Solutions to Homework 2: Search Trees

Solution 1: See Fig. 1. In the upper left, we show the balance factors of the nodes in the original tree. When key 19 is deleted, the ancestor with key 18 has a balanced factor of -2. It is heaviest on its left-right side, which implies that we perform a left-right double rotation (left at 15 and right at 18). Following this, the root with 13 has a balance factor of -2. Since it is heaviest on its left-left side, we do a single right rotation at the root, which leads to the final tree in the lower left. Now all the balance factors are good.

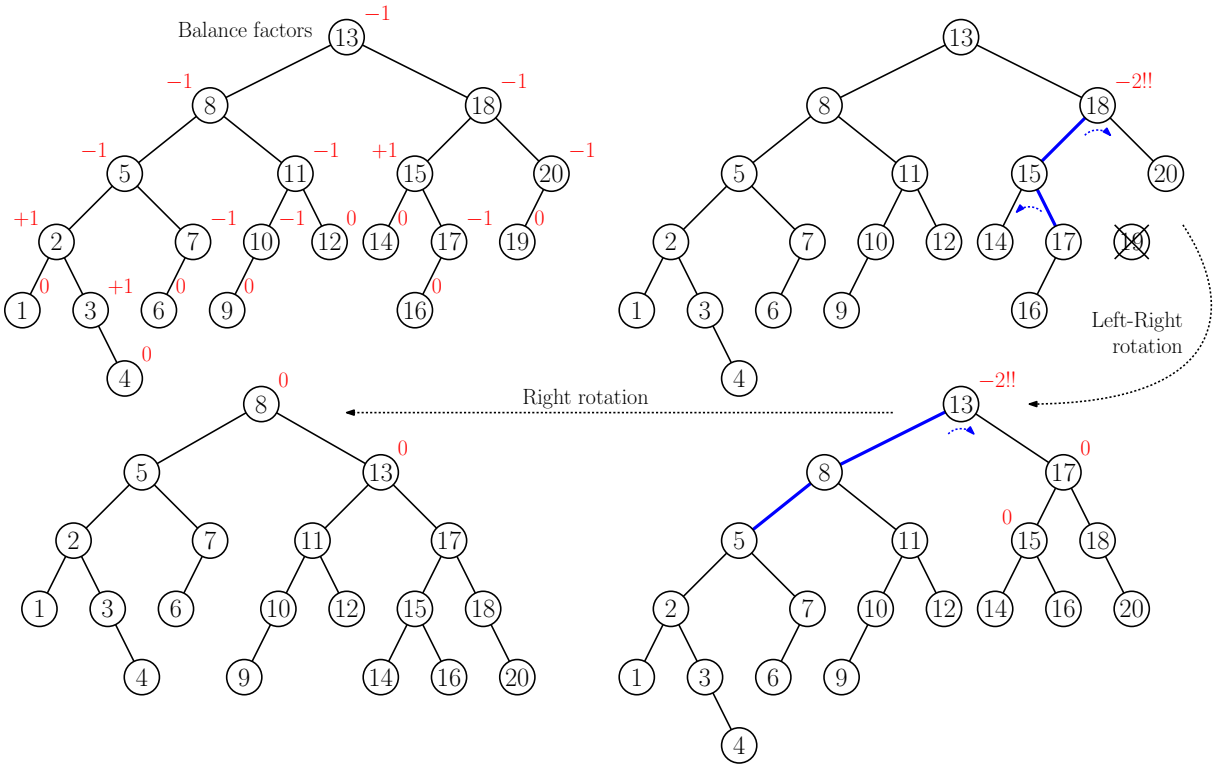

Figure 1: AVL balance factors and deletion.

Solution 2:

(a) See Fig. 2. First, we perform a find on 12 until we fall out of the tree at 11's right child. We insert 12 here and assign it to level 1. We then start walking back towards the root. At each node we perform a skew and a split. Many have no effect, but the following do:

11: (no change)

10: (split) has a right-right red grandchild (12), and its split rotates 11 to become the parent of 10 and 12, and 11 moves up to level 2.

