Solutions to the Midterm Practice Problems

Disclaimer: These solutions have not been carefully checked. If anything seems to be fishy, please check with me.

Solution 1:

(a) 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) **True**: External and internal nodes alternate in an inorder traversal. This can be proved by induction. Observe that in the inorder traversal of any extended binary tree, the first and last nodes visited must be external. So, by induction, the nodes of the left subtree alternate (ending in an external node), then the root is visited (internal), and then the nodes of the right subtree alternate (starting with an external node).

(c) **True**: There is always an external node at depth at most $d = \lceil \lg n \rceil$. If this were not true, then the first d levels would all be internal. It follows that the number of external nodes must be at least $2^{d+1} > 2^{\lg n} = n$, contradicting the hypothesis that there are n external nodes.

(d) Given a 2-3 tree with ℓ levels, there are at least $n_{\min}(\ell) = \sum_{i=0}^{\ell-1} 2^i$ nodes and at most $n_{\max}(\ell) = \sum_{i=0}^{\ell-1} 3^i$ nodes. By the formula for the geometric series, we have $n_{\min}(\ell) = 2^\ell - 1$ and $n_{\max}(\ell) = (3^\ell - 1)/2$. Solving for ℓ in each case, we have $\ell = \log_2(n_{\min}(\ell) + 1)$ and $\ell = \log_3(2n_{\max}(\ell) + 1)$. Thus, the number of levels ℓ is:

$$\log_3(2n+1) \;\leq\; \ell \;\leq\; \log_2(n+1).$$

(e) It was observed in class that in the insertion process, an AVL tree may perform either a single rotation or a double-rotation. After this, the subtree height is the same as in the original tree, so no further rotations are needed. Thus, the number of rotations following an insertion is *at most two*.

(f) It was observed in class that deletions from the AVL may propagate up to the root. Thus, the number of rotations can be proportional to the height, or $O(\log n)$.

(g) Min: 0, Max: $h+1$. A 2-3 tree of height h yields an equivalent AA tree with $h+1$ levels. Each node at a given level may give rise to a single red node (if the path goes through a 3-node) or not (if the path goes through a 2-node).

(h) In a skip list, nodes of variable sizes are allocated, because once allocated, the number of pointers in the node does not change. In contrast, in a B-tree, the number of keys stored in a node can vary as keys are inserted and deleted, and thus we always allocate nodes of the maximum possible size.

(i) 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 under the influence of the access distribution.

(j) A *finger search* means that, rather than starting the search at the root node, it starts from a given position in the search structure, for example, from the location of the most recently accessed node. Finger searches are important when a sequence of queries are being performed, and each query object is expected to be close to its predecessor in the sequence.

(k) We assert that node y is at maximum depth 2 in the resulting splay tree. Consider the last splay rotation (zig, zig-zag, or zig-zig) just before z was brought to the root. At this point x is already at the root. Since x, y, and z are consecutive in order, y is either the right child of x (zig-zig case) or y is the left child of z (in the zig or zig-zag cases). In the first case y becomes the left child of z (depth 1) and in the other case x becomes the left child of z and y becomes the right child of x (depth 2).

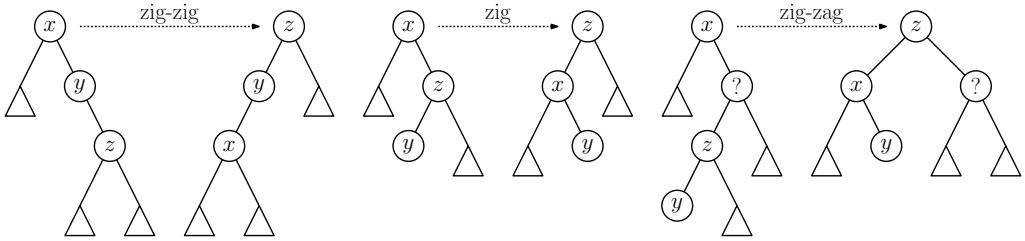

Figure 1: Depth of a node sandwiched between splays.

Solution 2:

(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 $\lfloor n/2 \rfloor$ nodes. If we start with a left chain and do $\lfloor n/2 \rfloor$ 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. We can rebalance each of these subtrees recursively (but reversing left and right on the right subtree).

