CMSC 420: Fall 2022

Solutions to Midterm 1 Practice Problems

Solution 1:

(a) The number of leaves is exactly $\lceil n/2 \rceil$. 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 $\lceil n/2 \rceil$.

(b) Answer: (i) Definitely full, (iii) Height larger by exactly 1. Explanations: 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) **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) 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 $\texttt{sibling}(i) = i \oplus 1$. Alternatively, we can do this by cases with $\texttt{sibling}(i) = i + 1$ if $i \bmod 2 = 0$ and $i - 1$ otherwise.

(e) Min: 0, Max: $(\lg n) \pm O(1)$. 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 $\lfloor \lg(n+1) \rfloor - 1$.)

(f) Min: $\log_3(2n+1)$, Max: $\lg(n+1)$ (where $\lg \equiv \log_2$).

Given a 2-3 tree with ℓ levels, the number of nodes is minimized when every node is a 2-node. Thus, $n \geq \sum_{i=0}^{\ell-1} 2^i$. By the formula for the geometric series, we have $n \geq 2^\ell - 1$. Now, solving for ℓ, we have $\ell \geq \lg(n+1)$.

Given a 2-3 tree with ℓ levels, the number of nodes is maximized when every node is a 3-node. Thus, $n \leq \sum_{i=0}^{\ell-1} 3^i$. By the formula for the geometric series, we have $n \leq (3^\ell - 1)/2$. Solving for ℓ, we have $\ell \geq \log_3(2n+1)$.

(g) 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.

(h) With standard binary search trees, the expectation was over all $n!$ insertion orders. With treaps, the expectation was over all $n!$ orders of the priority values. The latter is preferred, because the data structure's expected performance is not dependent on the insertion order.

(i) The node storing the *smallest key* x_1 is guaranteed to be black. This is due to the AA-tree constraint that that each red node is the right child of its parent, and hence its key must be larger than its parent.

(j) $n/8$: Recall that in order to reach level i, a node must throw i consecutive heads, which occurs with probability $1/2^i$. Therefore, there are $n/2^i$ such nodes in expectation, which yields $n/8$ for $i = 3$. (Observe that half of these nodes, $n/16$, terminate at this level and the rest continue to higher levels.)

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).

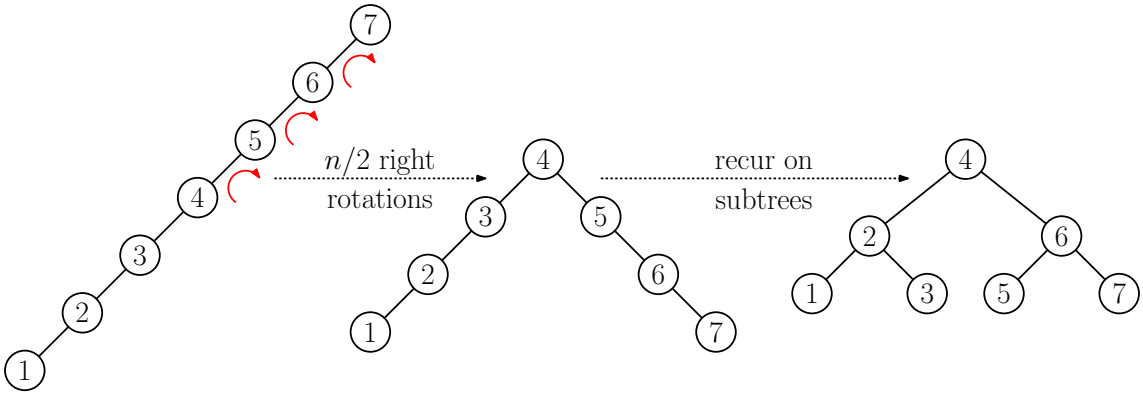

Figure 1: Rotating a tree into balanced form.

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)`.

```
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)
        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`.

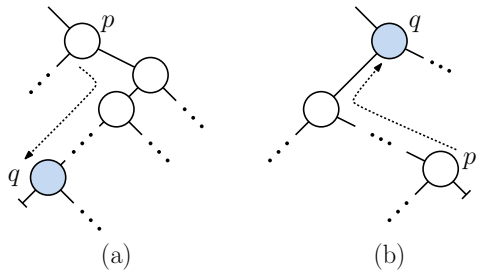

Figure 2: Solution to Problem 4(b): Inorder successor.

```
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 {
        Node q = p.parent                       // follow p's ancestor chain
        while (q != null && p == q.right) {     // until we are a left child
```

```
            p = q
            q = q.parent
          }
          return q
      }
  }
```

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)`.

```
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: 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, $\lfloor \frac{h-2}{2} \rfloor + 1$. 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 7:

(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
        else
          return null                   // no child following p
    }
}
```

(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.)

```
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 8:

(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

$$n(i) \;=\; 2^{\lceil i/2 \rceil} 3^{\lfloor i/2 \rfloor}.$$

(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

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

Solution 9: We will show that the amortized cost is t for some constant t. The expanded array has size γm 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 δm, but note that the size of the array being copied is γm, not m, which increases the copying cost by a factor of γ.) Therefore, we select t so that $(t - 1)(\gamma - 1)m \geq \delta\gamma m$. Setting $t = 1 + \delta\gamma/(\gamma - 1)$ satisfies this.

