we will ignore the first run. Consider a run that is described in parts (a) and (b), where we start with m' elements of which $m'/3$ are empty.

Whenever a stack operation is performed, one of the $c = 4$ tokens is used to pay for actual operation, and the remaining 3 tokens are banked. From (a) at least $m'/3$ operations will occur before the next expansion, implying that we have banked at least $3m'/3 = m'$ tokens. From (b), the expansion costs us m' units, implying that we have collected enough tokens to pay for the expansion at the end of the run. It follows that the average cost of an operation during the run is 4, as desired. (How did we choose $c = 4$? If you reverse-engineer the above argument, you will see that we need c to satisfy the inequality $(c - 1)(m'/3) \geq m'$, and the smallest c that works is 4.)

To make this more concrete, consider the example from the homework handout, where $m = 12$ (see Fig. 2). By (a), we may assume that the run started when $n_1 = n_2 = 6$, and we just expanded to an array of size $m' = 3 \cdot \max(n_1, n_2) = 18$. We can perform $m' - m = 6$ pushes before the array again becomes full. We collect $4 \cdot 6 = 24$ tokens from these pushes. Of these, 6 go to pay for the push operations, themselves, leaving 18 in the bank. The next expansion involves copying all $m' = 18$ elements, and we have enough in the bank to pay for this. (Note that we don't care about the size of the final array.)

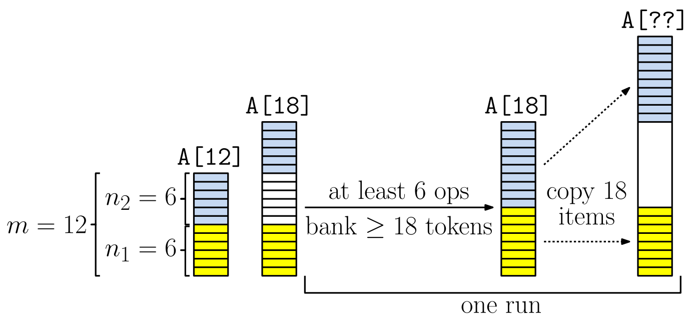

Figure 2: Dual stack example.

Solution 5:

(a) The initial recursive call is `merge(6, 3)`. Since $6 > 3$, we swap `u` and `v`, so the next call is `merge(3.right, 6)` = `merge(7, 6)` (see Fig. 3). This also swaps the arguments, so the next call is `merge(6.right, 7)` = `merge(10, 7)`. This also swaps the arguments, so the next call is `merge(7.right, 10)` = `merge(9, 10)`. In this case there is no swap. We find that `9.left == null`, so we set `9.left` point to the subtree rooted at `10`, and we return before making any more recursive calls.

So, in summary, the sequence of recursive calls is:

```
merge(6, 3)
merge(7, 6)
merge(10, 7)
merge(9, 10)
```

Note that the call `merge(null, 10)` does not take place.

(b) On return, the NPL values shown in red are updated as shown in Fig. 3. The leftist condition is violated at node `3`, and so its left and right children are swapped.

Solution to Challenge Problem 1: To insert, just create a new node with the key, and merge it with the current heap. Update the root node to the resulting tree. To extract the minimum, save the root's key as your result. Then merge its left and right subtrees. Since `merge` runs in $O(\log n)$ time, it follows easily that each of these runs in $O(\log n)$ time.

```
void insert(Key x) {
    LHNode u = new KHNode(x)
    root = merge(root, u)
}
Key extract-min() {
    if (root == null) Error - Extract from empty heap
    result = root.key
    root = merge(root.left, root.right)
    return result
}
```