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
}
```

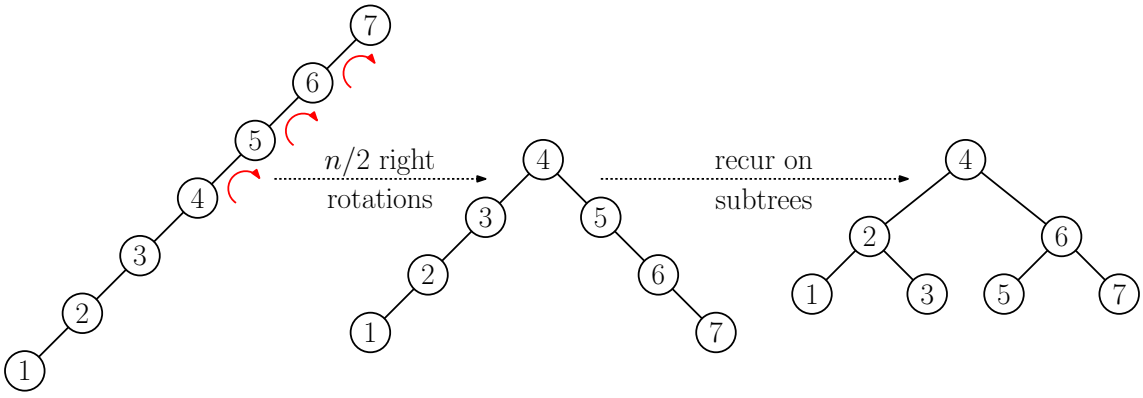

Figure 2: Rotating a tree into balanced form.

(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 3:

(a) The insertion code is similar to that of a standard binary search tree, but since we need access to the node's parent, we have two arguments, the current node `p`, and its parent `par`. To insert a node we begin with the usual descent used by the standard insertion algorithm. When we fall out of the tree, there are two cases. If we fall out on a left child link, then the newly created node's inorder predecessor is its parent's inorder predecessor (`par.left`) and its inorder successor is its parent (`par`). (See Fig. 3(a).) If we fall out on a right child link, then the newly created node's inorder successor is its parent's inorder successor (`par.right`) and its inorder predecessor is its parent (`par`).

We assume that the `BinaryNode` constructor is given four arguments: the key, the value, and the two threads. It sets both thread indicators to `true`.

```
BinaryNode insert(Key x, Value v, BinaryNode p, BinaryNode par) {
    if (p == null) {                          // fell out of tree
        if (par == null)                      // new node is the root
            p = new BinaryNode(x, v, null, null);
        else if (x < par.data)                // new leaf on left
            p = new BinaryNode(x, v, par.left, par);
```

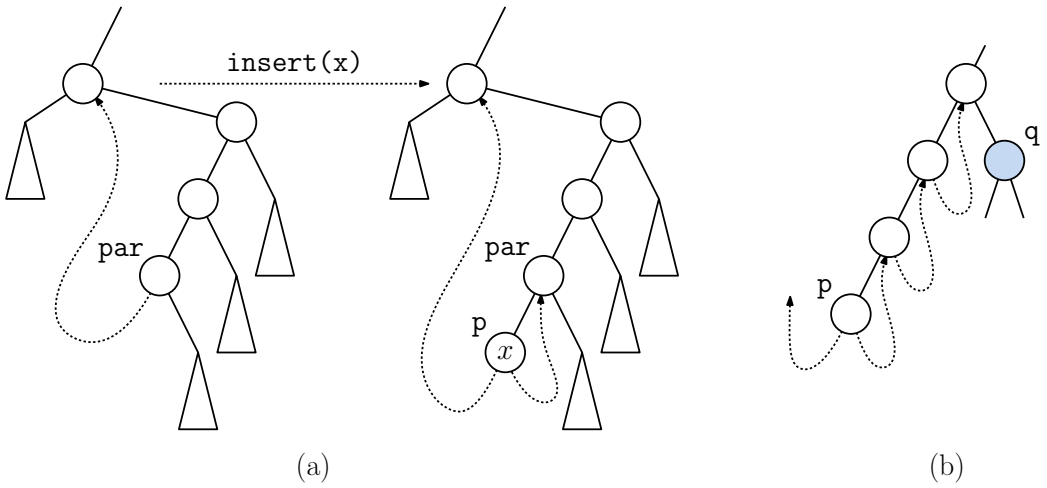

Figure 3: Insertion into a threaded binary tree

```
            else if (x > par.data)              // new leaf on right
                p = new BinaryNode(x, v, par, par.right);
        }
        else if (x < p.data) {                  // insert in left subtree
            p.left = insert(x, p.left, p);
            p.isLeftThread = false;
        }
        else if (x > p.data) {                  // insert in right subtree
            p.right = insert(x, p.right, p);
            p.isRightThread = false;
        }
        else throw DuplicateKeyException;
        return p;
    }
```

(b) If `p` has a left child, then its preorder successor is this child. Otherwise, if it has a right child, then the preorder successor is this right child. If it has neither (that is, this node is a leaf), we follow right threads until reaching the first node whose right-child link is not a thread (see Fig. 3(b)). The right child of this node is the preorder successor. If this chain ends in a null pointer, then we return `null` (since there is no preorder successor). To start the process, the initial node is the root.

