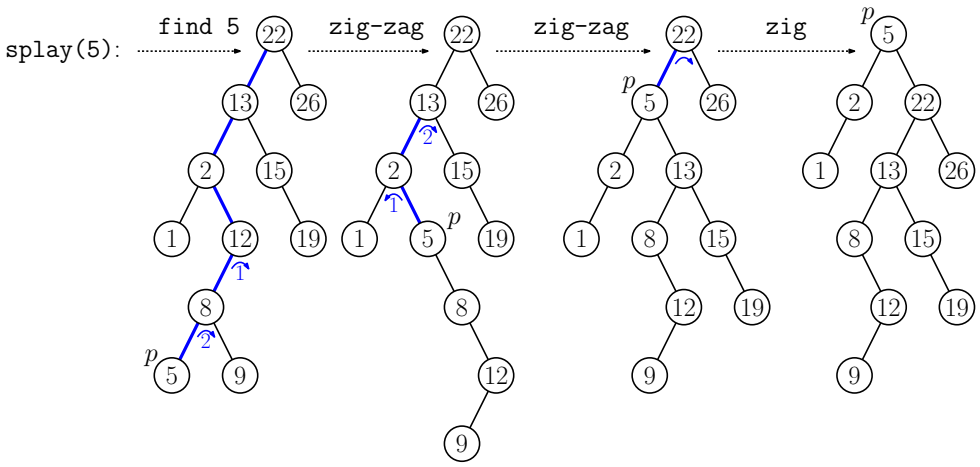
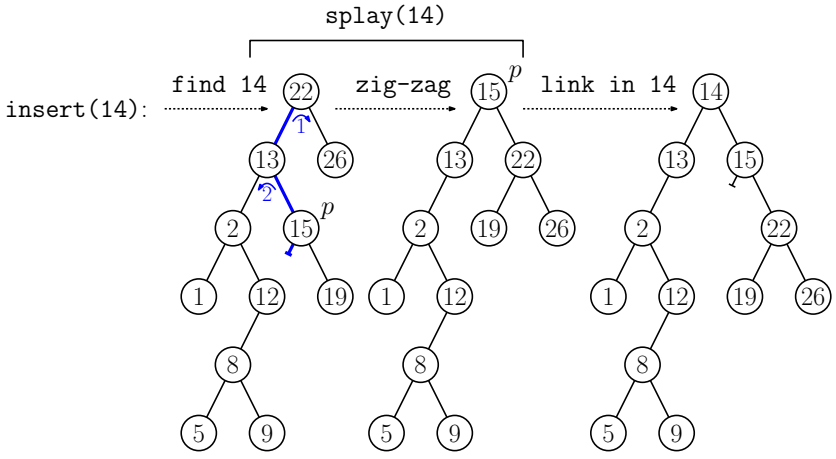
CMSC 420 (0201) - Solutions to Midterm Exam 1

Solution 1:

(a) To perform `splay(5)`, we first find node 5 in the tree (see the figure below). Since it is a left-left grandchild of 12, we perform a zig-zig rotation (involving 12, 8, and 5). This pulls the 5 up. Now, 5 is a left-right grandchild of 13, so we perform a zig-zag rotation (involving 13, 2, and 5), which pulls the 5 up. Now, 5 is the left child of the root 22, we do a zig rotation (involving 22 and 5), which makes 5 the new root.



(b) To perform `insert(14)`, we first perform a find for 14 (see the figure below).



We fall out of the tree at the left child of 15, and so the splay starts with this node. Since 15 is the left-right grandchild of the root, we perform a zig-zag rotation (involving 22, 13, and 15), which pulls 15 up to the root. Now, we create a new root node containing 14, and since $14 < 15$, we make 15 the right child of the root, and 15's left child (13) becomes the left child of 14.

Solution 2:

(a) 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`.

(b) Min: $-(n-1)/2$ Max: $+1$. A key fact is that in any full binary tree, there is one more leaf node than internal nodes. Thus, such a tree always has an odd number of nodes. If the tree has n nodes, then $\lfloor n/2 \rfloor = (n-1)/2$ are internal, and $\lceil n/2 \rceil = (n+1)/2$ are external. As you do an preorder traversal, you are generally seeing more internal nodes than leaves, so the maximum occurs when the traversal is finished, at which point the counter value is $+1$. The smallest value occurs when all the internal nodes come before any external node. This happens when you have a degenerate tree of consisting of a chain of nodes along the left side. In this case, the counter gets as low as $-\lfloor n/2 \rfloor$, before it starts seeing the leave nodes.

(c) $n-k$: 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.

(d) 4 (but we will give partial credit for values 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.

(e) $h+1$: 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 $h+1$.

(f) (2) $1/n$: The smallest key (and indeed any key) is placed at the root when its priority is the smallest among all the keys in treap. Since there are n keys, the probability that any fixed key has the smallest priority is $1/n$ (assuming distinct priorities).

