CMSC 420: Spring 2023

Practice Problems for Midterm 1

Problem 0. Expect at least one question of the form "apply operation X to data structure Y," where X is a data structure that has been presented in lecture. (Likely targets: Union-Find, leftist heaps, AVL trees, 2-3 trees, AA trees).

Hint: Intermediate results can be included for partial credit, but don't waste too much time showing intermediate results, since they can steal time from later problems.

Problem 1. Short answer questions. Except where noted, explanations are not required, but may be given to help with partial credit. Whenever asked to express an answer as a function of n, you should assume that n is a large number.

(a) Recall that a binary tree is *full* if every node either has 0 or 2 children. Given a full binary tree with n total nodes, what is the maximum number of leaf nodes? What is the minimum number? Give your answer as a function of n (no explanation needed).

(b) You have an inorder-threaded binary tree with n nodes. Let u be an arbitrary non-leaf node in this tree. **True or False**: There must be at least one thread that points into u.

(c) You have a binary tree with inorder threads (for both inorder predecessor and inorder successor). Let u and v be two arbitrary nodes in this tree. **True or false**: There is a path from u to v, using some combination of child links and threads.

(d) You build a union-find data structure for a set of n objects. Initially, each element is in a set by itself. You then perform k union operations, where $k < n$. Each operation merges two different sets. Can the number of union-find trees be determined from k and n alone? If not, answer "It depends". If so, give the number as a function of k and n.

(e) You are given a binary heap containing n elements, which is stored in an array as `A[1...n]`. Given the index `i` of an element in this heap, present a formula that returns the index of its sibling. (Hint: You can either do this by manipulating the bits in the binary representation of `i` or by using a conditional (if-then-else).)

(f) In a leftist heap containing a large number of elements n, what is the minimum possible NPL value of the root? What is the maximum? (Express your answers as a function of n. It is okay to be off by an additive error of $\pm O(1)$.)

(g) Your boss asks you to program a new function for your leftist (min) heap. Given a leftist heap with a large number of elements n, the operation returns the *third smallest* item in the heap (without modifying its contents). What is the minimum number of heap entries that you might need to inspect to be certain that the third smallest item is among them?

(h) You have just performed an insertion into a 2-3 tree of height h. What is the maximum number of split operations that might be needed as a result? (Express your answer as a function of h.)

(i) You are given a 2-3 tree of height h, which has been converted into an AA-tree. As a function of h, what is the minimum number of red nodes that might appear on any path from a root to a leaf node in the AA tree? What is the maximum number? Explain.

(j) You are given a sorted set of n keys $x_1 < x_2 < \cdots < x_n$ (for some large number n). You insert them all into an AA tree in some arbitrary order. No matter what insertion order to choose, one of these keys *cannot* possibly be a red node. Which is it? Explain.

Problem 2. Suppose that we are given a set of n objects (initially each item in its own set) and we perform a sequence of m unions and finds (using height balanced union and path compression). Further suppose that all the unions occur *before* any of the finds. Prove that after initialization, the resulting sequence will take $O(m)$ time (rather than the $O(m\alpha(m, n))$ time given by the worst-case analysis).

Problem 3. You are given a degenerate binary search tree with n nodes in a left chain as shown on the left of Fig. 1, where $n = 2^k - 1$ for some $k \geq 1$.

(a) Derive an algorithm that, using only single left- and right-rotations, converts this tree into a perfectly balanced complete binary tree (right side of Fig. 1).

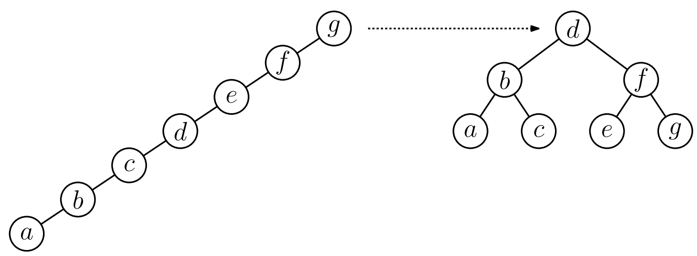

Figure 1: Rotating into balanced form.

