# COMP_SCI 214 (Winter 2024)
## Exam 1: Solutions, Explanations, and Grading

The purpose of this document is to elaborate on the answers to the first exam, explain how things were graded and why, provide some rationale for the whole process, and set up some guidelines regarding exam-related communication. Please read this document *in full* before contacting me regarding the exam.

# Prologue: Why exams?

We use two main assessment mechanisms in this class: programming assignments and exams. Each one assesses different things, and is incomplete on its own. Hence our using both.

Programming assignments are effective at evaluating the mechanical, applied portions of the material, in a way that also reinforces your learning. They allow for extended periods of time to think about problems, seek help, etc. This is an effective assessment mechanism, which is why it accounts for the major part of your final grade.

Exams, on the other hand, are effective at evaluating the conceptual parts of the material, and at exploring implications and combinations of material that are not useful to test in a programming assignment or cannot be tested. The fact that the questions are not known ahead of time serves as a forcing function to get students to engage with the entire material (rather than strictly the parts relevant for a given assignment). The time-limited setting gets students to "think on their feet" and develop quick intuitions about the material.

Neither of these mechanisms are perfect, but both are necessary to see a complete picture of your learning. And both are representative of skills you will need to be an effective software engineer and computer scientist. Programming assignments, of course, are meant to resemble the actual programming part of a developer's responsibilities. Exams, however, share a lot of similarities with activities like technical discussions or exploratory design, where being able to quickly explore implications (without needing to completely work out the details) and explain ideas to colleagues are key skills.

# Communication

After the exam, you may be tempted to get in touch for a variety of other reasons. Here are some guidelines regarding such communication.

- Any exam-related communication should go to me (Prof. Sruti) and be via a private post to me on Piazza.
- To confirm that you have read and understood this document in full, please begin your message with the text "I have read and understood the post-exam companion." just so we're all starting on the same page.

# Answers and explanations

Here are the answers to the questions on the exam, explanations for why these answers are the correct ones, and why others are not. For reference, a pdf of the exam has been uploaded to Canvas as well.

## Problem 1: Short-Answer Questions

### 1.i

As we learned in class, we only ever care about tight bounds of complexities. This problem tested your understanding of tightening bounds and combining complexities to make decisions about an algorithm to use. Linear search runs in O(n) time, the fastest sort we learned runs in O(n log n) time, and binary search runs in O(log n) time. Given these, here are the correct answers for this problem.

- $k = 42$: **Linear search**. Linear search over 42 searches would run in O(42n) = O(n). If we sorted the array first, we would do O(n log n) work and then O(42 log n) = O(log n) to binary search k times. Adding these together, total work for this 2nd option is O(n log n) after keeping the dominant term, which is slower than O(n) with just linear search.
- $k = \log n$: **Both approaches**. Linear search over k = log n searches, would be O(n log n) total time. If we first sorted and then did binary searches, it would be O($nlogn$ + $(logn)^2$) = O(n log n) after keeping only the dominant term. Therefore, both approaches have the same tight bound of complexity and are both fine for this case.
- $k = n$: **First sort and then binary search**. Linear search over k = n elements would be O($n^2$) total work. With the second option, the total complexity is O(n log n + n log n) = O(n log n). This is faster than the first option.
- $k = n \log n$: **First sort and then binary search**. Linear search over k = $n \log n$ elements would be O($n^2$ log n). With the second option, the total complexity is O(n log n + n $(logn)^2$) = O($n(logn)^2$) after reducing to its tightest bound. This is faster than the first option as $(logn)^2$ is faster than $n^2$.

An answer that got all 4 rows correct was eligible for full credit. If at least two of the rows were correct, the answer was eligible for 1 out of 2 points. Anything less than two correct rows was not eligible for credit.

### 1.ii

The question said to "Explain your answer by including specific details about the ADT, data structures, or complexities as applicable," therefore, we were looking for concrete details about operations and complexities. There were two possible answers to this question:

1. Unsorted array of (key, value) pairs: Assuming the arrays have extra space for most insertions, unsorted array has O(1) insertion since we don't care about checking duplicates and we can just add it at the next empty spot, whereas a sorted array has O(n) insertion. Sorted array has better lookup (O(log n)) vs O(n) but we'll do insertions a lot more than lookups, so **unsorted array** gives us better performance on that operation.
2. Sorted array of (key, value) pairs sorted by key: Assuming arrays only have as many slots as elements, insertion of new keys into both would be O(n) since a new array with more space would have to be created and elements copied over/shifted each time. However, sorted array at least gives better lookup performance in O(log n), so a **sorted array** would be the best data structure here.

