Problem Set Policies

The problem sets in CS166 will consist of a combination of design questions (where you'll devise your own data structures), theory questions (where you'll prove results related to various data structures), and implementation questions (where you'll implement data structures or run performance trials). This handout is designed to give you a sense of what we're looking for in your solution sets.

Submission Instructions

All written problems should be submitted online through GradeScope. You can sign up for GradeScope using this code:

`93DENM`

No hardcopy submissions for written problems will be accepted without the prior approval of the course staff; it's logistically quite difficult to handle both hardcopies and electronic copies.

Please do not submit handwritten solution sets. They're hard to read on GradeScope and it makes grading a lot more difficult. We recommend typesetting your answers with LaTeX; there are a number of online tools you can use to collaboratively edit LaTeX documents.

We ask that you submit your answers to programming questions separately from your written answers. You can submit your code electronically by `ssh`ing into one of the Stanford computer clusters (for example, `corn`), `cd`-ing into the directory containing your solution files, then running

```
/usr/class/cs166/bin/submit
```

in the directory that you want to submit. You'll be prompted for your name, whether you worked with a partner, and the problem set number. We'll test your code on the `corn` machines, so please make sure that your code works correctly there before submitting.

Answering Design Questions

Many questions on the problem set will ask you to design a data structure or algorithm that solves a problem within a particular time bound. When writing up answers to these questions, we recommend that you structure your solution as follows:

- **Begin with a short, high-level description of the idea behind the data structure**. This should be a two or three sentence paragraph describing the intuition behind the data structure. This will help the TAs get a better sense for how the data structure works.
- **Describe the representation of the data structure**. Give some details about how the data structure is actually put together. You can do this with details such as "store two max heaps called `a` and `b`," or by describing a modification of an existing data structure, such as "store a Fibonacci heap, but where each node stores a pointer into a balanced binary search tree."
- **Describe any invariants or accounting schemes for the data structure**. Some data structures maintain strict invariants on their internal representation. For example, a binary min-heap data structure ensures that each node always stores a value no larger than its children and that the tree is a complete binary tree. If your data structure doesn't have any invariants, you don't need to list anything. When we begin discussing amortized analysis, you can also list any charging schemes or potential functions here.
- **Describe each of the operations and give their runtimes**. For each operation, describe how that operation is performed. We'd prefer explanations in plain English, but if you think that pseudocode would be better, you can use that if you'd like. *Just make sure that your description is complete – there shouldn't be any ambiguities in how to perform each operation.* Then, explain why these operations are correct and justify why the data structure meets specified time bounds. You don't need to write a formal proof of correctness unless asked.

For example, consider the following problem:

> Design a data structure that supports the following operations: **insert(*x*)**, which inserts real number x into the data structure and runs in time $O(\log n)$, where n is the number of elements in the data structure, and **find-median()**, which returns the median of the data set if it is nonempty and runs in time $O(1)$.

This is great problem to work through if you haven't seen it before. We have a sample solution on the next page, so try this problem out before moving on. As a hint, try using heaps.

Here is a possible answer to this problem and a sample writeup. Note that you don't need to include section headers like these; we're just doing this because in this case we think it's easier to read.

Overview:

This data structure works by storing the data in a min-heap and a max-heap such that the two middle values are at the top of each heap. Since only O(1) enqueues and dequeues are required per insert and only O(1) find-mins are required per find-median, the data structure fits within the time bounds.

Representation:

A max-heap *left* and a min-heap *right*.

Invariants:

There are two invariants: the *ordering invariant*, which says that all elements in *left* are less than or equal to all elements in *right*, and the *size invariant*, which says that size(*left*) = size(*right*) if there are an even number of elements, and otherwise the sizes of *left* and *right* differ by only one. These guarantees mean that if there are an even number of elements in the data structure, the median is the average of *max*(*left*) and *min*(*right*), and otherwise the median is the *max* or *min* value of whichever heap is larger.