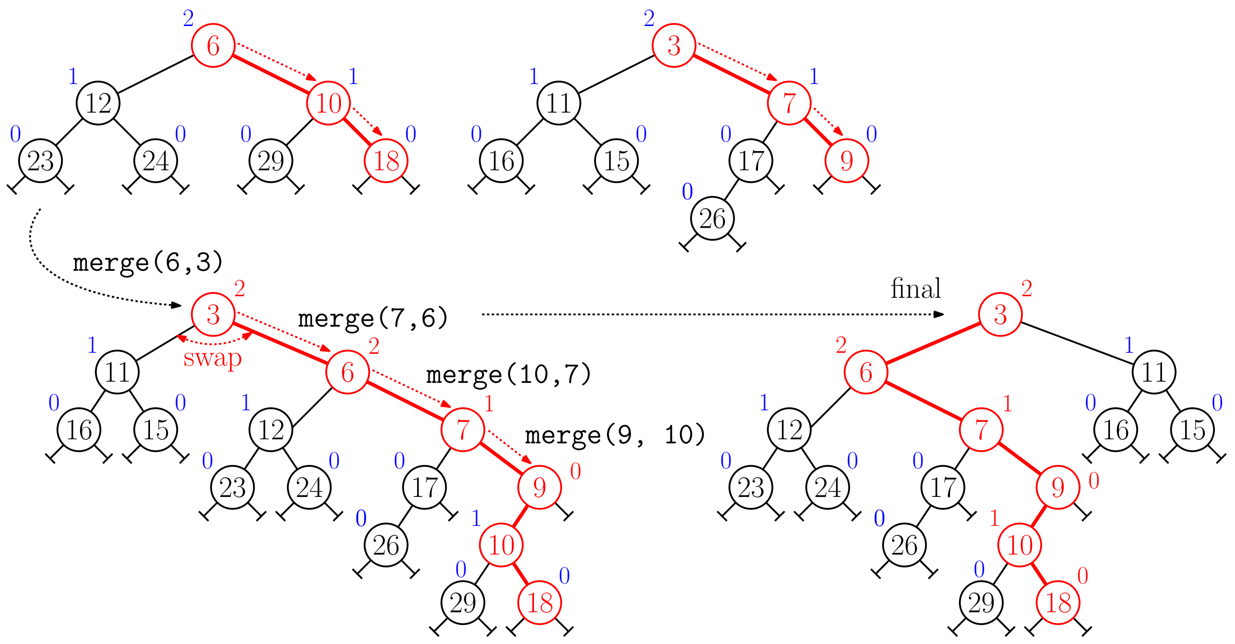

Figure 3: Merging leftist heaps.

Solution to Challenge Problem 2:

Lemma: In any binary tree with $n \geq 1$ nodes, there exists a path from the root to a node having a null child link, such that the path has at most $\lg(n+1)$ nodes.

Proof 1: In class, we showed that in any leftist heap with n nodes, the rightmost path has at most $\lg(n+1)$ nodes. The proof does not make use of any of the heap-ordering properties, just the fact that the tree is leftist. Any binary tree can be converted into a leftist tree by simply checking the NPL values at each node, and whenever the NPL value of the right child is more than the left child, we swap these two children. Swapping left and right children does not alter the length of a path from the root to a leaf. Therefore, by appealing to the result from the lecture notes, the resulting tree's rightmost path has the desired length.

Proof 2: This proof is by induction on the number of nodes in the tree. For the basis case of $n = 1$, the root nodes itself has a null child, and hence the path has one node. The path length is $1 = \lg 2 = \lg(n+1)$, as desired.

For $n \geq 2$, let us make the (strong) induction hypothesis that any tree having $1 \leq n' < n$ nodes has a path of the desired form consisting of at most $\lg(n'+1)$ nodes, and we will use this hypothesis to prove the result for n itself. Consider a tree T with n nodes, which consists of a root node and two subtrees, denoted T' and T''. Let n' and n'' denote the respective numbers of nodes in these subtrees. These node counts satisfy $n = 1 + n' + n''$. We may assume without loss of generality that $n'' \geq n'$ (otherwise, swap the subtrees). Thus, $n = 1 + n' + n'' \geq 1 + 2n'$. If $n' = 0$, then the root node itself has a null child link, and the lemma trivially holds. Otherwise, we have $1 \leq n' < n$, and so we may apply the induction hypothesis, which implies that T' has a path from its root to node with a null child link with at most $\lg(n'+1)$ nodes. By adding the root to this path, we have a path in T with node count at most

$$1 + \lg(n'+1) \;=\; \lg(2(n'+1)) \;=\; \lg((1+2n')+1) \;\leq\; \lg(n+1),$$

where in the last step we used the fact that $n \geq 1 + 2n'$. This completes the proof.