(b) As an asymptotic function of n, how many rotations are needed to achieve this? $O(\log n)$? $O(n)$? $O(n \log n)$? $O(n^2)$? Briefly justify your answer.

Problem 4. You are given a binary tree (not necessarily a search tree) where, in addition to `p.left` and `p.right`, each node `p` has a *parent link*, `p.parent`. This points to `p`'s parent, and is `null` if `p` is the root. Given such a tree, present pseudocode for a function that returns the inorder successor of any node `p`. If `p` has no inorder successor, the function returns `null`.

```
Node inorderSuccessor(Node p) {
    // ... fill this in
}
```

Briefly explain how your function works. Your function should run in time proportional to the *height* of the tree.

Problem 5. You are given a standard (unbalanced) binary search tree. Let `root` denote its root node. Present pseudocode for a function `atDepth(int d)`, which is given an integer $d \geq 0$, and outputs the keys for the nodes that are at depth d in the tree (see Fig. 2). The keys should be output in increasing order of key value.

If there are no nodes at depth d, the function returns an empty list. The running time of your algorithm should be proportional to the number of nodes at depths $\leq d$. (For example, in the case of `atDepth(2)`, there are 7 nodes of equal or lesser depth.)

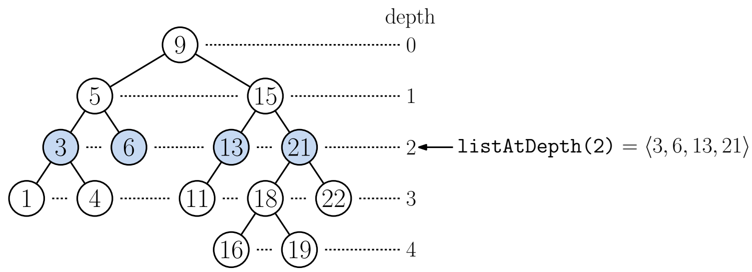

Figure 2: Nodes at some depth.

Hint: Create a recursive helper function. Explain what the initial call is to this function.

Problem 6. Assume that you are given a tree-based heap structure, which is represented by a binary tree (not necessarily complete nor leftist). Each node `u` stores three things, its priority, `u.key`, and the pointers to its subtrees, `u.left` and `u.right`. The keys are min-heap ordered (that is, a node's key is never smaller than its parent). There are no NPL values.

(a) Present pseudocode for a function `swapRight(Node u)` which is given a pointer to the root of a tree. It traverses the right chain of this tree and swaps the left and right subtrees of all nodes along this chain (see Fig. 3). It returns a pointer to the resulting tree. For full credit, your function should run in time proportional to the length of the right chain in the tree.

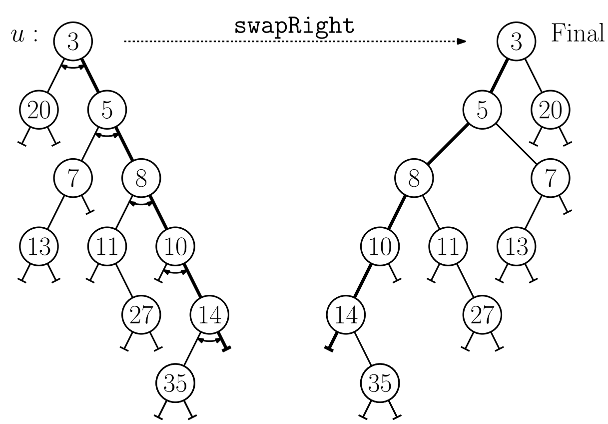

Figure 3: The function `swapRight`.

(b) Present pseudocode for a function `swapMerge(Node u, Node v)`, which is given pointers to the roots of two trees. It merges the right chains of these two trees according to min-heap order, and then performs `swapRight` on the resulting tree (see Fig. 4). It returns a pointer to the resulting tree.