9: (no change)

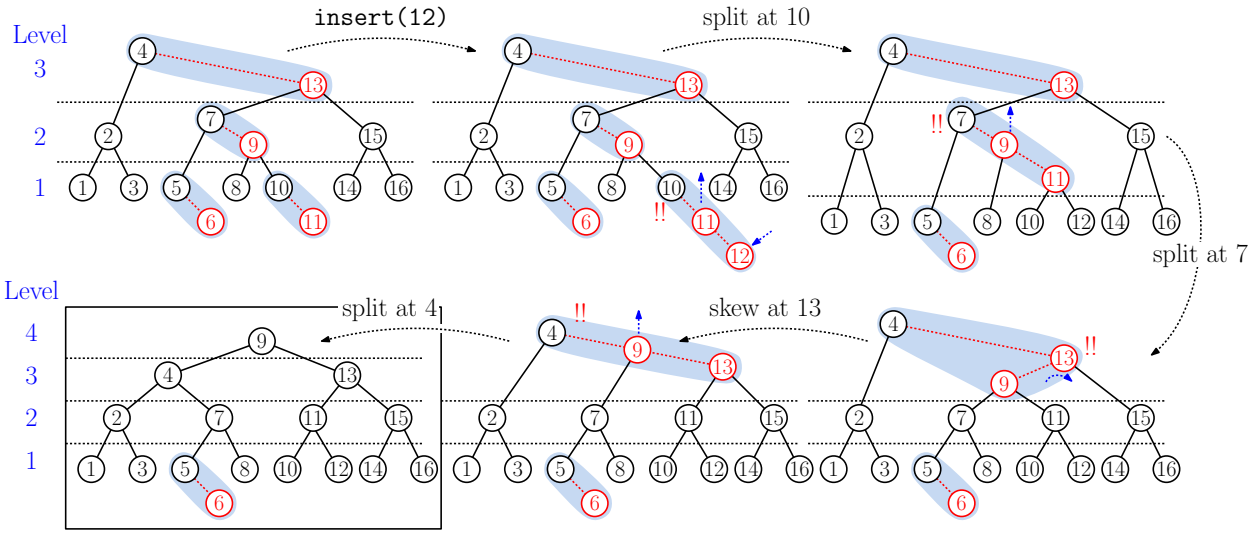

Figure 2: AA-tree insertion.

7: (split) has a right-right red grandchild (11), and its split rotates 9 to become the parent of 7 and 11, and 9 moves up to level 3.

13: (skew) has a left red child (9), and its skew rotates 9 to become its parent.

4: (split) has a right-right red grandchild (13), and its split rotates 9 to become the parent of 4 and 13, and 9 moves up to level 4.

root: 9 becomes the new root of the tree.

(b) See Fig. 3. We find 3 and delete this node. We walk back towards the root, and perform the various operations required by fix-after-delete. These involve update-level, three skews, and two splits. Most of these have no effect, but the following do:

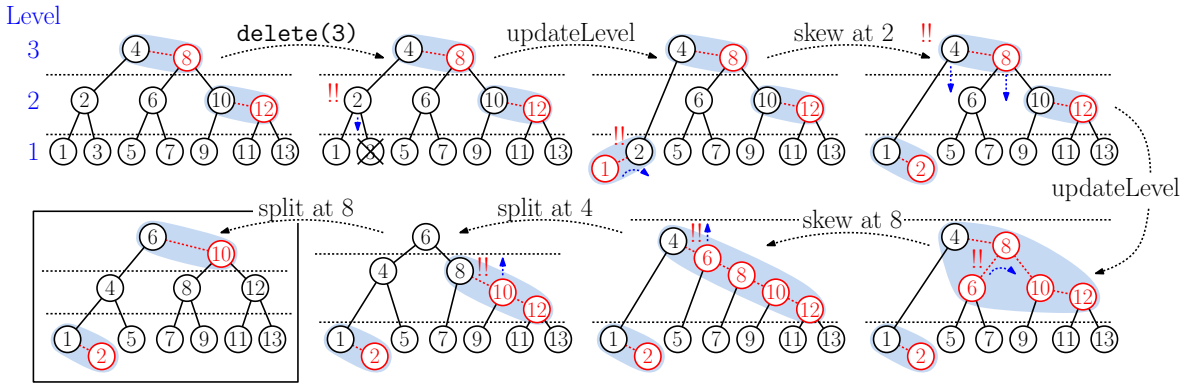

Figure 3: AA-tree deletion.

2: (update-level) When 3 is deleted, 2's right child points to `nil`, which is at level 0. This causes 2 to be demoted to level 1.

(skew) 2 has a left red child (1), and its skew rotates 1 to become its parent.

4: (update-level) Because 2 is demoted, 4's left child (1) is at level 1. This causes 4 to be demoted to level 2. Because 4's right child (8) is at the same level, it also moves down.

(skew) 8 has a left red child (6), and its skew rotates 6 to become its parent.