Operations:

insert(*x*): First, determine which heap should contain x to maintain the ordering invariant. If $x < max(left)$, then add x to *left*; otherwise add it to *right*. This may break the size invariant. The size invariant can only be violated if before adding the value, there were an odd number of entries in the data structure (since if previously there were an even number of values, the heaps would have to have the same size). Therefore, if after inserting the value there are an even number of elements, and if additionally and one heap has exactly two more elements than the other, dequeue from that heap and enqueue the appropriate value into the other heap. This operation preserves the ordering invariant, since the value removed is either the biggest value from *left* or the smallest value from *right*. This operation requires only O(1) heap inserts or deletes, so it runs in time O(log n).

find-median(): If there are an odd number of elements in the data structure, one of the two heaps must have one more element than the other. If it's the maximum element of *left*, then that element is greater than half the elements (namely, the other elements of *left*) and smaller than half the elements (the elements in *right*), so it's the median. Therefore, return *max*(*left*). By similar reasoning, if the odd element is in *right*, then *min*(*right*) is the median, so we can return it.

Otherwise, there are an even number of elements in the data structure. This means that the median element is the average of the two elements closest to the median point. Using reasoning analogous to the odd case, we know that *min*(*right*) and *max*(*left*) are those two elements, so we can return the average of *min*(*right*) and *max*(*left*).

Both of these operations only require calling *min* or *max* in *right* and *left*, and therefore run in time O(1).

Answering Theory Questions

Some of the questions on the problem set will be theory questions that ask you to prove various mathematical results that are relevant for the analysis of data structures. For questions like these, we expect that you'll write a formal mathematical proof of the result. However, for ease of grading, we'd like you to structure your answers as follows:

- **Give a high-level description of your analysis or proof.** If you're writing a proof, you might give a two or three sentence description of the main insight behind the proof and how you'll turn that insight into a proof. If you're asked to perform a calculation of some sort, you can explain how you went about performing that calculation.
- **Write the proof or calculation.** This is where you'll either write a formal mathematical proof or work through the steps in a calculation in detail.

As an example, consider the following problem:

> Consider a binary heap B with n elements, where the elements of B are drawn from a totally-ordered set. Give the best lower bound you can on the runtime of any comparison-based algorithm for constructing a binary search tree from the elements of B.

Here is one possible solution:

Proof Idea: The lower bound is $\Omega(n \log n)$, and this is a tight bound. We'll prove this by first showing that there's an $O(n \log n)$-time, comparison-based algorithm for constructing a BST from the elements of an n-element heap. Then, we'll show that any $o(n \log n)$-time, comparison-based algorithm for doing the conversion would make it possible to sort n elements in time $o(n \log n)$ using only comparisons, which we know is impossible.

Proof: First, we'll show that there is an $O(n \log n)$-time, comparison-based algorithm for constructing a BST out of the elements of B. Specifically, just iterate across the n elements of B and insert each into a balanced binary search tree. This does $O(n)$ insertions into a balanced binary search tree, which will take time $O(n \log n)$. This algorithm is also comparison-based because binary search tree insertion is comparison-based.

Next, we'll show that no $o(n \log n)$-time, comparison-based algorithm for constructing a BST from a binary heap exists. Assume for the sake of contradiction that such an algorithm exists. Then consider the following algorithm on an array of length n:

- Construct a binary heap B from the array elements in time $O(n)$.
- Create a binary search tree T from B in time $o(n \log n)$.
- Do an inorder traversal of T and output the elements in the order visited in time $O(n)$.

Note that the runtime of this algorithm is $o(n \log n)$, and each step is comparison-based. However, this algorithm will sort the elements of the array, because doing an inorder traversal over a BST will list off the elements of that BST in sorted order. This is impossible, since there is no $o(n \log n)$-time, comparison-based sorting algorithm. Therefore, no $o(n \log n)$-time, comparison-based algorithm exists for converting a binary heap into a binary search tree. ■

Answering Implementation Questions

Many of the problem sets will ask you to code up various data structures, possibly to gain the experience doing so, or possibly to ask you to get performance numbers on that data structure. We will usually provide you with starter files in a particular programming language, and it's our expectation that you'll write your solution in that language.

In any implementation question, we'd like you to submit your code using our submitter script (details at the front of the handout) in addition to the rest of your problem set. Please don't submit your code in your GradeScope submission; it's pretty much impossible to test it that way. ☺

Some coding questions might also come with auxiliary questions about the code you wrote (for example, asking you to justify your design decisions), and you should submit those with the rest of your short answer questions rather than in code.

Working in Pairs

You are welcome to work on the problem sets either individually or in pairs. If you work in a pair, you are required to submit a single joint assignment with your partner. We'll then grade that single assignment and assign you and your partner the same overall score. It is a violation of the Stanford Honor Code to work in a pair and submit assignments individually, since the work you submit would not be your own. See the handout on the Honor Code for more details.