```
    BinaryNode nextPreorder(BinaryNode p) {     // preorder successor of p
        if (!p.leftIsThread)                    // has a left child?
            return p.left;                      // ...return this
        else {                                  // no left child
            BinaryNode q = p;                   // start here and
            do {                                // ...follow right threads
                boolean isThread = q.rightIsThread;
                q = q.right;
            } while (q != null && isThread)     // until null or child
            return q;                           // return the result
```

```
        }
    }
```

Solution 4: There are a number of cases to consider. First, if `p` is the root, it has no predecessor. Otherwise, if `p` is a left child, then its preorder predecessor is its parent (see Fig. 4(a)). If `p` is a right child, there are two cases. If its parent has no left child, then its preorder predecessor is its parent (see Fig. 4(b)).

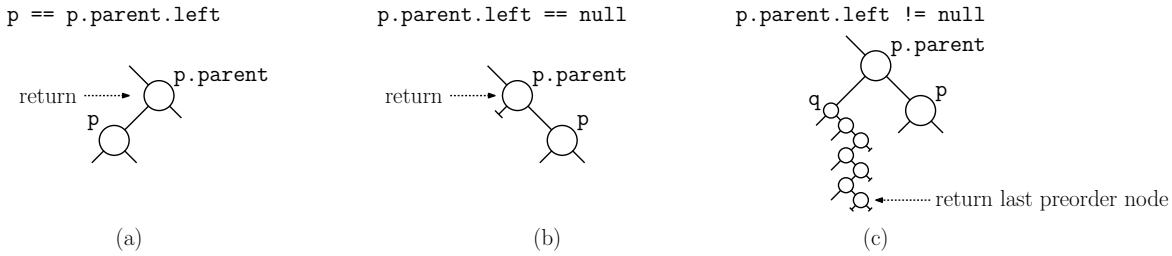

Figure 4: Cases arising in computing the preorder predecessor.

Otherwise, p's parent has a left child. Let `q` be this child (see Fig. 4(c)). The desired node is the *last preorder node* in `q`'s subtree. Computing this correctly takes a bit of thought. The key observation is that such a node must be a leaf (since an internal node comes earlier in preorder than either of its children). If a node has a single right child, the last preorder node comes from this child. If it has just a left child, it will come from there. We will give a recursive function to implement this (see the function `preorderLast` in the code block below).

Preorder Predecessor with Parent Links

```
Node preorderPred(Node p) {              // p's preorder predecessor
    if (p.parent == null)                // p is the root?
        return null;                     // ...no predecessor
    else if (p == p.parent.left)         // p is a left child?
        return p.parent;                 // ...parent is predecessor
    else {                               // p must be a right child
        if (p.parent.left == null)       // no left sibling?
            return p.parent;             // ...parent is predecessor
        else {
            return preorderLast(p.parent.left); // preorder last of parent's left
    }
}

Node preorderLast(Node q) {              // preorder last in q's subtree
    if (q.right != null)                 // right subtree is non-empty?
        return preorderLast(q.right);    // ...look for it here
    else if (q.left != null)             // left subtree is non-empty?
        return preorderLast(q.left);     // ...look for it here
    else                                 // arrived at a leaf
        return q;                        // ...this is it!
}
```

Solution 5: In the process of doing the rotation, in addition to `p` and `q`, the following nodes are affected: `s = p.sibling`, `r = q.right`, and `q.left` and `p.right` (see Fig. **??**). After doing the rotation itself, we make `q` and `s` siblings, we make `p` and `q.left` siblings, and we make `r` and `p.right` siblings. Except for `p` and `q`, any of these could be `null`.

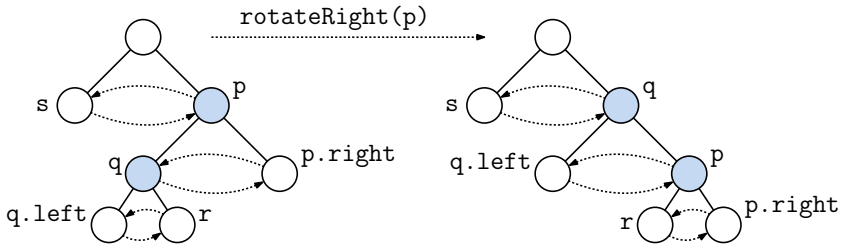

Figure 5: Rotation with sibling pointers.