(split) 4 has a right-right red grandchild (8), and its split rotates 6 to become the parent of 4 and 8, and 6 moves up to level 3.

(split) 8 has a right-right red grandchild (12), and its split rotates 10 to become the parent of 8 and 12, and 10 moves up to level 3.

root: 6 becomes the new root of the tree.

Solution 3: See Fig. 4(a)–(d).

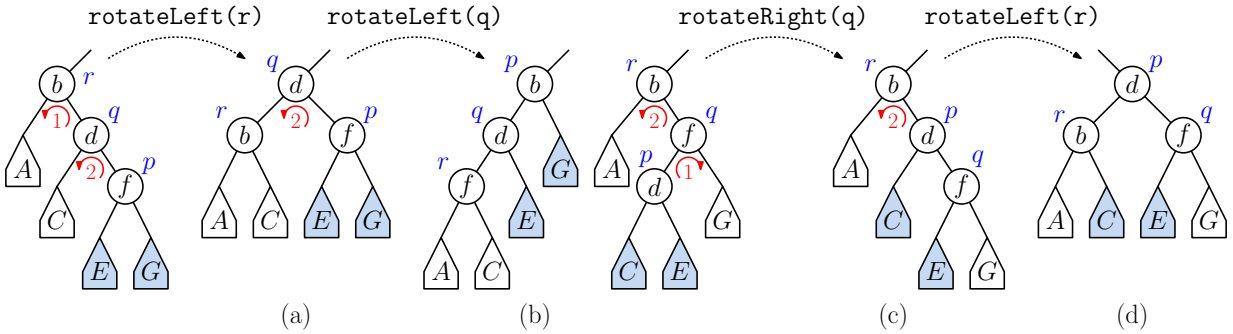

Figure 4: Solution to Problem 3: Splay rotations.

Solution 4:

(a) We employ a utility function `size`, which returns the size of a node `p`, returning the value zero if `p` is `null`. The rotate code is the same as usual, but we update the sizes of `p` and `q` as the sum of their left and right children plus one (for the node itself).

```
int size(Node p) { return (p == null ? 0 : p.size); }

Node rotateRight(Node p) {
    Node q = p.left;
    p.left = q.right;
    q.right = p;
    p.size = size(p.left) + 1 + size(p.right);
    q.size = size(q.left) + 1 + size(q.right);
    return q; // Not required by the problem statement (but should be here)
}
```

(b) The initial call is `countSmaller(x, root)`. If `x` is less than or equal to `p.key`, then we may ignore this node and its right subtree, since they are all too large, and simply recurse on `p`'s right subtree. Otherwise, we know that `p` and all the keys in its left subtree are strictly smaller than `x`, and so we include them in the count (`size(p.left) + 1`) and then we recurse on `p`'s right subtree to complete the count. (We employ the `size` utility from (a).)

```
void countSmaller(Key x, Node p) {
    if (p == null) return 0;
    else if (x <= p.key) return countSmaller(x, p.left);
    else return size(p.left) + 1 + countSmaller(x, p.right);
}
```

(c) The initial call is `getMinK(k, root)`. If the size of the current node's left subtree is at least k, we know that the smallest k nodes lie in the left subtree, and so we recurse on it. If the size of the left subtree is $k-1$, the kth smallest key is stored in `p`'s node, and we return its value. Otherwise, we know that the kth smallest key is in `p`'s right subtree. We recurse on this subtree, but first we decrease the value of k by `size(p.left) + 1` to compensate for the elements we have skipped over.

```
Value getMinK(int k, Node p) {
    if (p == null) return null;
    else if (size(p.left) >= k) return getMinK(k, p.left);
    else if (size(p.left) == k-1) return p.value;
    else return getMinK(k - (size(p.left) + 1), p.right);
}
```

Clearly, (a) runs in $O(1)$ time and (b) and (c) run in time proportional to the tree's height, since they simply follow a single path until falling out of the tree.

Solution 5: 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. 5(a)).

