Practice Problems for the Midterm

Exam Logistics:

- The Midterm Exam will be asynchronous and online. The exam will be made available through Gradescope for a 48-hour period starting at **12:00am the morning of Thu, Oct 29** and running through **11:59pm the evening of Fri, Oct 30**. The exam is designed to be taken over a 90-minute time period, but to allow time for scanning and uploading, you will have **2 hours** to submit the exam through Gradescope once you start it.
- The exam will open-book, open-notes, open-Internet, but the exam must be done on your own without the aid of other people or software. (You may use a simple arithmetic calculator, but I don't expect that you will need one.)
- Do not discuss any aspects of the exam with classmates during the exam's 48-hour time window, even if you have both submitted. This includes its content, its difficulty, and its length.
- If any questions arise while you are taking the exam, please either email me (`mount@umd.edu`) or make a *private* Piazza post. (Do *not* ask your classmates.) If you are unsure about how to interpret a problem and I do not respond in a timely manner, please do your best and write down any assumptions you are making. There will be no "trick" questions on the exam. Thus, if a question doesn't make sense or seems too easy or too hard, please check with me.
- If you experience any technical issues while taking the exam, **don't panic**. Save you work (ideally in a manner that attaches a time stamp), and contact me by email (`mount@umd.edu`) as soon as possible. I understand that unforeseen events can occur, and I will attempt make reasonable accommodations.

Disclaimer: These practice problems have been extracted from old homework assignments and exams. Material changes from semester to semester. These do **not** reflect the actual coverage, difficulty, or length of the midterm exam.

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.

Problem 1. Short answer questions. Except where noted, explanations are not required, but may be given to help with partial credit.

(a) 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) **True or false?** Let T be extended binary search tree (that is, one having internal and external nodes). In an inorder traversal, internal and external nodes are encountered in *alternating order*. (If true, provide a brief explanation. If false, show a counterexample.)

(c) **True or false?** In every extended binary tree having n external nodes, there exists an external node of depth at most $\lceil \lg n \rceil$. **Explain briefly.**

(d) What is the minimum and maximum number of levels in a 2-3 tree with n nodes. (Define the number of levels to be the height of the tree plus one.) Hint: It may help to recall the formula for the geometric series: $\sum_{i=0}^{m-1} c^i = (c^m - 1)/(c - 1)$.

(e) You have an AVL tree containing n keys, and you *insert* a new key. As a function of n, what is the maximum number of rotations that might be needed as part of this operation? (A double rotation is counted as two rotations.) Explain briefly.

(f) Repeat (e) in the case of *deletion* from an AVL tree. (You can give your answer as an asymptotic function of n.)

(g) You are given a 2-3 tree of height h, which you convert to 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?

(h) Both skip lists and B-trees made use of nodes containing a variable number of elements. (In the skip list, and node has a variable number of pointers, and in a B-tree a node has a variable number of keys/children.) In one case, we allocated nodes of variable size and in the other case, we allocated nodes of the same fixed size. Why did we do things differently in these two cases?

(i) Unbalanced search trees and treaps both support dictionary operations in $O(\log n)$ "expected time." What difference is there (if any) in the meaning of "expected time" in these two contexts?

(j) Splay trees are known to support efficient *finger search queries*. What is a "finger search query"?

(k) Consider a splay tree containing n keys $a_1 < a_2 < \cdots < a_n$. Let x, y, and z be any three consecutive elements in this sorted sequence. Suppose that we perform `splay`(x) followed immediately by `splay`(z). What (if anything) can be said about the depth of y at this time?

Problem 2. You are given a degenerate binary search tree with n nodes in a left chain as shown in Fig. 3, 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 (see Fig. 1).

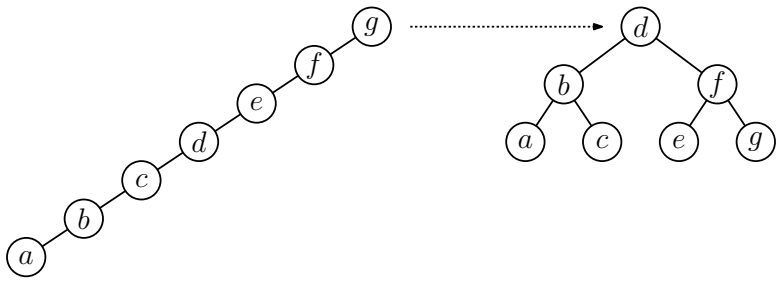

Figure 1: Rotating into balanced form.

(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 3. You a given a threaded binary search tree T (not necessarily balanced). Recall that each node has additional fields `p.leftIsThread` (resp., `p.rightIsThread`). These indicate whether `p.left` (resp., `p.right`) points to an actual child or it points to the inorder predecessor (resp., successor).