Answers that matched either of the above with the assumptions stated and correct complexities, were eligible for credit. All other answers were not eligible for credit.

## Problem 2: Analyzing an algorithm

This algorithm is a sorting algorithm called "bubble sort".

### 2.i

The structure of the implementation may appear reminiscent of selection sort, which can help us analyze its complexity. The outer loop runs n-1 times. The inner loop runs n-1 times on the first iteration of of the outer loop, n-2 times on the second iteration of the outer loop, n-3 times on the third iteration, and so on. The last iteration of the outer loop will have its inner loop running 1 time. If we add these up, we get #steps $= 1 + 2 + 3 + 4 + 5$ ... + n-1 for a whole run of the algorithm. Since we know there is already a term that is linear (n-1), adding another term $n$ to this should not increase the existing complexity. If we add $n$, we get a more familiar series: $1 + 2 +$ ... n which we also arrived at in class for selection sort. Therefore, our complexity of this algorithm is the same as selection sort which is $O(n^2)$.

### 2.ii

The only way there can be a better case than $O(n^2)$ is if the loop(s) can be exited early in some way. The inner loop on line 5 does not have any conditions to end early, but the outer loop has a condition to break early on lines 12-13. This break can only happen after a whole pass of the inner loop has been completed. Therefore, we cannot escape iterating through the array at least once even if we were to break early on the first outer loop iteration. Therefore, the best case complexity is $O(n)$. The answer was eligible for 1 out of 2 points if this complexity was correct.

The best case only occurs if there is no element in the array that is greater than the element that comes after it. Therefore, any sorted array of valid values was eligible for the other 1 point.

### 2.iii

Any sorting algorithm we learned in class was eligible for credit: selection sort, merge sort, or quick sort.

## Problem 3: Stacks and Duplicates

This problem tested your understanding of using interface functions correctly (without using internal data representations) and understanding their abstract functionality enough to solve a problem that uses them.

### 3.i

The correct completion of the starter implementation is below:

```
def stack_contains(s: STACK!, x) -> bool?:
    if s.empty?():
        return False
    let temp = s.pop()
    if temp == x:
        s.push(temp) # 1st blank
        return True
    let is_contained = stack_contains(s, x) # 2nd and 3rd blank
    s.push(temp) # 4th blank
    return is_contained
```

There are two components to the answer we expected: restoring the stack correctly, making the recursive call correctly. 1 point was allotted if the answer restored the stack correctly in the 1st and 4th blank lines. The other 1 point was allotted if the recursive call correctly passed in `s` and `x`.

### 3.ii

The correct completion of the starter implementation is below:

```
def stack_has_dupes(s: STACK!) -> bool?:
    let temp_s = ListStack()
    let dupes_exist = False
    while not s.empty?(): # 1st blank
        let temp = s.pop() # 2nd blank
        if stack_contains(s, temp): # 3rd blank
        # Alternative answer: stack_contains(temp_s, temp)
            dupes_exist = True
        temp_s.push(temp)
    while not temp_s.empty?():
        s.push(temp_s.pop()) # 4th blank
    return dupes_exist
```

For this problem, correct functionality relies on all 4 lines being correct and working together. If an answer got all 4 of the blanks correct, it was eligible for full credit. If the answer was almost correct by correctly implementing 3 out of the 4 lines, it was eligible for 1 out of 2 points. An answer that had fewer correct lines did not demonstrate a sufficient understanding and was not eligible for credit.

### 3.iii

`stack_contains` is a recursive function. Recall from class and your worksheet, when computing the complexity of a recursive function, the number of steps is proportional to the number of times the recursive call is made. When the function is first called, it counts as 1 step. Inside the first call, `stack_contains` is called again with the modified stack after shrinking the stack size by 1 element (after popping). This is 1 more step. If the stack keeps shrinking by 1 element before each call (1 additional step each time) and given that the recursive calls will stop once there is nothing left in the stack, we will make about as many recursive calls as there were elements in the stack to start with (order of $n$). Therefore, the complexity is $O(n)$.