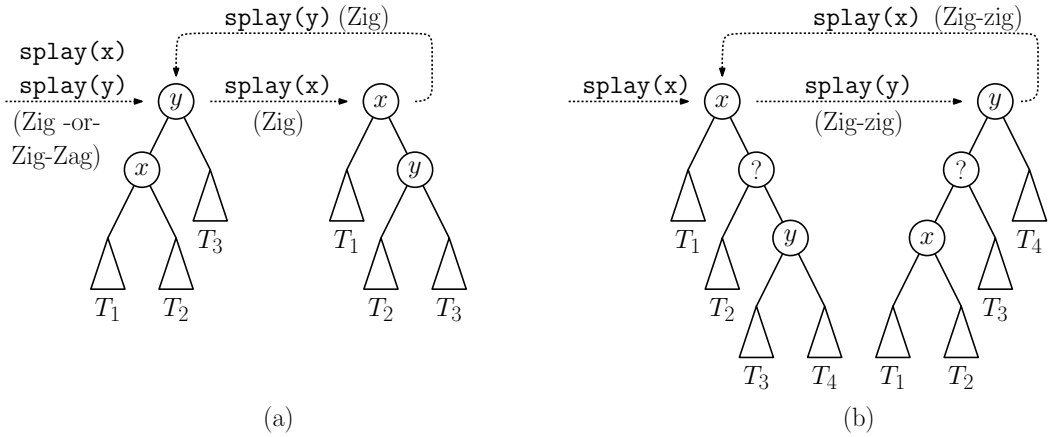

Figure 5: Solution to Problem 3: Repeated splaying.

(b) The root node is y and its left-left grandchild is x (see Fig. 5(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. 5(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. 5(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?

Solution 6:

(a) As in the standard search, we never want to overshoot the key, we move forward only if the key in the next node is $\leq y$ (`p.next[i].key <= y`). In the standard search, the level index decreases monotonically. The difference here is that, whenever possible, we will move up a level. We only move down when needed. This implies that the search moves forward as fast as possible. The code is very similar to the standard find function, except that it starts at level zero, and we include the "move up" option.

```
Value findForward(SkipNode p, Key y) {
    int i = 0                              // start at the lowest level
    while (i >= 0) {                       // (level will rise then fall)
        if (p.next[i].key <= y) {          // can we move forward?
            if (i+1 < p.next.length) i++   // move up, if possible
            else p = p.next[i]             // move horizontal, if not
        else i--                           // drop down a level
    }
    return (p.key == y ? p.value : null)   // return value if found
}
```

While it is not exactly apparent from the code, the algorithm has two phases, an *up-phase* and a *down-phase* (see Fig. 6). In the up-phase, whenever we have the opportunity to move up a level we do so. This phase continues until we reach a node `p` and level i such that the next link at this level takes us beyond y, that is, `p.next[i].key > y`. After this, we begin the down-phase, which operates in the same manner as the standard skip-list search.

(b) In class, we showed that, given a set of n nodes in a skip list, with high probability the maximum level of any node will be $O(\log n)$. This did not depend on any properties of the keys stored in these nodes nor the global properties of the data structure, since each node of a skip list determines its height independent of all other nodes. Therefore, the analysis can be applied to *any* subset of skip-list nodes, in particular, the k nodes lying between p and y.

(c) We will show that the algorithm visits a constant number of nodes at each level. In particular, in expectation, it visits two nodes on each level during the up-phase, and two nodes at each level during the down phase.

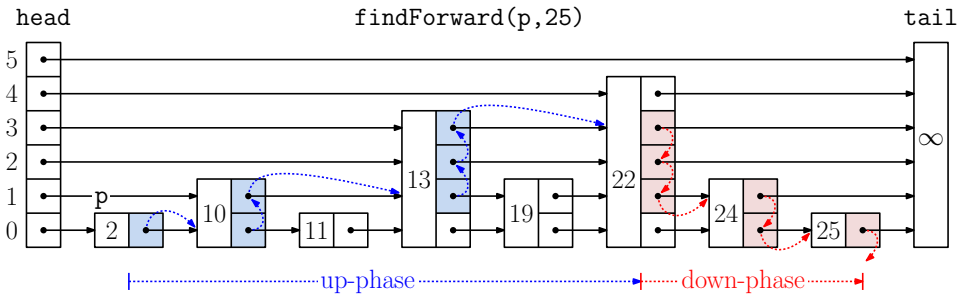

Figure 6: Solution to Problem 6: Find-forward in a skip list.

Up-Phase: During the up-phase, whenever we arrive at a node, we move up to the next higher level if the node contributes to this higher level. There is a $1/2$ probability of this event occurring. Therefore, the expected number of nodes visited at this level is exactly the same as the expected number of coin tosses made until first seeing heads, which is 2.

Down-Phase: Because the down-phase is the same as the standard search process in skip lists, the analysis of the down-phase is the same as the one given in class (based on a backwards analysis), which shows that the expected number of nodes visited per level is 2.

