Exercises

Please do the exercises on your own.

1. **(2 pt.)** For each of the following examples, if the nodes can be colored red or black to make a legitimate red-black tree, then give such a coloring. If not, then say that they cannot.

   (For reference, the \LaTeX code to make these trees is included at the end of the PSET, as well as instructions about how to color the nodes red or black. If you use this code, make sure you include \texttt{\usepackage{tikz}} before the \texttt{\begin{document}} command. You are also welcome to take a screenshot and then color the trees in MSPaint, or make a hand-drawn copy and take a photo, or...)

   ![Red-Black Tree Examples](image)

   \[\textbf{We are expecting: For each tree, either an image of a colored-in red-black tree or a statement “No such red-black tree.” No justification is required.}\]

   \[\textbf{SOLUTION:}\]

   ![Red-Black Tree Solutions](image)
2. (5 pt.) This exercise references the IPython notebook HW4.ipynb, available on the course website.

In our implementation of radixSort in class, we used bucketSort to sort each digit. Why did we use bucketSort and not some other sorting algorithm? There are several reasons, and we'll explore one of them in this exercise.

(a) (2 pt.) One reason we chose bucketSort was that it makes radixSort work correctly! In HW4.ipynb, we've implemented four different sorting algorithms—bucketSort, quickSort, and two versions of mergeSort—as well as radixSort.

Note: The IPython notebook is long, but just because it implements many different sorting algorithms. Don’t get scared!

There is a TODO statement in the IPython notebook where you can change the code to use different sorting algorithms; you just have to make sure that the sorting algorithm you want to use is the one that is not commented out. Modify the code for radixSort to use each one of these four algorithms within radixSort, and test it out on the examples suggested.

Which sorting algorithms seem to be correct as “inner sorting algorithms” for radixSort?

- Does using bucketSort always work correctly?
- Does using quickSort always work correctly?
- Does using mergeSort (with merge1) always work correctly?
- Does using mergeSort (with merge2) always work correctly?

[We are expecting: Yes or no for each part.]

SOLUTION:

- Does using bucketSort always work correctly? [Your answer]
- Does using quickSort always work correctly? [Your answer]
- Does using mergeSort (with merge1) always work correctly? [Your answer]
- Does using mergeSort (with merge2) always work correctly? [Your answer]

(b) (3 pt.) Explain what you saw above. What was special about the algorithms which worked? Why does this special thing matter? (You may wish to play around with HW4.ipynb to “debug” the incorrect cases.)

[We are expecting: A clear definition of the special property that the correct algorithms have, and a few sentences explaining why it matters. A minimal example of what might go wrong is a great way to explain why this property matters.

You do not need to justify why each of the algorithms do or do not have the property. ]

Note: yes, this property is alluded to in the reading. That’s why this is an exercise and not a problem! To get the most out of this exercise, play around with the examples in the code and make sure you really understand what’s going on.

SOLUTION:
Problems

You may talk with your fellow CS161-ers about the problems. However:

• Try the problems on your own before collaborating.

• Write up your answers yourself, in your own words. You should never share your typed-up solutions with your collaborators.

• If you collaborated, list the names of the students you collaborated with at the beginning of each problem.

3. (6 pt.) [Duck.] Suppose that the $n$ ducks from the previous two PSETs are standing in a line, ordered from shortest to tallest.

You have a measuring stick of a certain height, and you would like to identify a duck which is the same height as the stick, or else report that there is no such duck. The only operation you are allowed to do is \texttt{compareToStick}(j), where $j \in \{0, \ldots, n-1\}$, which has one of the following three outputs:

• \texttt{taller} if the $j$'th duck is taller than the stick,
• \texttt{shorter} if the $j$'th duck is shorter than the stick, or
• \texttt{the same} if the $j$'th duck is the same height as the stick.

As in HW3, you forgot to bring a piece of paper, so you can only store up to seven integers in $\{0, \ldots, n-1\}$ at a time.

At this point, the ducks have gotten accustomed to ice cream, so you have to pay a duck an ice cream cone in order to compare it to the stick.
(a) (3 pt.) Give an algorithm which either finds a duck the same height as the stick, or else returns “No such duck,” in the model above which uses $O(\log n)$ ice cream cones.

[We are expecting: Pseudocode AND an English description of your algorithm. You do not need to justify the correctness or ice cream cone usage.]

I collaborated with:

SOLUTION:

English description:

Pseudocode:

(b) (3 pt.) Prove that any algorithm that satisfies the requirements in part (a) must use $\Omega(\log n)$ ice cream cones.

[We are expecting: A convincing argument at the same level of rigor that we saw for the sorting lower bounds in class.]

I collaborated with:

SOLUTION:
4. (5 pt.) [Duck.] A wise duck has knowledge of an array $A$ of length $n$, so that $A[i] \in \{1, \ldots, k\}$ for all $i$ (note that the elements of $A$ are not necessarily distinct). You don’t have direct access to the list, but you can ask the wise duck any yes/no questions about it. For example, you could ask “If I remove $A[5]$ and swap $A[7]$ with $A[8]$, would the array be sorted?” or “are ducks related to grebes?”

This time you did bring a paper and pencil, and your job is to write down all of the elements of $A$ in sorted order.\footnote{Note that you don’t have any ability to change the array $A$ itself, you can only ask the wise duck about it.} You are allowed to take all the time you need to do any computations on paper with the wise duck’s answers, but as with the $n$ ducks above, the wise duck charges one ice cream cone per question.

$A = [6, 2, 4, 3, 3, 5, 2, 1, 2, 6]$
(a) (5 pt.) Design an algorithm which outputs a sorted version of $A$ which uses $O(k \log n)$ ice cream cones. You may assume that you know $n$ and $k$, although this is not necessary.