Correct answers received 1 point, otherwise they were not eligible for credit.

### 3.iv

The steps in `stack_has_dupes` before the first while loop are all O(1) operations, so they won't factor into our final complexity ultimately when reducing to tight bounds. The first loop has a condition to run until there is nothing left in the stack, so we first need to compute how many times this loop should run. Since `s` is shrinking by 1 element (because an element is popped in each iteration), the loop will run about as many times as there were elements in the stack to start with (order of $n$). Now, each iteration of the loop does some work. The pops and push statements and the `dupes_exist = True` statement are all constant-time work which we can ignore but we know `stack_contains` is non-constant time work. Therefore, the final complexity is O(n * cost of `stack_contains`).

By the end of the above loop, `temp_s` has `n` elements and so the second loop has complexity O(n) since it runs as many times as there were elements in `temp_s`. O(n) will either be the same as or lower than O(n * cost of `stack_contains`), therefore, we ignore this complexity and focus on the above complexity.

The correct answer, which was eligible for full credit is $O(n^2)$ because `stack_contains` is a linear operation. If your answer to 3.iii was incorrect, we still awarded full credit if your answer to this question was O(n * your complexity for 3.iii). All other answers were not eligible for credit.

## Problem 4: Least Recently Used Cache

This problem was the most challenging problem. It was meant to introduce you to a real-world concrete use case of data structures we've learned about and to test how your understanding of simple data structures

you've seen in class and your assignments can apply to complex combinations of data structures.

### 4.i

This question asks for a diagram of class fields and data structures. A valid diagram correctly depicting all of these is given below. We expected everyone to have read the question carefully which says that the diagram needs to represent class fields (`_head`, `_length`, `_cap`, etc.), however, we were more particular about the concrete data structures in the diagram than the class fields. The diagram we expected is provided at the end of this document, although our grading focused on the hash table and DLL portion of the diagram.

To be eligible for the 2 out of 2 points, the diagram needed to have: a 4-element array depicting the hash table, 3 linked list nodes that are connected to each other with both `next` and `prev` arrows going in both directions, and arrows from the value field of 3 of the hash table slots to point to a unique DLL node.

A diagram was eligible for 1 out of 2 points if it had the above boxes, but was missing any arrows. Any other answers were not eligible for credit.

### 4.ii

An answer was eligible for 1 of 3 points if `cache_data` was fetched from the hash table using the right key with the statement `self._ht.get(data)`. This demonstrated an understanding of correctly working with interface implementations and functions. If an answer tried to access the hash table using square brackets (e.g., `self._ht[data]`), that did not demonstrate the above understanding, and so it was not eligible for credit. We were not particular about small syntax bugs, e.g., `_ht.get(data)`, or `get(data)`, as long as they used the correct interface functions and passed the right key.

The answer was eligible for the remaining 2 points, if it correctly set `cache_data` to be the most recently used element, while satisfying invariants by the end. The starter code I included in the exam was supposed to include the following piece of code: under `cache_data.prev.next = cache_data.next`:

```
    if _cons?(cache_data.next):
        cache_data.next.prev = cache_data.prev
    else:
        self._tail = cache_data.prev
```

However, due to a mistake on my part, I forgot to include that block in the exam. Assuming the above was part of the implementation, there would only be 4 remaining lines of code, which is the reason there were 4 empty lines for you to fill in. However, due to my mistake, we considered a fully correct answer to be the following five lines.

```
# without worrying about if cache_data was the tail
cache_data.next.prev = cache_data.prev
cache_data.next = self._head
cache_data.prev = None
self._head.prev = cache_data
self._head = cache_data
```

The remaining 2 points for this problem were allotted based on these criteria:

- An answer that matched the above exactly or was functionally equivalent to all 5 lines was eligible for the remaining 2 points.
- Since some of you provided only four lines because of what was given to you, an answer that matched 4 of the above lines exactly, or was functionality equivalent to 4 of the above lines, was also eligible for the remaining 2 points.
- If an answer matched 3 of the above lines or was functionality equivalent to 3 of the lines, it was eligible for 1 of the 2 remaining points.