By (b), the maximum level that we will visit in expectation is $O(\log k)$, and by (c), the number of hops at any level is $2 + 2 = 4$, so therefore, the total time for the search is $O(4 \log k) = O(\log k)$, as desired.

Solution to the Challenge Problem:

(a) Starting at the root, the algorithm tosses a coin and recurses on the left with probability $1/2$ and recurses on the right with probability $1/2$. When an external node is hit, the algorithm returns a pointer to this node.

We assert that this algorithm runs in expected time $O(\log n)$, where n is the number of nodes in the tree. To see this, let $E(n)$ denote the expected-case running time of the algorithm on a tree with n external nodes. If $n = 1$, the algorithm returns after visiting this node, so $E(1) = 1$. Otherwise, one of the two subtrees contains at most $n/2$ of the external nodes. With probability $1/2$, we recurse on this (smaller) subtree, and with probability $1/2$, we recurse on other (larger) subtree. Whenever we recurse on the smaller subtree, we reduce the remaining number of external nodes by at least half. Otherwise, we may only reduce this by one. So, the algorithm's expected running time satisfies the following recurrence:

$$E(n) \;\leq\; 1 + \frac{1}{2}E\left(\frac{n}{2}\right) + \frac{1}{2}E(n-1).$$

Clearly, the expected running time increases monotonically with n, and so to simplify matters,

let's replace $E(n-1)$ with $E(n)$ above. This yields

$$E(n) \;\le\; \frac{1}{2}E\left(\frac{n}{2}\right) + \frac{1}{2}E(n) \;\Rightarrow\; \frac{1}{2}E(n) \;\le\; 1 + \frac{1}{2}E\left(\frac{n}{2}\right)$$
$$\Rightarrow\; E(n) \;\le\; 2 + E\left(\frac{n}{2}\right).$$

By expanding this, we have $E(n) \le 4E(n/4) \le 6E(n/8) \le \cdots \le 2iE(n/2^i)$. By setting $i = \lg n$, we have $E(n) = (2\lg n)E(1) = 2\lg n = O(\log n)$. Therefore, the expected number of nodes visited is $O(\log n)$.

(b) Given any deterministic algorithm A, let us imagine that we run the algorithm on an infinite binary tree consisting of only internal nodes, until it visits $(n-1)/2$ nodes of this tree. Because there are no internal nodes, the algorithm will run until it encounters this many nodes. Let $T_A(n)$ denote the subtree of nodes that were so visited (shaded nodes in Fig. 7(a)).

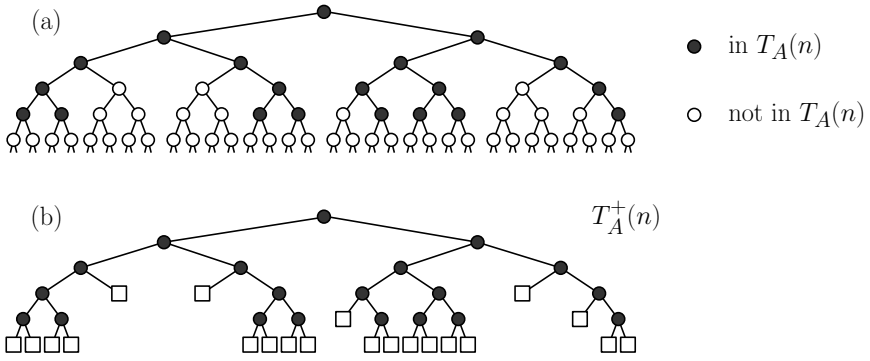

Figure 7: Solution to Challenge Problem (b).

Now, we define an extended binary tree, $T_A^+(n)$, as follows. We take the nodes of $T_A(n)$ and add external nodes to each of the unvisited left- and right-child links (shaded in Fig. 7(b)). We know that an extended binary tree with m internal nodes has $m+1$ external nodes. Thus, $T_A^+(n)$ has a total of $(n-1)/2 + ((n-1)/2 + 1) = n$ nodes.

Claim: When our deterministic algorithm is run with the input $T_A^+(n)$, it will visit $(n-1)/2$ internal nodes before encountering its first external node.

Proof: When we visit internal nodes, the two trees $T_A(n)$ and $T_A^+(n)$ appear identical. Therefore, the behavior of algorithm A on these two trees will be identical for the first $(n-1)/2$ nodes visited. (After this, no matter what node it visits, it will encounter an external node.)

In conclusion, this deterministic algorithm's running time is at least $\Omega(n)$.