[We are expecting:
  • Pseudocode AND a clear English explanation of what it is doing.
  • An explanation of why the algorithm uses $O(k \log n)$ ice cream cones.
You do not need to prove that your algorithm is correct. ]

I collaborated with:

SOLUTION:

English description:

Pseudocode:

Ice cream cone complexity:

[HINT: One approach is to think first about what you would do for $k = 2$: that is, when $A$ contains only the numbers 1 and 2. What information do you need to write down a sorted version of $A$ in this case? ]

(b) (1 BONUS pt.) Prove that any procedure to solve this problem must use $\Omega(k \log \frac{n}{k})$ ice cream cones.

[We are expecting: Nothing, this part is not required. To get the bonus point, should give a convincing argument at the same level of rigor that we saw for sorting lower bounds in class.]

I collaborated with:

SOLUTION:
5. (5 pt.) [Goose!] A goose comes to you with the following claim. They say that they have come up with a new kind of binary search tree, called gooseTree, even better than red-black trees!

More precisely, gooseTree is a data structure that stores data in a binary search tree. It might also store other auxiliary information, but the goose won’t tell you how it works. The goose claims that gooseTree supports the following operations:

- gooseInsert(k) inserts an item with key k into the gooseTree, maintaining the BST property. It does not return anything. It runs in time O(1).
- gooseSearch(k) finds and returns a pointer to node with key k, if it exists in the tree. It runs in time O(log n).
- gooseDelete(k) removes and returns a pointer to an item with key k, if it exists in the tree, maintaining the BST property. It runs in time O(log n).
- getGooseBST() returns a pointer to the root of the BST that the gooseTree maintains. It runs in time O(1).

Above, n is the number of items stored in the gooseTree. The goose says that all these operations are deterministic, and that gooseTree can handle arbitrary comparable objects (that is, it only interacts with the objects by comparing them).

You think the goose’s claims are a bit loosey-goosey. How do you know the goose is wrong?

Note that since the goose won’t tell you how gooseTree works, you cannot assume anything about how gooseInsert, gooseSearch, gooseDelete or getGooseBST() might be implemented.

[We are expecting: A proof that the goose must be wrong. (It is okay if it is a short proof). You may invoke results or algorithms that we have seen in class.]

[HINT: Consider a proof by contradiction...can you use the gooseTree to do something impossible? ]

I collaborated with:

SOLUTION:
Feedback

There’s no “correct” answer here, and it is completely anonymous.

6. (1 pt.) Please fill out the following poll, which asks about the pace of the class:

    https://forms.gle/PcLbqtNqcFcmiKySA

Did you fill out the poll?

[We are expecting: The answer “yes.”]

Not yet!
Helpful \LaTeX\ code!

Here is some code to generate the red-black trees; we’ve shown how to color one of the nodes green, you can use this example to color them red or black. You are also welcome to take a screenshot and color in your favorite paint program, or include a picture of a hand-drawn image, or any other way you can think of getting a picture of a colored red-black tree into your PSET.

If you use this code, make sure you include \usepackage{tikz} before the \begin{document} command.

\begin{center}
\begin{tikzpicture}[xscale=0.7]
\begin{scope}
\node[draw,circle,fill=green](b) at (0,0) {};
\node[draw,circle](b0) at (-2,-1) {};
\node[draw,circle](b1) at (2,-1) {};
\node[draw,circle](b00) at (-3,-2) {};
\node[draw,circle](b01) at (-1,-2) {};
\node[draw,circle](b10) at (1,-2) {};
\node[draw,circle](b11) at (3,-2) {};
\node[draw,circle](b000) at (-4,-3) {};
\node[draw,circle](b001) at (-2.5,-3) {};
\node[draw,circle](b010) at (-1.5,-3) {};
\node[draw,circle](b0000) at (-5,-4) {};
\node[draw,circle](b0001) at (-3,-4) {};
\draw (b) -- (b0);
\draw (b) -- (b1);
\draw (b0) -- (b00);
\draw (b0) -- (b01);
\draw (b1) -- (b10);
\draw (b1) -- (b11);
\draw (b00) -- (b000);
\draw (b00) -- (b001);
\draw (b01) -- (b010);
\draw (b000) -- (b0000);
\draw (b000) -- (b0001);
\draw (b010) -- (b011);
\draw (b0000) -- (b00000);
\draw (b0001) -- (b00001);
\end{scope}
\begin{scope}[xshift=10cm]
\node[draw,circle](b) at (0,0) {};
\node[draw,circle](b0) at (-2,-1) {};
\node[draw,circle](b1) at (2,-1) {};
\node[draw,circle](b00) at (-3,-2) {};
\node[draw,circle](b01) at (-1,-2) {};
\node[draw,circle](b10) at (1,-2) {};
\node[draw,circle](b11) at (3,-2) {};
\node[draw,circle](b000) at (-4,-3) {};
\node[draw,circle](b001) at (-2.5,-3) {};
\node[draw,circle](b010) at (-1.5,-3) {};
\node[draw,circle](b0000) at (-5,-4) {};
\node[draw,circle](b0001) at (-3,-4) {};
\draw (b) -- (b0);
\draw (b) -- (b1);
\draw (b0) -- (b00);
\draw (b0) -- (b01);
\draw (b1) -- (b10);
\draw (b1) -- (b11);
\draw (b00) -- (b000);
\draw (b00) -- (b001);
\draw (b01) -- (b010);
\draw (b000) -- (b0000);
\draw (b000) -- (b0001);
\draw (b010) -- (b011);
\draw (b0000) -- (b00000);
\draw (b0001) -- (b00001);
\end{scope}
\end{tikzpicture}
\end{center}