- Any other answer was not eligible for the remaining two points.

#### 4.iii

An answer was eligible for 1 of 2 points if the hash table was updated correctly by removing the entry with the key as `data` via the statement: `self._ht.del(self._tail.data)`. We were particular about the interface function `del` being used and the argument being `self._tail.data` for an answer to be eligible for credit. Passing in `self._tail` to the `del` function passes in the value not the key, and did not demonstrate an understanding of what the key of the dictionary is, or how to use the dictionary interface with the right arguments.

The other 1 point was allotted if their answer removed the least recently used element from the linked list, while satisfying invariants by the end. The correct implementation is given below:

```
if self._length == 1:
    self._head = None
    self._tail = None
else:
    self._tail.prev.next = self._tail.next # or 'None'
    self._tail = self._tail.prev
```

If the answer provided matched the above exactly or was functionally equivalent to the above, it was eligible for the additional 1 point, otherwise not.

# I didn't do as well as I had hoped. Now what?

When you see your graded exam, some of you may be disappointed. That's a natural reaction. But please keep the following in mind:

- Your score on this exam is *NOT* a measure of your worth as a human being. It's merely an imperfect measure of one aspect of your understanding of the material in the first half of one particular class. Nothing more. Don't lose track of the big picture.
- In most cases, this exam does not affect your final grade by a large amount. It's absolutely possible to recover from a disappointing performance.
- Remember: this is the big leagues here. *Everyone* here is super smart. Yes, that means you too. So even if your grade on this one exam is disappointing, *you're still pretty great.* We throw a lot of difficult things at you because that's how one learns the most and *because we know you can do it.* We believe in you, and we're here to help you get there.

Ultimately, how you feel about to your grade will not change it; barring genuine mistakes in grading (see below), your grade will not change. So my advice to you is: reflect on what went wrong, learn from it, move on, and do better next time.

To do better next time, we encourage you to take advantage of the resources we put at your disposal to help you succeed:

- Actively participate in lecture, including by asking questions (via Piazza, if you prefer to stay anonymous) and giving a genuine effort in in-class exercises.
- Watch the recordings to review concepts you may not have fully understood.
- Ask questions on Piazza, and try answering your colleagues' questions (the best way to learn is to teach!)
- Go to office hours; those are not just for debugging, conceptual questions are welcome too!
- Get in touch with me via Piazza to discuss your approach to the class.

# Seeing your graded exam

When we release the grades, we will also be giving you access to the graded scan of your exam on the GradeScope platform. There, you will be able to see your answers, and how they were graded.

The five possible outcomes for the exam—positive, neutral, single-negative, double-negative, or triple-negative modifier—will be denoted on Canvas as a score of 1, 0, -1, -2, and -3 on the exam assignment, respectively.

If you have questions about the exam, see Section "Communication" of this document.

# Did we make a mistake?

We are very careful when grading exams, with multiple graders looking at each exam, and lots of double-checking along the way. While this is unlikely, it is possible we may have made a mistake when grading.

Things that are grading mistakes:

- Clerical errors
- Not giving credit for an answer that is *the same* or *exactly equivalent* as one that we explicitly said we accepted

Things that are *not* grading mistakes:

- Disagreeing with a decision we made
- Being unhappy with your grade

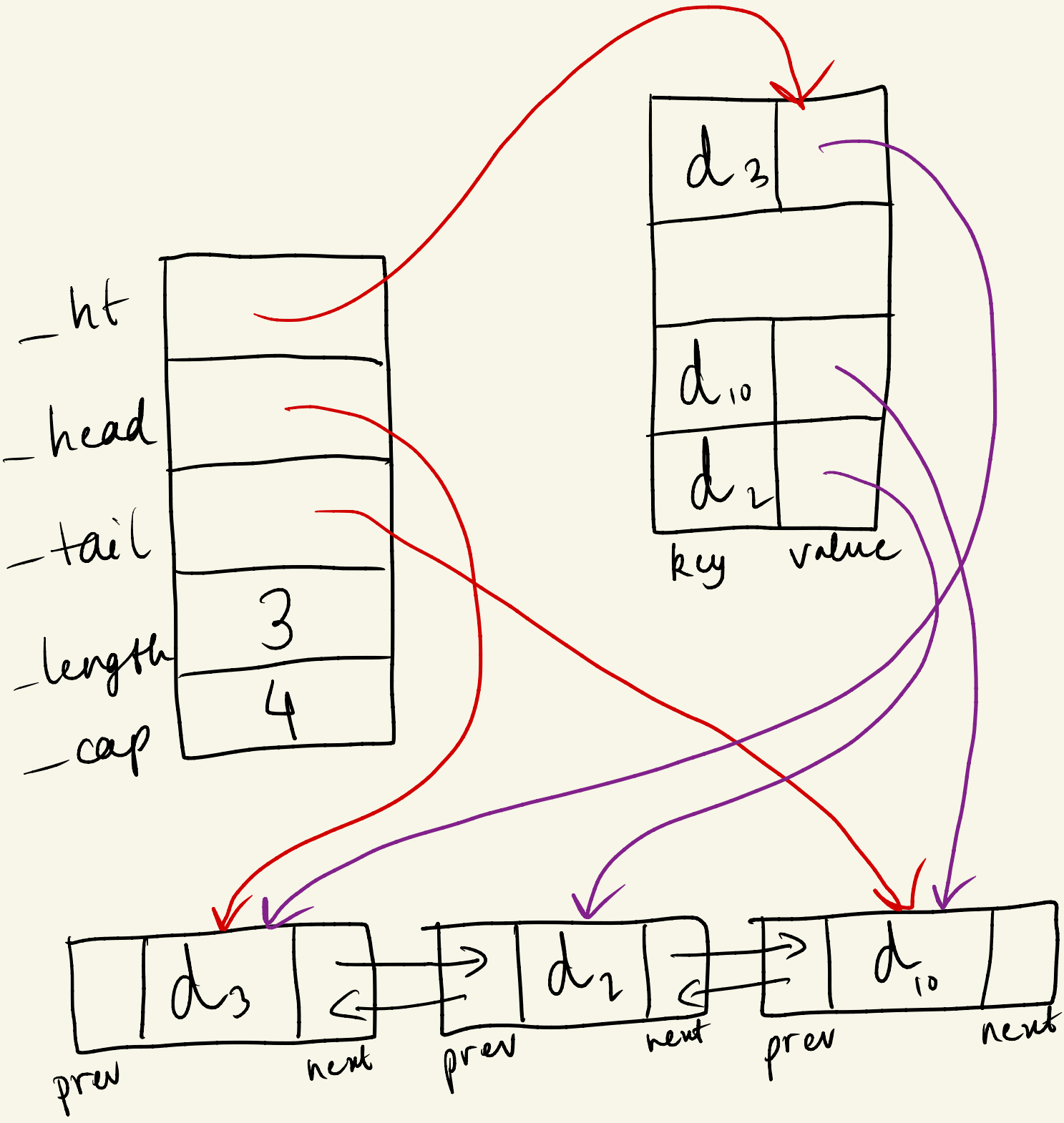
If you're not sure whether something is a grading mistake or not, I encourage you to discuss this with a member of our course staff in office hours.

If we *did* make a mistake when grading your exam, we will fix it. Please let us know by filing a grading correction request here: https://forms.gle/SUvEnL3jVoLmfMEz5

Bear in mind:

- Grading correction requests must be targeted: we do not do blanket regrades of an exam.
- Grading correction requests which are not about genuine grading mistakes will be ignored.
- Because scores on the exam get turned into modifiers, not all grade deltas are significant. For example, a score of 13 and a score of 14 both correspond to a neutral modifier. Submitting a grading correction request to go from 13 to 14 is not useful.
- Decisions on grading corrections (or lack thereof) are final.

All correction requests must be in by *Friday, Feb 9th, 5pm.* Late requests will not be considered.


_ht
_head
_tail
3
_length
4
_cap
$d_3$
$d_{10}$
$d_2$
key
value
$d_3$
$d_2$
$d_{10}$
prev
next
prev
next
prev
next