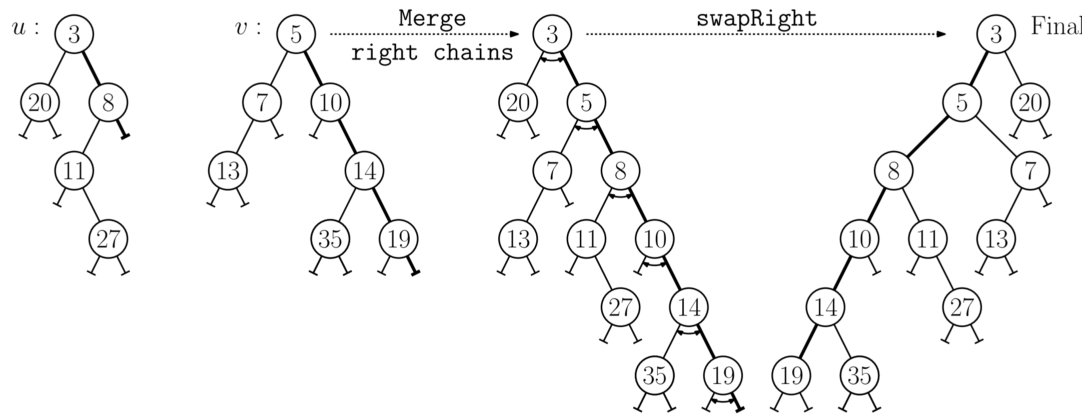

Figure 4: The function `swapMerge`.

Problem 7. Given any AVL tree T and an integer $d \geq 0$, we say that T is *full at depth d* if it has the maximum possible number of nodes (namely, 2^d) at depth d.

Prove that for any $h \geq 0$, an AVL tree of height h is full at all depths from 0 up to $\lfloor h/2 \rfloor$. (For example, the AVL tree of Fig. 2 has height 4, and is full at levels 0, 1, and 2, but it is not full at levels 3 and 4.)

Hint: Prove this by induction on the height of the tree.

Problem 8. Consider the following possible node structure for 2-3 trees, where in addition to the keys and children, we add a link to the parent node. The root's parent link is `null`.

```
class Node23 {                  // a node in a 2-3 tree
    int       nChildren         // number of children (2 or 3)
    Node23    child[3]          // our children (2 or 3)
    Key       key[2]            // our keys (1 or 2)
    Node23    parent            // our parent
}
```

Assuming this structure, answer each of the following questions:

(a) Present pseudocode for a function `Node23 rightSibling(Node23 p)`, which returns a reference to the sibling to the immediate right of node `p`, if it exists. If `p` is the rightmost child of its parent, or if `p` is the root, this function returns `null`. (For example, in Fig. 5, the right sibling of the node containing "2" is the node containing "8:12". Since the node containing "8:12" is the rightmost node of its parent ("4"), it has no right sibling.) Your function should run in $O(1)$ time.

(b) For a node `p` in a 2-3 tree, its *level successor* is the node to its immediate right at the same level. Give pseudocode for a function `Node23 levelSuccessor(Node23 p)`, which returns a reference to `p`'s level successor, if it exists. If `p` is the rightmost node on its level (including the case where `p` is the root), this function returns `null`. (For example, in Fig. 5, the level successor of the node containing "2" is the node containing "8:12", and the level successor of "8:12" is the node containing "19:21".)

