CMSC 420: Spring 2021

Practice Problems for Midterm 1

Exam Logistics: Please read these now, so you don't have to waste time during the exam.

- This 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, Mar 11** and running through **11:59pm the evening of Fri, Mar 12**. 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 be open-book, open-notes, open-Internet, but it 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.)
- You may freely use any information from class, without citing it. Information from external sources (e.g., books or online sources) can be used without penalty, but you *must* cite your sources (e.g., "I found this on Stack Overflow") and express the answer in your own words.
- Please 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, difficulty, and length.
- If you have any questions during the 2-day exam window, please email me (mount@umd.edu) or make a *private* Piazza post. (Please post privately even if you have not yet started. Others in the class may be working on the exam.)
- If you are unsure about how to interpret a problem and I do not respond in a timely manner, please do your best. 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.
- Uploading a large image pdf will take time. Please allow sufficient time to submit your final work. While I do not want to penalize people for having slow network connectivity, *I reserve the right to deduct 2% of the final grade if you do not complete your upload within the 2-hour deadline.* (You can submit multiple times.)
- 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.
- The exam will be long, and so be mindful of this. To get the most credit from each problem, on your first pass give just the answer and any required justification. If time permits, go back and fill in intermediate results and explanations to help with partial credit.

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. Here is an example from last semester.

(a) Consider the 2-3 tree shown the figure below. Show the **final tree** that results after the operation `insert(6)`. When rebalancing, use only splits, *no adoptions* (key rotations).

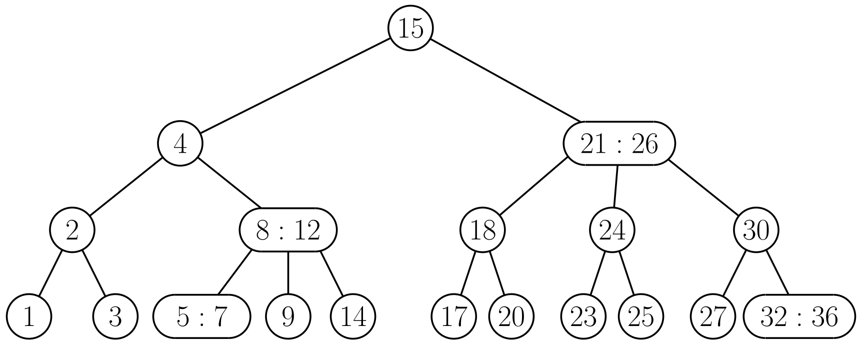

Figure 1: 2-3 tree insertion and deletion.

(b) Returning to the original tree, show the **final tree** that results after the operation `delete(20)` When rebalancing, you may use *both merge and adoption* (key rotation). If either operation can be applied, give priority to adoptions.

In both cases, you may draw intermediate subtrees to help with partial credit, but don't waste too much time on this.

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) 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?

(i) You have a set of n distinct keys, where n is a large even number. You randomly permute all the keys and insert the first $n/2$ of them into a standard (unbalanced) binary search tree. You then take the remaining $n/2$ keys, sort them in ascending order, and insert them into this tree. The final tree has n nodes. As a function of n, what is the expected height of this tree? (Select the best from the choices below.)

  (i) $O(\log n)$
  (ii) $O((\log n)^2)$
  (iii) $O(\sqrt{n})$
  (iv) $O(n)$

(j) You have a valid AVL tree with n nodes. You insert two keys, one smaller than all the keys in the tree and the other larger than all the keys in the tree, but you do no rebalancing after these insertions. **True or False**: The resulting tree is a valid AVL tree. (Briefly explain.)

(k) By mistake, two keys in your treap happen to have the same priority. Which of the following is a possible consequence of this mistake? (Select one)

  (i) The `find` algorithm may abort, due to dereferencing a `null` pointer.
  (ii) The `find` algorithm will not abort, but it may return the wrong result.
  (iii) The `find` algorithm will return the correct result if it terminates, but it might go into an infinite loop.
  (iv) The `find` algorithm will terminate and return the correct result, but it may take longer than $O(\log n)$ time (in expectation over all random choices).
  (v) There will be no negative consequences. The find algorithm will terminate, return the correct result, and run in $O(\log n)$ time (in expectation over all random choices).

Problem 2. You are given a degenerate binary search tree with n nodes in a left chain as shown on the left of Fig. 2, 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. 2).

(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 an inorder 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.

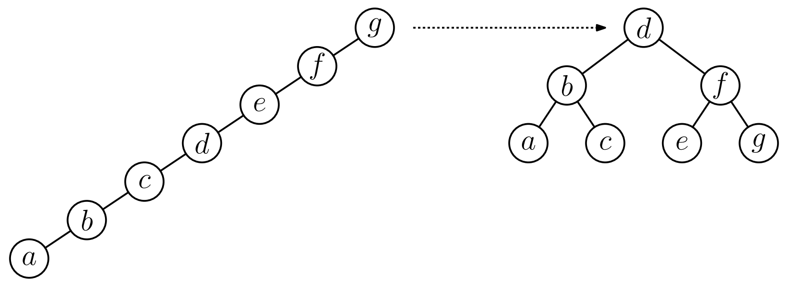

Figure 2: Rotating into balanced form.

(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. 3(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.)