Solution 10: To expose a node, we first apply a standard descent to find the exposed node, and we set the priority to $-\infty$ (actually `Integer.MIN_VALUE`). We then walk back up the search path to the root. As we return from a call to expose, the exposed node has replaced the child. Thus, if we apply `expose` to the left subtree, a single right rotation suffices to move it to the current node. We then return this value. (The right side is symmetrical.)

```
TreapNode expose(Key x, TreapNode p) {
    if (p == null)                          // error - key not in tree
        throw Exception("Key not found")
    else if (x < p.key) {                   // x is smaller - search left
        p.left = expose(x, p.left)
        return rotateRight(p)               // rotate the exposed node up
    }
    else if (x > p.key) {                   // x is larger - search right
        p.right = expose(x, p.right)
        return rotateLeft(p)                // rotate the exposed node up
    }
    else {                                  // found it
        p.priority = Integer.MIN_VALUE      // set priority to -infinity
        return p
    }
}
```

Solution 11: The algorithm operates essentially the same as the find operation for skip lists. The main difference is that, whenever we follow a link, we count the number of elements that

the link spans. The other modification is that, rather than advancing when `p.next[i].key <= x`, we instead use `p.next[i].key < x`. (Alternatively, you could leave the condition the same, but decrement the count by 1 if the key is found.)

```
int countSmaller(Key x) {
    int i = topmostLevel           // start at topmost nonempty level
    SkipNode p = head              // start at head node
    int count = 0                  // number of smaller elements
    while (i >= 0) {               // while levels remain
        if (p.next[i].key < x) {
            count += p.span[i]     // count number of skipped items
            p = p.next[i]          // advance along same level
        }
        else i--                   // drop down a level
    }
    return count                   // return final count
}
```

Solution 12: This is true, as shown in the following theorem.

Theorem: Given a splay tree T_0 and any two keys $x, y \in T$, the trees T_1 resulting from `splay(x); splay(y)` and T_2 resulting from (`splay(x); splay(y)`)^2 are identical.

Proof: We may assume that $x < y$, since the other case is left-right symmetrical. (This is because all the splay operations are left-right symmetrical.) We assert that, after performing `splay(x); splay(y)`, T_1 has one of two possible structures:

(a) The root node is y and its left child is x (see Fig. 3(a)).

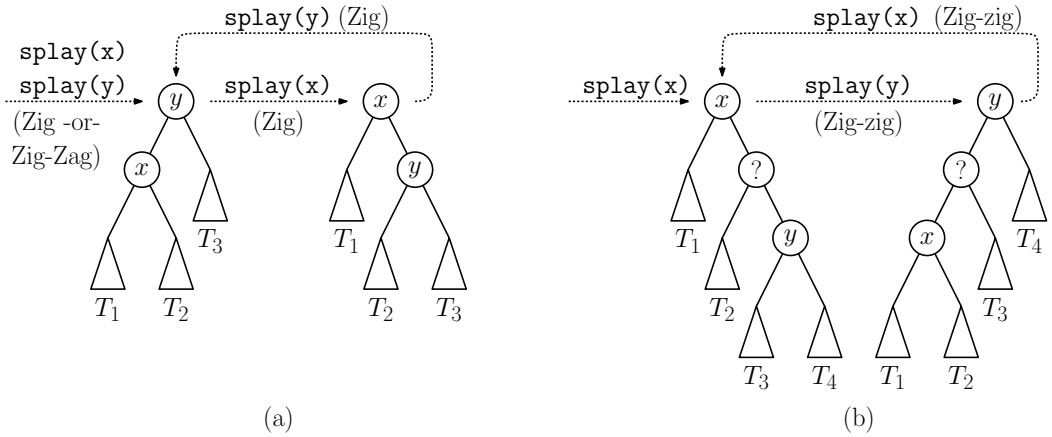

Figure 3: Repeated splaying.

(b) The root node is y and its left-left grandchild is x (see Fig. 3(b)).

To see this, observe that after `splay(x)`, x is at the root of the tree. The final rotation of `splay(y)` is either Zig (implying that x is now the left child of y), Zig-Zig (implying that y

was the right-right grandchild and now x is its left-left grandchild), or Zig-Zag (implying that y was x's right-left grandchild, and now x is y's left child).

In case (a), the next `splay(x); splay(y)` will right rotate and then left rotate the root (see Fig. 3(a)), which leaves the tree unchanged. In case (b), the next `splay(x); splay(y)` will Zig-Zig x back up to the root and then Zig-Zig y back up to the root (see Fig. 3(a)). In either case, we wind up back where we started.

Here is an interesting question, which is suggested by the above problem. Consider any sequence of distinct $k \geq 1$ keys, $\langle x_1, \ldots, x_k \rangle$ in a splay tree T. Are the trees resulting from $(\texttt{splay}(x_1) \ldots \texttt{splay}(x_k))$ and $(\texttt{splay}(x_1) \ldots \texttt{splay}(x_k))^2$ always the same? (Honestly, I don't know the answer.)