```
BinaryNode rotateRight(BinaryNode p) {  // right rotation at p
    BinaryNode q = p.left;
    s = p.sibling;                      // p's old sibling
    r = q.right;                        // q's old right child

    p.left = q.right;                   // do the rotation
    q.right = p;

    makeSiblings(s, q);                 // make s and q siblings
    makeSiblings(q.left, p);            // make q.left and p siblings
    makeSiblings(r, p.right);           // make r and p.right siblings
    return q;
}

void makeSiblings(BinaryNode p, BinaryNode q) {
    if (p != null) p.sibling = q;
    if (q != null) q.sibling = p;
}
```

Solution 6:

(a) In a zig-zag tree, all rotations are zig-zag rotations. The result (with all intermediate trees) is shown in Fig. 6.

(b) In looking at the figure, it is evident that the nodes whose original level was odd (`a`, `b`, etc.) are mapped to depth $(k+1)/2$, and nodes whose original level was even (`i`, `h`, etc.) are mapped to depth $1 + k/2$. (We ignore the splayed node itself, which is mapped to depth 0.) In summary, we have

$$\text{depth}(k) \;=\; \begin{cases} (k+1)/2 & \text{if } k \text{ is odd} \\ 1 + k/2 & \text{if } k \text{ is even.} \end{cases}$$

This provides some intuition for why splaying is good. The depth of every node along the search path decreases by roughly half.

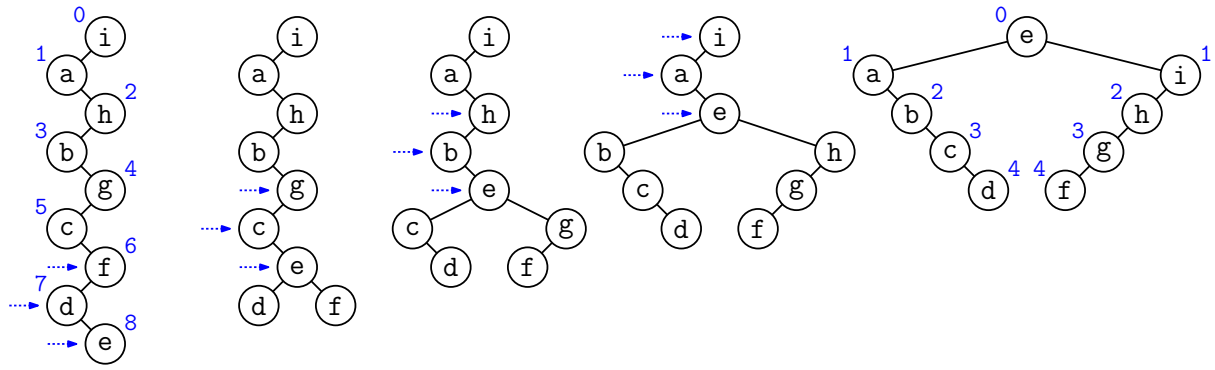

Figure 6: Splaying the deepest node in a zig-zag tree.

(c) To prove the correctness of the above formula, observe that whenever three nodes are involved in a zig-zag rotation, the depth of the topmost node (which is an even-level node) increases by one, and the depth of its child (which is an odd-level node) remains unchanged. Rotations below these nodes do not affect their depth. By the nature of zig-zag rotations, whenever such a rotation takes place above these nodes, the depth of the subtree containing them decreases by one. Thus, if these nodes are at levels $k' = 2\ell$ and $k'' = 2\ell + 1$, there are ℓ such rotations that occur above them.

Therefore, the final depth of the upper (even level) node is its original depth (2ℓ), plus 1 (when it is rotated), and minus 1 for each subsequent zig-zag rotation above it ($-\ell$):

$$(2\ell) + 1 - \ell \;=\; 1 + \frac{\ell}{2} \;=\; 1 + \frac{k'}{2} \;=\; \text{depth}(k').$$

Similarly, the final depth of the lower (odd level) node is

$$(2\ell + 1) + 0 - \ell \;=\; 1 + \ell \;=\; \frac{(2\ell + 1) + 1}{2} \;=\; \frac{k'' + 1}{2} \;=\; \text{depth}(k''),$$

as desired.

Solution 7:

(a) We start with n nodes at level 0, on average pn nodes survive to level 1, $p^2 n$ survive to level 2, and in general we expect $p^i n$ to survive to level i.

(b) Let h denote the number of levels in the skip list. (We actually don't care what this value is.) Summing the number of nodes that contribute to each level of the skip list, and employing the fact that, for $0 < c < 1$, $\sum_{i=0}^{\infty} c^i = 1/(1-c)$, the total expected number of links is