(g) (2) $O(1)$: The skiplist analysis shows that the number of hops that are made at any level of the skiplist is expected to be 2, which is $O(1)$.

(h) h: You can perform one skew at each of the levels of the AA-tree. This happens when all the nodes the nodes are the search path are black-red pairs, and the search path goes into the left child of the red node. Each of these pairs does a skew, then a split, and the split causes the promoted node to become a red-child of its parent. Thus, the skewing and splitting propagates all the way back to the root.

Solution 3: 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.)

```
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)`".

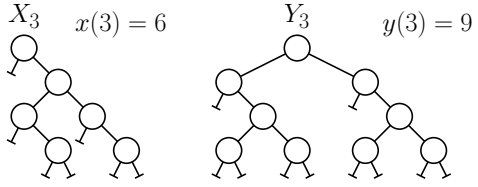
(c) `if (u.left == null) u.left = v`: Again, we still need to swap `v`'s right chain, so it should be "`u.left = swapRight(v)`". In our code, we simply skip this step, since this small optimization is not useful here.

```
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 4:

(a) See the figure below.



(b) The X-tree consists of a single node added to a single Y-tree of the next smaller height, so we have $x(h) = 1 + y(h-1)$. The Y-tree consists of a single node added to two copies of the X-tree of the next smaller height, so we have $y(h) = 1 + 2x(h-1)$.

(c) By plugging the definition of $y(h-1)$ into the definition of $x(h)$, we have $x(h) = 2+2x(h-2)$.

(d) We will prove that $x(h) = 3 \cdot 2^{h/2} - 2$ for all even values of $h \geq 0$. For the basis case ($h = 0$), we have $x(h) = 1$ by definition, and the formula yields $3 \cdot 2^0 - 2 = 3 - 2 = 1$, which matches.

For the induction step, consider a fixed even value of $h \geq 2$. The induction hypothesis states that the formula works for the next smaller even value h, that is, $h-2$, that is, $x(h-2) = 3 \cdot 2^{(h-2)/2} - 2$. We will show that it holds for h itself. (You could also assume it holds for h and prove it holds for $h+2$.)

By (c), we have $x(h) = 2 + 2 \cdot x(h-2)$, which by the induction hypothesis implies that $x(h) = 2 + 2(3 \cdot 2^{(h-2)/2} - 2)$. Simplifying yields,

$$x(h) \;=\; 2 + 2(3 \cdot 2^{(h-2)/2} - 2) \;=\; 2 + 3 \cdot 2 \cdot 2^{(h/2)-1} - 4 \;=\; 2 + 3 \cdot 2^{h/2} - 4 \;=\; 3 \cdot 2^{h/2} - 2,$$

as desired.

Solution 5:

(a) Answer: $m'(1 - 2/w)$. When the expansion takes place, the array is full, meaning that $m = n_1 + n_2$. The new array has size $m' = w \cdot \max(n_1, n_2)$. Subject to the constraint that $m = n_1 + n_2$, $\max(n_1, n_2)$ is minimized when $n_1 = n_2 = m/2$. This implies that $m' \geq wm/2 = (w/2)m$, or equivalently $m \leq (2/w)m'$. The number of new entries created in the expansion is

$$m' - m \;\geq\; m' - \frac{2}{w}m' \;=\; \left(1 - \frac{2}{w}\right) m'.$$

(If you prefer to express this as a function of m and w, we have $m(w/2 - 1)$.)

(b) Answer: $c = 2\frac{w-1}{w-2}$. (This can also be expressed as $c = 1 + \frac{1}{1-2/w}$.) 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 may ignore the first run. Consider a run that is described in part (a), where we start with m' elements of which $m'(1 - 2/w)$ are empty.

Whenever a stack operation is performed, one of the c tokens is used to pay for actual operation, and the remaining $c-1$ tokens are banked. From (a) at least $m'(1-2/w)$ operations will occur before the next expansion, implying that we have collected at least $(c-1)m'(1-2/w)$ excess tokens. The expansion costs us m' units to copy all the elements of the current array. We need to make c large enough to pay for the expansion. Thus, we have the constraint

$$(c-1)m'(1-2/w) \;\geq\; m' \quad\Longleftrightarrow\quad c \;\geq\; 1 + \frac{1}{1-2/w} \;=\; 2\frac{w-1}{w-2}.$$

Thus, setting $c = 2\frac{w-1}{w-2}$ yields the amortized cost. (This proof is admittedly weak, since we have just figured out what the cost needs to satisfy, without proving that it is the actual cost. At this point, we should take this value of c and work through the analysis all over again. But the value of c seems to be selected magically.)

(c) Answer: $w \geq 3$. The amortized cost will be a constant as long as $2\frac{w-1}{w-2}$ is not infinite, which implies that $w > 2$. Since we said that w is an integer, this would imply that $w \geq 3$.