Drill problems —

- A mistake in the original solution was corrected for one of the 2-4 tree removal problems and the question regraded accordingly, which means that some solutions marked correct originally may no longer be receiving full credit. You can (and should) fix this for revise-and-resubmit. (Also make sure you understand why the new solution is correct.)
- Also be sure to enter a dash for unused blanks rather than just leaving things empty. Full credit has been awarded in cases where that was the only problem, but properly entering your answers in the first place speeds things up and means that the autograding can give you a correct score.

Describing an algorithm requires clear steps that can be followed. This influences the organization — give the steps as steps, in the order they are to be done — as well as the level of detail in the description. The level of detail should be appropriate for the audience and context: the steps should be precise enough that someone who understands the relevant basic operations could carry them out without needing to guess what to do next, but not so detailed that every trivial action is spelled out unnecessarily. The goal is to make the procedure unambiguous while keeping the explanation readable and focused on the essential ideas.

When asked to design a data structure or algorithm to achieve a particular running time, be sure to address why your design does actually achieve that — assess the running time of your solution and explain where that comes from by referencing well-known operations used or analyzing the particular steps of your algorithm.

#2 (ADM 3-18) is about the implementation of *successor(x)* and *predecessor(x)*. As with *delete(x)*, whether x is the element itself or a pointer to its location makes a big difference in terms of the implementation and running time of the operations. The description of these operations at the beginning of section 3.3 (Dictionaries) says "Given a pointer x to a given data item ..." for *delete* and "immediately before (or after) item x" for *successor* and *predecessor*, so that would be the expected interpretation (x is the element itself for *successor* and *predecessor*). However, the different interpretation for *delete* compared to the other operations is a point of confusion so either interpretation was accepted — but it is important to state your assumptions so that it is clear that you realize there's a difference between knowing only the element's value and knowing where the element is in the tree.

Level of detail in the algorithm —

- You don't need to describe well-known standard operations such as search tree search, insert, delete — if you use them as is, just reference "standard balanced search tree insert" (or whatever), and if you make modifications, just address the

modifications ("standard balanced search tree insert but you do ... to each node visited on the way down the tree").

- "Identify the predecessor/successor" is well-defined in terms of an algorithmic step to carry out (the terms identify specific elements) but it insufficient for determining running time — *how* those elements are found is crucial there...
- ...though, arguably, finding the predecessor and successor are well-known operations — they aren't as common as search, insert, delete but they are a reasonably common extension to the basic Dictionary ADT and thus also reasonably common things to do in a search tree. Clarifying the implementation by referring to the standard predecessor (successor) implementation is sufficient.

It is worth improving on runtime even if the big-Oh doesn't change, especially if there is a simple(r) solution. For example, you can use the standard predecessor/successor operations to find the predecessor and successor for a new element being inserted (and that running time doesn't exceed what is needed for insert in the first place), but can you do better? $O(1)$ perhaps?

#3 (ADM 3-25) asked about how to efficiently implement the two heuristics described, which means identifying how to locate the bin with the smallest amount of space beyond what is needed for the current item (best fit) or the bin with the most space available (worst fit) — it wasn't about trying to find an expression for the actual minimum number of bins or critiquing the effectiveness of the heuristics at finding the smallest number of bins.

Be sure to specify how a chosen data structure is used if it is not the elements themselves that dictate ordering or other aspects of usage. In this case, bins by the themselves aren't orderable, so if you are ordering the bins to make it easier to find the desired one, identify the property (total capacity, space filled, space remaining, etc) by which you are ordering them.

Order matters for balanced parens in #5 (ADM 3-2) — `)(` is not a set of balanced parens even though there are the same number of open and closed. The problem is also looking for the sum of the lengths of sequences of balanced parens, not the longest such. The answer should be 6 rather than 4 for `(())()`.

Noticing the section an exercise is in can provide you with a strong hint about how to go about coming up with a solution — #5 is in a section titled "Stacks, Queues, and Lists"... While it is true that this problem is more of a case of "inspired by" than "actually uses" — you can capture the key aspect of the data structure without actually using the data structure itself — it is still worthwhile starting the problem by thinking about how a stack or queue or list could help solve the problem and to then reference that when giving the overview of the solution idea before launching into the details of your algorithm.