$$\sum_{i=0}^{h} p^i n \;\leq\; n \sum_{i=0}^{\infty} p^i \;=\; \frac{n}{1-p};$$

Observe that for any constant p, this is $O(n)$.

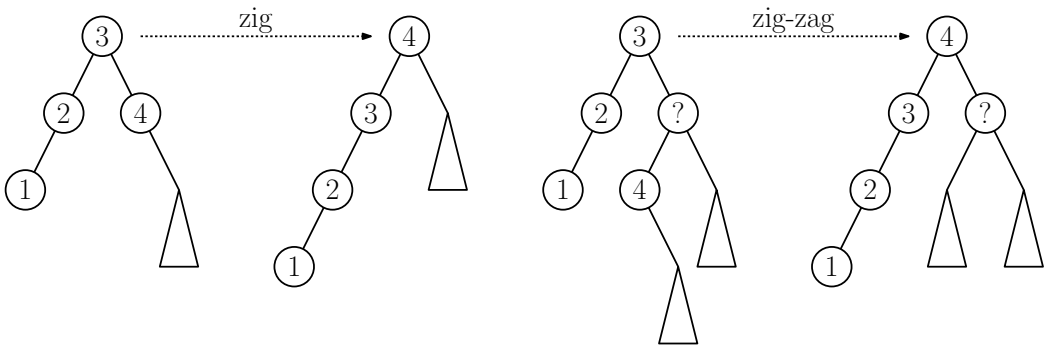

Figure 7: Splaying keys in order.

Solution 8: Let $x[1] < \cdots < x[n]$ denote the keys being stored in the splay tree. We will show that for $1 \leq i \leq n$, that after the call to `splay(x[i])`, the tree will have x_i as the root, and keys $x_1, \ldots, x_i$ will form a left chain (see Fig 7.)

Assuming by induction that this is true after the first $i-1$ splays, let's consider the result of `splay(x[i])`. At this point $x[i]$ is the smallest key in the right subtree of the root. The last rotation of the operation is either zig, zig-zig, or zig-zag. We assert that it cannot be zig-zig, since this would mean that there is a key smaller than $x[i]$ in the right subtree. The remaining two cases are illustrated in Fig. 7. It is easy to see that, in either case, $x[i]$ (shown as 4 in the figure) is rotated to the root, and the previous keys form a left chain beneath it.

Solution 9: It is tempting to maintain a single variable that stores the current minimum, but when the minimum element is popped off the stack, we would need to find a new minimum, which would take $O(n)$ time. (Note that if elements are pushed in random order, then the probability that the minimum is at the top of the stack of size n is $1/n$, and hence the expected amortized time would be $(1/n)n = 1$. But the assumption of randomness is critical for this to work.)

Our solution is to maintain two parallel stacks. The first, `stack`, stores the standard stack contents. The second, `min`, maintains the invariant that its top element is the minimum among all the items in the stack. Letting `top` denote the index of the stack's top, `getMin` simply returns `min[top]`. Whenever a new element is pushed, we update `stack` in the standard manner, and we push on `min` the minimum of the new item or the previous min. To pop the stack, we simply pop both stacks simultaneously. The pseudo-code appears in the following code block.

MinStack - A Stack with `getMin()` method

```
class MinStack {
    int top;                        // top of stack
    int stack[];                    // stack contents
    int min[];                      // minimum in stack

    MinStack(int n) {               // constructor
        stack = new int[n];
        min = new int[n];
        top = -1;
    }
                                    // operations (no error checking)
    boolean isEmpty( return top == -1; }
    int pop() { return(stack[top--]); }
    int getMin() { return(min[top]); }

    void push(int x) {              // push x on stack
        int newMin = (isEmpty() ? x : Math.min(x, min[top])); // new minimum
        stack[++top] = x;           // push x
        min[top] = newMin;          // push new minimum
    }
}
```