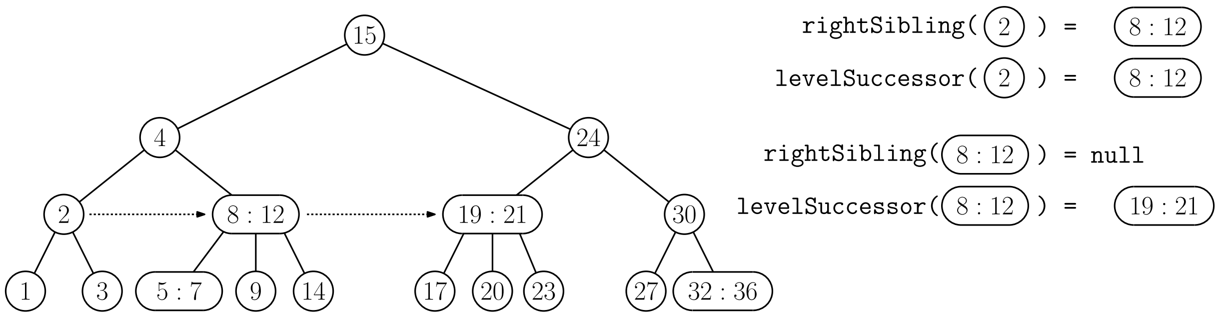

Figure 5: Sibling and level successor in a 2-3 tree.

Your function should run in $O(\log n)$ time. If you like, you may use `rightSibling`.

(c) Suppose we start at any node `p` in a 2-3 tree with n nodes, and we repeatedly perform `p = levelSuccessor(p)` until `p == null`. What is the (worst-case) total time needed to perform all these operations? (Briefly justify your answer.)

Problem 9. Each node of a 2-3 tree may have either 2 or 3 children, and these nodes may appear anywhere within the tree. Let's imagine a much more rigid structure, where the node types alternate between levels. The root is a 2-node, its two children are both 3-nodes, their children are again 2-nodes, and so on (see Fig. 6). Generally, depth i of the tree consists entirely of 2-nodes when i is even and 3-nodes when i is odd. (Remember that the *depth* of a node is the number of edges on the path to the root, so the root is at depth 0.) We call this an *alternating 2-3 tree*. While such a structure is too rigid to be useful as a practical data structure, its properties are easy to analyze.

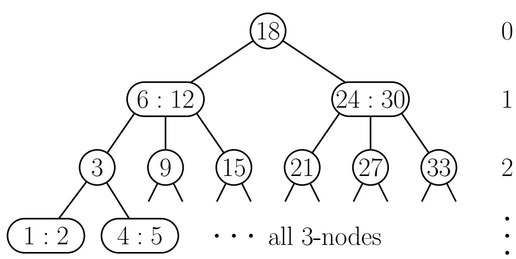

Figure 6: Alternating 2-3 tree.

(a) For $i \geq 0$, define $n(i)$ to be the number of nodes at depth i in an alternating 2-3 tree. Derive a closed-form mathematical formula (exact, not asymptotic) for $n(i)$. Present your formula and briefly explain how you derived it.

By "closed-form" we mean that your answer should just be an expression involving standard mathematical operations. It is *not* allowed to involve summations or recurrences, but it is allowed to include cases, however, such as

$$n(i) = \begin{cases} \dots & \text{if } i \text{ is even} \\ \dots & \text{if } i \text{ is odd.} \end{cases}$$

(b) For $i \geq 0$, define $k(i)$ to be the number of keys stored in the nodes at depth i in an alternating 2-3 tree. (Recall that each 2-node stores one key and each 3-node stores 2 key). Derive a closed-form mathematical formula for $k(i)$. Present your formula and briefly explain how you derived it. (The same rules apply for "closed form", and further your formula should stand on its own and not make reference to $n(i)$ from part (a).)

Problem 10. In this problem, we will consider variations on the amortized analysis of the dynamic stack. Let us assume that the array storage only *expands*, it never contracts. As usual, if the current array is of size m and the stack has fewer than m elements, a `push` costs 1 unit. When the mth element is pushed, an overflow occurs.

You are given two constants $\gamma, \delta > 1$. When an overflow occurs, we allocate a new array of size γm, copy the elements from the old array over to the new array. The total cost is 1 (for the push) plus δm (for copying). Derive a tight bound on the amortized cost, which holds in the limit as $m \to \infty$. Express your answer as a function of γ and δ. Explain your answer.