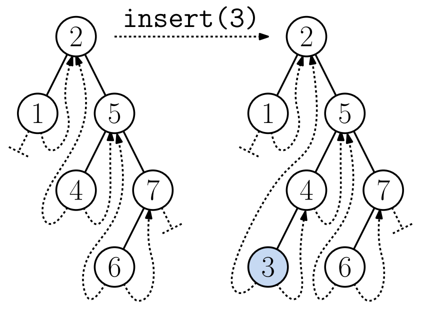

Figure 3: Threaded tree operations.

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 preorderPred(Node p)
```

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

Problem 5. 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. **??**, 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.

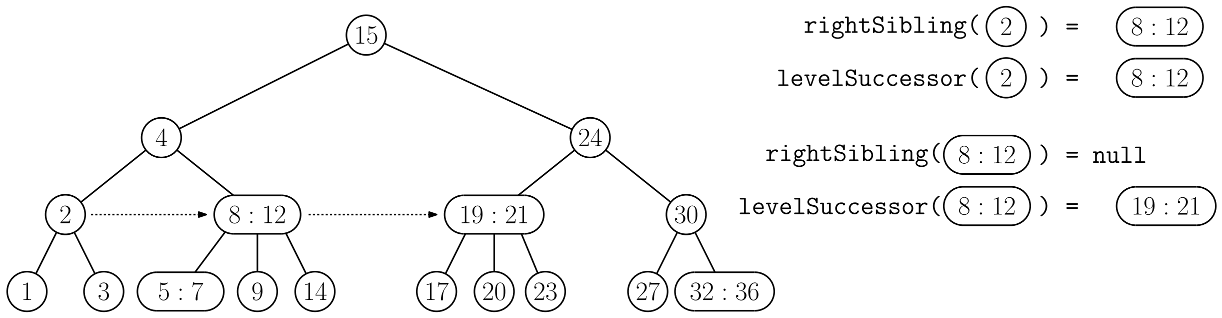
rightSibling(2) = 8 : 12
levelSuccessor(2) = 8 : 12
rightSibling(8 : 12) = null
levelSuccessor(8 : 12) = 19 : 21
15
4
24
2
8 : 12
19 : 21
30
1
3
5 : 7
9
14
17
20
23
27
32 : 36

Figure 4: Sibling and level successor in a 2-3 tree.

(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. 4, 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".)

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 6. A *social-distanced bit vector* (SDBV) is an abstract data type that stores bits, but no two 1-bits are allowed to be consecutive. It supports the following operations (see Fig. 5):

- `init(m)`: Creates an empty bit vector `B[0..m-1]`, with all entries initialized to zero.
- `boolean set(i)`: For $0 \leq i \leq m$ (where m is the current size of `B`), this checks whether the bit at positions i and its two neighboring indices, $i-1$ and $i+1$, are all zero. If so, it sets the ith bit to 1 and returns `true`. Otherwise, it does nothing and returns `false`. (The first entry, `B[0]`, can be set, provided both it and `B[1]` are zero. The same is true symmetrically for the last entry, `B[m-1]`.)

  For example, the operation `set(9)` in Fig. 5 is successful and sets `B[9] = 1`. In contrast, `set(8)` fails because the adjacent entry `B[7]` is nonzero.

There is one additional feature of the SDBV, its ability to *expand*. If we ever come to a situation where it is impossible set any more bits (because every entry of the bit vector is

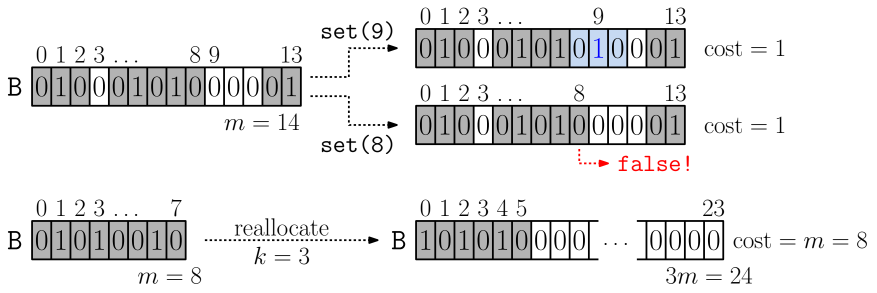

Figure 5: Social-distanced bit vector. (Shaded entries cannot be set to one, due to social-distancing.)

either nonzero or it is adjacent to an entry that is nonzero), we *reallocate* the bit vector to one of three times the current size. In particular, we replace the current array of size m with an array of size $3m$, and we copy all the bits into this new array, compressing them as much as possible. In particular, if k bits of the original vector were nonzero, we set the entries $\{0, 2, 4, \ldots, 2k\}$ to 1, and all others to 0 (see Fig. 5).

The cost of the operation `set` is 1, unless a reallocation takes place. If so, the cost is m, where m is the size of the bit vector *before* reallocation.

Our objective is to derive an amortized analysis of this data structure.

(a) Suppose that we have arrived at a state where we need to reallocate an array of size m. As a function of m, what is the minimum and maximum number of bits of the SDBV that are set to 1? (Briefly explain.)

(b) Following the reallocation, what is the minimum number of operations that may be performed on the data structure until the next reallocation event occurs? Express your answer as a function of m. (Briefly explain.)

(c) As a function of m, what is the cost of this next reallocation event? (Briefly explain.)

(d) Derive the amortized cost of the SDBV. (For full credit, we would like a tight constant, as we did in the homework assignment. We will give partial credit for an asymptotically correct answer. Assume the limiting case, as the number of operations is very large and the initial size of the bit vector is small.)

Throughout, if divisions are involved, don't worry about floors and ceilings.