Present pseudocode for each of the following operations. Both operations should run in time proportional to the height of the tree.

(a) `void T.insert(Key x, Value v)`: Insert a new key-value pair (x, v) into T and update the node threads appropriately (see Fig. 2(a)).

(b) `Node preorderSuccessor(Node p)`: Given a non-null pointer to any node p in T, return a pointer to its *preorder successor*. (Return `null` if there is no preorder successor.)

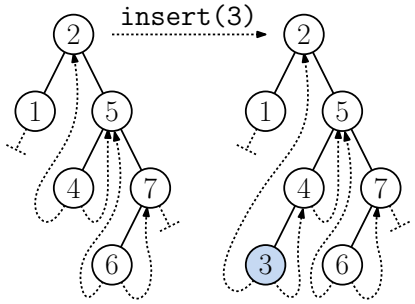

Figure 2: Threaded tree operations.

Problem 4. You are given a binary search tree where, in addition to the usual fields `p.key`, `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 pseudo-code for a function

```
Node preorderSuccessor(Node p)
```

which is given a non-null reference `p` to a node of the tree and returns a pointer to `p`'s *preorder successor* in the tree (or `null` if `p` has no preorder successor). Your function should run in time proportional to the height of the tree. Briefly explain how your function works.

Problem 5. Suppose that in addition to the key-value pair (`p.key` and `p.value`) and pointers to the node's left and right children (`p.left` and `p.right`), each node of a binary search tree stores a sibling pointer, `p.sibling`, which points to `p`'s sibling, or `null` if `p` has no sibling.

Recall the following code for performing a right rotation of a node in a binary search tree:

```
Node rotateRight(Node p) {
    Node q = p.left;
    p.left = q.right;
    q.right = p;
    return q;
}
```

Modify the above code so that, in addition to performing the rotation, the sibling pointers are also updated. (You may *not* assume the existence of other information, such as parent pointers or threads. Your function should run in constant time.)

Problem 6. A *zig-zag tree* is defined to be a binary search tree having an odd number of nodes that consists of a single path, alternating between right- and left-child links. An example is shown in Fig. 3, where we have also labeled each node with its *depth*, that is, the length of the path from the root.

(a) Draw the final tree that results from executing `splay("e")` on the tree of the figure below. (Intermediate results can be given for partial credit.)

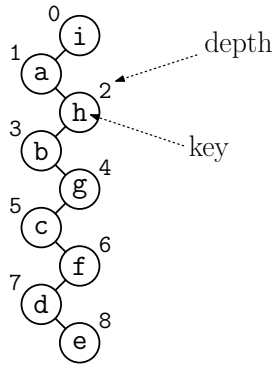

Figure 3: Zig-zag splay tree (with node depths shown).

(b) Let T be a zig-zag tree with n nodes, and let T' be the tree that results after performing a splay operation on T's deepest leaf. Consider a node p at level k in T, for $0 \leq k \leq n-2$. What is the depth of p in T'? Express your answer as a function of k and n. Your formula should apply to every node of the tree, *except the node that was splayed*.

(c) Give a short proof justifying the correctness of your formula.

Problem 7. You are given a skip list with n nodes in which, rather than promoting each node to the next higher level with probability $1/2$, we promote each node with probability p, for $0 < p < 1$.

(a) Given a skip list with n keys, what is the expected number of keys that contribute to the ith level. (Recall that the lowest level is level 0.) Briefly explain.

(b) Show that (excluding the header and sentinel nodes) the total number of links in such a skip list (that is, the total size of all the skip list nodes) is expected to be at most $n/(1-p)$. (Hint: It may be useful to recall the formula for the geometric series from Problem 1(d).)

Problem 8. Show that if all nodes in a splay tree are accessed (splayed) in sequential order, the resulting tree consists of a linear chain of left children.

Problem 9. The objective of this problem is to design an enhanced stack data structure, called `MinStack`. For concreteness, let's assume that the stack just stores integers. Your stack should support the standard stack operations `void push(int x)`, which pushes `x` on top of the stack, and `int pop()`, which removes the element at the top of the stack and returns its value. It must also support the additional operation, `int getMin()`, which returns the smallest value currently in the stack, *without altering the contents of the stack*. Finally, there is a constructor `MinStack(int n)`, which is given the maximum number n of items that will be stored in the stack.

Present pseudocode for a data structure that implements these operations. All operations should run in $O(1)$ time. (We will give partial credit if algorithm is correct, but your running time is worse than this.) Your answer should include the following things:

- Explain what objects are maintained by your data structure.
- Explain how the data structure is initialized (that is, what does the constructor do?)
- Present pseudocode descriptions of `push(x)`, `pop()`, and `getMin()`.

No error checking is needed. (No more than n elements will be in the stack at any time and no `pop` or `getMin` from an empty stack.)